MANUAL or AUTOMATED? A UI TESTING COMPARISON

(MANUEL veya OTOMATİK? BİR KULLANICI ARAYÜZÜ TEST KARŞILAŞTIRMASI)

by

Zeynep Ceren BAYRAM, B.S.

Thesis

Submitted in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

in the

GRADUATE SCHOOL OF SCIENCE AND ENGINEERING

of

GALATASARAY UNIVERSITY

Supervisor: Prof. Dr. Gulfem ISIKLAR ALPTEKIN

July 2025

ACKNOWLEDGEMENTS

I would like to sincerely thank Prof. Dr. Gulfem ISIKLAR ALPTEKIN for this guidance and support throughout my thesis journey.

I would like to thank Eray YAGIZ and Alp ERKAN, who have always shared their knowledge and support generously, as well as my family and friends Ayse, Dora, Ozlem, Burcu, Oguzhan and Mucteba who have always been by my side.

I dedicate this work to my dear brother, Mucteba BAYRAM, who has taken on the role of a father to me.

July 2025
Zeynep Ceren BAYRAM

TABLE OF CONTENTS

LIST OF SYMBOLS

PoC	**:** Proof of Concept
AI	**:** Artificial Intelligence
CI	**:** Continuous Integration and Continuous Delivery
CD	**:** Continuous Delivery
UAT	**:** User Acceptance Testing
GUI	**:** Graphical User Interface
UX	**:** User Experience
R&D	**:** Research and Development

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

In this paper, a comparison between manual and automation testing methods is conducted within the scope of user interface (UI) testing. To achieve a comprehensive evaluation, multiple metrics have been utilized to analyze each method from various perspectives. Initially, the user interfaces were identified, and corresponding test scenarios were developed. These scenarios were then executed manually and through automated scripts. To ensure reliability and generalizability of the results, the login functionalities of three different web applications were chosen for analysis. Numerical analyses based on selected metrics revealed critical insights regarding the relative advantages and disadvantages of each testing method. Ultimately, the findings of this study offer valuable guidance for companies and projects in deciding which testing approach manual or automated would be most beneficial according to their specific needs.

Keywords: Software Testing, Manual Testing, Test Automation, Regression Test

ÖZET

Bu çalışmada, kullanıcı arayüzü (UI) testi kapsamı içinde manuel ve otomasyon test yöntemleri arasındaki karşılaştırma yapılmıştır. Kapsamlı bir değerlendirme elde edebilmek için her bir yöntem, farklı perspektiflerden analiz edilmek üzere birden fazla metrik kullanılarak incelenmiştir. İlk olarak, kullanıcı arayüzleri belirlenmiş ve buna uygun test senaryoları geliştirilmiştir. Bu senaryolar daha sonra hem manuel olarak hem de otomatik betikler aracılığıyla uygulanmıştır. Sonuçların güvenilirliğini ve genellenebilirliğini sağlamak amacıyla, analiz için üç farklı web uygulamasının giriş işlevleri seçilmiştir. Seçilen metriklere dayalı sayısal analizler, her bir test yönteminin karşılaştırmalı avantajları ve dezavantajları hakkında kritik bilgiler sunmuştur. Sonuç olarak, bu çalışmanın bulguları, şirketlere ve projelere, spesifik ihtiyaçlarına göre manuel ve otomasyon test yöntemlerinden hangi yaklaşımın daha faydalı olacağına karar vermede değerli bir rehberlik sunmaktadır.

Anahtar Kelimeler: Yazılım Testi, Manuel Test, Test Otomasyonu, Regresyon Testi

1. INTRODUCTION

Today, software systems form the essential components of numerous industries, including finance, defense, and marketing. As reliance on software systems continues to grow, ensuring their quality has become increasingly critical. The testing phase of a software project plays a key role in verifying that user requirements are met and in evaluating overall system quality. Indeed, software quality can only be accurately assessed through rigorous software testing (Sneha and Malle 2017).

Software testing is primarily conducted using two distinct methods: manual testing and automation testing. Manual testing is straightforward to implement and does not necessitate extensive technical knowledge. Conversely, automation testing requires proficiency in coding and initial integration efforts. Although manual testing is simple to begin with, repeated execution, particularly during regression testing, can become costly and time-consuming. Despite the initial setup complexities, automation testing often provides substantial long-term benefits, such as efficiency gains and reduced human error. Nevertheless, many organizations hesitate to adopt automation testing, typically preferring traditional manual approaches.

Regression testing is conducted to verify that software updates, such as new feature implementations, bug fixes, or modifications, do not negatively affect existing functionalities. The primary purpose of regression testing is to ensure that previously functioning parts of the software continue to perform correctly after changes are introduced. Identifying, analyzing, and resolving regression related issues represent some of the most time-consuming tasks in software development, especially within large scale, production level systems (Badihi et al. 2023).
Automated regression testing significantly enhances test efficiency, accelerates feature deployment, and minimizes the potential for human errors (da Silva and Souza

Santos, 2023). By automating regression testing, teams can conduct frequent test cycles, promptly identify defects and maintain continuous integration and continuous delivery (CI/CD) practices effectively (Akin, Sentürk, and Garousi 2018). Thus, automated regression testing is essential for improving software quality, stability, security, and overall user experience, playing a crucial role in streamlining and accelerating the software development lifecycle.

In this study, we analyze and compare the advantages and disadvantages of manual and automation testing methods based on various metrics. The metrics considered include Speed (Test Execution), Time, Team and Skill Requirements, Reporting Quality, Automated Start Capability, Suitability for Different Platforms, Coverage (Intuition), Reliability (Human Error Factor/Digitalization Issue), and Cost. Additionally, the research investigates which testing method might prove more beneficial depending on the nature and requirements of different software projects. The issue of selecting the testing method based on the project is addressed as a gap in literature. While continuing with manual testing makes sense for short-term projects, it is suggested that transitioning to automation testing would be beneficial for long-term and continuously developed projects. This approach emphasizes the importance of choosing testing methods based on the project's requirements and continuity. The whole flow of the evaluation framework is given in Figure 1.1.
below.

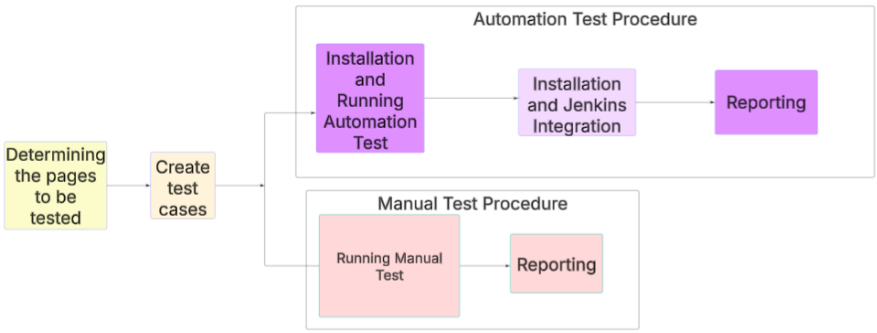

Figure 1.1: Test Process

2 TEST PHASE ON SOFTWARE LIFECYCLE

The Software Development Life Cycle (SDLC) is an approach used to make the software development process more efficient, systematic, and controllable. SDLC defines each stage in the software development process and typically includes phases such as planning, design, development, testing, deployment, and maintenance, all followed in a specific sequence. The methodologies of the software life cycle first emerged in the 1960s with the aim of significantly improving the effectiveness of software development processes. Early software projects were large, complex, and time-consuming, which made managing the development process extremely difficult. In the beginning, software development was often unplanned and disorganized. During the 1960s, computers and software applications began to advance rapidly. However, software development processes were still generally unstructured and lacked standardization. Early software systems were written without any predefined plans or designs, relying entirely on individual developers' efforts. As a result, many of these early software applications were error prone, difficult to maintain, and lacked scalability. To make the development process more structured, efforts in the 1970s introduced software practices governed by specific rules and order. The software development process was divided into phases. Dividing the development process into different phases brought many important benefits and significantly increased the success rate of software projects. These benefits include making project management more efficient, improving quality, reducing risk, and better aligning the outcomes with customer requirements.

2.1. Benefits of Dividing the Software Life Cycle into Phases

i. Better Planning and Management

Dividing the process into specific phases allows the objectives and deliverables of each stage to be clearly defined. This facilitates better planning to ensure that the project progresses in alignment with the overall timeline.

Time Management: Since dedicated timeframes are allocated to each phase, the likelihood of completing the project on schedule increases.

Resource Management: It becomes possible to plan when and where resources are needed, allowing for their efficient use.

Budget Control: Clearly defined phases make it easier to track project costs and help prevent budget overruns.

ii. Risk Management

Dividing the software development process into phases makes it easier to identify potential problems and risks in the early stages. Since a specific part of the software is completed in each phase, possible errors can be detected early on.

Early Error Detection: Through testing and evaluation phases, errors are identified and corrected at an early stage, which helps reduce overall project risks.
Project Monitoring: After each phase is completed, progress and risks are reviewed, allowing for immediate risk management.

Change Management: When feedback is received at the end of each phase, necessary changes can be made quickly, minimizing deviations from project goals.

iii. Enhanced Quality Control

Each phase addresses a specific aspect of the software (requirements, design, coding, testing, etc.). This structure creates opportunities to monitor and improve quality at every stage.

Advanced Testing Processes: Specific tests are conducted at each stage to continuously monitor the quality of the software.

Compliance with Standards: Adherence to the standards defined for each phase is ensured, which leads to higher software quality.

Fewer Errors: Reviewing and testing the software at every stage increases the likelihood of delivering an error free product to the end user.

iv. Compliance with Customer and User Requirements

Dividing the software development process into specific phases allows for the review of customer needs and the collection of feedback at each stage. In this way, customer expectations can be better met as the project progresses.

Adaptation to Content Changes: By collecting customer feedback at each stage, the software can be more closely aligned with the needs of the target users.

Flexibility: Changes in requirements can be accommodated more quickly, which contributes to increased customer satisfaction.

Customer Communication: Regular meetings and deliveries throughout the process ensure ongoing communication with the customer.

v. Improved Traceability and Monitoring

When the software development process is divided into phases, the outcomes of each stage and the actions taken at each step can be easily tracked. This makes project management more transparent and easier to monitor.

Monitoring and Reporting: Project managers receive reports and updates at the end of each phase, making it easier to track the project's progress.

Documentation: Documentation is prepared at the end of each stage, ensuring that a comprehensive record is maintained for every aspect of the software.

Performance Evaluation: Evaluations conducted at the end of each phase help measure the progress and development of the software.

vi. Easier Maintenance and Improvement

Since every aspect of the software is addressed in detail from the beginning to the end of the process, maintenance and improvement of the software become easier.

Ease of Maintenance: Since different aspects of the software are detailed in each phase, improvements and maintenance operations can be carried out more easily.

Sustainability: As each stage of the software is properly managed, the long-term maintenance of the software becomes more sustainable.

vii. Intra-Team Coordination and Communication

When the software development process is divided into phases, specific tasks and responsibilities are assigned at each stage. This helps team members understand their responsibilities more clearly and enhances coordination within the team.

Task Distribution: Each team member is responsible for a specific phase, which helps prevent confusion and conflicts.

Ease of Communication: Since what needs to be done at each stage is clearly defined, communication within the team becomes easier and more efficient.

Team Motivation: Clearly defined goals and delivery deadlines help keep team members more motivated.

viii. Reusability and Modularity

Dividing the software development process into phases enables the software to become modular. Each module or component can be reused when needed and may also prove useful in other projects.

Reusable Code: Each phase and module can be developed independently, allowing the same module or component to be reused in future projects.

Maintenance and Improvement: Thanks to the modular structure, only specific parts of the software can be updated, enabling maintenance with lower risk.

Dividing the software development process into distinct phases allows projects to be carried out more efficiently, systematically, in a more controlled manner, and with higher quality. This structured approach makes the software development process more transparent, reduces risks, enhances quality, and ensures better alignment with customer needs. The presence of clear goals and requirements within each phase significantly increases the likelihood of success in the software development process.

2.2. Phases of the Life Cycle

i. Planning and Requirements Analysis

The purpose of the software and user requirements are identified. A project plan is created, including timeline, resources, and budget.

ii. System Design

The overall structure and architecture of the software are designed. Detailed system design, data structures, and user interfaces are planned.

iii. Coding

Software developers write code in accordance with the design. Software modules and components are integrated.

iv. Test

Software testing is an investigative activity conducted to identify and eliminate defects. (Mori, 2020). Functional, performance, and security tests are carried out.

v. Deployment

The software is deployed for use by the end user. Installation and configurations are completed, and user training is provided.

vi. Maintenance and Support

After the software is released, bug fixes and updates are carried out. User support is provided, and new features are added. These phases cover the entire process from the beginning of software development to its delivery to the end user, and the subsequent maintenance phase. Throughout the software development life cycle, testing is critically needed to ensure the quality of both the development process and the final product

(Mustafa, Al-Qutaish, and Muhairat 2009). In this study, the testing phase of the software life cycle was addressed, and comparisons were made between different methods used during this phase. These comparisons were conducted across various metrics such as speed, time, cost.

3 TEST PROCESS

3.1. Determining the Pages to Be Tested

A total of 33 test cases were developed to comprehensively evaluate the specified metrics. These test cases specifically targeted the user login functionalities of three widely used educational websites. Preparing these test scenarios required approximately two days of effort. The selected websites offer various technical training courses and are known for their popularity and extensive user bases. Many websites incorporate 'human or robot' verification on their login pages to enhance security. If a login attempt is detected as originating from an automated tool rather than a human user, the corresponding IP address is typically blocked from accessing the system. This practice is especially prevalent among large corporations aiming to prevent unauthorized automated activities. However, the educational platforms selected for this study exhibit relatively low sensitivity to automated tools on their login pages. This characteristic makes these platforms ideal candidates for conducting comparative tests involving manual and automated approaches. Thus, the login pages of these three educational platforms were deliberately chosen for this testing study.

3.2 Create Test Cases

Login pages represent the initial interaction between users and software applications or websites, making their proper functioning crucial. The performance of the login page significantly influences user experience, security measures, and overall system functionality. To conduct the tests effectively, test cases were systematically documented using a structured Excel template containing essential components for thorough testing. The fields included in this template are as follows: (Yalamanchili and Kumari 2016).

Test Case ID: A unique identifier assigned to each test case, typically abbreviated from the tested class name, combined with a sequential number. For example, for user login tests, IDs might be structured as 1ULI, 2ULI, 3ULI, etc.

Test Case Name: A concise and descriptive title summarizing the purpose of the test case. For instance, a test case assessing the system's reaction to an incorrect password could be named "LogInWithWrong".

Test Case Steps: Clearly defined, sequential instructions outlining the exact steps required to perform each test.
Input: The specific data or values entered during the test execution, such as text strings or numeric entries.

Expected Result: The anticipated outcome based on established software requirements.

Actual Result: The actual output observed upon completing the test steps.

Result: An indicator signifying the test outcome. It is marked "Pass" when the actual result aligns with the expected result, or "Fail" if a discrepancy occurs.

Notes: Additional remarks or relevant information concerning the execution or outcome of the test case.

The test cases created were applicable for both manual and automated testing scenarios. Initially, tests were executed manually, after which automation scripts were developed to perform the same tests automatically. This structured approach ensured a consistent and thorough evaluation across both testing methodologies. A sample portion from the specified test cases is shown below.

	A	B	C	D	E	F	G	H
1	MODULE	TEST CASE ID	SCENARIO NAME	TEST STEPS	INPUT	EXPECTED RESULT	ACTUAL RESULT	NOTE
2		CA1	CheckPage	Open the codecademy Sign In Page	https://www.codecademy.com/login			
3				View the Giriş Yap title		"Log in to Codecademy" title should be displayed		
4				Open the codecademy Sign In Page				
5				Click e-posta attribute				
6				Type an invalid e mail	zeynep@gmail.com			
7		CA2	LoginInvalidMail	Click password attribute				
8				Type an valid password	Zeynepceren55.			
9				Click Giriş Yap button		Warning should be displayed like "Invalid email or password"		
10				Open the codecademy Sign In Page	https://www.codecademy.com/login			
11				Click e-posta attribute				
12				Type an valid e mail	zeynepcerenbayram@gmail.com			
13		CA3	LoginInvalidPassword	Click password attribute				
14				Type an invalid password	Zeynepceren			
15				Click Giriş Yap button		Warning should be displayed like "Invalid email or password"		
16				Open the codecademy Sign In Page	https://www.codecademy.com/login			
17				Click password attribute				
18		CA4	LoginEmptyMail	Type an valid password	Zeynepceren55.			
19				Click Giriş Yap button		Warning should be displayed like "*Required."		

Figure 3.1. Test Case Example

3.2.1 Creating Artificial Intelligence Test Cases

In Section 3.2, the test cases were created based on human judgment. To explore whether artificial intelligence could generate test cases similar to those written through human reasoning and observation, the task of writing test cases was also assigned to AI. The version of artificial intelligence used for this purpose was ChatGPT 4.0. Three web pages link each subject to testing were provided to the AI. The number of test cases generated by the AI for each of these links is shown in the table below.

Table 3.1: Comparing Test Case Counts of AI and Human

	EDX	KHAN	CODEACADEMY
ChatGPT 4.0	11	7	10
Human	11	11	11

Advantages: Artificial intelligence was able to write the essential functional test cases required for a login page. Artificial intelligence was able to generate negative test scenarios. For example, a test case designed to evaluate how the system responds when an invalid email address is entered was written by the AI. It did not only consider the fundamental "happy path" functions but also accounted for alternative and error conditions.

Disadvantages: Artificial intelligence was not able to generate test cases as comprehensively as those created through human judgment. For example, it failed to produce a test case for the scenario where the password field is left empty. Artificial

intelligence was unable to generate valid inputs. This requires a fully human driven process such as creating an actual user and executing test cases using that user.

3.3 Running Manual Test

Following manual testing, the execution time for each test case was recorded individually in seconds, allowing the total duration required to complete all tests to be accurately calculated. Each of the selected web pages was tested separately across three different browsers, and the execution times were meticulously measured for each scenario. This comprehensive approach provided detailed insights into the overall time requirements associated with manual test execution.

Table 3.2: The time spent running manual tests in each search engine (man/seconds).

	EDX	KHAN	CODEACADEMY
MOZILLA	265	275	291
FIREFOX	263	281	304
WEBKIT	293	292	319

The computation of the annual manpower of the project is done as follows:
1 iteration: 2,583 seconds -> 43 man/minutes

Weekly deployment=Average of 6 test environment deployments + 1 production deployment=7 deployments x 43 minutes = 301 man/minutes

Monthly: 1,204 man/minutes

Yearly: 14,448 man/minutes -> 240 hours -> (Dividing by 8 gives the number of workdays, assuming 1 workday = 8 hours, 240 / 8 = 30 workdays -> equivalent to one and a half months of manpower)

Thus, the annual manpower cost for a project with only 33 test cases amounts to 1.5 months of workdays.

3.3.1 Manual Test Execution by Artificial Intelligence

In this section, the ability of artificial intelligence to manually execute test cases was evaluated. The AI was instructed to manually execute the test cases it had created. It was determined that the AI is not capable of performing manual test execution. The reasons for this are outlined below. Manual test execution requires real user interaction. AI cannot click buttons or fill in text fields like a human can. In other words, it cannot interact with elements in a web interface.

Manual testing requires physical observation in a browser. AI cannot evaluate the web page's reactions after actions as true/false or pass/fail. This is because it cannot determine what the correct response should be and thus cannot make a proper judgment. AI cannot provide valid inputs like a real human. For instance, it does not possess valid email or password information and therefore cannot supply these values to the system or interpret whether the result is a pass or a failure. AI lacks capabilities such as launching a browser or navigating to a specific URL.

3.4 Installation and Running Automation Test

After completing manual test executions, preparations began for automation testing. The first step in this process involved selecting an appropriate tool to develop automation scripts. After extensive research, Playwright was chosen for writing automated software test cases. Playwright, a relatively new open-source test automation framework developed by Microsoft and released in 2019, specializes in modern web application testing, particularly excelling at handling dynamic content and JavaScript-based applications. It efficiently simulates user interactions on web pages, facilitates automated test creation, and supports various validations. One of Playwright's notable advantages is its simplicity in setup compared to other automation tools. The necessary dependencies can be easily installed by running a single command (npx playwright install). Moreover, Playwright is entirely free, making it cost-effective and accessible for organizations aiming to transition smoothly from manual to automated testing. Since companies often resist shifting to automation due to concerns about complexity and

cost, Playwright's user-friendly nature and easy adoption significantly help reduce this resistance.

Key features of Playwright:
Multi-Browser Support: Playwright supports testing on Chrome, Firefox, and WebKit (the engine for Safari), making it advantageous for ensuring consistency across different browsers.

Web Page Interactions: It can simulate user interactions like clicking, typing, submitting forms, and dragging and dropping elements, enabling tests to mimic real-world user scenarios.

Test Parallelism and High Performance: Playwright allows parallel execution of tests, saving time and enabling more efficient testing processes, especially for large test suites.

Screenshot and Video Recording: Screenshots and video recordings can be captured during tests, providing visual outputs for debugging and documentation purposes.
Standalone Operation and CI/CD Integration: Playwright can run independently and integrate with Continuous Integration (CI) tools, automating tests for every code update.

Ease of Writing Tests: Playwright supports multiple programming languages like JavaScript, TypeScript, Python, and C#, making it versatile for various development teams. Writing test automation scripts does not require advanced programming knowledge, and it is easier to learn compared to other tools, reducing training costs.
Companies generally show resistance when transitioning from manual to automated testing [find citation]. To minimize this resistance, it is crucial for the team to quickly learn and implement automation testing. For all these reasons, after extensive research, Playwright was chosen as the automation tool.

Our study specifically targeted testers without advanced software development expertise. To support the testers' learning process, a training course on Playwright was completed. This training consisted of approximately 6 hours of instructional video

content focused on writing automation scripts using TypeScript. Including both the training sessions and the initial tool installation, this phase of the project took around 2 weeks. Following the training period, coding the previously prepared 33 test cases with Playwright required 3 days.

Two significant aspects of test automation using Playwright were particularly emphasized:

i. Cross-Browser Execution Capability:

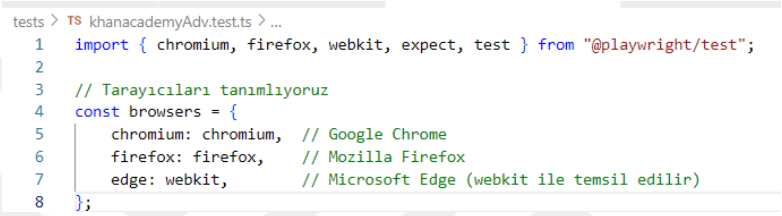

Figure 3.2: Execution on three different browsers

One of Playwright's strongest features is its ability to simultaneously execute tests across multiple browsers. The main purpose of running tests in parallel is to reduce the overall time and effort of automated browser testing. (Nagy, Maghawry, and Badr 2022). In this work, automated tests were successfully executed on Chrome, Firefox, and Edge browsers, utilizing relevant Playwright commands (Figure 3.2). Conducting cross-browser tests is crucial for ensuring web page compatibility, leading to a safer, more user-friendly, and professionally robust product. Browser compatibility testing also proactively identifies and resolves potential issues users may encounter.

Why Should a Web Page Be Tested on Different Browsers?
Testing a web page on different browsers is called "browser compatibility testing" and is a critical step in software testing for the following reasons:

Different Rendering Engines Used by Different Browsers: Each browser processes web pages using different rendering engines. For example:
Google Chrome: Uses the Blink engine.
Firefox: Uses the Gecko engine.
Safari: Uses the WebKit engine.

These engines can process the CSS, HTML, and JavaScript of a page differently. This may cause the page to appear or behave differently in each browser. For instance, a layout that works properly in one browser may break in another.

Differences Between Browser Versions: Browsers are constantly updated, and with each new version, features are added, removed, or modified. A web application may work on older browser versions but be incompatible with newer ones. Therefore, testing across different versions is essential to ensure the application reaches a wider audience.

Differences in JavaScript and CSS Support Across Browsers: Browsers may not support all JavaScript and CSS features in the same way. For example, some older browsers may not support modern JavaScript features at all or support them only partially. CSS features may also be handled differently depending on the browser, especially for modern features like CSS Grid and Flexbox.

User Experience (UX) and Visual Consistency: Web pages may appear differently in various browsers, particularly for responsive designs. Ensuring a consistent appearance and functionality across all browsers is critical for user experience.

Accessibility and Performance Issues: Different browsers may handle page load times, caching, and JavaScript execution differently, affecting performance and accessibility.
Diversity of User Base: Users have different browser preferences. Some may prefer Chrome, while others use Safari or Firefox. Ensuring that your web page works smoothly across all browsers ensures you cater to a broader user base. Testing your web page across different browsers ensures a safer, user-friendly, and professional product. With browser compatibility tests, you can identify and resolve potential issues users may encounter on various browsers in advance.

ii. Automated Screenshot Captures:

Playwright’s automated screenshot functionality allowed each executed test case to be visually documented through concise coding. These screenshots were automatically

saved to the desktop, providing test engineers immediate visual feedback regarding test execution outcomes. This capability significantly enhanced the quality of test reporting, enabling engineers to easily identify problematic cases, provide clear visual evidence to developers, and facilitate more insightful discussions within the team. Screenshots effectively addressed the 'Actual Result' component of the test cases, clearly illustrating the outcomes of each test (Figure 3.3).

Figure 3.3: Screenshot capture from test executions

3.4.1 Automation Test Execution by Artificial Intelligence

In this section, the ability of artificial intelligence to execute the test cases it created through automation was evaluated. The AI was instructed to write automation scripts for the test cases it had generated and to execute these scripts on its own. It was determined that the AI is not capable of running test cases through automation. The reasons for this are outlined below.

First of all, artificial intelligence can write automation scripts either for a case it created itself or one provided by a human in any programming language and using any test automation framework. This is a significant advancement widely utilized across the field of software engineering.

AI can write test automation code, but it cannot execute this code on its own. The written scripts must be refined and run locally by engineers. A physical or virtual browser (e.g., Chrome, Firefox) is required to execute the code. The AI's environment does not include a direct browser or graphical user interface (GUI). AI cannot perform live interactions such as clicking on web pages or filling out forms.

In order for the code to run properly, necessary dependencies (e.g., Selenium, ChromeDriver) must be installed on a user's machine or in a CI/CD environment. AI does not have access to an automated triggering or pipeline management environment like CI/CD platforms.

3.5 Installation and Jenkins Integration

Jenkins is a free, open-source Continuous Integration (CI) and Continuous Delivery (CD) tool designed to automate and streamline software development processes. Modern CI/CD practices aim to accelerate code validation and deployment by automating tasks such as code integration, testing, building, and deployment within software pipelines. Jenkins significantly simplifies DevOps workflows, enabling development teams to enhance efficiency, speed, and productivity. One key advantage of Jenkins is its ability to automatically initiate code executions directly from the Integrated Development Environment, eliminating the need for manual intervention. The tool provides a user-friendly interface that clearly displays test execution durations and results. Additionally, Jenkins integrates seamlessly with advanced reporting tools like Allure, enabling teams to generate comprehensive and easily interpretable reports. Due to these features, incorporating Jenkins into advanced test automation practices has become essential.

In this paper, learning Jenkins, configuring the environment, and integrating it into the Playwright automation setup took approximately 1 week. Initially, automation scripts failed to execute properly due to insufficient CPU resources, particularly when running tests in parallel. To resolve this, a custom function was developed to optimize and simplify the script, significantly reducing CPU load and increasing script readability. This optimization enabled the successful execution of parallel tests. Another significant benefit provided by Jenkins is its automated scheduling capability. As illustrated in Figure 3.4, the Jenkins pipeline was configured to trigger automatically at 21:00 each evening. As long as the server hosting the source code remains operational, Jenkins initiates the test runs at the designated time without manual input. At the conclusion of each execution, an intuitive visualization of the pipeline steps was provided through Jenkins' user interface, enhancing the clarity and readability of test outcomes.

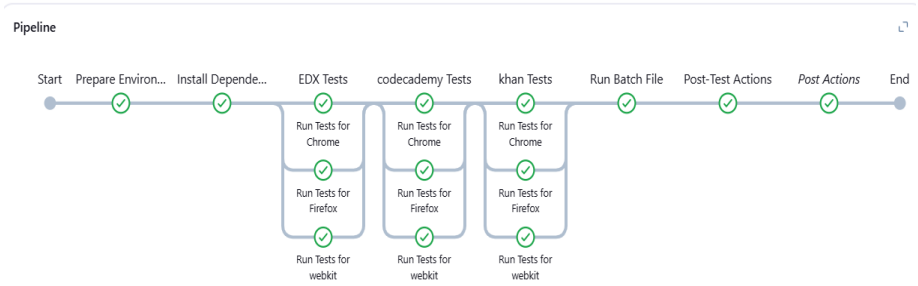

Figure 3.4: Jenkins pipelines

3.6 Reporting

Reporting is one of the most critical stages of the testing phase. The reports that are generated can be shared with all teams. These reports provide both the technical team and project managers with a brief summary of the project's status. Reports should be created after each test execution. For example, if five regression tests are executed for a module, there should be a separate report document for each regression test resulting in a total of five distinct reports. For manual tests, these reports must be created manually. For automation tests, manual reporting can still be used; however, to speed up the process, tools can be integrated into the automation framework to generate reports automatically. This way, test engineers do not need to spend time on reporting. Both the technical team and managers can review the reports immediately after automation test executions. In this study, an Excel-based report was created as an example of manual reporting. Additionally, the Allure reporting tool was integrated into the Jenkins environment, enabling the automatic generation of reports after each automation run.

To ensure that reporting is sufficiently comprehensive, there are certain components that must be included. An Excel template containing these components was systematically documented. This template consists of four sheets, which are titled: Summary, Test Report, Bug Report, and Legend.

3.6.1 Summary

Project/Module Name**:** Refers to the name of the application or module being tested.

Test Date: Indicates the date on which the testing was performed.
Test Environment: Includes information such as the operating system, browser, and database used.

Tester: Records the name(s) of the engineer(s) who performed the test.

Test Cycle / Release Name: Indicates the sprint in which the test was conducted (especially important for agile projects).

Total Test Cases: Contains information on the total number of test cases. See: 2.1

Passed Tests: Indicates the number of test cases that were successful, i.e., the number of test cases with a PASS status.

Failed Tests: Indicates the number of test cases that failed, i.e., the number of test cases with a FAIL status.

Chart: This section includes a chart that displays the percentage distribution of passed and failed test cases. By examining this chart, the final status of the testing phase can be quickly understood. This allows project managers in particular to easily assess the current state of testing. After each regression test, reviewing the project's charts enables a clear and quantitative judgment to be made regarding the progress of test case fixes.

3.6.2 Test Report

Test Case ID: The IDs in this section are identical to those listed in Section 2.1. The Test Case ID serves as a primary key for both the test cases and the test report.
Test Name: The test case names in this section are identical to those listed in Section 2.1.

Priority Level**:** This is the value assigned to test cases to assist with prioritization. It includes three levels: Low, Medium, and High. In different projects, additional options can be added to these three levels as needed.

Status: Records whether the test case has passed or failed.

Assignee: Stores the name(s) of the engineer(s) who executed the test.

3.6.3 Error Report

Test Case ID: The IDs in this section are identical to the test case IDs listed in Section 2.1. The Test Case ID serves as a primary key for both the test cases and the test report.
Bug Description: A non-technical explanation of what the issue is.

Severity Level: Assigned to bugs. Indicates the impact of the bug on the software.

Assignee: Stores the name(s) of the engineer(s) who performed the test.

Resolution Status: Indicates whether the technical team has resolved the issue.

3.6.4 Legend

Legend is a section that explains the meanings of the terms or categories used in the report. It is usually placed at the bottom of the report or added as a separate sheet. This helps both the technical team and project managers easily understand what each value represents. For example, the Severity value in the Bug Report section is often confusing with the Priority value in the Test Report section. Including these definitions in the Legend section will improve clarity and understanding among team members.
A portion of the test report described above is visualized as shown below.

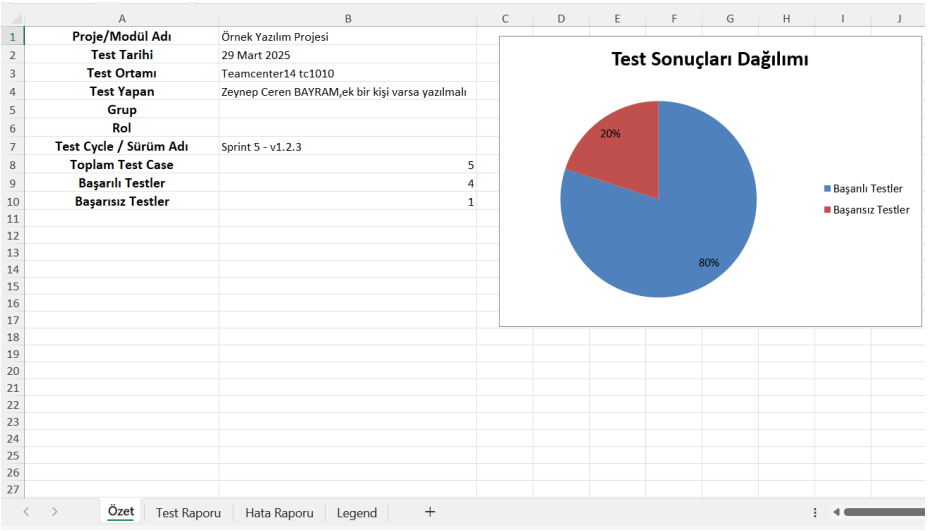

Figure 3.5: Test Report Example

Below is a sample excerpt from an Allure report that was automatically generated following the automated execution of the test automation. Time was only spent on the initial integration. After that, the report was created automatically without any additional effort.

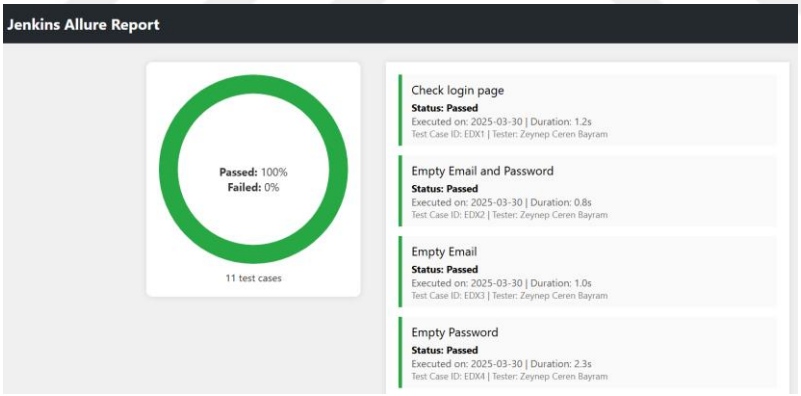

Figure 3.6: Allure Test Report Example

4 COMPARISON

4.1. Automation Test

Following the completion of the initial setup and training phase, subsequent executions of automation tests incur no additional time costs, since scripts run automatically without requiring manual intervention. Test executions are typically scheduled outside working hours, ensuring efficient utilization of human resources. The entire process including writing test cases, learning and installing Playwright, coding automation scripts, learning and implementing Jenkins, and executing automated tests was completed within 20 working days (Table 4.1).

Assuming an 8 hours workday, the total time investment amounted to: 20 days * 8 hours/day * 60 minutes/hour = 9,600 man/minutes.

Table 4.1: Time spent on automation testing

Task	Time
Writing test cases	2 days
Learning and installation of Playwright	10 days
Coding	3 days
Learning Jenkins	3 days
Jenkins implementation	2 days
Automated test execution	0 minutes (automated)

4.2. Manual Test

The manual testing approach consisted only of writing test cases and manual test execution. The total time invested for this method was significantly less initially (Table 4.1).

Table 4.2: Time spent on manual tests

Task	Time
Writing test cases	2 days
Running test cases manually	43 mins

The total elapsed time for manual testing was calculated as: (2 days * 8 hours/day * 60 minutes/hour) + 43 minutes = 1003 man/minutes**.**

4.3. Time Comparison of Manual and Automation Tests

First, it is important to address a topic that needs clarification. Software companies do not develop their projects directly in the live environment where real data is stored. Generally, companies work with three different servers: Development, Testing, and Production.

i. Development Environment: This is the environment where developers create ne**w** features and fix bugs. Changes made here do not affect the end user.

ii. Testing Environment: This is the environment where the software is tested, bugs are identified, and corrections are made. A structure similar to the production environment is used to verify that the software functions properly.

iii. Production Environment: This is the environment where real users interact with the software. The most stable and secure version of the software operates here.

There are different reasons for using three different development environments. These are as follows:

Risk Management: Testing the software independently at each stage helps prevent errors from occurring in the production environment.

Interaction and Isolation: Since each environment serves a different purpose, development, testing, and production processes can proceed without interfering with one another.

Performance and Scalability Testing: The testing environment is used to simulate how the software will perform in the production environment.

Controlled Deployment: After being gradually developed and tested, the software is deployed to production in a safe and controlled manner.

As a result, using three environments improves software quality, detects errors early, and ensures a secure production environment. Once developments are transferred to these environments, the testing process begins. In this study, the deployment amount has been planned as follows, and calculations have been made based on this amount.

Weekly Deployment: Average of 6 test and development environment deployments + 1 production deployment = 7 deployment within 1 week.

Cases that have already been tested and have received a "pass" status in the previous version of the software should be tested again after each deployment. These types of tests are called regression tests. Their importance is listed below.

Ensuring New Features and Bug Fixes Do Not Affect Other Parts: Adding a new feature or fixing a bug in the software can unintentionally disrupt existing functionalities. For instance, a change made to a user interface might adversely affect another feature or function. Regression testing is a critical step to detect such issues early.

Preservation of Existing Functionality's Accuracy: Numerous changes are made over time in software projects. It is essential that these changes do not negatively impact the existing functionality. Regression testing ensures the preservation of functionalities from earlier software versions. This allows users to continue using features available in previous versions of the software.

Continuity of Testing and Quality Assurance: With every change made in the software development lifecycle, the overall quality of the software must be maintained. Regression testing ensures consistency and quality in each new software release. Conducting these tests regularly enhances the software's reliability.

Time and Resource Savings: Regression testing can reduce the intensity of manual tests and provide faster feedback through continuous automated testing processes. Automation testing enables repetitive execution of the same tests, leading to significant savings in both time and resources.

End User Experience: Another significant benefit of regression testing is the preservation of user experience. Even minor changes in the software can impact user interactions. Regression testing helps prevent errors that might negatively affect end users.

Therefore, test cases created by test engineers should be repeated after each code update and deployment. Table 4.3 provides a comparative analysis of manual and automated testing over various intervals:

Table 4.3: Timing of manual and automation tests implementation

Scenario	Manual Testing	Automation Testing
Initial setup & 1st run	1,003 mins	9,600 mins
2nd run	43 mins	-
1 month (daily tests, 7*4 runs)	7*4*43 =1,204 mins	-
6 months (daily tests, 7*24 runs)	7*24*43 =7,224 mins	-
7 months (daily tests, 7*28 runs)	7*28*43 =8,428 mins	-
8 months (daily tests, 7*32 runs)	7*32*43 =9,632 mins	-

Weekly Deployment: Average of 6 test environment deployments + 1 production deployment.

When 7 deployments are performed weekly, the number of tests corresponds to 7 per week.

Thus, the monthly number of test executions would be: -> 7*4

The number of test executions over six months is calculated as follows ->7*24

Initially, manual testing requires significantly less time investment compared to automation. However, during frequent test cycles such as daily regression testing the repeated execution of manual tests quickly becomes time-consuming and costly. Within approximately 8 months, the cumulative time spent on manual testing surpasses the initial automation setup time (Table 4.3).

This analysis also considers the scenario in which the organization is transitioning from manual to automated testing for the first time, thus including the additional learning and initial implementation period. However, once automation skills are acquired, future projects can leverage the existing automation infrastructure with significantly reduced setup time. Therefore, a more realistic comparison should consider scenarios in which automation knowledge is already established within the team.

Table 4.4: Testing after acquiring necessary skills

Scenario	Automation Testing
Writing test cases	2 days
Learning and installation of Playwright	0
Coding	3 days
Learning Jenkinks	0
Implementation of Jenkins	0
Running test case with automation	0

Thus, as detailed in Table 4.3, the total elapsed time after skills acquisition is reduced to: 5 days * 8 hours/day * 60 minutes/hour = 2,400 man/minutes.

In contrast, manual testing still requires a consistent effort of 1,003 minutes for each testing cycle. The long-term advantage of automation becomes clear with repeated test cycles, as shown in Table 4.4. The time spent on manual testing will remain constant at 1,003 minutes: 2 days * 8 hours/day * 60 minutes/hour + 43 minutes = 1,003 man/minutes.

This comparison highlights that while the initial time cost for automation is higher (2,400 minutes), the manual testing time remains fixed at 1,003 minutes for each execution cycle. As the number of cycles increases, the cumulative time cost for manual testing will surpass the one-time automation setup cost, demonstrating the long-term efficiency of automated testing.

Table 4.5: Comparison of automation skills acquired

Scenario	Manual Testing	Automation Testing
1st run	1,003 mins	2,400 mins
2nd run	43 mins	-
1 month (daily tests, 7*4 runs)	1,204 mins	-
2 months (daily tests, 7*8 runs)	2,408 mins	-

As demonstrated, once automation is established within the organization, the initial investment for automation testing is quickly recovered. Within approximately 2 months, automation testing becomes more time-efficient compared to repeated manual testing delivering substantial long-term benefits. This emphasizes that despite the higher upfront cost, automated testing proves significantly more economical and efficient, especially in scenarios involving continuous regression testing (Halani, Kavita, and Saxena 2021).

4.4. Time Comparison of Manual and Automation Test Reports

Table 4.6: Time spent on manual reporting

Task	Time
Create a reporting template	1 day
Filling report as manually	120 mins

Table 4.7: Time spent on automation reporting

Task	Time
Implement the allure to jenkins	1 day
Create report as automatically	-

Table 4.8: Comparison of manual and automation reporting

Scenario	Manual Reporting	Automation Reporting
1st run	510 mins	480 mins
2nd run	30 mins	-
1 month (daily tests, 7*4 runs)	7*4*30=840 mins	-
2 months (daily tests, 7*8 runs)	7*8*30=1680 mins	-

The initial report generation step for a manual report includes preparing the manual test report template and performing a one-time reporting process. (8*60+30 = 510 mins)
The second execution includes the time spent preparing the report for a single test run.

Weekly Deployment: Average of 6 test environment deployments + 1 production deployment.

When 7 deployments are performed weekly, the number of reports generated corresponds to 7 reports per week.

The number of reports generated per month is calculated as follows: -> 7*4;

The number of reports generated over two months is calculated as follows -> 7*8 dir.

The initial report generation step for automated reporting includes the time taken to integrate Allure reporting into Jenkins. (8*60 = 480 mins)

4.5. Comparison of Different Testing Methods

Up to this point in the study, manual and automated tests have generally been compared with each other in the context of regression testing. However, test types are not limited to regression testing alone. In software testing, there are numerous types of tests. Not all of these test types aim to verify whether functions are working correctly. For example, user interface (UI) testing is one of the types of tests. In these tests, components such as interface layouts, fonts, buttons, and boxes are evaluated in terms of their appearance, dimensions, and colors. These elements are tested to determine whether they are user-friendly and how they are perceived visually by the user. Since these tests require human perception, it is more appropriate for them to be conducted manually. Below, a list of the most commonly used software testing methods well-known by everyone working in the field of test engineering is provided, along with a discussion on which method (manual or automated) is more suitable for each type of test.

4.5.1 Functional Tests

Functional tests constitute a significant part of software testing and are conducted to evaluate whether the software operates in accordance with its functional requirements. Functional testing checks whether the software performs correctly and measures its compliance with design documents, user requirements, and business rules. In other words, it examines whether the software performs a specific function and whether it does so correctly. The primary goal of functional testing is to verify whether the software meets users' expectations. These tests do not focus on the overall performance of the software or how the system responds to external conditions. Instead, the emphasis is placed on whether the software produces the expected output accurately.

Regression Testing: Regression testing is performed to check whether functions that previously worked correctly have been disrupted after a change in the software. The aim of this test is to ensure that the features which operated properly in earlier versions of the software continue to function correctly in the new version as well. It is crucial to verify that newly added features do not conflict with existing ones or cause errors.

Can be performed through automation: Since regression tests typically require the same tests to be executed in every software release, automating them is beneficial. Automation ensures that these tests are conducted quickly and accurately.

User Acceptance Testing – UAT: Acceptance testing determines whether the software meets the end-user requirements. This test checks whether the software is approved or accepted by the customer or end user. It is conducted after the software has been developed, to assess whether it fulfills the defined user requirements.

Best performed manually: User Acceptance Testing (UAT) is typically carried out by the end user, which is why it needs to be performed manually. During this process, users share their real-world experiences regarding the software's functionality.
System Testing: System testing evaluates whether the entire system functions correctly as a whole. This test is conducted after all components have been integrated, and its purpose is to verify that all functions of the software operate properly.

It can be performed manually or through automation: In large systems, system testing is essential to ensure that all components work together harmoniously. Most of these tests can be automated, as broad and comprehensive test scenarios need to be repeated quickly with each new release.

Integration Testing: Integration testing checks how different modules or components work together. After individual modules have been tested separately, they are combined, and the system is checked to ensure that it functions correctly as a whole.
Best performed through automation: Automation ensures the repeatability of integration tests, as the combined modules need to be tested continuously. Automated testing makes it easier to perform these repeated checks efficiently and reliably.

Smoke Testing: Smoke testing quickly checks whether the core functionalities of the software are working correctly. It is usually performed right after the initial installation

of the software. If the essential functions are operating properly, more detailed testing can then proceed. It can be performed through automation: This test is typically quick and frequently executed, so performing it through automation is beneficial. Core functionalities should be checked rapidly in every release.

4.5.2 Non-Functional Tests

Non-functional tests examine various aspects of the software that lie beyond its basic functionality. These tests evaluate the software's performance, security, usability, and more. Non-functional testing focuses on factors such as user experience, how the system utilizes resources, and how it responds to external conditions. It not only checks whether the software works correctly, but also whether it operates efficiently, securely, and in a user-friendly manner. Non-functional tests are critical for improving the overall quality of the software, because it is not enough for a system to merely function correctly it must also deliver a high quality, secure, and efficient experience.

Performance Testing: Performance testing evaluates the speed, response time, resource usage, and overall performance of the software. It is being conducted to assess how

efficiently the software operates and how well it responds to user demands. Can be performed through automation: Performance tests are typically conducted using automated tools, especially to observe how the software behaves under repeated load conditions. It is important for these tests to be executed quickly, as the software's performance may vary with each new release.

Load Testing: Load testing evaluates how the software performs under a specific workload. This test assesses the system's capacity to operate under normal usage conditions. Performed through automation: Load testing typically involves large amounts of data and processing, which makes automation the most suitable method for repeating and simulating these tests efficiently.

Stress Testing: Stress testing checks how the software behaves under extreme load conditions. This test is conducted to observe how the software responds when pushed beyond defined limits. Scenarios where the software is expected to crash or produce errors are intentionally tested. It can be performed through automation: Like load testing, stress testing involves a large volume of load and processing, so automating it is generally more efficient.

Security Testing: Security testing identifies vulnerabilities in the software and evaluates how resilient the system is against malicious attacks. These tests are conducted to detect security weaknesses within the software and to observe how the system responds to threats such as penetration testing. It Can be performed manually and through automation: In security testing, automated tests are typically used for configuration checks and system scans, while manual testing is necessary for more complex attack scenarios and situations such as social engineering.

4.5.3 Configuration Testing

These are tests conducted to determine whether the software functions correctly in different environments and system configurations. They check whether the software operates properly across various hardware and software settings.

Compatibility Testing: Compatibility testing checks whether the software operates correctly across different operating systems, browsers, or devices. This test evaluates the usability of the software by a wide range of users working in various environments. It Can be performed manually and through automation: In compatibility testing, manual tests are useful especially in situations that require visual inspection, while automation is commonly used to ensure consistency across different platforms.

Installation Testing: Installation testing checks whether the software is installed correctly and whether any errors encountered during the installation process are properly reported. This test verifies whether the software has been installed accurately and completely on the user's system. Best performed manually: Since software installation is a process that typically requires human interaction, installation tests are generally carried out manually.

4.5.4 User Interface (UI) and Usability Testing

User interface (UI) testing checks the accuracy of the software's visual elements and user interactions. Usability testing, on the other hand, evaluates whether the software is user-friendly.

User Interface (UI) Testing: User Interface (UI) testing checks whether the visual components of the software are functioning correctly, properly positioned, and free of interaction errors with the user. Elements such as buttons, menus, and form fields are tested. Best performed manually: The correct functioning of visual components, their proper appearance, and the absence of errors in user interaction are generally examined manually.

Usability Testing: Usability testing evaluates how user-friendly the software is and whether users can use it easily. It focuses on factors such as how long it takes for users to learn the software, how easy it is to navigate, and the overall user experience.
It Must be performed manually: User interactions and feedback are elements that need to be gathered in real-time; therefore, usability testing is generally conducted manually.

4.5.5 Error Management Testing

These are tests conducted to evaluate the software's error management processes and to verify whether errors are reported and corrected properly.

Negative Testing: Negative testing checks how the software behaves when given incorrect or unexpected inputs. It is conducted to verify whether the software can handle unexpected situations appropriately. Best performed manually: Manual testing is necessary to observe the software’s response to incorrect data and to report any resulting errors.

Each type of software testing serves a specific purpose and, when used in the appropriate context, contributes to improving software quality. Manual testing is especially critical in scenarios that require human interaction, such as detecting visual defects and evaluating user experience. On the other hand, automated testing enables the rapid execution of repetitive and time-consuming tests, ensuring that the software is tested consistently and efficiently. Using both manual and automated testing together enhances the overall effectiveness of the software testing process.

5 DISCUSSION ON COMPARISON METRICS

This section provides an in-depth analysis of various comparison metrics to evaluate manual and automated testing methods. The impact of each metric will be discussed based on the nature and requirements of different project types.

5.1 Time

Under this metric, the time required for setting up the automation infrastructure, executing test cases for the first time, and performing regression testing after each deployment will be evaluated. Activities considered include: Writing test cases (common to both methods), Preparing automation infrastructure, Coding test cases, Executing automated test cases, Executing manual test cases, Conducting regression tests

The set of 33 test cases was executed using both manual and automated testing approaches, and the total elapsed times were recorded and analyzed. Six scenarios were investigated to assess the efficiency and suitability of test automation under different project conditions:

i. Long-term, Multi-user Projects: These projects typically include banking applications, defense systems, ERP and CRM systems, extensive e-commerce platforms, telecommunications and network management software, e government initiatives, and large-scale data analytics platforms. Such projects have extensive budgets, long lifecycles, larger teams, and continuous maintenance requirements, emphasizing critical attributes like performance, security, and scalability. Such companies will need to

maintain and update the projects they develop for many years. With each update, performing regression tests will be essential. In addition to regression testing, adopting automation will become increasingly important.

ii. Existing Automation Infrastructure (Long-term Projects): If the company has already possessed an automation infrastructure the initial automation setup costs are amortized within 2 months compared to manual testing. Given that these projects typically continue well for 2 months, adopting automation consistently proves beneficial and profitable.

iii. Initial Transition to Automation (Long-term Projects): For companies newly adopting automation, the initial setup investment is recovered within approximately 8 months. Thus, automation remains a profitable decision if the project's lifecycle is anticipated to exceed this duration.

iv. Short-term Projects with Uncertain Continuity: Examples of such projects include R&D initiatives, proof of concept (PoC) studies, hackathons, rapid MVP developments, IoT projects, or laboratory-based research. These projects usually have limited budgets, smaller teams, tight timelines, and uncertain future commercialization. The primary objectives involve feasibility assessments, rapid prototyping, and technology experimentation.

v**.** Initial Automation Adoption (Short-term Project): For short-term projects where automation infrastructure does not yet exist, the substantial time required for automation setup (including training) may not justify the investment. Due to limited use and lack of regression testing continuity, the automation effort could result in an overall loss, rendering automation less advantageous in such cases.

vi. Existing Automation Infrastructure (Short-term Projects): Even for short-term projects, companies with established automation frameworks may still benefit from test automation. Project managers should carefully evaluate expected project duration, future potential, and overall project lifecycle when deciding whether to automate.

The analysis indicates that the decision to adopt automation testing evolves with project context (Rogers and Miles 2015). Project teams should comprehensively assess their specific circumstances, using the metrics and insights presented, to make informed decisions about automating their test processes.

5.2 Speed (Test Execution Time)

In this metric, test execution times for both manual and automated methods were measured and compared across the three selected user interfaces. A total of 33 test cases were executed using each method, and the time taken for each execution was recorded individually. Factors not directly related to execution, such as initial automation setup or scripting effort, were excluded to provide a fair comparison focused solely on execution speed. The results demonstrated that manual execution of the complete set of 33 test cases required approximately 43 minutes, whereas automated execution of the identical test cases took only 36 minutes, showing a reduction of 9 minutes in total execution time. Although the observed difference appears modest, it is likely to become more significant as the number of test cases increases. When both testing methods are initiated concurrently, automated testing consistently completes the execution cycle sooner, enabling quicker feedback and results delivery. This outcome aligns with previous findings indicating that automated testing accelerates execution cycles, enhances responsiveness in test processes, reduces repetitive manual tasks, and minimizes overall human effort (Klammer and Ramler 2017).

5.3 Team and Skill Requirements

Manual testing does not require advanced technical skills; once test cases are developed, testers can execute them using basic instructions and intuitive human interaction. In contrast, automation testing demands specialized technical knowledge, particularly coding proficiency, even when using existing test frameworks. Consequently, teams lacking this expertise must undergo specific training to effectively integrate test cases into automation frameworks.

Initially, the selection of an appropriate programming language and automation framework is critical. Subsequently, training must be identified and provided in a cost effective and timely manner. During this training period, it is recommended that manual testing continues concurrently to avoid project delays. The study examined two scenarios regarding automation implementation: companies with prior automation expertise (with acquired knowledge') and those without ('without acquired knowledge'). The analysis identified a significant difference approximately 15 working days in the time required to set up automation infrastructure between these two scenarios. This additional time and resource investment represent a common barrier, contributing to organizational resistance to automation adoption. Therefore, project managers must carefully evaluate team skill sets, project duration, and anticipate return on investment when deciding whether to transition from manual to automated testing.

5.4 Reporting Quality

Reporting is a critical component of the software testing phase, providing formal documentation of test outcomes, numerical data, and overall results. Clear and efficient reporting directly influences the quality and reliability of test processes. In manual testing, reports including pass/fail counts, percentages, and test execution summaries are prepared manually after each test cycle, requiring recurring effort from test engineers. Conversely, automated testing enables automatic report generation through integrated tools such as Jenkins and Allure. The automated approach eliminates repetitive manual calculations, providing immediate visibility of test outcomes, pass/fail statuses, and detailed execution summaries. In this study, Allure reporting was integrated with Jenkins, enabling automatic report generation upon each automated test execution. Integrating Allure required approximately 1 working day (480 minutes). After this initial setup, reports were generated automatically without additional time or cost for subsequent runs. In contrast, preparing equivalent manual reports (e.g., using Excel) took approximately 1 hour (60 minutes) per test run, which accumulates with repeated execution cycles.

Thus, the initial investment for automating the reporting process is amortized after approximately eight manual reporting cycles. Consequently, automatic report generation

through Jenkins-Allure integration significantly improves efficiency, reduces long-term reporting costs, and enhances the reliability of test documentation, particularly in projects involving frequent regression testing (Dobles, Martínez, and Quesada-López 2019).

5.5 Automated Test Initiation

Manual testing requires test execution schedules to be planned and initiated manually by test engineers, introducing potential scheduling issues and human errors. In contrast, automated testing frameworks, such as Jenkins, enable automatic initiation of test execution at predefined times, reducing the likelihood of human error and planning complexities.

In this study, automation scripts were scheduled via Jenkins to run outside working hours. By the start of the next working day, tests were completed, and results were readily available through automatically generated reports. Consequently, automation significantly saved testing time and reduced additional working hours required for manual test execution and reporting tasks. According to a survey of more than 400 software developers, approximately 70% reported that automated continuous integration practices enable earlier bug detection and reduce anxiety related to build failures (Li et al. 2023).

5.6 Cross-Platform Compatibility

Ensuring software reliability and usability across various browsers and platforms is crucial. Manual testing necessitates individual test executions across different browsers, which is repetitive and time-consuming. Automation frameworks, however, allow simultaneous execution of the same test cases across multiple browsers, greatly reducing testing effort and execution time. In this study, automated test cases written once were simultaneously executed across multiple browsers specifically Chromium, Mozilla, and WebKit demonstrating significant time savings compared to separate manual executions. Thus, automation offers substantial efficiency gains for projects requiring extensive browser compatibility testing.

5.7 Test Coverage (GUI Testing)

Graphical User Interface (GUI) testing, which involves validating graphical elements such as icons, buttons, images, menus, colors, and layouts, remains challenging for automated methods. Automated tests typically interact solely with the underlying HTML code and cannot accurately assess visual design aspects or dynamic graphical elements. Thus, GUI testing significantly benefits from human intuition, perception, and sensitivity, representing a clear advantage of manual testing. Automated testing frameworks cannot effectively detect visual inconsistencies or validate subjective design elements. Consequently, projects requiring detailed visual inspections and user experience assessments should continue utilizing manual testing to complement automated processes.

5.8 Cost

Automation testing may incur substantial costs if proprietary tools are employed. However, utilizing open-source or free tools effectively mitigates these expenses. In this study, automation was implemented using freely available tools, including Playwright, Jenkins, and Allure reporting. Additionally, training in automation skills (specifically TypeScript scripting) was conducted using freely accessible online resources, resulting in no direct software or licensing costs. Nevertheless, transitioning teams from manual to automated testing requires training, which may involve future expenditures if more advanced training becomes necessary. Initially, freely available resources can suffice; however, as team skills evolve and project complexity increases, additional paid training may be warranted. These potential training costs should be considered when evaluating the long-term investment in automation.

5.9 Flexibility

Flexibility, in the context of software testing, refers to how well a testing method can adapt to software development processes, changing requirements, innovations, or modifications. Flexibility is one of the key comparative metrics, especially when

evaluating both types of testing automation and manual. It determines how tests will perform and adapt under different conditions.

The flexibility of automation testing may be more limited at first, but when properly configured, it can offer a high degree of adaptability. To better understand the flexibility of automation tests, let's examine a few key factors:

Challenge: Automation tests can be sensitive to changes made in the software. Especially when modifications are made to the user interface or functional structures, automated tests need to be updated. For example, if the user interface changes, automated tests that interact with the UI must also be rewritten. Ease: However, a well-designed automation test infrastructure can quickly adapt to such changes. When tests are written using parameterized and modular configurations, it becomes easier to add new scenarios and adjust to changes. With increased modularity, only the affected parts need to be updated.

Challenge: Initially, setting up automated tests can be time-consuming, and integrating them into Continuous Integration (CI) processes can sometimes be challenging. Additionally, with every new feature or bug fix, the tests may need to be updated. Ease: Once established, automation tests provide repeatable and reusable testing capabilities. This is especially beneficial in Continuous Integration/Continuous Delivery (CI/CD) systems, where tests can be run continuously. Flexibility increases when a test can cover various scenarios and configurations.

Challenge: Automation generally offers broad coverage, but it may not be able to cover every type of test in all situations. Specifically, in human-centered areas such as user interaction and UI/UX, the flexibility of automated tests is limited. Ease: However, test automation can be extended to cover the functional aspects of the software. A properly configured automation framework provides the flexibility to test various configurations and environments of the software.

Manual tests are generally considered more flexible because the person writing the tests can quickly review any changes made in the software and develop appropriate tests

accordingly. However, to better understand the flexibility of manual testing, we can focus on the following points:

Challenge: However, in manual testing, tests need to be redone for every change, which can be time-consuming. The extended test duration may reduce overall efficiency. Ease: Manual tests can quickly adapt to changes in the software. When new features or functions are added, manual tests can be swiftly updated to reflect these changes. Testers can immediately detect any modifications in the software and adjust the test scenarios accordingly.

Challenge: Since each new scenario must be tested manually, this type of testing takes more time and can slow down the project. Ease: In manual testing, it is quite easy to create and apply a wide range of test scenarios. Especially when dealing with new functionalities or complex user interactions, testers can manually develop and execute new scenarios.

Challenge: If a large number of tests need to be conducted manually, this can lead to a significant labor requirement and make it difficult to comprehensively test all aspects of the software. Ease: Manual tests are more flexible when it comes to evaluating user experience and complex scenarios. Human testers are better equipped to assess interactive and aesthetic aspects of the software, error states, user feedback, and similar factors.

Flexibility is an important factor in both automation and manual testing, but it functions differently in each. Manual tests adapt quickly to software changes and can more flexibly evaluate aspects such as user interaction. On the other hand, when properly configured, automation tests can provide large-scale, repeatable testing even in the face of continuous software changes but they require an initial investment. The flexibility of each approach can be more beneficial in different scenarios. Automation is more suitable for large-scale, repetitive tests, whereas manual testing is more adaptable for complex, unique, or user-focused scenarios.

5.10 Ease of Maintenance

This metric evaluates how easily test scenarios, automation frameworks, and testing processes can be updated over time, how effectively errors can be identified and corrected, and how well the testing adapts to changes in the software. The challenges of test maintenance can increase in parallel with the evolution of the software. Each new feature addition, bug fix, or structural change in the software may require updates to existing tests. This process functions differently in manual and automation testing; however, in both approaches, ease of maintenance should be considered an important metric.

Ease: One of the most important factors that simplifies the maintenance of automation tests is writing them in a modular structure. In other words, when each test scenario is divided into small, independent modules that test specific functionalities, only the affected module needs to be updated when a change occurs in the software. This speeds up the maintenance process and requires less manual effort. Challenge: If the tests are tightly coupled and lack a modular structure, a single change in the software can impact on the entire test suite, making the maintenance process more complex. Let me know if you'd like to continue with the next part.

Ease: The flexibility of the automation testing infrastructure can simplify the maintenance process. The test automation framework should be able to quickly adapt to changes in the software. For example, it should make it easy to add new test scenarios or modify existing ones. Challenge: If the automation framework is not sufficiently flexible, significant updates may be required to accommodate new requirements or software changes, which can complicate the maintenance process.

Ease: Automation tests can generally be executed quickly. This makes it easier to rerun tests after software changes and allows for rapid maintenance. Additionally, integrating tests into Continuous Integration (CI) processes can automate the maintenance workflow. Challenge: However, if the test suite is very large and complex, rerunning all tests after each change can be time-consuming, which may slow down the maintenance process.

Ease: The maintenance of automation tests should be carried out in parallel with changes made to the software. If these changes affect the scope of the tests, updates must be integrated quickly. Automation tests should remain compatible with the latest version of the software. Challenge: Major structural changes in the software may require the automation tests to be rewritten. In such cases, maintenance becomes costlier and more time-consuming.

Ease: Because manual tests are more flexible, they can quickly adapt to changes made in the software. When new features or functionalities are added, manual tests can be rapidly updated, and new scenarios can be created. Testers can easily verify the modified functions and features. Challenge: However, updating and maintaining manual tests can be time-consuming. Since the tests must be performed manually with each new software release, this process requires additional labor and can reduce overall efficiency.

Ease: In manual testing, test scenarios are generally more descriptive and flexible. This means that changes made to the software can be quickly understood and applied by the person reviewing the tests. This flexibility makes maintaining manual tests easier. Challenge: However, if a large number of manual tests are being conducted, updating and coordinating each test properly can become difficult. In addition, the low repeatability of manual tests can make the maintenance process more complex in the long run.

Ease: Manual tests can quickly generate new test scenarios in response to changes made in the software. Manual testing can be especially flexible when it comes to evaluating user experience and complex interactions. Challenge: However, because manual tests often need to be rewritten for each new software release, continuously creating up to date and valid test scenarios can be time-consuming.

Ease of maintenance is a critical metric that affects sustainability and long-term efficiency of software testing. While automation tests offer advantages in terms of repeatability and speed at scale, their maintenance processes can be complex due to the initial investment required. Manual tests are more flexible and can quickly adapt to

changing software requirements, but over time, their maintenance costs and labor demands can become high. Both testing methods have their own advantages and challenges in terms of maintenance ease. This metric should be considered when evaluating which testing approach is most suitable for a given project.

5.11 Repeatability

i. Repeatability in Manual Testing

In manual testing, repeatability ensures that tests are conducted consistently each time; however, there are certain challenges and limitations involved:

Human Factor: In manual testing, the experience, skill, and level of attention of the testers directly impact the consistency of the tests. When the same test is executed by different testers or even by the same tester at different times minor errors, differing interpretations, or conscious/unconscious variations may occur.

Environmental Variables: The test environment, user conditions, or devices used may vary, which can influence the results of manual tests. For example, distractions affecting the tester or a noisy environment can disrupt the consistency of manual testing.

Difficulty of Repetition: One advantage of manual testing is its flexibility, but since each test must be performed from scratch, fully repeating previous tests can be difficult. Because tests are conducted manually each time, they may take longer and may not be executed exactly the same way.

Documentation Challenges: If the test procedures and steps to be followed are not well-documented, it can be difficult to repeat tests. Even with detailed documentation, testers may sometimes skip steps or introduce small variations.

ii. Repeatability in Automation Testing

Repeatability in automation testing is more consistent and at a higher level. Some of the reasons for this include: Consistency: In automation testing, the written test scripts

follow the same steps and perform the same actions each time they are executed. Since there is no human error involved, tests generally yield the same results every time. Conditions and outcomes are much more consistent with each execution.

Time Efficiency: Automation tests are typically faster, which allows them to be performed more frequently and efficiently. This increases repeatability, as each test can be quickly rerun under the same conditions.

Traceability and Recording of Tests: In automation testing, every stage of the test can be recorded, and results can be automatically reported. This creates a clearer and more consistent record of how the test was conducted and ensures that the same test scenarios are followed in each run.

Effect of Environmental Variables: Automation tests are usually executed on virtual machines or in controlled environments, minimizing the impact of environmental factors on test results. When tests are run in a fixed environment, the results tend to be consistent. However, minor changes in the test environment (e.g., software updates) can still cause inconsistencies in automation tests.

Maintenance Requirements: Automation tests require ongoing maintenance over time. When new features or changes are introduced, the test scenarios must be updated accordingly. If test scripts are not kept up to date, the repeatability of the tests may decrease.

While manual testing offers advantages in terms of flexibility and variability, it can face challenges in repeatability. Ensuring consistency is particularly difficult when tests are conducted by different testers or at different times. In contrast, automation tests provide much higher consistency and repeatability. Once written, they can be run repeatedly under the same conditions, producing much more consistent results. However, it is important to remember that automation also requires maintenance, and environmental changes can sometimes cause inconsistencies even in automated tests. In conclusion, automation tests are generally more advantageous than manual tests in terms of

repeatability. However, both types of testing are valuable when used appropriately within their respective contexts and according to specific needs.

5.12 User Experience

User experience (UX) can be an important metric in software testing, especially in manual tests and certain automation tests that involve evaluating specific user interactions. Although UX is not directly related to the software's functionality, it plays a significant role in assessing the effectiveness of tests. This is particularly critical in areas beyond functional testing, such as usability testing and user interaction testing.

i. User Experience in Manual Testing

In manual testing, user experience is evaluated directly through observations and feedback from test experts. Testers experience the software firsthand, observing challenges that users may encounter, non-intuitive interfaces, difficult navigation paths, and overall ease of use. The following aspects can be considered during user experience testing:

Interactive Feedback: When users (test experts) interact with the application, gathering their feedback is relatively easy. This provides direct insight into the issues users face while using the software.

User Perspective: Test experts can use the software as if they were real users, evaluating whether it is user-friendly. This is known as usability testing and is highly valuable in understanding how effective the software is for its target audience.

Real-Time Observations: In manual testing, factors affecting user experience can be observed instantly. For example, difficulties users face while navigating, complex interfaces, or error messages can be identified more quickly.

Sensory Perception: During interaction with the application, testers can assess the impact of elements such as interface design, colors, fonts, icons, and visual components on the user. By directly observing user experience during manual testing, valuable

insights can be gained into how efficient, accessible, and intuitive the software is for end users.

ii. User Experience in Automation Testing

In automation testing, user experience is generally a difficult metric to measure directly, as automation tests primarily focus on testing the software's functional features and simulating user interactions. However, some of the advantages offered by automation can indirectly contribute to improving user experience:

Automated Tracking of User Flow: Automation tests can simulate specific user flows (e.g., user registration, login processes, or checkout procedures). This helps indirectly evaluate user experience by ensuring expected functionality and smooth operation of the application.

Repeatability: User experience can become repeatable and consistently observable through automation tests that simulate user flows. However, obtaining meaningful feedback about how intuitive the experience is or how easily users can follow the process remains difficult in this context.

Performance Monitoring: While testing software performance, automation tests can observe factors that impact user experience such as fast or slow response times and delays during user interactions. These observations can be important for indirectly evaluating UX.

Accessibility Testing: Automation tools can also be used to simulate accessibility testing. For instance, testing screen reader compatibility or keyboard navigation can provide valuable data about the user experience. However, automation testing has limitations in directly evaluating user experience, as human factors, user behavior, and emotional responses cannot be fully measured through automated scripts. While automation is useful for functional and performance testing, manual testing is more effective in directly assessing usability, visual design, and sensory perception.

In manual testing, user experience can be measured much more directly through observations. Factors such as how users interact with the software, the challenges they encounter, and the ease of use can be assessed in real time. This allows for a more thorough evaluation of the software, particularly in terms of usability and visual design. In automation testing, user experience is typically evaluated through functional performance and the testing of user flows. However, due to the limitations of automation, it is difficult to gather meaningful insights into human-centered metrics such as user interaction, visual design, and intuitiveness. In conclusion, user experience can be a highly meaningful metric in manual testing, while in automation testing, it is approached more indirectly and within certain limits.

5.13 Risk Management

Risk management is a highly important factor in the software testing process and is used to manage various conditions that affect software quality. In the context of manual and automation testing, risk management varies depending on the advantages and challenges of each approach. The primary goal of risk management is to minimize the likelihood of software failure and reduce the impact of such failures.

i. Risk Management in Manual Testing
In manual testing, risk management is largely dependent on the human factor, which brings both advantages and certain limitations:

Rapid Improvement and Adaptation: Manual tests allow test experts to quickly identify new risks based on their intuition and experiential knowledge. For example, if a tester notices an unexpected error during a new user interaction, they can immediately develop a solution. Test scenarios can be modified flexibly and adapted quickly to new risks. This enables rapid response in dynamic software development processes.

Testing Complex Scenarios: User focused tests are more effective in identifying user centered risks related to user experience and visual design. For example, issues concerning software usability or user interface (UI) errors are often better detected through manual testing.

Human Errors: In manual testing, the human factor plays a significant role. A tester may occasionally skip a specific test step or make an incorrect judgment, which can lead to overlooked risks. Distractions, misinterpretations, or tester fatigue during the testing process increase the likelihood of missing potential risks. This is directly related to the tester's level of attention and experience.

Time and Resource Constraints: Manual testing takes more time, which can delay the process of identifying risks. Due to time limitations and resource shortages, it may not always be possible to conduct all necessary tests. This can result in limited test coverage. Especially in large and complex projects, some test scenarios may be overlooked due to time pressure, and the project may proceed without certain risks being identified.

Test Coverage and Consistency: The coverage of manual tests can sometimes be limited, as testers may not be able to execute all test scenarios. This may result in the missing certain critical risks within the software. For example, manually testing a large number of scenarios is time-consuming, and some cases may be overlooked. The same tester may produce different results at different times. Such inconsistency in testing can weaken the ability to accurately analyze risks.

ii. Risk Management in Automation Testing

Automation testing offers different advantages and limitations in terms of risk management. This type of testing can be particularly effective for identifying specific risks in advance and continuously testing them.

High Consistency and Repeatability: Automation tests consistently execute the same scenarios every time. Independent of the human factor, they ensure that tests are performed in the same way repeatedly. This leads to the repeated detection of the same errors and yields more consistent test results. Once written, test scripts can be executed after each change in the software development process, enabling the identification of similar bugs across different versions of the software.

Continuous Monitoring and Rapid Detection of Risks**:** Automation tests enable the software to be tested at any time. Following a new software update or change, tests can be executed automatically, allowing new risks to be detected quickly. In Continuous Integration and Continuous Delivery (CI/CD) processes, automation tests are triggered after almost every update. This facilitates the early identification of risks during the development cycle.

Coverage and Performance Testing: Automation tests offer extensive test coverage in large and complex software projects. They can execute thousands of test steps quickly and under the same conditions. Additionally, automation tests are more effective in areas such as performance testing, load testing, and stress testing. These types of tests help identify how the software behaves under high load and assist in detecting potential performance risks.

Error Reporting and Tracking: Automation tests record every step of the testing process and report each error in detail. This makes it easier to track and monitor issues. Error reports are generally more detailed and consistent.

High Initial Cost and Time Investment: The initial setup of automation tests can be time-consuming and costly. Writing test scripts and integrating appropriate automation tools requires a certain level of investment, which results in high upfront costs. It is important to choose the right strategy before making this initial investment, as in some projects, automation may turn out to be more expensive compared to manual testing.

Lack of Flexibility**:** Automation tests are typically based on predefined scenarios, and any change in the software may require corresponding updates to the test scripts. When significant modifications are made to the software, the test scenarios must also be adjusted accordingly, which can sometimes be time-consuming and complex. Moreover, subjective tests such as user interaction, visual design, and usability testing are difficult to fully perform through automation. These types of tests are generally more suitable for manual testing.

Failure Scenarios: When improperly configured, automation tests can produce misleading results. For instance, a small change in the software may affect the test scripts and cause the test to fail, even though the software is actually functioning correctly. This can lead to incorrect risk assessments. Additionally, tests conducted solely through automation may fall short in identifying deeper user experience related risks.

While manual tests can more dynamically detect user interactions and UI/UX risks through human observation and experience, automation tests are more effective at early risk detection and identifying performance issues through systematic and continuous testing. Both types of testing are important for managing risks within their respective contexts, and the best results are often achieved by using them in combination.

5.14 Reliability

The reliability of a test refers to whether the tests produce consistent and accurate results. This metric checks whether the tests yield similar outcomes under the same conditions each time they are run. Reliability is assessed differently in the context of manual and automated tests.

i. Reliability in Manual Testing

The reliability of manual tests depends on the human factor, which can affect the consistency, accuracy, and validity of the tests.

Experience and Attention of the Test Expert: Manual tests rely on the knowledge, experience, and attentiveness of the test experts. If the tester is experienced and careful, the tests are generally more reliable. However, a less experienced tester may occasionally make mistakes, skip steps, or report incorrect results factors that can reduce the reliability of the tests. Tester fatigue and lack of concentration are also important factors that affect reliability. During long testing sessions, human error becomes more likely, making test results less reliable due to the limitations of the human factor.

Variety and Complexity of Test Scenarios: In manual testing, test scenarios are generally flexible. This allows the tester to examine different situations within the software. However, this flexibility can sometimes lead to test scenarios not being followed accurately or result in inconsistent outcomes. Testing complex scenarios or edge cases can also be challenging in manual testing. Such tests may sometimes be overlooked by testers or misinterpreted.

Inconsistency and Human Errors: Although experts in manual testing can customize test scenarios, overlooked bugs or misinterpretations are common occurrences. In large projects especially, recurring errors may go unnoticed due to a tester's lack of attention. Manual tests can yield different results depending on the tester. When the same scenario is executed by different testers, the outcomes may vary. For instance, one tester may detect a bug in the software, while another may overlook it.

Time and Resource Constraints: Due to time pressure and limited resources, some tests may be skipped or conducted hastily. This often leads to critical bugs being overlooked and weakens the reliability of the testing process. In long-term projects, it is possible for test experts to miss certain details or overlook specific bugs during testing.
Emotional State and Psychological Factors: The emotional state of test experts can also impact the reliability of testing. For example, a tester working under stress may overlook errors or execute test steps incorrectly.

ii. Reliability in Automation Testing
Automation tests can provide much more consistent and reliable results compared to manual tests, as they operate independently of the human factor. However, there are still some factors that can affect the reliability of automation tests.

Accuracy of Test Scenarios and Scripts**:** The reliability of automation tests depends on the accuracy of the written test scripts. If the test scripts are not written correctly, automation tests may produce incorrect results. The test data used in automation tests must also be accurate and up to date. Data gaps or incorrect data can cause automated tests to yield false results. Additionally, errors made while updating the scripts when needed can also affect their reliability. Even if test scripts are initially written correctly,

they must be updated in accordance with changes made to the software. If these updates are not performed, automation tests may produce invalid or inaccurate results.

Test Environment and Configuration: When automation tests are executed in a specific environment, the reliability of the tests can be affected if the environment is not properly configured. For example, when tests need to be run on different operating systems or hardware platforms, there is a risk that the automation tests may not function correctly. Most automation tests are configured to run in a specific test environment. If this environment is not updated or is misconfigured, the test results may be inaccurate.

Maintenance and Update Requirements: Any change made to the software also requires the automation tests to be updated accordingly. If the software is continuously updated but the automation tests are not revised in parallel, outdated tests cannot provide accurate results. This is a factor that reduces the reliability of the tests. Maintaining automation tests can also be time-consuming, and if not properly managed, it can negatively impact the reliability of the tests.

Misleading Test Results (False Positives/Negatives): Automation tests can sometimes produce misleading results. In a false positive situation, a test may be incorrectly marked as passed, while in a false negative case, actual errors may be overlooked. These types of errors typically stem from misconfigurations in automation scripts or issues in the test environment. Incorrect test scenarios or flawed coding can lead to inaccurate test reporting, ultimately affecting the reliability of the tests.

Advanced Test Coverage: Automation tests generally perform well in predefined, repeatable scenarios. However, testing more complex user interactions can be challenging. Abstract issues such as spontaneous user behavior or visual defects during software use may be difficult to detect through automation. Overlooking such errors can limit the reliability of automation testing.

While manual tests are more flexible and customizable, they can lead to fluctuations in reliability due to the human factor. On the other hand, automation tests provide consistent and fast results; however, factors such as improperly configured test

scenarios, software updates, and misleading outcomes can negatively impact their reliability.

6 AUTOMATION ADOPTION EVALUATION MODEL

In this study, a multi-criteria decision-making model is proposed to evaluate whether a software project should transition to automation testing. The developed model offers a more objective and quantitative approach by taking into account both project characteristics and team competencies. The model is based on positive criteria (supporting the transition to automation) and negative criteria (limiting or making the transition unnecessary), and these criteria are scored to calculate the overall impact.

6.1 Justifications for Positive Criteria Supporting the Transition to Automation Testing

i. The project being in a live/production environment:
   In projects that are live, the cost of software defects is significantly high. Therefore, it becomes critical to carry out testing processes in a fast, repeatable, and consistent manner. Automation testing effectively meets these requirements and continuously ensures software quality.
ii. A high or increasing number of users:
   An expanding user base means that even minor software bugs can lead to large-scale impacts. In this context, it is necessary to broaden test coverage and shorten feedback cycles; automation plays a key role in achieving this.
iii. Frequency of regression testing:
   Frequent updates in the software increase the risk of regressions. Repeating the same tests manually is inefficient in terms of both time and human resources. Automation makes it possible to perform such repetitive tests with less effort and fewer error.

iv. Stability and low variability of test scenarios:

Stable test scenarios minimize maintenance costs in the automation process. If the test logic does not change frequently, investment in automation provides long-term benefits.

v. Sufficient knowledge and experience in the team regarding test automation:

Effective implementation of automation requires a team with the necessary technical knowledge and tool proficiency. The presence of such competencies supports the successful and sustainable execution of automation.

vi. Availability of Continuous Integration/Continuous Delivery (CI/CD) infrastructure:

When automation tests are integrated into CI/CD pipelines, they ensure continuous quality control throughout the development lifecycle. This integration allows tests to run in parallel with code integrations, thereby systematically improving software quality.

vii. Requirement for the software to run on multiple platforms:

Applications running on various browsers, operating systems, or devices require diverse testing. Automation testing provides significant advantages in maintaining consistency across these environments.

viii. Need for high-frequency test repetitions:

One of the most efficient use cases for automation is when tests need to be repeated frequently. Automating such tests contributes greatly to saving time and reducing the risk of human error.

6.2 Justifications for Negative Criteria Limiting the Transition to Automation Testing

i. The project being in an early stage or R&D phase:

In projects that are in early development stages or have an experimental purpose, requirements and functionalities change frequently. This situation seriously complicates the sustainability and maintenance ease of automation scripts.

ii. The project being small-scale with a limited number of modules:

In projects with a narrow test scope, manual testing often offers more cost-effective solutions. The initial setup costs of automation can be disproportionately high for small projects.

Table 6.1: Automation Transition Evaluation Table

Criteria	Criterion rating (x_i)	Weight (w_i)	Total Weighted Score ($w_i \cdot x_i$)
The project being in a live/production environment (positive)	X_1	W_1	$W_1 \cdot X_1$
A high or increasing number of users (positive)	X_2	W_2	$W_2 \cdot X_2$
Stability and low variability of test scenarios (positive)	X_3	W_3	$W_3 \cdot X_3$
Requirement for the software to run on multiple platforms (positive)	X_4	W_4	$W_4 \cdot X_4$
Frequency of regression testing (positive)	X_5	W_5	$W_5 \cdot X_5$
Sufficient knowledge and experience in the team regarding test automation (positive)	X_6	W_6	$W_6 \cdot X_6$
Need for high-frequency test repetitions (positive)	X_7	W_7	$W_7 \cdot X_7$
Availability of Continuous Integration/Continuous Delivery (CI/CD) infrastructure (positive)	X_8	W_8	$W_8 \cdot X_8$
The project being small-scale with a limited number of modules (negative)	X_9	W_9	$W_9 \cdot X_9$
The project being in an early stage or R&D phase (negative)	X_{10}	W_{10}	$W_{10} \cdot X_{10}$

Weight value (w_i / v_j): Indicates the importance level of the relevant criterion in the decision-making process.

Criterion score (x_i / y_j): Shows the extent to which the relevant criterion is met in the project.

In decision-making processes, especially in multi-criteria evaluations, it is crucial to measure the impact of each criterion on the decision accurately and objectively. In this context, the concept of weight represents the relative importance of the criterion in the decision process. In other words, which criteria are more critical for the project's transition to automation testing are determined and weighted based on expert opinions, experience, and literature support. For example, factors such as "the project being live" or "an increase in the number of users" have a high impact on the efficiency and necessity of automation, so higher weights are assigned to these criteria. Conversely, less influential or indirect criteria are assigned lower weights. The sum of the weight values given for different criteria must be equal to 1. In this way, the automation score resulting from the sum of the multiplications will have a value between 0 and 1, thus normalizing the result.

On the other hand, the criterion score quantitatively indicates the degree to which each criterion is currently met within the project context. This score generally takes a value between 0 and 1; where 0 means the criterion is not met at all, and 1 means it is fully met. For example, if the team has sufficient knowledge of automation, the score for that criterion would be close to 1, while if knowledge is lacking, this value would be lower. Thus, the criterion score objectively reflects the current status and readiness level of the project.

Finally, the net contribution of each criterion to the decision is obtained by multiplying the criterion's weight by its score. This approach not only considers the importance of the criterion but also how well the project aligns with that criterion, providing a more comprehensive and quantitative evaluation. Evaluating all criteria this way and calculating the total score ensures that the decision to transition to automation testing is made objectively, consistently, and based on scientific grounds. This method offers

great benefits to decision-makers by not only indicating which criteria are important but also highlighting areas in the current project that require improvement.

All of this data is combined in the following formula:

$$Automation\ Score = \sum_{i=1}^{N} X_i \times W_i \ - \sum_{j=1}^{M} X_j \times W_j$$

i = Positive criterion
j = Negative criterion
N = Sum of positive criteria
M = Sum of negative criteria

Threshold value: 0.7 and above → Proceeding with automation is considered feasible.
This formula calculates a net decision score by subtracting the weighted impact of negative criteria from the total weighted impact of positive criteria. The resulting score quantitatively indicates the project's suitability for automation.

6.3 Interpretation of the Automation Transition Decision Based on a Threshold Value

Within the scope of the multi-criteria decision-making model developed in this study, the suitability of a software project for transitioning to test automation is quantitatively evaluated using a Total Score, which is calculated by taking the weighted average of positive and negative factors. To enhance the interpretability of this score, a threshold value has been defined, inspired by widely accepted methods in the literature. This threshold has been set at 0.7 (or 70%)**.**

A total score of 0.7 or higher indicates that the project has reached a sufficient level of maturity and suitability for test automation. In such cases, it is assumed that most of the positive criteria have been met, and the benefits of automation such as increased efficiency, speed, and sustainability are likely to offset the investment cost. Therefore,

transitioning to automated testing in these projects is considered a technically and economically rational choice**.**

Conversely, a total score below 0.7 suggests that the project is not yet ready for automation, and that certain fundamental prerequisites have not been fulfilled. Proceeding with automation under such conditions without addressing issues such as team competency, stability of test scenarios, project scale, or technological infrastructure may lead to resource waste or inefficiencies in the process. Hence, it is recommended that deficiencies in these areas be addressed first, followed by a reevaluation to determine whether the project has become suitable for automation.

This approach enables more scientific, measurable, and systematic decision-making regarding test automation in software development projects. Furthermore, it allows for the definition of project-specific threshold values**,** making the model flexible and adaptable across different project contexts.

6.4 NUMERICAL EXAMPLE

In Chapter 6, an evaluation table was prepared using the metrics found throughout the study. This table was evaluated for the web interfaces used throughout the study, and an example was created. Below is an example using numerical values.

Table 6.2: Numerical Example for Automation Transition Evaluation

Criteria	Criterion rating (x_i)	Weight (w_i)	Total Weighted Score ($w_i \cdot x_i$)
The project being in a live/production environment(positive)	1	0,3	0,3
A high or increasing number of users(positive)	1	0,2	0,2
Stability and low variability of test scenarios (positive)	0,5	0,1	0,05
Requirement for the software to run on multiple platforms(positive)	0,8	0,05	0,04
Frequency of regression testing(positive)	0,8	0,1	0,08
Sufficient knowledge and experience in the team regarding test automation(positive)	0,1	0,05	0,005
Need for high-frequency test repetitions(positive)	0,6	0,05	0,03
Availability of Continuous Integration/Continuous Delivery (CI/CD) infrastructure(positive)	0,1	0,05	0,005
The project being small-scale with a limited number of modules (negative)	0,1	0,05	0,005
The project being in an early stage or R&D phase(negative)	0	0,05	0

When the values in the table above are evaluated based on the automation score, the result is 0.705. Since the threshold for transitioning to automation was set at 0.7, and 0.705 > 0.7, it can be concluded that a company with the values shown in the table is eligible for automation.

7 RELATED WORKS

This study compares automation and manual testing methods in software testing. This comparison has been the subject of many academic studies. Manual and automation tests have advantages over each other in different areas (Uslu, Sahin, and Yılmaz 2022).

When the literature is reviewed, one gap is identified: In real life scenarios, should a company transition to automation testing in software testing, or should it continue with manual testing?

The process began by first writing test cases that would be used for both manual and automation testing. The test cases created are common for all testing processes and were not created separately. A total of 33 test cases were created for the login pages of three popular educational platforms. The reason for testing the login pages is that every application includes a login page, which is an indispensable component for any application. Instead of a test case document that only contains the test name and fail/pass status, all relevant sections were included, so that anyone reading the test case can easily repeat the test and understand what was done in the test.

Afterwards, these test cases were executed manually, and the total time spent was calculated. Once the manual execution process and the time and cost calculations were completed, an infrastructure was created to run the same test cases using automation.
There are many software testing automation tools available to accelerate interface testing. In addition to these tools, new automation tools are still being developed. Although Selenium is one of the most popular tools for web interface automation, it remains traditional and cumbersome compared to newer technologies (Antonczak and Bylina 2024).

Instead, the Playwright tool, which provides faster adaptation, has been used. The benefits of this tool are explained in section 3.4.

One of the greatest benefits of automation testing is its ability to integrate with CI/CD tools. This allows tests to be executed without human intervention and time loss. Many different articles have discussed the features provided by Jenkins tools and how test automation works seamlessly with less manual effort (Deshpande, Veenadevi, and Aleti 2021).

Preparing all test cases at once and executing them manually took a total of 1003 man/minutes. The same calculation could not be directly applied for automation testing, as an infrastructure had to be set up for automation tests. The setup of the infrastructure and the execution of the test cases once took a total of 9600 man/minutes.

When looking at the initial costs, manual testing was performed with a profit of 86.55%. However, the cost of automation testing is zero once the initial cost is eliminated.

When examining the table in section 4.3, it is observed that the initial cost of manual software testing is much lower. However, as the same test cases are executed continuously, the initial cost for automation testing is amortized. In this study, it has been proven that for a company deploying and testing daily in different environments, automation amortizes its initial cost within a six-month period, and after six months, every executed test case results in profit.

After the CI/CD integration, another advantage of automation testing, the automatic report generation process, was implemented. Automatic test report generation is a feature provided by the combination of Jenkins integration and automation testing. For this, Allure reporting was used. The reporting process is explained in section 3.6. With reporting, the software testing process is concluded. As shown in Table 4.4, the reporting process started generating profit from the first run.

After all these experiments and proofs, manual and automation testing methods in software testing are compared with different metrics in Section 5.

The transition to automated testing allowed for parallel execution of tests, providing more detailed test results with step by step screenshots, and reducing man hours and execution time. Through test automation, the frequency of running regression tests increased, and human errors were detected earlier. Additionally, test automation provides more detailed test results (such as recorded screenshots) compared to manual testing.

Tests were automatically executed after code changes with continuous integration or at scheduled times. This allows for much faster feedback by only reviewing the failed tests in the test results. It can be determined which deployment code changes caused the failed regression tests.

As a result, automation testing is initially costly. However, as repetitive tests like regression tests are conducted, this cost is amortized over time, and companies start making a profit in the ongoing process. Once a fully automated test system is established, the company will eventually be fully profitable. Additionally, the reason for the high initial cost is the effort required to set up the automation infrastructure in a company that lacks an automation culture (Table 4.1).

When a company starts a new project, if it is expected that the project will not remain as research and development but will instead be a long-term project, the company should create an automation test plan for that project. If the company already has an automation infrastructure, there will be no initial cost when creating an automation plan for new projects. In this case, starting automation testing for the mentioned project will be much easier and more feasible.

However, for some projects, it is uncertain whether they will be sold or go live. For these types of projects, starting automation might be risky. This is because initiating automation testing is costly, and if the project does not go into production and will not require continuous integration or maintenance, it would be unnecessary to incur this initial cost.

8 LIMITATIONS AND FUTURE WORK

At the beginning of the software development process, it was difficult to grasp the complexity of the software, which meant that testing was also quite simple at first. In the early days of software development, applications were generally small and served a limited number of users. Since software was not very complex during this period, simple manual tests were sufficient to check whether the software functioned properly. As software grew larger and more complex, the limitations of such tests began to surface. In large projects in particular, performing manual tests for every component became time-consuming and prone to error. Moreover, when the same tests had to be repeatedly run across various versions of the software, the inefficiency of manual testing became apparent. This led to the widespread adoption of software test automation. Although software test automation became increasingly popular, companies did not entirely abandon manual testing. In other words, automated tests did not eliminate manual testing. Today, a similar concern is being experienced with artificial intelligence.

Artificial intelligence, which has started to be used effectively in every aspect of our lives, is also being applied in the software industry. The development and growing popularity of AI have raised concerns about the shrinking of current job fields and the disappearance of certain professions (Zhu 2023). The integration of artificial intelligence into software engineering has, in recent years, led to a significant period of transformation by remarkably improving the software development cycle in terms of progress and efficiency (Durrani et al. 2024). AI also offers various benefits in the field of software testing. These benefits can be explained as follows.

Test Scenario Generation**:** Artificial intelligence can analyze the functionality of the software and automatically generate test scenarios and data, thereby accelerating the testing processes.

Automated Bug Detection: Artificial intelligence can learn from historical data to detect potential errors in software and make error predictions. This is particularly useful in regression testing. By analyzing past test results, bug reports, and code changes, AI can predict where bugs are likely to occur within the software. In regression testing especially, it can identify which tests are most critical and execute only the necessary ones.

By focusing on areas of the code where bugs have previously occurred, complex structures, or recent changes, AI can conclude that "this part may cause issues." This approach saves time, reduces testing costs, and enables a smarter testing process.
Improvement of Test Automation: Artificial intelligence can write code for automation tests or enhance existing code to make it more efficient.

Interpretation and Reporting of Results: Artificial intelligence can quickly analyze test results and generate reports on errors and success rates, enabling developers to make the necessary improvements. If the AI is provided with previously executed test cases and their outcomes (fail/pass), it can draw conclusions such as: "Test cases that were run when the password was entered incorrectly failed in two previous versions of the software; they may also fail in the newly written code. Therefore, line x of the software code should be reviewed."

User Experience Testing: AI evaluates whether software is user friendly by analyzing user behavior. For example, by processing traffic analysis data from tools like Google Analytics, it can determine whether elements are user friendly and identify which components users struggle to interact with. AI can assess how users engage with a page and measure which designs are more user centric. Additionally, by learning from A/B testing, it can provide decision support regarding the "most effective version." This enables product teams to make more informed UX decisions.

All of these benefits can certainly accelerate the software testing process and enhance the quality of software testing. However, these advantages cannot completely eliminate the need for manual and automated testing methods.

8.1 Areas Where Manual Tests Outperform AI-Based Tests

The integration of artificial intelligence into software testing processes can accelerate testing and automate certain tasks. However, some aspects of software testing still require human intervention and observation. There are several key reasons why manual testing continues to hold critical importance.

Complex and Variable Test Scenarios: Manual testing provides flexibility in analyzing how software behaves under different scenarios and conditions. In particular, testing unexpected behaviors arising from user interactions is not always possible with automation tools. Unlike machines, humans possess cognitive abilities (Enoiu, Tukseferi, and Feldt 2020). Cognition is a scientific term used to describe a set of mental capabilities, including attention, memory, language use and comprehension, learning, evaluation, problem solving, and decision making**.**

Complex User Actions: On an e-commerce website, user interactions such as adding products to the cart, encountering issues during the payment process, or taking advantage of promotions cannot always be accurately tested through automation. Manual testers, by interacting with the software like real users, can observe the software's behavior in real-world scenarios more accurately.

User Experience (UX) Testing: Evaluating factors such as the ease of use, design, and overall functionality of software cannot be measured solely through the capabilities of artificial intelligence. Human test experts can assess whether an application is user-friendly, whether its design is appropriate, and how it will respond to user feedback.
Bug Detection and Contextual Interpretation**:** Manual testing requires a deeper understanding of context when identifying bugs and issues in software. While artificial intelligence can detect certain errors, it often struggles to recognize highly specific or context dependent bugs.

Contextual Interpretation: Although artificial intelligence can sometimes detect software errors, it may struggle to accurately analyze unexpected behaviors that occur during runtime. A manual test expert, however, can examine software issues not only on the surface but also in a deeper and more contextual manner.

New and Unique Bugs: Artificial intelligence may struggle to recognize types of errors that it has not previously encountered. For example, the way software behaves in an unexpected situation may not align with the AI's training data. However, manual testers can more easily identify such bugs by thoroughly examining all aspects of the software**.**

8.2 Areas Where Automation Tests Outperform AI-Based Tests

While artificial intelligence can make testing processes more efficient, there are still certain areas where automation tests remain superior. The advantages offered by automation tests include the following:

Repeatability and Efficiency: Automation tests allow repeated testing of the same parts of the software. This is highly beneficial in software development processes, as it ensures that a specific test scenario can be executed consistently. Artificial intelligence, on the other hand, may not always operate with the same level of efficiency due to its learning process and the need for continuous data updates.

Regression Testing: Automation tools can test the accuracy of changes in a new version of the software compared to previous versions. In contrast, artificial intelligence must be trained to continuously update these tests and analyze them with accurate data, which can be time-consuming.

Data and Time Efficiency: Automation tests can execute a large number of test scenarios quickly. In contrast, artificial intelligence may initially require a significant amount of time and resources to accurately guide the testing processes.

Precision and Predictability: Automation tests provide clear and definite results regarding the functioning of specific software. In other words, it is much more

straightforward to test whether a software application is working correctly. Artificial intelligence, on the other hand, may sometimes make predictions about test results but may not always produce accurate outcomes in every situation.

Consistent Test Results: In automation testing, it is expected that the same test will yield the same results each time. This enhances the consistency of the tests. Artificial intelligence, however, may sometimes make errors or produce varying interpretations in different situations, which can affect the accuracy of the tests.

Variable Parameters: In automation testing, the parameters used are predefined and fixed, which ensures that the results remain consistent each time. In contrast, artificial intelligence may learn to work with different data sets in each test, which can lead to variability in the results.

Management of Large-Scale Tests: In large systems, it is necessary to run numerous tests in parallel and conduct extensive testing. Automation tests are highly effective in such large-scale projects. Artificial intelligence, however, may occasionally struggle to manage these kinds of large test operations.

Scalability: Automation tests can execute parallel tests very quickly in large systems. Artificial intelligence, on the other hand, may face some challenges in accurately analyzing and managing large data sets when working at scale.

8.3 Advantages and Limitations of AI-Supported Tests

Artificial intelligence can be used in many different areas to support software testing; however, it is unlikely to fully replace traditional software testing processes. AI-supported tests can be beneficial in certain areas, but they also come with some limitations:

Benefits of AI-Supported Tests are listed below;
Error Prediction and Data Analysis: Artificial intelligence can predict which areas of the software require further testing by learning from past tests and previous errors.

Additionally, it can offer suggestions regarding potential bugs that may arise in the software.

Data Generation: Automated test data generation can significantly enhance the efficiency and accuracy of the testing process (Korel, 2025). Creating test data is often a manually demanding task. Artificial intelligence can generate test data for specific scenarios and quickly examine how the software behaves under various conditions.

Limitations of AI-Based Tests are listed below;
Lack of Contextual Understanding: Artificial intelligence may not always accurately comprehend the context of tests. This can hinder the effective testing of the software. Human test experts, by considering the functionality and design of the software, are able to create more effective testing scenarios.
Lack of Creativity: Human test experts can explore various usage scenarios of the software and create new test cases. Artificial intelligence, on the other hand, is generally limited to past data and may struggle to generate novel scenarios.

In conclusion, both manual testing and automated testing hold significant value in the software testing process, and artificial intelligence does not replace these processes. AI can help accelerate certain tests and make them more efficient; however, in some critical testing phases, the human factor and the precision and consistency provided by automated tests are still essential. Therefore, combining both approaches in software testing processes yields the most effective results.

9 CONCLUSION

In this study, manual and automated testing methods were evaluated based on several critical metrics. The findings indicated that neither method is universally superior; rather, each approach exhibits distinct advantages and disadvantages, depending on specific project characteristics and requirements. A categorical recommendation advocating exclusively for either manual or automated testing would therefore be inaccurate.

For long-term, continuously evolving projects requiring frequent regression testing such as enterprise applications, banking systems, e-commerce platforms, and defense software the advantages of automation testing are clearly demonstrated. Automated testing accelerates test execution cycles, reduces repetitive tasks, and significantly lowers the cumulative costs and efforts associated with ongoing regression testing. Consequently, companies engaged in sustained software development projects should strongly consider adopting automation testing as part of their standard processes.

Conversely, for short-term, experimental, or uncertain duration projects such as proof of concept studies, research driven initiatives, hackathons, or projects with unclear future development manual testing remains more practical. Given the high initial setup time, cost, and training investments required for automation, these projects generally do not benefit from automation, as repetitive regression tests may be infrequent or unnecessary. In such contexts, manual testing remains cost-effective and straightforward. An essential insight derived from this research is that the selection between manual and automated testing methods must be strategically determined at the outset of each project, considering factors such as project duration, complexity, budget constraints, team skillsets, and expected frequency of regression testing.

Further research may investigate automation testing scalability with significantly larger test suites, assessing performance degradation, infrastructure requirements, and

potential efficiency bottlenecks. With advances in AI and machine learning, future research should explore integrating AI-based test case generation and execution methods to improve automation efficiency, coverage, and effectiveness.

REFERENCES

Akin, A., Sentürk, S., & Garousi, V. (2018). "Transitioning from Manual to Automated Software Regression Testing: Experience from the Banking Domain," 2018 25th Asia-Pacific Software Engineering Conference (APSEC), Nara, Japan, 2018, pp. 591–597, doi: 10.1109/APSEC.2018.00074.

Antonczak, A., & Bylina, B. (2024). "Analysis of end-to-end test automation tools based on the examples of Selenium WebDriver and Playwright," Position Papers of the 19th Conference on Computer Science and Intelligence Systems (FedCSIS), Lublin, Poland.

Badihi, S., Ahmed, K., Li, Y., & Rubin, J. (2023). "Responsibility in Context: On Applicability of Slicing in Semantic Regression Analysis," 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), Melbourne, Australia, pp. 563–575, doi: 10.1109/ICSE48619.2023.00057.

da Silva, G., & de Souza Santos, R. (2023). "Comparing Mobile Testing Tools Using Documentary Analysis," 2023 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), New Orleans, LA, USA, pp. 1–6, doi: 10.1109/ESEM56168.2023.10304798.

Deshpande, A., Veenadevi, S. V., & Aleti, S. (2021). "Test Automation and Continuous Integration using Jenkins for Smart Card OS," 2021 12th International Conference on Computing Communication and Networking Technologies (ICCCNT), Kharagpur, India, pp. 01–05, doi: 10.1109/ICCCNT51525.2021.9580021.

Dobles, I., Martínez, A., & Quesada-López, C. (2019). "Comparing the effort and effectiveness of automated and manual tests," 2019 14th Iberian Conference on Information Systems and Technologies (CISTI), Coimbra, Portugal, pp. 1–6, doi: 10.23919/CISTI.2019.8760848.

Durrani, U. K., Akpinar, M., Adak, M. F., Kabakus, A. T., Öztürk, M. M., & Saleh, M. (2013–2023). "A Decade of Progress: A Systematic Literature Review on the

Integration of AI in Software Engineering Phases and Activities (2013–2023)," IEEE Access, vol. 12, pp. 171185–171204, 2024, doi: 10.1109/ACCESS.2024.3488904.

Enoiu, E., Tukseferi, G., & Feldt, R. (2020). "Towards a Model of Testers' Cognitive Processes: Software Testing as a Problem Solving Approach," 2020 IEEE 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C), Macau, China, pp. 272–279, doi: 10.1109/QRS-C51114.2020.00053.

Halani, K. R., Kavita, & Saxena, R. (2021). "Critical Analysis of Manual Versus Automation Testing," 2021 International Conference on Computational Performance Evaluation (ComPE), Shillong, India, pp. 132–135, doi: 10.1109/ComPE53109.2021.9752388.

Klammer, C., & Ramler, R. (2017). "A Journey from Manual Testing to Automated Test Generation in an Industry Project," 2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C), Prague, Czech Republic, pp. 591–592, doi: 10.1109/QRS-C.2017.108.

Korel, B. (2025). "Influence of the 1990 IEEE TSE Paper “Automated Software Test Data Generation” on Software Engineering," IEEE Transactions on Software Engineering, vol. 51, no. 3, pp. 751–753, March 2025, doi: 10.1109/TSE.2025.3540430.

Li, Z., Wang, W., Wang, S., Liang, P., & Mo, R. (2023). "Understanding Resolution of Multi-Language Bugs: An Empirical Study on Apache Projects," 2023 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), New Orleans, LA, USA, pp. 1–11, doi: 10.1109/ESEM56168.2023.10304793.

Mori, A. (2020). "Anomaly Analyses to Guide Software Testing Activity," 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST), Porto, Portugal, pp. 427–429, doi: 10.1109/ICST46399.2020.00055.

Mustafa, K. M., Al-Qutaish, R. E., & Muhairat, M. I. (2009). "Classification of Software Testing Tools Based on the Software Testing Methods," 2009 Second International Conference on Computer and Electrical Engineering, Dubai, United Arab Emirates, pp. 229–233, doi: 10.1109/ICCEE.2009.9.

Nagy, S. M., Maghawry, H. A., & Badr, N. L. (2022). "An Enhanced Parallel Automation Testing Architecture for Test Case Execution," 2022 5th International

Conference on Computing and Informatics (ICCI), New Cairo, Cairo, Egypt, pp. 369–373, doi: 10.1109/ICCI54321.2022.9756109.

Rogers, G. P., & Miles, P. (2015). "Manual testing's newfound place in the automated testing world," 2015 IEEE AUTOTESTCON, National Harbor, MD, USA, pp. 199–202, doi: 10.1109/AUTEST.2015.7356489.

Sneha, K., & Malle, G. M. (2017). "Research on software testing techniques and software automation testing tools," 2017 International Conference on Energy, Communication, Data Analytics and Soft Computing (ICECDS), Chennai, India, pp. 77–81, doi: 10.1109/ICECDS.2017.8389562.

Uslu, H., Sahin, C., & Yılmaz, M. (2022). "A Preliminary Survey on Software Testing Practices in Defense Industry," 16. Ulusal Yazılım Mühendisliği Sempozyumu Bildiri Kitabı, Ankara, Turkey.

Wang, S., Lian, X., Marinov, D., & Xu, T. (2023). "Test Selection for Unified Regression Testing," 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), Melbourne, Australia, pp. 1687–1699, doi: 10.1109/ICSE48619.2023.00145.

Yalamanchili, S., & Kumari, K. S. (2016). "Comparison of manual and automatic testing using genetic algorithm for information handling system," 2016 International Conference on Signal Processing, Communication, Power and Embedded System (SCOPES), Paralakhemundi, India, pp. 1795–1799, doi: 10.1109/SCOPES.2016.7955752.

Zhu, L. (2023). "Software Engineering as the Linchpin of Responsible AI," 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), Melbourne, Australia, pp. 3–4, doi: 10.1109/ICSE48619.2023.00012.