

Optimization of the software testing process to address challenges of shorter release cycles by the example of a global logistics software provider

Masterarbeit

Eingereicht von: **Lena Thoma, BA**

Matrikelnummer: 51841018

im Fachhochschul-Masterstudiengang Wirtschaftsinformatik
der Ferdinand Porsche FernFH GmbH

zur Erlangung des akademischen Grades

Master of Arts in Business

Betreuung und Beurteilung: DI Dr. Werner Toplak

Zweitgutachten: FH-Prof. Dipl.-Ing. Dr. techn. Dr.-Ing. Gernot Kucera, MA

Wiener Neustadt, Mai 2023

Ehrenwörtliche Erklärung

Ich versichere hiermit,

1. dass ich die vorliegende Masterarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle Inhalte, die direkt oder indirekt aus fremden Quellen entnommen sind, sind durch entsprechende Quellenangaben gekennzeichnet.
2. dass ich diese Masterarbeit bisher weder im Inland noch im Ausland in irgendeiner Form als Prüfungsarbeit zur Beurteilung vorgelegt oder veröffentlicht habe.
3. dass die vorliegende Fassung der Arbeit mit der eingereichten elektronischen Version in allen Teilen übereinstimmt.

Wiener Neustadt, 21.05.2023

Unterschrift

Abstract: Optimization of the software testing process to address challenges of shorter release cycles by the example of a global logistics software provider

The trend in software development is moving more and more towards shorter release cycles. Thereby, the manual effort associated with the software testing process is a major challenge. Each release cycle involves recurring activities which are worth to automated. Most of the people only think about the automation of manual test cases when trying to reduce the manual effort. Beside that also the automation of administrative tasks should be considered.

In this thesis, the software testing process at Alpega, an international logistics software provider, was analysed in order to identify weaknesses and increase the degree of automation. The implementation of a prototype showed that almost 95% of the manual administrative activities could be automated. This reduces the manual effort per release cycle by more than 30%. The optimisation was achieved by integrating the tools TestRail and GitLab which are in use at Alpega. The greatest savings in effort were achieved through the automatic creation and update of test plans, the centralisation of activities and the simplification of the test execution.

Keywords:

Test management; Test reporting; Test automation; CI/CD; TestRail; GitLab

Kurzzusammenfassung: Optimierung des Software Test Prozesses zur Bewältigung der Herausforderungen von kürzeren Release-Zyklen am Beispiel eines globalen Logistiksoftwareanbieters

Der Trend in der Softwareentwicklung geht immer mehr in Richtung kurze Release-Zyklen. Dabei stellt der manuelle Aufwand verbunden mit dem Softwaretestprozess eine große Herausforderung dar. Jeder Release-Zyklus bringt wiederkehrende Tätigkeiten zur Sicherung der Softwarequalität mit sich. Beim Versuch die manuellen Aufwände zu verringern, wird oft nur die Automatisierung von manuellen Testfällen in Betracht gezogen. Darüber hinaus sollten allerdings auch administrative Tätigkeiten wie beispielsweise die Erstellung von Testplänen berücksichtigt werden.

Im Rahmen der Arbeit wurde der Softwaretestprozess bei Alpega, einem internationalen Logistiksoftwareanbieter, untersucht, um Schwachstellen aufzudecken und den Grad der Automatisierung zu erhöhen. Die Implementierung eines Prototyps hat gezeigt, dass fast 95% der manuellen administrativen Tätigkeiten automatisiert werden konnten. Dadurch wurde der manuelle Aufwand pro Release-Zyklus um mehr als 30% reduziert. Die Optimierung wurde durch die Integration der eingesetzten Tools TestRail und GitLab erreicht. Die größten Aufwandseinsparungen wurden dabei durch die automatische Erstellung und Aktualisierung der Testpläne, die Zentralisierung von Tätigkeiten sowie durch die Vereinfachung der Testausführung erzielt.

Schlagwörter:

Testmanagement; Test Reporting; Testautomatisierung; CI/CD; TestRail; GitLab

Table of contents

1	Introduction	1
1.1	Motivation.....	1
1.2	Problem Definition.....	2
1.3	Research Objectives	3
1.4	Research Question and Hypothesis.....	4
1.5	Research Design	5
2	Literature Review	6
2.1	Software Testing.....	6
2.1.1	Agile Software Testing	8
2.1.2	Software Testing in CI/CD	10
2.2	Software Test Management.....	11
2.3	Test Automation	13
2.3.1	Test automation in a narrow sense	13
2.3.2	Test automation in a broader sense	15
2.4	Best Practices with TestRail	16
3	Review of Current Process.....	18
3.1	Software Testing Strategy.....	18
3.1.1	Terminology	22
3.1.2	Test documentation and reporting	23
3.1.3	Test automation.....	24
3.1.4	Test execution and maintenance.....	24
3.2	Release Process.....	29
3.2.1	Shorter release cycles	32

3.2.2	CI/CD	32
3.3	Software Testing Process.....	33
3.3.1	Workflow	33
3.3.2	Effort of Recurring Testing Activities	36
3.3.3	Weak Points.....	38
3.3.4	Improvement Opportunities.....	39
4	Solution Approach.....	40
4.1	Overview	40
4.2	Basic Release Workflow in TestRail.....	42
4.2.1	Create milestones for release	42
4.2.2	Start milestone "Release"	43
4.2.3	Start milestone "Regression".....	43
4.2.4	Start milestone "Smoke".....	44
4.2.5	Close test plans of previous Release.....	44
4.3	Additional Functionalities.....	44
4.3.1	Execution of all test cases within a test plan.....	44
4.3.2	Execution of failed test cases within a test plan.....	46
4.3.3	Test plan creation independent of milestone start	46
4.4	Test Plan Creation.....	47
4.4.1	Definition of Scope	47
4.4.2	Mapping of TestRail and GitLab projects	48
4.4.3	Relevant GitLab parameters.....	49
4.5	Test Plan Determination	51
4.6	Automatic Assignment of new Test Cases to Test Plans.....	51
5	Implementation of Prototype.....	52

5.1	Overview	52
5.2	Basic Release Workflow in TestRail.....	54
5.2.1	Create milestones for release	54
5.2.2	Start milestone "Release"	57
5.2.3	Start milestone "Regression".....	60
5.2.4	Start milestone "Smoke".....	64
5.2.5	Close test plans of previous Release.....	65
5.3	Additional Functionalities.....	66
5.3.1	Execution of all test cases within a test plan.....	66
5.3.2	Execution of failed test cases within a test plan.....	69
5.3.3	Test plan creation independent of milestone start	70
5.4	Test Plan Creation.....	73
5.5	Test Plan Determination	76
5.6	Automatic Assignment of new Test Cases to Test Plans.....	76
6	Evaluation.....	77
6.1	Comparison of Efforts	77
6.2	KPI system.....	82
6.3	Review of Research Question and Hypothesis	84
6.4	Summary of Outcomes.....	84
6.5	Review of Research Objectives	85
7	Conclusion and Outlook.....	87
	References	88
	List of Figures.....	91
	List of Tables.....	93

1 Introduction

Software testing is a key success factor in software development. It ensures that the requirements of the product are met, and the software is free of any defects. In the end the goal is to provide a software which satisfies existing and future customers. As it is the last step in the software development cycle before the software is delivered to the customer, Software Quality Managers often suffer with high time pressure. Especially with trend that software release cycles are getting shorter and shorter, it is even more important to optimize the software testing process (Alsaqqa et al., 2020).

Nowadays more and more software providers following the concept of Continuous Integration / Continuous Delivery (CI/CD) which means that the software is delivered to the customer immediately after development and testing. This leads to the fact, that the software testing activities are conducted in a more frequent basis. To address this challenge, it is essential to reduce the manual tasks to a minimum (Mascheroni and Irrazábal, 2018).

The implementation of automated tests is often seen as way to save time and costs for the test execution. Most of the time people don't consider the effort of test maintenance when thinking about test automation. It is important to also consider the activities beside the actual execution of the tests, like maintenance or the analyzation of the test results (Oliinyk and Vasyl, 2019).

1.1 Motivation

When talking about test automation most of the people think about the automation of the test execution and are unaware about the possibility to automate further testing activities in the process. Beside the actual execution of test cases, the software testing process includes various other testing activities which should be considered for automation like the test planning or test reporting (Garousi and Elberzhager, 2017).

This thesis concentrates on the automation of manual testing activities along the software testing process beside the automation of the test execution. A special focus is on the test planning and administration of the test execution and reporting.

1.2 Problem Definition

The Alpega Group (<https://www.alpegagroup.com>) is one of the leading global providers of logistics software with over 30 years of experience. The cloud-based software solution digitizes complex supply chain processes and optimizes them by providing an end-to-end visibility of the transportation flows. It offers modular solutions that support the logistic processes of its customers. The product range includes transport management systems (TMS) as well as freight exchanges. With a community of more than 200,000 users, Alpega is active in 80 countries. The company employs more than 650 employees of 31 different nationalities (Alpega Group, 2022a).

This thesis focuses on the Alpega TMS branch, which is distributed to several offices around the globe, namely Austria, Germany, Belgium, France, Sweden, Chicago, and Thailand. The software development engineers responsible for Alpega TMS are located mainly in Austria and Thailand, which leads to challenges in their daily work due to the geographic distance as well as the cultural differences.

Within the current year, Alpega will shorten the release cycles from quarterly releases to monthly releases, which is the first step to build up a Continuous Integration / Continuous Delivery culture.

With each release a set of regression tests is executed to ensure a high quality for the customers. Consequently, the shorter release cycles lead to a higher effort for the Quality Assurance (QA) team. To address this challenge the manual effort for the software testing process should be reduced.

The whole process is supported by TestRail™, which is a tool for test management and test documentation. Each software application which is offered by Alpega is represented as an own project in TestRail, while the individual projects contain all existing test cases. For the documentation of the test results, TestRail™ offers the possibility to create test plans based on milestones.

As precondition for each release a milestone needs to be created per TestRail project. Based on the single milestones an own test plan is created per module and customer which is associated with a high manual effort. Each test plan must be created individually, and it must be ensured that each test plan contains the correct test cases. With over 40 test plans and more than 4,000 test cases there is a high risk that a test case is missing in one of the test plans, which in turn increases the risk for the release.

1.3 Research Objectives

The goal of this thesis is to analyze the possibilities to automate the software testing process as far as possible.

Quality Assurance is an important part in each release cycle. Even though the test automation coverage is high, there is a lot of effort to maintain and administrate those tests. When talking about test automation it is important to also consider the administration and maintenance related to those tests. Especially, when releases are getting more frequent it is important to substitute as many manual tasks as possible.

The optimization of the test management process will mainly result in the enhancement of the used tools including customized UI scripts and the use of existing APIs.

In the end the optimization of the test management process should lead to a

- reduction of the manual effort for the administration of test plans,
- reduction of the risk of incomplete test plans and
- increase of available QA resources for other tasks.

Table 1 shows a comparison between the actual and target conditions.

Table 1: Research objectives actual vs. target

	Actual	Target
Test plan creation	0% automated	100% automated
Incomplete test plans	5-10%	0%
Execution	Greenkeeper and Lychnobite automated Regression and Smoke manual setup and trigger	Greenkeeper and Lychnobite automated Regression and Smoke manual trigger but automated setup
Reporting	Manual definition of test plan id, after that automated	Automated determination of correct test plan
Key performance indicators (KPI)	Not present	Introduced to track progress

At the moment, for each release 92 test plans need to be created (see *Table 6*). The creation of those test cases is done manually and should be fully automated in future. The automation of the test plan creation should also eliminate the presence of incomplete test plans which is between 5 and 10 % looking at the releases from the past year.

Also, the execution of the test plans can be slightly improved. While the execution of the Greenkeepers and Lychnobites which are test plans that are executed on a regular basis on the test environment (see *Table 4*) is already fully automated, the execution of the regression and smoke test plans is still done manually. The regression test plans are executed before each release on a production clone to ensure that the new code does not have any side effects on the existing functionalities. The smoke test plans are executed directly after the release on production to ensure that the system is up again. The goal is that the test plan execution can be triggered manually, and the setup is done automatically.

The reporting of the test results happens automatically after the execution of each test case. Therefore, it is still necessary to manually define the test plan ID to map the correct test plan. This parameter needs to be adapted with each release as the test plans are release dependent. In future, the either the definition of the test plan ID parameter or the determination of the test plan itself should happen automated.

1.4 Research Question and Hypothesis

To what extend can the manual effort in the software testing process for a global logistics software provider be reduced by the automation of recurring administrative tasks to address the challenges of shorter release cycles?

With the first insights into the possibilities of the used tools in the software testing process of Alpega, I hypothesize that the administrative tasks related to test plan creation, execution, and reporting can be fully automated to serve as reliable quality gate for future releases. As a result, the overall manual effort will be reduced by 25%.

1.5 Research Design

A literature study should give a general overview about agile testing and testing in CI/CD. Furthermore, it should point out the challenges of short release cycles and best practices to address them. The literature review should ensure a common understanding of the theoretical background and point out the state-of-the-art (Fettke, 2006).

As a next step the software testing process at Alpega is described in detail and broken down to single testing activities. It should be determined which parts of the process are already automated and which parts are eligible for an automation.

The use of quantitative questionnaires can be used to gain information about the quantity of a variable. It is an easy and fast way to get the input of several people (Bhat, 2018). A quantitative questionnaire among the software quality managers at the company should help to estimate the current manual effort of the software testing process. Those estimates will be used to determine to which extend the software testing process has been improved after the optimization of the process. In the end, it should show how much manual effort could be saved by the automation of testing activities.

To be able to design a solution approach, the current process will be analyzed in detail to identify weak points and elaborate improvement opportunities. Based on the outcome a solution approach will be designed and implemented as a prototype.

By the use of prototyping, the validity of the designed solution approach should be proofed (Salzburg Research, 2023). By the actual implementation of the approach for one of the products at the company the manual effort after the optimization can be determined. Those numbers can be used to compare the manual effort before and after the optimization to point out to which extend the process has been improved regarding the manual effort.

2 Literature Review

This chapter summarizes the outcome of a detailed literature research in the field of software testing with a focus on agile software testing. It provides a theoretical foundation for the upcoming chapters.

Furthermore, best practices for TestRail – the test management tool used at Alpega – are reviewed.

2.1 Software Testing

Software testing is an important part of the software engineering process. The main goals of software testing are to ensure the customer requirements are fulfilled and to prevent that any defects are delivered to the customer. It should ensure that only high-quality software is delivered to satisfy the customer (Mousaei, 2020).

The traditional way of software development follows the waterfall model which is a plan driven methodology. Thereby, the process is divided into different phases which are conducted sequentially. *Figure 1* gives an overview of the waterfall model (Mohsin Nazir, 2020).

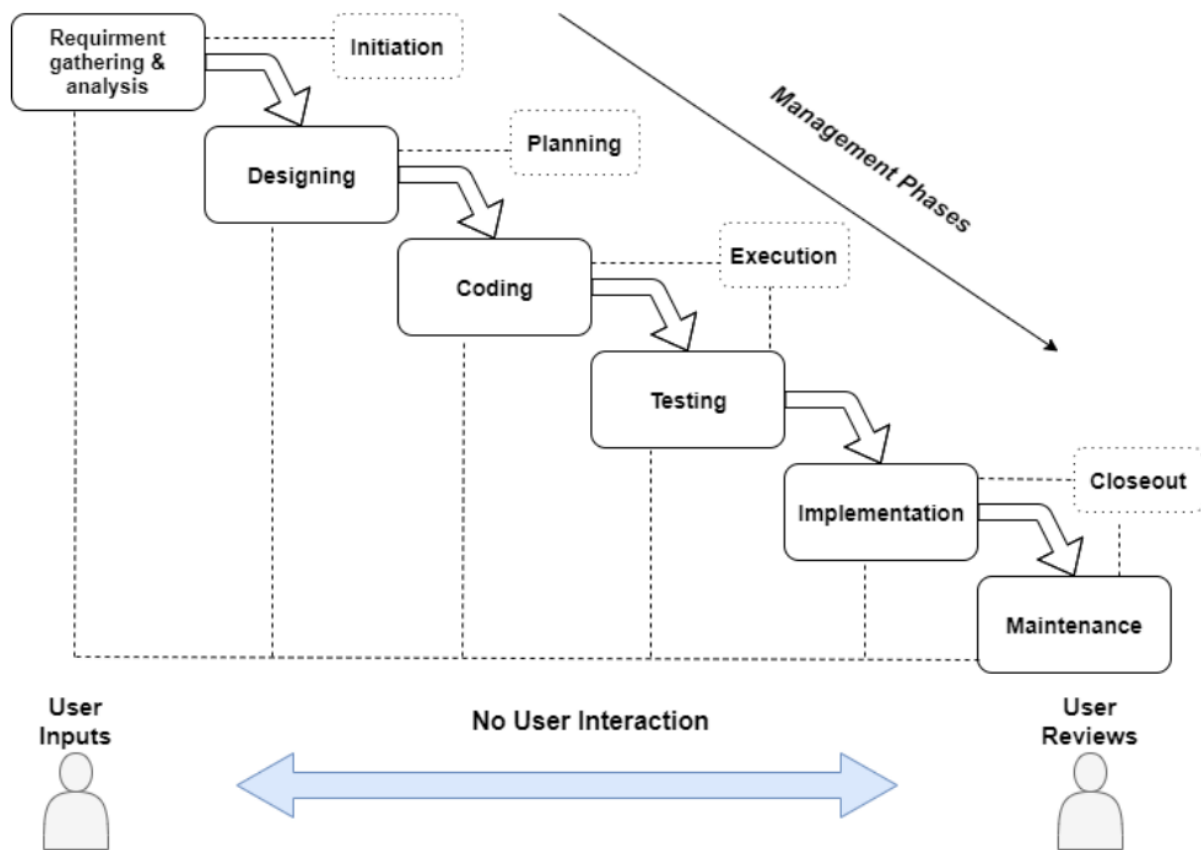


Figure 1: Waterfall model (Mohsin Nazir, 2020)

A common problem of the waterfall model is that the customers are not involved during the development process. The customer provides the input in the requirement stage which is the first stage of the model. Thereby, it is of high importance that all the requirements are defined in detail and fully understood as the whole model builds up on this information. During the upcoming phases there is no feedback from the customer, which brings the risk that the customer needs are not fully met (Mohsin Nazir, 2020).

Beside the missing feedback from the customer also the feedback from the software testers will happen on a late stage. Apart from unit testing or developer testing, the testing stage will only start after the coding has been completed, which results in a late detection of defects.

Especially in larger projects the missing continuous feedback can be a high risk for the success of the project. As alternative agile software development methodologies like Scrum can be used to increase flexibility and ensure continuous feedback.

2.1.1 Agile Software Testing

The goal of agile software development methodologies is to deliver new features in smaller pieces to the customer on a regular basis to get their feedback continuously. Also, software testing is shifted to the left which means the testing activities will start in an earlier stage and not after the coding has been done. Thereby, continuous feedback is provided, and defects are detected in an earlier stage, which can save a lot of time and costs (Alpega Group, 2022b; Mohsin Nazir, 2020).

Agile methodologies focus on flexibility and quality by creation products which meet the needs and expectations of the customer. Figure 2 below summarizes the agile values and principles (Mohsin Nazir, 2020).

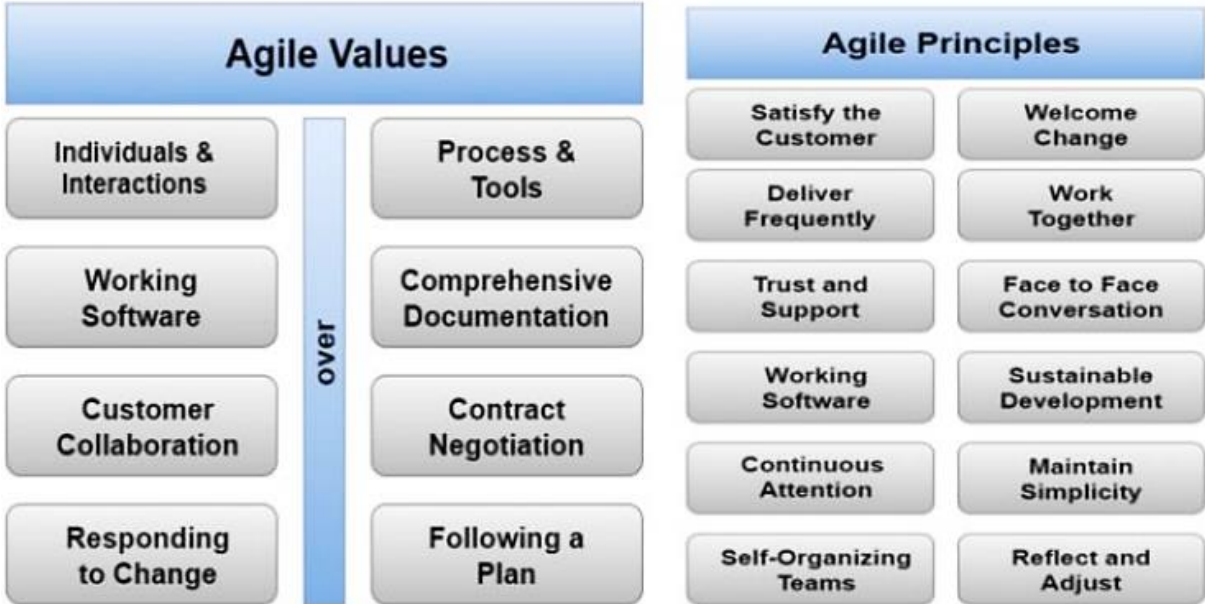


Figure 2: Agile values and principles (Mohsin Nazir, 2020)

There are four core values which are followed when using agile methodologies. The first value “Individuals & Interactions over Process & Tools” points out that the precondition is a well-working team that can communicate efficiently. The best processes and technologies are worth nothing without a strong and motivated team. The value “Working Software over Comprehensive Documentation” means that the focus is on the software itself instead of the documentation. The other two values “Customer Collaboration over Contract Negotiation” and “Responding to Change over Following a Plan” show that flexibility is an important factor in agile methodologies. It is important to collaborate with the customers and adjust the plan if necessary (Alpega Group, 2022b; Mohsin Nazir, 2020).

The twelve principles of the agile manifesto are based on the four agile values and describe the culture in an agile software development environment in more detail. Overall, it can be said that the goal is a flexible team with an efficient and strong collaboration that focuses on the satisfaction of the customers by considering their continuous feedback (Alpega Group, 2022b).

In agile models the developers and testers are working closely together and are part of the same team. Furthermore, there is a close collaboration with the customer to obtain feedback as soon as possible (Alpega Group, 2022b; Mohsin Nazir, 2020).

The following *Figure 3* shows how the mindset of the tester changes when switching from traditional to agile methodologies (Alpega Group, 2022b).

The goal of agile testing is to provide fast feedback in an early stage. Automated test should support to shorten the feedback loop. In agile testers and developers are in the same team and share the responsibility of delivering high quality software. Agile testers should support the team to become quality aware. As an agile tester an important goal is to step into the customers shoes and to ensure that the software meets the customer needs (Alpega Group, 2022b).

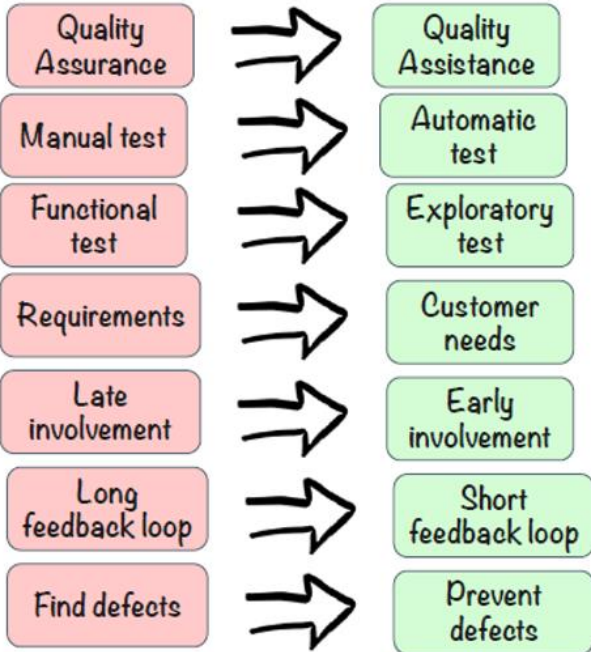


Figure 3: Agile tester mindset (Alpega Group, 2022b)

2.1.2 Software Testing in CI/CD

Continuous Integration / Continuous Delivery (CI/CD) is a concept to release the software into production quickly and frequently. The first part of the process is Continuous Integration which means that the developers implement small pieces of code and merge it to a shared version control repository. The process also includes the verification of the integrated code by automated tests to identify potential problems as fast as possible. The second part of CI/CD is Continuous Delivery which means that the completed code is automatically deployed to production (Elbaum et al., 2014; Mascheroni and Irrazábal, 2018).

Following *Figure 4* shows the different steps within the process of CI/CD, which are repeated in a loop.

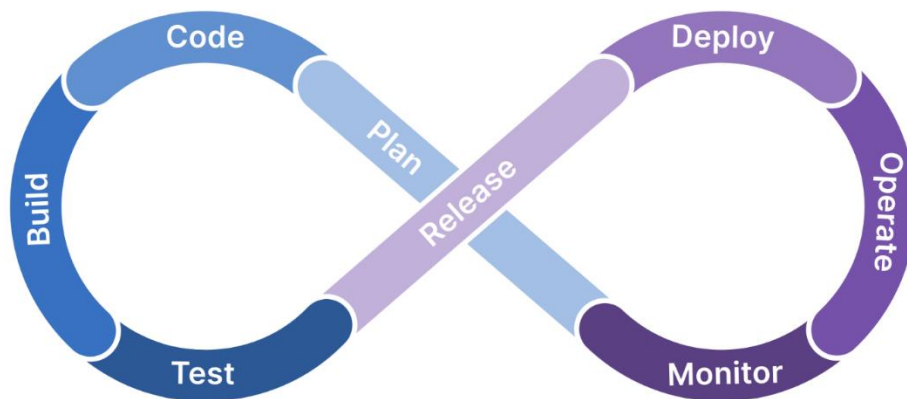


Figure 4: CI/CD process (Nahnsen, 2020)

Continuous Testing is an important part of the CI/CD concept and often one of the biggest challenges. The precondition for a working CI/CD pipeline is reliable automated testing process. Common problems are the time effort to conduct the testing and also unstable automated tests (Mascheroni and Irrazábal, 2018).

2.2 Software Test Management

“Test management is a method of planning, organizing, controlling, and ensuring traceability in the software testing process in order to deliver high-quality software applications.” (Ghani, 2021)

Every software development project consists of various testing activities which need to be managed. Software test management includes risk analysis, test estimation, test planning and test organization which can be grouped together to Planning, and test execution and monitoring, issues management and test reporting and evaluation which can be grouped together to Execution. It should ensure a smooth process of all testing activities during the software development life cycle. *Figure 5* gives an overview of the different software test management phases (Pawlak and Poniszewska-Marańda, 2018; Shannon, 2022).

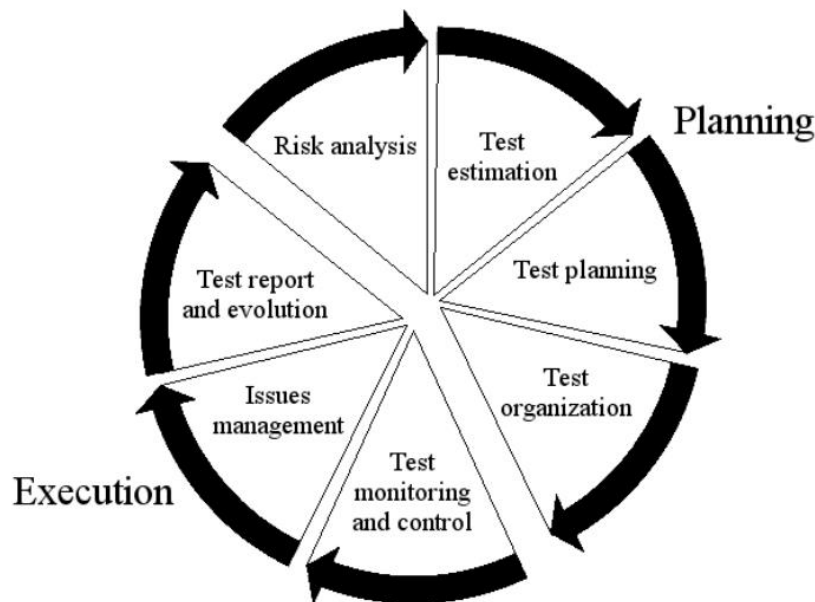


Figure 5: Software test management phases (Pawlak and Poniszewska-Marańda, 2018)

The usage of a software test management tool supports the whole process of software test management and provides the test manager as well as the rest of the team a big picture. It is the central tool for QA and plays a decisive role in their daily work. Therefore, the selection of the tool must be given careful consideration. There are plenty of tools available on the market and each tool has its individual benefits and limitations. In the end, it is important to ensure that the tool matches the requirements of the organization (Son, 2022).

The following list gives an overview of common features which should be supported by a test management tool (Munsamy, 2019; Son, 2022):

- Storage and organization of test cases
- Storage of test results
- Maintaining of history of test cases and results
- Generation of test reports
- Integration with other tools
- Creation of test plans
- Tracking of test progress

With the use of agile software development methodologies, the role of software test management is also changing. In agile, testing is not conducted at the end of the implementation phase, instead testing already starts at an early stage and follows an iterative approach. This also requires adaptations in test management, in agile the traditional software test management role doesn't exist anymore (Ghani, 2021; Greyling, 2022).

In agile the role of a test manager changes from a tactical to a more strategic role including following responsibilities (Ghani, 2021; Greyling, 2022):

- Develop an organisational test strategy
A main task of a test manager in agile is to establish an organisational test strategy and to continuously improve the testing process. There should be common guidelines which support the team to achieve high quality.
- Empower testers
The goal in agile is to empower the single employees and share the responsibilities among them. The role of the test manager is to provide guidelines rather than control the testers.
- Grow the skills of staff
The empowerment of the individual testers also leads to the need of new skills. Therefore, the development of skills is another important part in agile test management. The goal is to build a team with effective testing professionals which are self-sufficient and willing to take ownership.

- Build efficient teams

Another task of an agile test manager is to build up an efficient team based on his or her insights into business needs. The skills of the individual team members should cover all necessary needs.

In summary it can be said, that with the change to agile methodologies also the responsibility for quality changes. It moves away from dedicated software testers towards the whole team (Greyling, 2022).

2.3 Test Automation

The previous years have shown that the importance of test automation in the software development process is growing steadily. More and more companies are taking steps to increase the degree of automation. This also affects the role of software testers and requires new technical skills like programming (Oliinyk and Vasyl, 2019).

Most of the people are unaware about the diverse possibilities of test automation across the software testing process. Following two sub chapters are giving a short overview of test automation in a narrow and broader sense.

2.3.1 Test automation in a narrow sense

Test automation in a narrow sense is the automation of manual test cases. Automated testing will lead to a reduction of costs and time. Instead of a manual tester verifying the behavior of the software, test scripts are executed to ensure the software is working as expected (Garousi and Elberzhager, 2017).

When starting with the implementation of automated tests it is important to select the right test cases to get the most benefit out of it. Depending on the manual effort for the test execution as well as the effort to implement the automated tests, test cases need to be prioritized. Thereby, the most suitable test cases for automation can be identified (Sabev and Grigorova, 2015).

Good candidates for automated tests are ones which verify frequently used as well as critical functionalities with a high risk of errors. Also test cases which are testing background processes and are hard to reach manually are worth to automate. Another common use for test automation is load testing which is a kind of performance testing to determine the limits of the application (Oliinyk and Vasyl, 2019).

In the end a high coverage of automation frees up a lot of time for testers which allows them to focus on other topics which might not be able to automate. Another benefit is the faster feedback with the use of automated tests as the execution time is much shorter. Nevertheless, it is important to invest enough time in the initial implementation as well as the maintenance of automated tests. Only a reliable and stable test suite will bring the benefits mentioned above (Asfaw, 2015; Oliinyk and Vasyl, 2019).

While test automation leads to higher efforts due to the initial investments for the actual implementation of the test cases, it can save a lot of costs in long term. *Figure 6* shows the development of the costs for manual and automated testing over time. In the beginning the costs for automated testing are higher than for manual testing, but the costs are getting less over time while the costs for manual testing keep increasing more steeply (Oliinyk and Vasyl, 2019; Shah, 2019).

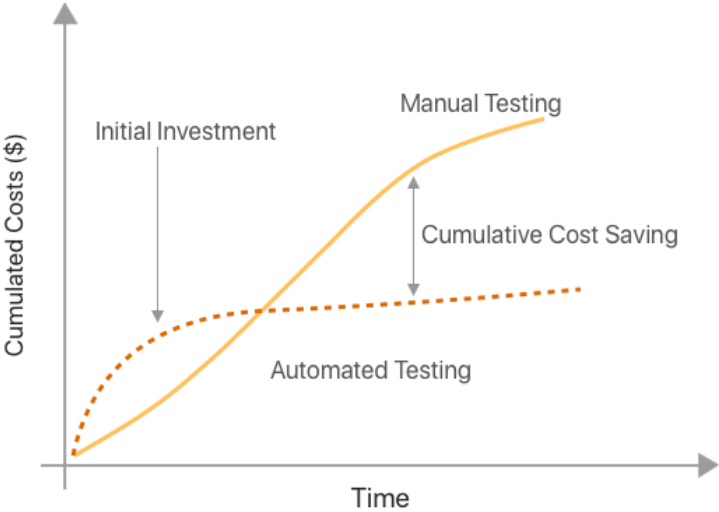


Figure 6: Costs of manual and automated testing (Shah, 2019)

In this context it is very important to note that also automated testing still leads to costs after the initial implementation as they need to be maintained continuously. This is the reason why the costs for the automated tests still decrease slightly after their initial implementation. The maintenance of the automated tests contains the analysis of the test results, the adaption in case of changes in the software and also the stabilization of the tests (Shah, 2019).

Test automation brings a lot of benefits, but it also has its limitations. Testing is more than only following a set of predefined steps. Test automation can never replace the testers intelligence and the specific knowledge about the application under test. Furthermore,

there are cases where the test automation would lead to more effort than manual testing. Therefore, it is important to combine the benefits of both manual and automated testing to ensure a high quality of the product (Oliinyk and Vasyl, 2019; Sabev and Grigorova, 2015).

2.3.2 Test automation in a broader sense

When talking about test automation people are often only thinking about the automation of the test execution. The focus is to automate manual test cases to be able to run them on a regular basis without the need of software tester. Beside the automation of the test execution, automation can be expanded to any other phase across the software testing process such as test planning, test reporting and test management. Even though the coverage of automated tests is high, there are various other activities which cause a lot of manual effort that should be taken into consideration (Garousi and Elberzhager, 2017; Loke Mun Sei, 2015).

Same as for the automation of manual test cases, the automation in other phases of the software testing process leads to higher effort and costs in the beginning but will lead to time and cost savings in a long term. Furthermore, it can be expected that processes are getting more stable and reliable (Garousi and Elberzhager, 2017; Loke Mun Sei, 2015).

2.4 Best Practices with TestRail

At Alpega TestRail is used as a test management tool to support the software testing process. The bachelor thesis “TestRail as a central information point” (Thoma, 2021) points out the advantages of the tool and describes it as following:

“TestRail™ is a comprehensive test case management tool. It allows you to organize your test cases and track the test results. Furthermore, it provides extensive reporting functions including real-time insights. TestRail is a complete web-based tool and has a simple user interface, which makes it easy to use.”

The thesis also shows the various possibilities to extend the functionalities of the tool using UI scripts and the provided APIs of TestRail. As outcome of the work, TestRail has been made to the central information point for QA of Alpega by the integration of all tools used by QA. The following chapter *3.1.2 Test documentation and reporting* shows how the single tools are connected to each other.

Meanwhile, also TestRail™ itself has published some blog post to aware about the possibility of triggering test automation jobs from TestRail and sending the results back via API. A dedicated blog series gives detailed instructions on how to streamline test automation with TestRail by using Jenkins (Rede, 2022).

Figure 7 gives an overview of how the tools are integrated with each other. It can be seen that also this integration uses UI Scripts on TestRail side to trigger the tests and the TestRail API to send back the results.

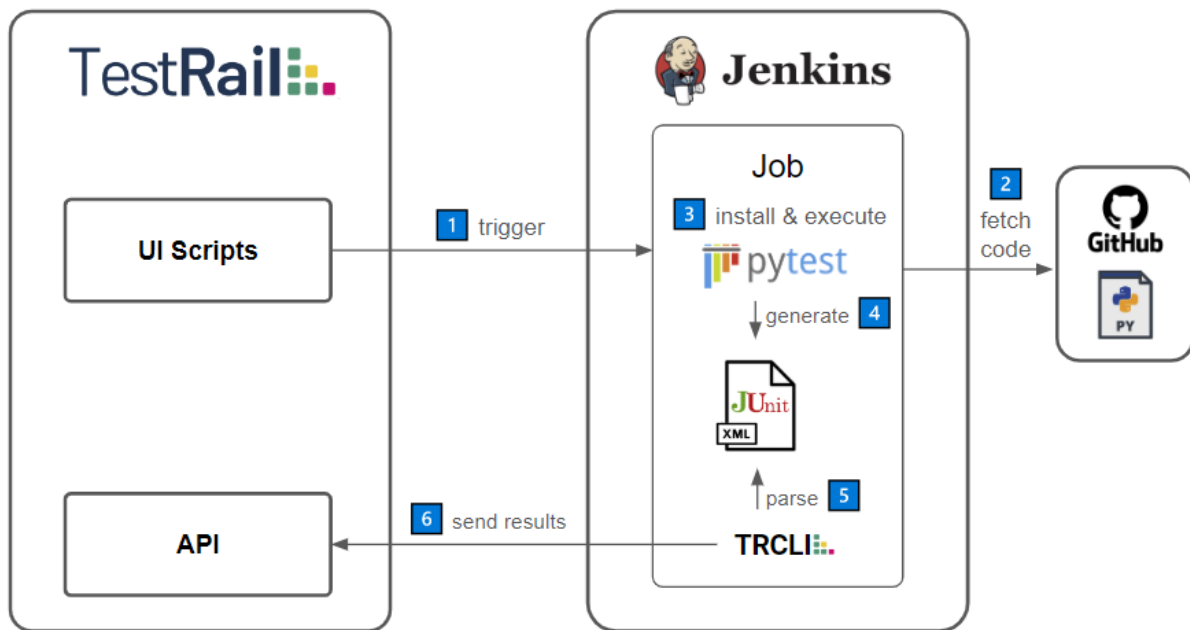


Figure 7: Overview TestRail integration with Jenkins (Rede, 2022)

Another blog post “Integrate Test Automation Results with TestRail – TestNG“ on SW Test Academy (Akduygu, 2019) shows how to create a test run at the beginning of the test execution and send the test results after their execution by using TestNG.

There are plenty of other sources with instructions on how to integrate the test automation framework with TestRail™. Overall, the implementation always focuses on

- triggering the test execution and
- sending the results back to TestRail

by using UI scripts and the TestRail APIs.

Those enhancements are leading to a big improvement in the software testing process and saves a lot of time during the execution and reporting of tests (Thoma, 2021).

Nevertheless, there would be other parts of the software testing process beside the test execution which would be worth to automate (see 2.3.2 *Test automation in a broader sense*). The review of current work shows that the automation of administrative tasks is not covered by now – even though TestRail™ offers various possibilities with its built-in integrations, UI scripts and APIs.

3 Review of Current Process

The following chapter should give an overview of the current state at Alpega. First, the software testing strategy is described. This includes the used tools as well as the company internal conventions for test documentation, reporting, automation, and execution. Furthermore, the software testing process which is conducted for each release is described in detail to point out its weak points and potential improvement opportunities.

3.1 Software Testing Strategy

At Alpega, the development process is supported by the agile methodology Scrum. Based on this framework a set of company internal principles, which are described in *Table 2* below, have been defined to be followed.

For interested readers, Atlassian provides a comprehensive guideline to the Scrum methodology (Atlassian, 2023).

Table 2: Principles of agile testing (Alpega Group, 2023b)

Principle	Description
<i>Continuous testing</i>	Testing is performed regularly and is done in conjunction with development.
<i>Short feedback loops</i>	Regular feedback is provided to achieve the goal of high quality.
<i>Immediate fixing of bugs</i>	All the defects which are raised are fixed within the same iteration.
<i>High level of automation</i>	Automation is part of each feature from the beginning.
<i>Less documentation</i>	Instead of comprehensive test documentation, the focus is on the test.
<i>Test driven</i>	Testing starts at the time of implementation not after it.

Traditionally, testing was a separate activity that came after the coding phase. In agile, testing is continuous, putting testers between product owners and developers. This creates an ongoing feedback loop which helps to improve the code.

- **Communication with product owners**

Testers interact with product owners to clearly establish project expectations and to satisfy customer needs.

- **Close interaction with developers**

Testing is linked to the development process. Testers are part of the development team, they report on quality issues that can affect users, and suggest how to improve the solution.

To ensure that testing is conducted within the implementation phase it is part of the “Definition of Done” which is of central importance within the agile methodology Scrum. The “Definition of Done” defines the requirements which need to be fulfilled that a story can be classified as done. This ensures a common understanding about the term “Done”.

At Alpega following rules listed in *Table 3* below have been defined as “Definition of Done” to ensure common standards and a high quality of the products.

Table 3: Definition of Done

Rule	Explanation
“A Sprint Backlog Item is Done, when...”	
... it is implemented and meets all functional and non-functional requirements.	The question if all requirements are met needs to be answered by the Product Owner or knowledgeable Stakeholders.
... it has been tested by the person who implemented it.	The person implementing the item must perform informal tests (developer testing).
... comprehensive unit tests are written, and the implementation passes them, unless it was agreed upon that unit tests are not needed.	Usually, we require unit tests to be written that comprehensively test the implementation. Under certain circumstances, the team can decide that unit tests are not needed. This must be an explicit decision that must be captured inside the Jira-Ticket and any meeting notes published. If no decision is made, unit tests are considered mandatory.
... it is reviewed by a team member other from the person who implemented it	Once the implementation is completed, the item must be reviewed by someone else in the team.
... internal documentation is created or updated.	By "Internal Documentation" we mean any form of documentation available to members of the Alpega Engineering organisation. This does not include customer-facing documentation like handbooks, presentations, or other informational material. Depending on documentation guidelines, both inline comments and other documentation must be updated.

<p>... changes are merged and deployed to test environments and proper configuration is done.</p>	<p>All changes to code (if there are any) are merged into the correct branch and deployed to test environments to allow stable and reproducible testing procedures.</p>
<p>... manual testing confirms the implementation meets the acceptance criteria and is error-free.</p>	<p>All items must be tested to make sure the implementation does what it is supposed to do and is error-free. The minimum requirement is that the manual test must be done by a person who is different from the one implementing the item, and the one reviewing the item.</p>
<p>... automated tests are written, and the code passes them, unless it was agreed that automated tests are not needed.</p>	<p>The default case is that automated tests are written to cover the new implemented functionalities. Writing automated tests is costly and might not be effective for every item completed during the Sprint. Instead of writing automated tests for every new development, we instead should focus on mission- and business-critical cases. Whether an automated test should be written must be decided during Refinement, or latest during the Sprint Planning, with heavy input from the Product Owner. It must be explicitly decided that no automated tests are needed, and this decision must be documented in the Jira-ticket and any meeting notes.</p>

The software testing process is supported by various tools which are partly integrated with each other. Furthermore, a set of conventions is defined to ensure smooth processes and a high quality of the provided software solutions.

3.1.1 Terminology

The following *Table 4* contains company specific terms which are relevant for the upcoming chapters of the thesis.

Table 4: Company specific terminology

<i>Test Suite</i>	A test suite is a collection of test cases which cover a specific functionality.
<i>Test Plan</i>	A test plan contains a set of test cases which cover a specific test scope (i.e., Regression). It is built based on test suites. A test result is always linked to a specific test plan.
<i>Lychnobite</i>	Lychnobite is a term used for test plans which are executed more than once a week on the test environment.
<i>Greenkeeper</i>	Greenkeeper is a term used for test plans which are executed once a week on the test environment.
<i>Test environment</i>	The test environment (Alpha) is used to test features as soon development is done. It points to the master branch.
<i>Stage environment</i>	The stage environment (Integration) is used during regression phase and to test Hotfixes. It points to the release branch.

3.1.2 Test documentation and reporting

The test documentation and reporting are done in TestRail (<https://www.gurock.com/testrail/>), which is a comprehensive test management tool to organize test cases and track their results. All test cases and test results are documented in TestRail which improves the visibility of the test coverage. Furthermore, the tool provides extensive reporting functionalities which increases the transparency.

TestRail™ supports the daily work of the Software Quality Engineers at Alpega and is used as a central information point. The integration with other tools like Jira, GitLab and the test automation frameworks simplifies the whole testing process. The tooling landscape in *Figure 8* visualizes how the single tools are integrated with each other.

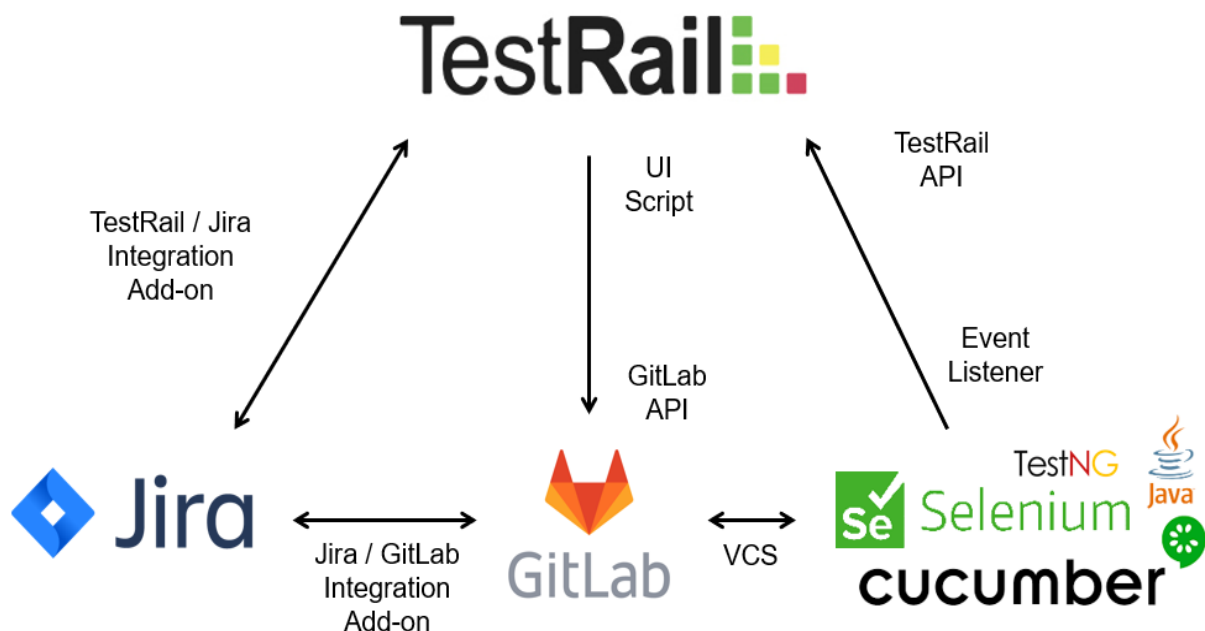


Figure 8: Tooling landscape (Thoma, 2021)

The integration of the tools enables following possibilities:

- **TestRail – Jira**
 - Add Jira tickets as references to test cases in TestRail (automatically creates a link to Jira)
 - Add Jira tickets as defects to test results in TestRail (automatically creates a link to Jira)

- **TestRail – GitLab**
 - Start selected automated tests directly from TestRail by triggering a pipeline in GitLab
 - on specific instance
 - on specific GitLab branch

- **TestRail – Test automation framework**
 - Send results from automated tests to TestRail to get a continuous reporting
 - including user, instance and
 - in case of failure also error message, link to error log and link to screenshot
 - Update content (title and steps) of test cases in TestRail to ensure consistency

3.1.3 Test automation

Alpega follows an agile testing strategy which means testing is part of the development process. As described in the beginning of this chapter the manual testing as well as automated testing is part of the “Definition of Done”. This underlines the importance of testing in the development process.

By default, the implementation of automated tests starts at the time of implementation of the actual code. There are only exceptional cases where no automated test is written, and this decision needs to be documented in detail.

The writing of automated tests should also not be deferred to a later point in time. Either an automated test is needed, then it needs to be written during the same Sprint, or the decision is taken that no automated test is needed, and it will not be written at all. It should not happen to create a separate backlog item in a Test Automation Backlog.

3.1.4 Test execution and maintenance

All automated test cases are executed on a regular basis. The execution is triggered by schedules defined in GitLab, and the results are sent to the corresponding test plans in TestRail. *Figure 9* shows the overview page of the GitLab schedules for one of the products provided by Alpega. The column “Target” shows on which branch the pipeline will be executed. The column “Last Pipeline” provides a link to the latest pipeline where the single jobs of the pipeline can be accessed. The column “Next Run” specifies when the next

pipeline will be triggered and also gives the information in case a schedule is inactive. The owner of the pipeline is the person who is notified in case a scheduled pipeline fails.

Based on the description of the schedule it can be noted for which customer and how often the schedule is executed. There are schedules for the standard solutions as well as for customer specific solutions. As described in *Table 4*, Lychnobites are executed several times per week and Greenkeepers are executed only once a week.

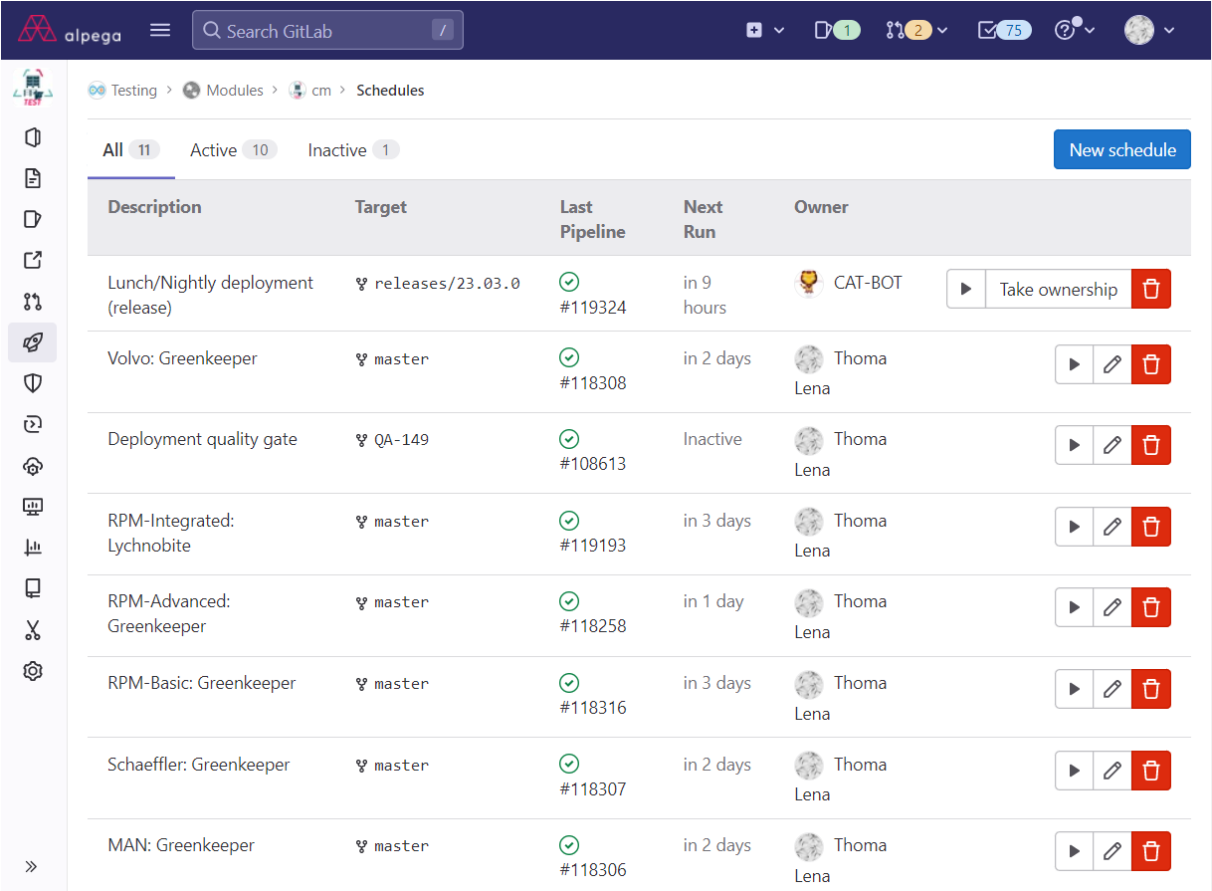


Figure 9: GitLab schedules

Figure 11 shows the schedule editor where the details of the schedule can be configured. The frequency and the exact execution time of the automated test can be defined in the field “Interval pattern” which is using a specific Cron syntax. The definition of the syntax is shown in Figure 10 below.

```
# _____ minute (0 - 59)
# _____ hour (0 - 23)
# _____ day of the month (1 - 31)
# _____ month (1 - 12)
# _____ day of the week (0 - 6) (Sunday to Saturday)
#
#
#
# * * * * * <command to execute>
```

Figure 10: Cron syntax (GitLab, 2023)

When defining the execution times of the individual schedules it is important that the executions are distributed as only a limited amount of GitLab runners is available to run the automated tests. Thereby, it is also important to consider the time zone which can also be defined for each schedule individually.

Furthermore, the target branch can be defined. By default, the schedules will be triggered on the master branch of the project. Beside that any other active branch of the project can be defined for the execution of the scheduled pipelines.

The use of variables provides the possibility to define the customer, TestRail test plan ID, test instance and execution set of the schedule. Those variables define which test cases are executed on which instance and for which customer within the pipeline that is triggered by the schedule. Furthermore, the TestRail test plan ID defines to which test plan the results of the execution should be sent to.

At the bottom of the schedule editor, the schedule can be activated or deactivated as needed.

inet Menu

Testing > Modules > cm > Schedules > #186

Edit Pipeline Schedule

Description

RPM-Integrated: Lychnobite

Interval Pattern

Every day (at 0:00am)
 Every week (Friday at 0:00am)
 Every month (Day 0 at 0:00am)
 Custom (Cron syntax)

0 2 * * 1,4

Cron Timezone

Vienna

Target branch or tag

master

Variables

Variable rpm-integrated

Variable alpha

Variable 24676

Variable regression

Variable Input

Hide values

Activated

Active

Save pipeline schedule Cancel

Figure 11: GitLab schedule configuration

Next to the regular execution of the test cases, the regular maintenance of the test results is of high importance. Thereby it is ensured that possible bugs are detected, and unstable test cases are identified.

In case of a bug a corresponding bug ticket is created in Jira and linked to the test result in TestRail to track the progress. Bugs which are detected by automated tests have high priority and are fixed within the current sprint.

In case of an unstable test a software quality engineer needs to improve the test code to increase the stability. The stability of the automated tests is a key factor for the quality. It reduces the maintenance time in future and is a precondition for the future goal of CI/CD. Only a set of stable automated tests can be used as reliable quality gate.

3.2 Release Process

The Release Process is composed of various phases and milestones which are described in *Table 5* below.

The Release Process includes the three phases “Build Features”, “Regression” and “Hotfix Window”, while the phase “Build Feature” is always in parallel to the phase “Regression” and “Hotfix Window” of the previous release.

With the “Scope Freeze” the “Regression” phase of the current release as well as the phase “Build Features” of the next release starts. The “Regression” phase ends with the “Golden Build” and is followed by the “Go-Live”. After that the “Hotfix Window” is open until the “Scope Freeze” of the next release.

Table 5: Release process - Phases and milestones

Phase/Milestone	Start	End	Definition
<i>Build Features</i>	Scope Freeze (previous release)	Scope Freeze	Features are continuously built, released, and tested on the Test environment following the agile methodology Scrum.
<i>Stage Clone</i>	-	-	Stage environment is prepared for the upcoming Regression Phase. It includes loading recent production data, and some post-processing like the execution of after clone scripts to ensure that the test data is in the expected state.
<i>Scope Freeze</i>	-	-	Upon scope freeze milestone (EOD), the master branch must be in a proper state for the release branch to be created in the following working hours (only fully tested features are allowed in the release branch).

			<p>No more features shall be added beyond this milestone.</p> <p>As outcome, the first release build is created and promoted to Stage environment within the next working day.</p>
<i>Regression</i>	Scope Freeze	Golden Build	<p>Planned regression tests are executed on Stage environment.</p> <p>Identified regression bugs are assessed for inclusion in the release based on a risk/value assessment.</p> <p>Authorized bugfixes are continuously promoted to Stage environment until Golden Build.</p>
<i>Golden Build</i>	-	-	<p>Golden Build must be delivered by every team, as outcome of regression phase, several days prior to Go-Live.</p> <p>Only fully tested bugfixes are allowed in the golden build. Unfinished bugfixes must be re-scheduled for hotfix or next release.</p> <p>No more code change shall be merged in release branch beyond this milestone.</p>
<i>Go-Live</i>	-	-	<p>With the Go-Live the new software version is deployed on Production. To ensure that the deployment has been done successfully a small set of smoke tests is executed on the production environment.</p>
<i>Hotfix Window</i>	Go-Live	Scope Freeze (next release)	<p>This is the time period where Stage environment is available for validation of hotfixes.</p>

Following *Figure 12* visualizes the release process at Alpega including the different phases and milestones. It can be seen that the testing during the phase “Build Features” is conducted on the Test environment while the testing during the “Regression” and “Hotfix Window” phase is conducted on the Stage environment. On the Test environment always uses the latest release version. The Stage environment which is a clone from the production has always the release version of the production environment.

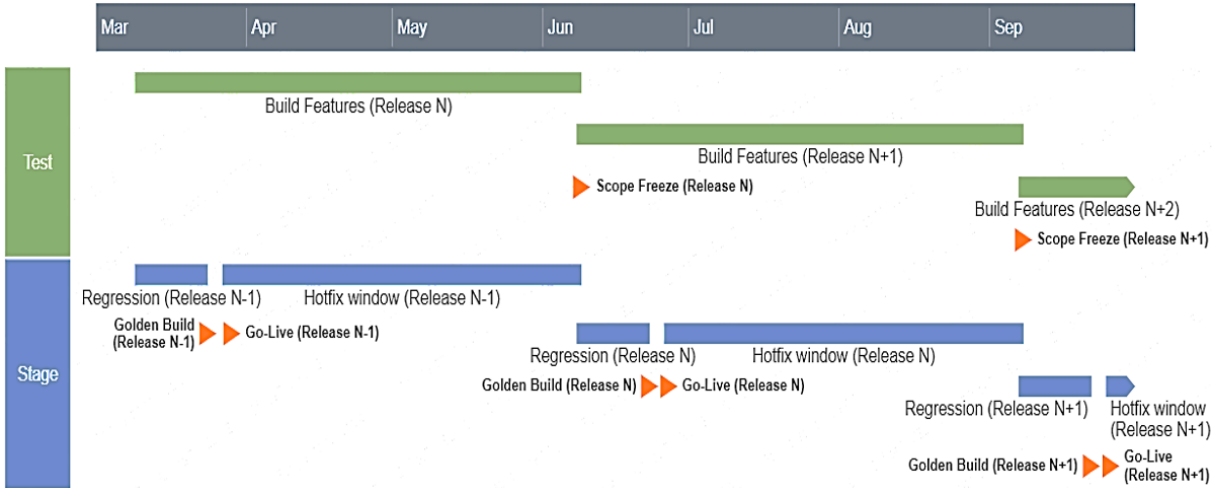


Figure 12: Release process (own illustration)

In the past there have been four releases within each year. The future goal at Alpega is to establish a Continuous Integration / Continuous Delivery culture. As intermediate step the release cycles should be shortened to monthly releases.

3.2.1 Shorter release cycles

The change towards monthly releases should serve as preparation for the establishment of a Continuous Integration / Continuous Delivery culture. The processes should be more and more automated and the manual effort in all departments, inter alia Quality Assurance, should be reduced to a minimum. Thereby the process should get less complex and in consequence also less risky.

With the shorter release cycles following advantages can be expected:

- The amount of newly delivered code decreases and thereby also the risk of each release is smaller than in the past.
- The time between two releases shrinks and thereby also the number of hotfixes will be smaller in future. With the shorter period it is easier to deploy fixes with the regular release and avoid the deployment of a hotfix which poses an additional risk and effort. Thereby, hotfixes are getting plannable deployment tasks.
- New features are delivered faster to the customer and therefore the feedback loop is getting shorter and can be considered at an earlier stage.

3.2.2 CI/CD

Building on the knowledge gained by the change towards shorter release cycles, the next step will be the establishment of CI/CD.

Following vision has been defined in the organization which should be reached with the introduction of a CI/CD culture.

“Every commit goes live through the delivery pipeline after having passed the quality gates. Releasing software should be so easy and automated that it becomes a non-event.” (Alpega Group, 2023a)

The successful implementation of a CI/CD pipeline will strengthen the benefits achieved with shorter release cycles even more. The frequent delivery of new features will significantly reduce the complexity and risk for the company as well as for the customers.

The establishment of CI/CD requires a fully automated delivery pipeline including stable automated tests to verify the quality of the delivered code. The change towards shorter release cycles and the optimization of the software testing process by automating manual activities is therefore an essential step to get closer to the main target of the organization, namely building up a CI/CD culture.

3.3 Software Testing Process

Depending on the phase of the release, different testing activities are relevant. The following subchapter 3.3.1 *Workflow* gives an overview of the respective tasks during the different phases described above in chapter 3.2 *Release Process*. Based on this summary the weak points of this process are identified to be able to define corresponding improvement opportunities.

3.3.1 Workflow

Each release consists of various testing activities depending on the current release phase which are described below. *Figure 13* gives an overview of the activities for each phase.

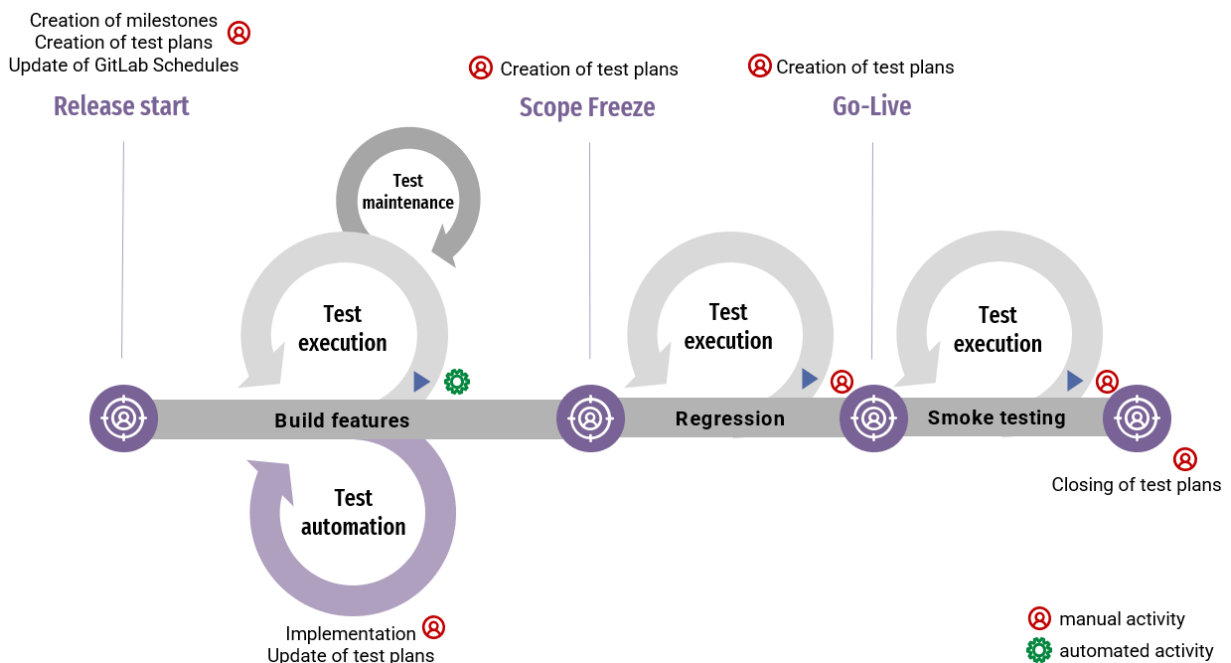


Figure 13: Software testing activities within the different release phases (own illustration)

For each phase there are preparation tasks as well as tasks which are done during the individual phases. In *Figure 13* each task is marked as either manual or automated activity. It can be seen that all tasks except the continuous test execution during the phase “Build features” are not automated yet. Goal of the optimization is to increase the degree of automation.

Below, the single activities are described in more detail.

Preparation for phase “Build features”

- *Creation of milestones*

For each release, an own milestone including the sub milestones for the scope freeze (beginning of regression phase) and go-live (day of smoke tests) needs to be created. This needs to be done for each TestRail project individually.

- *Creation of test plans (Greenkeepers and Lychnobites)*

The results of the continuous nightly and weekly test executions are stored in individual test plans in TestRail. The test plans are always valid for one release and are closed after that. This enables the possibility to track the results and related defects of each release separately.

- *Update of GitLab schedules*

The test results of the nightly and weekly test executions are automatically sent to TestRail. Therefore, the test plan ID needs to be defined as variable in the GitLab schedules. As the test plan ID changes with the creation of the new test plans, this also needs to be updated for each release.

Tasks during phase “Build features”

- *Automation of new test cases*

With the development of new features also new test cases are implemented which need to be added in TestRail and assigned to the corresponding test plans. This ensures that the new test cases are considered for the nightly and weekly test runs.

- *Continuous execution of automated tests incl. test maintenance*

The automated tests are executed on a regular basis based on the schedules defined in GitLab. The results which are automatically sent to TestRail need to be analysed by QA regularly.

Preparation for phase “Regression”

- *Creation of test plans (Regression)*

For every regression phase a test plan for each individual customer is created where the results of the automated and manual tests are stored. Based on the test plans the progress of the regression testing is monitored.

Tasks during phase “Regression”

- *Triggering of automated regression tests*

The execution of the automated tests needs to be triggered manually by setting up a corresponding schedule in GitLab or start them locally. The results are sent automatically to the individual test plans in TestRail.

- *Execution of manual regression tests*

In addition to the automated tests, there is a small set of manual regression tests which needs to be executed. The results of those tests cases are documented in the individual test plans in TestRail.

Preparation for “Go-Live”

- *Creation of test plan “Smoke”*

For the Go-Live a specific set of smoke tests is executed on the Production environment. A dedicated test plan is prepared for the day of the release.

Tasks for “Go-Live”

- *Triggering of automated smoke tests*

For the execution of the automated smoke tests a corresponding schedule in GitLab needs to be created (same as for the regression tests). The results are sent automatically to the test plan in TestRail.

- *Execution of manual smoke tests*

A small amount of smoke tests is executed manually. Those results are also stored in the corresponding test plan in TestRail.

3.3.2 Effort of Recurring Testing Activities

With the goal of shorter release cycles, especially the activities which must be performed for each single release need to be reduced to a minimum. As the manual effort strongly depends on the number of test plans which need to be handled, *Table 6* contains an overview of the number of test plans per project. In total there are 92 test plans which are created for each release.

Table 6: Number of test plans per project

Project	Lychnobite / Greenkeeper	Regression	Smoke
<i>Reusable Packaging Management (RPM)</i>	8	10	1
<i>Master data Management (MM)</i>	7	7	1
<i>Transportation Management (TM)</i>	18	25	1
<i>Integration Server (IS)</i>	2	2	0
<i>Freight cost Management (FM)</i>	3	6	1
<i>Total</i>	38	50	4

Following *Table 7* gives an overview of the recurring manual testing activities with a rough estimation regarding their effort. The overview only contains tasks which are related to the test administration, therefore the effort for the execution of the manual test cases is not in scope.

The estimations are based on a survey which has been conducted among all software testers working at Alpega. For the calculation of the effort the average of all estimations is used.

Table 7: Effort of recurring testing activities

Manual effort	Per year with quarterly releases	Per year with monthly releases
<i>Creation of milestones for all projects (RPM, MM, TM, IS, FM)</i>		
10 min per project 5 * 10 min = 50 min	200 min = 3 h 20 min	600 min = 10 h
<i>Creation of test plans (Lychnobites, Greenkeepers, Regression, Smoke)</i>		
38 Lychnobite/Greenkeeper * 20 min = 760 min 50 Regression * 20 min = 1,000 min 4 Smoke * 10 min = 40 min Total → 1,800 min	7,200 min = 120 h	21,600 min = 360 h
<i>Update of GitLab schedules (Lychnobites, Greenkeepers)</i>		
38 * 5 min = 190 min	760 min = 12 h 40 min	2,280 min = 38 h
<i>Update of existing test plans with new test cases (Lychnobites, Greenkeepers)</i>		
5 * 5 min = 25 min / week	1,300 min = 21 h 40 min	1,300 min = 21 h 40 min
<i>Triggering of automated test execution (Regression, Smoke)</i>		
(50 + 4) * 5 min = 270 min	1,080 min = 18 h	3,240 min = 54 h
SUM	175 h 40 min	483 h 40 min

3.3.3 Weak Points

In the workflow described in the previous sub chapter several weak points can be identified which are listed below.

- **Separate creation of milestones for each project**

The milestones for a release are the same for all products. Nevertheless, they need to be created for each project individually as TestRail does not provide the option to create the same milestone for multiple projects.

- **Manual test plan creation (Lychnobites, Greenkeepers, Regression, Smoke)**

The creation of the test plans needs to be done for each project and each customer individually. As the content of the test plan changes from release to release due to the implementation of new test cases, it is not possible to just copy the old ones. Beside the manual time effort, there is also the risk of incomplete test plans, as it might happen that test cases are forgotten to be added to the test plan.

- **Manual update of GitLab schedules**

For the automatic transmission of the test results to TestRail, the test plan ID needs to be defined in the individual GitLab schedules. As this ID changes with each release when creating new test plans, all GitLab schedules need to be updated manually.

- **Updating of test plans with new test cases**

In case new test cases are implemented they need to be assigned to the relevant test plans, which is an additional manual effort. There is the risk that a test case is not assigned to a test plan, which would mean the result is not visible and a possible bug would not be recognized.

- **Trigger of automated tests for Regression and Smoke tests**

For the execution of the automated tests a schedule needs to be set up for each test plan individually. As the test plan ID, which is needed for the transmission of the test results, changes with each release, it is not possible to just reuse the schedules of the previous release.

3.3.4 Improvement Opportunities

The workflow described above should be optimized especially by the automation of manual tasks. Following opportunities have been determined to eliminate the weak points identified and thereby improve the software testing process which is conducted for each release.

- **Central creation of milestones**

Through a central creation of the milestones, it should be avoided that the same task needs to be repeated for each of the projects. The goal is to provide the option to create the release milestones for multiple projects in one single step.

- **Automated test plan creation based on test case attributes**

Each test case is classified by a set of attributes (e.g., test run type or customer) which can also be customized. Based on those attributes and a defined set of rules, the test plans should be created automatically.

- **Ad hoc determination of test plan for transmission of test results**

The automated tests are executed in GitLab schedules based on variables which define the customer, the environment and the execution set which should be executed. Instead of defining the test plan ID as own variable, the test plan should be determined by the other variables. Thereby, it can be avoided that the test plan ID must be updated with each release.

- **Automatic assignment of new test cases to relevant test plans**

In case a test is executed but not part of the test plan, it should be determined based on test case attributes if the test case is relevant for the test plan. Depending on that the test case is added to the test plan or not and it is ensured that no test results get lost.

- **Option to trigger test execution directly from TestRail**

There should be the option to trigger the execution of the test cases directly from TestRail without the usage of any other tool. The trigger should happen within a specific test plan to be able to send back the results accordingly.

4 Solution Approach

The goal of this thesis is to optimize the software test management process to be able to address the challenges of shorter release cycles. Based on the analysis of the previous chapters and the detected improvement opportunities a solution approach has been designed which is the foundation for the implementation of the proof of concept.

The focus is to automate tasks related to the test administration which would multiply with the increase of the releases.

4.1 Overview

This sub chapter should give an overview on how the different tools interact with each other. *Figure 14* visualizes a tooling landscape of all involved tools, their responsibilities as well as the planned interactions between them.

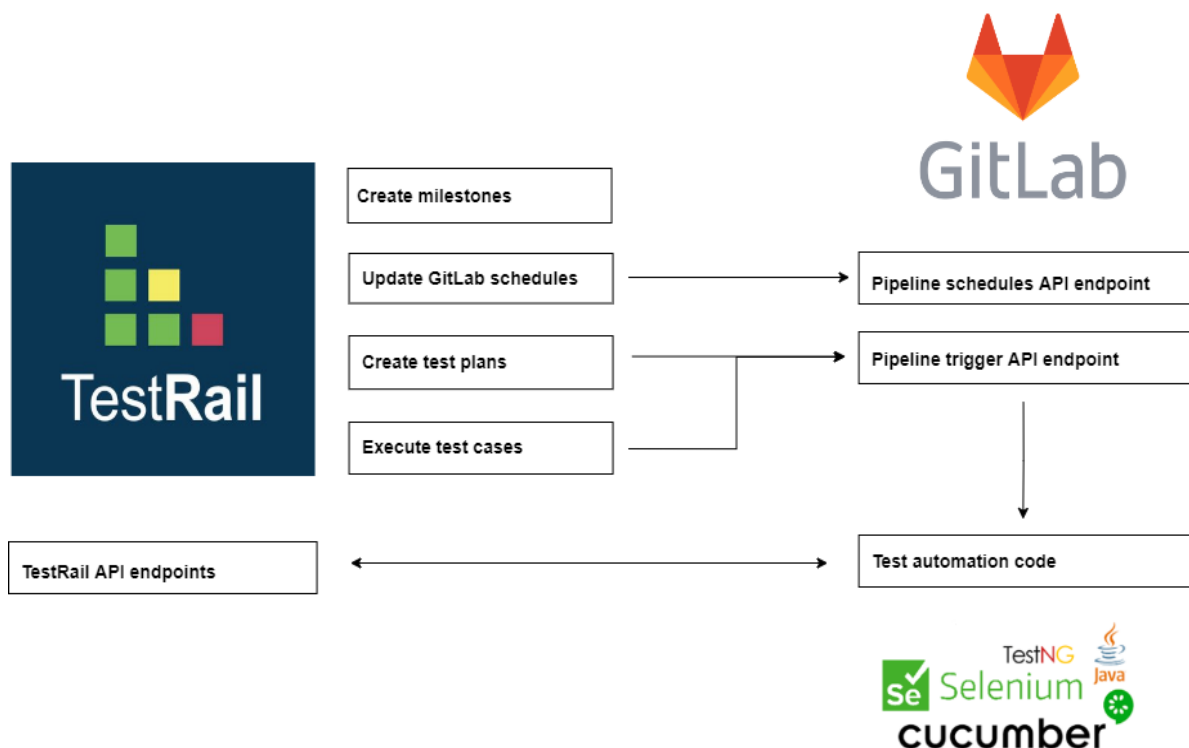


Figure 14: Overview tool interaction (own illustration)

The starting point of each process and the single point of truth is TestRail. From there individual processes are triggered by API calls to GitLab, to be more precise, by triggering pipelines in GitLab and providing the necessary information within GitLab variables. Based on the provided variables the code is executed within GitLab jobs and further communication between the test automation code and TestRail will happen.

There are several places where GitLab variables can be defined. *Table 8* gives an overview of the different options ordered by their priority (from highest to lowest) (Thoma, 2021).

Table 8: GitLab CI/CD custom environment variables (Thoma, 2021)

Level of Definition	Description
<i>Schedule variables defined in the UI</i>	When configuring a scheduled pipeline, the user also has the possibility to define variables for this specific schedule. Variables on this level will overrule the variables described below.
<i>Project variables defined in the UI</i>	In the project's Settings > CI/CD custom environment variables can be added. Those variables are available for the whole project and are the default values for the continuous pipeline.
<i>Global variables defined in YAML file</i>	Under <code>variables:</code> the user has the possibility to define custom environment variables as variable/value pairs. Those variables are valid for all jobs within the YAML file. Example: <pre>variables: TEST: "HELLO WORLD"</pre>
<i>Job variables defined in YAML file</i>	Inside a job there is again the possibility to define environment variables. Variables defined on this level are only valid for a single job. Example: <pre>job_name: variables: TEST: "HELLO WORLD"</pre>

4.2 Basic Release Workflow in TestRail

Following sub chapters describe the targeted future release process from testing perspective. All activities which are needed during each release should be automated as much as possible.

4.2.1 Create milestones for release

Instead of creating each single milestone for the individual projects manually in the UI, a new UI script should be implemented to provide the possibility to create the release relevant milestones for multiple projects in one step.

Following information must be provided by the user:

- Release version (format XX.XX, e.g., 22.07)
- Date of release start
- Date of scope freeze
- Date of release
- Projects for which the milestones should be created

Based on the entered data three milestones are created for each selected project:

1. Parent milestone “Release {Release version}”
 - a. Start date: {Date of release start}
 - b. End date: {Date of release}
2. Sub milestone “Regression {Release version}”
 - a. Start date: {Date of scope freeze}
 - b. End date: {Date of release} - 1
3. Sub milestone “Smoke {Release version}”
 - a. Start date: {Date of release}
 - b. End date: {Date of release}

For the milestone creation the TestRail API endpoint “add_milestone” is used.

```
POST index.php?/api/v2/add_milestone/{project_id}
```


4.2.2 Start milestone “Release”

With the beginning of each release the test plans of the previous release are closed, and new test plans are created. To ensure that the test results of the nightly and weekly runs are sent to the correct test plans, the GitLab schedules need to be updated.

To avoid the situation that the test plan ID for each schedule needs to be maintained the test plan ID should be determined during test execution. The determination is done based on the customer, scope, and release version. The details of the test plan determination are described in chapter *4.5 Test Plan Determination*.

With the new way of determining the correct test plan only the release version needs to be updated for each release. With the start of the milestone “Release” the release version of the GitLab project needs to be updated by an API call to GitLab.

```
PUT /projects/:id/variables/:key
```

This ensures that the correct test plan is used for the upcoming test executions.

4.2.3 Start milestone "Regression"

During regression, a set of test cases is executed for each customer. To track the progress a test plan for each customer is used to document the results. With the start of the milestone “Regression” the test plan creation should be triggered to automatically create the necessary test plans. Thereby, the user has the option to either create a test plan for each customer or only for specific ones.

The test plan creation happens by triggering a GitLab pipeline via API with the relevant parameters.

```
POST /projects/:id/pipeline
```

The details about the automatic test plan creation are described in chapter *4.4 Test Plan Creation*.

As soon as the test plans have been created and the test environment is ready the test cases can be started directly from TestRail instead of setting up the test execution manually. This additional functionality is described in chapter *4.3.1 Execution of all test cases within a test plan*.

4.2.4 Start milestone "Smoke"

When starting the milestone “Smoke” the same functionality should be provided as for the milestone “Regression”. The difference when creating the test plans is the scope (see details in chapter *4.4 Test Plan Creation*).

On release day the test execution can again be started directly from TestRail with the additional functionality described in chapter *4.3.1 Execution of all test cases within a test plan*.

4.2.5 Close test plans of previous Release

At the end of each release phase the old test plans need to be completed and closed. As it might happen that there are some open things to discuss within individual test plans, the closing of test plans will not be automated for now.

4.3 Additional Functionalities

Beside the basic process for each release there are additional activities which should be supported by TestRail to reduce the manual effort.

4.3.1 Execution of all test cases within a test plan

So far, the test execution was started directly in GitLab. Therefore, an own schedule has been created to be able to trigger a pipeline with the relevant parameters (e.g., customer, instance, test plan ID, etc.). In future it should be possible to start the execution directly from TestRail by triggering a GitLab pipeline via API.

```
POST /projects/:id/pipeline
```

Basically, the same parameters which have been defined manually in the past, should be sent to GitLab automatically based on the information which is available in the test plan. Additionally, the user has the possibility to select a specific instance the tests should be executed on.

The following *Table 9* gives an overview of the values which have been defined manually in the past and should be sent as API parameter in the future.

Table 9: API parameters for test execution

API parameter	Value from TestRail
Customer	Customer can be determined from test plan name, which needs to fulfil the naming convention [Customer]: [Scope] [Release version]
Scope	Scope can be determined from test plan name, which needs to fulfil the naming convention [Customer]: [Scope] [Release version]
Release version	Release version can be determined from test plan name, which needs to fulfil the naming convention [Customer]: [Scope] [Release version]
Instance	The instance is optional. Either the user selects a specific instance, or the instance is determined based on the scope.
Branch	The branch is optional. Per default the test cases are executed on the master branch, but the user has the possibility to select a specific branch if needed.

After the user has triggered the test execution a link to the pipeline in GitLab should be provided to offer the possibility to follow the status of the pipeline.

With the future functionality to assign new test cases automatically to relevant test plans, it is also ensured that the test plan is updated during the execution of the test cases. This might happen if new test cases have been implemented between the creation of the test plan and the execution of the test cases. More details are described in chapter 4.6 *Automatic Assignment of new Test Cases to Test Plans*.

4.3.2 Execution of failed test cases within a test plan

Beside the execution of the whole test plan, there should also be the option to restart only failed test cases. Normally, each single failed test case should be analysed individually and based on the outcome the decision can be taken if it should be either restarted or a ticket needs to be reported. This can either be the case when a bug has been identified or when the test case is unstable.

In exceptional cases it is justified to restart all failed test cases in one step without a detailed analysis of each individual test result, those are:

- Test instance was in maintenance mode and therefore not available.
- Master data was configured wrong.

4.3.3 Test plan creation independent of milestone start

Usually, the test plans are created when starting one of the milestones “Regression” or “Smoke”. In addition, there should be the option to create test plans independent of those milestones. Therefore, a separate wizard should be available in TestRail where the necessary information can be provided by the user. While the scope is already predefined when starting a milestone, the user needs to define the scope manually when creating a test plan independent of a milestone. Additionally, the release version and the customers need to be entered by the user. For each user an own test plan is created and based on the defined scope the relevant test cases are added.

The test plan creation happens – same as when starting one of the milestones “Regression” or “Smoke” by triggering a GitLab pipeline via API with the relevant parameters.

```
POST /projects/:id/pipeline
```

The details about the automatic test plan creation are described in the following chapter *4.4 Test Plan Creation*.

4.4 Test Plan Creation

The automated test plan creation should support the creation of recurring test plans like test plans for regression or the nightly test runs. The test plans are created based on a set of attributes depending on the scope for which they are created (see *4.4.1 Definition of Scope*). The automated creation of test plans should also reduce the risk of incomplete test plans which might happen during the manual creation. Furthermore, it ensures consistency between the different projects.

4.4.1 Definition of Scope

The scope defines under which circumstances the test cases are executed. Based on the scope the corresponding test cases are added to a test plan and the instance for the test execution is selected. Following *Table 10* gives an overview of the different scopes and the relevant parameters.

Table 10: Definition of Scope

Scope	Test run type	Execution type	Instance
<p><i>Default</i> Execution when pushing new test automation code to avoid side effects on existing code</p>	Smoke	Automated	Test
<p><i>Greenkeeper / Lychnobite</i> Weekly and nightly test execution</p>	Regression	Automated	Test
<p><i>Regression</i> Regression testing between Scope freeze and Release</p>	Regression	Automated + Manual	Stage
<p><i>Smoke</i> Smoke testing on day of Release</p>	Smoke	Automated + Manual	Production

4.4.2 Mapping of TestRail and GitLab projects

Each project in TestRail as well as the different projects in GitLab have their individual IDs. To ensure that the correct projects are used for the transfer of data between TestRail and GitLab a mapping of the projects is needed.

Following *Table 11* gives an overview about all TestRail projects and the corresponding GitLab projects. It can also be seen that some GitLab projects (i.e., API and TM) belong to more than one TestRail project. For the projects RPM Mobile and iTrace there is no automation in place and therefore no mapping is needed.

Table 11: Mapping between TestRail and GitLab projects

TestRail	GitLab
<i>RPM</i>	<ul style="list-style-type: none">• RPM• API
MM	<ul style="list-style-type: none">• MM
TM	<ul style="list-style-type: none">• TM• API• EDI
<i>FM</i>	<ul style="list-style-type: none">• TM• API
<i>IS</i>	<ul style="list-style-type: none">• EDI• IS
<i>RPM Mobile</i>	no automation
<i>iTrace</i>	no automation

4.4.3 Relevant GitLab parameters

Following *Table 12* gives an overview of all GitLab parameters which are relevant for the automatic test plan creation.

Table 12: GitLab variables

Variable	Description
<i>CUSTOMER</i>	The customer for which the tests are executed. Value: Must match the customer defined in TestRail, (i.e., RPM-Integrated)
<i>SCOPE</i>	The scope of the test run. Defines which test cases are included in the Test plan and which instance is used for execution. Value: See <i>4.4.1 Definition of Scope</i> (i.e., Regression)
<i>REL_VERSION</i>	The release version for which the tests are executed. Value: XX.XX (i.e., 22.03)
<i>TESTRAIL_PROJECT_ID</i>	The ID of the TestRail Project. The ID is needed to determine the Test plan based on the plan name. Value: TestRail project ID; (i.e., 11 for RPM)
<i>TESTRAIL_PLAN_CREATION</i>	In case this parameter is set the job only verifies if the corresponding test plans exist and creates the missing ones. Value: true/false

<p><i>TESTRAIL_PLAN_ID</i></p>	<p>Test plan ID can be defined manually (existing implementation), if Test plan can be found determination by name is skipped.</p> <p>Value: Test plan ID from TestRail</p>
<p><i>INSTANCE</i></p>	<p>Instance on which tests should be executed. If it is not defined it will be defined based on SCOPE.</p> <p>Value: any instance (i.e., alpha, integration)</p>
<p><i>EXECUTION_SET</i></p>	<p>Set of test cases which should be executed. If it is not defined it will be defined based on SCOPE.</p> <p>Value: "and" + tag which should be executed (i.e., "and @smoke")</p>

4.5 Test Plan Determination

In general, when executing a set of test cases, the results are sent automatically to a corresponding test plan in TestRail. So far, the test plan ID had to be defined and updated with each release.

To simplify this process, the test plan ID should be determined automatically based on the scope, customer, and release version. As the test plans follow the naming convention [Customer]: [Scope] [Release version], the test plan can be determined by filtering through all open test plans by test plan name. The open test plans can be requested with an API call to TestRail.

```
POST /api/v2/get_plans/%s&is_completed=0
```

4.6 Automatic Assignment of new Test Cases to Test Plans

To be able to store a test result a test case must be part of the test plan. Basically, every test case which is implemented after a test plan has been created, needs to be assigned to all relevant test plans. In case this step is not done, the API call to store the test result of the new test case will not be successful.

In future this manual step should be avoided. Instead of ignoring the failed API call, it should be checked if the test case is relevant for the test plan. In case the test case is relevant, it should be added to the test plan and the result should be sent again. Thereby, it can be ensured that the test plans are kept up to date.

5 Implementation of Prototype

This chapter summarizes all details regarding the implementation of the prototype based on the solution designed in the previous chapter *4 Solution Approach*. The sub chapters are following the same structure as in the previous chapter so that they can be easily linked with each other.

5.1 Overview

Following *Figure 15* gives an overview of the implementation of the prototype. It shows how the different tools are connected to each other.



Figure 15: Implementation details based on solution approach (own illustration)

5.2 Basic Release Workflow in TestRail

The focus of following sub chapters is on the automation of the activities within the basic release workflow.

5.2.1 Create milestones for release

The milestones are the same for each single project in TestRail. Nevertheless, they need to be created for each project individually. This means for all 7 projects which are available in TestRail a release milestone and two sub milestones for Regression and Smoke need to be created.

With a new UI Script a central milestone creation has been introduced. By a click on the new button “Create Milestones” which is available on the TestRail dashboard a wizard will be opened. Within the wizard the user can define the release version and all relevant dates as well as the projects for which the milestones should be created. Thereby, the user has the possibility to create all necessary milestones for a release for multiple projects within one step.

Figure 16 shows how the wizard looks like after providing the required data.

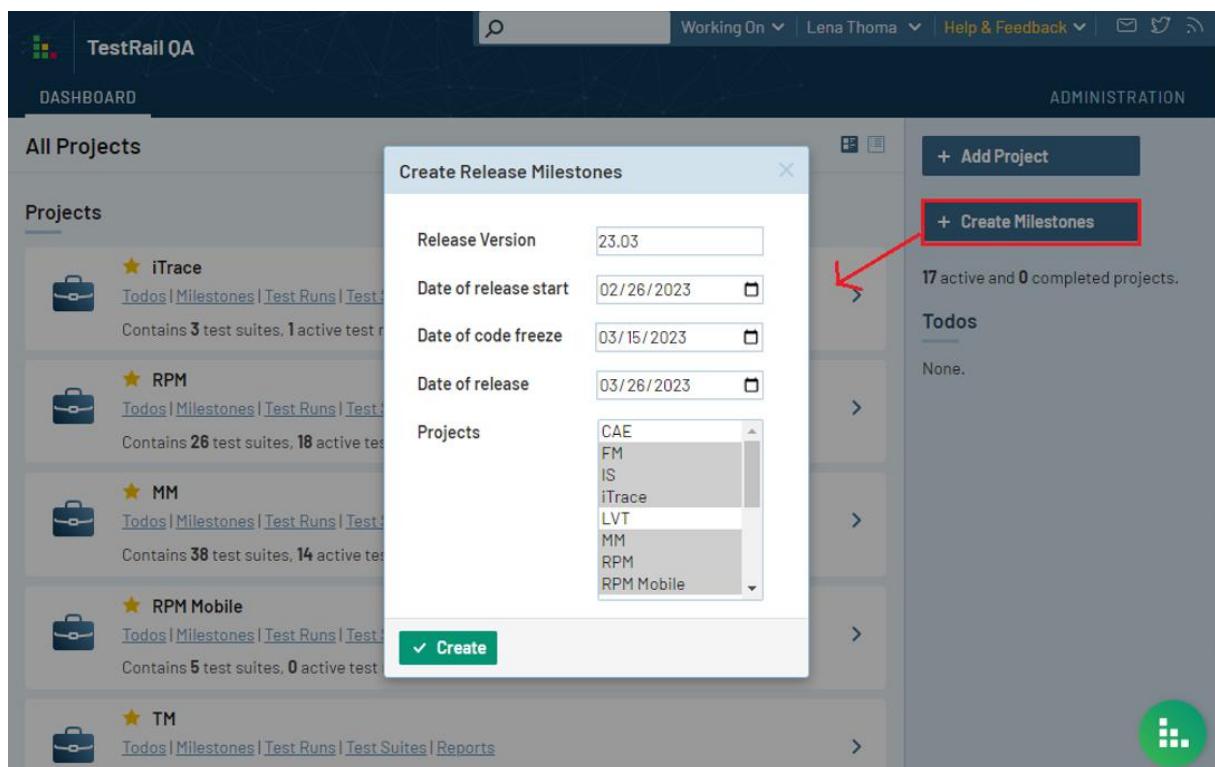


Figure 16: Central milestone creation via TestRail dashboard

After a click on the “Create” button, the names as well as the start and end dates of the single milestones are determined. The start and end dates are calculated based on guidelines define in chapter 4.2.1 *Create milestones for release*.

For the entered data this results in following:

1. Parent milestone “Release 23.03”
 - a. Start date: 26.02.2023
 - b. End date: 26.03.2023

2. Sub milestone “Regression 23.03”
 - a. Start date: 15.03.2023
 - b. End date: 25.03.2023

3. Sub milestone “Smoke 23.03”
 - a. Start date: 26.03.2023
 - b. End date: 26.03.2023

As suggested in the solution approach the milestones are created by a call to the TestRail API endpoint “add_milestone” for each of the selected projects.

```
POST index.php?/api/v2/add_milestone/{project_id}
```

After that the user will get the confirmation of the milestone creation including the selected projects as shown in *Figure 17*.

Figure 18 shows the release milestone including the sub milestones Regression and Smoke for the RPM project.

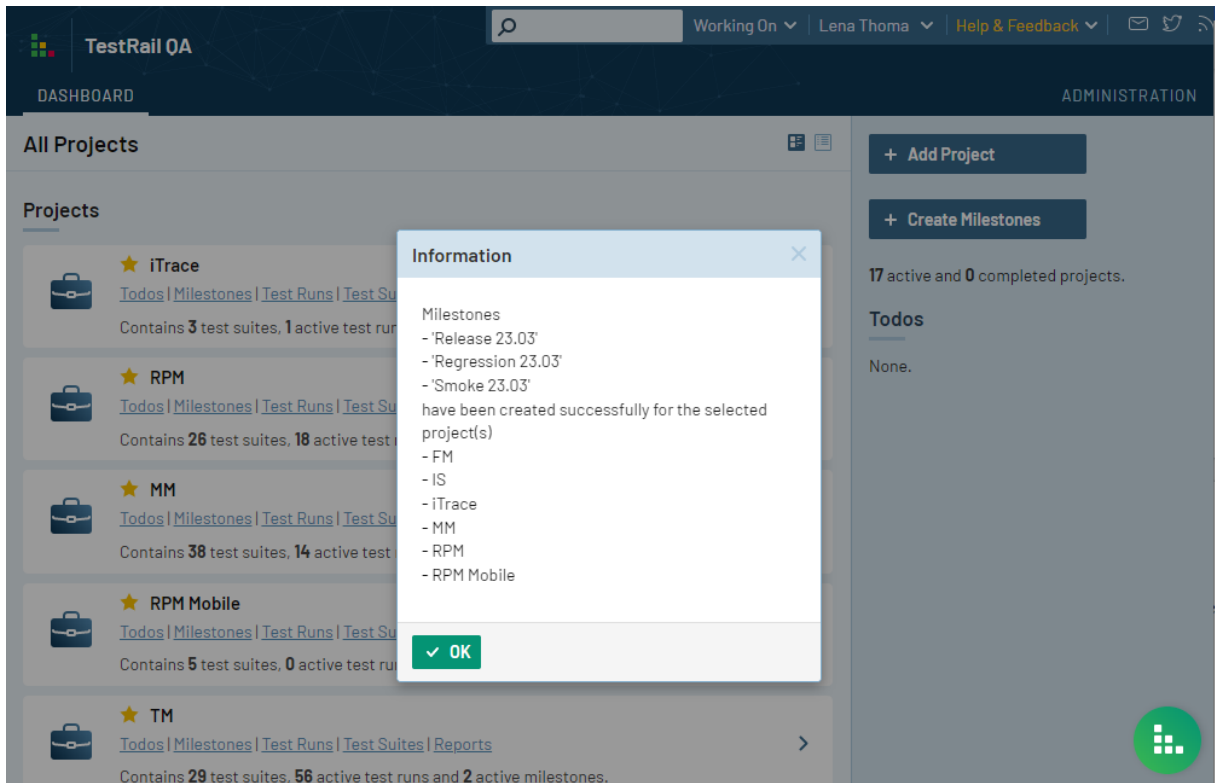


Figure 17: Confirmation of milestone creation

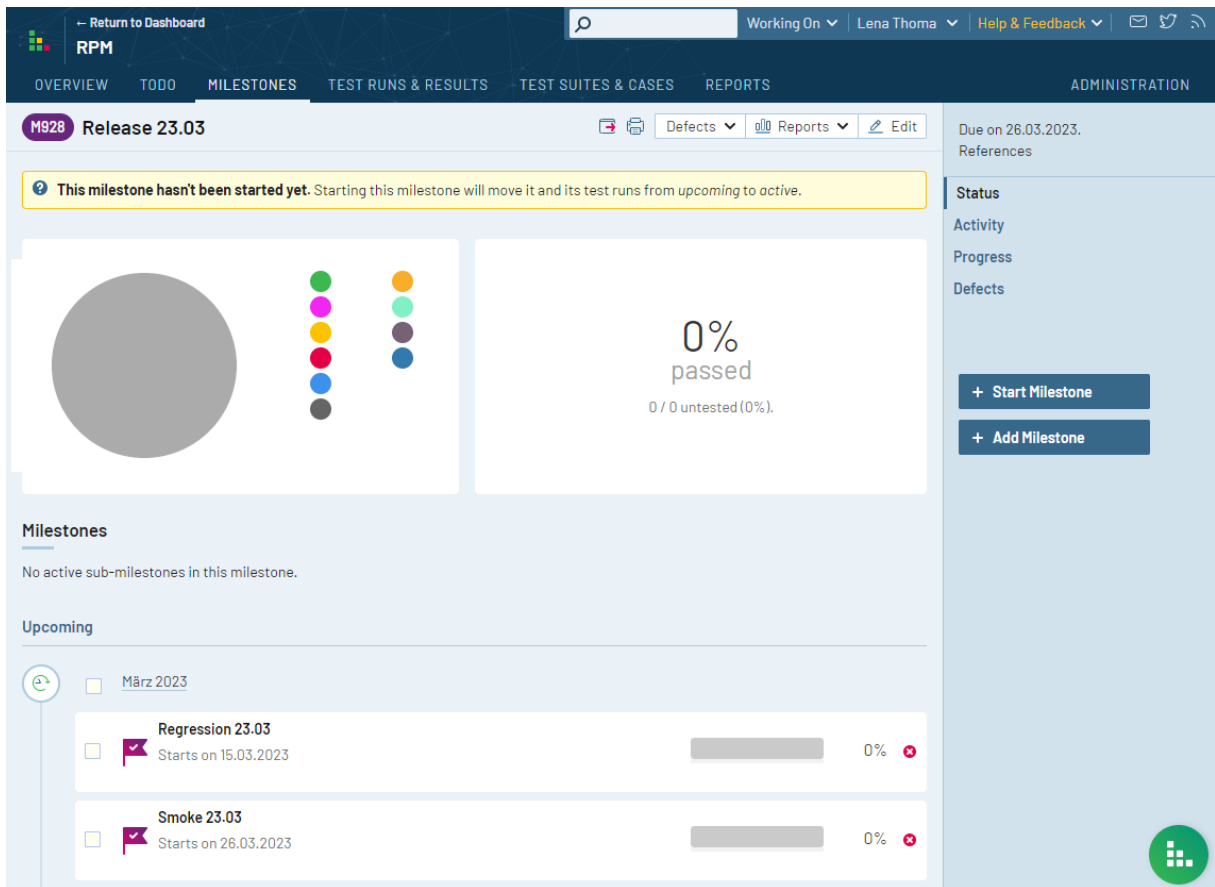


Figure 18: Release milestones created via central milestone creation

5.2.2 Start milestone “Release”

Another UI script has been implemented to update the GitLab variable “REL_VERSION” with the start of a new release milestone. This variable is needed on GitLab side to create or determine the test plans during the execution of the schedules (Lychnobites and Greenkeepers). In the build stage of each pipeline, it is ensured that the test plan for the customer exists for the current release to be able to send the test results. Thereby, the GitLab variable “REL_VERSION” defines the current release version. More details about how the test plans are created and determined can be found in chapter *5.4 Test Plan Creation* and *5.5 Test Plan Determination*.

The milestone can either be started on the overview page of all open and completed milestones with a click on the “Start” button next to the milestone name, which is shown in *Figure 19*, or on the right bar when opening the milestone detail page with a click on the button “Start Milestone”, which is shown in *Figure 20*.

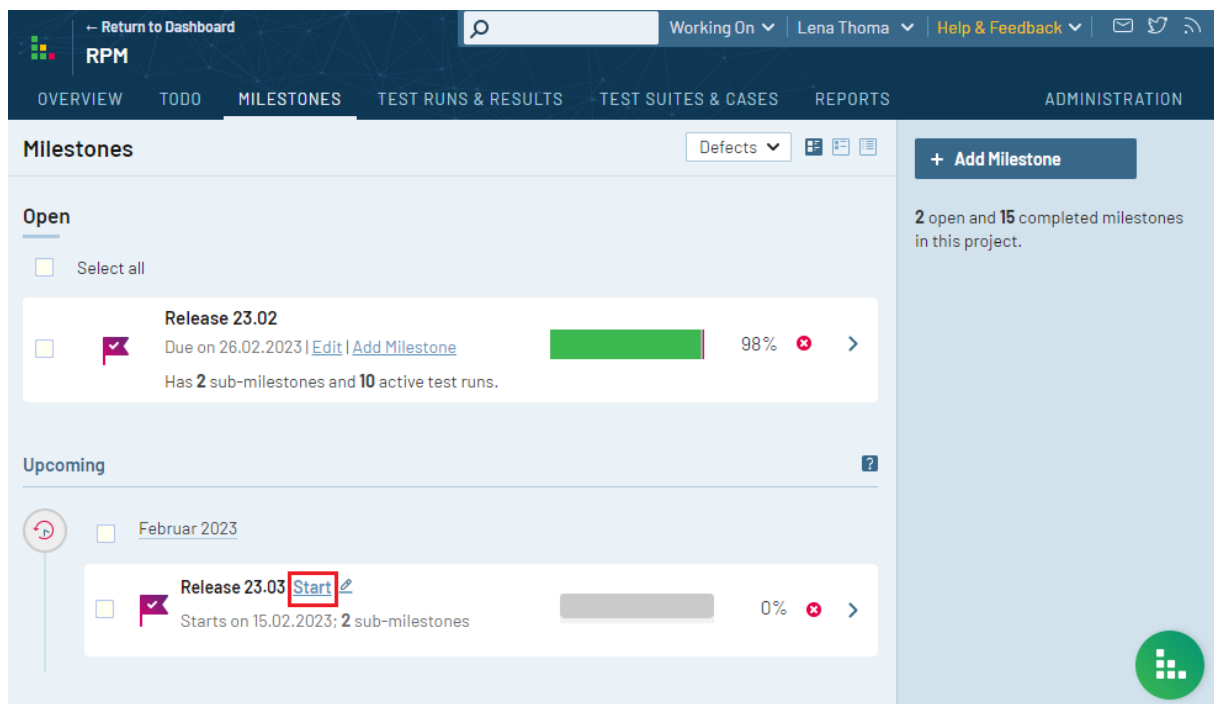


Figure 19: Start of release milestone to trigger update of GitLab variable (milestone overview page)

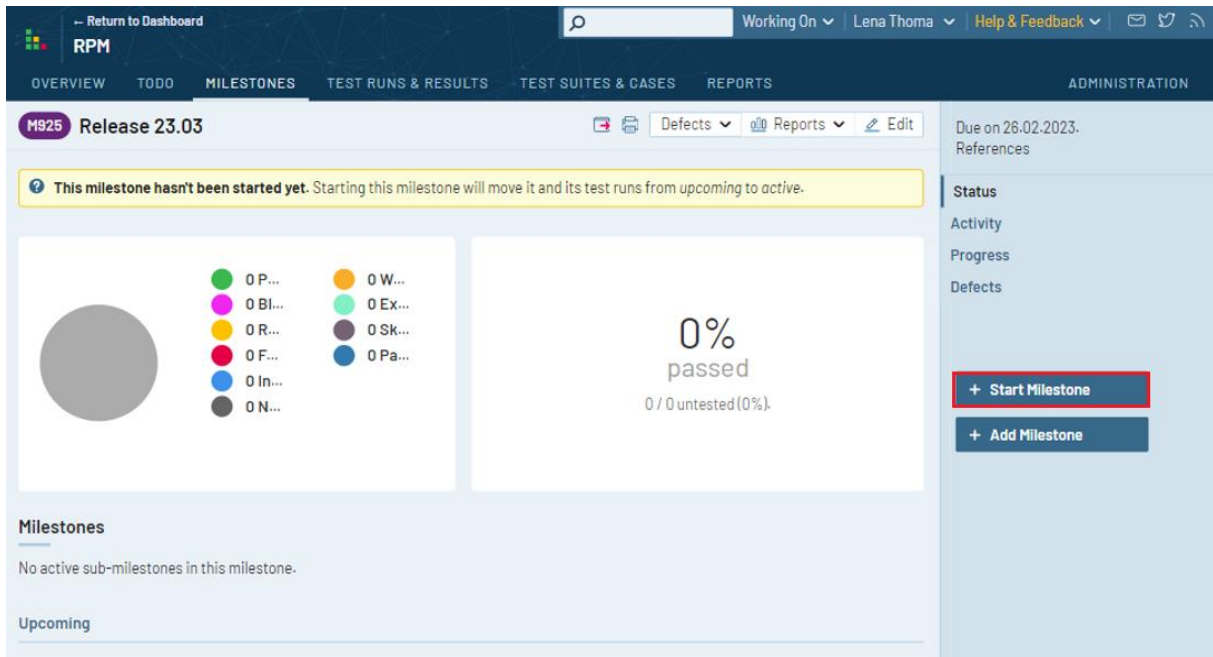


Figure 20: Start of release milestone to trigger update of GitLab variable (milestone detail page)

After the start of the milestone, the GitLab variable is updated for all GitLab projects which are mapped to the TestRail project. The mapping between the GitLab and TestRail projects is summarized in *Table 11*.

The variable is updated by an API call to GitLab for each mapped GitLab project.

```
PUT /projects/:id/variables/REL_VERSION?value=:release_version
```

In case the variable doesn't exist, the variable is created via API.

```
POST /projects/:id/variables?key=REL_VERSION&value=:release_version
```

Figure 21 shows the confirmation message which is shown to the user after the milestone has been started and the GitLab variables are updated. It provides the information for which projects the variable has been updated.

← Return to Dashboard

RPM

Working On | Lena Thoma | Help & Feedback

OVERVIEW | **MILESTONES** | TEST RUNS & RESULTS | TEST SUITES & CASES | REPORTS | ADMINISTRATION

Milestones Defects

+ Add Milestone

Successfully updated variable REL_VERSION to value "23.03" for following GitLab project(s): RPM, API

Successfully started the milestone.

2 open and 15 completed milestones in this project.

Open

Select all

<input type="checkbox"/>		Release 23.03 Due on 26.02.2023 Edit Add Milestone	<div style="width: 0%; height: 10px; background-color: #ccc;"></div>	0%		>
Has 2 sub-milestones and no active test runs.						
<input type="checkbox"/>		Release 23.02 Due on 26.02.2023 Edit Add Milestone	<div style="width: 98%; height: 10px; background-color: #28a745;"></div>	98%		>
Has 2 sub-milestones and 10 active test runs.						




Figure 21: Confirmation after start of release milestone

5.2.3 Start milestone "Regression"

With the start of the regression milestone all necessary test plans for the project should be created. The milestone can either be started from the detail page of the release milestone with a click on the “Start” button next to the milestone name, which is shown in *Figure 22*, or from the right bar in the detail page of the regression milestone itself with a click on the button “Start Milestone” (similar to the release milestone as shown in *Figure 20*).

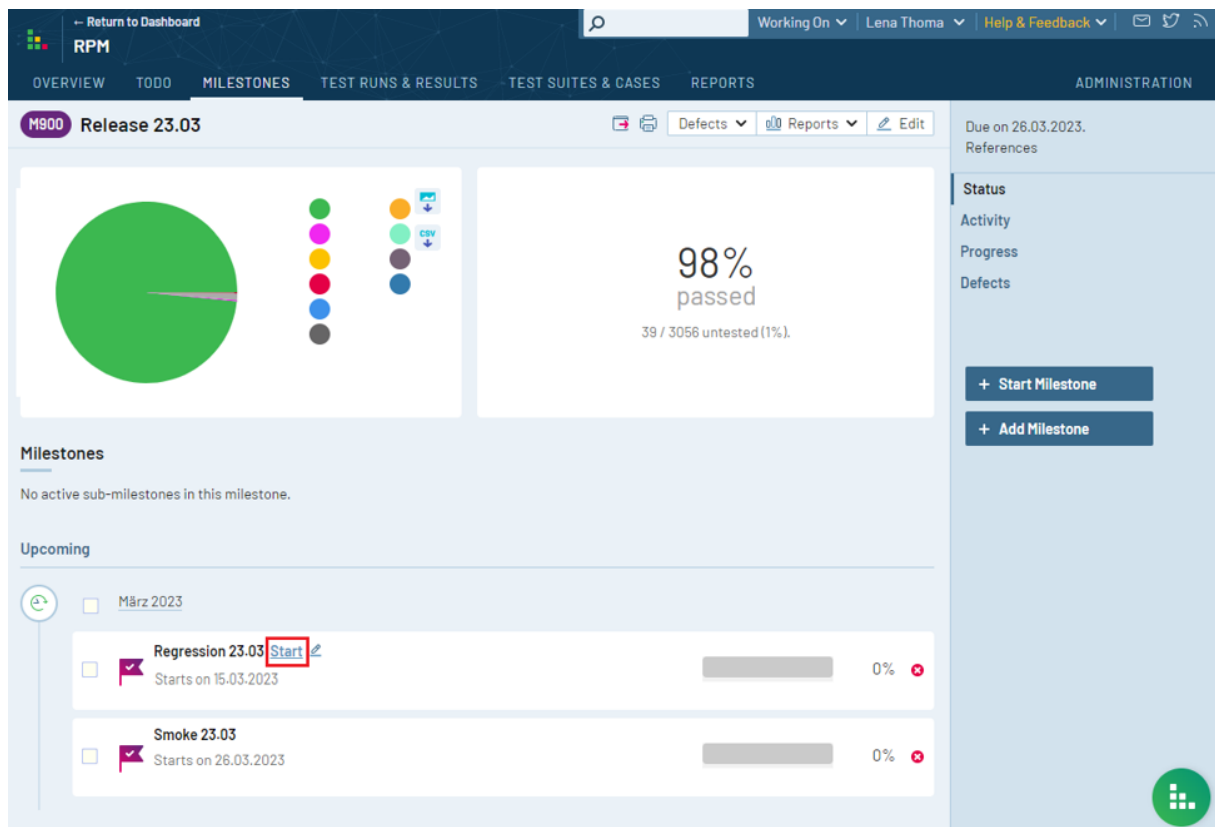


Figure 22: Start of regression milestone to trigger test plan creation (via release milestone detail page)

Figure 23 shows the wizard which is opened after clicking the start button to be able to define the customers for which the test plans should be created. The user can either choose the option “All customers” or select single customers from a multi select box. For the regression usually all customers are used. Furthermore, the start and end date could be adapted which is usually not needed as they are defined during the creation of the milestones.

Start Milestone
✕

Start Date

The actual start date of this milestone; can also be in the past. Leave empty to use *Today*.

End Date

The expected due or end date of this milestone.

Customers

Select customers for which a test plan should be created.

All customers

The following customers only:

- BMW
- Lear
- MAN
- Magna
- Migros
- RPM-Advanced
- RPM-Basic
- RPM-Integrated
- Schaeffler
- Volvo

You can select multiple customers by holding Ctrl on your keyboard.

✓ **Start Milestone**

✕ **Cancel**

Figure 23: Wizard when starting regression milestone

With the start of the milestone a pipeline is triggered in GitLab. Following *Table 13* shows the relevant variables which are sent when triggering the pipeline. The variables `SCOPE` and `CUSTOMER` define which set of test cases is executed on which instance and for which customer. With the combination of the variables `SCOPE`, `REL_VERSION` and `CUSTOMER` the test plan is determined by name when sending back the results while the variable `TESTRAIL_PROJECT_ID` defines the corresponding TestRail project.

Table 13: Parameters for regression test plan creation

Variable	Value
SCOPE	Regression
REL_VERSION	Release version from milestone name (e.g., 23.02)
CUSTOMER	List of all/selected customers of the project separated by “;”
TESTRAIL_PROJECT_ID	ID of the current TestRail project

Figure 24 shows the confirmation message which is provided to the user after starting the milestone. It summarizes that the test plan creation has been triggered including the defined customers. Furthermore, a link to the GitLab pipeline is provided that the user has the possibility to track the status of the test plan creation. Figure 25 shows the landing page of the link.

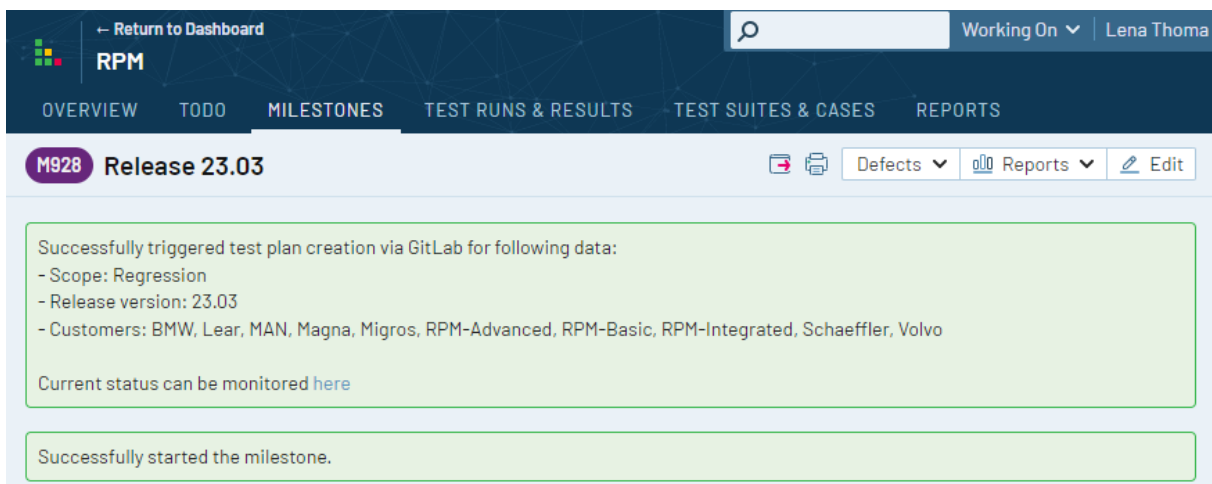


Figure 24: Information message after triggering test plan creation for regression milestone

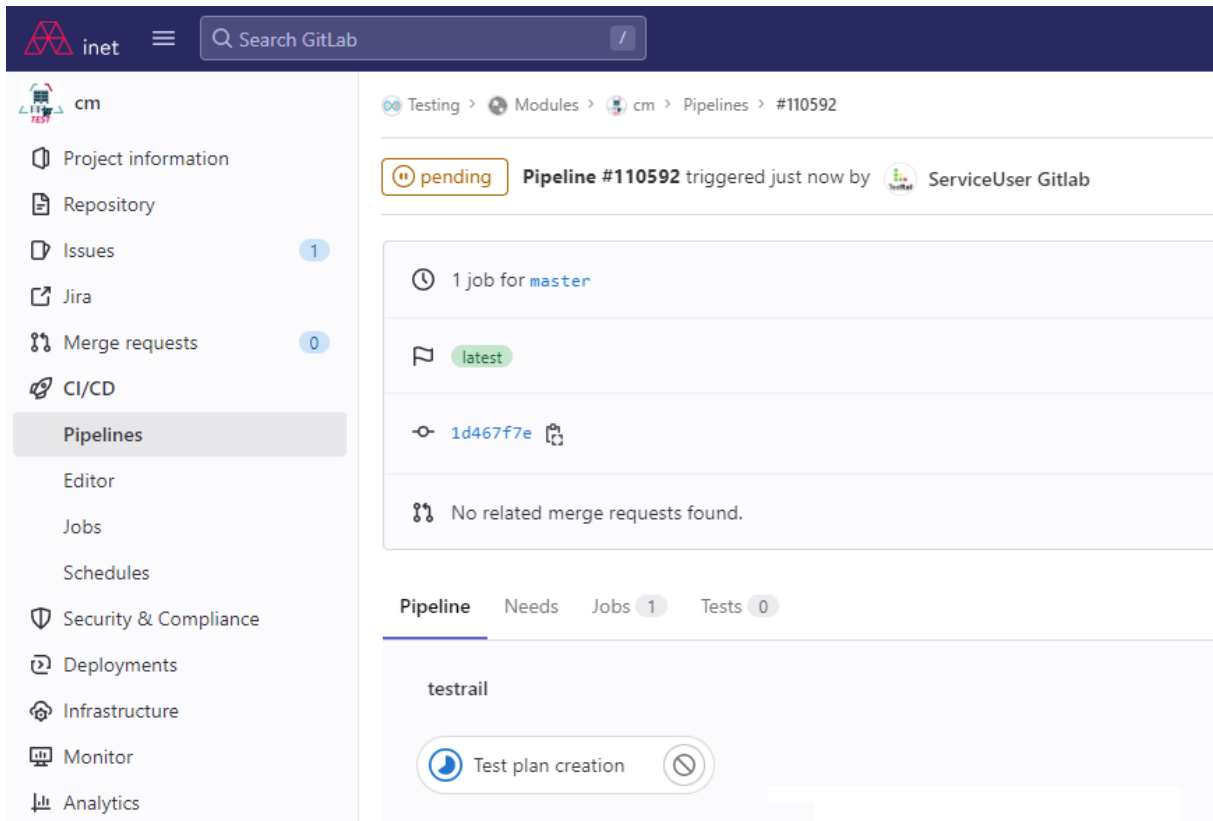


Figure 25: Triggered GitLab pipeline after start of regression milestone

The details about the test plan creation itself are described in chapter 4.4 *Test Plan Creation*.

The preparation of the test plans for regression was always connected to a huge manual effort as an own test plan is needed for each single customer. With the implementation of the automated test plan creation there is no manual effort to create the test plans anymore. Furthermore, the user will have the possibility to also trigger the test execution itself from TestRail (more information in chapter 5.3.1 *Execution of all test cases within a test plan*).

5.2.4 Start milestone "Smoke"

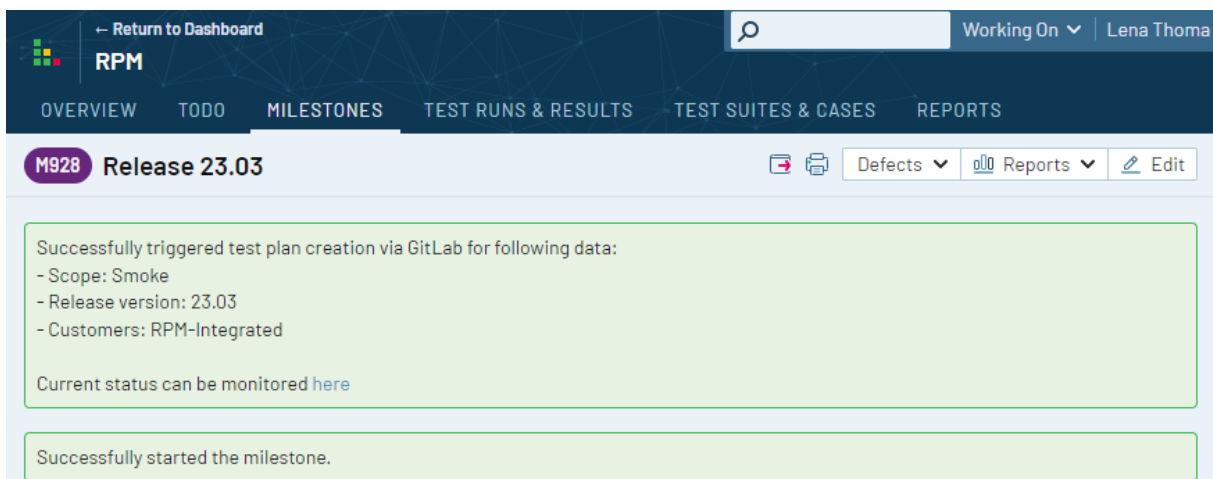
The functionality for the smoke milestone is basically the same as for the regression milestone described in the previous chapter. The only difference between those milestones is the scope and therefore the variables which are sent when triggering the pipeline for the test plan creation are slightly different.

Following *Table 14* summarizes the relevant variables. Only the variable SCOPE differs from the regression milestone.

Table 14: Parameters for smoke test plan creation

Variable	Value
SCOPE	Smoke
REL_VERSION	Release version from milestone name (e.g., 23.02)
CUSTOMER	List of all/selected customers of the project separated by “;”
TESTRAIL_PROJECT_ID	ID of the current TestRail project

For the smoke tests usually only a small set of customers are selected. As shown in the confirmation message in *Figure 26* only the customer “RPM-Integrated” has been defined.



The screenshot shows the TestRail interface for a project named "RPM". The top navigation bar includes "Return to Dashboard", a search bar, and the user name "Lena Thoma". The main navigation menu has "OVERVIEW", "TODO", "MILESTONES", "TEST RUNS & RESULTS", "TEST SUITES & CASES", and "REPORTS". The current view is "M928 Release 23.03". A confirmation message is displayed in a green box, stating: "Successfully triggered test plan creation via GitLab for following data: - Scope: Smoke - Release version: 23.03 - Customers: RPM-Integrated". Below this, it says "Current status can be monitored here" with a link. A second green box below states "Successfully started the milestone."

Figure 26: Information message after triggering test plan creation for smoke milestone

With this implementation the user does no longer need to create the test plans for the smoke tests on the release day manually. The test plans are created automatically for all selected customers and the test execution itself can then be triggered within the individual test plans as described later in chapter *5.3.1 Execution of all test cases within a test plan*.

5.2.5 Close test plans of previous Release

As this will remain a manual task for now, now automation is needed.

5.3 Additional Functionalities

Following sub chapters describe some additional functionalities which support the software testing process and are not related to a specific part of the release process.

5.3.1 Execution of all test cases within a test plan

This enhancement allows the user to trigger the test execution of all test cases within a test plan directly from TestRail.

With the implementation of an additional UI Script a new button is added to the test plan detail page. By clicking the button, a wizard is opened where the information is already prefilled based on the test plan name. In addition, the user can define on which instance and branch the tests should be executed. *Figure 27* shows the wizard with the prefilled information.

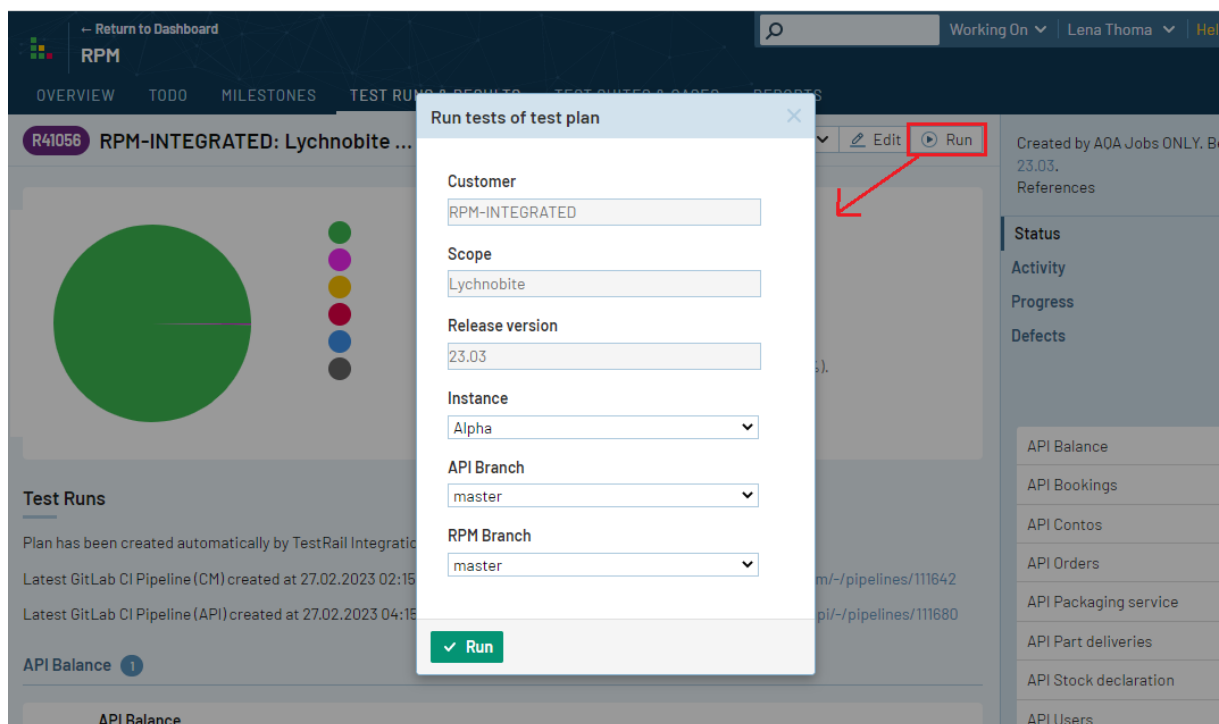


Figure 27: Wizard to run all tests of test plan

The active GitLab branches are determined by a call to the GitLab API.

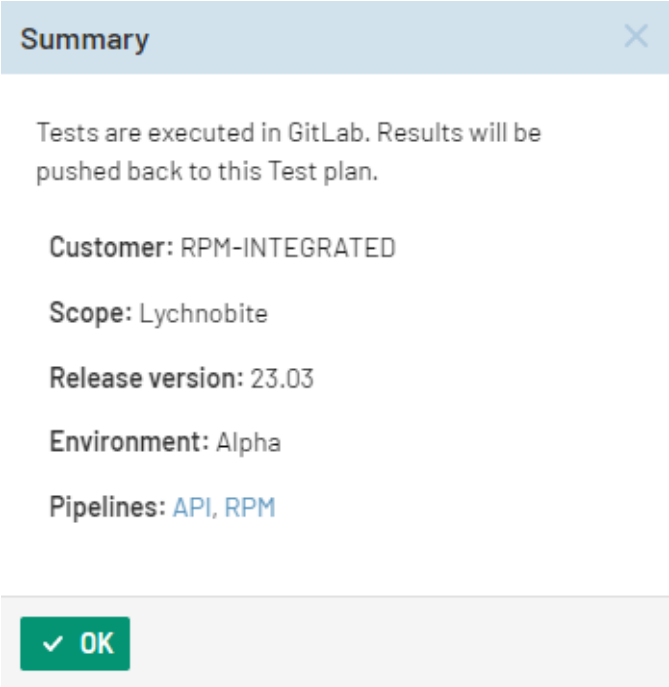
```
GET /projects/:id/repository/branches
```


Table 15 shows a detailed description of each field in the wizard.

Table 15: Field description of wizard when triggering execution of all test cases within plan

Field	Description
Customer	Value is determined from test plan name. The information is needed to define which properties should be used for the test execution.
Scope	Value is determined from test plan name. The information is used to predefine the field instance and branch. In addition, it is relevant to execute the right set of test cases.
Release version	Value is determined from test plan name.
Instance	<p>Drop down contains all available instances. The user can define on which instance the tests should be executed.</p> <p>The default value depends on the scope.</p> <ul style="list-style-type: none"> • “Greenkeeper” and “Lychnobite” → Alpha • “Regression” → Integration • “Smoke” → Production
Branch	<p>There can be either one or multiple fields for the branch selection depending on how many GitLab projects are linked to the current TestRail project.</p> <p>The drop down shows all active branches of the corresponding GitLab project.</p> <p>The default value depends on the scope.</p> <ul style="list-style-type: none"> • “Greenkeeper” and “Lychnobite” → master • “Regression” and “Smoke” → latest release branch

After starting the run, a pipeline for each related GitLab project is triggered by a call to GitLab. Furthermore, a summary is provided to the user including all defined details as well as the link to the triggered pipelines for each GitLab project. This allows the user to track the status of the test execution. *Figure 28* shows an example of the summary after the user has triggered the test execution.



The image shows a summary dialog box with a light blue header containing the word "Summary" and a close button (X). The main content area is white and contains the following text: "Tests are executed in GitLab. Results will be pushed back to this Test plan." followed by several key-value pairs: "Customer: RPM-INTEGRATED", "Scope: Lychnobite", "Release version: 23.03", "Environment: Alpha", and "Pipelines: API, RPM". At the bottom of the dialog is a grey bar with a green button containing a white checkmark and the text "OK".

Figure 28: Summary after triggering execution of all test cases

5.3.2 Execution of failed test cases within a test plan

The execution of all failed test cases within a test plan is an extension of the implementation described in the previous sub chapter. The wizard is extended with the option to select if either all test cases or only failed test cases should be executed. *Figure 29* shows the wizard with the new field to define the execution set.

Run tests of test plan

Customer
BMW

Scope
Greenkeeper

Release version
23.03

Instance
Alpha

RPM Branch
master

API Branch
master

Execution set
 All test cases Failed test cases

✓ Run

Figure 29: Extension of wizard to run tests of plan with field execution set

When the user selects the option to only trigger failed test cases, all test cases with status “FAILED” (Status ID = 5) are collected before triggering the pipeline. A first call to the TestRail API is used to get the details of the test plan itself and to determine the ID of all test runs within the test plan.

```
GET /api/v2/get_plan/:planId
```

For each test run within the test plan another API call is done to collect all test cases which are failed. To only get failed test cases the filter “status_id=5” can be used.

```
GET /api/v2/get_tests/:runId&status_id=5
```

After collecting all failed test cases, the pipeline is triggered with a call to the GitLab API. By defining the GitLab variable “CASE_ID” it is ensured that only those test cases are executed. In addition, the test result of all failed test cases is set to “RETEST” by another API call to TestRail for each test run which includes at least one failed test case.

```
POST /api/v2/add_results_for_cases/:runId
```

Same as when triggering the execution of all test cases the user will get a summary as well as a link to the GitLab pipeline after starting the run. *Figure 30* shows an example. It can be seen that the summary has also been enhanced with the information about the execution set (all or failed) which has been defined by the user.

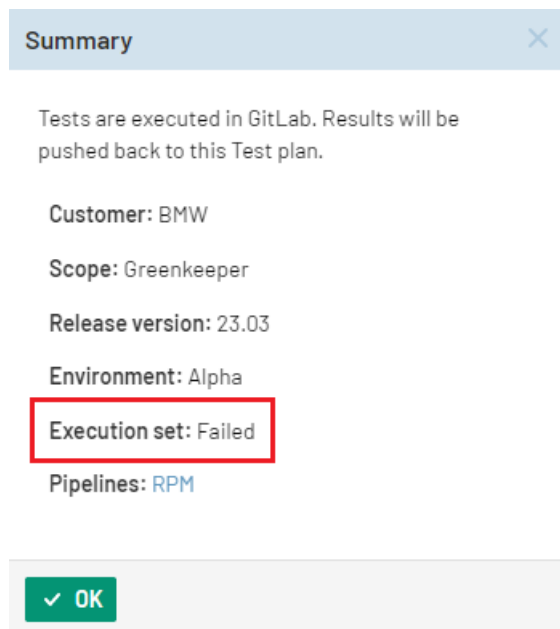


Figure 30: Extension of summary after triggering execution of test cases with execution set

5.3.3 Test plan creation independent of milestone start

The creation of test plans based on a scope should not only be possible with the start of the regression or smoke milestone. A separate wizard which is accessible from the tab “Test runs & results” allows the user to create new test plans based on the scope whenever it is needed independent from the start of a milestone. *Figure 31* shows the wizard where the user can define the scope, release version and the customers.

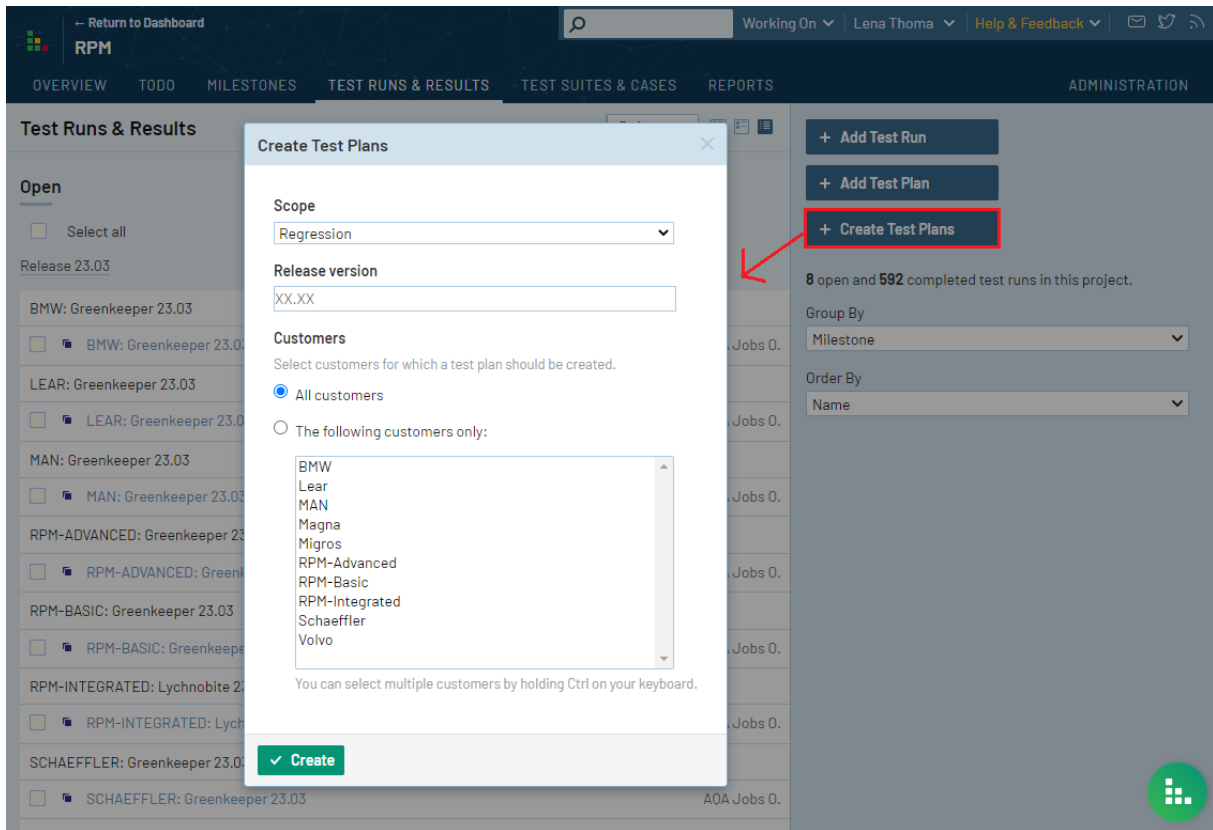


Figure 31: Wizard to create test plan independent from milestone start

The user can decide based on which scope the test plan should be created. The scope defines which test run types and execution types will be part of the test plan. The selection of the test cases is done based on *Table 10: Definition of Scope*.

Furthermore, the release version can be defined by the user. This information is relevant to be able to assign the test plan to the correct milestone.

The creation of the test plan can either be triggered for all customers or only a subset of customers. Therefore, the user can either select the option “All customers” or select one or multiple customers from the multi select box.

After clicking the button “Create” the test plan creation is triggered in GitLab using the same logic as for the creation of the test plans when starting one of the milestones “Regression” or “Smoke” as described in chapter *5.2.3 Start milestone "Regression"* and *5.2.4 Start milestone "Smoke"*.

Figure 32 shows the confirmation message which is provided to the user including the link to the GitLab pipeline to track the status of the test plan creation.

← Return to Dashboard

RPM

Working On | Lena Thoma | Help & Feedback

OVERVIEW | TODO | MILESTONES | TEST RUNS & RESULTS | TEST SUITES & CASES | REPORTS | ADMINISTRATION

Test Runs & Results

Defects

Successfully triggered test plan creation via GitLab for following data:

- Scope: Regression
- Release version: 23.03
- Customers: RPM-Integrated

Current status can be monitored [here](#)

Open

Select all

Release 23.03

BMW: Greenkeeper 23.03		
<input type="checkbox"/>	<input checked="" type="checkbox"/> BMW: Greenkeeper 23.03	AOA Jobs O.
LEAR: Greenkeeper 23.03		
<input type="checkbox"/>	<input checked="" type="checkbox"/> LEAR: Greenkeeper 23.03	AOA Jobs O.
MAN: Greenkeeper 23.03		

+ Add Test Run

+ Add Test Plan

+ Create Test Plans

8 open and 592 completed test runs in this project.

Group By
Milestone

Order By
Name

Figure 32: Confirmation after triggering test plan creation

5.4 Test Plan Creation

The automated test plan creation is done based on the attributes which are set for the individual test cases. *Figure 33* shows the detail page of a test case where the relevant attributes are highlighted. They are used to filter for all relevant test cases. Depending on the scope a different set of test cases is selected.

The relevant case fields for the selection of the test cases are:

- Test run type (Smoke or Regression)
- Execution type (Automated or Manual)
- Customer (e.g., RPM-Integrated, BMW)

The screenshot displays the RPM software interface for test case C2972. The main content area shows a table with the following data:

Type	Priority	Estimate	References
Regression	Medium	None	None
ExecutionType Automated	Test layer UI	TestRunType Regression	

Below the table, the 'Customers' field is highlighted with a red box and contains the text: RPM-Basic, RPM-Advanced, RPM-Integrated.

The 'Steps' section lists the following test steps:

- I create an entry booking from supplier to plant
- I login as "supplierBasis"
- I switch to the account "default supplier" from properties
- I open the movements page
- I define the movements filter criteria "booking number"
- I search for movements by booking number or routing id
- I can find my booking in the movements result list
- I open the document management from movement result
- I upload a document of type ".pdf"
- I can find the document

The right sidebar contains the following sections:

- Details
- Tests & Results
- Defects
- History
- People & Dates

The 'People & Dates' section shows the following information:

Created	Updated
AOA Jobs ONLY 4/28/2020 6:06 AM	AOA Jobs ONLY 11/28/2022 10:25 AM

Figure 33: Relevant test case fields for test case selection depending on scope

As described in *Table 16*, for the creation of the test plans the input parameters **CUSTOMER**, **SCOPE** and **REL_VERSION** are relevant.

Table 16: Relevant parameters for the test plan creation

Parameter	Description
<i>CUSTOMER</i>	For the test plan creation either one or multiple customers can be defined. For each customer a separate test plan will be created.
<i>SCOPE</i>	The scope defines which test cases are added to the test plan. During the test plan creation, the test cases are filtered by test run type and execution type.
<i>REL_VERSION</i>	The release version is needed to assign the test plan to the correct milestone in TestRail.

The test plan creation is implemented as follows.

1. Get all test suites and cases for the corresponding TestRail project

- Get all test suites

```
GET /api/v2/get_suites/:projectId
```

- Loop through all test suites and get the test cases

```
GET /api/v2/get_cases/:projectId&suite_id=:suiteId
```

2. Determine the milestone based on the release version

- Get all active milestones from the project

```
GET /api/v2/get_milestones/%d&is_completed=0
```

- Filter for the correct milestone by milestone name

- First choice would be the milestone for the corresponding scope
 - ➔ SCOPE + “ “ + REL_VERSION
- Fallback is the release milestone
 - ➔ “Release “ + REL_VERSION

3. Prepare test plan for each customer

- Filter for all relevant test cases depending on the scope and customer
 - Greenkeeper and Lychnobite
 - ➔ Test run type: All
 - ➔ Execution type: Automated
 - Regression
 - ➔ Test run type: All
 - ➔ Execution type: Automated + Manual
 - Smoke
 - ➔ Test run type: Smoke
 - ➔ Execution type: Automated + Manual
- Define all details like project ID, name, description, milestone ID for the test plan and set relevant test cases

4. Create prepared test plans

- Add plan for each customer

```
POST /api/v2/add_plan/:projectId
```

The automated test plan creation is used when starting the milestones “Regression” and “Smoke” as described in chapters *5.2.3 Start milestone "Regression"* and *5.2.4 Start milestone "Smoke"* or when triggering the test plan creation independent from the start of a milestone as described in chapter *5.3.3 Test plan creation independent of milestone start*.

The automated test plan creation also leads to the advantage that all test plans follow the same naming convention ([Customer]: [Scope] [Release version]) and are automatically assigned to the correct milestone.

5.5 Test Plan Determination

The test plan determination should lead to the advantage that the test plan IDs don't need to be updated for each release. The test plan is determined based on the customer, scope, and release version.

With the naming convention, which is used during the test plan creation, the test plan can be determined by using the test plan name. With an API call to TestRail all open test plans can be gathered which are then filtered by the test plan name.

```
POST /api/v2/get_plans/%s&is_completed=0
```

Furthermore, the test plan determination is connected to the test plan creation, which means in case no active test plan can be found, a new test plan is created.

Therefore, it is ensured that with the first execution of the Greenkeepers and Lychnobites in a release, a new test plan is created automatically. As the execution is triggered by GitLab schedules and also the release version is updated automatically with the start of a new release (see *5.2.2 Start milestone "Release"*) no manual effort is needed anymore to initiate the creation of the Greenkeeper and Lychnobite test plans for a new release.

5.6 Automatic Assignment of new Test Cases to Test Plans

The result of a test case can only be sent to a test plan in case it is assigned to the test plan. Therefore, in case a new test case is created, it should be added to all relevant test plans to be part of the test reporting.

When sending the result of a test case which is not part of the test plan, the API call will lead to an error. Instead of ignoring this error, it is now verified if the test case is relevant for the corresponding test plan. This is done in the same way as when creating a test plan (see *5.4 Test Plan Creation*). Depending on the scope and the customer of the test plan and the defined values for the test case fields "Test run type", "Execution type" and "Customers", it is determined if the test case is relevant for the test plan. In case the test case is relevant, it is added to the test plan and the result is sent again.

With this implementation it is ensured that the test case is added to the test plan with the first execution after the test case has been created. This is especially relevant for the Greenkeepers and Lychnobites as those test plans are open for a longer time range.

6 Evaluation

This chapter evaluates the outcome of the thesis. The main goal was to reduce the manual effort within the testing process. Therefore, first of all the manual effort before and after the optimization is compared. After that the research objectives, question and hypothesis are reviewed.

6.1 Comparison of Efforts

In *Table 7* the manual effort of recurring testing activities which are related to the test administration has been summarized. It also points out the impact when changing from quarterly to monthly releases.

After the optimization another estimation of the manual effort has been done to be able to see the improvement. In *table Table 17)* the manual effort before and after the optimization is compared. Furthermore, the improvement is visible which has been achieved. The comparison is done based on the effort with monthly releases.

Table 17: Effort of recurring testing activities before and after optimization

	Before optimization	After optimization	Improvement
<i>Creation of milestones for all projects (RPM, MM, TM, IS, FM)</i>	600 min = 10 h	120 min = 2 h	- 80 %
<i>Creation of test plans (Lychnobites, Greenkeepers, Regression, Smoke)</i>	21,600 min = 360 h	5 * 2 min * 12 = 120 min = 2 h	- 99 %
<i>Update of GitLab schedules (Lychnobites, Greenkeepers)</i>	2,280 min = 38 h	0 h	- 100 %
<i>Update of existing test plans with new test cases (Lychnobites, Greenkeepers)</i>	1,300 min = 21 h 40 min	0 h	- 100 %
<i>Triggering of automated test execution (Regression, Smoke)</i>	3,240 min = 54 h	54 * 2 min * 12 = 1,296 min = 21 h 36 min	- 60 %
SUM	483 h 40 min	25 h 36 min	- 94,7 %

For the creation of the milestones the effort could be reduced by 80%. While it was necessary to create the milestones for each project individually in the past, this can be done in one stop now. This means even with the creation of a new project in TestRail the effort would not increase. The creation of the test plans itself has been fully automated the only manual effort which is still left is to trigger the regression and smoke test plan creation for each project, therefore an effort of 2 minutes per project has been considered. The update of GitLab schedules as well as the update of existing test plans has been eliminated completely. The triggering of automated tests has also been simplified with the optimization which led to a reduction of the manual effort by 60 %. The details of how the values before the optimization are composed is shown in *Table 7*.

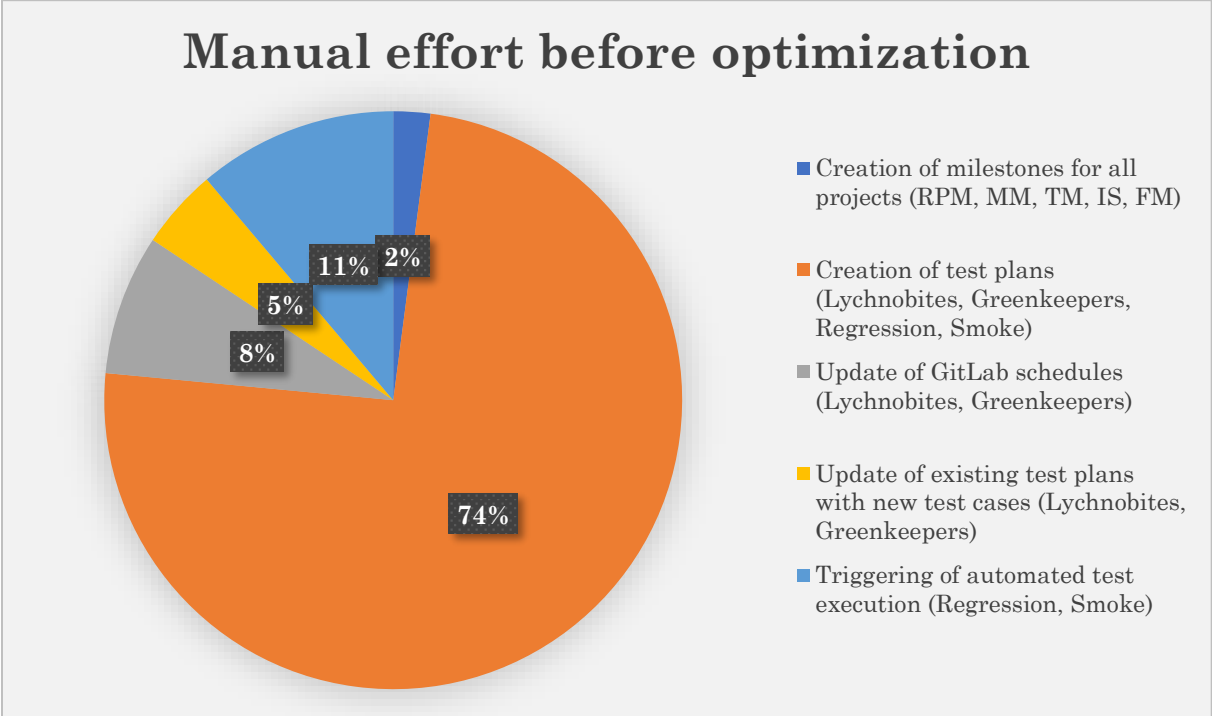


Figure 34: Manual effort for administrative tasks before optimization (own illustration)

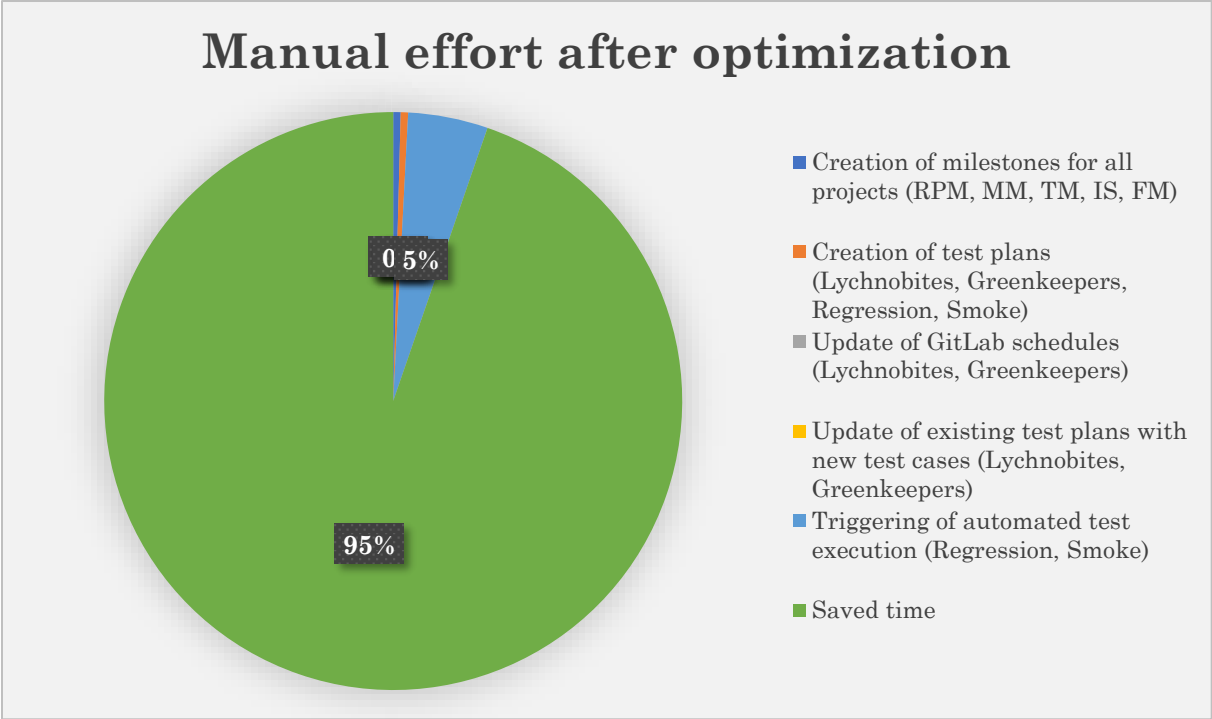


Figure 35: Manual effort for administrative tasks after optimization (own illustration)

The diagrams in Figure 34 and Figure 35 above visualize that the effort has been reduced enormously. The green piece in Figure 35 shows the percentage of the saved time through the improvement of the process. Before the optimization the creation of the test plans was by far the highest effort. With the automated test plan creation nearly three-quarters of the whole manual effort have been saved. After the optimization the triggering of the automated tests is the highest effort. At the moment, this cannot be fully automated as some manual configuration needs to be done before triggering the individual test plans. In future, this would be the most interesting task to reduce the remaining manual effort even more.

The comparison above only includes the testing activities related to test administration. To determine how much the manual effort of the overall testing process has been reduced with the optimization, the testing activities including manual testing and test maintenance of the project used for the prototype is compared. Table 18 contains the results of the comparison.

Table 18: Manual effort for the testing process in the RPM project with monthly releases

	Before optimization	After optimization	Improvement
<i>Creation of milestones for RPM project</i>	10 min	2 min	- 80 %
<i>Creation of test plans (Lychnobites, Greenkeepers, Regression, Smoke)</i>	370 min	2 min	- 99 %
<i>Update of GitLab schedules (Lychnobites, Greenkeepers)</i>	8 * 5 min = 40 min	0 min	- 100 %
<i>Update of existing test plans with new test cases (Lychnobites, Greenkeepers)</i>	5 min * 4 = 20 min	0 min	- 100 %
<i>Triggering of automated test execution (Regression, Smoke)</i>	(10 + 1) * 5 min = 55 min	(10 + 1) * 2 min = 22 min	- 60 %
<i>Execution of manual tests</i>	6 h	6 h	- 0 %
<i>Test maintenance</i>	10 h	10 h	- 0 %
SUM	24 h 15 min	16 h 26 min	- 32 %

The values for the administrative tasks which have been already part of the previous comparison have been calculated down to one month only including the RPM project. This means efforts which affect each project the same have been divided by 5, and for efforts which are related to the amount of test plans only the test plans of the RPM project have been considered – 8 Lychnobites/Greenkeepers, 10 Regression, 1 Smoke (see *Table 6*).

For the execution of the manual tests and the effort for test maintenance, the average of the logged time of the last 4 releases has been used.

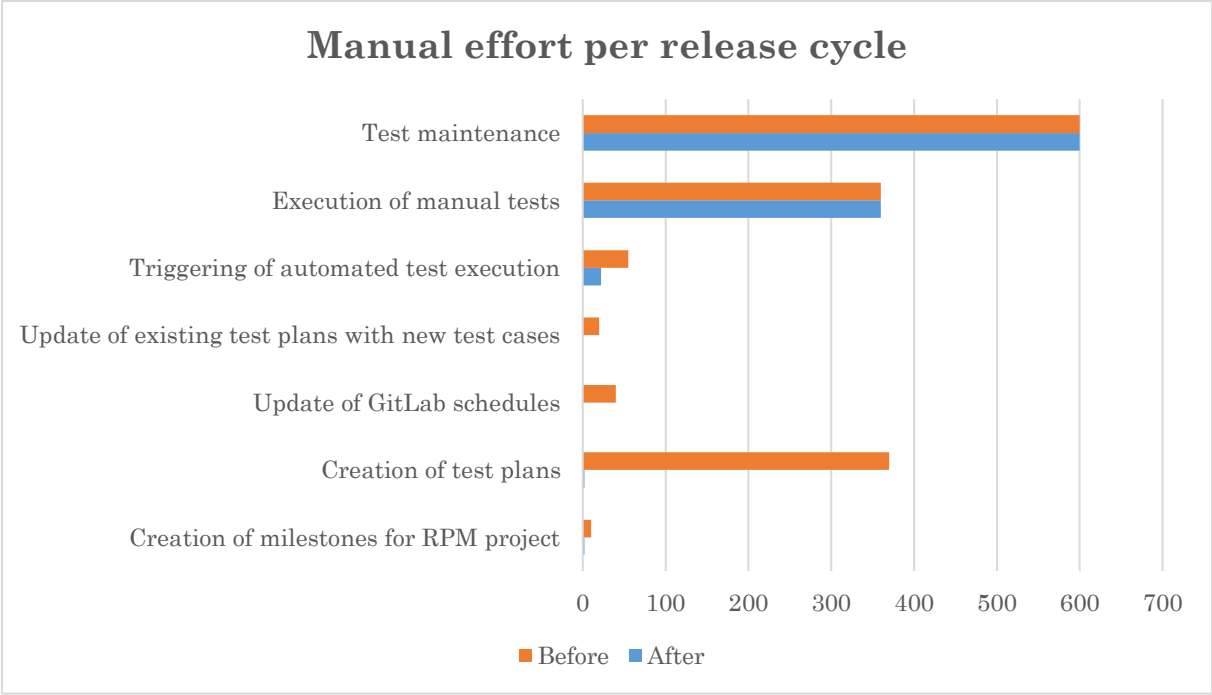


Figure 36: Manual effort per release cycle before and after optimization (own illustration)

Figure 36 compares the manual effort per release cycle before and after the optimization. It can be seen that most of the manual effort for recurring administrative activities has been eliminated. The main effort which is remaining is the test maintenance as well as the execution of the manual tests during regression. With the reduction of the administrative manual tasks, there will be more time to also improve the effort in those areas.

6.2 KPI system

The optimization of the software testing process has freed up valuable tester's resources, which can be reallocated to either work on automating remaining manual test cases or stabilizing existing automated tests. This will reduce the effort for recurring manual tasks even more and therefore result in further improvements.

To track the development of the software testing process a set of Key performance indicators (KPI) has been defined which are described below.

- **Automated test ratio**

This indicator measures how much of the total test cases are automated. The higher the ratio the better. The ratio should get higher because new test cases have to be automated and the remaining manual test cases should be automated over time.

$$\text{Automated test ratio} = \frac{\text{Number of automated tests}}{\text{Total number of tests}}$$

- **Quality ratio**

The quality ratio signifies the success rate of the executed test cases. Thereby, only the results of the last recent run of each test case are considered.

$$\text{Quality ratio} = \frac{\text{Number of passed tests}}{\text{Total number of executed tests}}$$

- **Test reliability**

The test reliability indicates how valuable the feedback of the tests is. Especially for automated tests this is an important KPI. It measures how many false positive and false negative test results have been reported. A false positive result means that the test shows a failure even though there is none. A false negative result means that a test doesn't show a failure even though there is a bug.

A low test reliability can be critical as the feedback of the tests is not trustworthy. Therefore, it is important to maintain and stabilize the tests.

$$\text{Test reliability} = \frac{\text{Total number of executions} - (\text{False positives} + \text{False negatives})}{\text{Total number of executions}}$$

- **Execution time**

This KPI measures the execution time of all test cases. Beside the total execution time it should also be possible to distinguish between the execution time of manual and automated test cases. While the total execution time might increase as there are more and more test cases, the execution time per test should get lower over time.

- **Detected defects**

The detected defects are the sum of all bugs found. The defects can be categorized by its severity in minor, major and critical defects.

- **Escaped defects**

The escaped defects indicate the number of defects which have not been detected by a tester but by the customer. Each escaped defect must be analysed in detail to find out why it has been overlooked by internal testing. A high number of escaped defects indicates that more extensive testing is needed.

- **Active defects**

This KPI counts the number of defects which are not resolved yet. This means the status can be either open, in progress or ready for testing. The goal is to keep the number low as it indicates a high level of quality.

Each single KPI should be measured on project level and on company level for each release to point out strengths and weaknesses. The KPIs should support the decision about future actions.

6.3 Review of Research Question and Hypothesis

At the beginning of this thesis following research question and hypothesis have been defined:

- **Research question**

“To what extend can the manual effort in the software testing process for a global logistics software provider be reduced by the automation of recurring administrative tasks to address the challenges of shorter release cycles?”

- **Hypothesis**

“With the first insights into the possibilities of the used tools in the software testing process of Alpega, I hypothesize that the administrative tasks related to test plan creation, execution, and reporting can be fully automated to serve as reliable quality gate for future releases. As a result, the overall manual effort will be reduced by 25%.”

The implementation of the prototype has shown that the administrative activities related to test plan creation, execution, and reporting can be fully automated. With the elimination of those manual tasks the manual effort for the administrative tasks of a whole year with monthly releases has been reduced over 94 %. Considering the manual testing and maintenance effort, the prototype has shown that the effort has been reduced by 32 % for each release cycle. Especially with regard to the more frequent releases which are planned for the upcoming year, this leads to huge savings of manual resources. Therefore, it can be stated that the hypothesis has been confirmed.

6.4 Summary of Outcomes

The results of the thesis have shown that the automation of recurring administrative activities within the software testing process can lead to a remarkable reduction of the manual efforts. With the automation of recurring tasks like the creation and update of test plans or the execution of the test cases a lot of the testers time has been freed up, which can be used for the automation of remaining manual test cases and the stabilization of existing automated tests. Thereby, it can be expected that the manual effort will be reduced steadily in future as the effort for the execution of manual test cases will be decreased more and more, and the maintenance work will also be reduced with the stabilization of the automated tests.

6.5 Review of Research Objectives

This chapter should summarize if and how the research objectives have been reached by the optimization of the test management process.






Following goals have been defined at the beginning of the thesis:

- Reduction of the manual effort for the administration of test plans
The manual activities related to the test administration have been reduced significantly. *Table 17* compares the manual effort before and after the optimization based on monthly releases. Overall, over 94 % of the manual effort has been eliminated by the optimization.
- Reduction of the risk of incomplete test plans
With the automated test plan creation and the automatic assignment of new test cases to existing test plans, the risk of incomplete test plans has been reduced significantly. For each new test plan all current test cases are filtered by a set of attributes depending on the scope and also new created test cases are assigned with the first execution after their creation.

Even though the risk has been lowered significantly, there is still the remaining risk that the test case fields are not set correctly. To reduce this risk the relevant test case fields are mandatory fields without default value, which means the user is forced to define a value when creating a new test case. With this additional adaption the risk is negligibly small.
- Increase of available QA resources for other tasks
As described above the manual effort for the administration of the test plans has been decreased a lot. Also, the execution of the tests has been simplified and only need little manual interactions, as the setup is done automatically and only the actual trigger remains a manual task.

In summary it can be stated that all objectives defined in *Table 1* have been fulfilled by the optimization of the test management process. Following *Table 19* gives an overview of the research objectives and contain a reference to the corresponding chapters.

Table 19: Review of research objectives

	Target	Status
Test plan creation	100% automated	 see 5.4 <i>Test Plan Creation</i>
Incomplete test plans	0%	 see 5.4 <i>Test Plan Creation</i> and 5.6 <i>Automatic Assignment of new Test Cases to Test Plans</i>
Execution	Greenkeeper and Lychnobite automated Regression and Smoke manual trigger but automated setup	 see 5.3.1 <i>Execution of all test cases within a test plan</i> and 5.3.2 <i>Execution of failed test cases within a test plan</i>
Reporting	Automated determination of correct test plan	 see 5.5 <i>Test Plan Determination</i>
KPI system	Introduced to track progress	 see 6.2 KPI system

7 Conclusion and Outlook

The outcome of the thesis shows that automation in software testing can cover more than only the automation of manual test cases. There are a lot of recurring manual activities which can be eliminated by automation. The automation of administrative tasks during the software testing processes brings enormous reductions of manual effort which frees up time for other tasks.

With the thesis the creation of the milestones for each release as well as the execution of the test plans has been simplified significantly. Furthermore, the creation and determination of the test plans have been fully automated, which was the biggest improvement when comparing the time effort before and after the optimization. Overall, the optimization of the process has saved over 94% of the effort for administrative tasks during the software testing process. The comparison of the manual efforts per release cycle for the RPM project has shown that the overall manual effort has been reduced by over 30%.

As a next step the new features will be rolled out for all solutions at Alpega. The process team which has been working on the prototype for the RPM project, will establish the new process for the whole QA department. The new introduced KPI system should make the future progress visible.

At the moment, there are still a lot of manual triggers in the software testing process which are not a lot of effort but still need manual interaction. This should remain until the whole process has been stabilized. In future, when establishing a CI/CD culture the goal is to also automate those triggers to have a fully automated process without the need of manual interaction.

The idea is to trigger the test plan creation and execution after a deployment to the test environment to serve as quality gate before the deployment to other environments. Before deploying the new version to other environments, the pass rate of the test plan is determined and only if a certain limit has been exceeded the deployment will start. In that case there is no need to trigger the creation of the test plans or the execution of the test cases.

References

- Akduygu, C. (2019) 'Integrate Test Automation Results with TestRail – TestNG', *SW Test Academy*, 23 September [Online]. Available at <https://www.swtestacademy.com/integrate-test-automation-results-with-testrail-testng/> (Accessed 19 February 2023.571Z).
- Alpega Group (2022a) *Über uns | Alpega* [Online], Alpega Group. Available at <https://www.alpegagroup.com/de/ueber-uns/> (Accessed 3 July 2022).
- Alpega Group (2022b) *Agile Tester - Scrum Master Chapter & Agile Working - Alpega Portal* [Online]. Available at <https://portal.wkts.eu/display/SMCHAPTER/Agile+Tester> (Accessed 11 December 2022).
- Alpega Group (2023a) *CI/CD - Alpega TMS Product* [Online]. Available at <https://portal.wkts.eu/pages/viewpage.action?pageId=100972895> (Accessed 4 February 2023).
- Alpega Group (2023b) *RPM - Testing - Reusable Packaging Management - Alpega Portal* [Online]. Available at <https://portal.wkts.eu/display/MOD/RPM+-+Testing> (Accessed 6 April 2023).
- Alsaqqa, S., Sawalha, S. and Abdel-Nabi, H. (2020) 'Agile Software Development: Methodologies and Trends', *International Journal of Interactive Mobile Technologies (iJIM)*, vol. 14, no. 11, p. 246.
- Asfaw, D. (2015) 'Benefits of automated testing over manual testing', *International Journal of Innovative Research in Information Security*, vol. 2, no. 1, pp. 5–13.
- Atlassian (2023) *Scrum - what it is, how it works, and why it's awesome* [Online]. Available at <https://www.atlassian.com/agile/scrum> (Accessed 6 April 2023).
- Bhat, A. (2018) 'Quantitative Survey Questions: Definition, Types and Examples', *QuestionPro*, 24 August [Online]. Available at <https://www.questionpro.com/blog/quantitative-survey-questions/> (Accessed 2 February 2023.765Z).
- Elbaum, S., Rothermel, G. and Penix, J. (2014) 'Techniques for improving regression testing in continuous integration development environments', *Proceedings of the 22nd*

ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 235–245.

Fettke, P. (2006) ‘State-of-the-Art des State-of-the-Art’, *1861-8936* [Online]. Available at <https://dl.gi.de/handle/20.500.12116/12575>.

Garousi, V. and Elberzhager, F. (2017) ‘Test Automation: Not Just for Test Execution’, *IEEE Software*, vol. 34, no. 2, pp. 90–96 [Online]. DOI: 10.1109/ms.2017.34.

Ghani, U. (2021) *The importance of test management in agile - Work Life by Atlassian* [Online]. Available at <https://www.atlassian.com/blog/add-ons/test-management-in-agile> (Accessed 12 December 2022).

GitLab (2023) *Cron | GitLab* [Online]. Available at <https://docs.gitlab.com/ee/topics/cron/> (Accessed 6 April 2023).

Greyling, D. (2022) ‘Test Managers in Agile - Inspired Testing’, *Inspired Testing*, 21 November [Online]. Available at <https://www.inspiredtesting.com/news-insights/insights/522-test-managers-in-agile> (Accessed 19 February 2023.281Z).

Loke Mun Sei (2015) *Automating Test Activities: Test Cases Creation, Test Execution, and Test Reporting with Multiple Test Automation Tools* [Online]. Available at <https://zenodo.org/record/1338500>.

Mascheroni, M. A. and Irrazábal, E. (2018) ‘Continuous Testing and Solutions for Testing Problems in Continuous Delivery: A Systematic Literature Review’, *Computación y Sistemas*, vol. 22, no. 3.

Mohsin Nazir (2020) ‘Software Quality Assurance and Android Application Development: A Comparison among Traditional and Agile Methodology’, *Lahore Garrison University Research Journal of Computer Science and Information Technology*, vol. 4, no. 4, pp. 1–29 [Online]. DOI: 10.54692/lgurjcsit.2020.0404105.

Mousaei, T. M. (2020) ‘Review on Role of Quality Assurance in Waterfall and Agile Software Development’, *Journal of Software Engineering & Intelligent Systems*, vol. 5.

Munsamy, S. (2019) ‘Why do we need a test management tool? - Inspired Testing’, *Inspired Testing*, 6 February [Online]. Available at <https://www.inspiredtesting.com/news-insights/insights/358-why-do-we-need-a-test-management-tool> (Accessed 19 February 2023.044Z).

Nahnsen, R. (2020) 'CI/CD: Was ist Continuous Delivery?', *Uptrends Blog*, 10 July [Online]. Available at <https://blog.uptrends.de/technologie/ci-cd-was-ist-continuous-delivery/> (Accessed 11 December 2022).

Oliinyk, B. and Vasyi, O. (2019) 'Automation in software testing, can we automate anything we want', *Proceedings of the 2nd Student Workshop on Computer Science & Software Engineering*, pp. 224–234 [Online]. Available at <http://ceur-ws.org/vol-2546/paper16.pdf>.

Pawlak, M. and Poniszewska-Marańda, A. (2018) 'SOFTWARE TEST MANAGEMENT APPROACH FOR AGILE ENVIRONMENTS', *Information System in Management*, vol. 7, no. 1, pp. 47–58 [Online]. DOI: 10.22630/ISIM.2018.7.1.5.

Rede, D. (2022) *Streamlining test automation with TestRail* [Online]. Available at <https://blog.gurock.com/streamlining-test-automation/> (Accessed 19 February 2023.667Z).

Sabev, P. and Grigorova, K. (2015) 'Manual to automated testing: An effort-based approach for determining the priority of software test automation', *International Journal of Computer, Electrical, Automation, Control and Information Engineering*, vol. 9, no. 12, pp. 2456–2462.

Salzburg Research (2023) *Prototyping | Methodenpool* [Online]. Available at <https://methodenpool.salzburgresearch.at/dsimethode/prototyping/> (Accessed 2 February 2023).

Shah, H. (2019) 'Automated Functional Testing: Building a Successful Strategy', *Simform*, 5 March [Online]. Available at <https://www.simform.com/blog/automated-functional-testing/> (Accessed 18 February 2023.963Z).

Shannon, J.-B. (2022) *The Importance of Test Management in Software Testing* [Online]. Available at <https://www.orientsoftware.com/blog/test-management-in-software-testing/> (Accessed 19 February 2023).

Son, H. (2022) *How to Choose the Right Test Management Tool (With Evaluation Checklist)* [Online]. Available at <https://blog.gurock.com/right-test-management-tool/> (Accessed 19 February 2023.765Z).

Thoma, L. (2021) *TestRail as a central information point*, Bachelorarbeit, Wiener Neustadt, Ferdinand Porsche FernFH.

List of Figures

Figure 1: Waterfall model (Mohsin Nazir, 2020)	7
Figure 2: Agile values and principles (Mohsin Nazir, 2020)	8
Figure 3: Agile tester mindset (Alpega Group, 2022b)	9
Figure 4: CI/CD process (Nahnsen, 2020)	10
Figure 5: Software test management phases (Pawlak and Poniszewska-Marańda, 2018)	11
Figure 6: Costs of manual and automated testing (Shah, 2019).....	14
Figure 7: Overview TestRail integration with Jenkins (Rede, 2022)	17
Figure 8: Tooling landscape (Thoma, 2021)	23
Figure 9: GitLab schedules	25
Figure 10: Cron syntax (GitLab, 2023)	26
Figure 11: GitLab schedule configuration.....	27
Figure 12: Release process (own illustration)	31
Figure 13: Software testing activities within the different release phases (own illustration)	33
Figure 14: Overview tool interaction (own illustration)	40
Figure 15: Implementation details based on solution approach (own illustration)	53
Figure 16: Central milestone creation via TestRail dashboard	54
Figure 17: Confirmation of milestone creation	56
Figure 18: Release milestones created via central milestone creation	56
Figure 19: Start of release milestone to trigger update of GitLab variable (milestone overview page)	57
Figure 20: Start of release milestone to trigger update of GitLab variable (milestone detail page).....	58
Figure 21: Confirmation after start of release milestone	59
Figure 22: Start of regression milestone to trigger test plan creation (via release milestone detail page)	60
Figure 23: Wizard when starting regression milestone	61
Figure 24: Information message after triggering test plan creation for regression milestone	62

Figure 25: Triggered GitLab pipeline after start of regression milestone.....	63
Figure 26: Information message after triggering test plan creation for smoke milestone	64
Figure 27: Wizard to run all tests of test plan.....	66
Figure 28: Summary after triggering execution of all test cases	68
Figure 29: Extension of wizard to run tests of plan with field execution set	69
Figure 30: Extension of summary after triggering execution of test cases with execution set	70
Figure 31: Wizard to create test plan independent from milestone start	71
Figure 32: Confirmation after triggering test plan creation	72
Figure 33: Relevant test case fields for test case selection depending on scope	73
Figure 34: Manual effort for administrative tasks before optimization (own illustration)	78
Figure 35: Manual effort for administrative tasks after optimization (own illustration)	79
Figure 36: Manual effort per release cycle before and after optimization (own illustration)	81

List of Tables

Table 1: Research objectives actual vs. target	3
Table 2: Principles of agile testing (Alpega Group, 2023b)	18
Table 3: Definition of Done	20
Table 4: Company specific terminology	22
Table 5: Release process - Phases and milestones.....	29
Table 6: Number of test plans per project.....	36
Table 7: Effort of recurring testing activities	37
Table 8: GitLab CI/CD custom environment variables (Thoma, 2021)	41
Table 9: API parameters for test execution	45
Table 10: Definition of Scope	47
Table 11: Mapping between TestRail and GitLab projects	48
Table 12: GitLab variables	49
Table 13: Parameters for regression test plan creation	62
Table 14: Parameters for smoke test plan creation.....	64
Table 15: Field description of wizard when triggering execution of all test cases within plan	67
Table 16: Relevant parameters for the test plan creation.....	74
Table 17: Effort of recurring testing activities before and after optimization	77
Table 18: Manual effort for the testing process in the RPM project with monthly releases	80
Table 19: Review of research objectives	86