



# **Technical University of Brandenburg**

Department of Business Informatics

## **[Return on Investment for Automation software testing in End-to-End System “Case Study”]**

### **Master thesis**

to obtain the academic degree

### **Master of Science**

submitted by:

**Kareem Elhosseney**

**Supervisor:** Prof.Dr.Olga Levina, TU of Brandenburg.

**Supervisor:** Eng.Olga Barbu,Head of Testing,Endava GmBh.

Brandenburg an der Havel, 06.08.2023

## Abstracts

This Master's thesis explores how automated testing can improve the quality of web applications. It's common to test software manually, which includes tasks like planning, writing reports, and analyzing results. However, this can be a slow and costly process. This study examines if automated testing, a quicker and potentially less expensive method, can provide a good return on investment (ROI) compared to manual testing.

Software testing is crucial for web applications, and it's a challenging task for quality assurance (QA) engineers and software developers. By comparing automated and manual testing, we can see the benefits and drawbacks of each and determine when it's best to use one or the other. This comparison will consider factors like time and money to calculate ROI.

For automated testing, we chose to use WebDriverIO. It's a free tool that can imitate real users, and it's flexible, supports multiple languages, and can be used across different platforms. The study will show how WebDriverIO differs from manual testing and will calculate the ROI for each method.

Additionally, this research describes how to set up an automated testing process using WebDriverIO. These tools and services can help maintain high-quality testing and support the ideas of continuous development and integration. In simpler terms, they allow for continuous updates and improvements in the software development process. The findings from this study can help QA engineers, software developers, and other decision-makers choose the best testing method for their needs.

## **Keywords**

Test Automation, WebDriverIO, Test Automation Framework, functional testing, End-to-end Testing, DevOps, Continuous Integration, Continuous Delivery, Gitlab pipelines, Repository,ROI.

## **Declaration of authorship**

I hereby declare that I have written this thesis independently and that I have not used any auxiliary materials other than those indicated. The passages of the work, which were taken from other sources in the word or the sense, are marked by indication of the origin. This also applies to drawings, sketches, pictorial representations as well as to sources from the Internet.

Brandenburg an der Havel, 06.08.2023

A handwritten signature in blue ink, appearing to read 'Kareem Elhosseny', is written above a horizontal dashed line.

Kareem Elhosseny

## Acknowledgements

This work, as much as it is a product of countless hours of relentless dedication and exhaustive scholarly pursuits, is equally an embodiment of gratitude and affection for those who stood beside me on this challenging yet fulfilling journey. As I present this thesis, it is with profound humility and heartfelt appreciation that I acknowledge the individuals who have not only facilitated this study but also shaped me into the person I am today.

My deepest appreciation extends to Prof. Dr. Olga Levina, who has served as an invaluable guide in this journey. Her profound knowledge and ingenious teaching methods have not only opened up new realms of understanding but also fundamentally transformed the way I approach academia. Her patience and flexibility throughout this process have played a pivotal role in making the challenges surmountable, and for this, I am eternally grateful.

Furthermore, I wish to express my heartfelt gratitude to Eng. Olga Barbu. Her unwavering belief in my capabilities has been a cornerstone of this project. She continually inspired me to aim high and never settled for less. Her confidence in my abilities fueled my motivation and drove me to push my boundaries, to learn, and to improve continually.

I owe the deepest gratitude to my beloved father, Mohamed Elhosseney. You have been the beacon guiding me through the stormy seas of my academic journey. Your unwavering faith in me, your ceaseless encouragement, and your silent sacrifices have been my rock, providing a firm foundation on which I built my dreams. Equally, I express my warmest gratitude to my loving mother, Nadia Abdelmoaty. Your nurturing love and tireless support have always been a source of strength for me. Your belief in me, even when I doubted myself, served as a constant reminder of my potential. You have been the quintessential symbol of resilience, embodying the strength that I have strived to mirror in this research.

To my extraordinary wife, Hayat Rashwany, words seem to fail in capturing the depth of my appreciation. Your unwavering love and enduring patience, your unfailing understanding, and your unending support in the face of all obstacles, have been a haven in the storm. Your companionship has enriched this journey immeasurably, making each stride lighter and each accomplishment sweeter. For my little sunshine, my daughter Khadega Elhosseney, you brought joy to even the most arduous days. Your innocent laughter and boundless curiosity kept me inspired, reminding me of the power of simple wonders and the joy of lifelong learning. This work is also a testament to my dreams for you, my little one.

In conclusion, I remain forever indebted to each of you, as this accomplishment would not have been possible without your unwavering support and unflagging faith. This master's thesis, therefore, is not just a testament to my scholarly journey, but a tribute to the love, guidance, and strength you have bestowed upon me.

## Table of content

<b>1</b>	<b><u>INTRODUCTION .....</u></b>	<b><u>1</u></b>
1.1	MOTIVATION .....	1
1.2	RESEARCH QUESTIONS .....	2
1.3	GOAL .....	2
1.4	OUTLINE OF THESIS .....	3
<b>2</b>	<b><u>SOFTWARE TESTING THEORETICAL BASICS .....</u></b>	<b><u>5</u></b>
2.1	MANUAL TESTING.....	6
2.1.1	TYPES OF SOFTWARE TESTING .....	7
2.1.2	USER STORIES, TEST CASES, TEST REPORTS .....	10
2.2	IMPORTANT COMMON TECHNIQUES IN SOFTWARE TESTING (AUTOMATION AND MANUAL) .....	13
2.2.1	EXPLORATORY TEST .....	13
2.2.2	RISK-BASED TESTING.....	14
2.2.3	REGRESSION TESTING.....	18
2.2.4	SMOKE TESTING.....	19
2.3	AUTOMATION TESTING .....	22
2.3.1	TEST AUTOMATION OBJECTIVES AND LIMITATIONS .....	24
2.3.2	MANUAL TESTING VS AUTOMATION TESTING .....	25
<b>3</b>	<b><u>TEST AUTOMATION FRAMEWORK AND TOOLS .....</u></b>	<b><u>27</u></b>
3.1	WEBDRIVERIO.....	27
3.1.1	WEBDRIVERIO ARCHITECTURE .....	29
3.2	BEHAVIOR DRIVEN DEVELOPMENT & CUCUMBER FRAMEWORK .....	30
3.3	PAGE OBJECT MODEL.....	33
3.4	TYPESCRIPT .....	34
3.5	ALLURE REPORT .....	35
3.6	CONTINUES INTEGRATION / CONTINUES DELIVERY CI/CD .....	37
3.6.1	CI/CD FUNDAMENTALS .....	37
3.6.2	CONTINUES INTEGRATION CI.....	39
3.6.3	CONTINUES DELIVERY CD .....	41
<b>4</b>	<b><u>RETURN ON INVESTMENT (ROI) OF AUTOMATION TESTING .....</u></b>	<b><u>43</u></b>
4.1	WHAT IS ROI IN AUTOMATION TESTING? .....	43
4.2	CALCULATING ROI .....	45
4.3	ENHANCE ROI IN TEST AUTOMATION.....	47
<b>5</b>	<b><u>EVALUATION BASED ON CASE STUDY .....</u></b>	<b><u>49</u></b>
5.1	MANUAL TESTING EVALUATION .....	50
5.2	AUTOMATION TESTING EVALUATION .....	52
5.3	ROI EVALUATION .....	54
5.3.1	THE REAL CASE .....	55

---

5.3.2	CHANGING VARIABLES .....	57
<b>6</b>	<b>CONCLUSION.....</b>	<b>59</b>
<b>7</b>	<b>AI TOOLS PROTOCOL: .....</b>	<b>62</b>
<b>7.1</b>	<b>CHATGPT - GPT-4 BY OPEN AI .....</b>	<b>62</b>
<b>7.2</b>	<b>BARD BY GOOGLE .....</b>	<b>63</b>
<b>7.3</b>	<b>WORKFLOW INTEGRATION OF AI TOOLS: .....</b>	<b>63</b>
<b>7.4</b>	<b>ILLUSTRATIVE EXAMPLES: .....</b>	<b>64</b>
<b>BIBLIOGRAPHY .....</b>		<b>ERROR! BOOKMARK NOT DEFINED.</b>
<b>LITERATURE.....</b>		<b>67</b>
<b>APPENDIX.....</b>		<b>70</b>

## List of Tables

TABLE 1: REGRESSION TEST VS RETESTING (KITAKABEE, 2023) .....	19
TABLE 2: ADVANTAGES IN MANUAL TESTING & AUTOMATION TESTING.....	26
TABLE 3: DISADVANTAGES IN MANUAL TESTING & AUTOMATION TESTING .....	26
TABLE 4 : PROCESS TIMING FOR MANUAL TESTING .....	51
TABLE 5: PROCESS TIMING FOR AUTOMATION TESTING .....	53
TABLE 6 : RECORD OF AUTOMATION TESTCASE IN MAP OF FUNCTIONALITY FILE.....	53
TABLE 7: REAL CASE VALUES TO CALCULATE ROI.....	55
TABLE 8: ROI COMPARISON .....	56
TABLE 9 : ROI COMPARISON WHEN FIXING TIME 20 MIN .....	57



## List of Figures

FIGURE 1: LIKELIHOOD, IMPACT OF FAILURE MATRIX, SOURCE: HTTPS://WWW.SOFTWARETESTINGHELP.COM/RISK-MANAGEMENT-DURING-TEST-PLANNING- RISK-BASED-TESTING/ .....	16
FIGURE 2: TESTING PRIORITY GRID, SOURCE HTTPS://WWW.SOFTWARETESTINGHELP.COM/RISK- MANAGEMENT-DURING-TEST-PLANNING-RISK-BASED-TESTING/ .....	17
FIGURE 3: LEVEL OF DETAIL OF TESTING, SOURCE: HTTPS://WWW.SOFTWARETESTINGHELP.COM/RISK- MANAGEMENT-DURING-TEST-PLANNING-RISK-BASED-TESTING/ .....	17
FIGURE 4: SMOKE TESTING CYCLE SOURCE: HTTPS://WWW.GURU99.COM/SMOKE-TESTING.HTML .....	20
FIGURE 5: THE GROWING POPULARITY OF WEBDRIVERIO SOURCE: WWW.NPMTRENDS.COM.....	29
FIGURE 6:WEBDRIVERIO ARCHITECTURE SOURCE:HTTPS://WWW.BROWSERSTACK.COM/GUIDE/WEBDRIVERIO-TUTORIAL-FOR-SELENIUM- AUTOMATION .....	30
FIGURE 7: CUCUMBER CODE SNIPPET.....	33
FIGURE 8: POM STRUCTURE, SOURCE: HTTPS://WWW.GURU99.COM/PAGE-OBJECT-MODEL-POM-PAGE- FACTORY-IN-SELENIUM-ULTIMATE-GUIDE.HTML .....	33
FIGURE 9: SCREENSHOT FROM ALLURE REPORT.....	37
FIGURE 10: CICD SOURCE: (SAMBAMURTHY, 2023).....	38
FIGURE 11: CI SYSTEMS SOURCE: (SAMBAMURTHY, 2023) .....	40
FIGURE 12 : DEPLOYMENT PIPELINE SOURCE: (SAMBAMURTHY, 2023).....	41
FIGURE 13: ROI VS RELEASES OR NUMBER OF TEST RUN .....	46
FIGURE 14: BPMN DIAGRAM FOR MANUAL TESTING PROCESS.....	50
FIGURE 15: RECORD OF TESTCASE IN MAP OF FUNCTIONALITY FILE.....	51
FIGURE 16: BPMN DIAGRAM FOR AUTOMATION TESTING PROCESS .....	52
FIGURE 17: ROI VS NUMBER OF RUN "REAL VALUES" .....	56
FIGURE 18: ROI VS NUMBER OF RUN "FIXING TIME 20 MIN" .....	58
FIGURE 19 : PROMPT TO CHATGPT EX1 .....	64
FIGURE 20: CHATGPT ANSWER EX.1 .....	65
FIGURE 21: PROMPT TO CHATGPT EX.2 .....	65
FIGURE 22 : CHATGPT ANSWER EX.2.....	66

## Glossary

- **E2E:** End-to-end
- **QA:** Quality assurance
- **UI:** User interface
- **IDE:** Integrated Development Environment
- **API:** Application Programming Interface
- **DOM:** Document Object Model
- **POM:** Page object model
- **JSON:** JavaScript object notation
- **HTTP:** HyperText Transfer Protocol
- **DevOps:** Development and Operation
- **CI:** Continuous Integration
- **CD:** Continuous Deployment
- **AI:** Artificial Intelligence
- **RBT:** Risk Based Testing
- **SDLC:** Software Development Life Cycle
- **SUT:** System under Test



# 1 Introduction

## 1.1 Motivation

In the rapidly evolving landscape of software development, high-quality assurance of web applications has become a significant challenge. Traditional methods of manual testing, while effective, can be slow and expensive. The introduction of automated testing brings a potential solution to this problem, promising efficiency, and cost-effectiveness. However, there is a critical need for a comprehensive understanding of the advantages and drawbacks of both manual and automated testing methods, and the circumstances under which each method excels.

This motivation led to the inception of this Study. Its purpose is not just to examine the potential of automated testing as a quicker and less expensive method, but also to measure its Return on Investment (ROI) in comparison to manual testing. Calculating the ROI will provide a clearer picture of the economic benefits, accounting for factors like time and money and other factors, which are crucial for decision-making in any organization.

Furthermore, understanding how to effectively implement an automated testing process and how it contributes to continuous software development and integration is crucial. The use of WebDriverIO, Selenium tools, and DevOps services like GitLab in automating the testing process not only streamlines operations but also enhances the quality of testing.

The hope is that this study will not only guide QA engineers, software developers, and decision-makers in selecting the most effective testing method but also contribute to the broader discourse on software testing and quality assurance. The findings could potentially reshape testing strategies, promoting a more efficient and economically sound software development lifecycle.

## 1.2 Research Questions

The research is guided by the following pivotal questions:

1. How has the Automation Testing framework evolved from both theoretical and practical perspectives?
2. How is the ROI for Automation Testing calculated?
3. What are the key factors for calculating ROI?

By answering these questions, this thesis aims to provide a comprehensive understanding of ROI in the context of software test automation, its importance in contemporary web application testing, and its interplay with CI/CD pipelines. The study will contribute to advancing the scientific understanding and industry practices in the field of software testing, contributing to the overall enhancement of software quality.

## 1.3 Goal

The aim of this Study is to conduct a thorough comparison between manual and automated software testing methods. We plan to investigate the advantages, drawbacks, and appropriate use-cases for each method, with a focus on determining the Return on Investment (ROI) of automated testing.

For this, we'll be using a specific technology stack, including WebDriverIO, TypeScript, Cucumber, and Allure report. These tools, chosen for their popularity and efficiency, will help to provide insights into the automated testing process.

Furthermore, we aim to distinguish between various techniques and methodologies used in both manual and automated testing. This will allow us to calculate the ROI more accurately by considering the impact of different techniques on the time and cost involved in testing.

Finally, the study seeks to understand the effects of automated testing on the business software industry. This understanding can help in making more informed decisions about testing strategies, potentially leading to more efficient practices and significant cost savings. Ultimately, our goal is to provide guidance to quality assurance engineers,

software developers, and decision-makers in the field to enhance software quality and the overall efficiency of the development process.

## 1.4 Outline of Thesis

This Study is divided into eight chapters. Here's a brief overview of each:

1. **Introduction:** This chapter explains why the research was conducted, the main questions it aims to answer, and the overall goal of the study.
2. **Theoretical Basics:** This chapter explores the fundamentals of software testing and test automation. It clarifies key concepts and definitions, discusses the differences between manual and automated testing, and highlights the advantages and challenges of automated testing.
3. **Framework Components and Tools:** In this chapter, we introduce all the software tools, design patterns, and the framework stack used in the study. It also provides insights into software DevOps and CI/CD (Continuous Integration/Continuous Deployment) pipelines.
4. **Return on Investment (ROI):** This chapter introduces ROI and discusses the significant factors required to calculate it. It explains how ROI is measured in the context of test automation and offers suggestions on improving ROI.
5. **Case Study Evaluation:** This chapter examines a real-life case study to compare the actual time and cost involved in both manual and automated testing. It calculates and presents the ROI for both methods based on simulations and equations.
6. **Implementation of Automation Testing:** This chapter delves into the practical aspect of automation testing. It includes code snippets and hands-on examples using the test automation framework.

7. **Conclusion:** The final chapter summarizes the findings of the study and discusses its future implications and perspectives.

Each chapter of this thesis builds on the previous ones, presenting a step-by-step exploration of manual and automated testing, their ROI, and the impact of automated testing on the business software industry.

## 2 Software Testing Theoretical Basics

The pervasiveness of software systems across various sectors, ranging from business applications such as banking to consumer products like automobiles, is unquestionable. Consequently, the impact of malfunctioning software can be detrimental, causing not only financial losses and reputational damage but also potentially leading to personal harm or fatality. Thus, software testing emerges as a critical process, aimed at evaluating software quality and minimizing the likelihood of operational failures. Moreover, software testing is a complicated process and expensive and requires a highly qualified person with technical, analytical skills.

**Objectives of Software Testing:** For a given project, the aims of software testing may encompass the following:

- Evaluating work components, such as requirements, user stories, design, and code, to preempt defects.
- Verifying the fulfillment of all specified requirements.
- Checking the test object's completeness and validating its expected operational efficacy.
- Enhancing confidence in the test object's quality level.
- Identifying and reducing the risk of defects and failures, thus mitigating the threat of subpar software quality.
- Providing stakeholders with adequate information to make informed decisions regarding the test object's quality.
- Complying with, and verifying the test object's compliance with, contractual, legal, and regulatory requirements, or standards.

Software testing can substantially mitigate operational failure risks. The detection and correction of defects contribute significantly to the overall quality of the components or systems. Furthermore, software testing may also be necessary to fulfill contractual, legal, and industry-specific standard requirements.

(ISTQB, 1 July 2021)



## 2.1 Manual Testing

Manual Testing is a primitive yet fundamental approach in software testing where test cases are manually executed by a tester. This approach is taken without the use of any automated tools, emphasizing the importance of human intervention in identifying and rectifying software issues. The principal aim of Manual Testing is to discover error<sup>1</sup>, defects<sup>2</sup>, failures<sup>3</sup> and other problems within the software application, thereby ensuring that the final product is of the highest quality and operates as per the specified functional requirements.

Manual Testing is often more labor-intensive, requiring substantial effort and time, but it offers a guarantee of delivering a virtually bug-free software product. Furthermore, manual testing does not necessitate knowledge of any automated testing tools, making it a more accessible testing method.

A cardinal rule in software testing asserts that "100% automation is not possible," hence, reinforcing the importance of Manual Testing. This rule maintains that it is impossible to automate every aspect of software testing and some aspects necessitate human insight and intervention for the most effective results.

Manual Testing allows testers to execute test cases from an end user's perspective, ensuring that the software application functions as outlined in the requirement document. These test cases are meticulously planned to cover nearly the entirety of the software application. This method allows for both visible and hidden defects within the software to be identified. These defects are defined by the discrepancy between the expected and the actual output provided by the software. Once identified, these defects are fixed by developers and returned to the tester for further retesting.

---

<sup>1</sup> error refers to a mistake made by a human. For example, a developer might make a coding error that leads to incorrect program behavior. In the context of software testing, an error could also mean a mistake in specifying or implementing the software requirements. (ISTQB, 1 July 2021)

<sup>2</sup> A defect or Bug, also often referred to as a bug, is a problem in the system that causes it to behave in undesired ways. In other words, a defect is the result of an error. It's an issue residing in the code that causes the software to malfunction. (ISTQB, 1 July 2021)

<sup>3</sup> A failure is the inability of a system or component to perform its required functions within specified performance requirements. This generally occurs when a defect is executed. In simpler terms, when the bug is evident during the operation of the system, that event is considered a failure. (ISTQB, 1 July 2021)

Creating and implementing Test Suites or Test cases are a critical component of the testing phase, designed to achieve 100% test coverage. This process ensures that all reported defects are rectified by developers, with re-testing subsequently carried out by testers.

Given its role in delivering a bug-free product, manual testing remains a vital part of user-friendly software development. Testers require patience, creativity, and an open mind to effectively think and act from an End User perspective. After all, both the testers and end-users are humans, and the ultimate goal is to develop software that resonates with the user's expectations and experiences. Hence, manual testing, despite its labor-intensive nature, remains an indispensable part of the software development process. (Hamilton, 2023)

### 2.1.1 Types of software testing

**A- Functional testing:** is a crucial process in the software testing cycle aimed at verifying the system's functions or features. These functions, often documented in business requirements, user stories, or functional specifications, define what the system is supposed to do. However, sometimes they may remain undocumented.

The procedure should be performed at every testing level<sup>4</sup>. The primary focus varies at each level, but remains on the system's functional behavior.

---

<sup>4</sup> Test levels represent a structured grouping of test activities managed collectively. Each test level embodies an application of the test process, and is executed in correlation to the software's stage of development. These stages range from individual units or components to entire systems or, where applicable, interconnected systems. Test levels are intertwined with other activities within the software development lifecycle.

The test levels include:

**Component Testing:** This level of testing focuses on individual software components (often called units) to verify their functionality. It's typically the first level of testing, performed by developers, and often automated.

**Integration Testing:** This testing level concentrates on the interaction between software components. It ensures that the components function correctly when integrated together, helping to identify any issues in the interactions between different units of the software.

Functional testing largely employs black-box<sup>5</sup> techniques to derive test conditions and cases for the system or component's functionality. This approach focuses on the software's behavior, regardless of its internal structures or workings.

An essential aspect of functional testing is functional coverage, which measures the extent to which the system's functionality has been tested. It is usually expressed as a percentage of the elements covered by the tests. By tracing the correlation between tests and functional requirements, the percentage of requirements addressed by the tests can be calculated. This can also highlight potential gaps in coverage.

Lastly, functional testing can require specialized skills or knowledge, such as understanding the specific business problem the software aims to solve. This might include industry-specific knowledge, for example, geological modeling software within the oil and gas sectors. (ISTQB, 1 July 2021)

Primarily, automated testing in End-to-End (E2E) scenarios focuses on "Functional Testing" to evaluate the software's functionality.

---

**System Testing:** At this level, the entire system is tested to ensure it meets the specified requirements. System testing is a comprehensive level that checks both the functional and non-functional aspects of the software.

**Acceptance Testing:** The final level of testing, acceptance testing, aims to confirm the system's readiness for delivery. It's often performed by clients or end-users and can take place in real-world environments to ensure the software meets the intended user needs and expectations. (ISTQB, 1 July 2021)

<sup>5</sup> Black box testing is a specification-oriented testing procedure. In this approach, test cases are solely derived from the specifications of the software component or system under test. The internal structure of the software, including its architecture and code, is not considered, hence the term "black box". The focus is on observing the externally visible behavior of the system.

This methodology can be applied at any testing level. However, for component and integration tests, a test environment, often referred to as a test bed, equipped with stubs and drivers is needed. This setup ensures there's an executable test object available.

In black box testing, the software's behavior is monitored from an external standpoint. The Point of Observation (PoO) is outside the test object, meaning that the software's performance is observed without interacting with its internal workings.

The control over the software's flow is only achievable through the selection of suitable input test data. The Point of Control (PoC), similar to the PoO, is also located outside the test object. This means that the tester does not interact directly with the system's internal processes, but rather controls the testing process via inputs and analyzes the corresponding outputs. (Wohnert, 2021)

**B-Non-Functional Testing:** is a vital aspect of software testing that focuses on the non-functional elements of a software application, such as performance, usability, and reliability. Unlike functional testing, which assesses specific software functions, non-functional testing is designed to ensure that the system is ready to handle real-world non-functional parameters.

For instance, one practical application of non-functional testing could be determining how many people can simultaneously log into a piece of software. The objectives of non-functional testing are to enhance the product's usability, efficiency, maintainability, and portability, and to minimize the risk and costs associated with its non-functional aspects.

Furthermore, this type of testing aims to optimize the way the product is installed, set up, executed, managed, and monitored. It aids in gathering and producing measurements and metrics for internal research and development, improving the understanding of the product's behavior and the technologies in use.

Various types of non-functional testing include:

- **Performance Testing:** Evaluates the speed, responsiveness, and stability under a workload.
- **Security Testing:** Assesses the system's ability to prevent unauthorized access and data breaches.
- **Load Testing:** Checks the system's ability to handle the expected volume of user traffic.
- **Compatibility Testing:** Ensures that the software can run in different environments, hardware, operating systems, and with other applications.
- **Usability Testing:** Checks the software's user-friendliness.
- **Stress Testing:** Tests the system's behavior under peak loads and extreme conditions.

(Hamilton, 2023)

### 2.1.2 User stories, Test Cases, Test reports

**A- User stories** are essential elements in Agile<sup>6</sup> software development, serving as simplified, non-technical explanations of a software feature, formulated from the user's perspective. Their aim is to illustrate the value a feature will bring to the customer.

Contrary to popular belief, user stories are not equivalent to software system requirements. Instead, they form a pivotal part of the human-centered approach in agile methodologies, placing the end-users at the heart of the developmental discourse. By presenting the development objectives in accessible language, user stories provide the development team with a clear understanding of the project's purpose, what they need to construct, and the value it will yield.

As foundational elements of an agile program, user stories promote a user-centric outlook in day-to-day tasks. This orientation fosters teamwork, ingenuity, and ultimately enhances the final product.

Defining Agile User Stories: In the context of agile, a user story is the smallest work unit. Rather than a feature, it represents an end goal, voiced from the software user's viewpoint.

User stories convey the anticipated value a specific task will offer the customer, which can include internal customers or colleagues within an organization, in addition to traditional external end users. These stories are brief, straightforward sentences that underscore the expected outcome without delving into details. Further requirements are established later, upon the team's consensus.

For both developers and Quality Assurance (QA) Engineers, user stories offer an initial reference point to understand what needs to be developed or tested, respectively.

Often, user stories follow this structure:

**"As a [persona], I [want to], [so that]."**

(Rehkopf)

---

<sup>6</sup> Agile methodology represents a unique approach to project management characterized by its segmental organization of projects and its focus on continuous enhancement and collaboration. This methodology guides teams through a recurring sequence of planning, execution, and evaluation.

Agile methodology is a strategy of project management that organizes projects into distinct stages, promoting a culture of continuous improvement and collaboration. Following this approach, teams engage in a cyclical process, incorporating phases of planning, execution, and evaluation. This iterative cycle fosters a fluid and adaptable project progression, enabling teams to respond efficiently to changes and continually refine the project as it unfolds.

**B- Test case** is a set of conditions used to authenticate a particular feature or function of a software application. It provides a detailed roadmap including procedures, test data, prerequisites, and postconditions for assessing the functionality of a software application.

The Software Testing Life Cycle (STLC) aims to detect and report any flaws or issues in software applications as early as possible. An integral phase of this cycle is the development of test cases, which involves crafting precise cases to facilitate testing.

A test case, in essence, is a fundamental element of the STLC. It strengthens the reliability and functionality of a software application. It can be perceived as an exhaustive outline of the required procedures and expected outcomes for specific test scenarios.

The composition of efficient test cases is crucial for confirming the software application's functionality and adherence to the Software Requirement Specification (SRS). However, crafting effective test cases necessitates a comprehensive understanding of the software application's function. It also requires the ability to think critically and creatively to pinpoint potential flaws in software applications.

A test case can be defined as a pre-established set of conditions or tasks. The primary objective of a test case is to ensure the functionality of a software program or system. It comprises a thorough account of the inputs, prerequisites, procedures to be performed, and anticipated outcomes. Test cases are devised with meticulous attention to target particular aspects or functionalities of the software, ensuring they comply with the pre-set requirements and fulfill the intended objectives.

The fundamental purpose of executing test cases is to locate any defects or issues within the application, facilitating their rectification prior to the software's release to end users.

For a comprehensive understanding of test cases, consider the following key points:

When creating and executing manual test cases: testers follow specified steps to confirm the application's functionality..

Automated Test Case Execution with Test Tools: Automated test cases are performed using testing tools and frameworks, adhering to the Software Requirement Specification (SRS). (Ahmad)

**C-Test Report** Test reporting is a critical component in the process of software testing, meant to provide an overview and convey information about testing activities, both during and upon the conclusion of a testing cycle. These reports are often divided into two types: Test Progress Reports, issued throughout the testing process, and Test Summary Reports, issued at the end of testing activities.

During the monitoring and control of testing, a Test Progress Report is frequently provided by the test manager for stakeholders. This report typically includes the status of ongoing test activities compared to the test plan, impediments to progress, plans for future testing, and an evaluation of the quality of the test object.

Once the pre-defined exit criteria are met, the QA Engineer creates a Test Summary Report, which encapsulates the testing performed based on the most recent Test Progress Report and other relevant data. This report usually contains a summary of testing performed, events during the testing period, deviations from the plan, status of testing and product quality in relation to the exit criteria, ongoing or potential impediments, various metrics (defects, test cases, coverage, activity progress, resource usage), residual risks, and any reusable test work products generated.

The content of a test report is influenced by several factors such as project complexity, organizational requirements, and the software development lifecycle. For example, projects with numerous stakeholders or those that are highly regulated may demand more detailed reporting compared to simpler software updates. Agile development, for instance, may incorporate test progress reporting into task boards, defect summaries, and burndown charts, and might be reviewed in daily stand-up meetings.

Moreover, test reports must be tailored to the specific audience they are intended for. Reports for a technical audience or a testing team may differ from those prepared for executive summary reports. In the former, detailed information about defect types and trends might be essential, while for the latter, high-level reports encapsulating status summaries of defects by priority, budget, schedule, and test conditions may be more appropriate.

(ISTQB, 1 July 2021)

## **2.2 Important Common Techniques in Software Testing (Automation and Manual) <sup>7</sup>**

### **2.2.1 Exploratory Test**

Exploratory testing is a dynamic testing method, where tests are informally designed, executed, documented, and evaluated on-the-go during the test execution. It uses the results of these tests to enhance the understanding of the component or system and to identify areas that may need further testing.

Exploratory testing is often conducted using a structure called session-based testing. Within a pre-set time limit, testers rely on a test charter with specified objectives to guide the testing process. They may utilize test session sheets to record the procedures followed, and the discoveries made during the testing.

This testing methodology proves to be particularly useful under conditions where specifications are minimal or inadequate or there is a lack of documentation and requirements, or when there's considerable time pressure on testing activities. Furthermore, exploratory testing is beneficial to supplement more formal testing techniques. The method can incorporate the use of other techniques such as black-box, white-box, and experience-based techniques. (ISTQB, 1 July 2021)

During the Exploratory testing, the QA Engineer does a map of functionality<sup>8</sup> containing the feature in the system.

---

<sup>7</sup> All these Techniques are used practical in the project.

<sup>8</sup> Map of functionality is an Excel sheet contain the feature in the system and it attached with the study.



### **2.2.2 Risk-based Testing.**

Risk-Based Testing (RBT) is a strategy that involves assessing potential risk factors in software to prioritize testing efforts. It revolves around the idea of risk probability, examining various elements such as software complexity, business criticality, usage frequency, and potential defect areas. The goal is to focus on the most impactful parts of the software that have a higher chance of containing defects.

This approach classifies the features, modules, and functionalities of software or a product based on their likelihood of failure in a live environment and the subsequent impact on the customers. By prioritizing in this way, it's possible to maximize testing efficiency and effectiveness.

In this context, risk refers to any uncertain event that can positively or negatively affect a project's success. This event could stem from past, present, or potential future circumstances, and may influence cost, business objectives, technical aspects, and quality standards of a project.

Risk itself can be seen as a double-edged sword. Positive risks, known as opportunities, can contribute to business sustainability. These could include actions like investing in a new project, revamping business processes, or developing new products. On the other hand, negative risks, also known as threats, can hinder a project's success, and it's essential to take necessary measures to minimize or eliminate them.

RBT can be particularly beneficial for projects that face constraints in terms of time, resources, or budget. It can also be utilized in situations where risk analysis can help identify vulnerabilities such as potential SQL injection attacks. Furthermore, RBT can prove to be useful in cloud computing environments where security testing is essential. High-risk projects, such as those involving unfamiliar technologies or business domains, can also leverage RBT. Lastly, RBT is also suitable for incremental and iterative models where changes occur frequently. (Hamilton, 2023)

### **2.2.2.1 Risk Analysis Procedure**

Risk analysis involves collaboration between key stakeholders from both technical and business sides of a program. This includes individuals like the product owner, product managers, business analysts, architects, testers, and customer representatives.

The process often entails organizing brainstorming sessions with these stakeholders to discuss and evaluate each feature of a product. The goal is to prioritize them according to their risk of failure and the potential impact on end-users in a live environment.

A range of project documents serve as inputs for these brainstorming sessions, including the requirements document, technical specification documents, architecture and design documents, business process document, and use case document, among others.

Stakeholders' knowledge about the product, as well as information about similar existing products in the market, also contribute to the discussion.

Additional sources of information might include data on the most frequently used functionality, advice from domain experts, and insights derived from previous versions of the product or comparable products in the market.

During these brainstorming sessions, risks associated with each feature are identified. These risks could be operational, technical, or business-related. Corresponding tests and scenarios are then assigned weights, and risk values are assessed based on the probability of risk occurrence and the potential impact of the risk. This comprehensive risk analysis process is critical in guiding the direction of testing efforts and ensuring product quality and reliability. (S., 2023)

### **2.2.2.2 Risk Analysis Methodology**

Risk-Based Testing (RBT) applies the concept of risk throughout all stages of the testing cycle: test planning, design, implementation, execution, and reporting. This approach can accommodate a vast number of possible test scenario combinations. The process unfolds as follows:

### Step 1: Risk Analysis with a 3x3 Grid

RBT uses a 3x3 grid for risk analysis. Each functionality, non-functionality, and related test case is evaluated by a team of stakeholders for its likelihood and impact of failure. Technical experts usually assess the 'Likelihood of Failure,' categorizing it as 'Likely to Fail,' 'Quite Likely,' or 'Unlikely' along the grid's vertical axis. The 'Impact of Failure,' if untested, is assessed by business specialists who categorize it as 'Minor,' 'Visible,' or 'Interruption' along the horizontal axis.

### Step 2: Placement based on Likelihood and Impact of Failure

All test cases are positioned on the grid based on their likelihood and impact of failure. High-risk cases (high likelihood and impact of failure) are grouped in the top right corner, indicating high importance. Conversely, 'Low Value' tests are grouped in the bottom left corner, representing lesser importance to the customer and requiring less focus.

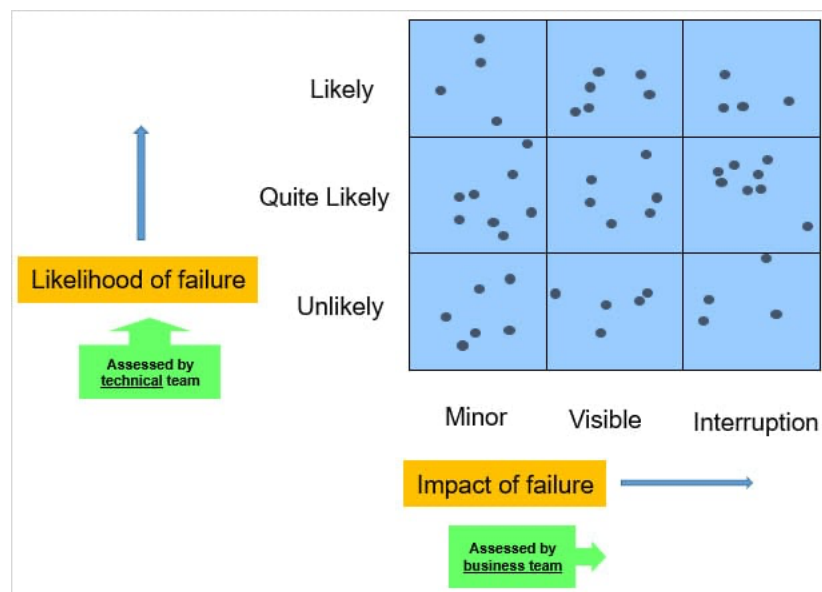


Figure 1: Likelihood, Impact of failure Matrix, Source: <https://www.softwaretestinghelp.com/risk-management-during-test-planning-risk-based-testing/>

### Step 3: Creating a Testing Priority Grid

Using the grid, tests are prioritized and assigned values from 1 to 5. The most crucial tests are given priority 1, while less significant ones are ranked 2, 3, 4, and 5. Test cases are then sorted and executed based on their priority numbers, with high-priority tests

executed first. Low priority tests are either executed later or even removed from the scope.

Testing Priority Grid		
4	3	1
5	4	2
5	5	3

Figure 2: Testing Priority Grid, Source <https://www.softwaretestinghelp.com/risk-management-during-test-planning-risk-based-testing/>

#### Step 4: Deciding the Depth of Testing

The level of detail in testing for the defined scope is decided based on the aforementioned ranking. High-priority tests (ranking 1) are tested more thoroughly by experts, while lower priority ones (ranking 2, 3, 4) are tested accordingly. Decisions to de-scope priority 5 features and tests can be taken based on time and resources available.

Level of Detail of Testing		
Identification	Thorough	More Thorough
De-scoped	Identification	Thorough
De-scoped	De-scoped	Thorough

Figure 3: Level of Detail of Testing, Source: <https://www.softwaretestinghelp.com/risk-management-during-test-planning-risk-based-testing/>

This detailed approach to testing helps testers identify high-value tests and decide on their detail level of testing based on priority rankings. This process guides better testing practices and helps reduce testing costs through optimization.

In essence, the RBT approach prioritizes tests based on the severity of the risks to pinpoint the most defective or risk-prone areas, which could significantly impact the business. The main objective is to differentiate between high-value and low-value items like product features, functionalities, requirements, user stories, and test cases. Greater focus is given to high-value test cases, while less emphasis is put on low-value ones. This initial step of risk analysis sets the stage for implementing Risk-Based Testing. The grouping of test cases into high and low value and assigning priority values is a crucial part of this process.

### **2.2.3 Regression testing**

Regression testing is a critical element in the software development cycle, designed to ensure the smooth operation of existing functionalities even after introducing new changes or additions to the software. It aids in identifying and tackling the recurrent issue of software bugs reemerging due to newly implemented modifications.

Essentially, the process involves drafting a test for a previously known bug and repetitively running this test after each modification to the code base. The objective is to instantly detect any alterations that might unintentionally revive the bug.

To illustrate how regression testing operates, let's imagine two scenarios. In the first, a software development company is preparing to launch an E-commerce product, with the initial build focusing only on core features. The company conducts a regression test with 1000 test cases to validate the basic functionalities. If the product passes these tests, it's ready for market release.

Upon the first product's successful reception, the team decides to add new premium features. These additions necessitate another round of regression tests. Along with 100 new test cases for the new features, the original 1000 test cases are rerun to confirm that the essential functionalities remain intact.

The second scenario follows a similar pattern but involves an social media website. The initial build undergoes 100 test cases before release. When new premium features are

added due to its success, these changes call for an additional 10 test cases for the new features and a rerun of the original 100 test cases. (Shaumik Daityari, 2022)

A common misunderstanding often arises, confusing regression testing with retesting. The following table delineates the differences between the two.

<b>Regression Testing</b>	<b>Retesting</b>
Regression Testing is software testing to ensure that a recent code change has not adversely impacted the existing functionalities.	Retesting is used to ensure that the test cases that used to fail previously have been fixed.
In Regression Testing, existing issues are not verified.	In Retesting, existing issues are verified to assess the failed test cases.
Regression testing determines the possibility if there could be any unanticipated negative effects.	Retesting ensures that the original problem which was found has been corrected.
Regression testing is done to catch new bugs if any in the software.	Retesting is done to fix existing bugs in the software.

Table 1: Regression Test VS Retesting (Kitakabee, 2023)

The Regression Test is time-consuming, requiring substantial time, effort, and cost. Therefore, it can be converted into an automatic test to save considerable time and expense. The power of automated testing becomes evident in regression tests, a topic we will delve into later.

#### 2.2.4 Smoke Testing

Smoke Testing is a part of the software testing process to insure the stability of a newly deployed software build. Often referred to as "Build Verification Testing", "Confidence Testing", or "Sanity Test" it provides the quality assurance (QA) team with a green light to proceed with more comprehensive testing ex. Functionality testing.

In essence, smoke testing involves conducting a rapid, minimal set of tests on each software build to confirm the major functionalities are working properly. It serves as an initial screening method to identify any significant issues or 'showstoppers' in the build, which could render further testing futile and wasteful in terms of time and resources.

Smoke testing becomes particularly important when new functionalities have been developed and integrated with the existing build deployed in a QA or staging environment. The main aim is to ensure all critical functionalities are operating as expected. During this process, the build is deployed in QA and a subset of test cases are selected. The testers then run these cases against the build, specifically testing the application's critical functionalities.

This method aims to unearth errors within the build. If the tests are successful, the QA team proceeds with Functional Testing. However, any failure necessitates sending the system back to the development team for further work. Consequently, Smoke Testing is repeated with every change to the build, ensuring its stability.

The importance of conducting smoke testing in the early stages of software development cannot be overstated. Any defects discovered later in the development process could not only be more costly to rectify but could also potentially disrupt the release of the product, acting as 'showstoppers'. Hence, smoke testing is an efficient preventative measure to ensure a smoother development process. (Hamilton, 2023)

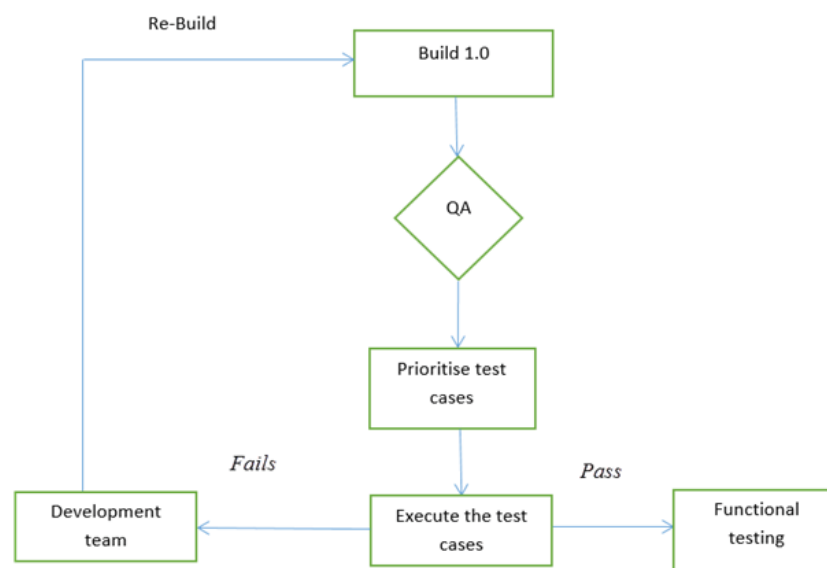


Figure 4: Smoke Testing Cycle Source: <https://www.guru99.com/smoke-testing.html>

The sanity test or smoke test is a quick method to detect the most severe bugs, or to quickly determine if a particular feature is functioning well. Therefore, it's a fast, time-saving method, but we cannot rely on it alone, nor can we replace it with comprehensive testing for the entire system. Features of this method are as follows:

- 1- The scope: It is limited to specific, important features within the system. Therefore, we cannot describe this method as comprehensive.
- 2- Its execution is fast: It takes from a few minutes to a few hours, with the aim of providing quick feedback on work without investing a lot of time in exhaustive testing.
- 3- Its standard is pass/fail: The result is binary, either white or black.
- 4- It can be automated: The test can be manual, but it is often automated to provide quick results. It also allows for repeated testing to ensure accurate results.



## 2.3 Automation Testing

Test automation, as defined by ISTQB®, is the application of software to carry out or assist in testing activities. In other words, it's the process where machines handle tasks that would normally require manual testing. This concept encompasses all activities related to ensuring software quality throughout the development stages, involving various development phases and testing levels. Everyone involved in the project, including developers, testers, analysts, and users, participate in these activities.

Even though "test automation" theoretically includes all activities in the testing process, it's often linked specifically to the automated execution of tests, which is typically performed using specialized tools or software. (Manfred Baumgartner, 2022)

Consider an engineering team that expands by adding a software engineer or two each quarter. They aim to please their customers by delivering more features at each sprint. Despite having a quality engineer or two to test these new features, they find that the faster they produce code, the more regression bugs pop up. Manual testing isn't sufficient; they need certain key test scenarios to be performed repeatedly, especially when changes are made. That's where test automation becomes invaluable. It's always advisable to start using test automation before experiencing a severe bug in production or a disastrous incident due to inadequate testing.

Testing isn't a one-off task; it's a necessary step each time there's a change in the software application. The more time that passes without testing, the greater the likelihood of failure. Therefore, continuous testing is not just an option but a vital requirement in the current Agile software engineering environment. Manual testing involves a tester manually executing predefined test cases on the system, as an actual user would. Automated testing allows these tests to be run by a script, saving the tester's time, and allowing them to concentrate on usability or exploratory testing. When executed correctly, automated tests can be more reliable than manual testing. Additionally, it gives the tester more time to glean valuable insights from the automated test results, which helps to increase overall test coverage of the software application.

Test automation is a key method for establishing and maintaining quality in a structured manner. The main advantage of test automation is its ability to detect and fix software bugs early and as close to the development environment as possible. This approach minimizes the negative impact of late defect detection and facilitates an ongoing feedback loop between the engineering and product teams. While the initial investment in automated tests might seem substantial, studies have shown that it pays for itself over time. Test automation allows teams to deliver new features more quickly and with better quality.

Test automation is a team effort that requires the dedication of business analysts, software engineers, and quality engineers or software development engineers in test (SDET). It relieves the team from the burdensome repetition of manual tests, paving the way for achieving quality quickly. While other activities like peer code reviews and internal design reviews help to identify defects early, test automation puts the team in a strong position to start testing the product with the end users.

There's a common misunderstanding that automated tests lessen the human interaction with the system being tested. While it's true that testers may not interact with the system as frequently as they do during manual testing, the process of creating and maintaining automated tests promotes team collaboration. This is achieved by team members providing feedback on the test code and design. Therefore, automated tests serve as a new channel of communication within the team, enhancing the system's quality and helping prevent bugs. (Sambamurthy, 2023)

### 2.3.1 Test Automation Objectives and Limitations

The adoption of test automation typically serves multiple purposes. Its core intention is not self-fulfillment, but rather enhancing testing efficiency, thus cutting down on overall testing costs. Other crucial benefits include shorter test cycles, reduced testing time, and the potential for more frequent test runs. This is particularly relevant in DevOps and DevTestOps testing methodologies where continuous integration, deployment, and testing are vital, only possible through well-functioning test automation.

Besides cost and time reduction, maintaining or elevating quality is a significant aim of test automation. This can be achieved by extending functional coverage and conducting tests that would require substantial time and resources if done manually. This may involve testing numerous data configurations, fault tolerance tests, or various forms of performance testing. Furthermore, running full test suites repetitively against different versions of the software under test (SUT) or in different environments (e.g., various browsers and mobile devices) is cost-effective only with automated testing.

However, test automation also has limitations. Sometimes the financial cost of automating certain tests outweighs the benefit. A machine can verify machine-readable results only. While test automation excels in comparing expected and actual behavior within the SUT, it falls short in system validation and assessing its suitability for intended use. It fails to detect faults in requirement definition or incorrect requirement interpretations. Test automation can't apply creativity or infer hidden meanings, hence it can't entirely replace structured dynamic or exploratory testing. To be effectively automated, the SUT must reach a certain level of stability and be relatively defect-free at user and system interfaces, or the tests will be subjected to constant changes.

(Manfred Baumgartner, 2022)

### **2.3.2 Manual Testing VS Automation Testing**

Manual testing is a process where a person acts like a user to test software. This kind of testing needs creativity and an understanding of what the software is supposed to do. It's useful for short projects and ones that will only be done once. Humans are great at catching bugs that machines might miss, especially in the user interface. But, manual testing can take a lot of time, especially when there's a lot of data to check. It can also get boring, like filling out the same form over and over again, and can sometimes be hard to debug.

Automated testing, on the other hand, uses special software to do the testing. It's fast and reliable and can repeat the same tests over and over. This is especially useful when you want to test software on different platforms or environments. Automated testing can also run anytime and can even send detailed reports to stakeholders. It's great for testing the user interface and how the software performs. However, it can't cover exploratory testing well, and tools for automated testing can be expensive. It can take time to analyse the test results and often requires coding skills. Automated testing is not suitable for short-term projects and requires a stable application to run. Sometimes, the testing tool might even introduce its own bugs. (Jose, 2021)

Here is a comparison between Manual Testing and Automation Testing in two aspects: Advantages, and Disadvantages.

Manual Testing	Automated Testing
1. Suitable for exploratory testing	1. Quick, reliable, and eliminates human error
2. Effective for short-term projects	2. Consistent and comprehensive coverage of data
3. Good for single-release projects	3. Tests are repeatable and extendable across platforms and environments
4. Humans can notice defects that automation overlooks	4. Test execution can be programmed and scheduled automatically
5. Good for user experience and usability testing	5. Reduces time for testing and improves productivity of human testers
6. No dependency on tools and infrastructure	6. Detailed test logs and execution reports
	7. Suitable for GUI and performance testing
	8. Assured test coverage

Table 2: Advantages in Manual Testing &amp; Automation Testing

Manual Testing	Automated Testing
1. Time-consuming compared to automated testing	1. Limited coverage on exploratory and negative testing
2. Some tasks can be tedious to perform	2. Static testing and reviews are limited
3. Requires heavy investment in human resources	3. Tools can incur high costs including infrastructure and maintenance
4. Debugging complex defects can be difficult	4. Analysis of test results is time and cost-intensive
5. Higher probability of errors	5. Requires coding skills
	6. High maintenance cost due to GUI dependency
	7. Not suitable for short-term projects
	8. Requires a stable application for execution
	9. May introduce its own defects

Table 3: Disadvantages in Manual Testing &amp; Automation Testing

## 3 Test Automation Framework and Tools

This chapter encompasses a description of each tool employed within the testing framework already used in the project.

### 3.1 WebDriverIO

WebdriverIO is a comprehensive framework designed for the testing of web and mobile applications. It allows for the execution of lightweight component tests, as well as more comprehensive end-to-end (e2e) testing scenarios in a user's environment, be it a browser or mobile device. The framework incorporates intelligent selector strategies, simplifying interactions with Frontend frameworks like React components or executing deep selector queries with nested shadow Document object Model DOM<sup>9</sup> trees. All interactions occur through a standardized automation protocol, ensuring native behavior and not mere JavaScript emulation. WebdriverIO is extendable, compatible, and feature-rich. Adding helper functions and command sets is useable, easy, and practical. WebdriverIO can interact and operate for cross-browser testing by WebDriver Protocol<sup>10</sup> and the Chrome DevTools Protocol<sup>11</sup> for Chromium<sup>12</sup>-based automation

---

<sup>9</sup> The Document Object Model, or DOM, acting as a programming interface for web documents. It enables programs to modify a webpage's structure, style, and content. Essentially, the DOM turns a web page into an object that can be manipulated by languages such as JavaScript.

<sup>10</sup> The WebDriver Protocol is a standardized interface used for automating web browsers. It defines a set of commands and actions that a program can use to control a browser, just like a human user would. This protocol is fundamental for tasks such as automated testing of web applications, and it's widely supported across various browser types. <https://w3c.github.io/webdriver/>

<sup>11</sup> The Chrome DevTools Protocol is a set of tools that allows developers to interact directly with Google Chrome and other Chromium-based browsers. It provides a range of features for inspecting, debugging, and profiling web applications, enabling direct control over the browser's internal features and functions. The protocol is especially useful for automating browser tasks, debugging performance issues, and understanding the inner workings of complex web applications. <https://chromedevtools.github.io/devtools-protocol/>

<sup>12</sup> Chromium is an open-source web browser project developed by Google. It forms the foundation for various browsers, including Google Chrome, Microsoft Edge, and Opera, by providing the bulk of their source code. Chromium is designed to be fast, stable, and secure, offering developers a platform to build and test their web applications across different operating systems.

using Puppeteer<sup>13</sup>. The framework also offers a multitude of built-in and community plugins, enabling users to customize their setup to meet specific needs. You can utilize WebdriverIO to automate modern web applications (such as those developed in React, Vue, Angular, Svelte, etc.), hybrid or native mobile apps (either in an emulator/simulator or on an actual device), native desktop applications (like those built with Electron.js), and unit/component testing of web components in browsers. WebdriverIO employs the WebDriver and WebDriver-BiDi protocols, which are widely supported by browser vendors and ensure a true cross-browser testing experience. Unlike other automation tools, WebdriverIO doesn't require modified browser engines or JavaScript injection to mimic user behavior. Instead, it relies on a universally agreed standard for automation, ensuring compatibility for the foreseeable future. It also supports proprietary automation protocols such as Chrome DevTools, which are used for debugging and introspection. WebdriverIO is a truly open-source project, run with open governance and owned by the non-profit entity, the OpenJS Foundation. This setup guarantees the project's growth and direction align with the interests of all participants, free from monetary influence. As a result, the project is developed independently, providing free 24/7 support through its community channel, fostering a sustainable community, and offering abundant opportunities for those who contribute to and engage with the project. (OpenJs, 2023)

WebdriverIO, an independent and customized version of WebDriverJS, was developed by Christian Bromann. WebDriverJS, the official JavaScript version of the Selenium API, runs on Node.js and is available as 'selenium-webdriver' in npm. WebdriverIO simplifies the complex and verbose syntax of JavaScript, along with its tricky asynchronous promise management. It offers users straightforward action commands, ensuring each test step is synchronous, eliminating any concerns about missed steps in the test code. WebdriverIO is incredibly flexible, letting users select their preferred assertion libraries, reporting tools, and other framework components. WebdriverIO is available through npm and operates on Node.js, a JavaScript run-time environment that enables JavaScript to run outside the browser. This means it can function on various

---

<sup>13</sup> Puppeteer is a Node.js library that provides a high-level API to control Chrome or Chromium browsers programmatically. It's often used for tasks such as generating screenshots or PDFs of pages, crawling websites for SEO audits, or automating UI testing of web applications. Essentially, Puppeteer allows developers to simulate user actions, like clicking buttons or filling forms, on a website without a human actually interacting with it. <https://pptr.dev/>

operating systems, including macOS, Linux, and Windows. A trend chart over the past five years shows an increasing number of WebdriverIO downloads, indicating its growing popularity and use year after year. (Shukla, 2021)

A trend chart over the past three years shows an increasing number of WebdriverIO downloads, indicating its growing popularity and use year after year.

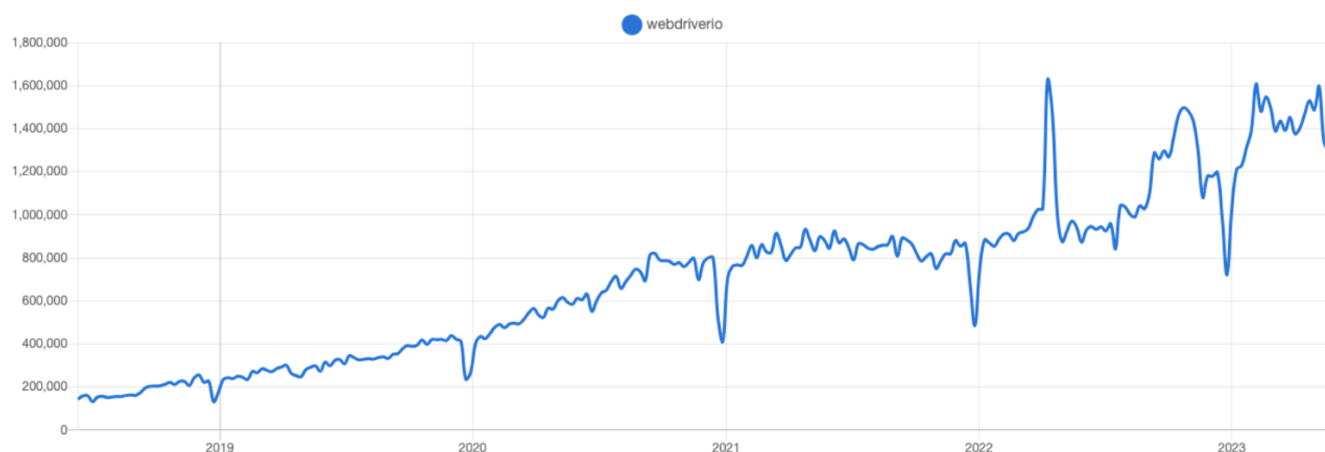


Figure 5: The growing popularity of WebDriverIO source: [www.npmtrends.com](http://www.npmtrends.com)

### 3.1.1 WebDriverIO Architecture

WebDriverIO is a tool built using NodeJS<sup>14</sup>, It uses a system called the JSON Wire Protocol<sup>15</sup> to perform automated testing, WebDriverIO uses a structure called RESTful architecture. This is like a set of rules for how information gets sent and received during the testing process.

To perform automated testing, WebDriverIO uses a structure called RESTful architecture. This is like a set of rules for how information gets sent and received during the testing process.

Here's how it works: you write a testing script using JavaScript and the WebDriverIO tool. This script is like a list of tasks you want to test on a website or application. When

<sup>14</sup> Nodejs is a popular open-source platform for creating applications.

<sup>15</sup> JSON wire protocol play the role of translator or as mediator between client libraries and Web Drivers. Because servers do not understand programming languages, JSON Wire Protocol uses the process of serialization (converting object data to JSON format) and de-serialization (converting JSON format to object)



you run this script, it sends a service request, like a message, using NodeJS. This request is sent as an HTTP command.

This message uses the JSON Wire Protocol and is forwarded to the browser by a services module. Once the browser gets this message, it performs the tasks that you wrote in your script. By doing this, it tests the different functions of your application to see if they're working properly. (Tiwari, 2023)

The subsequent figures illustrate the architecture of WebDriverIO and demonstrate its operational mechanism.

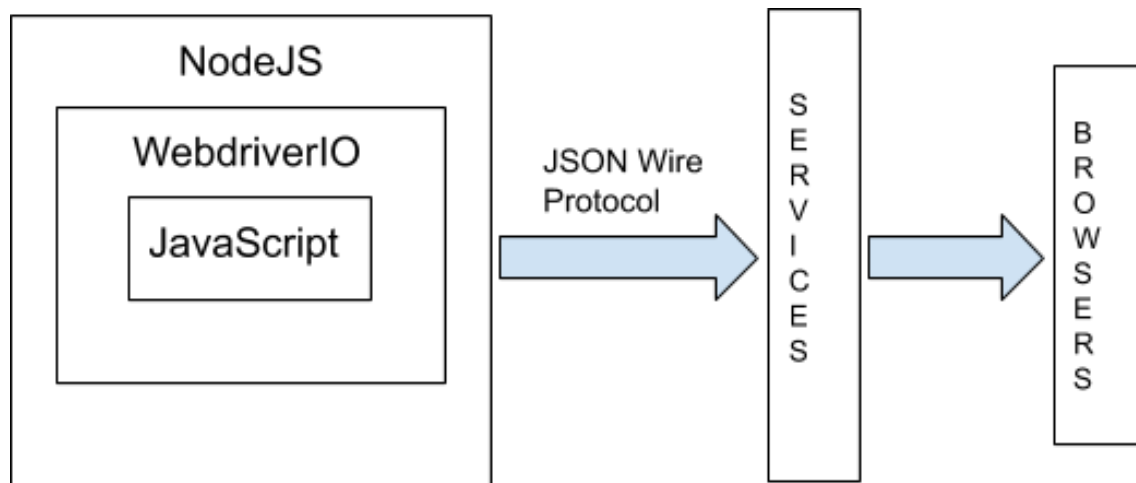


Figure 6: WebDriverIO Architecture source: <https://www.browserstack.com/guide/webdriverio-tutorial-for-selenium-automation>

### 3.2 Behavior Driven Development & Cucumber Framework

Software creation begins with an idea, often one that promises value or potential profit. The challenge for developers lies in transforming this conceptual spark into functional software. Often, the person conceiving the idea lacks the programming expertise to realize it, necessitating clear communication to relay this concept to those who can.

Software projects usually involve collaborative efforts; thus, high-quality communication becomes critical for success. Good communication encompasses not only the articulation of ideas but also the pursuit of feedback to confirm the clarity of the transmitted

ideas. Agile software teams have adopted working in small increments, utilizing the incrementally constructed software as feedback for stakeholders, verifying the alignment of the software with their expectations.

However, sometimes, misunderstandings arise, leading to the implementation of wrong concepts and functionality, deviating from the original idea. This not only results in the waste of resources but can also compromise the codebase integrity. Misunderstandings may proliferate in the codebase, further entrenching these errors.

This is where Behavior-Driven Development (BDD) and tools like Cucumber come into play. BDD, an extension of Test-Driven Development (TDD), emphasizes the importance of creating a shared, universal language to discuss the system. It advocates writing acceptance tests as legible examples, enabling a shared understanding amongst the team members. These tests also aid in obtaining early feedback from stakeholders, ensuring alignment with the original idea.

A term coined by Eric Evans in his book *Domain Driven Design*, "Ubiquitous Language," encapsulates the idea of a common language understood by everyone involved in the project. A ubiquitous language eliminates the need for translations between different dialects, minimizing the chance of misunderstandings.

Cucumber plays an instrumental role in fostering this shared language. It bridges the communication gap between the technical and business sides of a team. Cucumber tests interact with the developers' code, but they're written in a way that non-technical stakeholders can understand. By collaborating to write these tests, team members not only decide on the required behavior but also learn to describe it in a universal language.

The process of writing these tests before development begins helps identify and rectify potential misunderstandings, preventing them from infiltrating the codebase.

(Matt Wynne, 2017)

The term "Cucumber framework" typically refers to Cucumber, which is a widely used open-source tool for behavior-driven development (BDD) in software testing. Cucumber is designed to bridge the gap between technical and non-technical stakeholders, enabling collaboration and clearer communication in the software development process.

In the context of automation testing, Cucumber allows testers, developers, and business stakeholders to write executable specifications (known as feature files) in a human-readable format. These feature files describe the expected behavior of the software system from the user's perspective, using a structured language called Gherkin<sup>16</sup>

The keywords used in Gherkin are:

**Feature:** Describes a high-level feature or functionality of the system.

**Scenario:** Represents a single test scenario or test case.

**Given:** Defines the initial context or preconditions of the test scenario.

**When:** Specifies the action or event that triggers the desired behavior.

**Then:** Describes the expected outcome or result of the scenario.

**And / But:** Used for additional steps within a scenario to enhance readability.

**Example:**

The subsequent example elucidates the Gherkin syntax utilized by the Cucumber framework to characterize a feature for development or testing. This aims to unify comprehension amongst team members, ensuring even non-technical personnel, particularly project managers or clients, can grasp the content as proficiently as technical individuals. The feature file in focus details the login functionality and its prerequisites, such as navigation to the login page. The sequential steps include inputting the username and password, followed by clicking on the login button. The outcomes are then presented, contingent on whether the user has provided accurate credentials.

---

<sup>16</sup> Gherkin is a structured language used for writing human-readable specifications in behavior-driven development (BDD). It is specifically designed to facilitate communication and collaboration between technical and non-technical stakeholders in software development

```

1 Feature: As a user I able to Login with sales role
2
3 Background:
4   Given the user is on the login page
5
6 Scenario Outline: (Happy path) Sales user logs in and is redirected to the sales dashboard
7   When user select the Sales button
8   And user select the appropriate Car Company <marke>
9   And user enter valid sales user credentials <username> and <password>
10  And user click the Login button
11  Then user should be redirected to the sales dashboard
12  And the user should see a flash message <message>
13
14
15 Examples:
16   |marke|username| password|message|
17   |Seat|GF22008| *****|Welcome to the club,|

```

Figure 7: Cucumber code snippet

### 3.3 Page object Model

The Page Object Model (POM) is a widely recognized design pattern in test automation that establishes an Object Repository for web User Interface (UI) elements, enhancing both the readability and maintainability of automation scripts.

Under this model, each web page within the application corresponds to a specific Page Class. These classes identify the various Web Elements present on the web page and incorporate methods that execute operations on these elements. The nomenclature of these methods typically reflects the tasks they perform, ensuring clear and concise code.

The POM's primary advantages lie in its distinct separation of operations and flows in the UI from verification processes and its centralized object repository. As a result, the code is cleaner, more comprehensible, and easier to maintain.

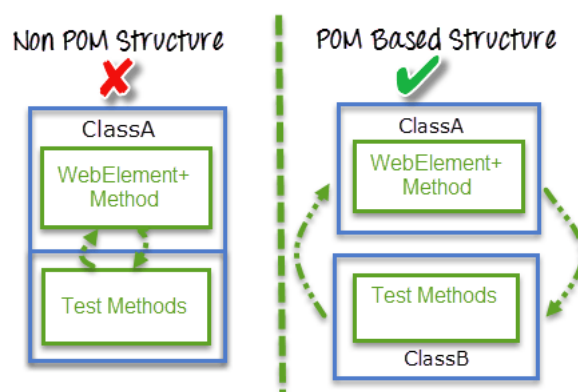


Figure 8: POM structure, source: <https://www.guru99.com/page-object-model-pom-page-factory-in-selenium-ultimate-guide.html>

1. **Code Segregation:** POM ensures that the UI operations and flows are decoupled from verification. This separation leads to cleaner code that's easier to understand, enhancing readability and maintainability.
2. **Centralized Object Repository:** With POM, the object repository is independent of the test cases, enabling its reuse for different tools and purposes. This centralization streamlines the utilization of web elements across various test scripts, reducing potential duplication.
3. **Code Optimization:** The application of POM results in more efficient, less redundant code due to the reusable page methods contained within the POM classes. This reusability not only promotes code consistency but also improves overall test maintenance.

In essence, POM provides a logical structure to automation scripts, resulting in tests that are easier to manage, more robust, and less susceptible to changes in the UI, thus lowering the cost of test maintenance. (Rungta, 2023)

### 3.4 Typescript

Initiated internally at Microsoft in the early 2010s and subsequently open-sourced in 2012, TypeScript has made a substantial impact on the realm of software development. The development of TypeScript was spearheaded by Anders Hejlsberg, who also played a pivotal role in developing eminent languages such as C# and Turbo Pascal.

TypeScript is often referred to as a "superset of JavaScript" or "JavaScript augmented with types," implying that any valid JavaScript code is also valid TypeScript code, with the addition of type annotations and other type-centric constructs. This interoperability provides developers with an accessible entry point, where they can gradually adopt TypeScript's type-checking features as needed. It has gained significant traction among developers for its potential to improve productivity, catch bugs early, and enable powerful tooling like autocompletion and refactoring.

At its core, TypeScript is comprised of four primary elements:

1. **Programming Language:** TypeScript includes all existing JavaScript syntax. Furthermore, it introduces exclusive TypeScript-specific syntax for defining and

using types. This is particularly beneficial in large-scale applications where type-safety can prevent bugs and enhance code quality.

2. **Type Checker:** TypeScript embodies a software that accepts files written in JavaScript and/or TypeScript, processes all the constructs (variables, functions, etc.) created, and signals if there are possible setup errors. This allows for early error detection during the development phase, saving valuable debugging time at runtime.
3. **Compiler:** TypeScript also operates as a compiler. This application executes the type checker, identifies any issues, and subsequently transcribes the scrutinized code into equivalent JavaScript. This allows TypeScript to be used in any environment that JavaScript runs.
4. **Language Service:** Lastly, TypeScript functions as a language service. This tool utilizes the type checker to facilitate editors like VS Code in providing beneficial utilities to developers. These utilities range from autocompletion and syntax highlighting to more sophisticated features like 'Go to Definition' and automated refactorings.

Thus, TypeScript offers a comprehensive solution that aids developers in writing safer, self-documenting code, ultimately leading to more robust, maintainable software.  
(Goldberg, 2022)

### 3.5 Allure Report

The Allure Framework is a multi-language test reporting tool that's both flexible and lightweight. It offers a clear overview of tested items and provides detailed insights into failed tests, such as test history, stack traces, attachments, and more. It is primarily designed to create detailed and informative reports in a continuous integration (CI) environment.

Allure Reports serve as a crucial component in test automation due to the following key features:

1. **Clear Representation of Test Execution:** Allure provides a clear, understandable structure of test execution, displaying each test case individually along with its status - passed, failed, or skipped, which makes it easy for both developers and non-technical stakeholders to understand the state of a project.

2. **Insightful Test Metrics:** It delivers insightful metrics on the total number of test cases, the ratio of passed to failed tests, severity, and broken-down statistics of test cases. This allows teams to monitor the overall quality of their software over time.
3. **Test Case History:** Allure Reports keep track of previous test executions, thereby providing a historical analysis of test cases. This enables developers to assess the performance of tests over time and identify any recurrent issues.
4. **Test Case Severity and Priority:** Allure Reports allow developers to mark test cases according to their severity and priority. This helps in understanding the impact of each test case and in deciding which issues need to be addressed first.
5. **Debugging Support:** In case of failed tests, Allure Reports provide detailed error logs, screenshots, and even video recordings (if integrated with respective tools) that can help in quick and effective debugging.
6. **Integration Capability:** Allure can be integrated with many popular testing frameworks, like JUnit (Java), PyTest (Python), NUnit (.NET), Mocha (JavaScript), and many more. It can also be integrated into the CI/CD pipeline for generating reports after every deployment.

Overall, Allure Reports is a powerful tool for visualizing test execution results in a comprehensive and user-friendly way, making it an essential component in a test automation setup.

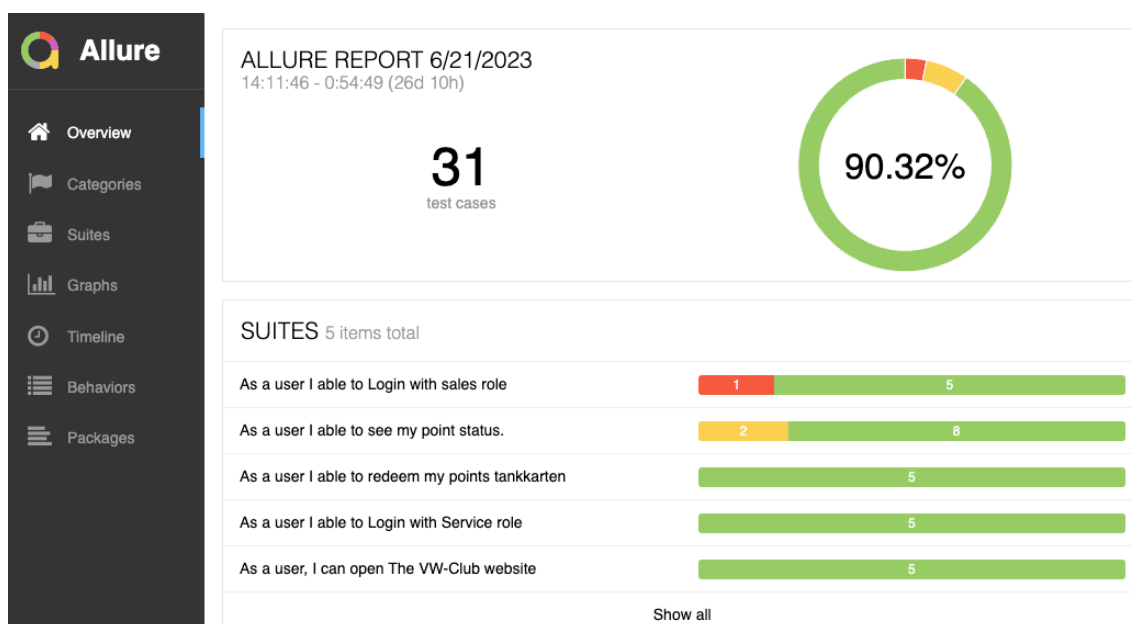


Figure 9: Screenshot from Allure Report

## 3.6 Continues Integration / Continues Delivery CI/CD

### 3.6.1 CI/CD Fundamentals

In this part, we will explore the significance of comprehensive testing within the software project lifecycle, as well as the critical role of Continuous Integration and Delivery, also known as CI/CD, in upholding software quality. CI/CD refers to the ongoing process of incorporating updates into software projects and consistently delivering these changes. Test Automation is an integral part of the CI/CD procedure. In fact, a CI/CD configuration is deemed incomplete without the incorporation of Test Automation in the process.

In software engineering, the practices of Continuous Integration and Delivery (CI/CD) enable us to automate the build and testing process of our code. This automation ensures the code meets specified requirements. Once the tests are successfully passed, teams are provided with options to automate the delivery of the code. The combined process of Continuous Integration, Continuous Delivery, and Continuous Deployment can significantly decrease the time taken to deliver application improvements. This allows the engineering team to concentrate solely on enhancing the product and its code.



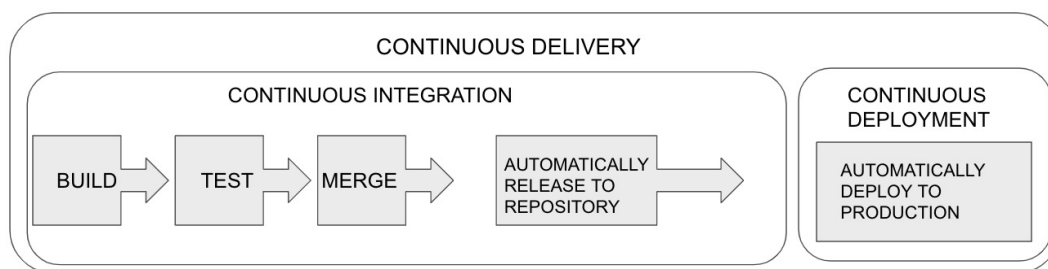


Figure 10: CICD Source: (Sambamurthy, 2023)

Every time a code or configuration change occurs, automated tests are carried out as part of the Continuous Integration (CI) system, effectively acting as an extensive regression testing platform. Modern CI systems have the capacity not only to perform tests that verify functional behavior, but also to evaluate the performance and security aspects of a software application. A well-executed CI system promotes best practices in development by offering continuous feedback.

One significant advantage of Continuous Integration (CI) systems is the transparency they offer to the entire team during the release and feedback process. With just a few clicks, any team member can monitor progress and instantly identify any failures.

The Continuous Integration and Continuous Delivery (CI/CD) process begins when developers commit<sup>17</sup> changes to their code in Git<sup>18</sup>. This triggers the CI system to initiate a new instance of the pipeline. During this stage, the code is compiled and initial tests, including unit and component tests, are performed. Code linting tools and code coverage analysis tools are also used at this point. If the build and tests are successful, the developer then creates a merge request to receive feedback from the team. Following this code review and feedback cycle, the developer integrates the code into the main or master branch. This action triggers the unit and component tests once more. Additionally, the code artifact created during the build process is deployed to the subsequent test environment. Here, smoke, and end-to-end (E2E) tests are carried out to verify business scenarios involving multiple services or user interfaces. This iterative feedback process

<sup>17</sup> Commit: A commit represents a snapshot of the repository at a specific point in time. It records changes made to the files and includes a unique identifier, author information, and a commit message describing the changes

<sup>18</sup> Git is a distributed version control system that is widely used in software development to manage source code and track changes made to files over time. It was developed by Linus Torvalds in 2005, in order to control the development of the Linux kernel. Git enables multiple developers to collaborate on a project by offering a mechanism to track and merge changes made by different individuals. Each developer can maintain their own local copy of the repository, which encompasses the complete project history, including all files and their respective versions.

continues until the final code is deployed to the production environment. Even at this stage, additional tests may be run to ensure the reliability and robustness of the code. (Sambamurthy, 2023)

### 3.6.2 Continues Integration CI

Continuous Integration (CI) effectively tackles the challenges of software integration, particularly as applications become more complex and their codebases expand with multiple components. CI allows engineers to frequently incorporate their work and promptly detect errors through code compilation, building, and unit tests. Engineers often commit daily, promoting the breakdown of larger changes into manageable chunks of code. The frequent, automated building and testing of the product throughout the day inspires higher confidence in the project's status.

The CI server routinely checks the version control repository, typically every minute or so. It runs a preconfigured build script after fetching the latest version of the source files. The CI software, equipped with a dashboard, provides options to initiate various stages of the build process. The software can also notify stakeholders about the current build status through emails or instant messaging platforms. It's generally a good practice to include database setup in the CI build script to maintain the coherence of the entire software application.

Implementing a CI system encourages several beneficial development practices, including:

- Faster identification of broken code
- Swift repair of broken builds
- Increased automated test coverage.
- Regular code reviews and peer feedback

CI systems efficiently compile and build source code once, deploying it across multiple environments. While the build's configurability may differ by platform, the core CI process remains consistent. Common practice involves executing the same build script against an environment-specific properties file. This ensures that we can release usable software at any time and in any environment, a significant advantage that eliminates considerable manual effort.

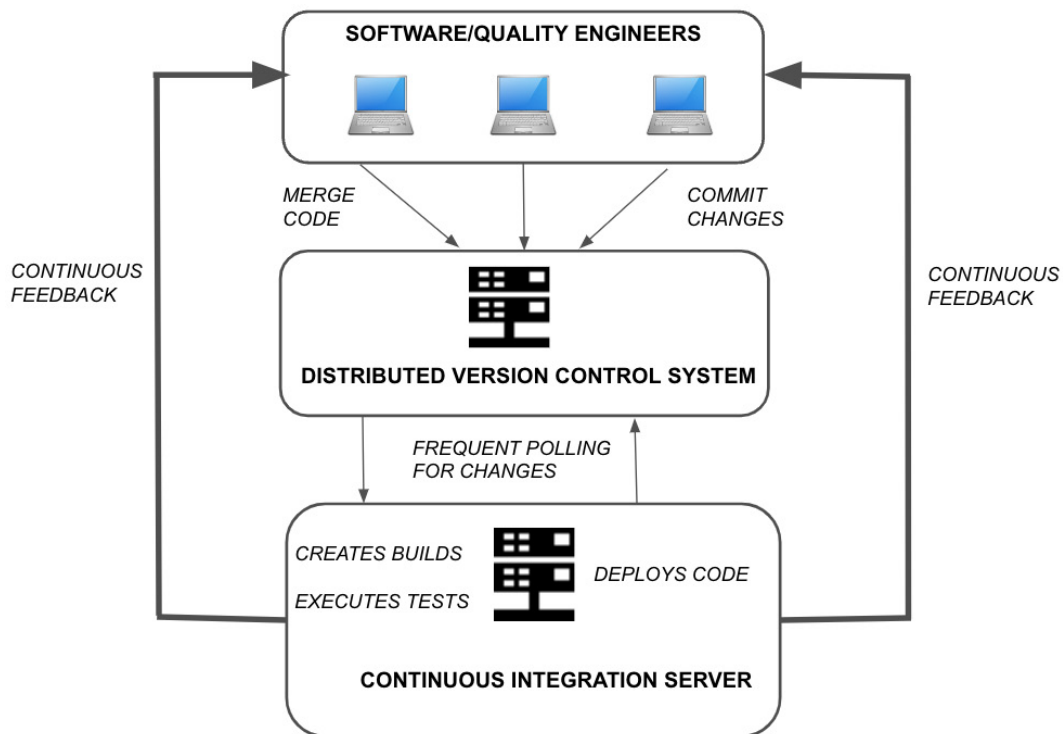


Figure 11: CI Systems Source: (Sambamurthy, 2023)

Incorporating a CI system into existing projects can seem challenging, as teams may resist modifying familiar processes. However, it's crucial to highlight the time saved by eliminating repetitive build tasks and the increased visibility gained by integrating individual changes into the codebase.

While the role of a quality engineer in the CI process may seem minimal, it's quite significant. Quality engineers must regularly review the automated tests run as part of the pipeline to enhance coverage and reduce instability. This also enables them to get involved early in the build and deployment process with every change. Quality engineers are also responsible for promoting the right quality mindset within the team. In case of a CI pipeline failure due to functional or end-to-end tests, it's often a quality engineer who responds first, debugs the failure, and logs a defect if needed.

(Sambamurthy, 2023)

### 3.6.3 Continues Delivery CD

Continuous Delivery (CD) is the practice of transferring software from the development phase to the end user seamlessly and efficiently via a deployment pipeline. This pipeline is a series of automated processes designed to transport the software from the engineer's workspace to the product's end users. The objective is to reduce the manual steps associated with deploying software, facilitating quicker and more stable transitions to environments akin to production. The main goal of CD is to establish a reliable and repeatable software delivery process. One of the significant benefits of the Continuous Integration/Continuous Delivery (CI/CD) system is its ability to revert to the latest functioning version of the software if critical errors arise. This rollback feature allows the quick detection and correction of configuration or code errors. Each modification to the code or configuration undergoes the same set of procedures.

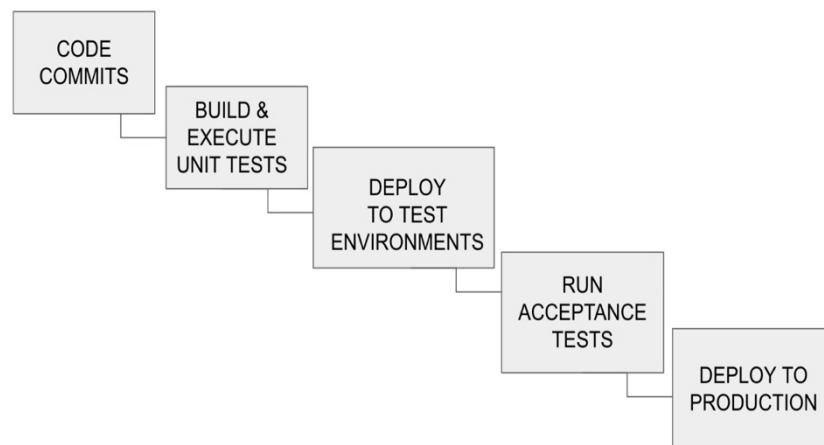


Figure 12 : Deployment Pipeline Source: (Sambamurthy, 2023)

By the time it reaches the production environment, we can be confident that:

- 1- The code has been constructed without errors.
- 2- Engineers have verified their changes based on the success of unit and functional tests in the pipeline.
- 3- Product and business teams have accepted the change, as evidenced by successful End-to-End (E2E) and user acceptance test cases within the pipeline.
- 4- The delivery system is ready, as the code has been deployed and tested in a production-like environment.
- 5- The most recent version of the code and configuration change is available in the version control repository.

Though it may sound ideal, the implementation of a CD process is challenging and requires the coordinated efforts of multiple teams. The key is to understand the existing deployment system, simplify it into repeatable steps, and utilize modern technologies for automation. This process can be implemented gradually, and over time, it evolves into a refined and efficient system. A crucial part of the CD process is the automation of acceptance tests, which are executed within the deployment pipeline. These tests could be End-to-End (E2E) User Interface (UI) or Application Programming Interface (API) tests. They play a significant role in preventing a build that fails to meet a key business specification from being deployed to production or a similar environment. By continually updating the automated tests for each code change, these tests ensure that the proposed build delivers the expected value to the end users. A common limitation of running automated acceptance tests within the pipeline is the extended duration it requires for a single merge to the master. This delay occurs because the application and all its configurations must be constructed and deployed before the tests' prerequisite steps can commence. Consequently, this increases the overall acceptance test cycle time, including test run times. Although acceptance tests can take anywhere from a few minutes to several hours, there are always opportunities to improve the efficiency of automated tests. Quality engineers and Software Development Engineers in Test (SDETs) should continuously seek ways to refine their tests, reducing the overall deployment pipeline runtimes. Continuous Delivery is not limited to collaboration between engineers and the infrastructure team. It requires support from product owners/managers, executive sponsors, and all other stakeholders. CD represents a paradigm shift and is the new standard in the software delivery industry. (Sambamurthy, 2023)

## 4 Return on Investment (ROI) of Automation Testing

Transitioning from manual testing to automated testing is a significant undertaking. Just like any major software project, there are numerous potential challenges that may hinder the implementation process. As a leader within your organization, it's crucial to ensure that your team's efforts provide a favorable return on investment (ROI). However, quantifying the ROI of software testing is a challenging task, and it becomes even more complicated when it comes to measuring test automation ROI. The difficulty in measuring test automation ROI stems from the multitude of variables involved. Stakeholders are primarily concerned with one crucial question: How beneficial would automating testing truly be, and when can we witness its impact on our business? Regrettably, the latter question doesn't have a clear-cut answer. There are various arguments supporting the adoption of an automated testing platform. The fact says that embracing such a platform is likely to save you both time and money. However, no-one can specify the exact amount. Nevertheless, this section will guide you on how to measure the savings resulting from test automation. In the following section, we will explore the key factors you need to evaluate when implementing a test automation platform. We will discuss the interplay of these variables and provide insights on how you can determine the extent to which test automation can save resources. (testim, 2020)

### 4.1 What is ROI in Automation Testing?

Return on Investment (ROI) is a crucial measurement that shows the potential financial return you can get from incorporating an automation strategy in your Quality Assurance (QA) processes. This measure is expressed as a percentage and can provide various useful insights:

- 1- It can help you understand the monetary benefits or drawbacks that come with investing in automation for your business.
- 2- ROI can offer you an estimated timeline for when the investment might start showing profits.
- 3- It provides a clear image of the potential profits and risks tied to the investment.

4- Lastly, a good ROI can strengthen your business pitch to potential investors, persuading them to back your plans.

ROI, in this case, is calculated by comparing the savings from replacing manual tests with automated tests to the costs involved in designing these tests and establishing the test environment.

Savings refer to the profit generated from transitioning from manual to automated tests. Costs, on the other hand, are the expenses incurred in the creation and implementation of the tests. Both can be expressed in terms of either time or money.

Benefits or savings can come from:

- a. The financial savings from automation.
- b. Reduced financial risks brought about by automation, such as the costs you might have to bear for troubleshooting if a defective product is released.
- c. Time saved by automating tasks.
- d. Money saved from not having to pay salaries for manual testing.

Costs, meanwhile, can include:

- a. The money spent on implementing automation (e.g., forming an automation team or hiring third-party testing services).
- b. The time required to create and implement automated tests.
- c. Expenses associated with maintaining and improving the testing environment.

(Zaremba, 2023)

## 4.2 Calculating ROI

The Return on Investment (ROI) is determined by dividing the savings realized by switching from manual regression tests to automated tests by the cost of adopting the test automation:

$$\text{ROI} = \text{Savings} \div \text{Investment}$$

\* Since ROI is dimensionless, the values for savings and investment can be either in money or time. To simplify the calculation, we'll use minutes as most of our data are time-based.

\*One approach to integrating financial values into our calculations would be to convert dollars into minutes by dividing the cost by the hourly rate of the individual responsible for the test automation and then changing those hours into minutes. Similarly, the financial savings from test automation can be estimated by deducting the investment costs from the savings (both in minutes), converting these minutes into hours, and then multiplying by the hourly rate.

### Cost Savings

Savings refer to the difference in cost between manually running a set of tests and running the same tests automatically multiple times over a certain period.

$$\text{Savings} = (\text{time to run one manual test case} - \text{time to run one automated test case}) * \text{No. of tests} * \text{No. of runs}$$

**Number of Tests:** This refers to the total count of unique test cases or scenarios that need to be executed. For example, if you're testing a software product, you might have 200 unique test cases that validate different functionalities of the product. These might include logging in, executing a specific operation, checking responses to certain inputs, etc. Each one of these scenarios is considered one test.

To calculate the number of tests, we can use the following formula:



$$\text{number of tests} = (\text{final number of tests} - \text{initial number of tests}) / 2$$

In this case, the final number of tests is 200 and the initial number of tests is 0, so the number of tests is:

$$\text{number of tests} = (200 - 0) / 2 = 100$$

**Number of Runs:** This refers to the frequency or how many times each of these test cases is executed. For instance, you might run your entire suite of 200 tests once a day, once a week, or maybe several times a day. This is especially relevant in the context of regression testing where the same test is executed multiple times to validate the behavior of the system. For example, you may run the same 200 test cases every day for a month. In this case, if we consider a month of 30 days, you'd have 30 runs.

This factor significantly influences the outcome of the return on investment. A direct correlation exists between this element and the value of the return on investment. As the number of tests runs increases, so does the return-on-investment value, and the opposite holds true as well.

As illustrated in the image below:

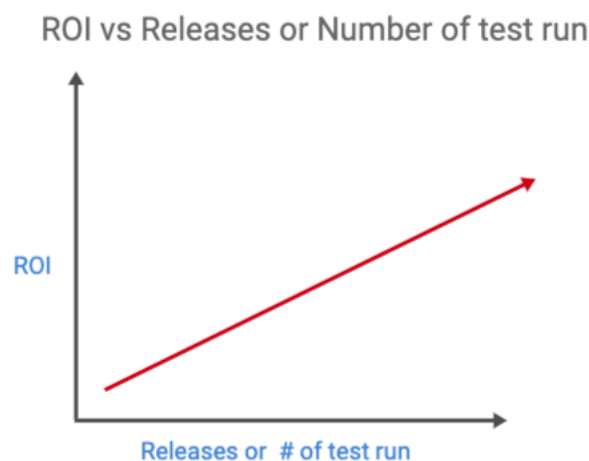


Figure 13: ROI vs Releases or Number of Test run

### Test Automation Investment

The investment is the sum of the fixed and ongoing costs of test automation. This includes the time used to develop and configure the test automation tools or framework, and the time spent writing or maintaining automated tests.

$$\text{Investment} = \text{time to build framework} + (\text{time to code one automated test} * \# \text{ of tests}) + \text{maintenance costs}$$

If there are purchase or licensing expenses, they can be included here after being converted from dollars to minutes using the method mentioned above.

**Maintenance costs** are the expenses incurred in investigating and resolving failed tests (both false positives and real failures). Both unpredictable tests and development bugs contribute to the maintenance cost because an investigation is needed to identify if the feature or the test is at fault. The more frequently tests fail, the longer it takes to reach a break-even point on ROI.

$$\text{Maintenance costs} = \text{maintenance time for one failing test case} * \text{percentage of failing test cases per run} * \# \text{ of test cases} * \# \text{ of runs}$$

(Galeano, 2020)

### 4.3 Enhance ROI in Test Automation

To get the best out of test automation and see a significant return on investment (ROI), you can't just set it up and expect magic. Instead, follow these simplified steps to optimize your test automation ROI:

#### 1- Choose Wisely What to Automate & Prioritize Essential Tasks:

Automation is efficient, but it's not always necessary for every task.

Instead of automating everything, decide what truly needs it. This helps in avoiding unnecessary costs and time spent on tools and systems that might not enhance your ROI.

Let's say you have several test cases ready for automation. Start with the most crucial ones. Randomly selecting test cases can hurt your ROI, especially if delays occur in releasing important features that users are eagerly awaiting.

**2- Use Industry-Standard Tools:**

While having a customized tool sounds good, using well-established test automation tools is often better. By using standard tools, you're more likely to meet regulatory standards and match the expectations of users and stakeholders.

**3- Design Test Cases for Reuse:**

One of the main advantages of automation is that you can write test cases to be used multiple times. When a quality assurance (QA) engineer doesn't need to run the same tests repeatedly, they can focus on creating new ones or exploring other parts of the product. This efficient use of time and resources enhances your ROI.

In summary, making smart choices and efficient use of resources in test automation will steer you toward a better ROI.

(Zaremba, 2023)

## 5 Evaluation based on Case Study

Within the expansive domain of software testing, the debate between the merits of manual versus automated testing remains evergreen. This evaluation ventures into a comprehensive case study, focusing on a specific website as a prototype, to furnish tangible comparisons between these two approaches.

### Objective

The main aim of this evaluation is to elucidate:

- 1-The detailed process involved in manual testing and its associated costs and time constraints.
- 2-The intricacies of automation testing, including its time and monetary implications.
- 3- To measure the Return on Investment (ROI) for Automation testing and when it gains profit based on real-time data from our case study.

A comparative analysis highlighting the key differences in processes, time, and costs between the two testing techniques.

### Conclusion

By delving deep into a real-world case study, this evaluation aspires to present an unbiased picture of the current landscape of manual and automation testing. Such a pragmatic approach aids in making informed decisions regarding the adoption and implementation of either testing strategy in practical scenarios.

In this study, we evaluate six distinct features, which are treated as separate test cases. These features were selected based on their significance as determined by a risk-based analysis. To consolidate the results and establish a comprehensive understanding, an average value will be calculated for all six features. This averaged data will then be incorporated into our designated equation. Our methodology begins with manual testing, followed by automated testing. Conclusively, this chapter will present the Return on Investment (ROI) pertaining to our specific use case.

## 5.1 Manual Testing Evaluation

In the realm of software testing, the manual testing procedure can be elucidated using the following Business Process Model and Notation (BPMN) diagram. Once a feature is developed and ready for testing, the first step entails conducting an analysis to comprehend its functionality and subsequently prioritize it. Following the analysis, test cases are meticulously crafted for the said feature. This involves specifying the prerequisites and delineating the step-by-step procedures for the test. Upon the formulation of the test case, it is then executed to evaluate the feature's functionality. The culmination of this procedure results in the generation of a report. This report serves to either validate that the feature operates as intended or, alternatively, to highlight any bugs or unusual behaviors observed during the testing phase.



Figure 14: BPMN diagram for Manual testing process

During our study, we meticulously documented the time allocation for each phase of the manual testing process.

The results are as follows:

Process (Step)	Timing
<b>Analysis and Prioritization of the Feature</b> <sup>19</sup>	20 min
<b>Test Case Formulation</b>	26 min 40 sec
<b>Test Execution</b>	5 min 40 sec
<b>Report Generation</b>	10 min
<b>TOTAL</b>	<b><u>62 min 20 sec ≈ 62 min</u></b>

Table 4 : Process Timing for Manual Testing

For further clarity, the subsequent section provides a visual representation. Included is a screenshot depicting the 'Functionality Map' document, which details a test case for a specific feature.

US_ID	Category	User stories	Priority	PM_Priority	Notes	TestCase Title	Preconditions	Test Steps:	Writing test case Time	Manual Exe Time (1)	Manual Exe Time (2)	Manual Exe Time (3)
10	shopping cart	As a user I able to see my point status.	1	1		Point Status Display - Positive	1- The user is logged in to their account. 2- The user has earned points through previous activities or transactions.	1- Navigate to the user's profile or points section. 2- Verify that the page displays the user's current point balance in the homepage. 3- Navigate to the userdashboard and also check if it has value.	23 min	4 min	3	4

Figure 15: Record of Testcase in Map of functionality file

<sup>19</sup> The comprehensive procedure for Analysis and Prioritization spanned approximately 2 hours. When this time frame is divided across the six test cases, it results in an average duration of 20 minutes allocated for each individual test case.

## 5.2 Automation Testing Evaluation

Within the scope of our research on software testing methodologies, the workflow for automation testing can be delineated using the accompanying Business Process Model and Notation (BPMN) diagram. This automated approach manifests distinct differences when juxtaposed with the manual testing method. When a feature is matured and positioned for testing, it undergoes an initial analysis phase, mirroring the procedure in manual testing, both in process and in time allocation.

After this, we embark on the task of formulating a feature in Cucumber, adhering to the Behavior-Driven Development (BDD) paradigm, and employing the Gherkin syntax. Following this phase, the 'Page Object' for the respective feature is developed, paving the way for the authoring of the step definition file. During this phase, any encountered errors are addressed and rectified. The process culminates with the execution of the test case. It's worth noting that the reporting segment is not time-stamped, as the report generation is automated and produced contemporaneously.

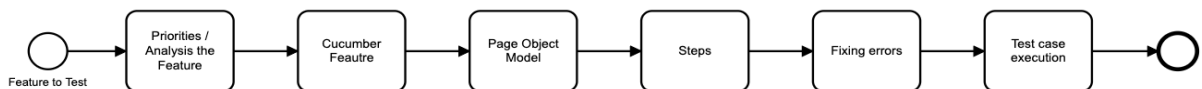


Figure 16: BPMN diagram for Automation testing process

It is imperative to emphasize that the sequence delineated in the BPMN may vary when applied to different automated testing contexts. This variance is contingent upon the specific Automation testing framework employed. For instance, if an automation test engineer opts for a framework other than Cucumber, the testing process and its sequence would likely undergo alterations.

During our study, we meticulously documented the time allocation for each phase of the manual testing process.

The results are as follows:

Process (Step)	Timing
Analysis and Prioritization of the Feature	20 min
Cucumber Feature	9 min 25 sec
Page Object	20 min 20 sec
Steps/test	24 min 20 sec
Fixing Error	10 min
Test Execution	≈ 8 sec
<b>TOTAL</b>	<b><u>84 min 12 sec ≈ 84 min</u></b>

Table 5: Process Timing for Automation Testing

At an initial assessment, the duration required to test a singular feature, termed "Test-case", within the automated testing paradigm appears to exceed that of its counterpart in manual testing. However, it is crucial not to make premature conclusions based solely on these preliminary observations. To provide a comprehensive comparison and derive a conclusive judgment, it is essential to conduct a Return on Investment (ROI) analysis and evaluate the resultant data.

For further clarity, the subsequent section provides a visual representation. Included is a screenshot depicting the 'Functionality Map' document, which details a test case for a specific feature in Automation testing and recording the time for every step.

US_ID	Category	User stories	Priority	PM_Priority	cucumber/feature	Page Object	Steps/test	learning Time	errors	execution (1)	execution (2)	execution (3)	Automation Done
0	Load	As a user, I can open The VW-Club website	1		3 min		15			6 Sec			ok
1	login	As a user I able to Login with sales role.	1	1	15 min	37 min	40 min		10 min error fixing	9 sec			ok
2		As a user I able to Login with Service role.	2	1									
10	shopping cart	As a user I able to see my point status.	1	1	5:30 min		0 5 min	5 + 3 + 8	10 min error fixing	10 sec			ok
14		As a user I able to show the shopping cart items.	1	1	8 min 10 min	9 min loginobject page 7 min home page object 6 min dashboard page object	10			8 sec			ok
32		As a user I able to redeem my points "tankkarten"	1	1	15 min	15 min home page object 20 min point redemption page object 18 min tankkarte redemption page object	60 min		30 min	6 Sec 12 sec			ok

Table 6 : Record of Automation Testcase in Map of functionality file



## 5.3 ROI Evaluation

From the preceding chapter, we acquired a comprehensive understanding of the Return on Investment (ROI) and the associated equations utilized for its computation. We have identified the following foundational equations. The subsequent step entails inputting our specific data into these formulas to calculate the ROI:

$$\text{ROI} = \text{Savings} \div \text{Investment}$$

$$\text{Savings} = (\text{time to run one manual test case} - \text{time to run one automated test case}) * \# \text{ of tests} * \# \text{ of runs}$$

$$\text{Investment} = \text{time to build framework} + (\text{time to code one automated test} * \# \text{ of tests}) + \text{maintenance costs}$$

$$\text{Maintenance costs} = \text{maintenance time for one failing test case} * \text{percentage of failing test cases per run} * \# \text{ of test cases} * \# \text{ of runs}$$

In the subsequent table, we will populate each variable within the equation using empirical data derived from our experimental activities. This will facilitate the computation of ROI across various scenarios.

### 5.3.1 The Real Case

Variable	Value
Time to run one manual test case	5 min 40 sec $\approx$ 6 min
Time to run one automated test case	8 sec
Number of Tests	78 <sup>20</sup>
Number of Run <sup>21</sup>	30
Time to build framework	60 min
Time to code one automated test	54 min 5 sec $\approx$ 54
Maintenance time for one failing test case	10
Percentage of failing test cases per run	%25

Table 7: Real Case Values to calculate ROI

**Savings** = (5 min 40 sec – 8 sec) \* 78 \* 30 = 5 min 32 sec \* 78 \* 30 = 12948

**Investment** = 60 min + (54 \* 78) + 5,850 = 60 + 4212 + 5,850 = 10,122

**Maintenance costs** = 10 \* .25 \* 78 \* 30 = 5,850

**ROI** = 12948 ÷ 10,122 = 1.28 \* 100 = 128%

The observed values have been derived based on empirical analysis and experimental runs. As indicated in the result findings, the Return on Investment (ROI) has been calculated to be 128%. This suggests that there would be a positive return on the investment after a duration of 30 days. This duration corresponds to the period over which the regression test was run daily. Specifically, one regression test was conducted every morning, encompassing all test cases.

We shall now conduct a comparative analysis by varying specific parameters. Specifically, the variable 'number of runs' will be adjusted to 15 and subsequently to 60, followed by a recalibration of the Return on Investment (ROI) based on these modifications.

<sup>20</sup> 39 features \* 4 Testcase = 156 Testcase as we described before (156 – 0) / 2 = 78

<sup>21</sup> Every day at the morning run a regression test.

Number of runs	ROI
15	89%
30	128%
60	162%

Table 8: ROI Comparison

Upon examination of the ROI values, it was observed that when the number of runs was restricted to 15, equivalent to a period of two weeks, the desired ROI value was not attained and was notably low. In contrast, an augmentation in the number of runs led to an escalation in the ROI values, as illustrated in the subsequent diagram.

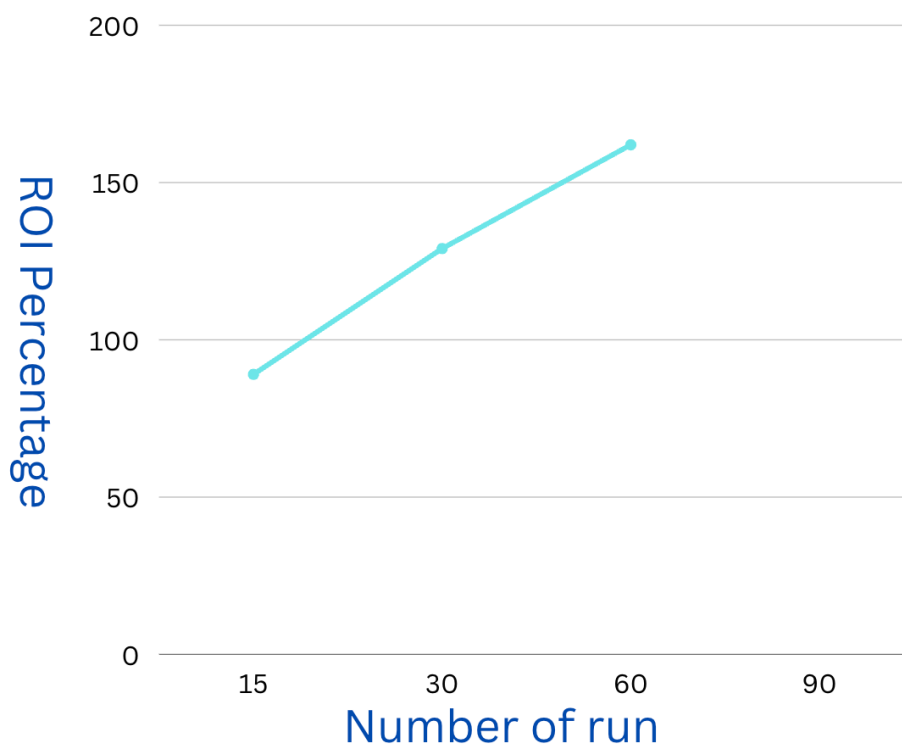


Figure 17: ROI vs Number of run "Real Values"

### 5.3.2 Changing Variables

For a comprehensive understanding, we will make an adjustment to the variables.

#### 5.3.2.1 *Percentage of failing test cases per run.*

The variable 'Percentage of failing test cases per run', changing its value from 25% to 35%. After this modification, the ROI was recalculated<sup>22</sup>. The derived ROI value was found to be 1.03, equivalent to 103%. While this is lower than the previously computed ROI, but we still gain a return on investment. It is crucial to underscore that, over time, this percentage is anticipated to diminish due to the steepening of the learning curve resulting from cumulative future practices.

#### 5.3.2.2 *'Maintenance time for one failing test case*

The variable 'Maintenance time for one failing test case', herein referred to as 'fixing time', was adjusted, setting its duration to 20 minutes from the initial 10 minutes. The subsequent recalculation of the ROI yielded a value of 81%. This is sub-optimal as the desired return on investment was not achieved. However, it's worth noting that this value pertains to only 30 runs, equivalent to a month. To further understand the impact over an extended duration, the ROI was recalculated for different numbers of runs: 60 runs (2 months), 90 runs (3 months), and 120 runs (4 months). It was discerned that a positive return on investment would be realized in the fourth month, with the ROI being 101% for that duration. The detailed outcomes are tabulated in the subsequent tables and visually represented in the accompanying chart.

Number of runs	ROI
30	81%
60	93%
90	98%
120	101%
150	103%

Table 9 : ROI Comparison when Fixing time 20 min

<sup>22</sup> All Calculation are attached in the Map of functionality excel sheet under sheet "ROI."

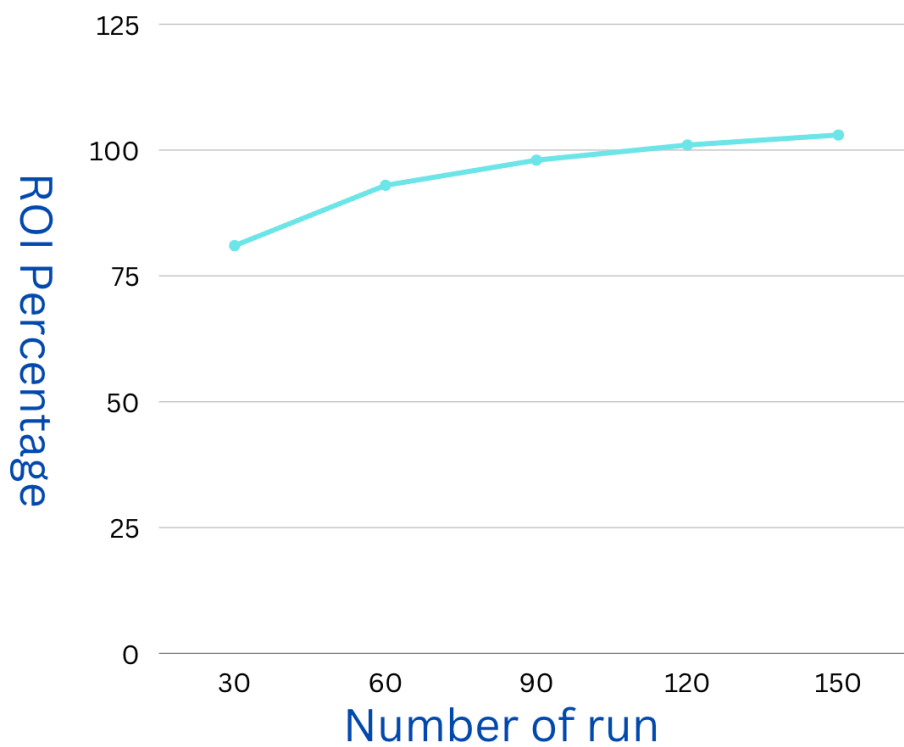


Figure 18: ROI vs Number of run "Fixing time 20 min"

## 6 Conclusion

As we venture into the digital age, the importance of effective software testing cannot be emphasized enough. The main objective of this master's thesis was to ascertain the return on investment (ROI) for automation software testing within an end-to-end system, based on a real-world case study. The investigation provided critical insights and the conclusions derived are as follows:

### **1.The Scale and Frequency of Projects:**

Automation testing has established its effectiveness predominantly for large-scale projects. This is not necessarily because of the complexity or the volume of the project, but due to the recurrence of testing required in scenarios that have multiple releases and versions. Small-scale projects, owing to their limited scope, often do not justify the initial investment needed for setting up automation tests. Moreover, for projects with frequent releases, the necessity of rerunning regression tests multiplies. Manual execution in such contexts is not only tedious but can also introduce errors due to oversight. Thus, in cases with frequent releases and versions, the ROI for automation testing can be significantly high. Automation ensures consistency and swift feedback, making it an invaluable asset for such projects.

### **2.Skill Set of the Test Automation Engineer:**

The efficiency of the automation process is, to a large extent, contingent on the proficiency of the Test Automation Engineer. Hiring an experienced individual, equipped with strong programming acumen and analytical skills, plays a pivotal role in maximizing ROI. Not only does their expertise minimize the probability of test failures, but it also ensures that any failures are promptly addressed, leading to a more efficient testing lifecycle.

### **3.Reusability in Test Frameworks:**

During this case study, a noticeable trend was the repetitive nature of code in test cases. This redundancy, if unchecked, can bloat the test suite, making maintenance challenging. It underscores the significance of reusability while designing and developing testing frameworks. By capitalizing on reusability, a Test Automation Engineer can ensure

concise, modular, and maintainable test suites, which in turn contribute positively to ROI.

#### **4.Choosing the Right Testing Framework:**

The adage "Don't reinvent the wheel" rings particularly true for automation testing. Leveraging pre-existing, proven testing frameworks can dramatically reduce setup time and potential errors. In our investigation, while Selenium was explored, it was observed that creating everything from scratch can negatively impact ROI, especially in projects where time-to-market is crucial. The need for a more out-of-the-box solution is evident in such cases.

#### **5.Relevance of Cucumber and Alternatives:**

Within the scope of this project, Cucumber was utilized for Behavior Driven Development (BDD). However, an unforeseen revelation was the non-technical stakeholders' indifference towards its outcomes. Contrary to popular belief that BDD helps bridge the communication gap between tech and non-tech stakeholders, in this specific context, it was perceived as an added layer of complexity. Based on this experience, it may be more ROI-effective to consider alternative libraries like chai or test café. The choice of tool should always align with the project's requirements and stakeholder preferences.

#### **6.Correlation between Test Runs and ROI:**

A direct correlation was discerned between the frequency of automated test runs and the ROI. As the number of runs increased, so did the value derive from automation, both in terms of cost and time savings. This relationship is intuitive since the initial investment in automation setup gets distributed over more runs, thus decreasing the cost per run over time. Conversely, fewer runs might not justify the initial setup costs, leading to a lower ROI.

#### **7.The Symbiosis of Automation and Manual Testing:**

A vital takeaway from this research is the understanding that automation, despite its efficiencies, is not an absolute solution. Certain test cases are best addressed manually, leveraging human intuition and creativity. The idea that manual testing might become

redundant is misconceived. Automation serves as an adjunct to manual testing, addressing its limitations. Both methodologies have distinct roles in the software testing landscape, and a balanced blend of the two is essential for comprehensive testing.

In conclusion, while automation software testing undeniably offers myriad advantages, its true value can only be harnessed by astute decisions related to project scale, personnel expertise, framework selection, and test design principles. Through this case study, it is hoped that future projects can be better informed, ensuring a high ROI and successful project outcomes.



## 7 AI Tools Protocol:

In the process of constructing this thesis, I integrated various Artificial Intelligence (AI) tools to enhance the quality, coherence, and uniqueness of the content. Chief among these tools were ChatGPT (GPT-4) and Google's Bard. Below is an overview of these tools and their utility in my research.

### 7.1 ChatGPT - GPT-4 by Open AI

#### **Overview:**

ChatGPT is based on the GPT (Generative Pre-trained Transformer) architecture, a state-of-the-art machine learning model designed for multiple natural language processing tasks. The version I utilized was GPT-4, a paid variant that offers more advanced capabilities.

#### **Capabilities:**

Text generation and completion.

Question answering.

Text summarization and rephrasing.

Language translation.

#### **Usage in this Thesis:**

Text Refinement: Used ChatGPT to reformat passages from reference materials to generate unique content for the thesis.

Scientific Text Generation: Leveraged its capabilities to generate or adjust sections of the thesis to maintain a scientific tone and clarity.

Linguistic Corrections: Leveraged for routine grammar and spelling checks.

## 7.2 Bard by Google

### Overview:

Bard is an AI tool developed by Google, designed for specific textual tasks, ensuring more refined and targeted results.

### Capabilities:

Advanced grammar checking.  
Text optimization and rephrasing.  
Content-specific insights and adjustments.

### Usage in this Thesis:

Grammar and Syntax Check: Utilized Bard to rectify grammar and syntax errors in the thesis, ensuring impeccable language quality.

Text Reformatting: Leveraged Bard to transform referenced material into original content, distinct from source texts.

## 7.3 Workflow Integration of AI Tools:

### 1.Initial Content Drafting:

I began by manually drafting sections of the thesis, ensuring that the primary ideas, arguments, and conclusions were well-established.

### 2.AI-Assisted Refinement:

Used ChatGPT or Bard to refine the drafted sections. Depending on the specific need — whether it was grammar check, rephrasing, or turning the text more scientific — the appropriate AI tool was chosen.

### 3.Text Reformatting for Originality:

Extracted content from reference books and papers, and then employed the AI tools, especially Bard, to ensure that the transformed content was unique and plagiarism-free.

### 4.Scientific Tone Adjustment:

Sections of the thesis that needed to adhere strictly to a scientific tone were run through ChatGPT. This ensured that the language, phrasing, and content aligned with academic and scientific standards.

## 7.4 Illustrative Examples:

The following are demonstrative examples of how these tools were applied, as can be seen in the accompanying pictures.



Write as a Conclusion for master thesis title [Return on Investment for Automation software testing in End-to-End System "Case Study"] the following points in a scientific context in at least 2 pages:

- 1- the Automation Testing is usefull for a large scale project not for a small projects , and also very usefull for the projects which has multibale releases and multibale versions , is this case the ROI for the Automation testing will be High because we need to run multibale time the Regression test.
- 2- Its very importat to hire an already expereence Test Automation enginer with high knowledge in programing and has a good programic skills beside analytical skills to increase the ROI at to decrease the percentage of failure and also to inhance the time of fixing this failures.
- 3- As an Experience during the Project its very apparent that the code will be decreased during the testcases because it'll be duplicated and Here should the Test Automation Enginer depend on the Reusability when he create design and develop his Testing framework.
- 4- Its very important to use already Testing Framework and not reenvent the wheel , i'm not a big fan of Selenium because in selenium i need to create everything from scratch which will has a bad impact to the ROI of Automation testing at the end .
- 5- I used Cucumber in the project but really i advice to not use it because it waste the time and the stackholders "non-Technical people" dont want to see it , at least from my experience. also if we dont use it we can use chai or testcaffe lib , and it'll diffenently increase the ROI.
- 6- The apparent relationship between the number of runs and the number of ROI , The higher the run, the higher the ROI value, and vice versa

Figure 19 : Prompt to ChatGPT ex1



#### Conclusion:



In this master thesis, we conducted a comprehensive investigation into the Return on Investment (ROI) for Automation Testing in the context of an End-to-End System, with a focus on a real-world case study. The findings and analysis have shed light on several key factors that influence the ROI of Automation Testing, which we will summarize in this conclusion.

#### 1. Automation Testing for Large-Scale and Multi-Release Projects:

Our research has demonstrated that Automation Testing is particularly valuable for large-scale projects and those with multiple releases and versions. In such cases, Regression Testing needs to be performed repeatedly to ensure software stability and compatibility. Automation Testing significantly reduces the time and effort required for repetitive testing, thereby increasing ROI. However, for smaller projects with limited testing needs, the upfront investment in Automation may not be justified, as manual testing might be more cost-effective.

#### 2. The Importance of Skilled Test Automation Engineers:

Our study highlighted the critical role of skilled Test Automation Engineers with strong programming knowledge and analytical skills. Hiring experienced professionals in this domain can lead to higher ROI by minimizing the occurrence of failures and effectively addressing any issues that arise during testing. The expertise of these engineers enables them to design and develop reusable testing frameworks, which leads to more efficient and cost-effective testing processes.

Figure 20: Chatgpt answer ex.1



Reformat, rewrite and correct the following text also write it in a simple and right context: CI addresses software integration problems effectively and efficiently. As the application under development becomes complex and the code base involves multiple components, it is vital to get feedback at every code change. The CI development practice enables engineers to integrate their work frequently and detect errors instantly through code compilation and building, and running unit tests. Each engineer usually commits at least once a day and hence encourages breaking down big changes into logical chunks of code. Building and testing of the product often occur numerous times a day in an automated manner leading to a much higher confidence level in the state of the project.

The CI server typically keeps polling the version control repository every minute or so and runs a preconfigured build script by fetching the latest version of the source files. The CI software has a dashboard and offers options to kick off various stages of the build process at will. The CI software is also capable of messaging the stakeholders of the current build via emails and/or instant messaging platforms. It is generally considered a good practice to perform database setup as well as part of the CI build script to ensure the coherence of the whole software application.

Figure 21: Prompt to ChatGPT ex.2



Continuous Integration (CI) effectively tackles the challenges of software integration, particularly as applications become more complex and their codebases expand with multiple components. CI allows engineers to frequently incorporate their work and promptly detect errors through code compilation, building, and unit tests. Engineers often commit daily, promoting the breakdown of larger changes into manageable chunks of code. The frequent, automated building and testing of the product throughout the day inspires higher confidence in the project's status.

The CI server routinely checks the version control repository, typically every minute or so. It runs a preconfigured build script after fetching the latest version of the source files. The CI software, equipped with a dashboard, provides options to initiate various stages of the build process. The software can also notify stakeholders about the current build status through emails or instant messaging platforms. It's generally a good practice to include database setup in the CI build script to maintain the coherence of the entire software application.

Figure 22 : Chatgpt answer ex.2

The integration of AI tools like ChatGPT and Bard has revolutionized the process of scientific thesis writing. They not only streamline the content creation process but also ensure the highest standard of linguistic and scientific quality. Through careful and strategic use, these tools can significantly elevate the caliber of academic research and writing.

## Literature

[ISTQB. 1 July 2021. *Certified Tester Foundation Level (CTFL) Syllabus*. s.l. : International Software Testing Qualifications Board, 1 July 2021.

ISTQB. 2016. *Certified Tester Advanced Level Syllabus Test Automation Engineer*. s.l. : International Software Testing Qualifications Board, 2016.

Manfred Baumgartner, Thomas Steirer, Marc-Florian Wendland, Stefan Gwihs, Julian Hartner, Richard Seidl. 2022. *Test Automation Fundamentals*. Heidelberg : Rocky Nook, 2022. 978-3-96910-870-3.

Jose, . 2021. *Test Automation A manager's guide*. Swindon : BCS, THE CHARTERED INSTITUTE FOR IT, 2021. 978-1-78017-5461.

Shukla, . 2021. *Practical WebDriverIO*. Mandla : Apress, 2021. 978-1-4842-6660-1.

Sambamurthy, . 2023. *Test Automation Engineering Handbook*. Birmingham : Packt Publishing, 2023. 978-1-80461-549-2.

Matt Wynne, Aslak Hellesoy, Steve Tooke. 2017. *The Cucumber Book, 2nd Edition*. s.l. : Pragmatic Bookshelf, 2017.

Goldberg, . 2022. *Learning TypeScript*. s.l. : O'Reilly Media, Inc., 2022. 978-1-098-11033-8.

## Online-Resources

**Ahmad, Nazneen.** How To Write Test Cases - A Complete Guide With Examples And Best Practices. *Lambdatest*. [Online] [Cited: 6 5, 2023.] <https://www.lambdatest.com/learning-hub/test-case>.

**Hamilton, . 2023.** Manual Testing Tutorial: What is, Types, Concepts. *guru99*. [Online] 8 4 2023. [Cited: 4 6 2023.] <https://www.guru99.com/manual-testing.html>.

**Rehkopf, Max.** User stories with examples and a template. *Atlassian*. [Online] Atlassian. [Cited: June 5, 2023.] <https://www.atlassian.com/agile/project-management/user-stories#:~:text=software%20user's%20perspective,-,A%20user%20story%20is%20an%20informal%2C%20general%20explanation%20of%20a,value%20back%20to%20the%20customer..>

**Wohnert, Melanie. 2021.** BLACK-BOX-TEST. *software-testing.academy*. [Online] 25. 1 2021. [Zitat vom: 4. 6 2023.] <https://www.software-testing.academy/black-box-test.html>.

**Hamilton, . 2023.** Risk Based Testing: Approach, Matrix, Process & Examples. *guru99*. [Online] 13 May 2023. [Cited: 9 jun 2023.] <https://www.guru99.com/risk-based-testing.html>.

**S., STH team members – Gayathri Subrahmanyam and Swati. 2023.** The Ultimate Guide To Risk Based Testing: Risk Management In Software Testing. *softwaretestinghelp*. [Online] 4 May 2023. [Cited: 9 Jun 2023.] <https://www.softwaretestinghelp.com/risk-management-during-test-planning-risk-based-testing/>.

**Shaumik Daityari, Community Contributor and Shreya Bose, Technical Content Writer. 2022.** Regression Testing: A Detailed Guide. *browserstack*. [Online] 9 May 2022. [Cited: 12 Juni 2023.] <https://www.browserstack.com/guide/regression-testing>.

**Kitakabee. 2023.** browserstack. *Automated Regression Testing: A Detailed Guide*. [Online] 10 February 2023. [Cited: 12 Juni 2023.] <https://www.browserstack.com/guide/automated-regression-testing>.

**Hamilton, . 2023.** What is Smoke Testing. *guru99*. [Online] 13 May 2023. [Cited: 12 Juni 2023.] <https://www.guru99.com/smoke-testing.html>.

**Hamilton. 2023.** What is Non Functional Testing. *guru99*. [Online] 12 May 2023. [Cited: 12 Juni 2023.] <https://www.guru99.com/non-functional-testing.html>.

**OpenJs. 2023.** <https://webdriver.io/docs/why-webdriverio/>. *Why Webdriver.IO?* [Online] OpenJs Foundation, 2023. <https://webdriver.io/docs/why-webdriverio/>.

**Tiwari, . 2023.** WebdriverIO Tutorial: Getting started with Test Automation using Selenium and Appium. *browserstack*. [Online] 20 March 2023. [Cited: 16 June 2023.] <https://www.browserstack.com/guide/webdriverio-tutorial-for-selenium-automation>.

**Rungta, . 2023.** Page Object Model (POM) & Page Factory in Selenium. *GURU99*. [Online] 6 May 2023. [Cited: 20 June 2023.] <https://www.guru99.com/page-object-model-pom-page-factory-in-selenium-ultimate-guide.html>.

**testim. 2020.** Testim. *test-automation-roi*. [Online] April 2020. [Cited: 18 Juli 2023.] <https://www.testim.io/blog/test-automation-roi/>.

**Zaremba, . 2023.** How to Calculate ROI Test Automation And What You Need? *testsigma*. [Online] 7 June 2023. [Cited: 19 June 2023.] <https://testsigma.com/blog/roi-test-automation/>.

**Galeano, . 2020.** What is the ROI of my test automation? *medium*. [Online] 7 April 2020. [Cited: 19 June 2023.] <https://medium.com/slalom-build/what-is-the-roi-of-my-test-automation-10ae7bf0d9ed>.



## Appendix

### Mind map at Miro board

[https://miro.com/app/board/uXjVPuNPCLw=?share\\_link\\_id=124042848617](https://miro.com/app/board/uXjVPuNPCLw=?share_link_id=124042848617)

Please access my mind map, it's for anyone on the internet, No sign up required.

Link for Code

[https://fhbrandenburgde-my.sharepoint.com/:f:/g/personal/mohamed\\_th-brandenburg\\_de/EtLPZvP9fJxDvT7JBCmG1NEB8YVVg6AkfEAnAbZY2--TAA?e=f3Sz5P](https://fhbrandenburgde-my.sharepoint.com/:f:/g/personal/mohamed_th-brandenburg_de/EtLPZvP9fJxDvT7JBCmG1NEB8YVVg6AkfEAnAbZY2--TAA?e=f3Sz5P)

Link to Map of functionality

[https://fhbrandenburgde-my.sharepoint.com/:f:/g/personal/mohamed\\_th-brandenburg\\_de/EtLPZvP9fJxDvT7JBCmG1NEB8YVVg6AkfEAnAbZY2--TAA?e=f3Sz5P](https://fhbrandenburgde-my.sharepoint.com/:f:/g/personal/mohamed_th-brandenburg_de/EtLPZvP9fJxDvT7JBCmG1NEB8YVVg6AkfEAnAbZY2--TAA?e=f3Sz5P)