1 2 9 0

UNIVERSIDADE Đ
COIMBRA

Pedro Tiago dos Santos Marques

OPTIMIZATION OF APPROVAL TIME IN WEB UI TESTS

Dissertation in the context of the Master in Informatics Engineering, specialization in Software Engineering, advised by Professor Vasco Pereira and Eng. Emanuel Teixeira and presented to the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra.

July of 2023

1 2 9 0

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE D
COIMBRA

DEPARTMENT OF INFORMATICS ENGINEERING

Pedro Tiago dos Santos Marques

Optimization of approval time in Web UI tests

Dissertation in the context of the Master in Informatics Engineering, specialization in Software Engineering, advised by Prof. Vasco Pereira and Eng. Emanuel Teixeira and presented to the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra.

July 2023

1 2 9 0

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE Ð
COIMBRA

DEPARTAMENTO DE ENGENHARIA INFORMÁTICA

Pedro Tiago dos Santos Marques

Optimização do tempo de aprovação em testes Web UI

Dissertação no âmbito do Mestrado em Engenharia Informática, especialização em Engenharia de Software, orientada pelo Professor Doutor Vasco Pereira e Eng. Emanuel Teixeira e apresentada ao Departamento de Engenharia Informática da Faculdade de Ciências e Tecnologia da Universidade de Coimbra.

Julho 2023

Acknowledgements

I want to express my sincere gratitude to my Dissertation advisors, Prof. Vasco Pereira and Eng. Emanuel Teixeira, for their guidance and support, and without whom this work would not be possible. Their invaluable suggestions, expertise, and perseverance were crucial to the accomplishment of this Dissertation.

To Stratio Automotive, where I got to know amazing talented people and for providing me with excellent work conditions and resources.

I would also like to thank all my friends and colleagues from DEI, especially Francisco Bugalho, Diogo Filipe, Nuno Silva, José Reis, José Gomes, Bruno Gandres, Henrique Teixeira (and so many more!), for all the fun memories, advice and support!

And to my family, for their unconditional support. For the everlasting encouragement every day by my mother and father. A special thank you to my brother Miguel Marques, for always being there when I needed and for pushing me to be better.

Abstract

As software progresses at an outstanding pace, the need for efficient and reliable user experiences while using a variety of software applications grows even stronger. Component testing and end-to-end (E2E) testing are two crucial types of testing that are used to guarantee the reliability and quality of software systems. E2E testing simulates the end-user experience and traverses the system from start to finish to ensure the correctness of the functionalities according to the requirements. Component testing aims to isolate components or modules of the system individually and make sure they are working as intended.

This Dissertation explores the quality process at Stratio Automotive and proposes various improvements through automated testing in order to shorten the approval time of Web UI tests for the Foresight Platform, which is a predictive maintenance dashboard the company develops. A study is conducted to extend the current state of art regarding testing tools for automation and functional testing. Moreover, a detailed examination is performed to understand not only the potential of each automation driver but also as a development tool that easily integrates into the CI pipeline of Stratio Automotive. The presented work contributes with changes to the validation process of the product, rendering the automated tests a necessity to launch new versions of the product continuously. To that end, we identify three situations where the company spends too many resources performing manual Web UI tests and propose significant changes within the current quality process to include different technologies.

The obtained results show considerate improvements of up to ten times faster approval times for Web UI tests. Regarding the Foresight Platform, the present work achieves code coverage for components of 83.69% and test coverage of 98% of all functionalities. Finally, a test report is included with the defects prevented and identified by the employed testing strategies.

Keywords

End-to-End Testing, Component Testing, Quality.

Resumo

À medida que *software* progride a um ritmo significativo, surge a necessidade de experiências eficientes e aprazíveis aquando da utilização de diferentes aplicações. Testes de componente e *end-to-end* (E2E) são dois tipos cruciais de testagem que são utilizados para garantir confiabilidade e qualidade de sistemas de *software*. Testes E2E simulam a experiência do utilizador final ao percorrer o sistema do início ao fim, para garantir bom funcionamento do mesmo. Testes de componente, por outro lado, têm como objetivo isolar componentes ou módulos do sistema de maneira individual, garantindo que são executados como pretendido.

Esta Dissertação explora o processo de qualidade da Stratio Automotive e propõe várias melhorias através da automatização de testes. O principal objetivo é reduzir o tempo de aprovação de testes *Web UI* do Foresight Platform, que é uma *dashboard* de manutenção preditiva que a empresa desenvolve. É realizado um estudo para estender o estado da arte para ferramentas de automatização de testes e testes funcionais. Efetua-se também uma análise detalhada que compreende o potencial de cada ferramenta de testes e como se pode integrar na *pipeline CI* da Stratio Automotive. O presente trabalho contribui com alterações ao processo de validação do produto, tornando a etapa de testes automatizados uma obrigação para lançar novas versões do produto continuamente. Para esse fim, são identificadas três situações onde a empresa gasta demasiados recursos a realizar testes à *Web UI* manualmente, onde propomos alterações significativas ao atual processo de qualidade da empresa.

Os resultados obtidos apresentam melhorias notáveis, com tempos de aprovação até dez vezes mais rápidos para testes *Web UI*. Relativamente à Foresight Platform, o presente trabalho alcança cobertura de código de 83.69% e cobertura de testes de 98% para todas as funcionalidades. Finalmente, um relatório de testes é incluído com os defeitos prevenidos e identificados através das estratégias de teste utilizadas.

Palavras-Chave

Testes *End-to-End*, Testes de componente, Qualidade.

Contents

Acronyms

API Application Programming Interface.

BDD Behavior Driven Development.

CD Continuous Delivery/Deployment.

CI Continuous Integration.

CSS Cascading Style Sheets.

DOM Document Object Model.

E2E End-to-End.

HTML Hypertext Markup Language.

PO Product Owner.

POC Proof of Concept.

POM Page Object Model.

QA Quality Assurance.

TDD Test Driven Development.

UAT User Acceptance Test.

UI User Interface.

Web UI Web User Interface.

List of Figures

List of Tables

Chapter 1

Introduction

The present document outlines both the theoretical and practical work consummated during the first and second semester of the academic year 2022/2023 and represents the final Dissertation. The following sections are meant to illustrate the theme of research, the work plan for the year 2022/2023, goals, objectives, risks and methodology.

The curricular internship titled "Optimization of approval time in Web UI tests", hosted by Stratio Automotive, is included in the academic program of the Master's in Informatics Engineering in the Software Engineering branch.

1.1 Context and Motivation

The primary mode of interaction with the digital world is **software**. It consists on a set of programs, procedures and routines that instruct a computer to perform a task [1]. Nevertheless, software is not immune to defects that may constrain its functionality, compromising quality and the user experience.

Software testing is an important process to develop good projects/systems that correlate with user needs and expectations. The main objective is to find **defects**, which cause the system to migrate from a correct state to an incorrect state, and fix them before delivering the product. The work developed in this Dissertation involves only **functional Web User Interface (Web UI) software testing**, so software development and testing will refer to Web UI software unless otherwise stated.

As an example, users anticipate a smooth experience when browsing the Internet since they **expect the website to work flawlessly**. To better understand the need for Web UI testing, let us consider a user accessing the Stratio Foresight Platform, which is a web application offered by Stratio Automotive. After the page loads the main interface, the user can interact with a range of features within the front-end without noticing any issues or defects. This ideal scenario is the result of extensive testing and bug fixes before deploying a stable version to the production environment. During development, it is expected that some components do not work as intended or that the overall experience is flawed, rendering the importance of having a well defined test strategy for components in place. This is known as component testing, which mounts components individually and test them independently from the system. Additionally, another techniques like End-to-End (E2E) testing may be used to validate the entire system from start to finish to guarantee proper functioning. By testing all of the **components** and **interactions** between them, the development team can be relatively certain that the current build is stable, without major defects. The typical workflow of component and E2E tests is illustrated in Figure 1.1.

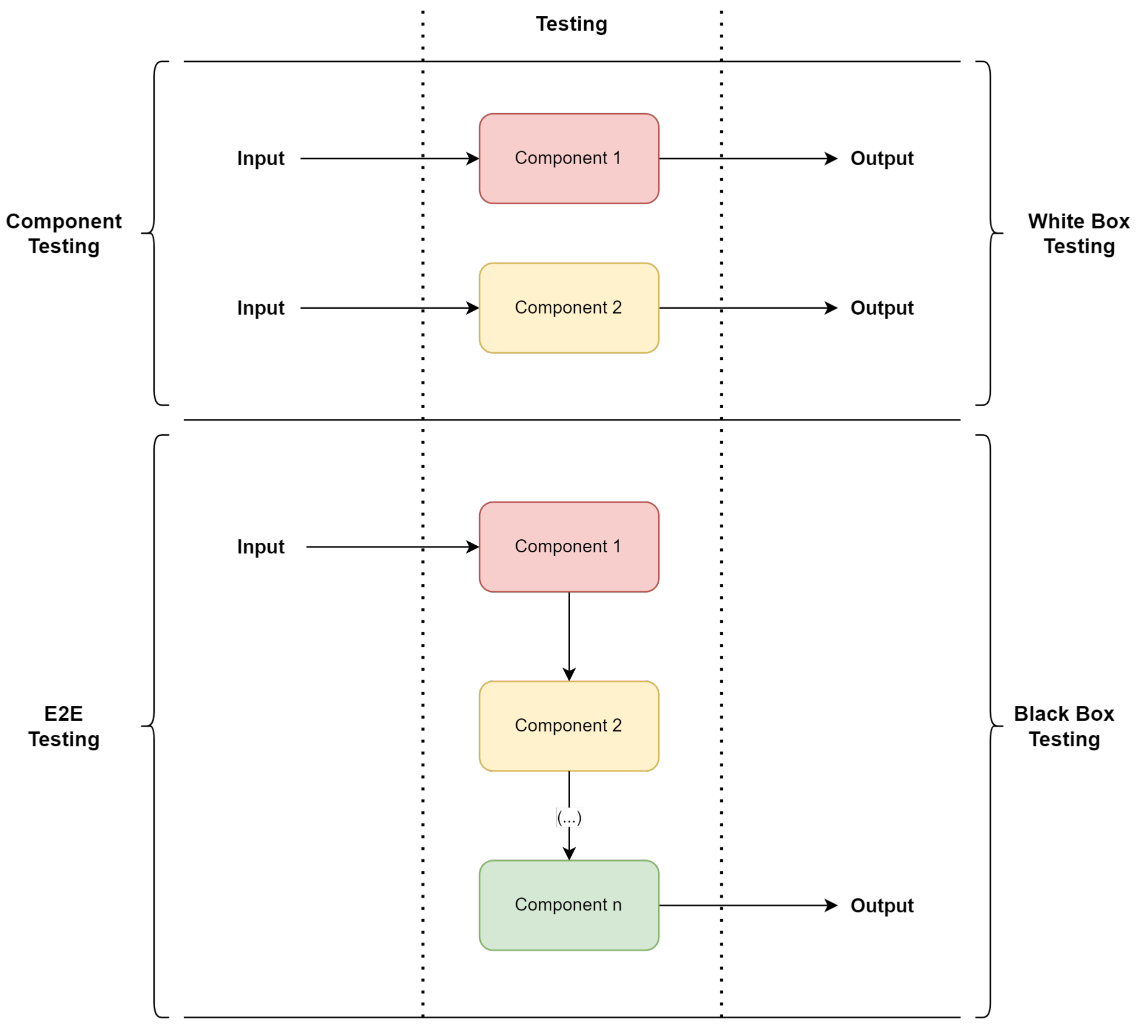


Figure 1.1: Schematic representation of the workflow of component and E2E testing.

Currently, multiple tools such as Cypress [2] and Playwright [3] exist to simulate user's behaviors through Web UI testing, since they aim to mimic the actions of potential users on a specific website to **detect defects**. Also, there are tools like Storybook [4] to individually test the components that compose a page, granting a more granular and individual approach to Web UI testing.

Stratio Automotive [5] offers a predictive vehicle fleet maintenance platform with the main objective of preventing breakdowns, reducing costs, and eliminating downtime. The company combines data processing at scale with sophisticated machine learning algorithms to prevent breakdowns in vehicle fleets. This market is huge given that, in order to maintain transport efficiency, undesired anomalies cannot occur.

Additionally, Stratio Automotive owns and develops a Web UI solution for its customers. A big chunk of the continuous development involves guaranteeing that the produced software achieves maximum quality and a smooth experience for all users, without defects. This Dissertation will present a solution that **reduces the approval time of Web UI tests** for Stratio Automotive. In other words, through automated testing, the goal is to lessen the time it takes to launch a new product iteration to the clients. To achieve this goal, it is necessary to analyse and understand the current Continuous Integration (CI) approach and quality process of the company and propose changes.

At the present, the quality process of Stratio Automotive has the opportunity to evolve and automate many operations that are currently done **manually** on the Web UI platform. In order to approve a new version of the software for deployment, the product team has to **manually** validate the whole Web UI platform. This process can take up to **four hours** depending on how detailed and extensive the analysis is, which can dramatically increase the approval time of new versions. Additionally, the validation of new functionalities and bug approvals are done by hand. All this time starts to add up and, for each release, the product team spends many hours validating the changes made by the software department.

To tackle this process, the adoption of Kanban [6] as an agile methodology will be put into practice in order to visualise workload and limit work in process. After attaining continuous improvement, it is important to **evaluate the current quality process** and understand where it is possible to integrate new changes. The first evolution with immediate impact is the creation of a battery of E2E tests that cover the functionalities manually validated by the product team of the Web UI platform before each deployment. The other natural improvement is to associate component tests and a new E2E test for each new functionality or bug fix developed during each sprint. This advancement guarantees that the developer can receive feedback instantly and correct bugs earlier. Regarding the current CI approach, Stratio Automotive's development team uses GitLab CI [7] to test, build, and deploy new versions of their product. This forms the basis to implement automated testing methodologies.

1.2 Objectives

The main objective is to optimize the approval time of Web UI tests by transforming the quality process at Stratio Automotive. The main effort consists of introducing changes to aspects that are currently delaying testing times through the integration of different tools and the composition of different types of tests. This improvement not only reduces conflicts like excessive back and forth messages between the product and development team, but also proposes a new mindset, testing software, and a different acceptance process. To accomplish the proposed solution, the main objectives of this Dissertation are as follows:

1. Study the theoretical background of the quality assurance paradigm with main focus in component and E2E testing;
2. Research various tools to achieve the testing methodologies mentioned in point 1;
3. Study the CI/CD approach and quality assurance model of Stratio Automotive regarding the approval process in Web UI tests;
4. Conduct a Proof of Concept (POC) to select the most adequate tool for component testing and another POC for E2E testing;
5. Implement front-end tests using both component and E2E testing methodologies in the Stratio Automotive's ecosystem;
6. Integrate the testing tools within the CI pipeline of Stratio Automotive to enable automated testing;
7. Transform the quality assurance process of Stratio Automotive;
8. Measure the impact of each change through testing/code coverage, defects prevented/identified and approval time.

1.3 Methodology

An agile methodology was used in the proposed solution. More specifically, Kanban (section 2.2.1), which was compatible with the workflow of the Engineering Operations team of Stratio Automotive, integrated by the Quality Assurance (QA) members.

During the development lifecycle, the Kanban board principles [6] were practiced, with various tasks assigned to the **backlog**. While planning, specific tasks were designated (represented by a card) for the **To Do** column. When the developer begins the task, they move the associated card to the **In Progress** column until the assignment is finished and submitted for approval to the QA tech lead.

In spite of working independently, two meetings were scheduled throughout the week with the Engineering Operations team during the development of the POC:

1. **Weekly Status:** every Wednesday, the intern would get the chance to interpolate their progress, what they pretend to accomplish and eventual difficulties;
2. **Planning/Refinement:** every Friday, the intern repeats the process from the weekly status meeting and appoints a new task (from the backlog) to perform during the next week if applicable.

1.4 Work Plan

The work plan for the **first semester** was focused on studying and analysing various aspects regarding quality in software. The main activities performed were the following:

- Understand testing levels;
- Study automation for front-end;
- Researching the state of art for front-end automation testing tools;
- POC for E2E testing with three relevant tools;
- Write Dissertation document.

The work plan for the first semester can be seen in Figure 1.2 and there were no deviations from the intended work.

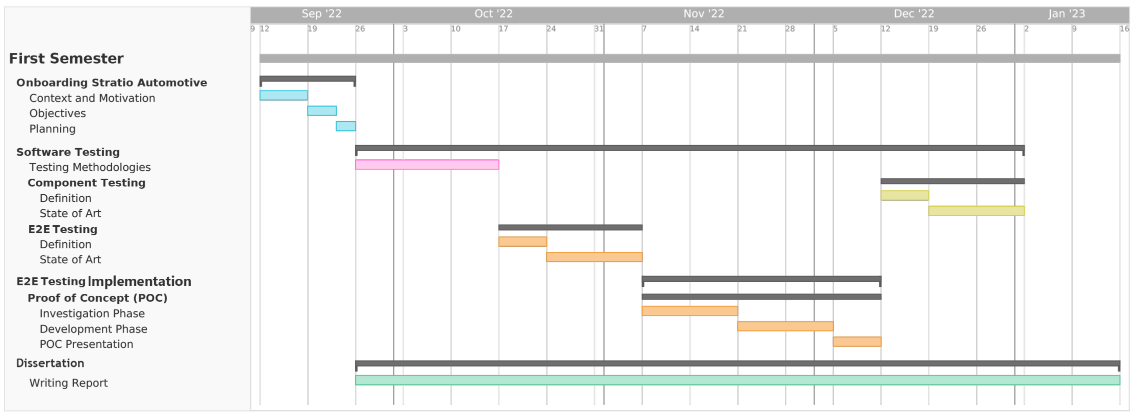


Figure 1.2: Work plan for the first semester.

The work carried out during the **second semester** focused on the practical component. The initial version of the work plan is illustrated in Figure 1.3, with the main focus on the development of component and E2E tests, including only GitLab CI integration within the latter.

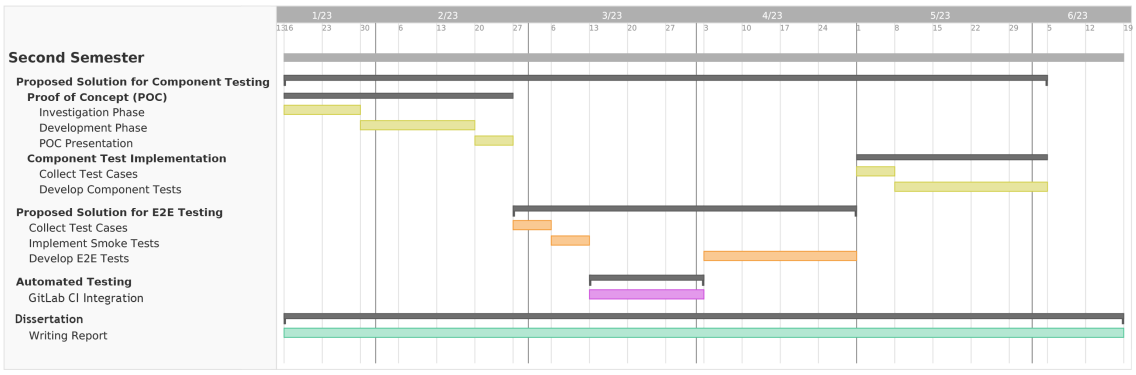


Figure 1.3: Initial work plan for the second semester.

After the intermediate presentation, the previous strategies were revised and the work plan was refactored. Before implementing the tests, it was necessary to conduct a more profound analysis of the quality assurance process of Stratio Automotive and the Foresight Platform, which was the main testing target. Additionally, both component and E2E testing will be implemented within the GitLab CI pipeline to enable automated testing, and a Test Driven Development (TDD) approach will be proposed. Finally, the time taken to retrieve metrics will be taken into consideration.

The main planned activities accomplished are as follows:

- Analyse the current quality assurance Process of Stratio Automotive and propose changes through automated testing;
- POC for component testing with two relevant tools;
- Compose component tests;
- Implement smoke tests;
- Compose E2E tests;
- Integrate both testing tools on the GitLab CI pipeline to enable automated testing;
- Explore a TDD approach with E2E tests;
- Retrieve metrics to measure the impact of the internship before and after the testing implementations;
- Write Dissertation document.

The work plan for the second semester can be seen in Figure 1.4.

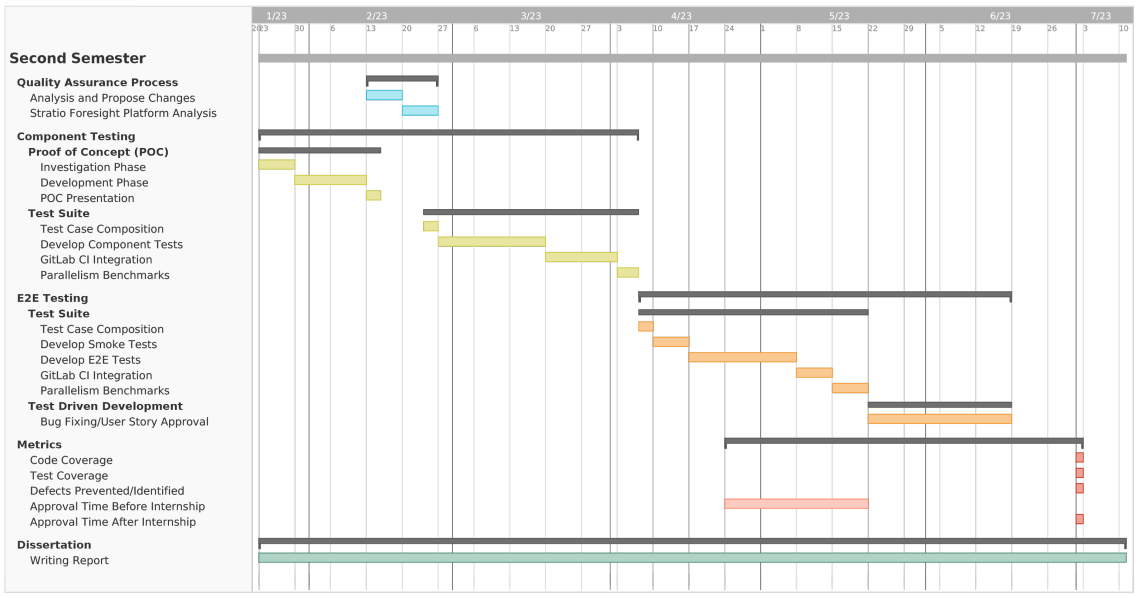


Figure 1.4: Actual work plan for the second semester.

1.5 Risk Management

In this section, all the foreseen causes of an unsuccessful internship are listed and examined, including the success criteria, risk analysis and mitigation plans.

Risk management is a crucial step that will not be overlooked, since it represents the process of detecting, assessing, tracking and reporting risks. Note that a risk is defined as an event or condition that could possibly cause a negative impact on the goals of a project it occurs [8]. The risk management strategy enables proactive risk management, impact reduction and plan handling.

1.5.1 Success Criteria

The success of this project depicts the point of view of both the intern and Stratio Automotive. The crucial conditions for success must be completed during the internship, within the expected time frame. These key conditions are as follows:

- Enable testing automation that reduces by half the manual testing and approval time of the Web UI platform;
- Reach target test coverage of 90% of the Stratio Foresight Platform features through E2E testing;
- Reach target code coverage of 75% for components through component testing;
- Integration of the selected tools for component and E2E tests in the GitLab CI pipeline to automate the Web UI tests.

1.5.2 Risk analysis

Before displaying the respective risks, it is important to define the likelihood of it occurring (Table 1.1) and the overall impact of a risk (Table 1.2).

Table 1.1: Probability scale.

Scale	Probability
Low	Low chance of occurrence.
Medium	Medium chance of occurrence.
High	High chance of occurrence.

Table 1.2: Impact scale.

Scale	Impact
Low	Little to no impact to the work plan or development.
Medium	Considerate impact to the work plan or development.
High	Potential to immensely impact the work plan or development.

Risk Identification

Below, the identified risks are stated (Table 1.3, Table 1.4, Table 1.5 and Table 1.6).

Table 1.3: Risk #1 - Steep learning curve for new technologies.

ID	R1
Statement	The learning curve for unfamiliar technologies is steep.
Probability	Medium
Impact	High
Consequences	Certain issues might take longer to resolve than anticipated.
Actions	**CP-1 Contingency Plan - Steep learning curve for new technologies:** Admit more time and effort to overcome hardships. Reach out for advice among the advisors and other Stratio Automotive workers who already have experience with these technologies.

Table 1.4: Risk #2 - Stratio Automotive's platforms are temporarily unavailable.

ID	R2
Statement	The platforms that the intern is going to test are temporarily unavailable for E2E testing.
Probability	Medium
Impact	High
Consequences	Impossibility of carrying out E2E testing on the appropriate platforms, delaying the work plan.
Actions	**CP-2 Contingency Plan - Stratio Automotive's platforms are temporarily unavailable:** Coordinate efforts with the software department and understand if there are any time periods where the platform is going to be down.

Table 1.5: Risk #3 - There are no viable tools to perform component or E2E tests.

ID	R3
Statement	During research, not finding any compatible tool to carry the pretended testing.
Probability	Low
Impact	High
Consequences	Impossibility of carrying out testing due to unavailable work methods.
Actions	**CP-3 Contingency Plan - There are no viable tools to perform component or E2E tests:** Use the studied tool that comes the closest to the intended work and coordinate with advisors better alternatives to manage the work plan.

Table 1.6: Risk #4 - Not enough time to retrieve adequate metrics to validate the proposed solution.

ID	R4
Statement	Not having enough time to retrieve all necessary metrics to fully validate the proposed solution.
Probability	High
Impact	Medium
Consequences	Certain conclusions may not be entirely confident or definitive.
Actions	**CP-4 Contingency Plan - Not enough time to retrieve adequate metrics to validate the proposed solution:** Utilize the retrieved data to validate the proposed solution and measure its impact as much as possible. Discuss the collected measurements and extrapolate adequate conclusions.

1.5.3 Overcoming Project Risks

During development, some of the identified risks emerged throughout the project. An evaluation of the risks encountered follows:

- **R2 - Stratio Automotive's platforms are temporarily unavailable:** a problem in the databases ended up disabling the Stratio Foresight Platform for a week. This issue delayed the retrieval of parallelism metrics for E2E tests for a few days, prompting the intern to anticipate the proposed TDD approach while the platform was unavailable;
- **R4 - Not enough time to retrieve adequate metrics to validate the proposed solution:** the measured metrics for this dissertation comprise three months for component tests and nearly two months for E2E tests. Nevertheless, section 4.8 includes a deep analysis of the achieved results with as few extrapolations as possible.

1.6 Dissertation Outline

Considering the main objectives described in Section 1.2, this Dissertation is organised as follows:

- **Chapter 1** (corresponds to this chapter) states the context and motivates the objectives for this Dissertation;
- **Chapter 2** presents the core details to understand Software Testing;
- **Chapter 3** delineates the state of art for the most popular solutions used on Web UI testing, including component testing and E2E testing;
- **Chapter 4** details the proposed solution for the quality process at Stratio Automotive, the POC's for component and E2E tests and the metrics contemplating the impact of the proposed solution;
- **Chapter 5** contains remarks about the work carried out in the internship as well as the plans for the future improvement of the solution.

Chapter 2

Background Information

In this chapter, some concepts and introductory topics will be explored in order to establish an important foundation for the next chapters of this document. This includes key aspects regarding software development, including its life cycle and work methodologies. Afterwards, an overview regarding the definition of software testing, white box, black box and gray box testing will be stated. Then, the four testing levels of the Testing Pyramid will be presented, focusing on **component** and **End-to-End (E2E)** testing, followed by a description of CI/CD.

2.1 Software Development Life Cycle

The software development life cycle is the process that a software goes through from its conception until its retirement. There are multiple different stages [9], represented in Figure 2.1.

1. **Planning Stage:** involves requirement analysis after meeting with the customer or owner of the software system. The main outcomes of this phase include quality assurance, risk identification and feasibility report;
2. **Defining Stage:** consists of the process of defining the key goals of the system according to the requirements from the involved stakeholders, leading to the software requirement specification document. One important aspect from this stage is the definition of **use cases** that represent a written description of actions performed by end-users. In other words, they describe the system behavior by capturing an intention/contract between stakeholders;
3. **Designing Stage:** focuses on the establishment of the software stack to perform the product, as well as the system's architecture. The main objective is to set how the solution will work with the quality attributes and the functional requirements of the system;
4. **Building Stage:** where the software engineering team is responsible for implementing the system. In this stage it is possible to utilise Agile methodologies, like Kanban;

5. **Testing Stage:** the software is thoroughly tested to ensure verification and validation. Note the importance of the use cases from the strategy phase, as they allow the conceptualization of **test cases** [10]. The latter corresponds to a set of actions that represent scenarios to be asserted against the software's intended functionalities and behavior. The result of these test cases is also an indication of where the undesirable behavior is located;

6. **Deployment Stage:** the sixth stage ensures the software is available in the production environment. All users have the opportunity to acquire the product;

7. **Maintenance Stage:** naturally, software is susceptible to changes that evolve its functionalities and user experience. During this stage, the development team fixes bugs or adds new features to adapt the product to the constantly evolving market.

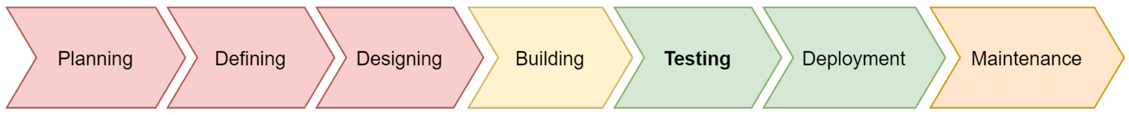


Figure 2.1: Schematic representation of the Software Development Life Cycle.

2.2 Process Management

In this section, crucial aspects regarding software development will be explored since it is important to ensure that the proposed software is concluded without many deviations. Process management seeks to provide a thorough understanding of the development and its significance in the software development business.

2.2.1 Work Methodologies

During this internship, Kanban will be used as an Agile methodology. This is based in the Agile Manifesto [11] which is a set of values and principles that enables teams to deliver software increments quicker. They achieve this rhythm of work by producing small, consumable chunks rather than a large project. Note that requirements, plans and results are assessed continuously so teams have a flexible and adaptive planning process that allows for changes and adjustments to be made during the development process.

Kanban

Kanban is an Agile work management system which allows the team to monitor their work through a set of principles [6]:

- **Easy work visualization:** usage of cards to represent tasks in a Kanban board (Figure 2.2) with three columns:
  - Requested: contains "To Do" tasks, yet to start;
  - In Progress: contains tasks that are being executed;
  - Done: contains concluded tasks.
- **Limit work in progress:** main emphasis on finishing work, minimizing multitasking, and preventing shifting the team's focus to other problems before finishing the current one;
- **Maximize efficiency:** to prevent bottlenecks, Kanban is a pull-based system to guarantee the team only starts to work on a different problem when resources are available to do so;
- **Explicit process policies:** the work process must be well specified in order to improve the team's organization;
- **Feedback loops:** it is possible to identify weak practices and enhance the work process by regularly evaluating the board.

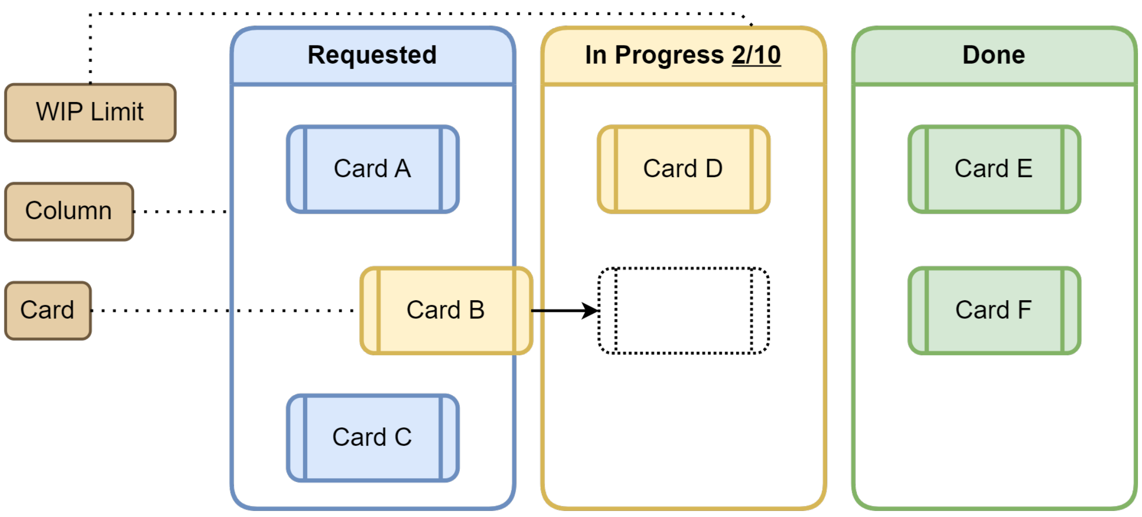


Figure 2.2: Schematic representation of the Kanban board.

2.2.2 Test Driven Development

Test Driven Development (TDD) inverts the usual flow of development and testing, since **test cases are developed first** in order to validate what the code will do. Rather than fitting a test to validate segments of the code, it is expected to implement code changes until it complies with the test case [12]. Figure 2.3 illustrates the typical workflow of the TDD approach. First, a new feature is **requested**, followed by **test** composition which will fail due to no code being developed. The feature is then implemented through the minimum amount of code until the unit test succeeds. Finally, the newly implemented code is refactored to improve readability, reduce complexity, and guarantee maintainability. After each refactor, it is important to execute all the preceding unit tests to ensure no bugs have been introduced (regression testing).

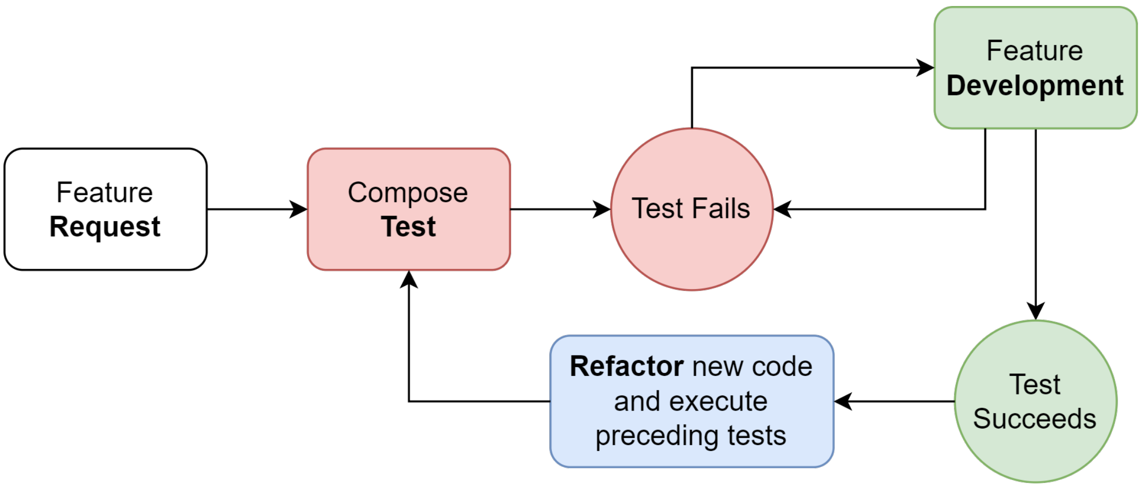


Figure 2.3: Schematic representation of the TDD approach.

2.2.3 Behavior Driven Development

Behavior Driven Development (BDD) may be used to complement TDD in order to broaden the development process for all the involved stakeholders. In the context of this Dissertation, it would be the quality assurance, development and product teams. This approach would improve the quality of tests since they are **projected from an end-user perspective** through a set of user stories and scenario templates. The latter is expressed by a syntax called "Given-When-Then" [13]:

- **Given:** provides the context;
- **When:** specifies events that occur;
- **Then:** represents the expected outcome.

In the other hand, user stories are expressed through the following structure:

As a **[Role]**, I want a **[Feature]**, so that **[Benefit]**.

2.3 Software Testing Overview

Software testing is a broad theme with countless concepts to define the best strategies to develop software with the best possible quality. It is relevant to distinguish static from dynamic approaches:

- **Static Approach:** encourages the tester to improve the code without executing it, through software reviews. This technique aims to identify possible defects early in the development phase. The cost of doing these kind of reviews is normally high since it is relevant to analyse big chunks of code at a time [14];
- **Dynamic Approach:** through this technique, the software is executed and the code is put under various conditions to pinpoint defects and/or issues. Dynamic testing gathers white box and black box methodologies to test the code [14]. **Note that during this document, we will be focusing on this testing paradigm with main attention in Web User Interface (Web UI) testing.**

Dynamic testing is frequently combined with static testing to create a thorough and comprehensive testing strategy for ensuring the quality and reliability of software systems.

2.3.1 Determining Testing Impact

There are various approaches to measure the impact of tests. In section 4.8, we will analyse various metrics to assess the overall impact of the proposed solution.

The **test coverage**, which is a measurement of how much a software is being exercised by a test suite, is a crucial factor. This criterion assists the team to understand what areas of the code are being tested, while measuring the total in percentage. Note that the better the test coverage, the greater the quality of the software system, in spite of the increased efforts to manage a larger test suite.

Another important measurement is **code coverage**, which assesses various indicators including the percentage of lines of code, statements, branches, and functions that are successfully executed by tests.

Furthermore, evaluating the **number of defects prevented/identified** through the implemented battery of tests is also a good indicator.

Before retrieving statistics, it is important to quantify approval times and code/test coverage **ahead** of performing new testing techniques. This would allow us to accurately compare the impact of the newly implemented tests.

2.3.2 Types of Software Faults

For more complex projects, it is important to distinguish different types of **bugs/defects**. In the interest of acting accordingly through the identification of different software bugs, it is an important part of the software development process to understand how they can be solved or tolerated.

The first type is **Bohrbugs** and they are the most **straightforward** to find during testing because they cause failure deterministically. A fault tolerance mechanism introduces design diversity and redundancy in the system to provide different sets of conditions to avoid this kind of failures [15].

The second type is **Mandelbugs** which are uncertain to cause failures under the same set of testing conditions. Due to their inconsistency, they are **very difficult** to disclose and it is complicated to describe a cause of failure. Retrying and using checkpoint strategies in the code to retain valid system states is a recommended fault tolerance mechanism [15].

The third most frequent type are **aging-related bugs**. Like the name indicates, they are activated after long periods of system run-time and are **difficult** to find during testing since they necessitate a lengthy execution time of the system. Logically, a fault tolerance mechanism would be to implement rejuvenation, that is, to periodically stop the software in order to refresh its state and remove any accumulated faults or failures [15].

2.3.3 Regression Testing

After integrating new code into an existing project, it is common to introduce problems into older features or add new defects. Regression tests aim to **retest** older code (not the one that was recently integrated) to ensure it is **still working as expected and no bugs were introduced.** Rerunning the existing component tests and E2E tests is sufficient to regression test the system, ensuring that new modifications integrated in a system did not negatively affect the overall workflow of the product.

2.4 White Box Testing

This testing technique requires the tester to have full knowledge and access of the code and internal workings to be tested. To design **test cases**, the tester discloses the product's structure and behavior with full knowledge of the internal system in mind. For example, this includes the **definition of specific code paths** to execute and inspect distinct functionalities.

White box testing is generally applied to systems to assert their internal logic and functionalities which can be seen in Figure 2.4. Since this methodology requires the tester to access the source code of the system, it is pertinent to test features that are difficult to access through the user interface. Some advantages of this testing methodology include [16]:

- Tests are deep and thorough, with the main goal of maximizing the tester's effort;
- Better code optimization as well as identification of security features;
- Ease of communication between developers and testers, since both parties thoroughly understand the system.

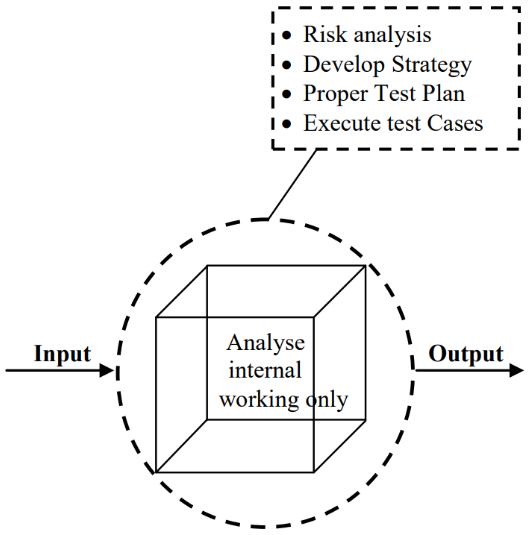


Figure 2.4: Schematic representation of the white box testing technique [16].

2.4.1 Procedure

Before undertaking white box testing [17], the tester must go through a **preparation phase** to become familiar with the development languages and tools. After understanding the product's characteristics, they **scrutinise the source code and inspect the internal system** in order to better understand the internal workings of the target application.

After studying the project, the tester focuses on the **creation and execution of the test cases**. Note that the tester should only carry on to this phase after a thorough understanding of the system. The next step is the documentation of test cases and their execution to assess the software system for defects and bugs.

The last phase corresponds to the **documentation**, where the tester registers a detailed description of the whole activity as well as a rundown of the tasks addressed.

The typical workflow to perform white box testing can be seen in Figure 2.5.

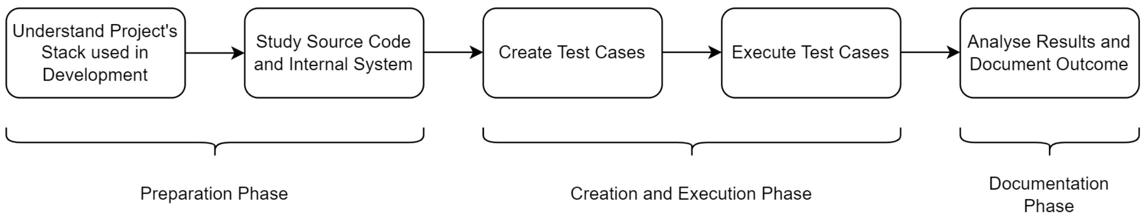

Understand Project's Stack used in Development
Study Source Code and Internal System
Create Test Cases
Execute Test Cases
Analyse Results and Document Outcome
Preparation Phase
Creation and Execution Phase
Documentation Phase

Figure 2.5: Schematic representation of the white box testing procedure.

2.4.2 Drawbacks

Implementing white box testing methodologies has the drawback of necessitating a **strong knowledge** of the internal system and a **careful investigation** of the source code. Generally, this process is **not immediate** and requires a considerate amount of time and resources to attain. Additionally, there may be **test scenarios that are not realistic** since they do not represent code paths that will be triggered by actual end-users. Finally, for very complex software systems, there will be **untested paths** since it is unfeasible to test all possibilities [17].

2.5 Black Box Testing

Black box testing requires the tester to only know the input and the expected output [18]. Thus, due to the "darkness" over the code, which should not be analysed in this testing methodology, the tester must understand the envisioned operation of the system from a user's perspective. To create **test cases**, the tester studies the requirements of the software and defines high level tests, for instance, simulating end-user actions through the user interface.

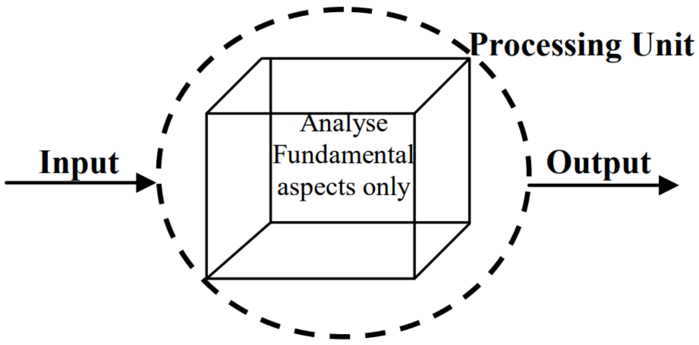


Figure 2.6: Schematic representation of the black box testing technique [16].

This testing methodology's core point is the functional requirements of the software, highlighted by Figure 2.6, since it tests the implemented features to validate if they are working as expected. Some advantages of this testing methodology include [16]:

- With low code tools, testers do not require extensive technical knowledge;
- Smaller preparation phase than white box testing. It is not required to study the source code and internals about the system to be tested;
- It is more efficient for large code segments, as the tester does not need to scrutinize the code;
- Rapid test case development.

2.5.1 Procedure

The workflow to perform black box testing is similar to the the methodology described in the previous section. The main difference is in the **preparation phase**, where the tester does not need to understand the project's stack or the source code. Instead, they carefully analyse the behavior of the system as well as the requirements.

The typical workflow to perform black box testing can be seen in Figure 2.7.

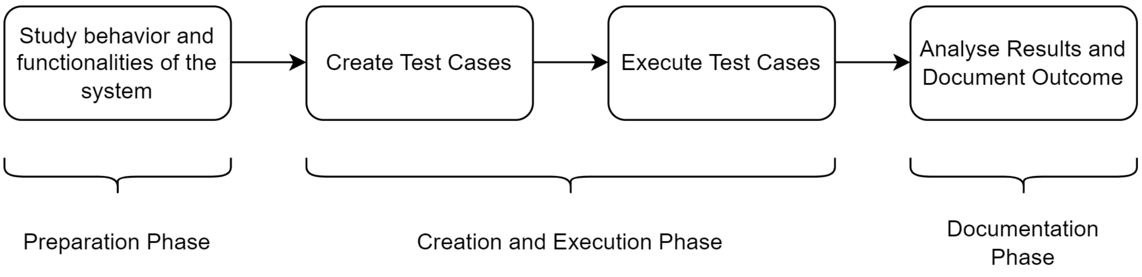

Study behavior and functionalities of the system
Create Test Cases
Execute Test Cases
Analyse Results and Document Outcome
Preparation Phase
Creation and Execution Phase
Documentation Phase

Figure 2.7: Schematic representation of the black box testing procedure.

2.5.2 Drawbacks

Black box testing methodologies may select only a restricted number of test scenarios, leading to **limited coverage**. Since this technique is broader in terms of amount of functionalities possible to test at each time, **test cases may be difficult to design**, time consuming and resource intensive. In fact, if the tester is unfamiliar with the inner workings of the software, it is possible that they will not be able to spot errors brought on by defects with the code or data structures [16].

2.6 Gray Box Testing

This testing methodology represents a combination of white box and black box testing, with the main objective of straightening out gaps from the previous techniques. The tester has some knowledge of the internal working structure, illustrated by Figure 2.8. For example, they may recognise components of the application to be tested but not how they interact [19].

If performing either white box testing or black box testing is neither practical nor viable, gray box testing is frequently performed. For instance, when the software is too vast or complex to test at a low level, or when the tester does not have full access to the source code, gray box testing is often a relevant option. Some advantages of this testing methodology include [16]:

- Instead of relying on source code, the tester focuses on interface definition and functional specification;

- The tests are developed from the user's perspective rather than the designer's;
- Includes benefits of both white box and black box testing.

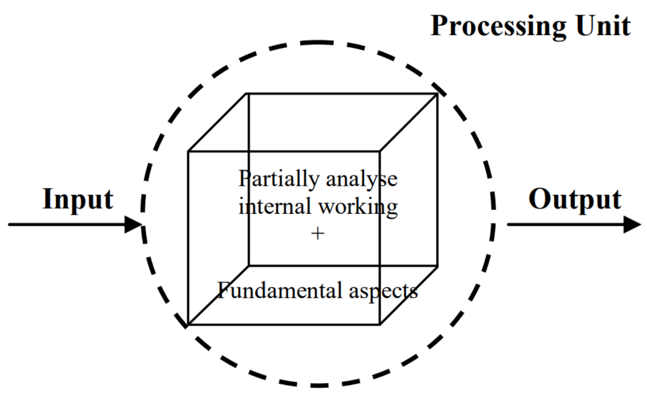


Figure 2.8: Schematic representation of the gray box testing technique [16].

2.6.1 Procedure

Akin to white box and black box testing, it is necessary to prepare before creating test cases, executing and analysing them. The tester must **understand the internal system they have access to** and then **combine the test case creation process** of both methodologies of testing [19].

2.6.2 Drawbacks

Given the restricted access to the source code, it might be **challenging to achieve the declared test coverage**, since some program paths may remain untested. Because the tester does not have access to the entire source code, this method is typically **less effective in finding bugs than white box testing**. Additionally, **gray box testing might not represent an end-user behavior** since the tester still has some knowledge of the internal system (contrary to black box testing) [16].

2.7 Testing Pyramid

In software testing, it is very important to define and comprehend all levels of testing. A Testing Pyramid [20] is an excellent tool for listing and describing the different testing levels, and the following steps can be found here:

1. **Unit Tests (Level 1):** being a unit the smallest piece of code that can be logically isolated in a system, unit tests aim to verify that each unit can function correctly when isolated from the rest of the code;

2. **Component/Module Tests (Level 2):** generally speaking, any software is made up of a number of components or modules, such as distinct classes, subprograms, and subroutines. To test the items in this category, white box testing is typically used;

3. **Integration Tests (Level 3):** these tests integrate several units from unit tests, combining them, and testing everything as a whole, typically black box testing;

4. **E2E Tests (Level 4):** at this level, it is assured that the application's user interface or Application Programming Interface (API) functions properly;

5. **Acceptance Tests (Level 5):** manual or automated method of testing, where a system is tested for acceptability. The main goal is to assess whether a specification's requirements are fulfilled.

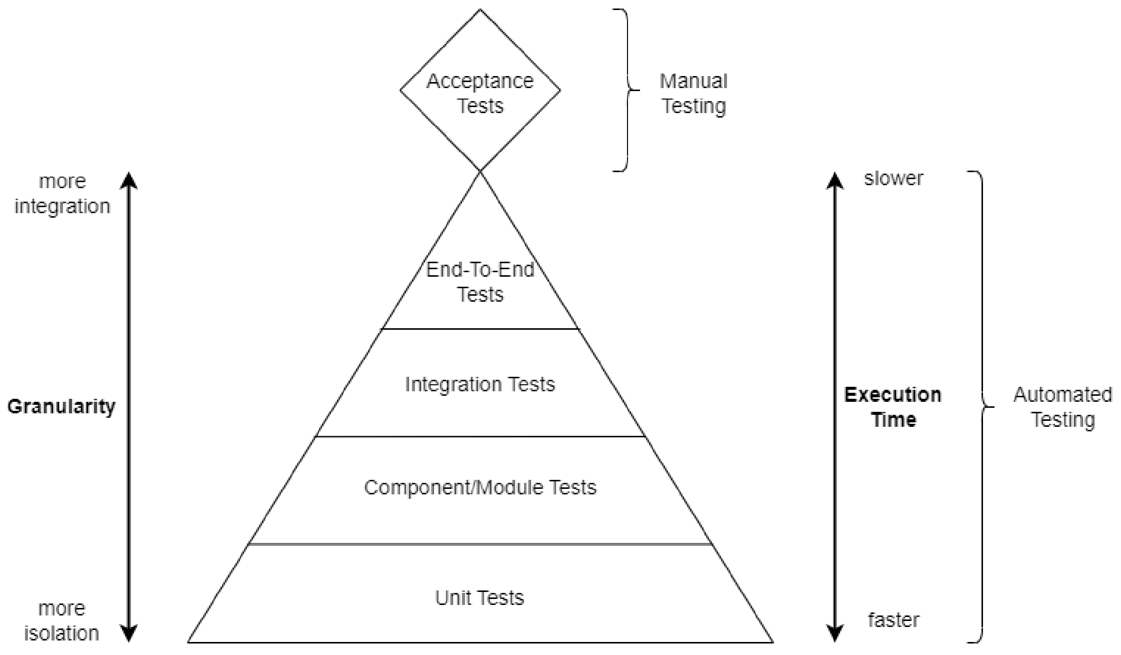


Figure 2.9: Schematic representation of the Testing Pyramid.

It should be taken into account that the Testing Pyramid is mainly used to separate different types of tests and make sure that there are no hidden bugs or issues in the software. This raises the question of whether there is a specific order of the tests the team should follow or what the quantity of testing for each level should be. Firstly, it is crucial to comprehend the reasoning behind the pyramid's four levels before delving into the preceding statements. The full model of the Testing Pyramid is represented in Figure 2.9.

2.7.1 Four Levels of Testing

The Testing Pyramid is composed of four testing levels, all of which can be automated. To execute automated tests in projects, the pyramid represents the key to doing so since each layer represents a different level of testing with the sole purpose of covering as much as possible. The representation used aims to divide tests in an unambiguous way and achieve Continuous Delivery and Deployment.

2.7.2 Order of Execution and Testing Quantity

The Testing Pyramid delineates the foundation for an automated test suite. To achieve that, each level gets less and less granular and the execution time slows down. The rule of thumb is to write tests with varying granularity, and the more high-level you get, the fewer tests you should have.

Execution should be done in level order. For instance, testing a specific module or component without first running unit tests is ineffective.

2.7.3 When to Level Up

As discussed earlier, each level has different methodologies, priorities, and tools. Since a tester should begin at the bottom of the Testing Pyramid, the methodology ensures that the scope of testing is minimal and units of code are isolated. As the level of the pyramid increases, the testing granularity escalates and the execution time decreases, so proper documentation of when to test each level and what is the "trigger" that moves up level needs to be created.

Note that there are different levels for different scenarios, since each level covers several scenarios that other levels are not capable of. For instance, to ensure a system communicates well with a specific database, integration tests are used (level 3). In other words, each level serves a purpose that lower levels can not grant.

2.8 Component Testing

Component tests isolate a specific feature that a component implements. To better comprehend this kind of testing, the definition of "component" must be explained. A component represents a logical piece of the system, usually described as a group of one or more Hypertext Markup Language (HTML) elements (and a TypeScript class if using Angular [21]) in the context of the front-end.

2.8.1 Definition

Component testing is a form of white box testing performed by the quality assurance team. It includes all the testing done on each component of the software individually and without isolation. Before starting, there should be a minimum number of components developed and unit tested. The exit criteria demands that all the tested components work as expected and that there are no critical, high or medium severity and priority bugs. It is important to stress the difference between component and unit testing from a practical point of view [22]. Component tests generally mount the component individually and the tester interacts with it. Unit tests are a white box approach where the tester validates if a small isolated unit of code works as expected.

As mentioned in section 4.1, the Stratio Foresight Platform, which is the primary test target, uses the Angular framework. This architecture benefits from this testing practice [23] due to the fact that the platform is composed of very complex components that lack some form of testing.

2.8.2 Motivation to use Component Tests

One of the main goals of quality assurance is to find as many bugs as possible and increase the quality of the product during the development phase. More specifically, component testing provides a broad view regarding the software and how each component executes the corresponding functionality. The component test strategy workflow can be seen in Figure 2.10.

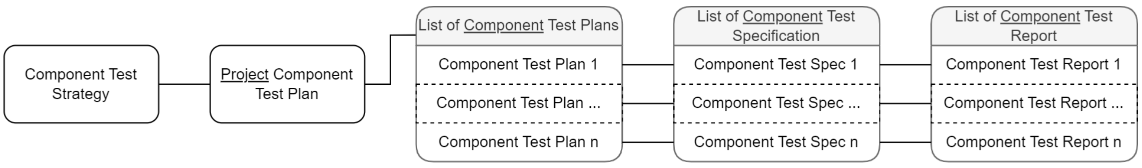


Figure 2.10: Schematic representation of the Component Test Strategy Workflow.

Another reason component testing is a prevalent testing technique is because of the front-end developer's tendency to increase the flexibility of the User Interface (UI) they build, as well as the complexity and embedded logic in it. Components are interchangeable pieces of a screen that isolate the application business logic within [24]. The tester must decompose complex user interfaces into smaller components that should be tested individually.

2.8.3 Procedure

Before starting to test every component in a specific application, the user must develop a **component test strategy** and a **project testing plan**. The first refers to the methodology chosen, the tactics to approach and which modules to validate, whereas the latter includes the scope, objective and impact of each test case [25]. After creating these documents, each component must have a testing plan, a testing specification (scenarios that will be tested and their frequency of testing) and a testing report.

2.8.4 Data Mocking

Since component testing requires the isolation of a component, the task of validating the embedded logic can become difficult since more complex components may need input data from other parts of the application (user input, for example). One solution to allow more complex scenarios to be tested is to include forged data to artificially enrich the testing specification. The main advantage of this technique is the simulation of cases that rarely happen and the increased test coverage. On the other hand, the tester should be well aware of the application since they can end up forging impossible data that would never occur during normal operation of the software.

2.9 End-to-End Testing

This black box testing technique aims to cover the functionalities of the application by simulating an actual user. In other words, black box testing is used to replicate an end-user behavior to validate the system in terms of both data integration and integrity.

The methodology of this testing type encapsulates the integration within the application as well as the external interfaces [26]. The Testing Pyramid classifies E2E testing as the fourth level of the pyramid, which means that this technique is not granular, i.e., it is inconceivable to completely isolate elements when performing this kind of testing.

The main objective of E2E testing is to detect bugs and increase the test coverage of the software by analysing the application's workflow through a series of steps previously defined to replicate real user's behaviors. Note that this testing method also validates how "...the application communicates with hardware, network connectivity, external dependencies, databases, and other applications." [27].

2.9.1 Page Object Model (POM)

The main purpose of this design pattern is to enhance the maintainability and reusability of test scripts when writing automated tests for different **Page Classes**. These are defined as the class file of a specific web page that encapsulates its locators and methods. The POM consists of a repository for Web UI elements for each individual corresponding Page Class, which provides an identification for all of them [28]. Figure 2.11 exemplifies the strategy:

- Class A contains **both** the Web UI elements and methods, as well as the test methods;
- Class B holds all the Web UI elements and methods **separately** from the Class C that holds the test methods;

- **Without** POM structure, it is **not feasible to re-use** the Web UI elements and methods in multiple classes at the same time;
- **With** a POM based structure, the Web UI elements and methods are defined in Class B, where it can be re-used in multiple classes.

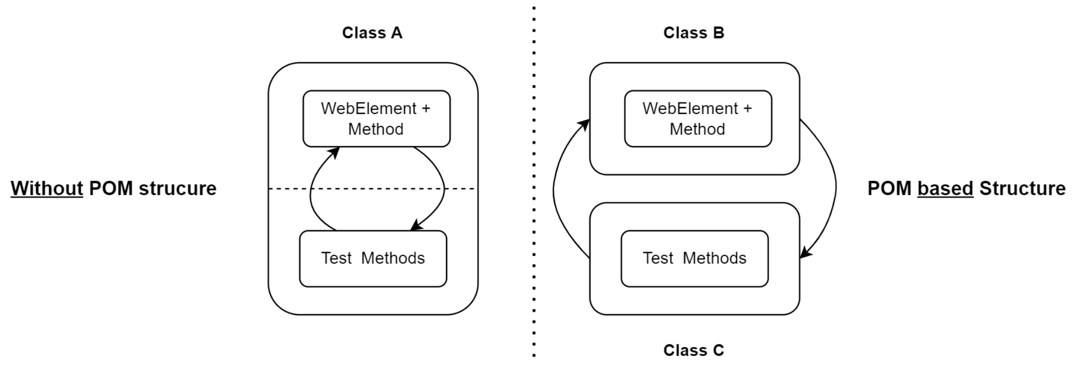


Figure 2.11: Schematic representation of the Page Object Model design pattern.

POM guarantees better optimization since it is possible to reuse the same code (Class B) as it is contained in another class apart from the test methods.

2.9.2 Smoke Tests

Before starting E2E tests, it is a good practice to perform smoke testing. This technique can be considered a build verification test which can be automated to analyse and ensure that the most important functionalities of the software are working as expected [29]. Since these kind of tests are rapid and easy to execute, they guarantee that the product is stable enough to sustain a more complex kind of testing, like E2E tests.

After the development team delivers a new build of the application, it is logical to use smoke tests to find more obvious defects and improve the code before undergoing more exhaustive tests. As it was discussed, a typical smoke test includes the most critical functionalities of the system and are designed to validate the basic functionalities of the application. In the case of Stratio Automotive, a few smoke tests cases are as follows:

- Perform login;
- Visit the main Dashboard;
- Consult a specific vehicle details;
- Consult the active alerts of a vehicle.

2.9.3 Best Practices to develop Web Automation Tests

The process of writing tests for Web Automation is quite simple. However, there are some principles that should be followed to allow for a better workflow and steadier development.

Independent Tests

Coupling more than one test together is not good practice. They must always run independently from one another, i.e., if the test can be executed and pass without the intervention of others, then it is considered a good test [30]. If the opposite happens, the code should be refactored and another approach should be analysed. A good workaround is to combine multiple tests that depend on each other into one larger test.

Adding Multiple Assertions in a Single Test

A common mistake when writing E2E tests is to create very small tests with a single assertion, resembling unit tests [31]. Instead, a better practice is to implement multiple assertions in a single test:

- It is not necessary to rely on the test's title to understand where it failed since it can be observed visually which specific assertion failed when running a large test with multiple assertions;
- This practice also ensures there are no performance penalties since it is not needed to reset tests multiple times and there is only one test with multiple assertions.

Unnecessary Waiting

It is a normal temptation to apply manual *wait* periods to stand by until an element is loaded. Many tools like Playwright [32] dictate that it is not necessary since there are other alternatives:

- Refactoring the code and testing it in another way;
- Using assertions, as they prevent the tool from continuing until an explicit condition is met.

Design Patterns to Follow

Test specifications should be isolated and programmatically controlled by the application's state. A good practice is to use the POM as a design pattern, which avoids duplication and improves maintainability, simplifying interactions between pages. It creates a repository for Web UI elements for each individual corresponding Page Class.

Correctly Select Elements

Web Automation Tests benefit from using the field **data-*** in different HTML elements to provide context to the selectors, allowing them to be isolated from Cascading Style Sheets (CSS) or JavaScript changes. The main objective of using these kinds of selectors is to guarantee the usage of fields that are resilient to change.

Consider the following example for Cypress [33]. Given that we have the following button:

```
<button
  id="main"
  class="btn btn-large"
  name="submission"
  role="button"
  data-cy="submit">
  Submit
</button>
```

Here are the possible selectors to click that button:

- **Never use:**
  - *cy.get('button').click()* - very generic form of identifying the element. Does not work in the ocasion of having multiple buttons;
  - *cy.get('.btn.btn-large').click()* - this manner is coupled to styling which implicates that is subject to change.
- **Susceptible to problems:**
  - *cy.get('#main').click()* - coupled to styling or JavaScript event listeners;
  - *cy.get('[name="submission"]').click()* - the 'name' attribute has HTML semantics and might be altered;
  - *cy.contains('Submit').click()* - the text content might change.
- **Good practices:**
  - *cy.get('[data-cy="submit"]').click()* - since it uses an unique tag, it is completely isolated and not susceptible to change.

Testing in Production

By testing in production, we conduct automated tests after the project has been deployed, meaning that we get to know if there is any problem with the application. On the other hand, not all software can be tested in production since there may be cases where it is impossible to update it in the future.

Account Usage

When developing automated tests, it is important to understand the product and the most preferable approach to testing it. A good convention is to create a specific and new testing account, i.e., do not use an existing profile of a real client.

2.10 CI/CD

Standing for "Continuous Integration and Continuous Delivery/Deployment", CI/CD is a common technique among interactive development processes. The main advantages of adopting this method include efficiency, scalability and testing automation.

Component and E2E testing are usually paired with CI/CD pipelines since they ensure that the entire system works as expected before it is deployed to production. When using this technique, a programmer submits their code changes, which are automatically built, tested and deployed (Figure 2.12). The testing process may include component and E2E tests to detect any bugs or issues that may appear as a repercussion of the introduction of functionalities or changes. Developers are then notified of eventual bugs, allowing them to identify and fix any issues to maintain software quality.

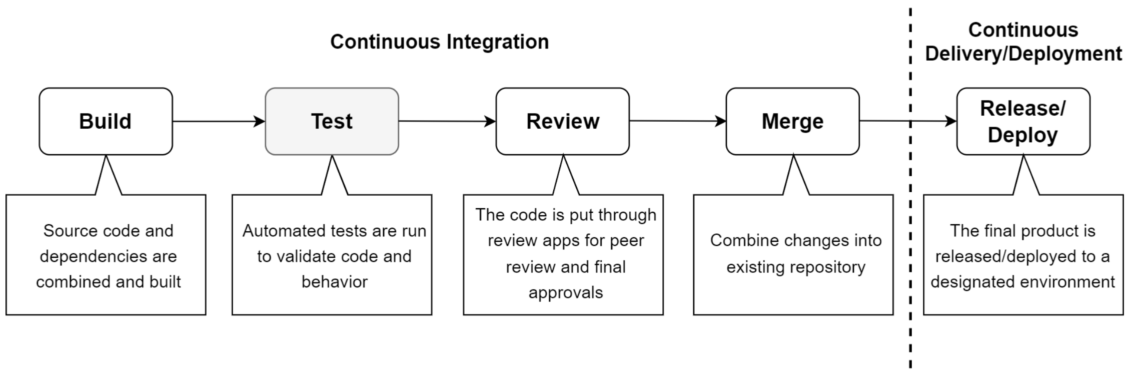

Continuous Integration
Continuous Delivery/Deployment
Build
Test
Review
Merge
Release/ Deploy
Source code and dependencies are combined and built
Automated tests are run to validate code and behavior
The code is put through review apps for peer review and final approvals
Combine changes into existing repository
The final product is released/deployed to a designated environment

Figure 2.12: Schematic representation of the CI/CD workflow.

2.10.1 Continuous Integration (CI)

The main purpose of this method is to merge all code changes from a given repository into a single project. Every development step is automatically and continuously built and tested (Figure 2.12), allowing the coder to detect and fix errors quickly, reduce integration problems by merging small changes and enable the team to work faster and with less apprehension [7].

2.10.2 Continuous Delivery/Deployment (CD)

With Continuous Delivery, the application is deployed continuously, in spite of **triggering the deployments manually**. That is the main difference from Continuous Deployment, where the **triggers are automatic** [7]. CD ensures all changes are releasable (Figure 2.12) and lowers the risk of introducing new changes in the repository.

2.10.3 GitLab CI/CD

Stratio Automotive uses GitLab [7] to adopt the DevOps lifecycle in their development workflow. After planning and coding, the team commits their changes to the GitLab repository, which triggers the projected CI/CD pipeline for the project (Figure 2.13). Component and E2E testing will fit into CI since it includes automated tests for each commit, meaning that a developer can validate their code changes through a battery of automated tests.

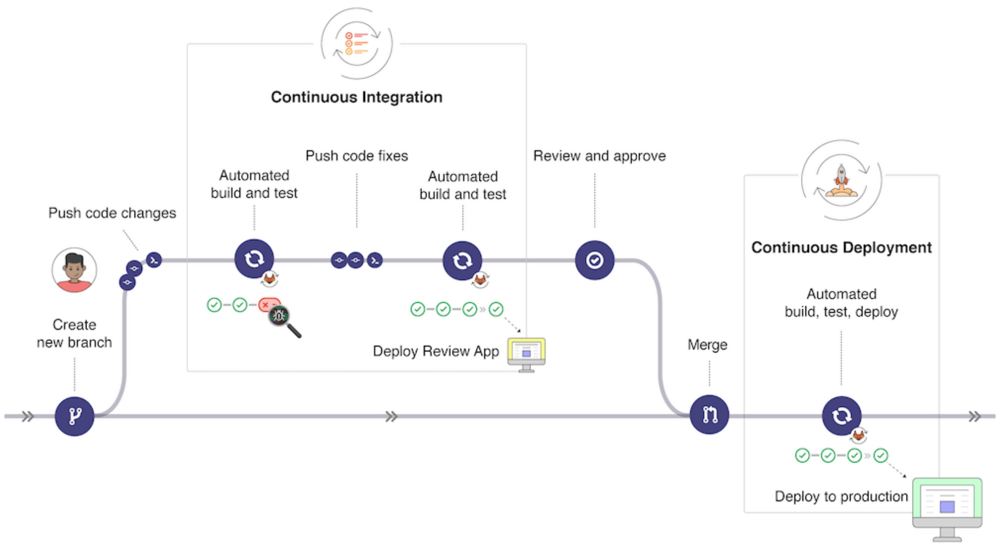


Figure 2.13: Schematic representation of the CI/CD method with GitLab [7].

Chapter 3

State of Art

In this chapter, a summary of the key properties of different testing tools will be provided along with the main features and trade-offs of each option. By analysing the current state of art for both component testing and E2E testing, a better overview of the most adequate options for future development will be evident as well as key areas for future research and key trends in the field.

3.1 Component Testing

In this section, component testing tools will be presented. Along the study, different characteristics will be analysed to evaluate the state of art of the most relevant component testing tools available. Note that it is important to study tools that are compatible with Stratio Automotive's stack. In this case, some form of **support for the Angular Framework is mandatory**. The preceding can be found and explained in Tables 3.1 and 3.2.

Table 3.1: Properties studied for each component testing tool explored.

Open Source/Free	In spite of most tools being open source and free to use, they can still have additional features that are locked behind a paywall. It is very important to study what the tool offers and if there are functionalities that fit into our product.
Browser Support	The variety of multiple browser engines to mount components is an important aspect that should not be overlooked.
Language Bindings	Depending on the tool, the number of supported languages varies. Typically, JavaScript or TypeScript are both supported, unless it is a low code solution.

Table 3.2: Properties studied for each component testing tool explored (Cont.).

Support Documentation	Since the programmer leans on support documentation for further information, this characteristic can be seen as the core of the testing tool since it bolsters the performance when installing or writing tests. The community's provided documentation is also an important factor since it directly impacts the programmer's experience.
Framework/Library Support	Different testing tools are developed with distinct objectives. It is important that the selected tool is capable of testing in Angular since that is part of the stack of Stratio Automotive.
Parallelism	Parallel testing is the practice of testing multiple versions concurrently with the same input to reduce the execution time and increase test coverage [34]. Naturally, this characteristic is very important and should be available for most tools.
Code Complexity	It is important to analyse the effort of setting up the adequate testing environment to mount the pretended components without issues. Besides, it is also important to study the tool's potential and its capabilities to mount components, pass data, intercept requests and test event handlers.
Key Features	Some tools have certain core aspects that distinguish them from the competition. Logically, key features must be presented and studied.
Trade-offs/Disadvantages	As well as advantages, some tools present trade-offs or disadvantages that can invalidate their use in a specific project.

Following the Testing Pyramid paradigm, component tests are less complex and more granular than E2E tests. Inducing these characteristics, it is expected for component tests to be quicker, smaller and easier to maintain.

3.1.1 Cypress

Cypress provides a testable component bench [35] to isolate and test components. According to the documentation, the test runner is browser-based, admitting access styles, API and the isolation of different components. This strategy works well, allowing to split the application into blocks while testing.

- **Open Source/Free:** Cypress is free and open source under the MIT license. However, there is a paywall for using the Cypress dashboard with fewer or no restrictions. In the beginning of 2022, it launched the beta for component testing. Since version 11 (August 2022), component testing for Cypress has been available for general use and is being perfected for each release;

- **Browser Support:**
  - Google Chrome;
  - Mozilla Firefox;
  - Microsoft Edge;
  - Opera;
  - Electron;
  - Brave.
- **Language Bindings:** JavaScript/TypeScript;
- **Support Documentation:** The support documentation for Cypress Component Testing is adequate for the functionalities it proposes. The official source displays tutorials for installation for each supported framework and how to start component testing. Besides these instructions, it also explains to the tester how the mount process functions, the available functionalities, theoretical concepts of the advantages of component testing, as well a collection of YouTube videos in the official channel;
- **Framework/Library Support:**
  - React;
  - Vue;
  - Svelte;
  - Angular.
- **Parallelism:** Since version 3.1.0, Cypress can execute recorded tests in parallel across multiple machines through the Cypress Dashboard;
- **Code Complexity:** The process of including component tests in a project is straightforward. After opening the Cypress Dashboard, the Launchpad will guide the user through the installation process, detecting the available framework and creating the necessary files for the user to start component testing;
- **Key Features:**
  - Browser-based Test Runner, granting the user access to the component's functionality, style and appearance. This changes will happen in real time and it is possible to interact with the component;
  - Component tests share all the features of regular Cypress, like parallelization, APIs, plugins and ecosystem.
- **Trade-offs/Disadvantages:**
  - No support for the solid framework.

3.1.2 Playwright

Microsoft Playwright is capable of (experimental) component testing [36] by using vite to assemble bundle components and serve them. Since component testing runs on the browser, it is possible to visualize the process during execution.

- **Open Source/Free:** Playwright is a free and open source option that launched an experimental way of component testing for various front-end frameworks;
- **Browser Support:**
  - Google Chrome;
  - Mozilla Firefox;
  - Microsoft Edge;
  - Opera;
  - Apple Safari.
- **Language Bindings:**
  - Java;
  - Python;
  - C#;
  - JavaScript;
  - TypeScript.
- **Support Documentation:** The documentation for component testing is very vague and scarce since it is a recently added feature. However, it indicates details about the rendering methods for components, how to mount (simple) components and execute the tests;
- **Framework/Library Support:**
  - React;
  - Vue;
  - Svelte;
  - Solid;
  - Angular (non-official support) [37].
- **Parallelism:** This tool runs tests in parallel, using various worker processes that run at the same time. Parallelism is a default feature of Playwright however, tests in a single file run in order in the same worker process. This feature can be tailored by the programmer;
- **Code Complexity:** It is trivial to integrate component testing into an existing React, Vue, Svelte or Solid project using Playwright. It is necessary to install Playwright Test for components and include style sheets, themes or code from your application into the page where the component is mounted: "playwright/index.ts";

- **Key Features:**
  - Since tests run in Node.js and components are executed in the browser, they are mounted in isolated testing environments, with real clicks being triggered and layouts and visual regression being executed;
  - Component tests share all the features of regular Playwright like parallelization, parameterized tests and trace history.
- **Trade-offs/Disadvantages:**
  - Only possible to import components from TSX/JSX/Component files;
  - Not possible to pass variables to the mount method of Playwright.

3.1.3 Karma & Jasmine with TestBed

Jasmine [38] is a framework to test JavaScript/TypeScript code with a clean and easy to understand syntax. It is usually coupled with Karma [39] which takes care of executing the tests on multiple real browsers. Along with these tools, TestBed [40] will be associated, which configures and initializes an environment for both unit and component testing exclusively to Angular.

- **Open Source/Free:** All the tools analysed in this section are free and open source;
- **Browser Support:**
  - Google Chrome;
  - Mozilla Firefox;
  - Microsoft Edge;
  - Opera;
  - Apple Safari.
- **Language Bindings:** JavaScript/TypeScript;
- **Support Documentation:** All the tools present a brief explanation of its functionalities and how to use them. Since the community frequently uses this combination to test Angular applications, there is a lot of feedback and support regarding best practices for using these tools;
- **Framework/Library Support:** Angular;
- **Parallelism:** As a result of Karma being responsible for executing the tests across multiple browsers, there are external plugins to shard tests without changing their build process. The main purpose of the package 'karma-parallel' [41] is to speed up the time it takes to run tests;
- **Code Complexity:** Since this combo is included in Angular projects by default, it is unequivocal to setup the testing environment and all the necessary configurations to start testing right away;

- **Key Features:**
  - Already included in Angular projects;
  - Low overhead with a simple to read syntax;
  - Rapid testing.
- **Trade-offs/Disadvantages:**
  - Mainly used for unit tests only.

3.1.4 Storybook

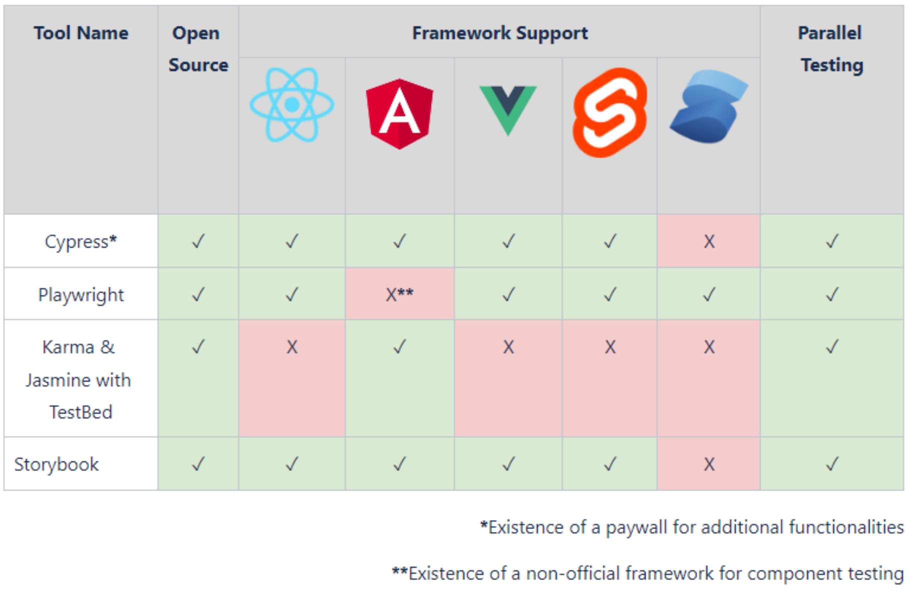
Storybook [42] provides methods to build, test and visualize UI components in isolation. Since many front-end frameworks like Angular organize their code into separate components, it is important to test them individually to assert possible problems. Storybook allows the user to study their components without data dependencies or business logic, granting the ability to share the work with other development teams to validate each component.

- **Open Source/Free:** Storybook is free and open source;
- **Browser Support:**
  - Google Chrome;
  - Mozilla Firefox;
  - Microsoft Edge;
  - Apple Safari.
- **Language Bindings:** JavaScript/TypeScript;
- **Support Documentation:** The support documentation for Storybook is fair since it covers various technical aspects regarding the functionalities this tool offers as well as a tutorial for setting up Storybook for each supported front-end framework. The community's support is also valid by virtue of the adequate amount of tutorials, videos and repositories that use and explain Storybook;
- **Framework/Library Support:**
  - React;
  - Vue;
  - Svelte;
  - Angular;
  - Web Components;
  - Ember.

- **Parallelism:** For more complex applications, many components exist, causing uncertainty about the number of features each story must test. Storybook composition combines multiple storybooks into one, granting a faster development experience since it is possible to interact with multiple components at once [43];
- **Code Complexity:** The process of integrating Storybook into an already existing project with a lot of applications might be challenging since there may be some migrations to be done. Fortunately, the documentation presents a tutorial on how to do so and the process for creating stories for each component;
- **Key Features:**
  - Each component is rendered in real time, presenting a schematic representation of itself;
  - It is possible for the user to interact with each rendered component;
  - Cypress and Playwright compatibility to automate interaction within each component.
- **Trade-offs/Disadvantages [44]:**
  - Necessary to maintain the components and the Storybook library;
  - Hard to transition into existing projects due to migration work.

3.1.5 Summary Board

The generic information of each investigated tool can be found in the comparative board illustrated by Figure 3.1.



Tool Name	Open Source	Framework Support					Parallel Testing
Cypress*	✓	✓	✓	✓	✓	X	✓
Playwright	✓	✓	X**	✓	✓	✓	✓
Karma & Jasmine with TestBed	✓	X	✓	X	X	X	✓
Storybook	✓	✓	✓	✓	✓	X	✓

*Existence of a paywall for additional functionalities

**Existence of a non-official framework for component testing

Figure 3.1: Summary board of the software testing tools for component testing.

3.2 End-to-End Testing

In this section, E2E testing tools will be presented. In addition to the installation process and brief description, different characteristics that were important to evaluate and assess were highlighted. The preceding can be found and explained in Tables 3.3 and 3.4.

Table 3.3: Properties studied for each E2E testing tool explored.

Open Source/Free	Even though most tools are open source and free to use, they can still have additional features that are locked behind a paywall. Understanding the qualities that are not immediately available has significant weight.
Browser Support	This property is very important to analyse. Some software does not support Safari from Apple, for example, which can be a dealbreaker for some tools.
Language Bindings	Depending on the tool, the number of supported languages varies. Typically, JavaScript or TypeScript are both supported, unless it is a low code solution.
Support Documentation	Since the programmer leans on support documentation for further information, this characteristic can be seen as the core of the testing tool since it bolsters the performance when installing or writing tests. The community's provided documentation is also an important factor since it directly impacts the programmer's experience.
CI/CD Integration	CI/CD main purpose is to increase quality and development speed. In this practice, the chain of development and testing proceeds on its own, and testers notify developers when they identify bugs that can be rapidly fixed [45]. Continuous testing guarantees that bugs are discovered before any significant issues develop. Some properties worth analysing include integration capabilities and the application type supported.

Table 3.4: Properties studied for each E2E testing tool explored (Cont.).

Parallelism	Parallel testing is the practice of testing multiple versions concurrently with the same input to reduce the execution time and increase test coverage [34]. Naturally, this characteristic is very important and should be available for most tools.
Locators/Selectors	When creating Web UI tests, it is crucial to easily identify the elements of the page since that is the main way of interacting with them. Evidently, a good testing tool should provide a mechanism to quickly find specific elements.
Flakiness	Some tests occasionally pass or fail, i.e., they do not always produce the expected result and are considered flaky. Different aspects can lead to this behavior, like issues with code, external factors, or even the test itself [46]. Some testing tools minimize flaky tests.
Key Features	Some tools have certain core aspects that distinguish them from the competition. Logically, key features must be presented and studied.
Trade-offs/Disadvantages	As well as advantages, some tools present trade-offs or disadvantages that can invalidate their use in a specific project.

3.2.1 Cypress

This testing tool provides an open source test runner [47] that can simulate an end-user in a specific web browser. In other words, it is described as a tool for reliability testing. The first commit was in 2014, and since then, Cypress has been on the rise in terms of new features, support, and popularity. One of its key characteristics is the Cypress Dashboard, a service that ensures access to recorded test results and various insights into the tests conducted.

- **Open Source/Free:** Cypress is free and open source under the MIT license. However, there is a paywall for using the Cypress dashboard with less/no restrictions;
- **Browser Support:**
  - Google Chrome;
  - Mozilla Firefox;
  - Microsoft Edge;
  - Opera;
  - Electron;

  - Brave.

- **Language Bindings:** JavaScript/TypeScript;
- **Support Documentation:** It is simple to find help on the web through community posts or the official Cypress documentation. They cover technical, functional, and architectural aspects of the testing tool as well as limitations and trade-offs;
- **CI/CD Integration:** It is possible to use Cypress Dashboard with GitLab CI/CD;
- **Parallelism:** Since version 3.1.0, Cypress can execute recorded tests in parallel across multiple machines through the Cypress Dashboard;
- **Locators/Selectors:**
  - ID;
  - Class;
  - Tag Name;
  - Attribute;
  - CSS Selector;
  - Document Object Model (DOM) Selector;
  - XPath (through a plugin).
- **Flakiness:** Since Cypress perceives everything that happens in the browser in a synchronous manner, it identifies all events and knows how long an element is taking to be drawn/animated and will wait for it to become visible and enabled. Additionally, Cypress automatically executes the overwhelming majority of commands inside the browser without network lag and waits for the application to reach a certain state before continuing. Because of these features, we reduce flakiness and the same test will produce the expected results more consistently;
- **Key Features:**
  - A snapshot is saved after each step is done during testing, allowing the programmer to better understand the root of the problem in case the test fails;
  - Debugging can be done directly from the browser web tools;
  - Automatically waits for async events.
- **Trade-offs/Disadvantages [48]:**
  - Cypress is only meant to be used as a testing tool (it is not an all purpose automation tool);
  - It is executed inside the browser, meaning it is more difficult to communicate with the backend;

  - This tool can not be executed in multiple tabs/browsers since it runs in the browser;
  - Cypress cannot visit two domains of different origins in the same test;
  - Must install a plugin to use XPath.

3.2.2 Selenium

This tool covers a webdriver [49] and a low-code option [50] through an IDE. The project was conceived in 2004 and consists of a suite of test tools for automating web browsers. The webdriver version simulates an end-user either locally or remotely. The IDE is an add-on for Google Chrome, Mozilla Firefox, and Microsoft Edge that lets users create new tests from scratch and record and replay existing ones in the browser.

- **Open Source/Free:** Selenium is an open source and free tool, available for everyone;
- **Browser Support:**
  - Google Chrome;
  - Mozilla Firefox;
  - Microsoft Edge;
  - Opera;
  - Apple Safari.
- **Language Bindings:**
  - Java;
  - Python;
  - C#;
  - JavaScript;
  - Ruby.
- **Support Documentation:** Considering the time Selenium Webdriver has been available to the public, there is a lot of community support and learning material. Developers can also get a lot of assistance from the official documentation;
- **CI/CD Integration:** Selenium Webdriver is capable of CI/CD integration, for example: using a headless driver;
- **Parallelism:** This feature is also available on the Selenium Webdriver;
- **Locators/Selectors:**
  - ID;

  - Name;
  - Locate Element By Name using Filters;
  - Link Text;
  - CSS Selector;
  - DOM Selector;
  - XPath.

- **Flakiness:** Since the Selenium Webdriver protocol is deterministic, it reduces the chances of flaky tests occurring. Some factors like lack of synchronization or accidental load testing, can still introduce flakiness in the test suite;
- **Key Features:** It is a W3C recommendation [51], meaning it leads to a more standardized environment and consistent behaviour;
- **Trade-offs/Disadvantages:** Supports web-based applications only.

3.2.3 Playwright

Playwright [52] is a Node.js library made for browser automation by Microsoft. It is possible to perform automated E2E testing using this free and open source tool.

- **Open Source/Free:** Playwright is a free and open source option that provides fast and reliable E2E testing for modern web apps;
- **Browser Support:**
  - Google Chrome;
  - Mozilla Firefox;
  - Microsoft Edge;
  - Opera;
  - Apple Safari.
- **Language Bindings:**
  - Java;
  - Python;
  - C#;
  - JavaScript;
  - TypeScript.
- **Support Documentation:** The official documentation is very complete and straightforward. The Playwright's website also references the community posts both on GitHub and Stackoverflow, which is a great help;

- **CI/CD Integration:** This feature is present in Playwright;
- **Parallelism:** This tool runs tests in parallel, using various worker processes that run at the same time. Parallelism is a default feature of Playwright however, tests in a single file run in order in the same worker process. This feature can be tailored by the programmer;
- **Locators/Selectors:**
  - Text;
  - CSS;
  - Attribute;
  - XPath;
  - React (experimental);
  - Vue (experimental).
- **Flakiness:** One of the main properties of Playwright is resilience, which avoids flaky tests. Some properties that guarantee consistent tests are:
  - auto-wait, since it waits for elements to be rendered and ready before performing actions, erasing the need for artificial waits;
  - web-first assertions, where they are created specifically for the dynamic web and checks are automatically retried until they meet the necessary conditions to continue;
  - tracing, where it is possible to configure the strategy of test retries and the snapshot of the execution to better understand and eliminate flakes.
- **Key Features:** It has a code generator, where it is possible to record a test and Playwright will provide the code;
- **Trade-offs/Disadvantages:** There is no support for legacy versions of Microsoft Edge and Internet Explorer or mobile devices.

3.2.4 TestCafe

TestCafe [53] runs on Node.js and its main purpose is to provide a simple experience for the programmer since the installation process is very simple, as is the programming of tests.

- **Open Source/Free:** TestCafe is free and open source, distributed under the MIT license;
- **Browser Support:**
  - Google Chrome (and mobile);
  - Microsoft Edge;

  - Internet Explorer (11+);
  - Mozilla Firefox;
  - Opera;
  - Apple Safari (and mobile).

- **Language Bindings:**
  - JavaScript;
  - TypeScript.
- **Support Documentation:** The official documentation is vast and complete, allowing the programmer to grasp the purpose and utility of various functions and technologies of TestCafe;
- **CI/CD Integration:** TestCafe supports various technologies to achieve CI/CD integration;
- **Parallelism:** TestCafe supports parallelism since it can run multiple tests concurrently;
- **Locators/Selectors:**
  - ID;
  - Name;
  - CSS Selector;
  - DOM Selector;
  - XPath.
- **Flakiness:** TestCafe reduces flakiness by using good code practices. For example, the smart assertion query mechanism is great, since it is possible to specify a timeout and guarantee the assertion executes as planned;
- **Key Features:**
  - Clean and concise code, providing a simpler experience for the programmer;
  - Human readable code;
  - Capable of using multiple browser windows to test complex interactions.
  - Allows testing with connectivity issues by simply feeding sample data through mock requests to emulate HTTP responses.
- **Trade-offs/Disadvantages:**
  - The browser does not recognize that test cases are being executed, leading to some errors in edge automation and debugging;
  - No control of the browser since TestCafe is incapable of so;
  - Events like clicks are simulated, which do not represent a real user experience. For example, an inactive button would be clickable by TestCafe.

3.2.5 Puppeteer

Puppeteer [54] is a Node.js library developed by Google with the main objective of supporting any chromium based browser. This high level API allows the programmer to manipulate the browser when running tests.

- **Open Source/Free:** This tool is open source and free;
- **Browser Support:**
  - Google Chrome;
  - Microsoft Edge;
  - Opera;
  - Mozilla Firefox (experimental).
- **Language Bindings:** JavaScript;
- **Support Documentation:** The official documentation covers all the functions of Puppeteer;
- **CI/CD Integration:** It is possible to integrate CI/CD into Puppeteer;
- **Parallelism:** Puppeteer′s exposure to browser contexts makes it possible to effectively parallelize test execution;
- **Locators/Selectors:**
  - ID;
  - Name;
  - CSS Selector;
  - DOM Selector;
  - XPath.
- **Flakiness:** One of the main pillars of Puppeteer is stability, since it prevents flaky tests and memory leaks. The event-driven architecture reduces flakiness a lot, eradicating the need to add artificial sleep commands;
- **Key Features:**
  - Really fast since it has nearly null overhead on an automated page;
  - Secure because it operates off-process with respect to Chromium i.e, it is safe to automate malicious pages.
- **Trade-offs/Disadvantages:**
  - Limited to Chrome, no Firefox support;
  - Smaller community support.

3.2.6 WebDriverIO

According to WebDriverIO [55] they represent a “Next-gen browser and mobile automation test framework for Node.js” rich in features for E2E automated browser testing.

- **Open Source/Free:** WebDriverIO is open source and free;
- **Browser Support:**
  - Google Chrome;
  - Microsoft Edge;
  - Internet Explorer;
  - Mozilla Firefox;
  - Opera
  - Apple Safari.
- **Language Bindings:** JavaScript;
- **Support Documentation:** Like the majority of browser testing tools, WebDriverIO offers a catalog of the functionalities it provides, as well as guides and other core concepts explained in a concise manner;
- **CI/CD Integration:** WebDriverIO supports CI/CD integration;
- **Parallelism:** In the event of having multiple test files, it is possible to run them concurrently;
- **Locators/Selectors:**
  - ID;
  - Link Text;
  - Aria roles;
  - Name;
  - CSS Selector;
  - DOM Selector;
  - XPath.
- **Flakiness:** Since WebDriverIO supports implicit timeouts that specify how long the driver should wait until the supposed element appears, it can avoid interactions with components that have not loaded or are not available;
- **Key Features:**
  - WebDriverIO is extendable since it is really simple to add helper functions or other existing commands;
  - Compatibility is also a key feature because this tool can be executed on the WebDriver Protocol for true cross-browser testing.

- **Trade-offs/Disadvantages:**
  - More effort to set up a browser driver;
  - Slower than most testing tools.

3.2.7 Katalon Studio

Katalon Studio [56] is a low code option that can be used for automated E2E testing.

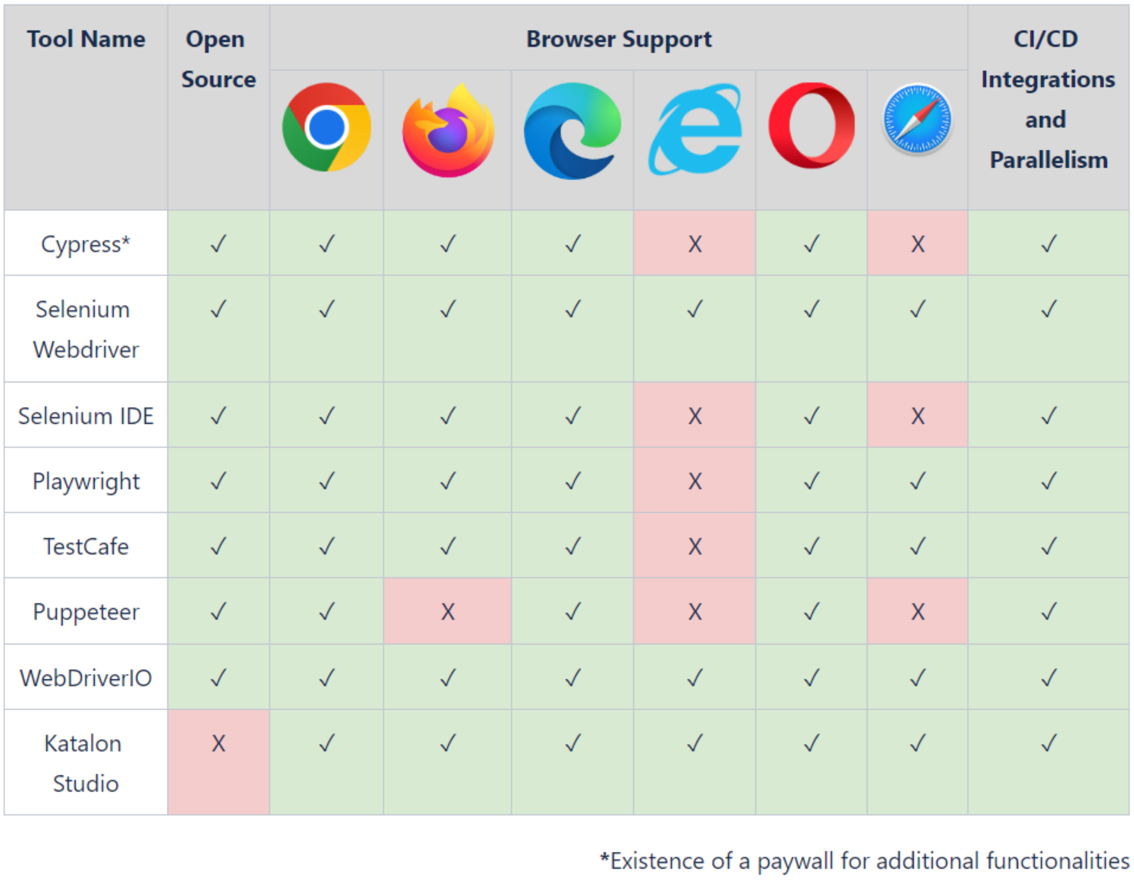
- **Open Source/Free:** Katalon Studio is not open source. Additionally, it offers various paid plans to unlock different functionalities;
- **Browser Support:**
  - Google Chrome;
  - Mozilla Firefox;
  - Microsoft Edge;
  - Internet Explorer;
  - Apple Safari;
  - Opera.
- **Language Bindings:** Since Katalon Studio is a low code option, there is no programming involved;
- **Support Documentation:** The official documentation is vast and easy to understand. Since Katalon Studio is a low code option, lots of study material is in the form of videos;
- **CI/CD Integration:** Katalon Studio provides automation testing guaranteeing CI/CD integration;
- **Parallelism:** It is possible to manage test suites in Katalon Studio and activate parallel mode to execute tests concurrently;
- **Locators/Selectors:**
  - Simple locators: id, name, class, tag, link text and attribute;
  - Advanced locators by combining the simple locators above: XPath and CSS;
  - Using functions in XPath: contains, sibling and ancestor.
- **Flakiness:** Katalon Studio understands the existence of flaky test cases and monitors their flakiness. Additionally, it makes it possible to study their origins to help the tester debug them.
- **Key Features:**

- Low code, meaning the tester is not required to have coding skills;
- Very high level tool.

- **Trade-offs/Disadvantages:**
  - It is not open source;
  - Existence of a paywall for important features.

3.2.8 Summary Board

The generic information of each investigated tool can be found in the comparative board illustrated by Figure 3.2.



Tool Name	Open Source	Browser Support						CI/CD Integrations and Parallelism
Cypress*	✓	✓	✓	✓	X	✓	X	✓
Selenium Webdriver	✓	✓	✓	✓	✓	✓	✓	✓
Selenium IDE	✓	✓	✓	✓	X	✓	X	✓
Playwright	✓	✓	✓	✓	X	✓	✓	✓
TestCafe	✓	✓	✓	✓	X	✓	✓	✓
Puppeteer	✓	✓	X	✓	X	✓	X	✓
WebDriverIO	✓	✓	✓	✓	✓	✓	✓	✓
Katalon Studio	X	✓	✓	✓	✓	✓	✓	✓

*Existence of a paywall for additional functionalities

Figure 3.2: Summary board of the software testing tools for E2E testing.

3.2.9 Auxiliary Software

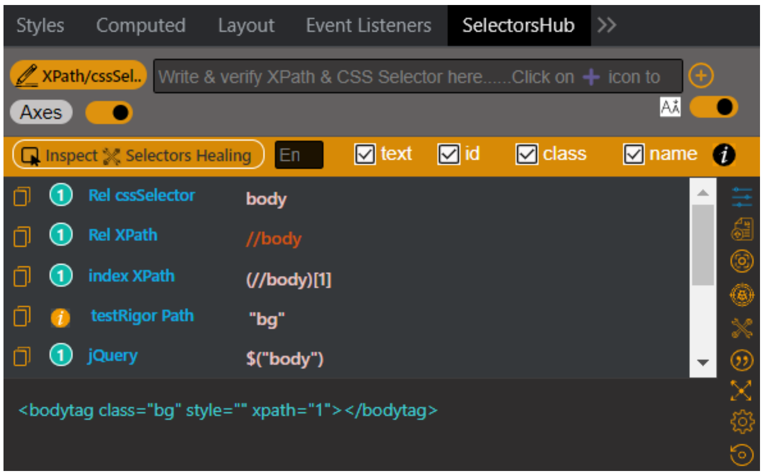
This subsection focuses on enumerating different tools that enhance the performance of the programmer implementing E2E testing.

CodeceptJS

This framework integrates with different helpers (Playwright, WebDriverIO, Puppeteer and TestCafe) where each test is written like a linear scenario with a special BDD style syntax [57]. Since it integrates with multiple testing tools, it can inform the programmer about which driver to choose for a specific restriction. For example, if one of the requisites is cross-browser support, then Selenium would be a great choice.

Selectors Hub

This tool is an extension available for Chrome [58], Firefox, Edge, Opera, Brave, Safari, Chromium and Tor, consisting of a free and next-gen XPath and CSS selector. To use it, the tester must install the extension and inspect the page elements. Then, identify the XPath of a specific element, as shown in Figure 3.3.



Rel cssSelector	body
Rel XPath	//body
index XPath	(//body)[1]
testRigor Path	"bg"
jQuery	$("body")

```
<bodytag class="bg" style="" xpath="1"></bodytag>
```

Figure 3.3: Demonstration of Selectors Hub tool.

TestCase Studio

Like Selector's Hub, this software is a browser extension available for Chrome [59], Firefox, Edge, Opera, Brave, Safari, Chromium and Tor. TestCase Studio's main feature is a recorder where it is possible to simulate user actions and generate XPath, CSS Selector and automation code for the recorded steps. Along with the test case, it also records a screenshot, which helps the programmer to memorize each step.

Chapter 4

Proposed Solution

In this chapter, every step that led to the current solution regarding component and E2E tests will be outlined. This approach was created by carefully analysing the state of art, as well as the particular requirements and limitations of the current quality process at Stratio Automotive. It is important to firstly analyse the **Stratio Foresight Platform**, followed by the quality assurance process of the company in order to identify the weak points and tackle them with a solution.

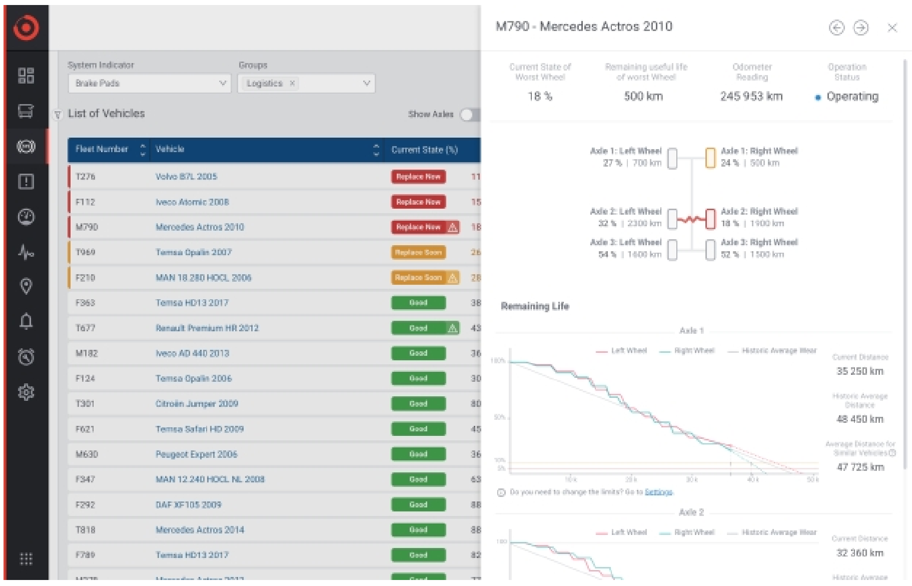
A Proof of Concept (POC) for both types of tests was implemented and assessed to confirm the viability of the suggested approach. Following that, a test suite for component and E2E tests was developed and implemented in the GitLab CI pipeline where various metrics were retrieved to measure the impact of the proposed solution.

4.1 Stratio Foresight Platform

The Stratio Foresight Platform (Figure 4.1) is the point of interaction between fleet managers and their automotive fleet. This web application allows its users to monitor the state of their vehicles through various metrics and properties, as well as the performance of their drivers.

Studying the platform thoroughly is crucial to understand its testing needs and potential corner cases. An example is a scenario where it is necessary to have multiple tabs open to perform a specific functionality.

The following sub-sections breakdown the platform into target test repositories and present features.



System Indicator: Brake Pads

Groups: Logistics

List of Vehicles

Show Axles

Fleet Number	Vehicle	Current State (%)
T276	Volvo B7L 2005	Replace Now 11
F112	Iveco Atomic 2008	Replace Now 15
M790	Mercedes Actros 2010	Replace Now ⚠ 18
T969	Temsa Opalin 2007	Replace Soon 26
F210	MAN 18.280 HOCL 2006	Replace Soon ⚠ 28
F363	Temsa HD13 2017	Good 38
T677	Renault Premium HR 2012	Good ⚠ 43
M182	Iveco AD 440 2013	Good 36
F124	Temsa Opalin 2006	Good 30
T301	Citroën Jumper 2009	Good 80
F621	Temsa Safari HD 2009	Good 45
M630	Peugeot Expert 2006	Good 36
F347	MAN 12.240 HOCL NL 2008	Good 63
F292	DAF XF105 2009	Good 88
T818	Mercedes Actros 2014	Good 88
F789	Temsa HD13 2017	Good 82

M790 - Mercedes Actros 2010

Current State of Worst Wheel	Remaining useful life of worst Wheel	Odometer Reading	Operation Status
18 %	500 km	245 953 km	● Operating

Axle 1: Left Wheel 27 % | 700 km

Axle 1: Right Wheel 24 % | 500 km

Axle 2: Left Wheel 32 % | 2300 km

Axle 2: Right Wheel 18 % | 1900 km

Axle 3: Left Wheel 54 % | 1600 km

Axle 3: Right Wheel 52 % | 1500 km

Remaining Life

Axle 1

Current Distance 35 250 km

Historic Average Distance 48 450 km

Average Distance for Similar Vehicles 47 725 km

Do you need to change the limits? Go to Settings.

Axle 2

Current Distance 32 360 km

Figure 4.1: Example of the user interface of Stratio Foresight Platform [60].

4.1.1 Repositories

Stratio Automotive uses GitLab as a management platform for version control. While the Stratio Foresight Platform consists of multiple repositories, it is primarily composed of two key repositories that encapsulate the platform and its components.

Front-end Commons

The **Front-end Commons** repository contains all Angular components of the Stratio Foresight Platform. Naturally, all **component tests** will be performed within this repository. Before developing tests and implementing them in the pipeline, it is crucial to study its source code, since component tests are a white box testing approach. Then, it is important to understand how Storybook can be integrated into the repository and where to include its test runner in the GitLab CI pipeline. Figure 4.2 depicts the CI/CD pipeline of the repository.

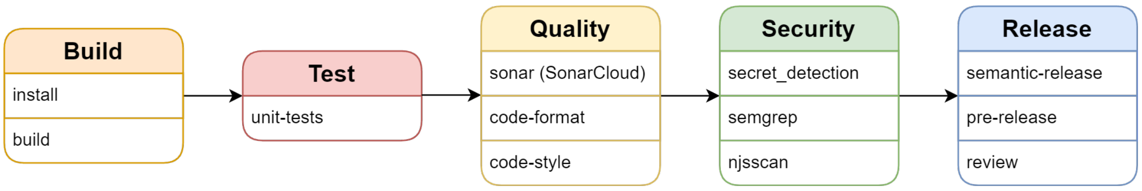


Figure 4.2: Schematic representation of the GitLab CI/CD pipeline of the Front-end Commons repository.

Stratio-Web

The **Stratio-Web** repository contains all the necessary code to build and deploy the Stratio Foresight Platform. All smoke tests and E2E tests will be developed within this repository, and executed against a deployed version of the Stratio Foresight Platform in the **staging environment**. Figure 4.3 depicts the CI/CD pipeline of the repository.

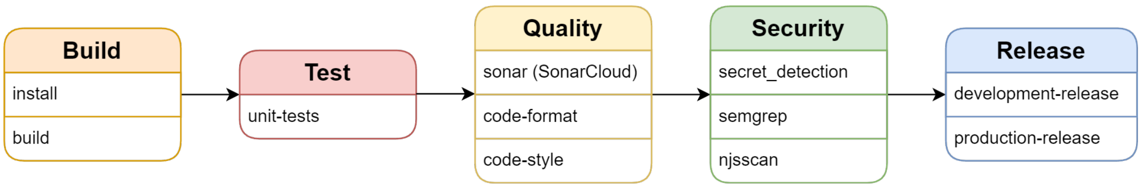


Figure 4.3: Schematic representation of the GitLab CI/CD pipeline of the Stratio-Web repository.

A brief rundown of the different stages are as follows:

- **Build:** installation of all dependencies, followed by the build of the project;
- **Test:** before the internship, **only unit testing is done on both repositories**;
- **Quality:** full code analysis regarding its format, style and quality with Sonar Cloud [61] to deliver clean and concise code consistently;
- **Security:** different scans to detect security breaches in code;
- **Release:** deployment process.

Both pipelines are extremely similar, with the **Release** stage being the main difference. In fact, the deployment process of each repository is different, due to Front-end Commons only containing the components that are used in Stratio-Web. Most of the changes that will be introduced will be included in the **Test** stage, with component tests in the Front-end Commons repository and E2E tests in the Stratio-Web repository.

4.1.2 Target Features for E2E Testing

Before listing all the target features for E2E tests, it is important to clarify that there are different types of users, with different privileges:

1. **Normal User:** the average end-user of the Stratio Foresight Platform;
2. **Master User:** similar to the normal user, with access to features that the normal user does not have;
3. **Super Master User:** end-user with maximum privileges, limited to the Stratio Automotive development team.

The target test features of the Stratio Foresight Platform are enumerated in Tables 4.1 and 4.2, with the **Super Master User** exclusive features marked as yellow.

Table 4.1: Stratio Foresight Platform features.

Stratio Foresight Platform		
Functionality	**Feature**	**Description**
Login	Authentication	Authenticate in the Stratio Foresight Platform
	Recover Password	Ask for a new password
Logout	Log Off	Perform logout
Dashboard	Edit Panels	Possibility of adding new panels or re-ordering them
Fleet Condition	Consult Vehicle Details	Examine details of a specific vehicle
	Consult Active Alerts	Examine the active alerts of a specific vehicle
	Consult Active DTCs	Examine the active DTCs of a specific vehicle
System Indicators	Overview	Examine various cards regarding system indicators statistics
	Starter Battery	Examine starter battery of all vehicles
	Brake Pads	Examine brake pads of all vehicles
	Available Engine Torque	Examine engine torque of all vehicles
	Air Leaks	Examine air leaks of all vehicles
	Battery Pack - EV	Examine battery pack of all EV vehicles
	Potential Fault	Submit form w/ potential faults of vehicles
Metrics	Throttle Pedal	Examine throttle pedal usage of all vehicles
	Coolant Temperature	Examine coolant temperature of all vehicles
	Consumption	Examine consumption of internal combustion, hybrids and electrics
	EV Charging	Examine EV charging of all vehicles
	Operation	Examine operation mode of all vehicles
Vehicle Recent Data	Vehicle Data Display	Search for a specific vehicle and consult its data
Occurrences	Filtering	Filter from existing occurrences
	Mark as Read	Mark a specific occurrence as read
Service Plans	Create Service Plan	Fill a form to create a new service plan
	Mark Service Plan as Read	Examine and mark a specific service plan as read
	Cleanup of Service Plan	Examine and delete a specific service plan
Reports	Consult Reports	Examine reports

Table 4.2: Stratio Foresight Platform features (cont.).

Functionality	**Feature**	**Description**
Maps	Authentication	Authenticate in the Stratio Foresight Platform
	Map	Interact with Google Maps widget and search drivers/vehicles
	Trips	Examine existing trips of various vehicles
	Driver Hours of Service	Examine driver's hours of services
	Geo-referenced Occurrences	Examine geo-referenced occurrences with the possibility of marking them as read
	Messages	Search existing messages and create new ones
	Operational Events	Examine operational events
	Geo-referenced Alerts	Search existing geo-referenced alerts and create new ones
	Shared Vehicle Locations	Search existing shared vehicle locations and create new ones
Ecodrive	Overview	Examine various cards regarding driving statistics and how ecological are they style of driving
	Driver Score	Examine driver scores of all drivers
	Vehicle Score	Examine vehicle scores of all vehicles (no need to validate export pdf)
	Bus Line Score	Examine bus line scores of all bus lines
	Driver Configuration	Examine driver's configurations of all drivers
	Groups Management	Search groups and create new ones
My Settings	General Settings	Select different languages and time zones
	Measurement Units	Select different measurement units for distance, temperature and pressure
	Notifications	Examine the notifications options
	Edit Profile	Examine edit profile form
Global Settings	Groups	Creation of new groups
	Idle Time	Examine idle time by vehicle type
	Ignored Trouble codes / Alerts	Examine the list of ignored trouble codes and alerts
List Users	Disable User	Disable a specific user
	Enable User	Enable a specific user
	Examine User	Examine a specific user

4.2 Quality Process

Before presenting the different types of testing executed, it is important to understand the current quality process at Stratio Automotive and study the most frail aspects within.

The main goal is to identify weaknesses and propose changes that will enhance the approval time of new iterations of the product, while also ensuring greater levels of quality.

4.2.1 Paradigm Before Proposed Changes

It is possible to categorize different procedures in the quality process at Stratio Automotive. During a development sprint, the Product Owner (PO) is responsible to **manually** validate bug fixes, new functionalities and to perform smoke tests on the whole platform before deploying it to the production environment. Bug fixes and new features will only be deployed simultaneously at the end of the sprint, even if it they were approved at the beginning.

Bug Fix Approval

This course commences through the identification of a defect in the platform. If one of the clients reports a bug to the team, the person responsible for the report will create a new **ticket** for the product team to validate its existence. Then, the PO tries to reproduce the defect in their system. If the bug is reproducible, the PO forwards it to the architecture or development team, according to its nature. After the defect is corrected, the PO validates it again and marks it as solved. The bug fix will then be available in the next deployment of the product. Figure 4.4 illustrates the course of the bug fix approval life cycle, from its creation until it is solved.

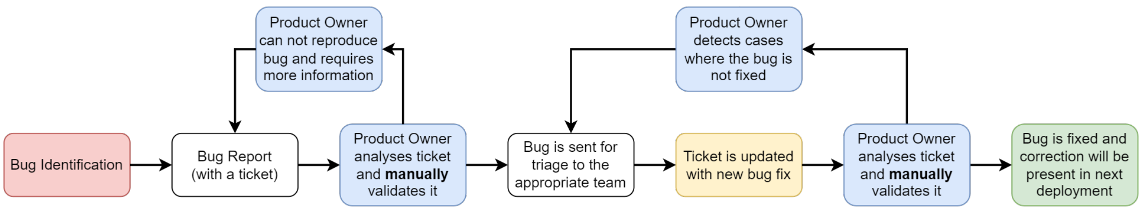

Product Owner can not reproduce bug and requires more information
Product Owner detects cases where the bug is not fixed
Bug Identification
Bug Report (with a ticket)
Product Owner analyses ticket and **manually** validates it
Bug is sent for triage to the appropriate team
Ticket is updated with new bug fix
Product Owner analyses ticket and **manually** validates it
Bug is fixed and correction will be present in next deployment

Figure 4.4: Schematic representation of the course of bug fix approval life cycle.

User Story Approval

In every sprint, a set of new features is settled through user stories. After the functionality is developed, the PO is responsible for approving and **manually** validating it to guarantee there are no bugs. Figure 4.5 represents the life cycle of a typical user story approval course: the course initiates with a developer finishing a functionality and submitting it for approval through a new ticket. Then, the PO **manually** validates it and marks it as done or sends it back for further development if bugs are found.

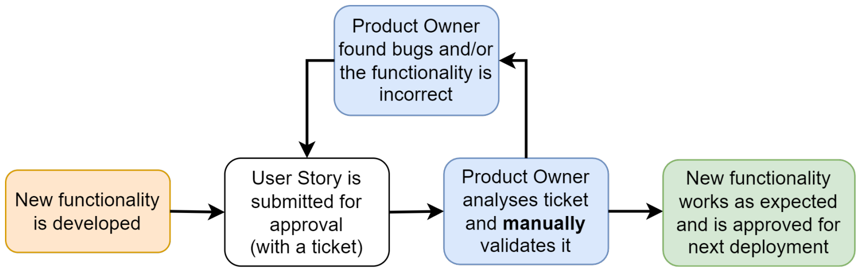

Product Owner found bugs and/or the functionality is incorrect
New functionality is developed
User Story is submitted for approval (with a ticket)
Product Owner analyses ticket and **manually** validates it
New functionality works as expected and is approved for next deployment

Figure 4.5: Schematic representation of the user story approval life cycle.

Full Product Validation

Right before the end of a sprint, the PO is responsible to **manually** validate the **Stratio Foresight Platform** before deployment. They have a set of test cases that are performed on the platform to ensure all the most important and crucial features are working as expected and the newly implemented changes have not introduced new defects or issues. Consequently, this course of validation takes up to **four hours** and quickly escalates if new bugs are found.

4.2.2 Solution Purpose and Changes

The efforts presented in this Dissertation will allow for changes in the product validation method so that automated testing becomes a necessity. The main objective is to launch bug fixes and user stories automatically into the production environment without validation from the PO. Furthermore, the addition of E2E tests will also reduce the involvement of the PO in the full product validation while also providing **instant feedback** on new changes made by the development team. In other words, the present work aims to deploy the product through **continuous deployment**.

Currently, the company gathers all new functionalities and bug fixes and then deploys them at the same time. With the automated tests, it would be possible to iteratively send them to the production environment, reducing delays between the development team and the product team.

The most frail points in the current quality process at Stratio Automotive are the manual validations that the product team is responsible for. The main objective is to **reduce** most of the manual validation the PO needs to do for each ticket until the process is **completely automated**.

This strategy would diminish the effort needed by PO's to constantly validate the product and reduce back-and-forth tickets with the development team. On the other hand, the development team is prone to receiving instant feedback regarding their changes, which may **prevent or identify bugs earlier**, since most of the bugs would be caught during the automated testing in the CI/CD pipeline.

Bug Fix Approval

Figure 4.4 highlights that the PO is responsible for asserting the existence of the bug and redirecting it to the appropriate team for fixing. Furthermore, it is possible to optimize the correction of the defect with automated tests.

Figure 4.6 illustrates that a test case for the bug fix should also be developed while the appropriate team corrects it. The changes are then validated through automated tests, and the bug fix is continuously deployed into the live production environment. Therefore, the fix is delivered to end-users thoroughly tested and without delays.

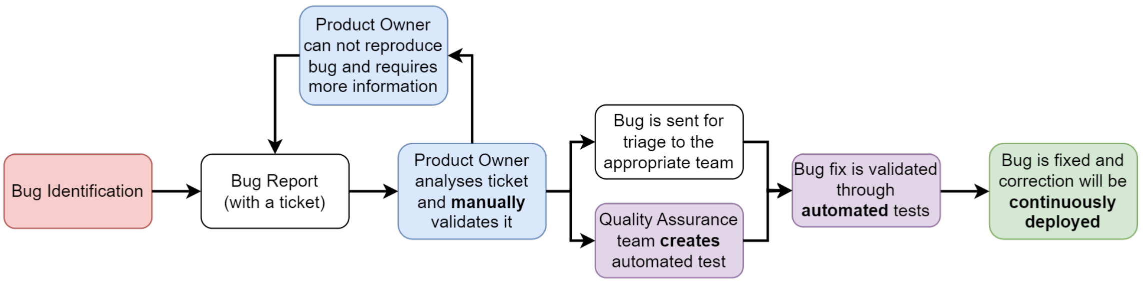

Bug Identification
Bug Report (with a ticket)
Product Owner can not reproduce bug and requires more information
Product Owner analyses ticket and manually validates it
Bug is sent for triage to the appropriate team
Quality Assurance team creates automated test
Bug fix is validated through automated tests
Bug is fixed and correction will be continuously deployed

Figure 4.6: Schematic representation of the proposed change for the course of bug fix approval.

User Story Approval

Figure 4.5 features the manual validation of the PO for each user story of the current sprint. The proposed change includes the development of the test case along with the new feature. Then, the functionality is automatically tested and continuously deployed. Figure 4.7 illustrates the proposed user story approval life cycle.

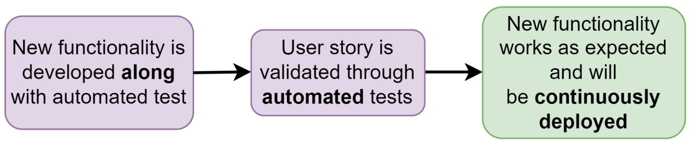

New functionality is developed along with automated test
User story is validated through automated tests
New functionality works as expected and will be continuously deployed

Figure 4.7: Schematic representation of the proposed change for the user story approval.

Full Product Validation

Since the PO is responsible for manually validating the Stratio Foresight Platform before deployment, the logical strategy would be to create various test cases that cover most of the important functionalities to reduce the amount of time the PO spends validating the software. This approach is suitable for Stratio Automotive's current workflow since it will allow continuous integration through automated testing and continuous deployment of the product.

Increasing Coverage

Since the **software team only has a set of unit tests ready**, it is important to increase the coverage through another type of testing. Following the Testing Pyramid, it is logical to perform component tests, since they represent the next level of testing. By increasing the coverage with these techniques, it is possible to prevent bugs early on in development, reducing the cost and effort of fixing them. Additionally, team confidence increases due to already testing critical features and resolving potential defects.

4.3 POC for Component Tests

Since the aim repository to conduct component tests used Nx[1] with Angular, it was pertinent to choose testing tools available in the official Nx packages[2]. Cypress [2] and Storybook [4] were then selected from the state of art to integrate the monorepo development environment and showcase the generic capabilities of each one.

The main focus was to select the most pertinent that could aid in development and increase testing coverage of the Angular components. During this section, various key points like development cost and documentation quality will be highlighted, with the experimental conditions and test cases defined in Appendix A.1.

4.3.1 Tools

As previously declared, Cypress and Storybook were compared in order to select the most fitting tool to Stratio Automotive's component library.

The main key points made in the presentation of the POC for component tests are as follows:

1. **Setup Complexity:** setting up Cypress and Storybook for component tests can be complex for specific frameworks. Since the intended repository used

[1]`https://nx.dev/`

[2]`https://nx.dev/packages`

Nx with Angular for developing the components, a strategic approach was used to integrate both testing tools, where they were installed through the official Nx packages. These consist of a set of tools and utilities that integrated either Cypress or Storybook to the Nx monorepo development environment. Table 4.3 represents the complexity of integrating the testin tools into the monorepo development workflow;

Table 4.3: Description of the scale for Setup Complexity.

Scale	Setup Complexity
★☆☆☆☆	More than 8 hours
★★☆☆☆	6 to 8 hours
★★★☆☆	4 to 6 hours
★★★★☆	2 to 4 hours
★★★★★	Less than 2 hour

2. **Development cost:** what tool was the most dispendious to develop both in time and effort. While Cypress provided a **component workbench** to build and test multiple components, Storybook used **stories**, which capture the rendered state of a Web UI component. Table 4.4 describes the effort needed to program the test cases;

Table 4.4: Description of the scale for Development Cost.

Scale	Development Cost
★☆☆☆☆	More than 4 hours
★★☆☆☆	3 to 4 hours
★★★☆☆	2 to 3 hours
★★★★☆	1 to 2 hours
★★★★★	Less than 1 hours

3. **Documentation:** weighted how much each tool's documentation helped during the preparation of the test cases, both in terms of coding conventions and generic issues that arose. Table 4.5 describes the quality of each tool's documentation;

Table 4.5: Description of the scale for Documentation Quality.

Scale	Documentation Quality
★☆☆☆☆	No official documentation.
★★☆☆☆	Frail documentation with only basic examples.
★★★☆☆	Difficult to find the solution to eventual problems.
★★★★☆	Covers most of the frequent issues and theoretical elements.
★★★★★	Complete documentation with examples and technical aspects.

4. **Development process:** brief description of the advantages and difficulties of each testing tool.

Cypress

Even though the main focus of this tool is E2E testing, Cypress also offers an alternative to mount components and test the logic within. It provides a component workbench to build and test multiple components, while offering the strong selectors and assertions available from the main API. Additionally, it offers **spies** to validate changes within the component and intersect requests.

Storybook

This front-end workshop allows assembling Web UI components and pages in isolation. The main key point about this tool is the large library of different **add-ons** with various functionalities. It uses **stories**, which capture the rendered state of the component and simulates user interactions through the **interactions add-on** [62]. It also includes a test runner [63] that turns all stories into executable tests, capable of running in the CI pipeline and reporting their test coverage.

4.3.2 POC Closure

A meeting with the front-end team was conducted to present a comparison between each tool and decide which was the most appropriate to perform component tests. At the end of the presentation and discussion, **Storybook was chosen as the preferred tool** since it also displayed advantages not only for component testing but also as a form of documentation for each component and accessibility analysis. The generic information for each investigated tool can be found in Table 4.6.

Table 4.6: General details regarding the POC for Component Testing.

	Cypress	**Storybook**
Setup Complexity	★★☆☆☆	★★★☆☆
Development Cost	★★★☆☆	★★★★☆
Documentation Quality	★★☆☆☆	★★★★☆

4.4 Component Tests Implementation

After selecting Storybook [4] as the preferred tool to perform component tests, a list of all web components was created. Each one was categorized with a specific priority according to its importance, how frequently it was used in the Stratio Foresight Platform and the coverage impact (coverage-based approach). Table 4.7 describes each one of the selected components, along with the respective priority.

Table 4.7: General details regarding the selected components for testing.

Name	Priority	Description
Alert Modal	High	Represents a pop-up window with a message, buttons and an icon
Sub-Menu	High	Displays a lateral menu with sub-options and an embedded filter
Table	High	Data is shown in a tabular manner and users can interact with it
Drawer	High	Provides a sliding panel with additional information
Layout	High	Represents a side bar with various options and a logout button
Quick Search	Medium	Provides a text area for users to type data
Columns Selector	Medium	Small drop down with options regarding which columns to display in a table
Value Selector	Medium	Displays a selector where users can select one or more options from a predifined list of values
Label Highlight Box	Medium	Highlights key information through a border to emphasise it
Label	Medium	Represents text that identifies a specific property or information
Operation Status	Medium	Represents a small animation indicating if the system is active or not
Lists Resume	Medium	Displays a tooltip with a predefined list of items
Breadcrumbs	Low	Shows a trail of links to assist users whilst navigating
Page Header	Low	Similar to Breadcrumbs, with a reference to return to the home page
Sub-Menu Filters Button	Low	Button regarding the sub-menu filters
Table Export Button	Low	Button regarding the table export report functionality
Table Refresh Button	Low	Button to refresh contents of the table
Table Row Action Button	Low	Button to interact with a specific row of the table

4.4.1 Storybook Setup

As it was mentioned previously, the Front-end Commons repository (section 4.1.1) served as the conducted target to perform component tests. To install Storybook, it was important to integrate the pertinent package [64] into the monorepo. After following the installation steps, a set of basic **stories** (rendered state of a component) was created for each one of the components in the repository. Then, the configuration of the arguments for each component was made to ensure they were being correctly rendered. Figure 4.8 represents the Alert Modal component built and rendered in isolation by the Storybook UI, where the icon and all text labels were manually chosen to tailor the specific test needs (Figure 4.9).

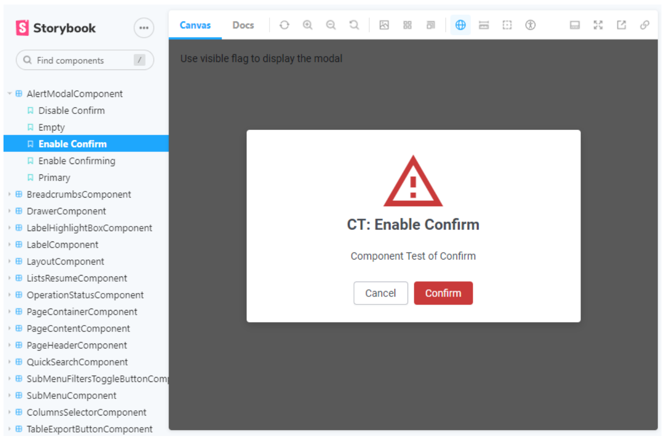
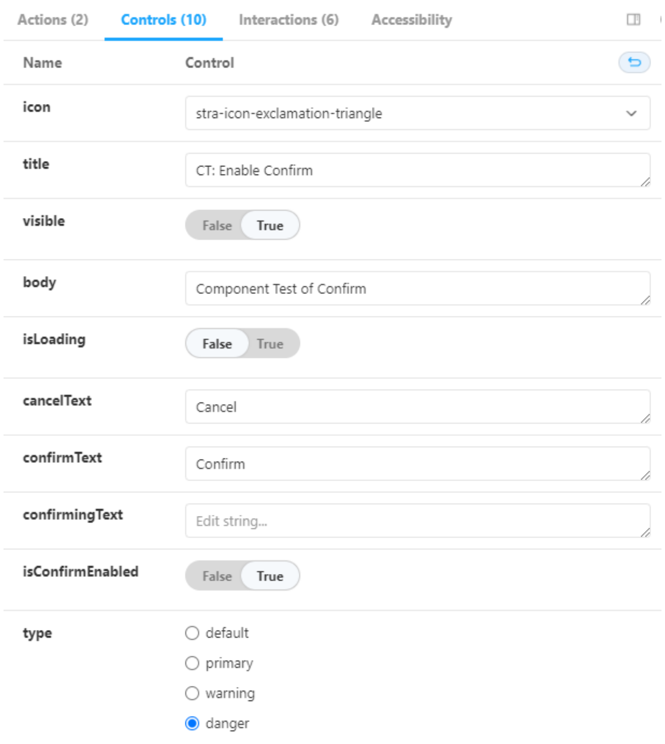


Figure 4.8: Storybook UI rendering the Alert Modal component.

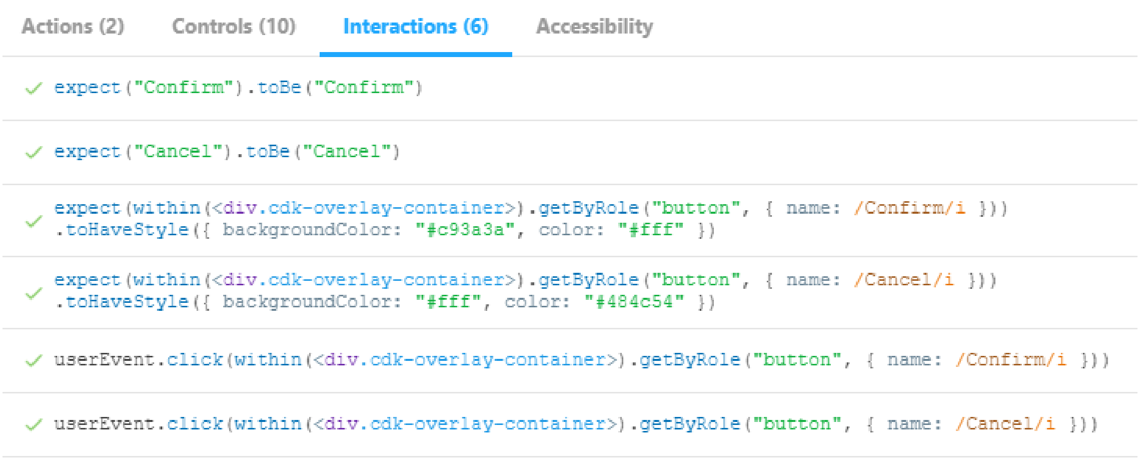


Name	Control
icon	stra-icon-exclamation-triangle
title	CT: Enable Confirm
visible	False [✓ True]
body	Component Test of Confirm
isLoading	[✓ False] True
cancelText	Cancel
confirmText	Confirm
confirmingText	Edit string...
isConfirmEnabled	False [✓ True]
type	default, primary, warning, [✓ danger]

Figure 4.9: Storybook UI stating the Alert Modal arguments.

In this stage of component testing, the Storybook UI allows components to be rendered and **manually** tested, since it is possible to edit the controls (Figure 4.9) and validate changes in real time within the component. Note that each component consists of a set of **different stories**, where each one incorporates a different set of controls and interactions for the component. In other words, **a story is a self-contained, isolated instance of a component with a previously defined set of input parameters.**

However, this form of component testing was not enough to validate all the functionalities of the component, given that some form of interaction to simulate an end-user using the component was crucial. Storybook offers an interactions add-on [62], which enables the development of play functions to integrate a specific story. A **play function** is a block of code to be executed after the story renders, enabling interactions within the component to trigger scenarios only possible through user intervention. The installation process of this add-on was straightforward, since it was only necessary to add the interactions package, the Testing Library[3] for locators and the Jest[4] add-on for assertions to the monorepo. Figure 4.10 illustrates the Storybook UI stating the interactions in the Alert Modal component from the play function.



```
✓ expect("Confirm").toBe("Confirm")
✓ expect("Cancel").toBe("Cancel")
✓ expect(within(<div.cdk-overlay-container>).getByRole("button", { name: /Confirm/i }))
  .toHaveStyle({ backgroundColor: "#c93a3a", color: "#fff" })
✓ expect(within(<div.cdk-overlay-container>).getByRole("button", { name: /Cancel/i }))
  .toHaveStyle({ backgroundColor: "#fff", color: "#484c54" })
✓ userEvent.click(within(<div.cdk-overlay-container>).getByRole("button", { name: /Confirm/i }))
✓ userEvent.click(within(<div.cdk-overlay-container>).getByRole("button", { name: /Cancel/i }))
```

Figure 4.10: Storybook UI stating the Alert Modal interactions.

After having all the stories and play functions developed, it was necessary to find a way to automate the testing process and execute them with a coverage report. Storybook offers another add-on named Test Runner [63], powered by Jest and Playwright, which converts all the stories and play functions within into executable tests on different browser engines (Chromium, Firefox and Webkit).

To obtain testing coverage, Storybook also provides a coverage add-on [65]. However, since the repository we are working on uses Angular configured with Webpack, it is required additional configuration to enable code instrumentation i. e. the ability to modify source code in order to collect information about its run time. Storybook points to a foreign repository to enable code instrumentation in an Angular project configured with Webpack [66].

After Storybook was thoroughly configured, a collection of test cases was written for each component to cover all of its functionalities. Appendix B.1, includes all the test cases for component testing.

[3]https://testing-library.com/
[4]https://storybook.js.org/addons/@storybook/addon-jest

4.4.2 Achieved Coverage

During this sub-section, various metrics regarding the testing coverage for component tests will be presented. There are three different testing periods:

1. **Before adding interactions add-on and performing the POC:** the test runner only verifies whether the component was rendered without errors;
2. **After POC:** includes the test cases presented during the POC, refering to Tables A.1, A.2 and A.3;
3. **After performing all the test cases:** includes coverage for all test cases (Appendix B.1) developed for the components selected from Table 4.7.

Storybook Test Runner can use the generated testing data to create LCOV [67] reports, which create an HTML page with data regarding the percentage of lines, functions and branches that were executed during testing. Table 4.8 summarizes the percentage of lines covered in different testing periods.

Table 4.8: Percentage of lines covered of all components.

Name	**Priority**	**Covered Lines** (%)		
		Before Interactions	After POC	After all test cases
All Components	N/A	50.36	54.18	83.69
Alert-Modal	High	78.94	84.21	94.73
Sub-Menu	High	42.55	68.00	78.00
Table	High	50.27	54.70	69.18
Drawer	High	84.61	84.61	100
Layout	High	87.50	87.50	87.50
Quick Search	Medium	73.07	73.07	96.15
Columns Selector	Medium	53.84	53.84	92.30
Value Selector	Medium	53.57	53.57	88.46
Label Highlight Box	Medium	100	100	100
Label	Medium	43.47	43.47	100
Operation Status	Medium	100	100	100
Lists Resume	Medium	73.91	73.91	95.65
Breadcrumbs	Low	100	100	100
Page Header	Low	88.23	88.23	93.33
Sub-Menu Filters Button	Low	100	100	100
Table Export Button	Low	83.33	83.33	100
Table Refresh Button	Low	75.00	75.00	100
Table Row Action Button	Low	60.00	60.00	100

Note that some components, like the Sub-Menu Filters Button, already had perfect coverage without any interactions. This means that the rendering of the component itself in different browser engines covers all the lines of the source code.

Additionally, it is crucial to clarify that the component tests allow for style assertions and confirmation of animations within the component.

4.4.3 Integration in the CI Pipeline

To introduce the automation of component tests in the existing GitLab CI pipeline (Figure 4.2), a new job named **component-tests** was created to be executed during the **Test** stage. Storybook Test Runner can use the generated testing data to create reports through LCOV [67], which is a tool able to provide insights of the covered code of the component tests. The reports can be directly fed to Sonar Cloud through the automated tests performed in the CI pipeline, depicted in Figure 4.11. The main advantage of this integration was the ability to run visual regression tests on Storybook components and assess code coverage for the components being produced, as well as providing a coverage report to Sonar Cloud[5].

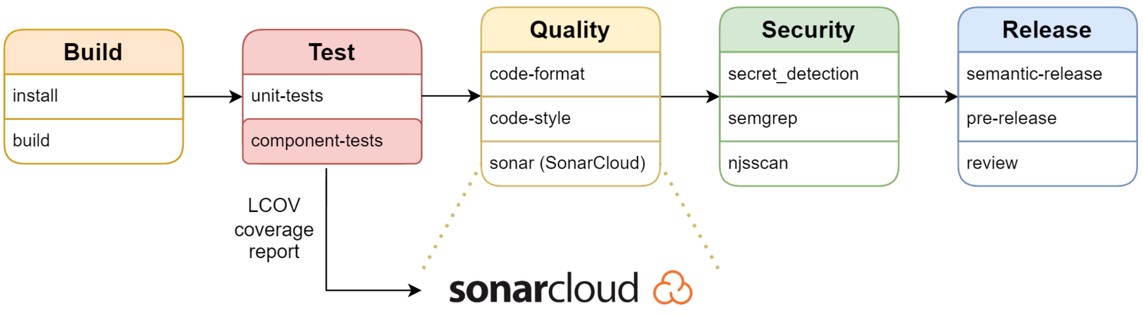


Figure 4.11: Schematic representation of the GitLab CI/CD pipeline of the Front-end Commons repository after performing component tests.

4.4.4 Component Tests Parallelism

Storybook Test Runner is able to use parallelism in component testing to reduce the time it takes to execute test suites. The main objective is to reduce testing time by selecting the optimal number of maximum workers. Appendix B.2 contains the raw data and computer specifications used to conduct this study.

In order to study how efficiently Storybook Test Runner manages to run component tests concurrently, the test suite composed for all the components was executed within the GitLab pipelines, varying the number of maximum workers **from one to eight, with ten repetitions of each execution**. It is important to note that the CI pipeline used by Stratio Automotive has the GitLab workers hosted locally (within the company's server), meaning that it is not optimal to use as many Storybook workers as possible. Figure 4.12 represents the execution time of Storybook Test Runner with a different amount of maximum workers.

[5]`https://docs.sonarcloud.io/`

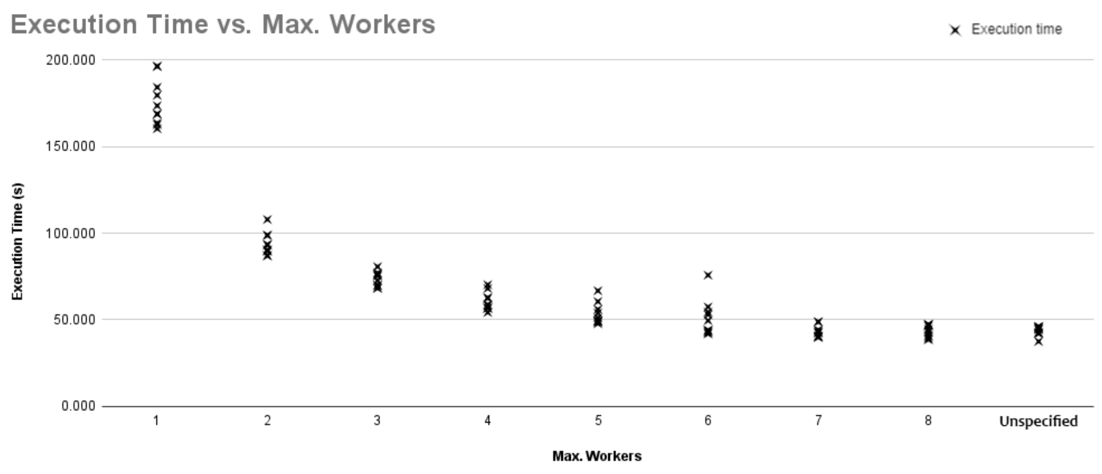


Figure 4.12: Execution time of Storybook Test Runner with different maximum workers.

A brief analysis states that beyond six workers, the execution time is nearly equal to an unspecified amount of workers, meaning that the execution time of Storybook Test Runner does not benefit significantly from an increase in more than seven workers. Table 4.9 represents different metrics, including the best, worst and average execution times, as well as the standard deviation.

Table 4.9: Storybook Test Runner Parallelism Statistics.

Max Workers	**Parallelism Statistics (seconds)**			
	Best Execution	Worst Execution	Average Execution	Standard Deviation
1	160.281	196.381	175.399	13.305
2	86.885	107.897	93.617	6.542
3	68.197	80.676	73.393	4.239
4	54.245	70.250	60.520	5.280
5	47.762	66.769	53.305	6.184
6	41.877	75.772	50.597	10.446
7	39.809	48.897	43.146	3.341
8	38.513	47.318	43.445	3.508
Unspecified	37.495	46.158	43.661	2.651

To better understand the parallelism statistics, Figure 4.13 displays a chart for a more complete analysis regarding the execution time and maximum workers.

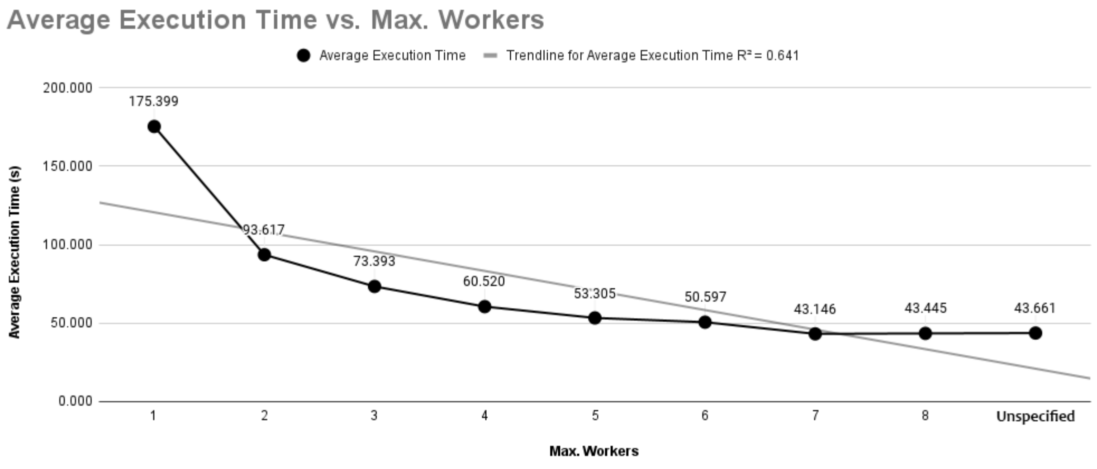


Figure 4.13: Average Execution time of Storybook Test Runner with different maximum workers.

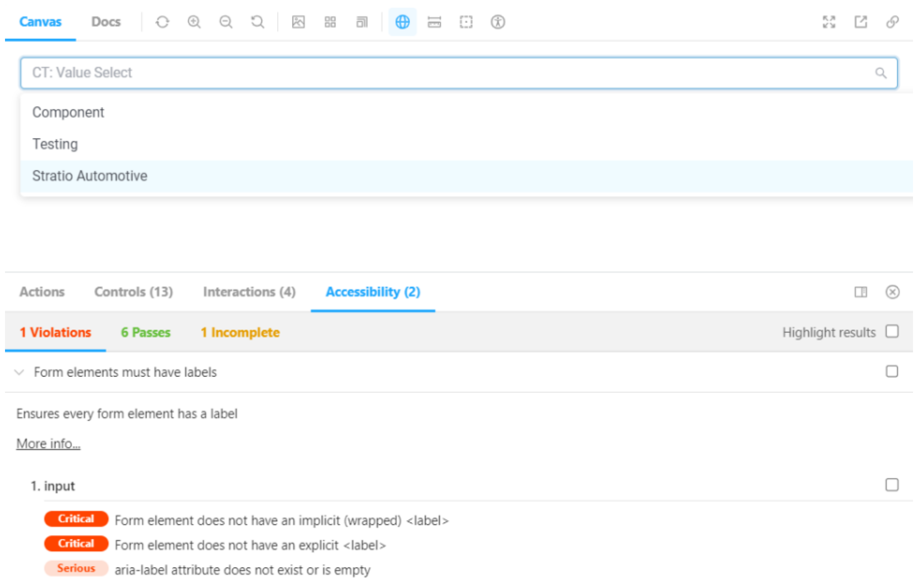
A few considerations include:

- It is **not worth it to go beyond seven workers**, since the performance gains are not significant;
- When the component-tests job is being executed, it is **not** ideal to have all the resources of that GitLab worker allocated to the Storybook Test Runner, since more intensive jobs might be running in the same worker;
- The standard deviation is quite large until six workers (between 11.747 and 4.086 seconds), meaning that the execution time varies significantly. The lowest standard deviation occurs when there is an unspecified amount of workers;
- The R-squared is 0.641, meaning the number of workers used has a moderate impact in execution time.

After analysing the data and considering the execution time, it was decided to maintain an unspecified amount of workers when executing the test suite in Storybook Test Runner. This approach allows reasonable execution times and does not bottleneck the local GitLab runners.

4.4.5 Influence on the Development Team

Even though Storybook proved to be an excellent tool to perform component tests, it also assists the development of Angular components and enhances the programming experience of the front-end development team. To help the programmer creating easy to use components, the accessibility add-on [68] was included in the project. The main goal is to make the UI as accessible as possible, alerting the programmer of possible adjustments to enhance the components of the Stratio Foresight Plataform. Figure 4.14 illustrates the accessibility add-on stating design violations in the Value Selector Component.



CT: Value Select

Component
Testing
Stratio Automotive

1 Violations 6 Passes 1 Incomplete

Form elements must have labels

Ensures every form element has a label

More info...

1. input

Critical Form element does not have an implicit (wrapped) <label>
Critical Form element does not have an explicit <label>
Serious aria-label attribute does not exist or is empty

Figure 4.14: Storybook UI stating design violations in the Value Selector Component.

Another remark is the ability to develop component tests and immediately validate applied changes visually on the Storybook UI instead of performing a new release/deployment every time, which is time consuming.

4.5 POC for End-to-End Tests

The POC for E2E tests created for Stratio Automotive consists of two phases. The first aims to present all the testing tools for E2E automation from a theoretical point of view. The second intends to use three automation testing tools to execute Web UI testing in the Stratio Foresight Platform.

4.5.1 Investigation Phase

Before collecting the use cases with the product team for the POC, various tools were selected to conduct a brief study and present a state of art for Web UI testing. Among the covered topics, the following were highlighted:

- **Research Initiatives:** certain appliances, like the Testing Pyramid, were revealed to be excellent starting points since they showcased all the testing levels and delineated the importance, granularity and execution time of each level;
- **State of Art:** eight options were studied and presented to the teams;

- **Relevant Options:** following the advantages and disadvantages of each state of the art option, three options were selected to cover use cases yet to be defined. The testing tools were: Cypress, Playwright and Puppeteer.

Alongside the theoretical information, a Cypress demo was displayed, showcasing the generic possibilities of using an automation tool for E2E testing as well as the current advantages and disadvantages of this technology.

4.5.2 Development Phase

The main objective of this phase aims to use three automation testing tools to execute Web UI testing in the Stratio Foresight Platform which is a control panel used by fleet managers. Inside this web application, the manager can monitor their vehicle, create planning services and control notifications. The experimental conditions and test cases are defined in Appendix A.2.

4.5.3 Tools

As was stated in the first phase of this POC, three different automated testing tools were selected for this project: Cypress, Playwright, and Puppeteer. Among the topics discussed, a theoretical explanation of each tool was given as well as a description of the test cases and a demo showcasing each tool. During the presentation, a code analysis was conducted before executing the E2E tests.

The main key points made in the second version of the presentation of the POC are as follows:

1. **Development Cost:** what tool was the most dispendious to develop both in time and effort. This is a difficult area to quantify because the development of test cases using the first tool comprises solution and logic. After programming the test cases with Cypress, the development with Playwright and Puppetter is straightforward since it only requires syntax changes of the code previously created. Table 4.10 describes the effort needed to set up the testing environment, understand the framework and program the test cases;

Table 4.10: Description of the scale for Development Cost.

Scale	**Development Cost**
★☆☆☆☆	More than 8 hours
★★☆☆☆	6 to 8 hours
★★★☆☆	4 to 6 hours
★★★★☆	2 to 4 hours
★★★★★	Less than 2 hours

2. **Documentation:** weighted how much each tool's documentation helped during the preparation of the test cases, both in terms of coding conventions and generic issues that arose. Table 4.11 describes the quality of each tool's documentation;

Table 4.11: Description of the scale for Documentation Quality.

Scale	Documentation Quality
★☆☆☆☆	No official documentation.
★★☆☆☆	Frail documentation with only basic examples.
★★★☆☆	Difficult to find the solution to eventual problems.
★★★★☆	Covers most of the frequent issues and theoretical elements.
★★★★★	Complete documentation with examples and technical aspects.

3. **Result Analysis:** benchmarks of time taken to execute the tests without parallelism (available for all tools) and with parallelism (Playwright exclusive since it was needed to set up a CI pipeline for the other tools). Table 4.12 describes the expected time for both tests;

Table 4.12: Description of the scale for Execution Time.

	Average Execution Time (t) (seconds)	
Scale	TC1 (UC1)	TC2 (UC2+UC3)
★☆☆☆☆	$t > 13$	$t > 19$
★★☆☆☆	$11 < t < 13$	$17 < t < 19$
★★★☆☆	$9 < t < 11$	$15 < t < 17$
★★★★☆	$7 < t < 9$	$13 < t < 15$
★★★★★	$t < 7$	$t < 13$

4. **Component Testing Basics:** Cypress and Playwright exclusive.

4.5.4 POC Closure

After presenting various theoretical aspects regarding E2E automation for front-end, the present teams (Product, Internal Tooling, Software) engaged in a discussion to decide which tool was most adequate for the project. At the end of the dialogue, **we collectively choose Playwright as the preferred tool** since it did not display any disadvantages to the pretended work and it also granted the advantage of having multiple pages (or tabs) open, where Cypress can not open more than once at a time. The generic information of each investigated tool can be found in Table 4.13.

Table 4.13: General details regarding the POC for E2E Testing.

	Cypress	Playwright	Puppeteer
Development Effort	★★★★☆	★★★★☆	★★★☆☆
Documentation	★★★★☆	★★★★☆	★★★☆☆
Execution Time	★★☆☆☆	★★★★★	★★★☆☆
Component Testing	Available	Available	Not Available

4.6 End-to-End Tests Implementation

After concluding and presenting the POC for E2E tests, a meeting with the product team was scheduled to gather insights about the Stratio Foresight Platform and write appropriate test cases for **full product validation**. Tables 4.14 and 4.15 summarize the covered features of the platform.

According to the information provided in Table 4.14, it has been confirmed by the product team that the **Dashboard** functionality will receive an overhaul and is set to be heavily modified, hence the status "No".

A specific feature can either be covered by:

- **Smoke Tests:** software testing process to validate if a specific functionality is stable, consisting of non-complex assertions and fewer interactions. If the build is stable, it is advisable to proceed towards more comprehensive software testing (User Acceptance Test (UAT) validation through E2E tests);
- **E2E Tests:** contains a battery of tests to simulate end-user behaviour on a specific functionality of the Stratio Foresight Platform. The primary objective of this form of validation is to guarantee that all parts of the system work together seamlessly within a real-world scenario.

Features marked in **yellow are exclusive to super master users** (section 4.1.2) and **blue features correspond to server-sided reserved features** which should be treated separately when multiple tests are running concurrently (in parallel). For instance, if the platform's language is switched in general settings, certain locators may become obsolete in other tests.

All **test cases** can be consulted in Appendix C.1.

Table 4.14: Stratio Foresight Platform Coverage of E2E tests

Stratio Foresight Platform		
Functionality	**Feature**	**Covered**
Login	Authentication	Yes (E2E Tests)
	Recover Password	Yes (E2E Tests)
Logout	Log Off	Yes (E2E Tests)
Dashboard	Edit Panels	No
Fleet Condition	Consult Vehicle Details	Yes (Smoke Tests)
	Consult Active Alerts	Yes (Smoke Tests)
	Consult Active DTCs	Yes (Smoke Tests)
System Indicators	Overview	Yes (Smoke Tests)
	Starter Battery	Yes (Smoke Tests)
	Brake Pads	Yes (Smoke Tests)
	Available Engine Torque	Yes (Smoke Tests)
	Air Leaks (Beta)	Yes (Smoke Tests)
	Battery Pack - EV	Yes (Smoke Tests)
	Potential Fault	Yes (E2E Tests)
Metrics	Throttle Pedal	Yes (Smoke Tests)
	Coolant Temperature	Yes (Smoke Tests)
	Consumption	Yes (Smoke Tests)
	EV Charging	Yes (Smoke Tests)
	Operation	Yes (Smoke Tests)
Vehicle Recent Data	Vehicle Data Display	Yes (E2E Tests)
Maps	Map	Yes (E2E Tests)
	Trips	Yes (Smoke Tests)
	Driver Hours of Service	Yes (Smoke Tests)
	Geo-referenced Occurrences	Yes (Smoke Tests)
	Messages	Yes (E2E Tests)
	Operational Events	Yes (Smoke Tests)
	Geo-referenced Alerts	Yes (E2E Tests)
	Shared Vehicle Locations	Yes (E2E Tests)
Occurrences	Filtering	Yes (E2E Tests)
	Mark as Read	Yes (E2E Tests)
Service Plans	Create Service Plan	Yes (E2E Tests)
	Mark Service Plan as Read	Yes (E2E Tests)
	Cleanup of Service Plan	Yes (E2E Tests)
Ecodrive	Overview	Yes (Smoke Tests)
	Driver Score	Yes (E2E Tests)
	Vehicle Score	Yes (E2E Tests)
	Bus Line Score	Yes (Smoke Tests)
	Driver Configuration	Yes (Smoke Tests)
	Groups Management	Yes (E2E Tests)
Reports	Consult Reports	Yes (Smoke Tests)

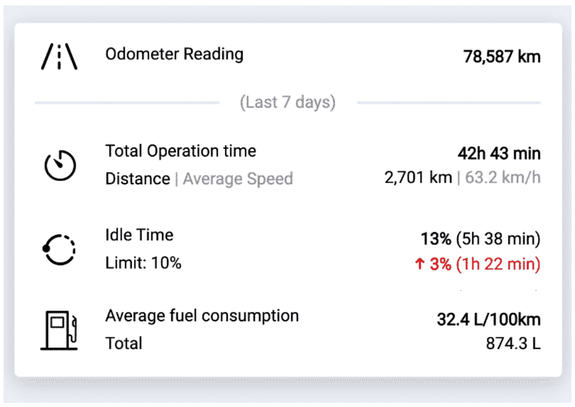
Table 4.15: Stratio Foresight Platform Coverage of E2E tests (Cont.)

Functionality	Feature	Covered
My Settings	General Settings	Yes (E2E)
	Measurement Units	Yes (E2E)
	Notifications	Yes (Smoke Tests)
	Edit Profile	Yes (Smoke Tests)
Global Settings	Groups	Yes (E2E Tests)
	Idle Time	Yes (Smoke Tests)
	Ignored Trouble codes / Alerts	Yes (Smoke Tests)
List Users	Disable User	Yes (E2E Tests)
	Enable User	Yes (E2E Tests)
	Examine User	Yes (Smoke Tests)

4.6.1 E2E Testing vs. Smoke Testing Criteria

Even though both kinds of tests serve different purposes in the upcoming test suite, it is important to explain what criteria were used to decide if a specific feature should be covered by smoke or E2E tests. Different features require a different level of interaction to be engaged by the end-user, hence the importance of classifying the tests (Tables 4.14 and 4.15) into different types.

Generally, **smoke tests** include more lightweight features, where the end-user only needs to log in and access a page with fewer ways of interaction. Looking at an example regarding the **Fleet Condition** functionality, more specifically in the **consult vehicle details** feature (Test Case C.4), mostly text assertions are performed, and the end-user only needs to browse the vehicle details page. Figure 4.15 depicts the **Operation Metrics** of the referred page, where only metrics are displayed.



Odometer Reading 78,587 km

(Last 7 days)

Total Operation time 42h 43 min
Distance | Average Speed 2,701 km | 63.2 km/h

Idle Time 13% (5h 38 min)
Limit: 10% ↑ 3% (1h 22 min)

Average fuel consumption 32.4 L/100km
Total 874.3 L

Figure 4.15: Operation Metrics component from the Fleet Condition functionality [5].

Alternatively, **E2E tests** include the most complex features of the Stratio Foresight Platform, where the end-user needs to perform a lot of steps to engage in different procedures. A possible example would be the **Service Plans** functionality, regarding the **service plan creation feature** (Test Case C.21), where the end-user must complete a three-step form. Figure 4.16 depicts one of the form components.

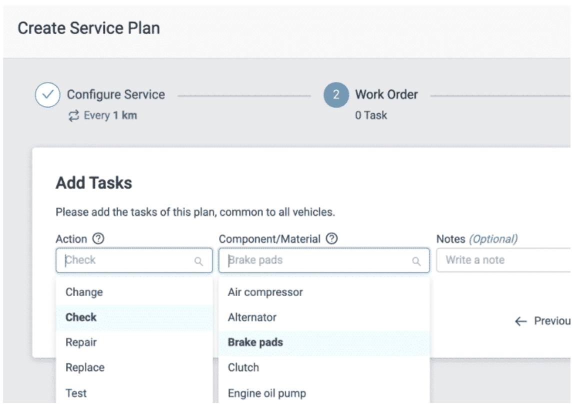


Figure 4.16: Form component from the Service Plans functionality [5].

4.6.2 Playwright Setup

The installation process of the automation framework [69] into the Stratio Web repository (4.1.1) was straightforward. After configuring TypeScript as the preferred language and specifying the target test folder, the next step involved setting up the **Playwright configuration file** [70]. The most relevant properties include:

- **Test timeouts:** since smoke and E2E tests are less granular than component tests, a timeout of one minute was specified for each test, to grant enough time to be executed;
- **Parallelism:** the ability to run tests concurrently in the Playwright test-runner was set to active, since it allows for faster execution, early detection of issues and better scalability;
- **Number of retries:** there is always the possibility of some assertion failing during testing. The number of test retries was set to **two**, to allow a test case to repeat in case of any flaky behavior being found. Additionally, the test trace of failing tests is stored to capture the step-by-step instructions of occurring errors;
- **Reporter type:** an HTML report was set to ensure an easy inquiry while validating test results;
- **Which browsers to use:** similarly to component tests, the three most common browser engines were set: **chromium**, **firefox**, and **webkit**.

After thoroughly configuring Playwright, a directory structure was decided, to keep a tidy work environment. Figure 4.17 depicts the chosen file arrangement:

- **Assertions:** includes one directory per functionality, populated with the assertions of each test case;

- **Test Fixtures:** contains one file per functionality, consisting of additional information per test case. For example, sample data of a specific vehicle to validate;
- **Test Pages:** in order to implement a POM approach (section 2.9.1), this directory consists of one folder per functionality, populated with the HTML locators and various methods of each test case;
- **Test Suite:** comprises directories organized by functionality and smoke tests, populated with test case specifications. Additionally, it includes a directory of scripts to execute all tests either locally or in the GitLab CI pipeline.

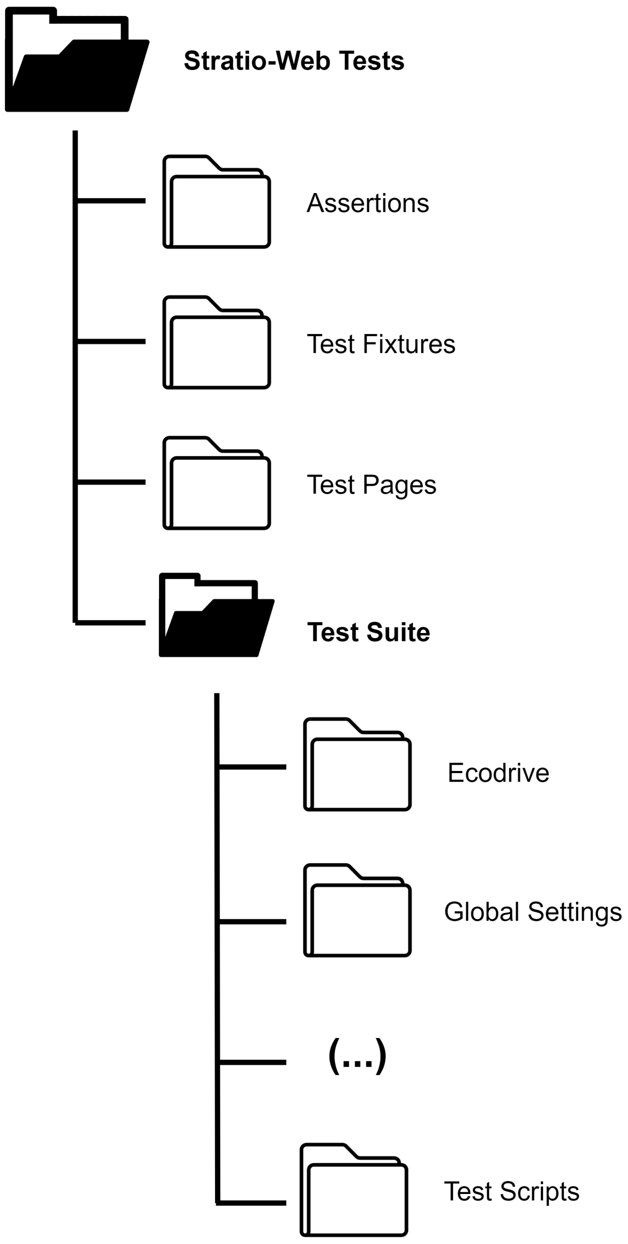


Figure 4.17: File structure of the test suite in Stratio-Web repository.

After defining a pertinent file structure, different categories were created to segregate tests, imposing the necessity of creating shell scripts to distribute the test cases across different testing phases:

- **Vault Auth Shell Script:** most test cases require login credentials to be executed. Since this information is sensitive and should not be displayed in plain text in the "Test Fixtures" directory, we used **Hashicorp Vault** [71] as a secret management solution in order to ensure a centralized platform for storing, managing, and accessing sensitive data. This shell script securely accesses Stratio Automotive's Vault and gets the necessary sample data to execute Stratio-Web tests;
- **Smoke Tests Shell Script:** this shell script executes all smoke tests, excluding those reserved for the super master user. Tables 4.14 and 4.15 indicate which are the smoke tests;
- **UAT Validation Shell Script:** this shell script executes all E2E test cases, **excluding** the super master and server-sided exclusive. Tables 4.14 and 4.15 indicate which are the E2E tests;
- **Server-sided Validation Shell Script:** this shell script executes all **server-sided exclusive** test cases. Tables 4.14 and 4.15 indicate which are server-sided exclusive features (marked as blue);
- **Security Validation Shell Script:** this shell script executes all **super master user exclusive** test cases. Tables 4.14 and 4.15 indicate which are the super master exclusive features (marked as yellow).

Figure 4.18 illustrates the file structure of the "Test Scripts" directory.

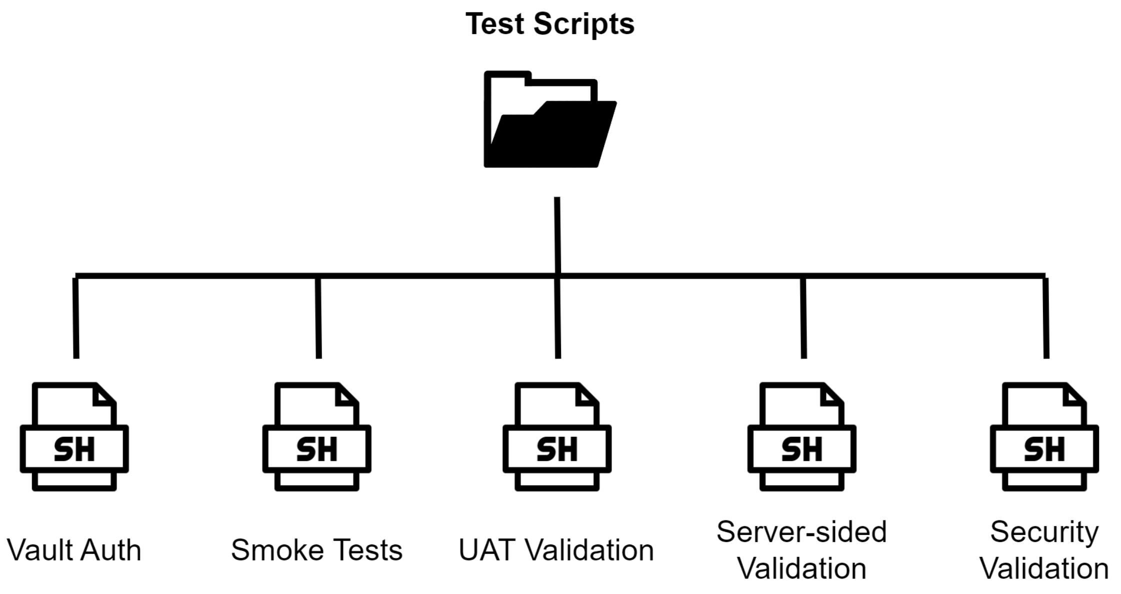


Figure 4.18: File structure of the Test Scripts directory.

4.6.3 Integration in the CI Pipeline

In order to establish the automation of all the smoke and E2E tests in the existing GitLab CI pipeline (Figure 4.3), a new job named **e2e-tests** was created to be executed during the **Test** stage. Similarly to the "Test Scripts" directory, the job was split into different stages that followed a specific execution order. Additionally, a project was included in the pipeline that allowed the environment to access Vault secrets, in order to securely retrieve sensitive data (authentication credentials). The e2e-tests job is partitioned into the following:

- **e2e-smoke-tests:** stage responsible for the execution of smoke tests;
- **e2e-uat-validation:** stage in charge of the execution of E2E tests, **excluding** the super master and server-sided exclusive;
- **e2e-server-sided-validation:** stage accountable for the execution of server-sided tests;
- **e2e-security-validation:** stage tasked with the execution of super master user exclusive tests.

Figure 4.19 illustrates the applied changes to the GitLab pipeline, along with access to Vault secrets. It is important to note that the jobs follow a specific execution order, where the smoke tests are performed before all others, to ensure that all functionalities are operational and more complex testing can be followed.

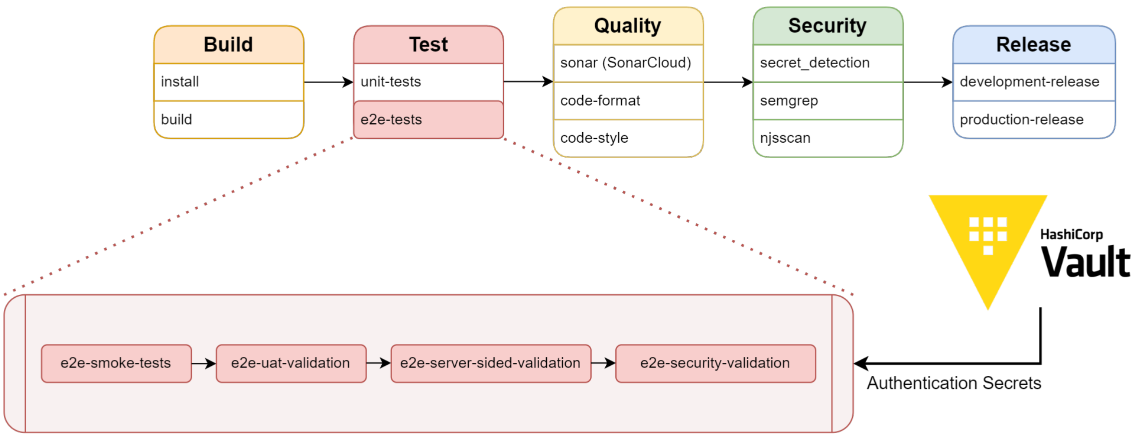


Figure 4.19: Schematic representation of the GitLab CI/CD pipeline of the Stratio-Web repository after performing E2E tests.

4.6.4 E2E Tests Parallelism

Playwright test runner is able to perform parallel testing in order to reduce the time needed to execute test suites. In this sub-section, a brief study regarding the optimal number of workers for the developed test suite will be disclosed. Appendix C.2 contains all the raw data used to conduct this study.

To carry out this research, it is important to combine the parallelism setting of the GitLab CI runner since Playwright allows sharding between multiple jobs through the **parallel** keyword [72]. It should be highlighted that **two properties** will be studied during this research:

- **Playwright Workers:** points out to the Playwright configuration file, where it is possible to define the parallel execution capability **provided by the Playwright test runner**;
- **GitLab Sharding:** refers to a **feature provided by the GitLab CI pipeline** that divides a job into numerous shards to concurrently execute multiple tasks across different resources.

Figure 4.20 represents the execution time of the Playwright test runner with a range of one to three shards and one to four playwright workers:

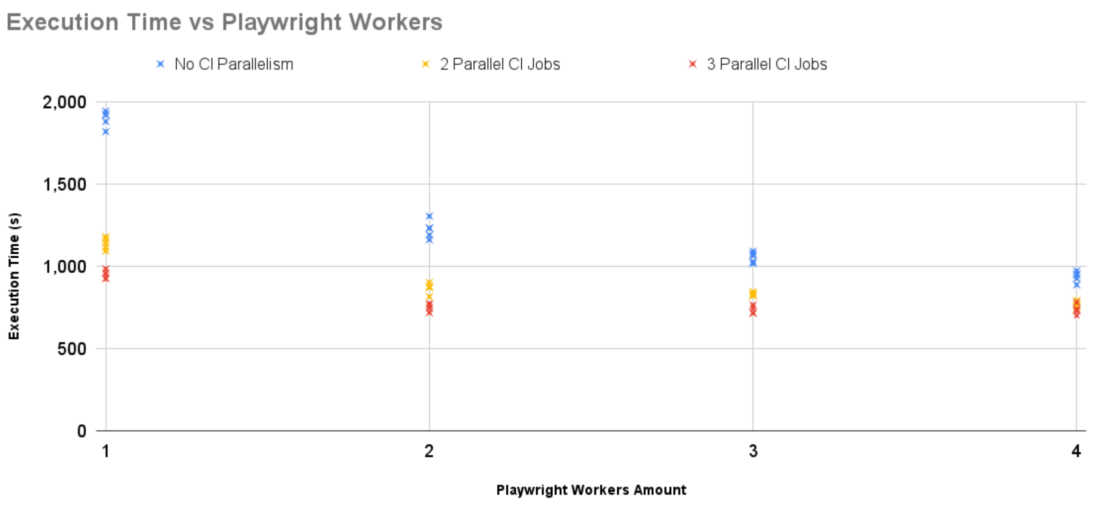


Figure 4.20: Execution time of Playwright Test Runner with different workers/shards.

It is worth mentioning that GitLab sharding significantly enhances performance, particularly when there are fewer Playwright workers involved. Table 4.16 represents different metrics, including the best, worst and average execution times, as well as the standard deviation.

Table 4.16: Playwright Test Runner Parallelism Statistics

GitLab Jobs	Workers	Parallelism Statistics (seconds)			
		Best Execution	Worst Execution	Average Execution Time	Standard Deviation
1	1	1,819.0	1,943.0	1,896.8	49.1
	2	1,162.0	1,305.0	1,224.4	54.1
	3	1,017.0	1,092.0	1,055.8	31.7
	4	888.0	973.0	937.8	32.3
2	1	1,093.0	1,180.0	1,139.6	35.6
	2	816.0	903.0	868.4	31.9
	3	821.0	845.0	832.4	10.9
	4	740.0	794.0	764.2	19.7
3	1	922.0	985.0	948.2	25.6
	2	719.0	777.0	753.6	22.5
	3	716.0	766.0	738.8	22.1
	4	700.0	781.0	730.8	32.7

To ease the analysis of the parallelism statistics, Figure 4.21 displays a chart comparing the average execution time of the Playwright test runner with the number of workers.

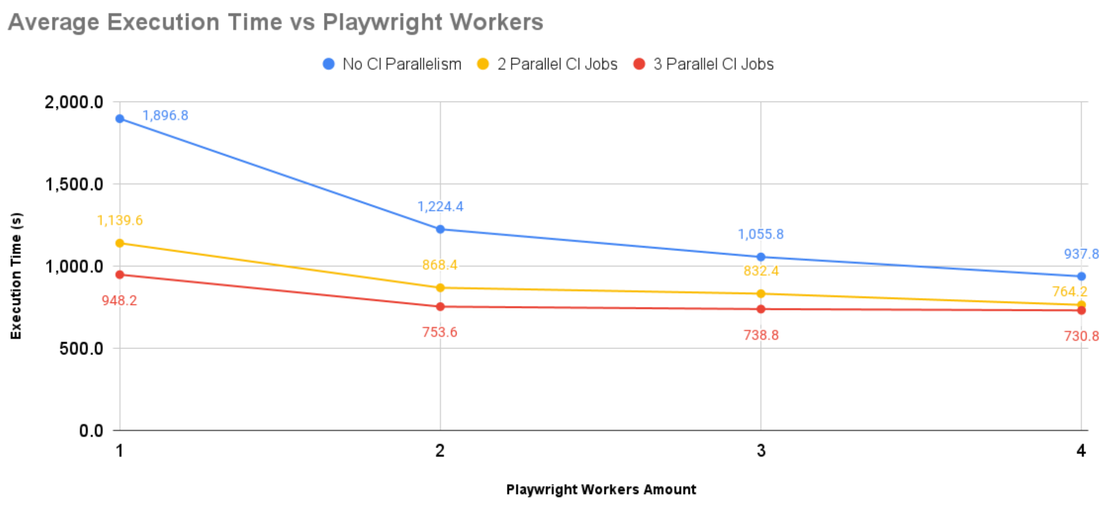


Figure 4.21: Average execution time of Playwright Test Runner with different workers/shards.

Similarly to the component tests pipeline, the CI pipeline used by Stratio Automotive has the GitLab workers within the company's server, denoting the fact that they are hosted locally and E2E testing should not bottleneck the system. Certain aspects to be aware of include:

- If the **number of Playwright workers is prioritized** and we use four, then the amount of GitLab CI parallel jobs does not have a significant impact beyond two;

- If the **number of GitLab CI workers is prioritized** and we use three, then the amount of Playwright workers does not have a significant impact beyond two;
- Some tests may produce **flaky behavior**. This conveys that the platform may not be stable, a nondeterministic error occurred or a timeout may have appeared, which means that there is some discrepancy in testing time. This points to the fact that the **standard deviation** is generally larger than twenty seconds.

After a thorough analysis of the parallelism data regarding E2E tests, it was decided to take advantage of multiple Playwright workers and GitLab CI jobs and use **two of each**.

4.6.5 Influence on the Product Team

As it was previously mentioned, the PO is responsible for manually validating the Stratio Foresight Platform (section 4.2.2), which means they are responsible for individually rectifying all the functionalities of Tables 4.1 and 4.2. The developed test suite would enable them to focus on validating more specific features, introducing **test redundancy**, in order to increase test confidence.

4.7 Continuous Testing

After creating a test suite for the validation of the Stratio Foresight Platform, the intern worked together with the development team during three sprints (fifteen days each) to promote and support them for better testing principles. To evolve the process regarding the approval of new user stories or bug fixes, an adaptation of a TDD approach was employed. Typically, a TDD strategy uses **unit tests** [73], however, a new procedure regarding automated E2E tests will be proposed.

As mentioned in section 2.2.2, the TDD strategy inverts the flow of development, where tests are composed first. Figure 4.22 breaks down the proposed approach in the following key points:

1. The path commences with a **request**, either a user story or a bug fix;
2. A **new E2E test** is developed for that request, which will fail due to no code being developed;
3. The request is **developed** using the minimum amount of code necessary to make the test pass;

4. The code is subsequently **refactored** to improve readability, reduce complexity, and ensure maintainable and efficient code. Then, the full battery of E2E tests must be executed, in order to perform regression testing and ensure no bugs have been introduced with recent changes.

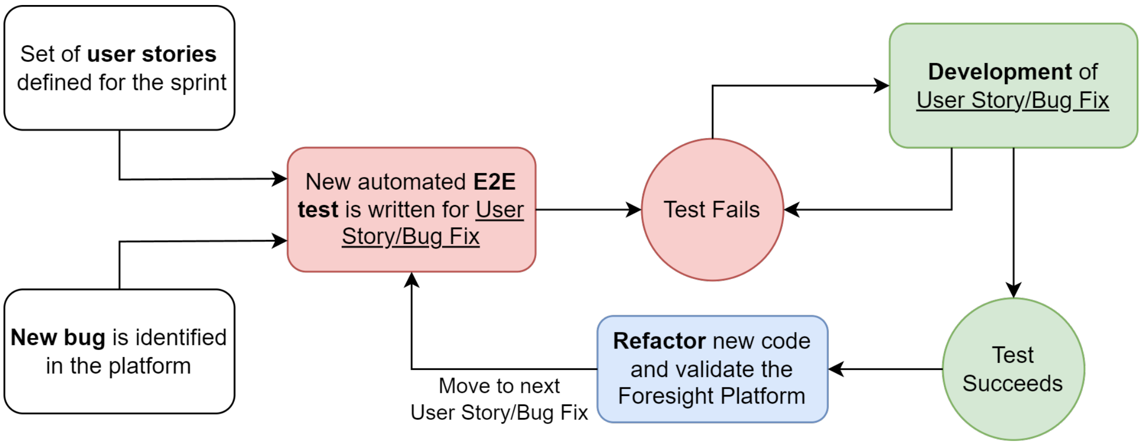

Set of **user stories** defined for the sprint
New automated **E2E test** is written for User Story/Bug Fix
New **bug** is identified in the platform
Test Fails
Development of User Story/Bug Fix
Test Succeeds
Refactor new code and validate the Foresight Platform
Move to next User Story/Bug Fix

Figure 4.22: Schematic representation of the proposed TDD approach with E2E tests.

4.7.1 Integration in the CI/CD Pipeline

To incorporate the proposed TDD approach within the CI/CD pipeline, we focused on a workflow that could represent the routine presented in Figure 4.22. In practical terms, both the Quality Assurance (QA) and development team must coordinate efforts to take advantage of automated testing.

Figure 4.23 represents how the continuous integration (CI) flow was adjusted in order to validate bug fixes and user stories.

The sequence of steps is as follows:

1. The QA and development team create a new branch for the proposed changes;
2. The QA team creates the appropriate test cases and the development team plans their changes;
3. The QA team develops E2E tests for the bug fix or user story and merges the automated tests into the main branch;
4. Before finishing implementation, the development team **rebases** their branch with the newly created automated tests and try to deploy changes:
   (a) If the automated test **fails**, then the bug fix or user story is not ready and **refactor** is needed;
   (b) If the test **succeeds**, the changes are **validated** and the **deploy is merged**.

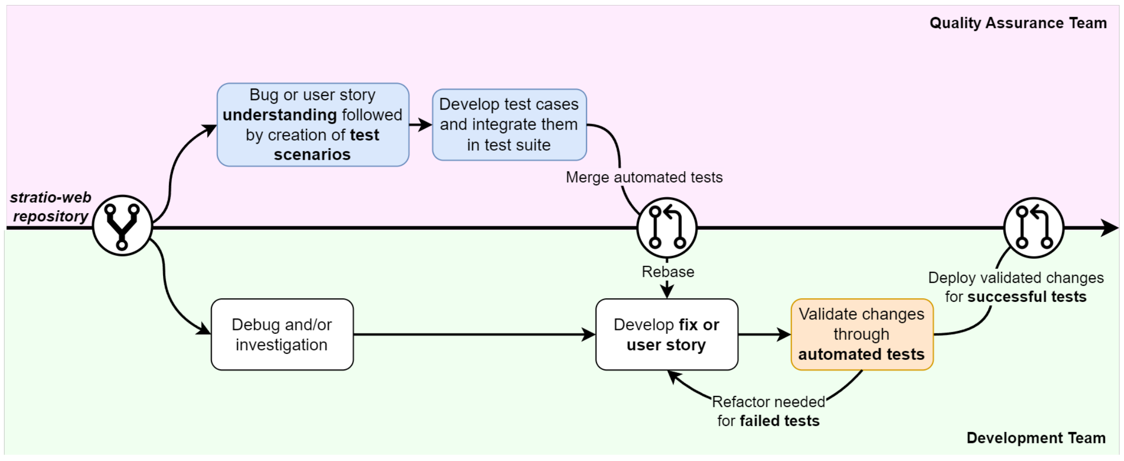

Quality Assurance Team
stratio-web repository
Bug or user story understanding followed by creation of test scenarios
Develop test cases and integrate them in test suite
Merge automated tests
Rebase
Debug and/or investigation
Develop fix or user story
Validate changes through automated tests
Refactor needed for failed tests
Deploy validated changes for successful tests
Development Team

Figure 4.23: Schematic representation of the pipeline operating with the proposed TDD approach.

This approach was practiced between the intern and the development team with examples for **bug fixing** and **user stories**. Figure 4.24 depicts the pipeline **without** the proposed TDD strategy.

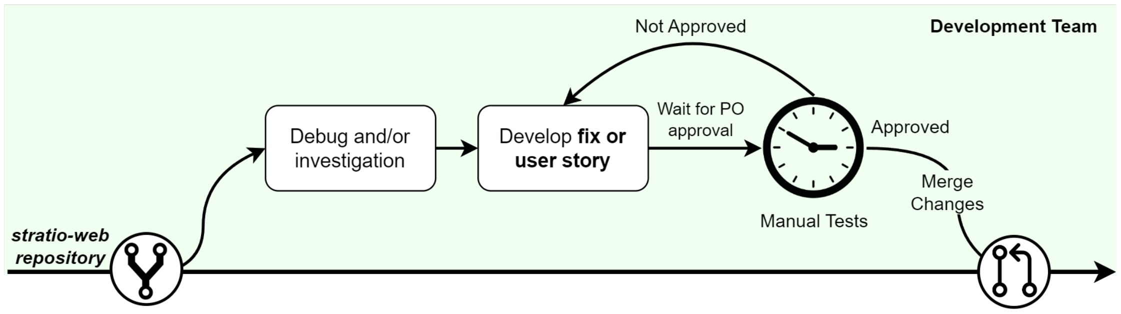


Figure 4.24: Schematic representation of the pipeline operating without TDD.

The main objective was to clear the **manual validation** performed by the PO and replace that interaction with tailored E2E tests for that specific bug fix or user story. To enforce this practice, the intern engaged in a **weekly technical planning** meeting to identify features or bugs that would benefit from the TDD workflow.

4.7.2 Coordination with Development and Product Teams

To assess the proposed TDD approach, the intern presented the strategy to the development and product teams in order to measure its benefits. Table 4.17 represents a total of two bug fixes and two user stories that were tested with this approach. Before merging the changes, an E2E test was used to validate the necessity of further refactoring. A total of five executions for task with two GitLab jobs and Playwright workers were executed:

Table 4.17: Summary of approval time for the proposed TDD approach.

Type	Tasks	Average Execution Time (seconds)
Bug Fixes	**Vehicle Details:** Disabling a device is duplicating Operation metrics data	118.8
	Maps: Operational Events - Idle Consumption with wrong units	98.6
User Stories	**Ecodrive:** Bus Lines - Apply filters across details	113.4
	Vehicle Details: Navigate to Maps through selected vehicle	101.2

A few considerations follow:

- The approval time of a bug fix or user story depends on its complexity, such as the amount of pages navigated or the extent of the feature. Nevertheless, the time required for both types of approvals should generally be within the **range of two to three minutes**;
- Since the team is currently working on component revamps, it was difficult to find adequate bug fixes and user stories to validate. The ones presented in the previous table correspond to already deployed features and prove that the proposed TDD approach can be used to validate bug fixes and user stories. Due to that limitation, it was impossible to analyse the number of prevented defects and required refactors.

4.8 Overall Impact of the Proposed Solution

In this section, various metrics will be presented to quantify the impact of the proposed solution and better understand if the modifications to the quality process and the addition of a new battery of tests were significant. In this case, the most relevant to tackle are the **defects identified**, **test coverage**, **team feedback** and the **approval time of new Web UI versions of the product**.

4.8.1 Before Automated Testing

To correctly measure the impact of the proposed solution, the intern accompanied the product team during two development sprints (fifteen days each) in order to retrieve approval metrics of the time spent evaluating user stories, bug fixes and the Stratio Foresight Platform through **manual** web UI tests. In this context, "user stories" refer to newly developed features by the software team, while the latter corresponds to the manual validation of the whole product before deployment. Table 4.18 presents a summary of the metrics collected for different types of approvals. The second column, labeled "Average Bugs Found", indicates the average number of bugs discovered per approval type. For example, for every five user story approvals, one bug is identified and the **task must be refactored and analysed by the PO again**. The full study data can be found in Appendix C.3.

Table 4.18: Approval time metrics before performing automated testing.

Approval Type	Average Bugs Found	Average Validation Time (min)
User Story	0.2	11.2
Bug Fix	0.4	18.9
Full Product	1.0	132.5

Key components to emphasize include:

- The "Validation Time" property only counts towards the time the PO opened the user story, bug fix or product and performed the validation process. It does not include the time spent updating tickets or writing status messages to the development teams;

- On average, it is **faster to approve user stories than bug fixes** due to the latter requiring thorough validation of the affected feature and its related functionalities. The PO ends up performing regression testing (section 2.3.3) **manually**;

- Usually, the present **bugs** in the Stratio Foresight Platform are **non-deterministic** (referred to as **mandelbugs**, as mentioned in section 2.3.2) which can give the developers the false impression of being wrongfully fixed. As a result, these bugs tend to reappear more frequently in bug fix approvals, causing them to have a superior count regarding average bugs found when compared to user story approvals;

- The full product validation is a convoluted process where the **PO takes more than two hours to validate the Stratio Foresight Platform**, finding one bug on average.

4.8.2 Defects Prevented or Identified

Monitoring the number of bugs found **during testing** is a challenging process since the product itself **only had unit tests** that were thoroughly settled. However, it is still possible to conclude a testing report regarding the bugs found with the new testing techniques.

Component Testing

Since component testing occurs alongside development, it plays a vital role in **preventing** bugs instead of identifying them. An analysis of Stratio Automotive's GitLab pipeline identifies the following situations:

- During a three month period, there were **seven instances** where the component-tests job failed in the pipeline (in a total of forty five executions). However, after a refactoring process, the pipeline was successfully executed, indicating **preemptive** defect mitigation;
- The developed component tests **prevented defects in four components** (Drawer, Alert-Modal, Label and Lists-Resume) across five merge requests.

E2E Testing

The battery of E2E tests proved to be an adequate tool to identify defects in the Stratio Foresight Platform. Table 4.19 represents a summary of the identified bugs in the Stratio Foresight Platform within a two month period.

Table 4.19: Summary of identified defects through E2E tests.

Stratio Foresight Platform		
Functionality	**Feature**	**Defects**
Fleet Condition	Consult Vehicle Details	- The Operation Metrics card was wrongfully duplicated upon vehicle location deactivation.
Service Plans	Cleanup of Service Plan	- After clicking the 'X' button to delete a service plan, a pop-up to confirm appears. However, the 'cancel' text was not rendering in the correct button.
Ecodrive	Driver Score	- "Export Report" button did not generate a report.
My Settings	General Settings	- After switching the language from 'English' a few translation errors were identified.
	Measurement Units	- After switching the pressure unit from 'bar' to 'psi', the tables from the 'Air Leaks' feature in the 'System Indicator' functionality did not switch from 'bar' to 'psi'.

4.8.3 Code Coverage

This metric measures the percentage of code that is executed by tests. Since the only existing form of testing before this internship was unit testing, **we had zero coverage regarding component testing**.

As explained in section 4.4.3, the Storybook Test-Runner can use the generated testing data to create reports of the percentage of lines covered by automated tests. Table 4.8 from section 4.4.2 presents a summary of the code coverage gained from component testing with an **achieved coverage of 83.69%** throughout all components, **exceeding the target coverage for components of 75% of the success criteria**.

4.8.4 Test Coverage

The test coverage metric specifies the range to which the functionality and requirements of a system are covered by tests. Tables 4.14 and 4.15 from section 4.6 present which functionalities and features were covered. It is important to note that every functionality was addressed with the exception of the Dashboard functionality which will receive an overhaul soon and it is not justified to develop an E2E test. Out of fifty different features, a total of forty nine (98%) were covered through E2E testing strategies, **surpassing the target coverage of 90% of the Stratio Foresight Platform features from the success criteria**.

4.8.5 Contribution to the Development and Product Teams

Given that the quality process within the product and development teams have undergone changes, it is important to understand their point of view regarding the new practices. Humans are typically resistant to change and it is important to ensure that the proposed solution actually improves their workflow and does not hinder their performance. The primary aspects consist of the following:

- One **presentation** was conducted for **each POC**, covering both component and E2E tests, aiming to demonstrate the capabilities of these testing techniques;
- After implementing the battery of **component tests**, the intern supported the development team **refactoring multiple components** as well as how to update the tests;
- A **training session** was provided for the development and product team regarding automated E2E tests and how they are integrated in the CI pipeline;
- Integration of different automated testing tools for component and E2E tests in the GitLab CI pipeline, in order to meet the **success criteria**.

4.8.6 Approval Time of Product Iterations

It is possible to compare the average time that a PO took to manually validate the entire product before approving it for the staging environment and correlate it with the time the full battery of E2E tests developed takes. Since there is no human interaction to execute the composed tests and the full process is automated through the CI/CD pipeline, it is possible to conclude that the product team does not need to spend large amounts of time validating new functionalities, bug fixes or the product itself, since all the processes have been automated.

Since a rough estimate of the average time per evaluation before the automated tests has been calculated in section 4.8.1, it is possible to predict the amount of time spent approving user stories, bug fixes and validating the Stratio Foresight Platform per sprint (fifteen days). To calculate these estimates, the total number of user stories and bugs were collected from the last twenty four sprints, considering that only one analysis from the PO was enough to validate each instance. Table 4.20 contains a comparison between the average time spent per sprint manually approving and with automated Web UI tests:

Table 4.20: Summary of manual approval time metrics against automated Web UI tests.

Approval Type	Average Instances p/ Sprint	Average Time p/ Sprint (min)	
		Manually Approving	With Automated Tests (2 GitLab Jobs and Playwright Workers)
User Story	4	44.8	7.2
Bug Fix	7	132.3	12.5
Full Product	1	132.5	14.5

Time management is a major consideration in each sprint. Implementing automated Web UI tests instead of manual approvals could **save the PO team up to five hours and ten minutes per sprint**, which is a considerate amount of time. It is important to note the following:

- Implementing the proposed **TDD approach**, the PO **would not spend any time** approving either **user stories** nor **bug fixes** since automated E2E tests are being executed within the GitLab pipeline;
- Based on section 4.7.2 and using the average approval times of Table 4.17, we estimate that the approval time is around **six times faster for user stories** and **ten times faster for bug fixes** with automated E2E tests;
- According to section 4.6.4, the automated E2E test battery takes around fourteen minutes (with two GitLab jobs and Playwright workers) to perform a **full product validation** which is around **nine times faster than a manual validation**, **exceeding the success criteria to reduce by half the manual testing and approval time of the Web UI platform**;
- Evidently, it is crucial to understand the amount of time needed to create an automated test. Section 4.6.2 illustrates the file structure and Playwright configuration, organized in a modular way to easily create E2E tests. Most of the time, the locators needed to create a new test are previously defined and, depending on the feature, it takes less than twenty minutes;
- On the occasion of a feature needing to be tested multiple times, it is possible to repeat the same corresponding automated test instead of manually validating the feature multiple times.

Chapter 5

Conclusion

This chapter is divided into two sections: the first denotes an overview of the main topics researched and how the proposed solution impacted the quality assurance process of Stratio Automotive through production level component and E2E tests. The final section includes a small reflection on what could be done in the future to improve this project.

5.1 Dissertation Remarks

The objective of this Dissertation was to optimize the approval time of Web UI tests. To that end, we presented a deep analysis of Stratio Automotive's quality assurance process, including the most frail points regarding Web UI testing. A quick examination pointed out that there were instances where slow manual tests were performed to approve user stories, bug fixes and the Stratio Foresight Platform.

To broaden Stratio Automotive's testing standards, two testing paradigms were introduced from scratch into the company: **component testing** and **E2E testing**, which ensure the quality and reliability of software systems. The **Testing Pyramid** aided in the identification and study of the two topics, emphasizing their differences regarding execution time and granularity.

After identifying weak points where the approval time of Web UI tests was substantial, a complete overhaul was proposed with new techniques to incorporate both component tests and E2E tests, allowing for test automation. These changes completely modified the validation process of the product offered by Stratio Automotive, since it is now possible to increase the quality of testing whilst also ensuring **quicker validation times** and **less manual** testing. To validate the Stratio Foresight Platform, a battery of component and E2E tests was developed; however, to approve bug fixes and user stories, an adaptation of a TDD approach was presented with E2E tests.

In terms of the proposed solution, out of fifty different features, a total of forty nine (98%) were covered through E2E testing strategies, surpassing the target coverage of 90% and the intended coverage of 75% for components has been exceeded with a coverage of 83.69%. On the other hand, the approval time is six times faster for user stories, ten times quicker for bug fixes, and nine times faster to perform a full product validation, exceeding the success criteria to reduce by half the manual testing and approval time of the Web UI platform. Furthermore, both component and E2E tests have been integrated in the GitLab CI pipeline in order to automate Web UI tests. The work carried out during this internship positively impacts the development team since it provides instant feedback during the process of merging new code, effectively preventing and identifying bugs within the repositories of the Stratio Foresight Platform. Regarding the product team, the changes proposed in the quality assurance process reduce the manual approval time considerably.

Significant experience was acquired during this internship. There was an opportunity to work with various teams simultaneously (quality assurance, product and development teams) in an Agile mindset while understanding various levels of testing and their importance to developing a quality product. Since component tests consist of a white box testing technique, the intern had the opportunity to study and understand Angular components in order to develop quality tests in TypeScript. Regarding E2E tests, it was not trivial to create test cases since it was necessary to examine the software functionalities of the Stratio Foresight Platform. In relation to both testing paradigms, a deep understanding of browser automation was crucial, as was test integration within components and the system as a whole. Nevertheless, some technical knowledge of DevOps was also necessary to implement test automation within the GitLab CI pipelines.

5.2 Future Work

Even though a new solution was proposed and the positive impact of its efficiency has been confirmed, it is still crucial to move towards a hybrid between the proposed TDD and a BDD approach. This fusion has the main objective of increasing collaboration with the quality assurance, development and product teams in order to define testable user stories from an end-user perspective. Furthermore, by dismissing manual testing in favor of automation, it would be possible to enable a continuous deployment scenario where small increments could be deployed instantly, while being thoroughly tested and exempt from manual testing.

These changes in the quality process can take up to a year to fully implement in Stratio Automotive, since it is required to change the habits of various workers and how they operate on a daily basis. Some considerations on how to speed up the integration process follow:

- Prepare a lecture to all the involved teams, stating the weak points of the quality process and what are the solutions to tackle areas that need improvement;
- Present attainable goals and objectives, like test and code coverage or tar-

get approval times, to be met through the proposed changes in the quality process;

- Explain the benefits of test automation and how they can improve Stratio Automotive's product;
- Implement all the proposed changes and retrieve metrics to prove the effectiveness of the new strategy.

Another interesting remark reflects the creation of a **mock environment** without the backend logic. If a local server with these characteristics is created, the developer will have full control over the responses each component gives (through mock requests), granting the tester full liberty when testing the software system.

Finally, due to the importance and positive impact of the proposed tests, the next phase involves expanding the proposed testing techniques to all platforms of Stratio Automotive.

References

[1] The Editors of Encyclopaedia Britannica. "software", 15 May, 2023. https://www.britannica.com/technology/software, last accessed 19 June 2023.

[2] Cypress. Test. automate. accelerate. https://www.cypress.io/, last accessed 15 January 2023.

[3] Playwright. Playwright enables reliable end-to-end testing for modern web apps. https://playwright.dev/, last accessed 15 January 2023.

[4] Storybook. Build uis without the grunt work. https://storybook.js.org/, last accessed 15 January 2023.

[5] Stratio Automotive. The world's #1 predictive fleet maintenance platform. https://stratioautomotive.com/, last accessed 15 January 2023.

[6] kanbanize. What is kanban? explained for beginners. https://kanbanize.com/kanban-resources/getting-started/what-is-kanban, last accessed 10 January 2023.

[7] GitLab. Ci/cd concepts. https://docs.gitlab.com/ee/ci/introduction/, last accessed 7 December 2022.

[8] IBM. What is risk management? https://www.ibm.com/topics/risk-management, last accessed 10 January 2023.

[9] Shylesh S. A study of software development life cycle process models, June 10, 2017. https://ssrn.com/abstract=2988291, last accessed 15 June 2023.

[10] Kate Brush. Definition - test case, March, 2020. https://www.techtarget.com/searchsoftwarequality/definition/test-case, last accessed 15 January 2023.

[11] Claire Drumond. Is the agile manifesto still a thing? https://www.atlassian.com/agile/manifesto, last accessed 15 January 2023.

[12] Ian Sommerville. *Software Engineering*. Pearson, 9th edition, 2010.

[13] Heba Elshandidy, Sherif Mazen, Ehab Hassanein, and Eman Nasr. Using behaviour-driven requirements engineering for establishing and managing agile product lines. *International Journal of Advanced Computer Science and Applications*, 12(2), 2021.

[14] Thomas Hamilton. Static vs dynamic testing: Difference between them, June 2023. `https://www.guru99.com/static-dynamic-testing.html`, last accessed 17 July 2022.

[15] Henrique Madeira. Software quality and dependability, software testing overview [powerpoint presentation], February 2022.

[16] Mohd Ehmer Khan and Farmeena Khan. A comparative study of white box, black box and grey box testing techniques. *International Journal of Advanced Computer Science and Applications*, 3(6), 2012. last accessed 29 December 2022.

[17] Thomas Hamilton. White box testing – what is, techniques, example & types, November 19, 2022. `https://www.guru99.com/white-box-testing.html`, last accessed 29 December 2022.

[18] Mohd Khan. Different approaches to black box testing technique for finding errors. *International Journal of Software Engineering & Applications*, 2, 10 2011. last accessed 30 December 2022.

[19] Adam Murray. Gray box testing guide, February 4, 2021. `https://www.mend.io/resources/blog/gray-box-testing/`, last accessed 30 December 2022.

[20] Shreya Bose. How to jumpstart a test automation pyramid?, October 28, 2022. `https://www.browserstack.com/guide/testing-pyramid-for-test-automation`, last accessed 15 January 2023.

[21] Angular. The web development framework for building the future. `https://angular.io/`, last accessed 15 January 2023.

[22] Thomas Hamilton. What is component testing? techniques, example test cases, November 2, 2022. `https://www.guru99.com/component-testing.html`, last accessed 3 January 2023.

[23] Angular. Basics of testing components, February 28, 2022. `https://angular.io/guide/testing-components-basics`, last accessed 3 January 2023.

[24] Componentdriven. Component driven user interfaces. `https://www.componentdriven.org/`, last accessed 3 January 2023.

[25] Thomas Hamilton. Test plan vs test strategy – difference between them, December 31, 2022. `https://www.guru99.com/test-plan-v-s-test-strategy.html`, last accessed 3 January 2023.

[26] Thomas Hamilton. What is end-to-end testing? e2e example, October 2022. `https://www.guru99.com/end-to-end-testing.html`, last accessed 8 December 2022.

[27] Shreya Bose. What is end to end testing?, October 31, 2022. `https://www.browserstack.com/guide/end-to-end-testing`, last accessed 8 December 2022.

[28] Krishna Rungta. Page object model (pom) & page factory in selenium, October 2022. `https://www.guru99.com/page-object-model-pom-page-factory-in-selenium-ultimate-guide.html`, last accessed 7 December 2022.

[29] Alexander S. Gillis. Definition - smoke testing, November, 2022. `https://www.techtarget.com/searchsoftwarequality/definition/smoke-testing`, last accessed 20 February 2023.

[30] Cypress. Having tests rely on the state of previous tests, June, 2023. `https://docs.cypress.io/guides/references/best-practices#Having-Tests-Rely-On-The-State-Of-Previous-Tests`, last accessed 29 June 2023.

[31] Cypress. Creating "tiny" tests with a single assertion, June, 2023. `https://docs.cypress.io/guides/references/best-practices#Creating-Tiny-Tests-With-A-Single-Assertion`, last accessed 29 June 2023.

[32] Playwright. Auto-waiting. `https://playwright.dev/docs/actionability`, last accessed 11 January 2023.

[33] Cypress. Best practices - selecting elements. `https://docs.cypress.io/guides/references/best-practices#Selecting-Elements`, last accessed 7 December 2022.

[34] Thomas Hamilton. What is parallel testing? definition, approach, example, November 5, 2022. `https://www.guru99.com/parallel-testing.html`, last accessed 9 November 2022.

[35] Cypress. Cypress component testing. `https://docs.cypress.io/guides/component-testing/overview`, last accessed 3 January 2023.

[36] Playwright. Experimental: components. `https://playwright.dev/docs/test-components`, last accessed 3 January 2023.

[37] jscutlery. Playwright component testing for angular (experimental). `https://github.com/jscutlery/devkit/tree/main/packages/playwright-ct-angular`, last accessed 3 January 2023.

[38] Jasmine. Jasmine behavior-driven javascript. `https://jasmine.github.io/`, last accessed 3 January 2023.

[39] Karma. Karma - spectacular test runner for javascript. `https://karma-runner.github.io/latest/index.html`, last accessed 3 January 2023.

[40] Angular. Testbed. `https://angular.io/api/core/testing/TestBed`, last accessed 3 January 2023.

[41] Joel Jeske. karma-parallel. `https://github.com/joeljeske/karma-parallel`, last accessed 12 January 2023.

[42] Storybook. Introduction to storybook for angular. https://storybook.js.org/docs/angular/get-started/introduction, last accessed 3 January 2023.

[43] Dominic Nguyen. Introduction to storybook for angular, June 24, 2020. https://www.chromatic.com/blog/storybook-composition/, last accessed 3 January 2023.

[44] Ben Anas. Quick overview: Storybook with react, June 4, 2021. https://medium.com/edonec/quick-overview-storybook-with-react-439e1ccce5a7, last accessed 3 January 2023.

[45] Maria Homann. Continuous testing in ci/cd: What, why and how. https://www.leapwork.com/blog/ci-cd-continuous-testing-what-why-how#what-is-continuous-testing, last accessed 9 November 2022.

[46] Eric Avidon. Definition flaky test, May 2019. https://www.techtarget.com/whatis/definition/flaky-test, last accessed 9 November 2022.

[47] Cypress. Writing your first e2e test. https://docs.cypress.io/guides/end-to-end-testing/writing-your-first-end-to-end-test, last accessed 3 January 2023.

[48] Cypress. Trade-offs. https://docs.cypress.io/guides/references/trade-offs, last accessed 3 January 2023.

[49] Selenium. Webdriver, December 7, 2021. https://www.selenium.dev/documentation/webdriver/, last accessed 3 January 2023.

[50] Selenium. Selenium ide. https://www.selenium.dev/selenium-ide/, last accessed 3 January 2023.

[51] W3C. Webdriver - w3c recommendation, June 5, 2018. https://www.w3.org/TR/webdriver1/, last accessed 15 January 2023.

[52] Playwright. Installation. https://playwright.dev/docs/intro, last accessed 3 January 2023.

[53] TestCafe. Getting started. https://testcafe.io/documentation/402635/getting-started, last accessed 3 January 2023.

[54] Puppeteer. Getting started. https://pptr.dev/, last accessed 3 January 2023.

[55] WebdriverIO. Getting started. https://webdriver.io/docs/gettingstarted, last accessed 3 January 2023.

[56] Katalon. Katalon studio. https://katalon.com/katalon-studio, last accessed 3 January 2023.

[57] CodeceptJS. Codeceptjs - getting started, July 2022. https://codecept.io/basics/, last accessed 9 November 2022.

[58] selectorshub.com. Selectorshub, November 2022. https://chrome.google.com/webstore/detail/selectorshub/ndgimibanhlabgdgjcpbbndiehljcpfh, last accessed 9 November 2022.

[59] selectorshub.com. Testcase studio, November 2022. https://chrome.google.com/webstore/detail/testcase-studio/loopjjegnlccnhgfehekecpanpmielcj, last accessed 9 November 2022.

[60] Paulo Homem. Plataforma de gestão remota da stratio duplica o número de utilizadores desde o início da crise, May, 2020. https://posvenda.pt/plataforma-de-gestao-remota-da-stratio-duplica-o-numero-de-utilizadores-desd
last accessed 27 April 2023.

[61] Sonar. As a service. sonarcloud., May, 2023. https://www.sonarsource.com/products/sonarcloud/, last accessed 4 May 2023.

[62] Storybook. Interactions, March, 2023. https://storybook.js.org/addons/@storybook/addon-interactions, last accessed 23 March 2023.

[63] Storybook. Test runner, March, 2023. https://storybook.js.org/addons/@storybook/test-runner, last accessed 23 March 2023.

[64] Nrwl. @nrwl/storybook, March, 2023. https://nx.dev/packages/storybook, last accessed 23 March 2023.

[65] Storybook. Story coverage, February, 2023. https://storybook.js.org/addons/@storybook/addon-coverage, last accessed 23 March 2023.

[66] Yann Braga. Storybook coverage recipes - angular, January, 2023. https://github.com/yannbf/storybook-coverage-recipes/tree/main/angular, last accessed 23 March 2023.

[67] Linux Test Project. Lcov (version 1.14), October 10, 2022. https://github.com/linux-test-project/lcov, last accessed 4 May 2023.

[68] Storybook. Accessibility, March, 2023. https://storybook.js.org/addons/@storybook/addon-a11y, last accessed 23 March 2023.

[69] Playwright. Installation, 2023. https://playwright.dev/docs/intro#installing-playwright, last accessed 18 May 2023.

[70] Playwright. Test configuration, 2023. https://playwright.dev/docs/test-configuration, last accessed 18 May 2023.

[71] HashiCorp. Manage secrets & protect sensitive data with vault, 2023. https://www.vaultproject.io/, last accessed 17 May 2023.

[72] Playwright. Gitlab ci - sharding, 2023. https://playwright.dev/docs/ci#sharding-1, last accessed 17 May 2023.

[73] Grant Steinfeld. 5 steps of test-driven development, February, 2020. https://developer.ibm.com/articles/5-steps-of-test-driven-development/, last accessed 25 March 2023.

Appendices

Appendix A

Information for Component and E2E Testing POC

In Appendix A, the detailed test cases applied in the POC for the component and E2E automated tests are described, as are the experimental conditions.

A.1 Component Tests

Details regarding the experimental conditions and test cases of the component testing POC.

Experimental Conditions

During the component testing POC, the following technologies were used:

Software

- Cypress 11.2.0;
- Storybook 6.5.9;
- Windows 11 Pro 21H2.

Hardware

- CPU: Intel Core i7-8550;
- RAM Memory: SDRAM DDR4-2400 32 GB (16 GB x 2);
- Storage: SSD 512 GB PCIe Gen 3x4 NVMe TLC.

Test Cases

Table A.1: Test case POC-CT-1 - Assert Alert-Modal Component.

ID	POC-CT-1
Description	The test runner must assert a range of properties from the Alert-Modal Component.
Prerequisites	N/A.
Steps	1. Mount the Alert-Modal component; 2. Assert 'Confirm' button text and style; 3. Assert 'Cancel' button text and style; 4. Click the 'Confirm' button; 5. Click on the 'Cancel' button.
Expected Results	All assertions are correctly validated.

Table A.2: Test case POC-CT-2 - Assert Sub-Menu Component.

ID	POC-CT-2
Description	The test runner must assert a range of properties from the Sub-Menu Component.
Prerequisites	N/A.
Steps	1. Mount the Sub-Menu component; 2. Click the 'Filter' button; 3. Assert 'Clear' button text and style; 4. Assert 'Filter' title text and style; 5. Click on the 'Menu' button.
Expected Results	All assertions are correctly validated.

Table A.3: Test case POC-CT-3 - Assert Sub-Menu Component.

ID	POC-CT-3
Description	The test runner must assert a range of properties from the Table Component.
Prerequisites	N/A.
Steps	1. Mount the Table component; 2. Assert Table Component has 3 columns: id, title and ct; 3. Click 'Columns' button; 4. Disable 'title' column by clicking in the respective checkbox; 5. Assert Table Component has 2 columns: id and ct.
Expected Results	All assertions are correctly validated.

A.2 E2E Tests

Details regarding the experimental conditions and test cases of the E2E testing POC.

Experimental Conditions

During the E2E POC, the following technologies were used:

Software

- Cypress 10.7.0;
- Playwright 1.25.1;
- Puppeteer 17.1.1;
- Windows 11 Pro 21H2.

Hardware

- CPU: Intel Core i7-8550;
- RAM Memory: SDRAM DDR4-2400 32 GB (16 GB x 2);
- Storage: SSD 512 GB PCIe Gen 3x4 NVMe TLC.

Test Cases

Table A.4: Test case 1 - Mark Occurence as Read.

ID	1
Description	The user is able to consult the occurrences and mark the ones pretended as read.
Prerequisites	User is logged in.
Steps	1. Click on the 'Occurrences' tab; 2. Click the checkbox of the pretended occurrence; 3. Click on the 'Mark as Read' button.
Expected Results	Occurrence is no longer marked as bold (if applicable) and it is presented a new pop-up stating the number of occurrences marked as read.

Table A.5: Test case 2 - Mark Service Plan as Done.

ID	2
Description	The user is able to consult the service plans and mark the ones pretended as done.
Prerequisites	User is logged in.
Steps	1. Click on the 'Service Plan' tab; 2. Click on 'Service Plans' option; 3. Click on 'Create' button; 4. Complete the form with relevant information; 5. Click on 'Create' button; 6. Click on the 'Service Plan' tab. 7. Click on the created service; 8. Click on the 'Mark Service as Done' button; 9. Repeat step 8.
Expected Results	Service plan is marked as done and it is presented a new pop-up stating that the service occurrence was registered.

Appendix B

Information Regarding Component Tests

Appendix B includes information regarding component tests, including all the test cases, coverage information and parallelism data for Storybook Test Runner.

B.1 Test Cases for Component Testing

The documentation of the test cases for the component tests performed in the Front-end Commons Repository are as follows:

Alert Modal

Table B.1: Test case CT-1.0 - Alert-Modal Component.

ID	CT-1.0
Description	The test runner must be able to correctly render the component.
Prerequisites	N/A
Test Data	{ icon: 'stra-icon-exclamation-triangle', title: 'Title', visible: true, body: 'Description example', isLoading: false, cancelText: '', confirmText: '', confirmingText: '', isConfirmEnabled: true, type: ColorScheme.danger }
Steps	1. Mount the Alert-Modal component;
Expected Results	The component is mounted without errors.

Table B.2: Test case CT-1.1 - Alert-Modal Component.

ID	CT-1.1
Description	The test runner must be able to click the ′Confirm′ and ′Cancel′ button.
Prerequisites	N/A
Test Data	Same as Test Case B.1 (CT-1.0) with the following differences: { cancelText: ′Cancel′, confirmText: ′Confirm′ }
Steps	1. Mount the Alert-Modal component; 2. Assert ′Confirm′ button text and style; 3. Assert ′Cancel′ button text and style; 4. Click the ′Confirm′ button; 5. Click on the ′Cancel′ button.
Expected Results	All assertions are correctly validated.

Table B.3: Test case CT-1.2 - Alert-Modal Component.

ID	CT-1.2
Description	The test runner must be able to click the ′Confirming′ and ′Cancel′ button.
Prerequisites	N/A
Test Data	Same as Test Case B.1 (CT-1.0) with the following differences: { cancelText: ′Cancel′, confirmText: ′Confirming′, isLoading: true }
Steps	1. Mount the Alert-Modal component; 2. Assert ′Confirming′ button text and style; 3. Assert ′Cancel′ button text and style; 4. Assert ′Confirming′ button is disabled; 5. Click on the ′Cancel′ button.
Expected Results	All assertions are correctly validated.

Table B.4: Test case CT-1.3 - Alert-Modal Component.

ID	CT-1.3
Description	The test runner must be able to assert the ’Confirm’ button is not visible.
Prerequisites	N/A
Test Data	Same as Test Case B.1 (CT-1.0) with the following differences: { confirmText: ’Confirm’, cancelText: ’Cancel’, visible: true, isLoading: true, isConfirmEnabled: false, type: ColorScheme.default }
Steps	1. Mount the Alert-Modal component; 2. Assert ’Cancel’ button text and style; 3. Assert ’Confirm’ button is invisible.
Expected Results	All assertions are correctly validated.

Sub-Menu

Table B.5: Test case CT-2.0 - Sub-Menu Component.

ID	CT-2.0
Description	The test runner must be able to correctly render the component.
Prerequisites	N/A
Test Data	{ appTitle: ”, featureTitle: ”, isLoading: false, isFilterEnable: true, isFilterActive: false, items: undefined, isFiltering: false }
Steps	1. Mount the Sub-Menu component;
Expected Results	The component is mounted without errors.

Table B.6: Test case CT-2.1 - Sub-Menu Component.

ID	CT-2.1
Description	The test runner must be able to navigate to the filter menu.
Prerequisites	N/A
Test Data	Same as Test Case B.5 (CT-2.0) with the following differences: { appTitle: ’CT: Filter’, featureTitle: ’Component Test Filter’, isFilterActive: true, }
Steps	1. Mount the Sub-Menu component; 2. Click the ‘Filter’ button; 3. Assert ‘Clear’ button text and style; 4. Assert ‘Filter’ title text and style; 5. Click on the ‘Menu’ button.
Expected Results	All assertions are correctly validated while navigating the sub-menu component.

Table B.7: Test case CT-2.2 - Sub-Menu Component.

ID	CT-2.2
Description	The test runner must be able to render Sub-Menu in filtering mode.
Prerequisites	N/A
Test Data	Same as Test Case B.5 (CT-2.0) with the following differences: { appTitle: 'CT: Filtering', featureTitle: 'Component Test Filtering', isFilterActive: true, isLoading: true, isFiltering: true, items: null }
Steps	1. Mount the Sub-Menu component in filtering mode.
Expected Results	The component is mounted without errors.

Table B.8: Test case CT-2.3 - Sub-Menu Component.

ID	CT-2.3
Description	The test runner must be able to render Sub-Menu with Filters disabled.
Prerequisites	N/A
Test Data	Same as Test Case B.5 (CT-2.0) with the following differences: { appTitle: 'CT: No Filter', featureTitle: 'Component Test No Filter', isFilterEnable: false, items: null, originalIsCollapsed: true }
Steps	1. Mount the Sub-Menu component with Filters disabled.
Expected Results	The component is mounted without errors.

Table

Table B.9: Test case CT-3.0 - Table Component.

ID	CT-3.0
Description	The test runner must be able to correctly render the component.
Prerequisites	N/A
Test Data	Default properties with custom 'tableHeaders'
Steps	1. Mount the Table component;
Expected Results	The component is mounted without errors.

Table B.10: Test case CT-3.1 - Table Component.

ID	CT-3.1
Description	The test runner must be able to disable a column and assert it.
Prerequisites	N/A
Test Data	Same as Test Case B.9 (CT-3.0)
Steps	1. Mount the Table component; 2. Assert Table Component has 3 columns: 'id', 'title' and 'ct'; 3. Click 'Columns' button; 4. Disable 'title' column by clicking in the respective checkbox; 5. Assert Table Component has 2 columns: id and ct.
Expected Results	All assertions are correctly validated with one column disabled.

Table B.11: Test case CT-3.2 - Table Component.

ID	CT-3.2
Description	The test runner must be able to interact with all buttons.
Prerequisites	N/A
Test Data	Same as Test Case B.9 (CT-3.0)
Steps	1. Mount the Table component; 2. Assert style of columns selector button, ′refresh′ button and ′download′ button; 3. Clicks columns selector button, ′refresh′ button; 4. Navigates between pages and asserts rows changes; 5. Clicks ′download′ button and asserts options.
Expected Results	All assertions are correctly validated.

Drawer

Table B.12: Test case CT-4.0 - Drawer Component.

ID	CT-4.0
Description	The test runner must be able to correctly render the component.
Prerequisites	N/A
Test Data	{ isVisible: false, isClosable: false, displayNavigation: false, enablePreviousNavigation: false, enableNextNavigation: false, footerTemplate: null, drawerOffset: 0, title: 'Drawer' }
Steps	1. Mount the Drawer component;
Expected Results	The component is mounted without errors.

Table B.13: Test case CT-4.1 - Drawer Component.

ID	CT-4.1
Description	The test runner must be able to interact with all buttons.
Prerequisites	N/A
Test Data	Same as Test Case B.12 (CT-4.0) with the following differences: { isVisible: true, isClosable: true, displayNavigation: true, enablePreviousNavigation: true, enableNextNavigation: true, title: 'Visible' }
Steps	1. Mount the Drawer component; 2. Assert close, previous and post buttons style; 3. Clicks close, previous and post buttons; 4. Awaits animation to finish and asserts changes.
Expected Results	All assertions are correctly validated as the buttons present different styles depending on the interaction.

Layout

Table B.14: Test case CT-5.0 - Layout Component.

ID	CT-5.0
Description	The test runner must be able to correctly render the component.
Prerequisites	N/A
Test Data	{ sidebarItems: [{ url: 'https://stratioautomotive.com/', iconCssClass: 'stra-icon-alarm', iconLabel: 'icon label', name: stratioAutomotive, isActive: true}, { url: 'https://careers.stratioautomotive.com/', iconCssClass: 'stra-icon-driver', iconLabel: 'icon label', name: stratioCareers, isActive: false}] }
Steps	1. Mount the Layout component;
Expected Results	The component is mounted without errors.

Table B.15: Test case CT-5.1 - Layout Component.

ID	CT-5.1
Description	The test runner must be able to click the 'Logout' button.
Prerequisites	N/A
Test Data	Same as Test Case B.14 (CT-5.0).
Steps	1. Mount the Layout component; 2. Assert 'Logout' button text and style; 3. Click on the 'Logout' button.
Expected Results	All assertions are correctly validated as the 'Logout' button presents different styles depending on the interaction.

Quick Search

Table B.16: Test case CT-6.0 - Quick Search Component.

ID	CT-6.0
Description	The test runner must be able to correctly render the component.
Prerequisites	N/A
Test Data	{ search: undefined, label: undefined, placeHolder: undefined, isTableSideConfig: false, isFullWidth: false, withLabel: true, distinctValues: true }
Steps	1. Mount the Quick Search component;
Expected Results	The component is mounted without errors.

Table B.17: Test case CT-6.1 - Quick Search Component.

ID	CT-6.1
Description	The test runner must be able to search in the text box.
Prerequisites	N/A
Test Data	Same as Test Case B.16 (CT-6.0) with the following differences: { label: searchText, placeHolder: 'Write here!' }
Steps	1. Mount the Alert-Modal component; 2. Mount Quick Search component; 3. Assert label with 'Search Test' value; 4. Assert style of input text box; 5. Type 'CT: search!' in input and assert text; 6. Click button to clear text from input text box; 7. Assert input text box is empty.
Expected Results	All assertions are correctly validated.

Columns Selector

Table B.18: Test case CT-7.0 - Columns Selector Component.

ID	CT-7.0
Description	The test runner must be able to correctly render the component.
Prerequisites	N/A
Test Data	{ tableHeaders: [{id: 'id', isVisible: true, title: 'Component'}, {id: 'id', isVisible: true, title: 'Testing'}, {id: 'id', isVisible: true, title: 'Stratio Automotive'}] }
Steps	1. Mount the Columns Selector component;
Expected Results	The component is mounted without errors.

Table B.19: Test case CT-7.1 - Columns Selector Component.

ID	CT-7.1
Description	The test runner must be able to select a column from the drop down.
Prerequisites	N/A
Test Data	Same as Test Case B.18 (CT-7.0).
Steps	1. Mount Columns Selector component; 2. Assert 'Columns' button style and click it; 3. Assert options: 'Component', ' Testing' and 'Stratio Automotive'; 4. Disable 'Testing' checkbox; 5. Assert only 'Testing' checkbox is not checked.
Expected Results	All assertions are correctly validated as the 'Testing' checkbox is not checked.

Value Selector

Table B.20: Test case CT-8.0 - Value Selector Component.

ID	CT-8.0
Description	The test runner must be able to correctly render the component.
Prerequisites	N/A
Test Data	{ label: labelText, placeHolder: 'CT: Value Select', allowSearch: true, selectedValue: null, values: [{key: 'id', value: 'Component' }, {key: 'id2', value: 'Testing'}, {key: 'id3', value: 'Stratio Automotive'}] }
Steps	1. Mount the Value Selector component;
Expected Results	The component is mounted without errors.

Table B.21: Test case CT-8.1 - Value Selector Component.

ID	CT-8.1
Description	The test runner must be able to select a value from the drop down.
Prerequisites	N/A
Test Data	Same as Test Case B.20 (CT-8.0).
Steps	1. Mount Value Selector Component; 2. Assert label and selector style; 3. Type 'Stra' in input selector and press 'Enter'; 4. Validates the selected option is 'Stratio Automotive'.
Expected Results	All assertions are correctly validated and the component filled the text box with 'Stratio Automotive'.

Table B.22: Test case CT-8.2 - Value Selector Component.

ID	CT-8.2
Description	The test runner must be able to render the component with the selector disabled.
Prerequisites	N/A
Test Data	Same as Test Case B.20 (CT-8.0) with the following differences: { isDisabled: true, }
Steps	1. Mount the Value Selector component.
Expected Results	The component is mounted without errors.

Table B.23: Test case CT-8.3 - Value Selector Component.

ID	CT-8.3
Description	The test runner must be able to select multiple options at the same time.
Prerequisites	N/A
Test Data	Same as Test Case B.20 (CT-8.0) with the following differences: { selectMode: ′multiple′, }
Steps	1. Mount the Value Selector component; 2. Assert text box style; 3. Type ′Co′ and press ′Enter′; 4. Type ′Te′ and press ′Enter′; 5. Assert ′Component′ and ′Testing′ values are selected.
Expected Results	All assertions are correctly validated with multiple values selected.

Label Highlight Box

Table B.24: Test case CT-9.0 - Label Highlight Box Component.

ID	CT-9.0
Description	The test runner must be able to correctly render the component.
Prerequisites	N/A
Test Data	{ label: 'Highligh text', labelColor: '', highlightElement: 'label', leftAlign: false, title: 'Title', tooltipText: 'Tooltip' }
Steps	1. Mount the Label Highlight Box component;
Expected Results	The component is mounted without errors.

Label

Table B.25: Test case CT-10.0 - Label Component.

ID	CT-10.0
Description	The test runner must be able to correctly render the component.
Prerequisites	N/A
Test Data	{ title: 'Label', width: 150, modifier: ColorScheme. (danger | default | primary | success | info | warning), tooltipTitle: 'Label Tooltip', tooltipPosition: 'label', iconClass: 'stra-icon-exclamation-triangle', type: 'label' }
Steps	1. Mount the Label component;
Expected Results	The component is mounted without errors.

Table B.26: Test case CT-10.1 - Label Component.

ID	CT-10.1
Description	The test runner must be able to click the label button.
Prerequisites	N/A
Test Data	Same as Test Case B.25 (CT-10.0).
Steps	1. Mount the Label component; 2. Assert 'icon' button style; 3. Click button 'icon'.
Expected Results	All assertions are correctly validated.

Operation Status

Table B.27: Test case CT-11.0 - Operation Status Component.

ID	CT-11.0
Description	The test runner must be able to correctly render the component.
Prerequisites	N/A
Test Data	{ isOperating: false, withLabel: true, lastOperationDate: new Date() }
Steps	1. Mount the Operation Status component;
Expected Results	The component is mounted without errors.

Lists Resume

Table B.28: Test case CT-12.0 - Lists Resume Component.

ID	CT-1.0
Description	The test runner must be able to correctly render the component.
Prerequisites	N/A
Test Data	{ listsResumeTitle: ′CT: Lists Resume′, tooltipLabel: ′ToolTip′, list: [′Stratio′, ′Automotive′, ′Frontend′, ′Commons′], fontWeight: ′bold′, itemsToShow: 0 }
Steps	1. Mount the Lists Resume component;
Expected Results	The component is mounted without errors.

Table B.29: Test case CT-12.1 - Lists Resume Component.

ID	CT-1.1
Description	The test runner must be able to interact with the tool tip.
Prerequisites	N/A
Test Data	Same as Test Case B.28 (CT-12.0) with the following differences: { itemsToShow: 5 }
Steps	1. Mount Lists Resume Component; 2. Click ′ToolTip′ button; 3. Assert Label through list property ([′Stratio′, ′Automotive′, ′Frontend′, ′Commons′]).
Expected Results	All assertions are correctly validated as a Label appears with the defined ′list′ property from the Test Data.

Breadcrumbs

Table B.30: Test case CT-13.0 - Breadcrumbs Component.

ID	CT-13.0
Description	The test runner must be able to correctly render the component.
Prerequisites	N/A
Test Data	N/A
Steps	1. Mount the Breadcrumbs component;
Expected Results	The component is mounted without errors.

Table B.31: Test case CT-13.1 - Breadcrumbs Component.

ID	CT-13.1
Description	The test runner must be able to click the 'home' button.
Prerequisites	N/A
Test Data	N/A
Steps	1. Mount Breadcrumbs Component; 2. Assert content by inspecting the word 'page'; 3. Click 'home' link.
Expected Results	All assertions are correctly validated.

Page Header

Table B.32: Test case CT-14.0 - Page Header Component.

ID	CT-14.0
Description	The test runner must be able to correctly render the component.
Prerequisites	N/A
Test Data	{ showBreadcrumbs: true, additionalInfo: undefined }
Steps	1. Mount the Page Header component;
Expected Results	The component is mounted without errors.

Table B.33: Test case CT-14.1 - Page Header Component.

ID	CT-14.1
Description	The test runner must be able to click the ′Confirm′ and ′Cancel′ button.
Prerequisites	N/A
Test Data	Same as Test Case B.32 (CT-14.0) with the following differences: { additionalInfo: ′Stra. Breadcrumbs′ }
Steps	1. Mount Page Header Component; 2. Assert ′additionalInfo′ property content; 3. Click ′home′ link;"
Expected Results	All assertions are correctly validated.

Table B.34: Test case CT-14.2 - Page Header Component.

ID	CT-14.2
Description	The test runner must be able to correctly render the component without extra content.
Prerequisites	N/A
Test Data	{ showBreadcrumbs: false, additionalInfo: null }
Steps	1. Mount Page Header Component.
Expected Results	The component is mounted without errors.

Sub-Menu Filters Button

Table B.35: Test case CT-15.0 - Sub-Menu Filters Button Component.

ID	CT-15.0
Description	The test runner must be able to correctly render the component.
Prerequisites	N/A
Test Data	{ isFiltersOpen: false, isFiltersActive: false, counter: 0 }
Steps	1. Mount the Sub-Menu Filters Button component;
Expected Results	The component is mounted without errors.

Table Export Button

Table B.36: Test case CT-16.0 - Table Export Button Component.

ID	CT-16.0
Description	The test runner must be able to correctly render the component.
Prerequisites	N/A
Test Data	{ isLoading: (false | true) }
Steps	1. Mount the Table Export Button component;
Expected Results	The component is mounted without errors.

Table B.37: Test case CT-16.1 - Table Export Component.

ID	CT-16.1
Description	The test runner must be able to interact with the button.
Prerequisites	N/A
Test Data	Same as Test Case B.36 (CT-16.0).
Steps	1. Mount Table Export Button Component; 2. Assert ′download′ button style; 3. Click ′download′ button; 4. Assert ′download′ button style.
Expected Results	All assertions are correctly validated, as the button presents different styles depending on the interaction.

Table Refresh Button

Table B.38: Test case CT-17.0 - Table Refresh Button Component.

ID	CT-17.0
Description	The test runner must be able to correctly render the component.
Prerequisites	N/A
Test Data	N/A
Steps	1. Mount the Table Refresh Button component;
Expected Results	The component is mounted without errors.

Table B.39: Test case CT-17.1 - Table Refresh Button Component.

ID	CT-17.1
Description	The test runner must be able to interact with the button.
Prerequisites	N/A
Test Data	N/A
Steps	1. Mount Table Refresh Button Component; 2. Assert 'refresh' button style; 3. Click 'refresh' button; 4. Assert 'refresh' button style.
Expected Results	All assertions are correctly validated, as the button presents different styles depending on the interaction.

Table Row Action Button

Table B.40: Test case CT-18.0 - Table Row Action Button Component.

ID	CT-18.0
Description	The test runner must be able to correctly render the component.
Prerequisites	N/A
Test Data	{ iconTooltipTitle: 'Component Test: Stra. Automotive', iconCssClass: 'stra-icon-driver' }
Steps	1. Mount the Table Row Action Button component;
Expected Results	The component is mounted without errors.

Table B.41: Test case CT-18.1 - Table Row Action Button Component.

ID	CT-18.1
Description	The test runner must be able to interact with the button.
Prerequisites	N/A
Test Data	Same as Test Case B.40 (CT-18.0).
Steps	1. Mount Table Row Action Button Component; 2. Assert 'icon' button style; 3. Click 'icon' button.
Expected Results	All assertions are correctly validated, as the button presents different styles depending on the interaction.

B.2 Front-end Commons Tests Parallelism Data

The documentation of the parallelism test data for the component tests performed in the Front-end Commons Repository are as follows:

Table B.42: Component Tests Parallelism Data

Component Tests Parallelism Data					
Max. Workers	**Execution Time (s)**				
1	196.381	162.659	168.659	184.233	163.548
2	98.635	107.897	89.529	90.030	93.155
3	68.358	80.676	77.053	68.197	72.636
4	68.077	58.236	62.866	56.885	56.755
5	47.762	49.045	49.634	49.287	51.200
6	54.425	75.772	57.447	52.980	43.276
7	40.472	42.600	40.292	44.215	39.809
8	41.086	38.513	47.318	47.052	44.524
Unspecified	37.495	45.422	46.095	41.978	44.897

Table B.43: Component Tests Parallelism Data (Cont.)

Component Tests Parallelism Data (Cont.)					
Max. Workers	**Execution Time (s)**				
1	179.573	160.281	173.566	168.851	196.243
2	86.885	93.551	87.128	90.401	98.958
3	76.293	75.667	69.686	75.330	70.034
4	62.430	70.250	54.245	56.635	58.823
5	66.769	48.865	53.929	55.995	60.561
6	42.691	49.556	41.877	44.082	43.864
7	43.430	40.494	42.458	48.795	48.897
8	39.224	47.317	42.824	46.335	40.259
Unspecified	45.281	44.252	42.558	42.478	46.158

High Level Local GitLab runner specifications:

- 8 vCPU;
- 16 GB Ram;
- 500Gb SSD;
- Windows Server 2019 std.

Appendix C

Information Regarding E2E Tests

Appendix C includes information regarding E2E tests, including all the test cases, parallelism data for Playwright Test Runner and approval metrics before automated tests.

C.1 Test Cases for E2E Testing

The documentation of the test cases for the E2E tests performed in the Stratio-Web Repository are as follows:

Login

Table C.1: Test case E2E-1.0 - Login: Authentication.

ID	E2E-1.0
Description	Authenticate in the Stratio Foresight Platform.
Prerequisites	N/A
Test Data	Any valid user credentials.
Steps	1. Open Stratio Foresight Platform; 2. Fill the username text box; 3. Fill the password text box; 4. Click 'Login' button; 5. Assert user's dashboard has been loaded.
Expected Results	User is correctly authenticated.

Table C.2: Test case E2E-1.1 - Login: Recover Password.

ID	E2E-1.1
Description	Ask for a new password.
Prerequisites	N/A
Test Data	N/A
Steps	1. Open Stratio Foresight Platform; 2. Click Recover Password Option; 3. Assert new page to have URL '/Account/ForgotPassword'; 4. Fill recovery email text box; 5. Click 'Recover Password' button; 6. Assert recovery message and click 'Ok' button.
Expected Results	User is sent an email with the steps to replace their password.

Logout

Table C.3: Test case E2E-2.0 - Logout: Log Off.

ID	E2E-2.0
Description	Perform logout.
Prerequisites	A user is signed in to the Stratio Foresight Platform.
Test Data	N/A
Steps	1. Click the 'Logout' button; 2. Assert user is redirected to the login page.
Expected Results	User is correctly unauthenticated.

Fleet Condition

Table C.4: Test case E2E-3.0 - Fleet Condition: Consult Vehicle Details.

ID	E2E-3.0
Description	Smoke test to examine details of a specific vehicle.
Prerequisites	A user is signed in to the Stratio Foresight Platform.
Test Data	Any valid user credentials and a vehicle ID (vehicleNumber).

Steps	1. Redirect page to '/fleetCondition/vehicleNumber';
	2. Assert vehicle's cards information:
	(a) Name, brand and model;
	(b) Databox Connection Status;
	(c) Active Alerts;
	(d) Active DTC;
	(e) Next Services;
	(f) System Indicators;
	(g) Operation Metrics.

Expected Results	All assertions are correctly validated.

Table C.5: Test case E2E-3.1 - Fleet Condition: Consult Active Alerts.

ID	E2E-3.1
Description	Smoke test to examine the active alerts of a specific vehicle.
Prerequisites	A user is signed in to the Stratio Foresight Platform.
Test Data	Any valid user credentials and a vehicle ID (vehicleNumber) with active alerts.
Steps	1. Redirect page to '/fleetCondition/vehicleNumber'; 2. Click 'View Alerts' text; 3. Assert Pop-up content (list of active alerts); 4. Click 'View all Active ALerts'; 5. Assert new list of current alerts.
Expected Results	All assertions are correctly validated.

Table C.6: Test case E2E-3.2 - Fleet Condition: Consult Active DTCs.

ID	E2E-3.2
Description	Smoke test to examine the active DTC of a specific vehicle.
Prerequisites	A user is signed in to the Stratio Foresight Platform.
Test Data	Any valid user credentials and a vehicle ID (vehicleNumber) with active DTC.
Steps	1. Redirect page to '/fleetCondition/vehicleNumber'; 2. Click 'View DTCs' text; 3. Assert Pop-up content (list of active DTC); 4. Click 'View all Active DTC'; 5. Assert new list of current DTC.
Expected Results	All assertions are correctly validated.

System Indicators

Table C.7: Test case E2E-4.0 - System Indicators: Overview.

ID	E2E-4.0
Description	Smoke test to examine various cards regarding system indicators statistics.
Prerequisites	A user is signed in to the Stratio Foresight Platform.
Test Data	Any valid user credentials.
Steps	1. Redirect page to '/system-indicators'; 2. Assert system indicator's cards information: (a) Starter Battery; (b) Brake Pads (Estimated); (c) Available Engine Torque; (d) Air Leaks (Beta); (e) Electric Vehicles - Battery Pack.
Expected Results	All assertions are correctly validated.

Table C.8: Test case E2E-4.1 - System Indicators: Starter Battery.

ID	E2E-4.1
Description	Smoke test to examine starter battery of all vehicles.
Prerequisites	A user is signed in to the Stratio Foresight Platform.
Test Data	Any valid user credentials.
Steps	1. Redirect page to '/system-indicators/battery-indicators'; 2. Asserts table and filter properties; 3. Clicks the first row and asserts: (a) History; (b) Battery Data Insight; (c) Users Feedback.
Expected Results	All assertions are correctly validated.

Table C.9: Test case E2E-4.2 - System Indicators: Brake Pads.

ID	E2E-4.2
Description	Smoke test to examine brake pads of all vehicles.
Prerequisites	A user is signed in to the Stratio Foresight Platform.
Test Data	Any valid user credentials.
Steps	1. Redirect page to '/system-indicators/brake-pads'; 2. Asserts table and filter properties; 3. Switch from 'Remaining Life' to 'Current State' and assert table changes.
Expected Results	All assertions are correctly validated.

Table C.10: Test case E2E-4.3 - System Indicators: Available Engine Torque.

ID	E2E-4.3
Description	Smoke test to examine engine torque of all vehicles.
Prerequisites	A user is signed in to the Stratio Foresight Platform.
Test Data	Any valid user credentials.
Steps	1. Redirect page to '/system-indicators/engine-torque'; 2. Asserts table and filter properties.
Expected Results	All assertions are correctly validated.

Table C.11: Test case E2E-4.4 - System Indicators: Air Leaks.

ID	E2E-4.4
Description	Smoke test to examine air leaks of all vehicles.
Prerequisites	A user is signed in to the Stratio Foresight Platform.
Test Data	Any valid user credentials.
Steps	1. Redirect page to '/system-indicators/air-leaks'; 2. Asserts table and filter properties.
Expected Results	All assertions are correctly validated.

Table C.12: Test case E2E-4.4 - System Indicators: Battery Pack - EV.

ID	E2E-4.4
Description	Smoke test to examine battery pack of all EV vehicles.
Prerequisites	A user is signed in to the Stratio Foresight Platform.
Test Data	Any valid user credentials.
Steps	1. Redirect page to '/system-indicators/ev-battery-assessment'; 2. Asserts table and filter properties.
Expected Results	All assertions are correctly validated.

Table C.13: Test case E2E-4.5 - System Indicators: Potential Fault.

ID	E2E-4.4
Description	Submit form with potential faults of vehicles.
Prerequisites	A user is signed in to the Stratio Foresight Platform.
Test Data	Any valid user credentials and sample data used for completing the potential fault form.
Steps	1. Navigate to System Indicators - Potential Faults; 2. Asserts table and filter properties; 3. Click 'Report Other Faults' button; 4. Fill in form with placeholder data; 5. Click 'Submit' button; 6. Assert confirmation pop-up.
Expected Results	A new potential fault is created with the defined placeholder data, a pop-up confirming its creation is issued, and the potential fault is available for future consultation.

Metrics

Table C.14: Test case E2E-5.0 - Metrics: Consumption.

ID	E2E-5.0
Description	Smoke test to examine consumption of internal combustion, hybrids and electrics.
Prerequisites	A user is signed in to the Stratio Foresight Platform.
Test Data	Any valid user credentials.
Steps	1. Redirect page to '/metrics/consumption'; 2. Asserts table and filter properties; 3. Asserts 3 categories: (a) Internal Combustion; (b) Hybrid; (c) Electric.
Expected Results	All assertions are correctly validated.

Table C.15: Test case E2E-5.1 - Metrics: Throttle Pedal.

ID	E2E-5.1
Description	Smoke test to examine throttle pedal usage of all vehicles.
Prerequisites	A **super master** user is signed in to the Stratio Foresight Platform.
Test Data	Valid **super master** credentials.
Steps	1. Redirect page to '/metrics/throttle'; 2. Asserts table and filter properties.
Expected Results	All assertions are correctly validated.

Table C.16: Test case E2E-5.2 - Metrics: Coolant Temperature.

ID	E2E-5.2
Description	Smoke test to examine coolant temperature of all vehicles.
Prerequisites	A **super master** user is signed in to the Stratio Foresight Platform.
Test Data	Valid **super master** credentials.
Steps	1. Redirect page to '/metrics/coolant-temperature'; 2. Asserts table and filter properties; 3. Switch from 'Time' to 'Percentage' and assert table changes.
Expected Results	All assertions are correctly validated.

Table C.17: Test case E2E-5.3 - Metrics: EV Charging.

ID	E2E-5.3
Description	Smoke test to examine EV charging of all vehicles.
Prerequisites	A user is signed in to the Stratio Foresight Platform.
Test Data	Any valid user credentials.
Steps	1. Redirect page to '/metrics/evCharging'; 2. Asserts table and filter properties.
Expected Results	All assertions are correctly validated.

Table C.18: Test case E2E-5.4 - Metrics: Operation.

ID	E2E-5.4
Description	Smoke test to examine operation mode of all vehicles.
Prerequisites	A user is signed in to the Stratio Foresight Platform.
Test Data	Any valid user credentials.
Steps	1. Redirect page to '/metrics/operation'; 2. Asserts table and filter properties.
Expected Results	All assertions are correctly validated.

Vehicle Recent Data

Table C.19: Test case E2E-6.0 - Vehicle Recent Data: Display.

ID	E2E-6.0
Description	Search for a specific vehicle and consult its data.
Prerequisites	A user is signed in to the Stratio Foresight Platform.
Test Data	Any valid user credentials and a vehicle identifier with a valid date.
Steps	1. Navigate to Vehicle Recent Data; 2. Click 'Historic' button; 3. Fill text box with the vehicle identifier; 4. Select valid date; 5. Analyze pretended category; 6. Assert generated graphic properties.
Expected Results	The generated graphic properties contain appropriate labels for the displayed graphic (x-axis, y-axis, and title).

Occurrences

Table C.20: Test case E2E-7.0 - Occurrences: Filtering and Marking as Read.

ID	E2E-7.0
Description	Filter from existing occurrences and mark a specific occurrence as read.
Prerequisites	A user is signed in to the Stratio Foresight Platform.
Test Data	Any valid user credentials and sample data for filter properties.
Steps	1. Navigate to Occurrences; 2. Click 'Filters' button; 3. Configure filters parameters: (a) Period; (b) Vehicle select; (c) Type; (d) Severity; (e) System; (f) DTC filter toogle; (g) Aggregate notifications toggle; 4. Click 'Accept' button; 5. Select filtered occurrence's checkbox; 6. Click 'Mark as Read' button; 7. Assert confirmation pop-up; 8. Asserts row is not bold anymore.
Expected Results	Occurrence is no longer marked as bold (if applicable) and it is presented a new pop-up stating the number of occurrences marked as read.

Service Plans

Table C.21: Test case E2E-8.0 - Service Plans: Create service plan, mark it as done and cleanup.

ID	E2E-8.0
Description	Fill a form to create a new service plan, examine and mark a specific it as done, followed by its deletion.
Prerequisites	A user is signed in to the Stratio Foresight Platform.
Test Data	Any valid user credentials and sample data for service plan creation.
Steps	1. Navigate to Service Plans; 2. Click 'Service Plans' text; 3. Click '+ Create' button; 4. Configure three-step form of service plan creation: (a) Name; (b) Recurrence; (c) Description; (d) Task (Action, Component/Material and Notes); (e) Vehicle selection; (f) Click 'Create' button; (g) Assert success pop-up. 5. Click 'Next Services' text; 6. Click newly created service plan; 7. Click 'Mark Service as Done' button; 8. Click 'Service Plans' text; 9. Click 'X' icon in the row of the newly created service plan; 10. Assert success pop-up.
Expected Results	Service plan is marked as done and it is presented a new pop-up stating that the service occurrence was registered.

Reports

Table C.22: Test case E2E-9.0 - Reports: Consult Reports.

ID	E2E-9.0
Description	Smoke test to examine reports.
Prerequisites	A user is signed in to the Stratio Foresight Platform.
Test Data	Any valid user credentials.
Steps	1. Redirect page to ′/reports/received-reports′; 2. Asserts table and filter properties.
Expected Results	All assertions are correctly validated.

Maps

Table C.23: Test case E2E-10.0 - Maps: Map.

ID	E2E-10.0
Description	Interact with Google Maps widget and search drivers/vehicles.
Prerequisites	A user is signed in to the Stratio Foresight Platform.
Test Data	Any valid user credentials and placeholder data for vehicle and driver properties.
Steps	1. Navigate to Maps - Map; 2. Search through plate number; 3. Click the pretended vehicle; 4. Click 'See More' text; 5. Assert summary information; 6. Interact with Google Maps widget: (a) Click eye icon and assert content; (b) Click routes widget button and assert content; 7. Navigate to Trailers tab; 8. Quick search for a trailer and assert successful inquiry; 9. Navigate to Locations tab; 10. Quick search for a location and assert successful inquiry; 11. Navigate to Addresses tab; 12. Quick search for an address and assert successful inquiry.
Expected Results	All assertions are correctly validated in the 'Maps', 'Trailers', 'Locations' and 'Addresses' tabs.

Table C.24: Test case E2E-10.1 - Maps: Trips.

ID	E2E-10.1
Description	Smoke test to examine existing trips of various vehicles.
Prerequisites	A user is signed in to the Stratio Foresight Platform.
Test Data	Any valid user credentials.
Steps	1. Redirect page to '/maps/trips'; 2. Asserts table and filtering properties.
Expected Results	All assertions are correctly validated.

Table C.25: Test case E2E-10.2 - Maps: Driver Hours of Service.

ID	E2E-10.2
Description	Smoke test to examine driver's hours of services.
Prerequisites	A user is signed in to the Stratio Foresight Platform.
Test Data	Any valid user credentials.
Steps	1. Redirect page to '/maps/driverHoursOfService'; 2. Asserts table and filtering properties.
Expected Results	All assertions are correctly validated.

Table C.26: Test case E2E-10.3 - Maps: Geo-referenced Occurrences.

ID	E2E-10.3
Description	Smoke test to examine geo-referenced occurrences with the possibility of marking them as read.
Prerequisites	A user is signed in to the Stratio Foresight Platform.
Test Data	Any valid user credentials.
Steps	1. Redirect page to '/maps/geo-referenced-occurrences'; 2. Asserts table and filtering properties.
Expected Results	All assertions are correctly validated.

Table C.27: Test case E2E-10.4 - Maps: Messages.

ID	E2E-10.4
Description	Search existing messages and create new ones.
Prerequisites	A user is signed in to the Stratio Foresight Platform.
Test Data	Any valid user credentials and placeholder data for message creation properties.
Steps	1. Navigate to Maps - Messages; 2. Click '+ New Message' button; 3. Fill Message form: (a) Send to 'Vehicles' or 'Drivers'; (b) Select vehicles; (c) Fill message field; (d) Optionally add other locations; 4. Click 'Send' button; 5. Assert successful pop-up.
Expected Results	A new message is created with the placeholder data, a pop-up confirming its creation is issued, and the message is available for consultation.

Table C.28: Test case E2E-10.5 - Maps: Operational Events.

ID	E2E-10.5
Description	Smoke test to examine operational events.
Prerequisites	A user is signed in to the Stratio Foresight Platform.
Test Data	Any valid user credentials.
Steps	1. Redirect page to '/maps/events/operational'; 2. Asserts table and filtering properties.
Expected Results	All assertions are correctly validated.

Table C.29: Test case E2E-10.6 - Maps: Geo-referenced Alerts.

ID	E2E-10.6
Description	Search existing geo-referenced alerts and create new ones.
Prerequisites	A user is signed in to the Stratio Foresight Platform.
Test Data	Any valid user credentials and placeholder data for geo-referenced alerts creation properties.
Steps	1. Navigate to Maps - Geo-referenced Alerts; 2. Click '+ Create' button; 3. Fill in three-step form: (a) Alert Name; (b) Description; (c) Event; (d) Locations; (e) Vehicles; (f) Delay; (g) Alert Expiration; (h) Trigger Repetition; (i) External User Emails; 4. Click 'Save' button; 5. Assert successful pop-up; 6. Quick search for newly created geo-referenced alert; 7. Click row; 8. Assert creation properties; 9. Click delete button; 10. Assert deletion pop-up.
Expected Results	A new geo-referenced alert is created with the placeholder data, a pop-up confirming its creation is issued, and the geo-referenced alert is available for consultation.

Table C.30: Test case E2E-10.7 - Maps: Shared Vehicle Locations.

ID	E2E-10.7
Description	Search existing shared vehicle locations and create new ones.
Prerequisites	A user is signed in to the Stratio Foresight Platform.
Test Data	Any valid user credentials and placeholder data for shared vehicle locations creation properties.
Steps	1. Navigate to Maps - Shared Vehicle Locations; 2. Click '+ Create' button; 3. Fill in creation form: (a) Vehicle; (b) Additional information; (c) Period; (d) Share Name; (e) Email; (f) Language; (g) Message; 4. Click 'Share' button; 5. Assert successful pop-up; 6. Quick search for newly created shared vehicle location; 7. Click delete button; 8. Assert deletion pop-up.
Expected Results	A new shared vehicle location is created with the placeholder data, a pop-up confirming its creation is issued, and the shared vehicle location is available for consultation.

Ecodrive

Table C.31: Test case E2E-11.0 - Ecodrive: Overview.

ID	E2E-11.0
Description	Smoke test to examine various cards regarding driving statistics and how ecological are they style of driving.
Prerequisites	A user is signed in to the Stratio Foresight Platform.
Test Data	Any valid user credentials.

Steps	1. Redirect page to '/ecodrive/dashboard';
	2. Asserts dashboard cards content:
	(a) Scoring Overview;
	(b) Eco Score Efficiency;
	(c) Top Drivers;
	(d) Consumption;
	(e) Consumption by Group;
	(f) Top Vehicles;

Expected Results	All assertions are correctly validated.

Table C.32: Test case E2E-11.1 - Ecodrive: Driver Score.

ID	E2E-11.1
Description	Examine driver scores of all drivers.
Prerequisites	A user is signed in to the Stratio Foresight Platform.
Test Data	Any valid user credentials and sample data for a valid driver.

Steps	1. Navigate to Ecodrive - Driver Score;
	2. Assert insights:
	(a) Average Global Score;
	(b) Metrics;
	(c) Average Consumption;
	3. Generate xls report and download it;
	4. Assert download pop-up;
	5. Quick search for driver;
	6. Click row and asserts driver′s metrics:
	(a) Global Score;
	(b) Average Consumption;
	(c) Calculated Distance;
	(d) Score Graph;
	(e) Additional Information;
	7. Export PDF with driver′s statistics;
	8. Assert download pop-up.

Expected Results	All assertions are correctly validated and the appropriate xls reports are downloaded.

Table C.33: Test case E2E-11.2 - Ecodrive: Vehicle Score.

ID	E2E-11.2
Description	Examine driver scores of all vehicles.
Prerequisites	A user is signed in to the Stratio Foresight Platform.
Test Data	Any valid user credentials and sample data for a valid vehicle.

Steps	1. Navigate to Ecodrive - Vehicle Score;
	2. Assert insights:
	(a) Average Global Score;
	(b) Metrics;
	(c) Average Consumption;
	3. Quick search for vehicle;
	4. Click row and asserts vehicle's metrics:
	(a) Global Score;
	(b) Average Consumption;
	(c) Calculated Distance;
	(d) Score Graph;
	(e) Additional Information;

Expected Results	All assertions are correctly validated.

Table C.34: Test case E2E-11.3 - Ecodrive: Bus Line Score.

ID	E2E-11.3
Description	Smoke test to examine bus line scores of all bus lines.
Prerequisites	A user is signed in to the Stratio Foresight Platform.
Test Data	Any valid user credentials.
Steps	1. Redirect page to '/ecodrive/bus-lines-scoring'; 2. Asserts table and filtering properties.
Expected Results	All assertions are correctly validated.

Table C.35: Test case E2E-11.4 - Ecodrive: Driver Configuration.

ID	E2E-11.4
Description	Smoke test to examine driver's configurations of all drivers.
Prerequisites	A user is signed in to the Stratio Foresight Platform.
Test Data	Any valid user credentials.
Steps	1. Redirect page to '/ecodrive/drivers-configuration'; 2. Asserts table and filtering properties.
Expected Results	All assertions are correctly validated.

Table C.36: Test case E2E-11.5 - Ecodrive: Groups Management.

ID	E2E-11.5
Description	Create Ecodrive groups, search and edit them.
Prerequisites	A user is signed in to the Stratio Foresight Platform.
Test Data	Any valid user credentials and sample data for the creation of a valid Ecodrive group.
Steps	1. Navigate to Ecodrive - Groups Management; 2. Click '+ Create' button; 3. Fill in creation form: (a) Group Name; (b) Drivers Managers; (c) Drivers; 4. Click 'Save' button; 5. Assert successful pop-up; 6. Quick search for newly created group; 7. Edit group by adding another driver; 8. Cleanup of newly created group; 9. Assert deletion pop-up.
Expected Results	A new Ecodrive group is created with the placeholder data, a pop-up confirming its creation is issued, and the Ecodrive group is available to edit/add drivers.

My Settings

Table C.37: Test case E2E-12.0 - My Settings: General Settings and Measuring Units.

ID	E2E-12.0
Description	Select different languages and time zones from General Settings and different measurement units for distance, temperature and pressure from Measuring Units.
Prerequisites	A user is signed in to the Stratio Foresight Platform.
Test Data	Any valid user credentials and placeholder data for different settings properties.
Steps	1. Navigate to My Settings; 2. Select different language; 3. Select different timezone; 4. Click 'Save' button and assert success pop-up; 5. Navigate to Dashboard and assert language and timezone changes; 6. Navigate to General Settings and reset general settings changes; 7. Click Measuring Units tab; 8. Select different units system; 9. Select different temperature units; 10. Select different pressure system; 11. Click 'Save' button and assert success pop-up; 12. Navigate to Metrics - Consumption and assert units system changes; 13. Navigate to Metrics - Coolant Temperature and assert temperature units changes; 14. Navigate to Vehicle Data and assert pressure system changes; 15. Navigate to Measuring Units and reset general settings changes.
Expected Results	All assertions are correctly validated.

Table C.38: Test case E2E-12.1 - My Settings: Notifications.

ID	E2E-12.1
Description	Smoke test to examine the notifications options.
Prerequisites	A user is signed in to the Stratio Foresight Platform.
Test Data	Any valid user credentials.

Steps	1. Redirect page to '/ProfileSettings/Notifications';
	 2. Assert page properties:
	 (a) Severity Level;
	 (b) Email;
	 (c) SMS;
	 (d) Pop-up;
	 (e) Notification checkboxes.

Expected Results	All assertions are correctly validated.

Table C.39: Test case E2E-12.2 - My Settings: Edit Profile.

ID	E2E-12.1
Description	Smoke test to examine edit profile form.
Prerequisites	A user is signed in to the Stratio Foresight Platform.
Test Data	Any valid user credentials.

Steps	1. Redirect page to '/ProfileSettings/UserDetails';
	 2. Assert page properties:
	 (a) Name;
	 (b) Email;
	 (c) Phone number;
	 (d) Password.

Expected Results	All assertions are correctly validated.

Global Settings

Table C.40: Test case E2E-13.0 - Global Settings: Groups.

ID	E2E-13.0
Description	Creation of new groups.
Prerequisites	A user is signed in to the Stratio Foresight Platform.
Test Data	Any valid user credentials and placeholder data for group creation properties.
Steps	1. Navigate to Global Settings; 2. Click '+ Add new group' button; 3. Fill in group name; 4. Click 'Submit' button; 5. Assert success pop-up; 6. Cleanup of newly created group; 7. Assert success pop-up;
Expected Results	A new group is created with the placeholder data, a pop-up confirming its creation is issued, and the group is available for edit.

Table C.41: Test case E2E-13.1 - Global Settings: Idle Time.

ID	E2E-13.1
Description	Smoke test to examine idle time by vehicle type.
Prerequisites	A user is signed in to the Stratio Foresight Platform.
Test Data	Any valid user credentials.
Steps	1. Redirect page to '/CompanySettings/IdleTime'; 2. Asserts table and maximum idle time by vehicle type.
Expected Results	All assertions are correctly validated.

Table C.42: Test case E2E-13.2 - Global Settings: Ignored Trouble Codes/Alerts.

ID	E2E-13.2
Description	Smoke test to examine list of ignored trouble codes and alerts.
Prerequisites	A user is signed in to the Stratio Foresight Platform.
Test Data	Any valid user credentials.
Steps	1. Redirect page to '/CompanySettings/IgnoredFaults'; 2. Asserts Trouble Code table; 3. Switch from 'Trouble code' to 'Alerts' and assert table changes.
Expected Results	All assertions are correctly validated.

List Users

Table C.43: Test case E2E-14.0 - List Users: Disable/Enable User.

ID	E2E-14.0
Description	Disable a user and confirm they can not log in. Then, enable the same user and confirm they can log in into Stratio Foresight Platform.
Prerequisites	A **super master** user is signed in to the Stratio Foresight Platform with another test-runner open with another user on standby.
Test Data	A valid **super user** and any other user credentials.
Steps	

Super Master User	**Other User**
1. Navigate to 'Account-Management/ListUsers'; 2. Open pretended user profile; 3. Disable user; 4. Assert user is inactive;	
	5. Try to login into the platform; 6. Assert error message: 'User is inactive';
7. Enable user; 8. Assert user is active;	
	9. Try to login into the platform; 10. Assert that user's dashboard is correctly loaded.

Expected Results	The other user is disabled by the super master user from the Stratio Foresight Platform temporarily.

Table C.44: Test case E2E-14.1 - List Users: Examine User.

ID	E2E-14.1
Description	Smoke test to examine a specific user.
Prerequisites	A **super master** user is signed in to the Stratio Foresight Platform.
Test Data	A valid **super master** user credentials and sample data of an existing user.
Steps	1. Redirect page to '/AccountManagement/ListUsers'; 2. Asserts page properties: (a) Title; (b) Email; (c) Role; (d) State; (e) Last login. 3. Click the pretended user; 4. Assert user properties: (a) Name; (b) Phone number; (c) Email; (d) Notification settings.
Expected Results	All assertions are correctly validated.

C.2 Stratio-Web Tests Parallelism Data

The documentation of the parallelism data for the E2E tests performed in the Stratio-Web Repository are as follows:

Table C.45: E2E Tests Parallelism Data.

E2E Tests Parallelism Data						
GitLab Parallel Jobs	**Playwright Workers**	**Execution Time (s)**				
1	1	1921	1880	1819	1943	1921
	2	1190	1233	1305	1232	1162
	3	1092	1076	1029	1017	1065
	4	953	927	888	973	948
2	1	1143	1093	1166	1116	1180
	2	903	877	875	816	871
	3	828	843	821	845	825
	4	767	740	764	794	756
3	1	985	960	927	947	922
	2	719	746	768	758	777
	3	717	716	766	753	742
	4	781	740	729	704	700

High Level Local GitLab runner specifications:

- 8 vCPU;
- 16 GB Ram;
- 500Gb SSD;
- Windows Server 2019 std.

C.3 Approval Metrics without Automated Tests

Data retrieved by accompanying the product team during two sprints (fifteen days each) in order to extrapolate adequate approval metrics.

Table C.46: Approval Metrics from the product team, without automated tests.

ID	Complexity	Bugs Found	Validation Time (min)	Type
1	Low	1	12	User Story
2	Low	1	13	Bug Fix
3	High	0	60	UAT Validation
4	Low	1	8	Bug Fix
5	Low	0	17	Bug Fix
6	Low	1	11	Bug Fix
7	Low	0	12	Bug Fix
8	Medium	1	29	Bug Fix
9	Low	0	13	User Story
10	High	0	120	UAT Validation
11	High	0	240	UAT Validation
12	Low	0	14	User Story
13	Low	0	15	User Story
14	Low	0	5	User Story
15	Low	0	11	Bug Fix
16	Low	0	5	Bug Fix
17	Low	1	13	Bug Fix
18	Low	0	5	Bug Fix
19	Low	0	9	Bug Fix
20	Low	0	5	Bug Fix
21	Low	1	5	Bug Fix
22	Low	0	7	Bug Fix
23	Medium	0	120	Bug Fix
24	Low	0	5	Bug Fix
25	High	4	110	UAT Validation
26	Medium	1	60	Bug Fix
27	Low	0	8	User Story
28	Low	0	5	Bug Fix

Complexity Classification

- **Low:** small validation of an independent feature;
- **Medium:** complex feature with a high level of integration (dependent on other functionalities);
- **High:** maximum integration, reserved for the "UAT Validation" type of approval.