MASTER'S THESIS 2023

Improving Feedback Loop by Two-step Continuous Integration

Andreas Erlandsson, Hannes Lantz

ISSN 1650-2884
LU-CS-EX: 2023-13

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2023-13

Improving Feedback Loop by Two-step Continuous Integration

Förbättra Återkoppling genom Tvåstegs Kontinuerlig Integration

Andreas Erlandsson, Hannes Lantz

Improving Feedback Loop by Two-step Continuous Integration

Andreas Erlandsson
tna15aer@student.lu.se

Hannes Lantz
ha7117la-s@student.lu.se

July 6, 2023

Master's thesis work carried out at an undisclosed company.

Supervisors: Lars Bendix, lars.bendix@cs.lth.se
Christian Pendleton, christian@curiousisland.se
Per-Anders Lundblad, pa@curiousisland.se

Examiner: Emelie Engström, emelie.engstrom@cs.lth.se

Abstract

Long feedback times are a big problem for any software developer. Gaining feedback on a recently made change can sometimes take over 15 hours when it is only built and tested during the night. This problem creates the need for developers to work on a number of tasks in parallel. This task switching is time-consuming and hurtful to the performance of the developers. This work, therefore, sets out to try to bring feedback in 15 minutes or less as desired by the developers. The research questions of our work are: **RQ1** - What are the key considerations and practices for designing a suitable pipeline for the case company? **RQ2** & **RQ3** - What strategies for reducing **build** respectively **test** times are suitable for the case company?

The methods used to investigate our research questions are a combination of literature study, interviews, and experimental implementation of a pipeline solution at the case company.

To summarise our results we managed to build an automatic pipeline that enables the developer to gain feedback on changes when needed, instead of waiting for the nightly integration process. The build times for this pipeline were reduced by only building the bare minimum of what was necessary. The test times were reduced drastically which was possible through the use of test selection and parallel test execution. Together these results solve the initial problem of long feedback times. The implemented pipeline brings the developers feedback in less than 15 minutes. Additionally, it also has room for further improvements and added functionality.

Keywords: Continuous Integration, Regression Testing, Test Selection, Developer Feedback Loops, Improving Feedback Loops, Improving Build Pipeline.

Acknowledgements

We would like to thank our dear supervisors, Christian Pendleton, Per-Anders Lundblad, and Lars Bendix that have all been very helpful, supportive and kind to us throughout this project. Especially Lars for putting up with us through long supervision meetings and for all the valuable feedback he has given on our work.

Contents

Chapter 1
Introduction

When developing software in an agile way, receiving fast feedback is a crucial aspect to be able to have continual improvement and an adaptive development process. Without fast feedback, the productivity and efficiency of the development can suffer. A common practice among agile software development projects is therefore to practice continuous integration, where code changes are tested and integrated rapidly, often multiple times per day. Continuous integration of changes makes development easier and faster with smaller incremental changes, this lets developers receive feedback on their changes sooner which increases the pace of development. It also provides the developers with security and confidence that their changes build and test successfully.

This master thesis was performed at a company, which is suffering from long feedback loops where changes are only integrated daily through an extensive nightly build. These long feedback loops cause the developers to start other tasks while waiting for feedback from the previous task. This task switching leads to multiple simultaneous tasks in the developers' heads, which makes them overall less productive [1]. Therefore we want to investigate how to shorten the developer feedback loops and how that would affect the company's development process.

Because of the current long and extensive nightly build and test processes at the company, we also need to find ways to shorten these processes if we want to shorten their feedback loops with faster continuous integration. To be able to do that we first need to construct an integration pipeline that automatically starts on every commit which we can then use to further investigate ways to shorten the build and test times on.

Our research questions are therefore:

RQ1 - What are the key considerations and practices for designing a suitable pipeline for the case company?

RQ2 & **RQ3** - What strategies for reducing **build** respectively **test** times are suitable for the case company?

The methods chosen for investigating our research questions were interviews with developers at the case company, a literature study, and through deeper experimentation of the

processes at the case company. By interviewing developers and other stakeholders as well as investigating the current development processes, we gained knowledge that was used to implement an automatic pipeline that builds and tests changes on every commit instead of every night. To make it fast and practical we also used a literature study to investigate relevant optimisation techniques that we could apply to our pipeline to reduce the test and build times. The effectiveness of our pipeline and optimisations techniques were evaluated mostly through interviews with developers but also through data analysis and observations.

In the next chapter, we will provide the reader with relevant background information about the context, the research questions, and the relevant theory. Chapter three delves into our research method. Then we present our initial analysis that was used to formulate the requirements for our implementation. Then we present our design choices and final implementation. Chapter six presents the results of our research questions. In the seventh chapter, we discuss our results, our reflections, the generalisations of our findings if they are valid, and possible future work. And finally, we present our final conclusions.

Chapter 2
Background

This chapter contains relevant background information to enhance the reader's understanding of the purpose and context of this master thesis project. The chapter starts by presenting the current situation at the case company, followed by the problem statement, research questions, and research method. The final section of this chapter, the theory section, outlines the core theories and concepts that serve as the foundation of this report. This master thesis project is a collaborative effort between the Department of Computer Science at Lund University and an anonymous company, with additional collaboration from Curious Island, who is currently contracted by this undisclosed company.

2.1 Context

This section aims to present the necessary context to understand the underlying motivation behind our research questions. It starts by presenting background information about the company this project was conducted at. Followed by an explanation of the investigated team's current work processes. Lastly, a more detailed explanation of the case company's problem is provided.

2.1.1 Case company

The case company is a multinational company that operates in the transportation sector. The office we conducted our work at works with software development. This thesis focuses on one of their software development teams. The products they develop are strictly regulated, have safety ratings, profound validations of each release, and the products must be of the highest quality with minimal failures.

The company is interested in modernising and developing their ways of working to increase their overall productivity. In our project, we are focusing on only one of the company's products. From this point forward will we refer to the office where we conducted our work as the case company. The case company works in different ways compared to other parts of the same organisation.

2.1.2 Ways of Working

The case company has its own ways of working and release structures. Currently, they deliver a new release of the GP approximately once every year or longer. The development of the company's products follows mostly an Agile way of working with integration of code every night, changes are made by the developers and accumulate during the day and then during the night the products are built and thoroughly tested. This nightly integration, building, and testing can be seen in figure 2.1 below.

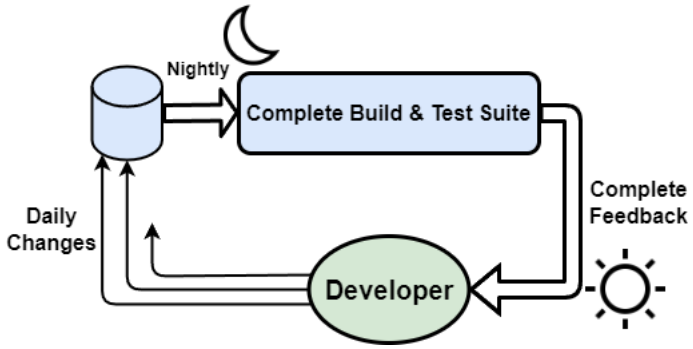


Figure 2.1: Current Development Loop

This ensures that the changes made during the day have not impacted the products in any negative way and they still work as intended. It means that if a faulty behaviour is found during the night it will be investigated the following day. The current process of building

and testing the main branch every night is a long time-consuming process that takes several hours. Therefore it is performed during the night since this allows the process to use all the resources uncontestedly. Another reason is that performing it during the day would not be beneficial for the developers or efficient in its current form, since the process is slow and would then be performed on a shared resource.

They follow the European Committee for Electrotechnical Standardization (CENELEC) standard where the requirements for the products act as the direction of the development and the implementation of the tests. This also means that the developer teams cannot see the source code for the tests and likewise the test teams cannot see the product source code. The reasoning behind this work model is to ensure valid tests that can verify the product correctly to ensure it is of high quality. Since the tests are made to match the requirements the number of tests is ever-growing, currently, there are over 2000 test scripts that are performed for the nightly build of the product. There are no varying severity rankings for the tests as all requirements are seen as necessities.

They work with developing new features and perform maintenance the same way. They work with shared code ownership where on a normal workday the developers have a meeting in the morning, from where they update each other on current individual tasks. There are a handful of developers directly working on this project. Currently, they make approximately one or two changes per developer per day. Each change also requires at least two independent code reviews before being accepted into the nightly build and testing.

2.1.3 Problem Statement

With this background information, we now present the current problem and future wishes of the case company, which led to us investigating this further in our project. The case company identifies that the current development model with the nightly build and test setup leads to long feedback loops for the developers. Meaning that if they make a change it takes a long time until they can get some feedback or verification for that change. Therefore they have to start other tasks while waiting for feedback on the original task. This leads to the developers having multiple simultaneous tasks in their heads at the same time, which makes them feel less productive and more confused [1]. To mitigate this problem the case company wants to shorten these feedback loops as short as possible while still ensuring the same quality of the products. This would lead to more productive development and the developers would be able to focus on the task at hand and get relevant fast feedback. One proposed solution by the case company would be to build and test automatically on every commit instead of every night.

2.2 Research Questions

With the initiating problem clearly formulated, we now need to establish and motivate the research questions. Our goal was to design a faster development pipeline for the developers at the case company to shorten the development feedback loop. As proposed by the case company, the idea is to build and test on every commit, rather than only through the current nightly builds. However, the current nightly build and test process takes several hours, therefore we first needed to find ways to shorten them. Without a faster build and test process, the

feedback loop would be too long with the current process being run on every commit. This would cause delays since the process started from one developer would block other developers from building and testing their changes.

Therefore, our research questions aim to design a more practical pipeline, in order to do so we need to shorten the build and test times. The research questions are summarised as follows:

- **RQ1** What are the key considerations and practices for designing a suitable pipeline for the case company?
- **RQ2** What strategies for reducing build times are suitable for the case company?
- **RQ3** What strategies for reducing test times are suitable for the case company?

One possible way to decrease the test times was to reduce the test suite for every commit to yield faster feedback but give more uncertainty in the verification process. Then during the night, the complete test suite can still be performed to make sure that the product meets all the requirements. Other investigations could include, investing in faster hardware, or removing redundant test cases. We also need to investigate if there are ways we can optimise the build process. With the shorter build and test time we aim to build a prototype solution for new ways of working at the case company that can shorten the feedback loop and thereby increase the development productivity.

Apart from optimising the test and build process, we imagined a two-step continuous integration process as a likely candidate solution that we wanted to investigate further. One short feedback loop where changes can get fast feedback from a smaller more uncertain test suite. This is combined with the current longer feedback loop where all the changes made during the day are built again and tested more thoroughly during the night. An overview of how this combination would work can be seen in Figure 2.2 below. This gives the developers the ability to quickly test their changes and receive some fast feedback without needing to wait for the nightly build to get complete feedback.

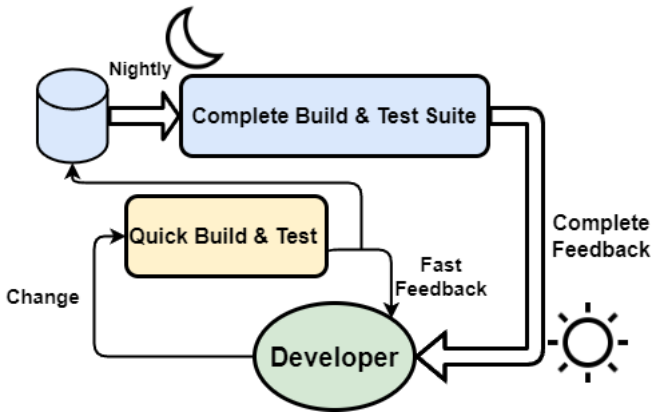


Figure 2.2: Idea of our Prototype Solution

An important aspect for this solution to be beneficial for the developers is to find the sweet-spot of the size of the test suite for the faster loop. A test suite that is too small would not give enough feedback, similarly, a too large test suite would take too long time to execute. This challenge to find the desired sweet-spot is visualised in Figure 2.3 below.

Figure 2.3: Finding the sweet-spot

To be able to see if our findings and our proposed solution would be beneficial for the case company we also needed to perform some kind of evaluation. This would let us see what benefits and improvements our proposed solution could bring to the team's ways of working and development processes. We also needed to collect data on how much we were able to reduce the build and test times. As this data can then be weighed against any potential compromises that were made to bring the test and build times down. Including an analysis of other potential strategies to reduce build and test times that we did not use. Another important point is evaluating if the chosen sweet spot for the size of the short feedback loop's test suite was correct. We also needed to evaluate any potential compromises made there, if they were valid.

2.3 Theory

In this section, we delve into the central theory that serves as the foundation of the report. The purpose of this is twofold; firstly, to provide the reader with a comprehensive understanding of the topics covered in the report, and secondly, to present information on the techniques and research that form the basis of this thesis.

2.3.1 Software Configuration Management

According to Bendix and Ekman [3] Software configuration management (SCM) is a set of processes for managing changes and modifications to software systems during their entire life cycle. SCM activities are traditionally split into four different activities.

- Configuration Identification
- Configuration Control
- Configuration Status Accounting
- Configuration Audits

Configuration Identification is an activity that can be summarised as dividing a system into uniquely identifiable components called configuration items. With each configuration item, we save its characteristics, interfaces, and change history. These configuration items are then often alone or grouped into baselines. In SCM a baseline is an agreed definition of a product

at a point in time, which describes the form, fit, and function. Access to the baseline is tightly controlled to ensure integrity.

Configuration control as from its name is about controlling the changes made to these configuration items. It needs to be made in a formal and controlled way including: evaluation, coordination, approval or disapproval, and implementation of changes.

Configuration Status Accounting is the activity that logs all data about the configuration items throughout the product life-cycle. This gives users the possibility to track the status and implementation of changes.

Configuration Audits handle the process of verifying or validating configuration items automatically. This could be used to assure both management and customers that the requirements of the configuration items are met.

2.3.2 Continuous Integration

The goal of Continuous Integration (CI) is to frequently integrate new code to the mainline and automatically test the integration. Using CI in software development provides many benefits. Mainly through the integration of small changes often, which makes the development easier, faster, and less risky. Additionally, it also leads to more confident developers and a faster pace of development. Teams using CI are also able to discover more bugs than teams who are not using CI. This is because the teams using CI try to integrate more frequently, leading to more and faster testing. Finding more bugs also improves the overall quality of the software[21]. As a guide to implementing CI, Fowler, and Foemmel [8] defined the following ten core CI practices:

1. Maintain a single source repository
2. Automate the build
3. Make your build self-testing
4. Everyone commits to mainline every day
5. Every commit should build the mainline on an integration machine
6. Keep the build fast
7. Test in a clone of the production environment
8. Make it easy for anyone to get the latest executable
9. Ensure that system state and changes are visible
10. Automate deployment

Feedback

Feedback is a word that has been mentioned many times in this report thus far. However, it has not been fully explained and why it is important to the software development process. There are many ways for developers to get feedback: through compiling, building, testing,

reviewing, and more. Continuous feedback helps developers find bugs and improve the quality of their work. It also helps bring a sense of closure to the developer as it allows him or her to know when to feel completely done with a task. With slow feedback loops, the developer has to context swap.

Context swap or switching is when the developer starts other tasks while waiting for feedback from the first task. If the feedback takes a long time to arrive, the developer has already forgotten what he or she had done with the original task. Frequent context switching reduces productivity and makes them less focused on the task at hand [19].

Slow feedback is also harmful to the practice of continuous integration. Slow feedback causes infrequent integration and other practices that are not optimal in CI [5].

Pipeline

In our work, we define a pipeline as an automatic process that contains a sequence of events following each other. Usually, it starts with building the system followed by different testing events. Events can both be entirely in sequence or partially with some events being performed in parallel. The pipeline should also be an automatic process from start to end.

2.3.3 Software Building

Building is a crucial step in software development and should not take too long time. Kent Beck [2] states that build waiting times should optimally be between 2-10 minutes in order to give developers valuable feedback. However Laukkanen and Mäntylä [12] think that there is not enough scientific evidence to back Kent Beck's claim, but they agree that build waiting time should be low because it affects the developer's dedication to CI. Feedback and task completion encourage developers to commit frequently. But complex integration steps and long waiting times discourage developers to commit more. Lower waiting time results in commits to mainline being made more often. This is also true for lower waiting time for test results [12].

The easiest way of improving build times is to acquire more and/or faster hardware. Another way of improving build times is by only rebuilding the parts that have been affected by the change [7].

2.3.4 Software Testing

The aim of Software testing is to validate and verify the behaviour of the artefact and detect failures. Software testing can be performed at different levels, unit, integration, and system testing. Unit or module tests refer to tests that target a specific part of the code or function. Integration tests refer to tests that test that the interface and interaction between multiple components are correct. System tests aim to validate and verify the entire integrated system. Test cases are usually collected in test suites that contain multiple test cases [23][16].

Regression Testing

The aim of regression testing is to make sure that previously validated software still works as intended. As other new features or code is added it could affect previously developed code,

which could generate unintended faults. To detect these kinds of problems the original test cases are therefore rerun automatically every time new features or code is added [22].

White-box and Black-box

In software testing, there are generally two types of methods or perspectives that testing falls under. In White-box testing, the person writing the code for the tests has access to the code of the artefact that is tested. Therefore it can test the inner structures of the code or make sure that every part of a code snippet is executed. This can be useful for example in full code coverage, where we want to execute every line of the code. It can be used at any level but is mostly used for unit or module testing [16].

In Black-box testing the person writing the test is not allowed to see the code of the program being tested. Therefore the tester can only test how it is intended to function and not the inner workings of the function or code. That makes it more suitable for testing the end-user experience, as they would not have access to the code while using the program. Black-box tests are usually written to match the requirements for a specific function to verify its functionality [23][16].

Smoke Testing

Smoke testing can mean a lot of different things to different people, in our case, it is a type of testing that aims to quickly determine if a new build is stable or not before being sent to more rigid testing. Usually, it is a determined subset of the normal test suite that aims to quickly see if the build has any obvious flaws. According to Chauhan [4] the importance of smoke testing are: that it saves time since we do not have to wait for the entire test suite to be executed before we find out that basic functionality is still working as intended. It also saves effort since with the smoke test in place we do not have to restart the entire test suite again to be sure our fix to the original problem was solved. Avoiding these time and effort problems leads to a reduction of costs for development. It also mitigates integration problems and increases the overall quality. Further, it also gives management an easier way to assess the development progress.

Automated Testing

In software testing, there are repetitive steps like test execution that can be automated. To be able to automate the tests we need a separate program or tool that can automate those processes for us. According to Umar [20] the benefits of automating parts of software testing are: improved accuracy and time to find bugs, saving time and effort making testing more efficient, increased test coverage from multiple parallel testing tools, and creating repeatable automated test scripts. The drawbacks that Umar lists are: choosing the right tool is a difficult task, it is also sometimes expensive with costly tools, and a knowledgeable tester is required to write the automation test scripts and understand the test tools.

Chapter 3
Research Method

In order to effectively address our research questions, it is crucial to develop a comprehensive research plan and select the most appropriate methods to gather the necessary information. This chapter, therefore, aims to give the reader an overview of the various research activities that were used to investigate our research questions. Each section delves deeper into each of the chosen method's purpose and motivation. To investigate the initiating problem further and increase our knowledge of the current development processes and setup we used interviews, a literature study, and further investigation of the current nightly build setup. After implementing our pipeline prototype we wanted to evaluate how it affects the team's development processes, this was mostly done through a second set of interviews but also through some observations and analysis of analytical data.

3.1 Research Overview

Our research can be split into two parts, the first part being an investigation into the initial problem, the development processes, and the needs of the potential users. The knowledge gained from the first part was then used to create our pipeline implementation. After having implemented our prototype the second part aimed to evaluate it and to see how it affected the developers and if it solved the initial problem. This can be visualised in Figure 3.1, where the methods used are listed from left to right in the order they were executed. The first three which are coloured blue were performed as the first step before implementing our prototype and the fifth and final is the evaluation that was performed after implementing our pipeline.

Figure 3.1: Research timeline

3.2 Literature Study

First, we performed a literature study to see if there were existing methods or techniques that we could have used to decrease the build and test times. The study was also used to deepen our knowledge of the subject area as a preparation for the interviews.

We decided that it would be beneficial to perform the literature study before the interviews as it would give us the possibility to use the insights we gained from the literature in the discussions during the interviews. If we would do it in the other order instead we could have used insights from the interviews in our literature study which would have saved us time since we could then narrow the search area somewhat. But we decided that doing the literature study first would be more beneficial as we could use the literature findings in the interview and also we could return to our literature findings after the interviews and see what was the most valuable and suitable.

To perform our literature study we used LUBsearch and Google Scholar to find suitable articles. The keywords we used in our search were: Continuous Integration, Test Selection, Regression Testing, Developer Feedback Loops, Improving Developer Feedback Loops, and Improving Build pipeline. The keywords Test Selection and Regression Testing were chosen to help us find suitable methods to reduce our test times. The remaining keywords were chosen to identify ways to improve the case company's ways of working, shorten the feedback loop, and discover other design choices for our pipeline but also if possible to find methods to improve the build times.

3.3 Interviews

Interviews with developers and other stakeholders were an important activity that had multiple purposes. Firstly it gave us a better understanding of the developer's problems and needs.

Secondly, it gave us a better understanding of the development processes and the underlying motives behind them. Thirdly it was used to identify possible bottlenecks in the team's software development, building, or testing processes that could be addressed or investigated further.

To investigate ways to improve the case company's ways of working we first needed to get a better understanding of how they currently work and why. We required this deeper understanding of the team's current ways of working to identify points of improvement and how they could be performed to best suit them. As any identified motives for the case company's ways of working could be important when designing our pipeline. Since those motives might still be valid, although perhaps can be acknowledged in a different way. We decided that the best way to gain this information would be through a set of interviews. As this would greatly help us in understanding the problems and identify possible bottlenecks in the software development, building, or testing processes that could be addressed or investigated further.

To identify potential areas for improvement or opportunities for optimisation in our pipeline implementation, it was important to gain a better understanding of the current build and test setup. Therefore, we conducted interviews to gather information and insights that could help us reduce build and test times.

We decided on using interviews as it would also give us the possibility to easily ask follow-up questions to be able to find the root causes of the case company's identified problems. We decided the best idea would be to interview our subjects in pairs. With the idea that this would encourage and yield better discussions between the interviewees since they could have different views about the root causes.

Having them in pairs would also give them the possibility to hear each other's views on a subject which could help them further process their own views and discuss them together. Interviewing them in pairs would also be less time intensive compared to single interviews with the same amount of people, although harder to plan since both of them would have to be available at the same time.

Our original idea was to interview first only pairs of developers and then a separate set of interviews with higher stakeholders, including the Test Lead, configuration manager (CM), system engineer, and project leaders. But as there were very few developers working on the product in question we decided that it would be a better idea to pair one developer with one higher stakeholder. This would lead to more interviews and pairing a developer with a higher stakeholder could give us information from different perspectives and help us understand their reasonings, which could lead to better discussions overall. With the interviewees different backgrounds, it would also provide us with information about the problems from different points of view and from different technical levels. As there are more stakeholders than developers, we decided it would be in our interest to also plan an additional interview with only stakeholders.

Below are two tables that give the reader an overview of the structure of our interviews and which roles the interviewees had. In table 3.1 we present the developer interviews where the first interview was with only the software lead and was used as a way to test our interview guide. Table 3.2 present our interviews with higher stakeholders where we used a different set of questions that were more suited for their roles and responsibilities. The interview guides used for these interviews can be found in appendix A.

Another idea could have been to increase the number of participants to three per inter-

Interview1	Interview2	Interview3	Interview4
Software Lead	System Engineer	Test Lead	CM
-	Developer1	Developer2	Developer3

Table 3.1: Developer interviews

Interview5	Interview6
Project Planner	V&V Team-lead
-	Head of Engineering

Table 3.2: Higher stakeholder interviews

view, this would give us the possibility to have at least one developer in each interview. We decided that keeping the limit at two per interview would be enough as it otherwise could lead to participants not taking enough part in the discussion and by keeping the limit at two we could possibly extract more information from them.

We wanted to use open-ended questions to encourage discussion but we decided to prepare an interview guide that would help us steer the discussions and keep the discussion in the different interviews on the same subject so that we could easier compare the results to each other. Before the interviews, the interviewees were given a shorter version of the questions to be able to prepare for the interviews. The aim of this was to give them an idea of the nature of the interview as well as give them some time to contemplate the questions by themselves beforehand. As this yielded better discussions and provided us with more meaningful interviews.

An alternative to using interviews could instead have been to prepare a questionnaire survey that could be sent to the developers and stakeholders. We decided this would not be a good idea as it would not be as easy to ask follow-up questions. We also wanted to focus on good discussions which would not be possible with a questionnaire compared to an interview. A questionnaire would be less time intensive and possibly have a larger spread but the complexity of the questions would have to be lower to be suitable to use. A questionnaire would be better for yes/no questions, but we felt the need for deeper discussions that would not be possible from a questionnaire, as we wanted to find and understand the root causes behind the reported problems.

3.4 Investigation of current Processes

In order to understand the processes at the case company in greater detail, we conducted an in-depth investigation. Primarily this was made through further discussions with the CM and the Test Lead but also from our own experimenting. The information gathered from the previous interviews was used as a starting point for this investigation. The focus of this investigation was to understand how the current integration, building, and testing processes function, to be able to reuse parts of it in our implementation. But we also performed a further investigation into the test logs, to be able to identify important test cases that we wanted to include in our smaller test suite.

3.5 Pipeline Creation

The implementation was conducted in an agile manner. The approach was to start with creating a simple solution which then was improved over a series of iterations. This allowed us to have a more structured way of creating our prototype compared to if we were to try and optimise and include every requirement directly.

3.6 Pipeline Evaluation

To evaluate our pipeline implementation we first let developers use our prototype pipeline in their daily work and then used a second set of interviews after that to see how it affected the development processes and their thoughts about the implementation. In combination with the interviews, we also used observations and analysed analytical data to further strengthen and confirm our findings.

Chapter 4
Problem Evaluation

The purpose of this chapter is to provide the reader with an overview of the reasonings behind our design decisions. To achieve this, we will begin by presenting the key insights and analysis derived from the interviews and literature review. We will then look into our investigation of the processes at the case company, including an analysis of the test logs, to provide additional information. The culmination of these findings and analyses will be condensed into requirements that will guide the design of our solution.

4.1 Literature study

The aim of the Literature study was to identify ways to reduce both test and build times as well as find potential strategies that could be useful when designing our pipeline. Additionally, the literature study also served as a means of increasing our knowledge of the subject area in preparation for the interviews.

4.1.1 Ways of Improving Test Times

Using the literature study we wanted to find ways to shorten the test times.

Running every available test case has a high cost on hardware resources. One way of lowering that cost is selecting a subset of those test cases. Selections should be based on what test cases are most relevant for the change [11]. Basically, if change A only changes code in module B then only executing test cases relevant to module B will result in faster feedback at the cost of lower quality assurance. This would only be feasible if the test cases in question have low coupling and are not dependent on other modules.

Paralleling test case execution instead of executing one by one can save a lot of time [6]. For example, executing 10 test cases, each taking 1 second. Running them one by one will take 10 seconds whilst running them at the same time in parallel will take only 1 second.

When limiting the amount of time that will be dedicated to testing it is important to find a trade-off between choosing test cases that would likely detect faults and the number of total tests executed [13]. This can be done by using algorithms like ROCKET (Prioritisation for Continuous Regression Testing) [13] that requires as input: The regression test suite, historic failure status for each test, historical execution time for each test, and the upper bound of execution time allowed. According to Marijan [13] using ROCKET algorithm compared to manual or random test prioritisation would yield an increase in efficiency and a faster time to detect faults in a regression test suite.

The most common way to speed up testing is by reducing the number of tests executed by choosing an optimal subset of test cases. Stratis [18] proposes techniques that could reduce the amount of time spent on executing test suites without reducing the number of tests. They include data transformation which improves test suite compilation time and a series of test case scheduling algorithms that produce time-efficient test case execution orders.

Another solution to decrease the amount of time required to execute a test suite is an increase in computation hardware. With faster or more powerful hardware capabilities we can execute tests faster or if performed in parallel we can execute more tests in parallel simultaneously [6].

4.1.2 Continuous Integration

To design our pipeline we wanted to first analyse the continuous integration principles to see what would be suitable for our implementation.

Laukkanen and Mäntylä [12] claimed that abiding by the 10 principles shown in 2.3.2 would the speed of software development feedback increase and software quality improve. The principles that are of interest in our work and that we would like to include in our experiment are "Automate the build", "Make it self-testing", "Everyone commits to mainline

every day", "Every commit should build the mainline on an integration machine" and to "keep the build fast". We will later see in 4.2 that these maps pretty well to the overall goal that the case company wants to achieve and summarise the foundation we want to achieve with our pipeline. The remaining principles stated are not of interest to focus on in our project. The first bullet on the list, "Maintain a single source repository", is already satisfied at the case company. Another point, "Testing every commit in a production-like environment", seems unnecessary as that is what they have the nightly build for. Similarly the remaining 3 principles, "Make it easy for anyone to get the latest executable", "Ensure that system state and changes are visible" and "Automate deployment", are also more in line with the nightly build.

4.1.3 Ways of Improving Build Times

Using the literature study we wanted to find ways to shorten the build times. For build times optimisations we only used other sources than academic literature as it was hard to find useful sources. Many of the strategies used in industry were easier to find using forums or articles.

Improving build times is an important step to decrease the overall build and test times. Many ways of improving build times depend on the development language, platform, and project-specific optimisations. But there are some general strategies to improve build times for most types of projects.

According to Fernandez [7] there are ways to improve the build times in pipelines, firstly to increase the available hardware for the process. By increasing the amount of available hardware we can perform the building steps faster, this is a type of vertical scaling. Another approach can instead be to horizontally scale by paralleling the process over multiple machines. This is only feasible if the build process is able to be run in parallel.

Another approach used by Microsoft MSBuild [15] is incremental builds, where only what was changed is rebuilt. Meaning if a developer made some changes to one module, only this module will need to be rebuilt. This approach makes the building process faster but can be more unstable compared to a full rebuild. It is also possible to incrementally checkout changes but rebuild everything. This is faster than a clean checkout combined with a full rebuild. Although it is slower than an incremental build it provides more stability.

4.2 Interviews

This section aims to present important findings from our interviews, combined with analysis of the interviewees problems and then the identified requirements for our solution that will be the basis of the design.

4.2.1 Findings and Analysis

The purpose of this section is to present to the reader our findings from the interviews, and relevant analyses that will be used to formulate requirements for our solution.

Current Development Process

One aspect of the interviews was to give us a better understanding of the current development processes. Our findings can be summarised in the following 4 points, which are further explained below.

- Requirement on following CENELEC standards in their development process.
- Handles safety classified products, which includes extra validation and security steps.
- Long feedback loops for developers.
- Developers usually make around 1-2 code changes per day.

The developers report that their current ways of working are a result of them wanting to work in a more agile way while also keeping the more traditional requirements they have. They need to follow the CENELEC standard, where system engineers first create requirements, and then both the developers and testers independently write code and tests to match them.

Generally, the developers claim that the way of working for a new feature goes from first receiving the new requirements for a feature. And then developing code for this feature. The code change is then reviewed by two independent reviewers before being merged to the mainline and tested in the nightly build. The testing team simultaneously writes new tests for this requirement that are added to the nightly build when they are finished. The next day after testing the changes during the night, the developer receives feedback on the change made. The code is then altered if any faults were detected and further altered after future nightly builds until all tests pass. The developers report that they have altered this process somewhat and are now according to them working more in rotations where they do not have to perform all the steps in this long and strict order and can therefore shorten this lengthy process somewhat. This development process from the developer's perspective can be summarised in the figure 4.1 below.

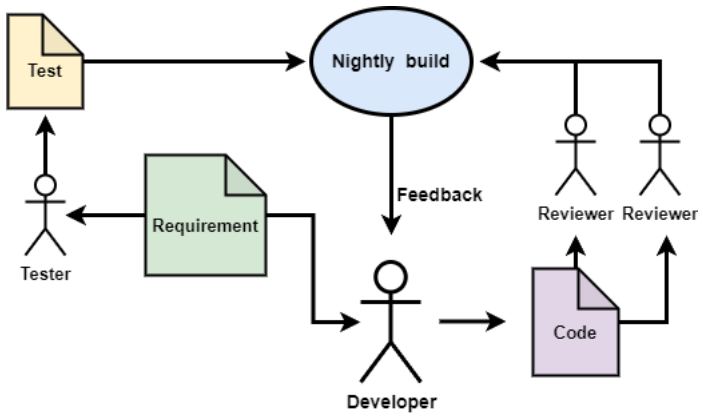


Figure 4.1: Current Development process

Since they are working with safety classified products they also need to perform a lengthy analysis when faults are detected, leading to longer maintenance phases of their products.

With the current setup, developers argue that feedback is slow and developers are feeling that the time waiting for feedback is too long. The reason for this is reported to be because of the current setup with build and tests only being performed once per night, leading to feedback only being received in mornings. Therefore we can see a need for a solution with a faster feedback loop.

Developers reportedly made on average one or two changes per day, and in combination with the code reviews, they argue that it leads to few high quality changes per day. We could see that the mentality of the developers was careful, they were very keen on pushing high quality changes with minimal failure rate. Since they knew they would not gain any feedback on their changes until the next day they were more inclined to only make one or two larger changes per day instead of multiple small ones. The developers said if they were to get some feedback before putting up a change for review they probably would push changes more often.

Identified Problems

In the interviews, there were a couple of organisational and development problems identified. Because of the nature of interviews and our limited knowledge about the product, the problems reported are far from the whole truth but can be an indication of where problems might exist. These findings can be summarised in the following 5 points, which are further explained below.

- The developing team is not always in sync with the testing team.
- Limited amount of resources limits development and time to refactoring.
- An instability in the test environment might lead to faulty test results.
- Inconsistent manual testing processes, preferably should be automatic.
- Dependencies on certain key roles.

A major problem that developers highlight is that the testers and developers are not always in sync, the developers can develop code for a feature according to the requirements but need to wait until the testers have written test scripts for that feature to gain meaningful feedback. If that time is long the developers will start working on other features and can forget what they had done earlier when the feedback eventually arrives. The reason this problem exists is due to limited resources, the testing team is currently too few and cannot create test scripts fast enough.

The limited amount of resources is also a problem for the developers as they feel a lack of time to properly investigate fault reports, and work on old problems. The developers also argue that the limited resources could also lead to a low priority on refactoring old designs or test scripts. According to the Test Lead, approximately half of the current test suite could probably be removed if investigated properly, as many test cases overlap. Some of the test scripts are also considered old and outdated, and a lack of priority for maintenance of them is voiced.

Although it is said that testing is not hard, it can sometimes be hard to only test the part of the product that has been altered with a code change. There could also be an instability in the test environment that sometimes gives false test results. According to the interviewed

developers, if a fault is discovered from a nightly build and test, it is usually solved in time for the next nightly build, but sometimes it could take up to a week for it to be completely solved depending on the complexity or priority of the problem.

There is a general rule set up in the development team that developers should test their change themselves before sending it to the mainline but there is no exact requirement that decides how this should be done, therefore the level of testing could vary from developer to developer. An old and simple type of smoke test exists and is used by some of the developers to quickly test their changes. It consists of 3 test scripts that should test basic functionality. But this process is manual, therefore it would be beneficial if those scripts or a similar and better process could be performed automatically on every change, providing developers with better, faster, and easier feedback. The developers believe this would give them more confidence in their changes and find faults earlier instead of needing to wait until the nightly build.

The higher stakeholders also voiced concern about being too dependent on certain key roles. An example they give is the Configuration Manager (CM), who currently is a key individual with a deep understanding of how the build system is managed. This dependence on a single person creates a potential risk, as their absence could disrupt the product building process. As a result, there is a request to explore options that would enhance the clarity and comprehensibility of the setup, reducing the reliance on any one individual.

Case Company Preferences

From our interviews, we wanted to gain information about the developer's preferences when it comes to the amount of time elapsed before receiving feedback. We also wanted to know the higher stakeholder's opinions related to possibly increasing hardware resources.

When asking our interviewees about how fast they currently gain feedback and what time would be ideal according to them, we got very mixed results. Some felt that feedback currently could be gained in 20 minutes using the previously discussed smoke test. And around a day for feedback from the nightly build. On the other hand, some felt that it at best takes one day to gain relevant feedback, and for new features or requirements, it could take up to two weeks if the tests for the requirements in question were not completed yet. One thing that they all had in common was the perceived usefulness and importance of the independent code reviews. They claim that it gives very good feedback in a reasonable amount of time. This time would usually be around 1-2 days. Optimally they claim to want feedback after anything from 1 minute to 30 minutes depending on the quality of the feedback and the number of tests executed. But preferably under 15 min with as many relevant tests executed as possible. For a smaller test suite with faster feedback, they would like tests to be either dynamically chosen or a standard set of tests that guarantees that the standard functionality works. It could also be a combination where a standard set of test scripts are combined with dynamically chosen tests that test especially the modules or impacted code where the change is taking place. Another developer also requested the possibility to be able to choose some tests themselves to be added manually with the rest of the already decided tests for execution.

We were interested in finding out if the higher stakeholders would be more interested in increasing the amount of hardware to be able to build and execute tests faster or decreasing the number of tests for faster execution in our solution. They were not interested in any new hardware as they want to eventually move to a cloud-based solution instead. As this would remove the burden from them on hardware maintenance. With a cloud based solution, they

could also potentially execute all the test scripts in parallel. But before taking that step they are interested in using a smaller test suite that with relevant test cases chosen can be executed fast to provide the developers with faster feedback.

4.3 Investigation of Company Processes

In order to understand the processes at the case company in greater detail, we conducted an in-depth investigation. Mostly by sitting down with the CM and the Test Lead but also through our own experimentation. The information gathered from the previous interviews and discussions with relevant personnel was used as a starting point, but the further insight was necessary to properly design and implement a feasible prototype solution.

4.3.1 Current Integration Process

The team and project we are looking at have the nightly integration setup. We wanted to investigate it further to get a better idea about their current problems and how we could use or take inspiration from it when creating our pipeline.

The current setup involves a review process with two independent reviewers. When a change is made it is sent to two independent reviewers to review the change. When approved by both reviewers the change is then merged to the mainline. After the workday is over, currently on a timestamp at 17, the building process starts. After building and preparing the test environment the testing starts, currently, they have over 2000 test scripts that are executed on 90 VMs in parallel. After testing, the test log is prepared to be viewed by the team in the morning.

4.3.2 Building Processes

The normal way to build the product requires building three different platforms. These platforms are the developer platform and then two additional ones that are required for running the system on the real product hardware. Building the developer platform is enough to be able to execute the test scripts. But for more extensive testing with the real product hardware, all three are required. The developer platform should be possible to build locally and individually.

But in reality that was not as easy as we had been told and some modifications had to be made first. After solving this problem we could now shorten the build time in our solution as we no longer had to build all three platforms to be able to execute the tests we wanted. We did not need the remaining two platforms as we were not interested in performing extensive testing in real world situations in our fast feedback pipeline. Measuring the time it takes to build only the developer platform is between 2-4 minutes. Compared to around 10 minutes if building all three. This two-minute difference in build time range is a small mystery. We cannot get any detailed logs when we make the developer platform and what little output we do get when comparing them it looks the same. One possible explanation that also seems to be the most likely one, is that since the server is a shared resource, all the developers are utilising it at the same time, meaning it could be the number of available resources at the time of building.

4.3.3 Testing Processes

The case company has a dedicated test team that writes subsystem tests that match each requirement. These are written like black-box tests as the dedicated test team has no insight into the code of the functions and only follows the requirements set. As new functions and features are added new tests are written and added to the test suite. For the case company, this is currently not a problem as they execute them during the night anyways, when there is plenty of time to spend. They also have integration tests that are more complex and broad.

Currently, there are a little over 2000 test scripts in total. They use the nightly build and tests as a form of regression testing where every test case is performed every night. Also, since the test cases are executed during the night, where there is plenty of time to spend, tests are not always optimised to use as little time as possible. Therefore an approach to decrease the execution time of test cases could be to refactor redundant test cases and rewrite poorly time-optimised written test cases. Refactoring and maintenance of existing test cases are done based on priority towards other activities in the test team's backlog.

In the nightly build, they have 90 virtual machines (VMs) that execute the entire test suite in parallel. Without this parallelisation it would take approximately two entire days to execute the whole suite. In the nightly build, excluding any other steps and only looking at the execution of the tests it takes around 30 minutes. But the entire nightly process of building the entire product, preparing the test environment, copying over all the necessary files, and rerunning the tests takes a few hours.

4.3.4 Test Log Analysis

The aim of the test log analysis was primarily to find test cases that were failing more often than others. It was also used to give us a better idea about the average amount of tests that usually failed during the night.

The reason that we wanted to find test cases that failed often, was to possibly include them in our test suite. A test script that failed often could indicate that it was testing parts of the code that had a high failure rate. Tests that fail often would be ideal to include in our faster test suite as it would increase the chance that it could provide the developer with feedback earlier than in the night build. One thing to keep in mind is that a test with a high failure rate could also be a flaky test that is not reliable. Finding tests with a high failure rate would therefore require a discussion with the Test Lead. Avoiding flaky tests in our test suite is important since it needs to produce reliable feedback, otherwise, developers would not trust the results and avoid the benefits of fast feedback.

Unfortunately, we could only find test logs for the last 10 nightly builds. This gave us a much smaller window of information that would not necessarily be representative of the whole picture. This led us to look at the test logs multiple times with at least 10 days time apart to gain a wider picture. What we found then was that approximately 15 out of the 2000 test scripts failed on a regular basis. There was no nightly build that was completely clean without any failed tests. In the 10-day windows, we could see that many of the failed test scripts were failing for multiple nights while 9 were consistently failing every time.

We decided this could be worth investigating and asked the Test Lead about the tests that seemed to consistently be failing. Unfortunately, this did not lead to anything as the Test Lead explained that they were actually not failing but being reported as failing anyways.

This was because of instability in the test environment and poorly written test scripts. This was an interesting find by itself, but we decided because of this instability we could not trust the test logs. Therefore we did not want to use any of the failing or flaky test scripts in our test suite.

If they had kept test logs for a longer period or did not have problems with falsely failing test scripts, this activity could have provided us with data over the tests that could have been very beneficial to our solution. Finding test scripts that were usually failing could have been a good idea to prioritise in a smaller test suite to faster find faults in the change. Executing them early in our test suite would have led to faster fault detection since they had a higher risk of failing compared to other test scripts.

The information that we gained from the test logs was that there are some problems with some of the test scripts being unstable and in need of maintenance. Unfortunately, this meant we could not use the test logs to find tests to prioritise in our test suite. But it also gave us an idea about the average amount of faults detected per night.

As this number was quite low, if you separate the tests that were falsely failing consistently it was around 0 to 6. This means that the changes made during the day were either very few or had high quality with low failure rate.

4.4 Requirements

This section aims to present an analysis of the most important findings, from both the literature study and the interviews, and formulate them into requirements. These requirements will later be used for the design of our proposed solution.

4.4.1 Analysis

The aim of this section is to give the reader an understanding of the reasoning behind the formulation of our requirements.

The central goal of the solution is to provide developers with faster feedback than they currently are from the nightly build. From the interviews, we see that their desire for a time span of the feedback ranges from 1 to 30 minutes, but optimally under 15 minutes. Compared to the nightly build, any process that can be performed and give developers some feedback is already an improvement. The desired time span will affect the number of test cases that can be executed. Here we will need to make a decision when implementing to find the optimal sweet spot with the number of tests executed versus the amount of time spent. With the goal of meeting their preferences of time with as good feedback as possible.

From our investigation of the building process, we found that it takes 2-4 minutes to build the developer platform which then leaves us potentially with up to 11-13 minutes after building to execute tests and still be under the 15 minute mark that the developers requested. This means that it could be possible to decrease the total time for feedback even lower, for example by using a smaller set of tests to execute. According to Kent Beck[2] if we could get the total time under 10 minutes that would be more beneficial to the developers.

We would most likely need to decrease the number of tests executed compared to the nightly build to meet this time criteria. In the case of a smaller test suite, the interviewees request if possible a dynamically chosen set of tests that focus on the modules that have

changed. This would only be feasible if the functions that are tested have low coupling and do not impact multiple modules as this otherwise would make it impossible to separate the test scripts that should be executed. If it would not be possible to dynamically choose tests, they request that the subset is chosen with care to test basic functionality.

Automation of the process is important to provide the developers with feedback as fast as possible without any additional manual work. It would also provide the team with homogeneous testing instead of relying on developers to test their own code properly. It would also encourage them to do it more often as it takes less energy from the developer.

Currently, they only make a few changes per developer per day, every change is also reviewed by two independent reviewers before being accepted into the nightly build. It would not be beneficial for our pipeline to include these code reviews since it would not lead to any faster feedback for the developers with a manual step blocking the automatic fast iterations. The code reviews could instead be performed when larger changes are sent from the faster iteration loop to the longer nightly build.

We will also be constrained by the hardware limitations put forward by the higher stakeholders. This means that we will not investigate ways to increase hardware to be able to execute even more tests or to execute them even faster.

4.4.2 Identified Requirements

This section provides the reader with an outline of the requirements for our proposed solution, as identified through our analysis. The requirements have been established based on the central goal of providing faster feedback to developers. Four requirements were identified:

1. Minimal Feedback Time: The most critical requirement for the solution is to minimise the time it takes for the developers to receive feedback. Based on the interviews, a feedback time shorter than 15 minutes is desired, with an emphasis on obtaining feedback as quickly as possible.

2. Automation: To save time and increase ease of use, it is important that the solution is fully automated.

3. Relevant Test Scripts: If the number of tests would be reduced to increase the test time the feedback should be based on the results of relevant test scripts. The selection of relevant test scripts could be dynamic, based on the change in question, or decided through alternative means. The stakeholders also express the desire for the possibility to manually choose test scripts on occasion, either as a standalone option or in combination with a static or dynamic selection process.

4. Hardware Limitations: The solution must be designed and implemented within the constraints of the current hardware available.

Chapter 5
Design and Implementation

In this chapter, we explain our design and the implementation process of our pipeline. This gives the reader an understanding of our design choices and implementation. This is important since the implementation will later be used by developers and other stakeholders at the case company. Understanding the design reasoning and the final implementation gives the reader the necessary context to understand the subsequent analyses and results. The design phase of the research project was approached in a step-by-step manner, with the aim of gradually building a comprehensive solution that addresses the identified problems and requirements. Our approach was to start with something that was relatively simple and easy to implement and then reevaluate the design at the end of each step to assess its effectiveness and make necessary adjustments. This iterative design process allowed us to gradually refine our solution and ensure that it was both practical and efficient, while also allowing us to respond quickly to any unexpected challenges or obstacles that arose along the way. Through careful analysis and consideration, we were able to develop a design that meets the requirements that were previously identified in Chapter 4.

5.1 Software Tools

Before diving into the topic at hand, it is important to first understand the role of Jenkins and Gerrit in the process. As they are software tools that the case company already uses and that we will use for implementing our design.

5.1.1 Jenkins

Jenkins is an open-source automation server [10]. It is a useful tool for automating parts of software development related to, building, testing, deploying, and CI/CD. Jenkins has support for plugins that allow its user to tailor the tool to better fit whatever need the user has. One such plugin that is very useful for this project is Gerrit Trigger which allows Gerrit to trigger builds inside Jenkins when a "patch set" is created or updated. We will use Jenkins to automate and run each step of our pipeline. As the case company already uses Jenkins to automate the nightly build it seemed like the best option for us was to use the same tool.

5.1.2 Gerrit

Gerrit is a free code collaboration tool [9]. It allows developers to give feedback on each other's modifications to the collaborative code base. The main form of feedback is approval or rejection of the modification, however, comments can also be made. This is done using a web browser. Gerrit is closely integrated with Git. The case company uses Gerrit as the hub for their collaborative repository for the project that we focus on. Figure 5.1 shows an example of how Gerrit can be integrated with Jenkins and what we will implement can basically be the arrow that loops around Jenkins.

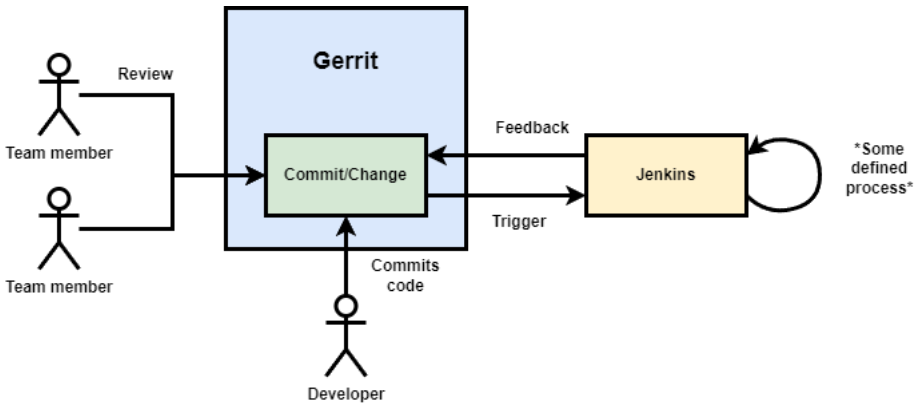


Figure 5.1: Example on how Gerrit can be integrated with Jenkins

5.2 Possible Solutions

In this section, we will examine each requirement listed in 4.4.2 and present potential solutions that can help meet each requirement effectively. The purpose of this section is to provide a comprehensive overview of the available options. The options are presented in Table 5.1.

ID	Requirement	Possible solutions
R1	**Minimal Feedback Time**: The most critical requirement for the solution is to minimise the time it takes for the developers to receive feedback. Based on the interviews, a feedback time shorter than 15 minutes is desired, with an emphasis on obtaining feedback as quickly as possible.	Only build what is necessary to build. Execute tests in parallel. Only test a subset of test cases.
R2	**Automation**: To save time and increase ease of use, it is important that the solution is fully automated.	Trigger an automation tool to run each necessary step of the build and test process for each commit.
R3	**Relevant Test Scripts**: If the number of tests would be reduced to improve the test time. The feedback should be based on the results of relevant test scripts. The selection of relevant test scripts could be dynamic, based on the change in question, or decided through alternative means. The stakeholders also express the desire for the possibility to manually choose test scripts on occasion, either as a standalone option or in combination with a static or dynamic selection process.	Allow user to manually specify a subset of tests when committing. Make the automated process decide a suitable subset relevant for each commit.
R4	**Hardware Limitations**: The solution must be designed and implemented within the constraints of the current hardware available.	Only utilising the currently available hardware.

Table 5.1: List of possible solutions for each requirements

5.3 Design Iteration 1

The goal of our first iteration was to investigate RQ1, the pipeline design. We did this by creating a minimal viable product, that would work from end to end. This would let us understand the process from end to end and identify further problems with the pipeline design.

Our priority was to make sure that the main steps were functional. The main steps of our pipeline are depicted in Figure 5.2.

While we did not put too much emphasis on optimising every step, to be able to have a functional prototype faster. We did already incorporate a few elements that we knew would make the pipeline faster than the current nightly build. We looked at both strategies to improve build and test times to start investigating both RQ2 and RQ3. These optimisations allowed us to early streamline the process and get a better understanding of what further improvements could be made in the future. It also made the entire pipeline faster which

made it easier to run and test, therefore it could already yield small user feedback from the case company.

Since this pipeline would be automatically started when a commit was made, compared to the nightly build that starts on a time stamp. It could already provide users with some faster feedback than the nightly build. Although it is very slow and not time- or resource-efficient and further optimisations would be required for it to be an efficient solution.

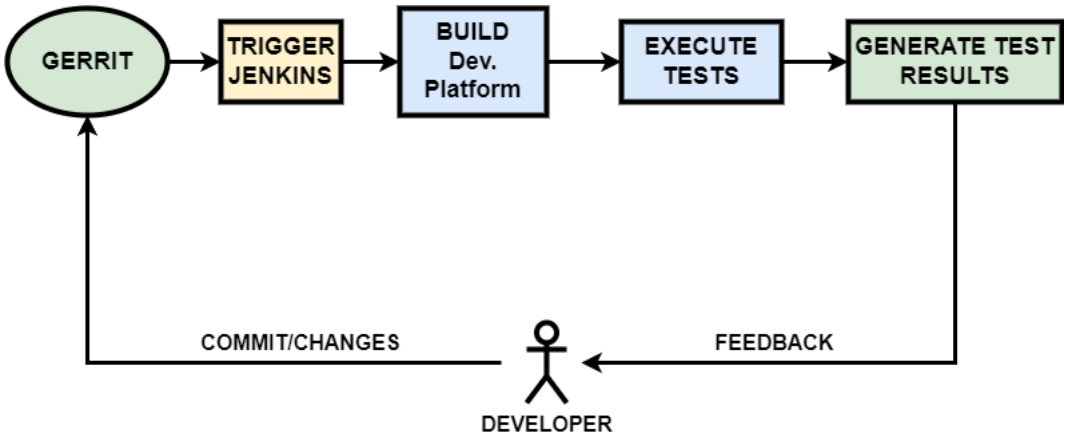


Figure 5.2: First iteration design of our pipeline

5.3.1 Building

Although RQ1 was our main focus of the first iteration we decided to start looking at strategies to improve the build times (RQ2) already to make the pipeline more functional. In 4.3.2 we talk about the Building process at the case company. There it is highlighted that by just building the developer platform is it still possible to run all the tests. Since the remaining platforms are only required to run the code on the real system hardware or the case company's real system test simulator. As these capabilities are not in our interest in the pipeline, building only the developer platform is an efficient way for us to build faster. As we will save time and still be able to execute all the test cases we want. The real life tests are used for a separate more careful and precise testing environment before releasing products over to a later stage in the product life cycle and are not part of the normal development process.

5.3.2 Test Selection

We decided to also start optimising the test times to improve the overall end-to-end times for our pipeline. This lets us start investigating test times (RQ3). Executing all the 2000 test cases would not be a time efficient way, as it would take more than an entire day to execute those tests on a single commit. That would make our pipeline perform worse than the nightly build. Therefore we thought about some sort of test selection that could fit well. As this was our first iteration we did not implement a way of selecting test cases either manually or from a more advanced selection from the most relevant test cases depending on the changes in the commit. Instead through collaboration with the Test Lead who helped us by formulating a list of 45 test cases, that would function as a good baseline as it would cover and test basic

functionality. To those 45 we added the 3 tests that developers usually run before committing a change. These 3 tests are the current baseline or quality gate and were added so developers would not need to run them manually before committing.

5.3.3 Evaluation

In our first iteration, we mainly wanted to evaluate our pipeline process, RQ1. As this would let us find points of improvement and also identify possibilities in further iterations to improve on the build and test times for RQ2 and RQ3. After the pipeline was functional and the initial optimisations were done for RQ2 and RQ3, we took a look at the overall performance and also let some developers try it out. We uploaded a new commit to Gerrit and it triggered our script automatically as intended. The building process took around 2 minutes to complete compared to 22 minutes in the nightly build with all the platforms. Then executing all the 48 test cases took 1 hour and 7 minutes compared to 30 minutes for the 2000 test scripts by the nightly build. As we can see the nightly testing is much more efficient than ours, this drastic difference in test times comes from our testing being sequential while the nightly build uses 90 VMs to parallelise the tests.

This is still a long time to wait for feedback after every commit, as we are aiming for under 15 minutes. Another problem is that our Jenkins job can only run one job at a time. Technically it would be possible to change this, however, due to the hardware limitations this is a problem. If we allowed running two or more jobs coexistent, they would each get fewer resources and thus take a longer time to finish and our pipeline job is not the only job utilising Jenkins. That means it is possible for even higher resource demands. That is why we thought a queue would be the best option as then we ensure that each job gets as many resources as available and if some other task not related to our pipeline would run on Jenkins it would not put too much restraint on us and vice versa.

The feedback that we got from the developers that tried it was that it was a nice start but over 1 hour for every commit is way too slow to be used practically. Therefore in future iterations, we want to focus on improving the test times (RQ3), as this was the longest part of the pipeline.

5.4 Design Iteration 2

After evaluating the results of iteration 1 we identified our sequential test execution to be the main bottleneck. Therefore our focus on the second iteration was mainly on RQ3, to find a way to shorten the test time. From 3.1.1 we know that executing tests in parallel can drastically reduce test times. Thus the focus of iteration 2 became to enable executing tests in parallel. Paralleling tests is a functionality already used in the nightly build, Therefore we wanted to investigate how the nightly paralleling test process is setup. Our new design idea can be seen in figure 5.3.

5.4.1 The path to Parallel Execution

In order to figure out the easiest way to do parallel test execution we looked at the nightly build. We found that the way the nightly build manages to achieve parallel execution is by

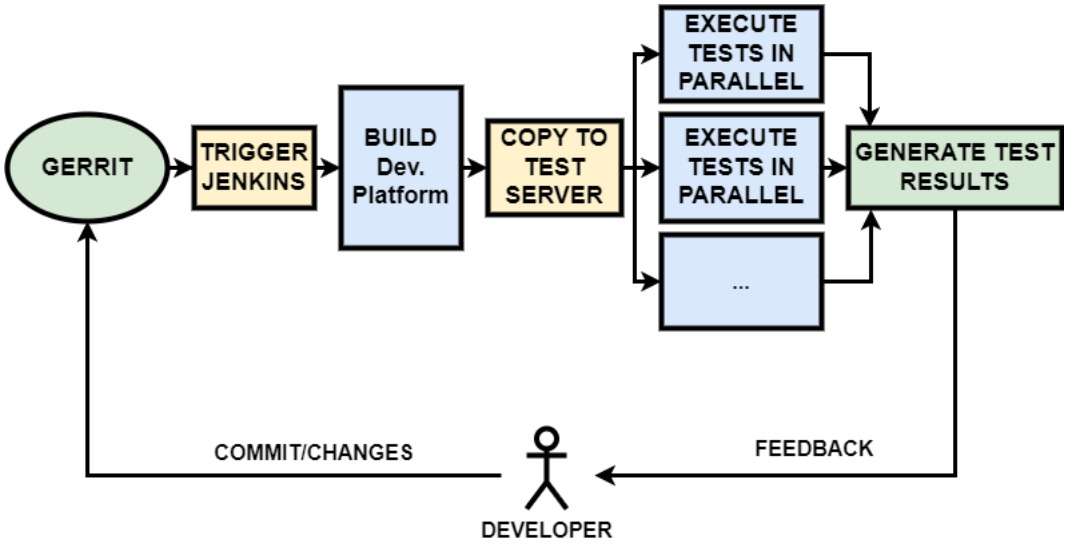


Figure 5.3: Second iteration design of our pipeline

copying everything it has built and all extra dependencies over to a dedicated "test-server". On this test-server it has access to 90 VMs that it uses to parallelise the test execution. The task of copying everything is a lengthy process. In the nightly build it takes an average of 50-60 minutes. Unfortunately, it is not possible to build directly on the test-server, therefore we would need to make a copy as well.

Since the copying process is so long it would not be a faster solution to copy and then parallelise compared to the previous sequential test execution. Therefore we needed to find ways to optimise the copying process to be able to parallelise our test execution efficiently. We came up with the idea that the developer platform shares dependencies with the nightly build. Theoretically, it should be possible for us to reuse parts of what the nightly build copied over previously. If we would manage to do that it would save us valuable time. We discussed this possibility with the project's CM to understand exactly what from our build we need to copy to the test-server and what we could reuse from the nightly build. We found out that there are 4 directories that we need to copy over to get all the binaries that would have changed from our builds.

Copying these 4 directories from our build to the test server takes around 15-20 minutes, which is still quite a long process just for copying as it does not leave us any time to execute tests and still be in a reasonable time. However, it is not every binary in these directories that actually updates from build to build. This means that if we could identify and transfer only the binaries that have changed since the latest nightly build we could shorten the copying process even more. There is a command for Linux that does just that and it is called rsync. This method of copying allows us to transfer our copy in around 11 minutes. However, our first copy for each day still takes 15-20 minutes as the test-server is cleaned at the end of each day before the new nightly build is copied. Although this time is still quite long it allows us now to parallelise our test execution.

Paralleling and executing the tests were pretty straightforward. First, we needed to set up the test environment, then we could reuse the same script that the nightly build uses for dividing up our list of tests from iteration 1, and then order which VM to run which test. The average time for executing our tests this way was 6 minutes.

In order to copy the 4 directories to the test server, a new downstream job in Jenkins was created. This caused a loss of connection to Gerrit, leading to an inability to provide direct test feedback back to the original job. This would not be a problem if the Gerrit trigger plugin that we use in order to start our pipeline was updated, as in newer versions it is capable to be used inside a Jenkins job that can handle multiple steps. The reason the plugin was not updated is a lack of maintenance and someone being responsible at the case company, since multiple plugins are very old, updating some of them can also break the dependencies for other plugins causing the Jenkins to stop working.

To overcome this issue without updating the plugins, we attempted to pass the email of the author of the commit to the downstream job, which would then send an email upon completion or failure. To enable this, a new plugin was needed for Jenkins to send emails. The attempted installation of the required plugin failed due to outdated dependencies on other plugins, this resulted in a forced update that caused Jenkins to go offline and fail to restart. As this caused disruption to the case company's normal development our supervisors said that it would probably be best if we avoid installing or updating any Jenkins plugins to avoid further disruptions and we agreed.

5.4.2 Evaluation

In our second iteration, we wanted to find ways to shorten the test times, RQ3. This resulted in a parallelisation of our testing process. This provided us with a significant reduction in test time, from over an hour to less than 6 minutes. However, this came with two major drawbacks. The required copy process to the test server became a new bottleneck and a new downstream job in Jenkins was created to facilitate the copy, which also caused a loss of connection to Gerrit. This resulted in an inability to provide direct feedback to Gerrit, breaking the full circle process. Although we managed to improve the test times drastically which was our goal for the second iteration, our solution for improving test times introduced a new bottleneck in the testing process that needs to be solved, but most importantly we also broke the end-to-end function of our pipeline meaning that in future iterations we needed to focus again on RQ1, the pipeline design. Since in its current form, it would not be usable, which would not let us yield any user data from the case company.

5.5 Design Iteration 3

In the third iteration, we aimed to address the issue of closing the feedback loop for the developer. As this was a very important step for the developers to be able to use it in practice. Since even if we could optimise build or test times further the pipeline was not able to provide the developers with any feedback in its current state. Therefore our focus was mainly again on RQ1.

But we also wanted to investigate ways to optimise the copying process further as this was now the biggest bottleneck in our pipeline. One day we accidentally realised that someone had recently updated the Gerrit trigger plugin to a version that was now beneficial for us. As we discussed before with this plugin updated, it allows us now to have all the steps of our pipeline within the same Jenkins job, which in turn allows us to send feedback back to Gerrit and notify the developer. This change also provided an added bonus in the form of a user

interface element in Jenkins that displayed the real-time status of each step in our pipeline in an intuitive manner. Figure 5.4 illustrates how Jenkins presents each step in the pipeline and 5.5 shows a crop out of the test result view. Each row in figure 5.4 is a commit/change and the colours, green, yellow, and red represent success, unstable, and failure respectively.

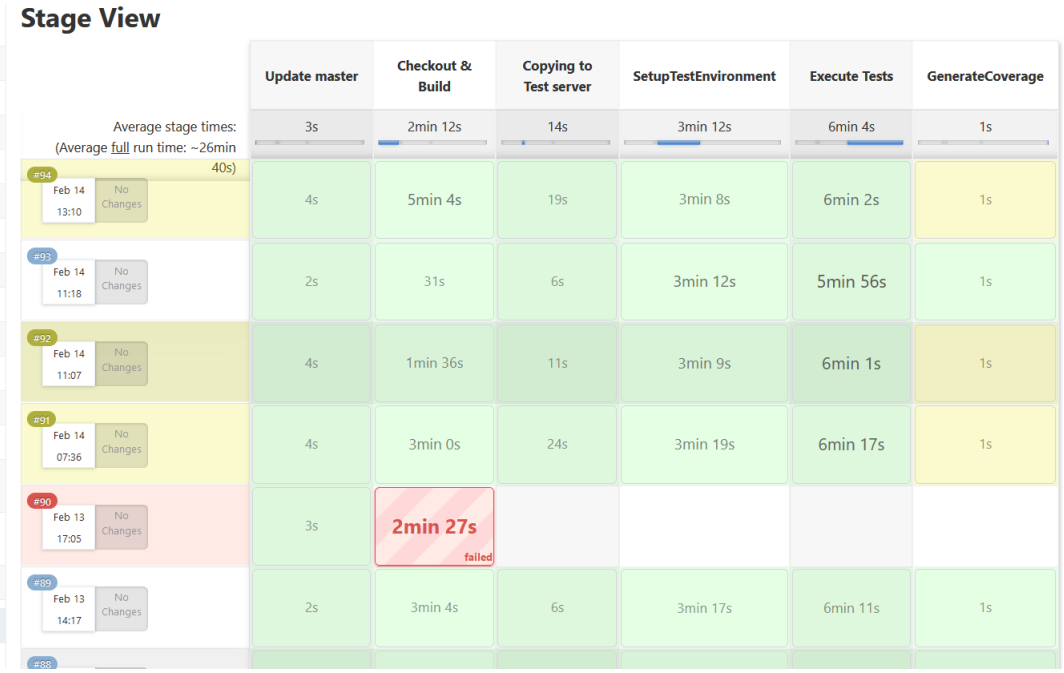


Figure 5.4: Screenshoot of Jenkins pipeline

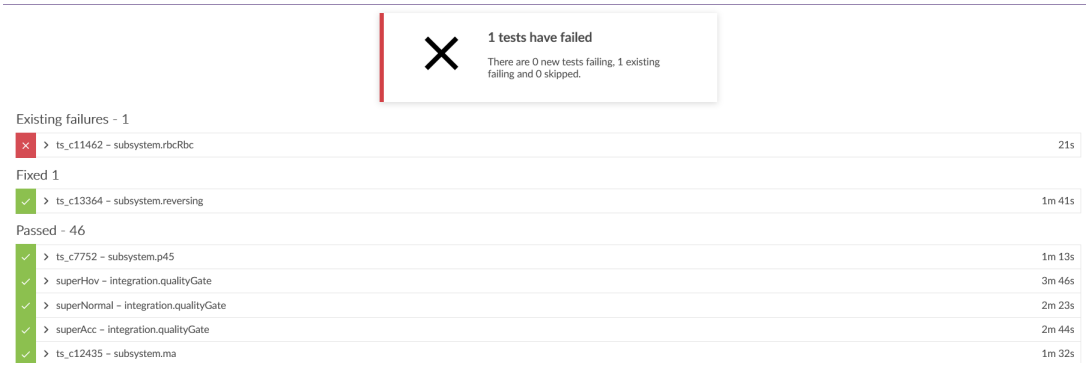


Figure 5.5: Screenshoot of a Jenkins view that displays test results

We also sought to address the new bottleneck introduced in our second iteration, to be able to shorten the test process for RQ3. With the bottleneck being the copying of files over to the test server. Since it now takes more than 50% of the total time of our pipeline it was something that definitely needed to be solved for the entire pipeline process to be able to finish in under 15 minutes. After trial and error and asking around for helpful ideas, we found that the problem was that we synced from a network drive. By syncing over Secure Shell (SSH) instead, the transfer time was down to only 5-11 seconds. The first copy of the day is still the slowest but it averages around 15 seconds now. This brings the total time of the pipeline down to only 11 minutes.

5.5.1 Evaluation

The main focus in the third iteration was on making sure that we could close the feedback loop again. As this would let developers use our pipeline in practice, which was very important for our RQ1 as we otherwise would not be able to collect any user data.

We also successfully investigated ways to improve the copying process as it halted our earlier success in improving the test times for RQ3. The successful resolution of these bottlenecks in our third iteration has resulted in a significant improvement in the end-to-end time for our pipeline. Combined with our earlier iterations we can see that we now are under the 15 minute mark that we were aiming for with an approximate time of 11 minutes. This was possible through our investigations in the build and test times for RQ2 and RQ3. In the first iteration, we improved the build time and formed a simple test selection. In the second iteration, we drastically improved the test times by paralleling the execution. And in the third iteration, we solved the issue of a long copy process.

As we can see that the end-to-end time now meets the requirement of under 15 minutes that we formulated by interviewing the developers. Since they wanted it to be fast enough to be able to take a short break and "grab a coffee" while the pipeline finishes executing. We now feel like our prototype is fast and stable enough to be tested on a broader set of developers. This will give us more feedback on our solution as well as give us user data that we can analyse further.

Since we only use 48 out of the 90 VMs currently we could increase the number of tests executed to a total of 90 at least and still have the same execution time. As long as we do not add tests that are longer than our current longest test. Since we use a static list of tests this could easily be extended but would require further discussions with the Test Lead to find more tests that would be suitable to use. It would only be beneficial if the test list was extended to at least 90 tests and still stay within the same time frame. However, we decided not to investigate possibilities for this yet due to time restraints and we needed the developers to start using the pipeline as soon as possible in order to start gaining user feedback and data.

We also wanted to investigate the possibility to execute the entire nightly test suite. As this would give the developers even better feedback. Executing the 2000 tests on the 90 VMs in our pipeline took 29 minutes and 42 seconds. The total pipeline time for this run was 33 minutes. With a simple comparisons between executing the 48 tests compared to the 2000 we can see that:

$$\frac{2000\ Tests}{30min} \approx 67\ Tests/min \tag{5.1}$$

$$\frac{48\ Tests}{6min} = 8\ Tests/min \tag{5.2}$$

From this, it seems likely that we should be able to execute a lot more tests than we currently do and still be within the 6 minutes that we currently are. However, in reality, there is quite a difference in time between all the tests. Some tests take a couple of seconds and some a couple of minutes, with the longest in our pipeline taking 4 minutes. This brings the question of why does it take our pipeline 6 minutes to finish the test step when the longest test only takes 4 minutes? The simple answer is that the Jenkins scheduler for the VMs is doing a bad job.

5.5.2 Minor Changes

During the evaluation time for the last iteration, we made a minor change to our pipeline. If a tester uploaded a commit containing changes to test cases that would also trigger our pipeline to build and execute tests on that build. However, that became an issue for the tester as they had to wait for Jenkins to give irrelevant feedback to their commit. So when a commit only contains test cases it does not make a new build. Instead, it executes the test cases within the commit on the last stable build. That way testers can get relevant feedback.

5.6 Design Summary

This section will summarise the design highlights from each iteration. In iteration 1, the goal was to create a minimal viable product with a functional pipeline, as this would let us investigate RQ1 further but also would serve as a foundation on that we could investigate RQ2 and RQ3. This pipeline had several steps including building and test selection. In the first iteration, we also started finding ways to shorten build times for RQ2. **We optimised the building process by only building the developer platform.** Another early optimisation was on the test times for RQ3. A **test selection was performed by creating a list of 45 test cases** with the Test Lead together with the 3 test cases that developers usually run manually before a commit. Evaluation of the first iteration showed that the pipeline was too slow, with a build time of only 2 minutes but with a lengthy test execution time of up to 70 minutes in total.

In iteration 2, the focus was on shortening the long testing times for the pipeline to be more practical and to investigate our RQ3. This was done by parallel execution of tests since this led to us being able to shorten the lengthy test times, which was identified as the bottleneck in iteration 1. The pipeline design was updated based on the method used by the nightly build, which executes tests in parallel. The 4 required build directories were copied to the test server, which took an average of 11 minutes, and the rest was reused from what the nightly build had already transferred over. The **test execution time was reduced to 6 minutes by using** the test server's **parallel test execution** capability. Now the next bottleneck became transferring our required files over to the test server and we could no longer close the feedback loop.

Iteration 3 solved the problem with the feedback loop by containing all our steps inside the same process in Jenkins. This was an important step to be able to use our pipeline in further investigations in RQ1. The problem with transferring speed from the build server to the test server was also fixed. Instead of 11 minutes, it now takes an average time of 10 seconds. **The total time it takes from end to end in the pipeline is now 11 minutes, with a build time of around 2 minutes and a test time of around 6 minutes, the remaining 3 minutes are mostly used for setting up the test environment**.

5.7 Implementation Summary

This section will summarise the highlights from each iteration's implementation phase. In iteration 1 we set up a simple **pipeline that started when a developer made a commit through**

Git. The commit was then sent to Gerrit which **automatically started** our **Jenkins** job. In the Jenkins job, we first checked out the change from Gerrit and then **built only the developer platform**. Then we **executed our test selection of 48 tests on this build**. The test result was then able to be viewed in Jenkins by the developer who made the change. The pipeline from start to end took around 70 minutes in total.

In iteration 2 we introduced new steps in our previous pipeline. Firstly we introduced a process that would copy the build to a dedicated test server. This **copy process** was **optimised with** the Linux command **rsync**, as this **lets us transfer only the binaries that have changed** since the last tested build. On this test server, we had access to 90 VMs that we parallelised our test execution on. The copying process was possible by introducing the creation of another downstream Jenkins job. This second job caused a disconnect from start to end of our pipeline, this led to the final downstream job not being able to connect back to Gerrit. This would not be a problem if we could do all the steps in one Jenkins job. However, since the Jenkins plugins used were outdated we could not create jobs with multiple steps. As a workaround, we tried to pass the email of the commit creator to the downstream job as this would let us send the test results back to the developer that way instead. But this did not work as the plugin required to pass the email was missing and trying to update the Jenkins plugins did not work.

In iteration 3 we solved the problems from iteration 2 with the passing of the test results back to the creator of the commit by using one Jenkins job for all the steps. This was now possible as someone had now updated the Jenkins plugins that we required to set up a Jenkins job that can handle multiple steps on different nodes. The previous copying over rsync was very slow. We found that **copying over SSH instead of** over the **NFT network drive increased the speed significantly**. This was a huge optimisation. Figure 5.6 shows how the time spent on the major steps of our pipeline has changed between each iteration.

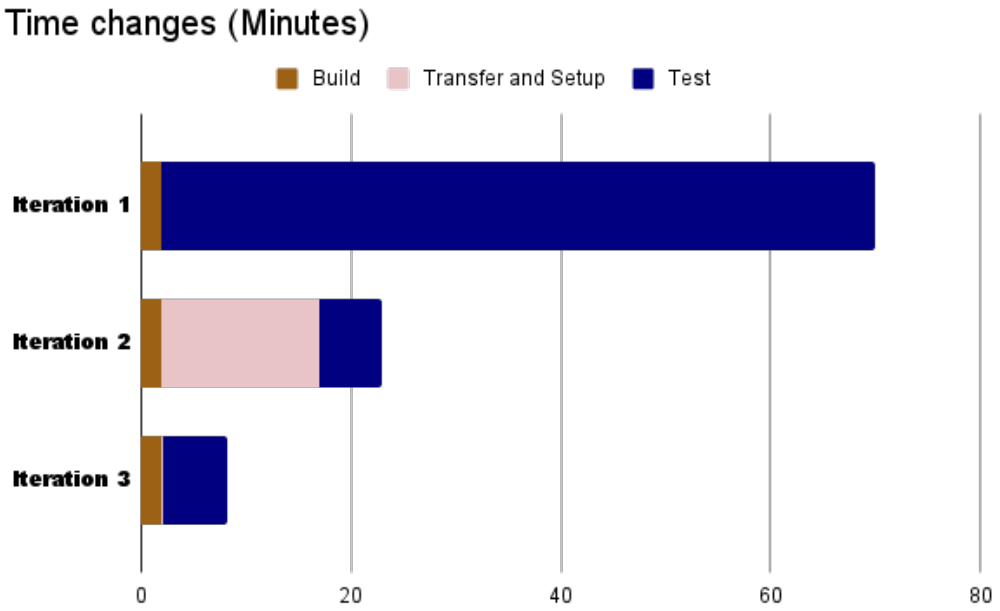


Figure 5.6: Time changes between each iteration.

Chapter 6
Results

The purpose of this chapter is to present our investigative findings and analysis, providing necessary information to understand our reasonings in the subsequent discussion. We begin by explaining our motivation for examining RQ1 using our finished pipeline implementation. Outlining the information we aimed to find and what methods we used to acquire them. Then we present our results and findings from our evaluation and how it connects to RQ1. Lastly, we summarise our previous findings for RQ2 and RQ3, the build times, and test times. The ways in which we optimised these times, by how much, and if any compromises were made.

6.1 RQ1 - Pipeline Design

The purpose of this section is twofold. Firstly, to provide motivation and clarification regarding the methods used in evaluating our pipeline implementation with the developers at the case company. Secondly, to present the findings resulting from this evaluation. The latter is crucial for comprehending subsequent discussions.

6.1.1 Method

To evaluate our implementation we used mainly interviews with developers but also through some observations and analysing analytical data.

Our objective was to have the case company's developers and other stakeholders integrate our pipeline into their daily development process, as it would provide us with valuable insights into its benefits and potential impact on the team's workflow. Since our implementation starts automatically on every commit and reports back with build and test results, the developers can easily use the feedback provided in their daily work. To ensure the effectiveness of our investigation, we devised a plan outlining the specific information we seek, the reasons why, and the methods we will employ to acquire it.

The first method we used were a second set of interviews. Similarly to the first set of interviews, this enabled us to engage in more extensive and detailed discussions with the users of our implementation, resulting in deeper insights. Additionally, interviews were also an appropriate method, as they could provide unexpected or previously unconsidered findings.

In the interviews, we wanted to investigate if the users perceived any benefits from the faster feedback loop. And if that has caused them to make changes to the way they work, for example by committing more often before sending the change for peer review. We also investigated whether users felt that faster feedback makes them more productive and efficient and which parts of the pipeline they believed should be developed further in the future. Furthermore, we used the interviews to evaluate the effectiveness of our pipeline implementation in addressing the initial problem of a long feedback loop, which led to task switching while waiting for feedback. Avoiding task switching by receiving feedback faster would increase the productivity of developers as it keeps the focus on a single task at a time.

In addition to the interviews, we planned to perform observations of developers, to gain a deeper understanding of how the developers interact with our pipeline and how it affects the development process. Firstly, we were interested in observing any changes in behaviour, such as whether users are making commits more frequently. Secondly, we were interested in seeing how they used the feedback from our pipeline and how it affected the team's productivity. This data was then used in combination with the interviews to draw stronger conclusions about how our pipeline has improved the developer's ways of working and if our implementation is an effective solution to the initiating problem.

To further complement the interviews and observations, we planned on gathering additional analytical data. As this would let us find actual trends that could then be compared with the perceived benefits or trends that we gathered from the interviews or saw in our observations.

Using Gerrit, we gathered data on the number of commits made per day and compared it with the amount that was made earlier, before they started using our implementation. This lets us identify whether the implementation has resulted in an increase in the number of

commits made per day. This data was then used in combination with the interviews and observations to further strengthen any claims made about the benefits of our implementation.

Another problem we wanted to investigate through a combination of the three methods was if our pipeline detects problems that could be solved before the changes enter the nightly build. And approximately the amount of faults that were detected in the nightly build that our pipeline has missed. Since both these findings are important to evaluate the actual benefits of our implementation.

In addition to assessing the impact of our implementation on the developers at the case company, we intended to explore any additional features or requirements they requested. This information aided us in identifying potential areas for future work and enhancing the implementation. We intended to gather this information through the previously mentioned interviews and observations.

Instead of using interviews an alternative method for gathering our results could have been to conduct surveys. The surveys would be less time intensive to execute and reach a broader set of developers. But considering there is only a handful of developers using our pipeline, the time required for performing interviews is not that much larger than for surveys. Using interviews we were also able to gain more in-depth answers and further reflect on the answers together with the interviewees. In the interviews, we could also easier ask follow-up questions and clarify responses which would be harder through a survey.

6.1.2 Results

The results and findings from our investigations using our pipeline implementation are important to understand subsequent reasonings in the discussion. We used three methods for gathering our findings: interviews with users of our implementation, observations of the users when they were using it, and collecting analytical data of their usage patterns.

One factor that negatively impacted our results from the pipeline evaluation is the development's current phase. While using our implementation they were in the end phases of the team's product cycle. This led to much fewer commits with code changes and the users focus was on producing documentation and similar activities. The lower amount of commits gave us a smaller sample size to investigate, and a lower commit intensity also gave the developers fewer possibilities to use our implementation. This factor needs to be taken into consideration when analysing our results.

Interviews

Our main method of evaluating our pipeline implementation was through interviews with developers and stakeholders that were affected by and used our pipeline in their daily work.

We were interested in seeing what perceived benefits the developers could see when using our implementation in their daily work. The biggest benefit that everyone interviewed agreed on, was the fast build times. As our builds were stable and fast compared to manual building, it gave the developers the security and confirmation that the changes they made and pushed were working and that it would not break the nightly build. Since it was an automatic and uniform process it also gave them the security that other changes made by other developers had passed through the pipeline. Furthermore, it also provides more security for the reviewers, since they now can be sure that the change they are reviewing has successfully

passed through the pipeline and therefore builds correctly. This takes some pressure off the reviewer to find misspellings or other small mistakes that can cause a build failure in the nightly build. Which lets them save time in the reviewing process.

By the process being automatic the developers also reported that it saves time by being faster than an equivalent manual building and testing process. Because of the automatic nature of the process, they also do not have to perform the multiple manual steps required to build and test and can spend the short time waiting for the pipeline to finish, on other small tasks or taking a quick break. They were also satisfied with the total time for the pipeline.

One concern that was voiced by everyone is the unstable test environment, they voiced a concern that the test environment in our implementation seems to be more unstable than the nightly build. This makes it harder to use the testing part of the pipeline, with its flaky nature with tests sometimes failing for no reason other than the unstable environment makes it harder to trust and use the test results. The problem with this instability of the test environment is an important problem that the case company should investigate as it causes problem in our pipeline, tests that crash increase the need for re-triggers which takes up further time, and it can cause confusion in distinguishing actual faults from the failed tests. But most importantly it reduces the trust in the testing, which makes it not used and the benefits it could give diminishes. The cause of the instability is unknown, after some discussions with the Test Lead it could possibly be that the resources are not as free during the day compared to the night and it, therefore, causes more problems during daily use.

The developers also see some potential ways that our implementation can be improved to suit all their needs. Firstly as previously mentioned they want to make it more stable as this would make the test results much more trustworthy and usable. Currently, multiple jobs are also queued as the implementation can only execute one job at a time. Therefore they request the possibility to run multiple jobs in parallel as it would make it more scalable with more developers using it simultaneously.

They also see the benefits that a smarter test selection can bring. They had some different opinions on the different options, some preferred random or manual tests to be added to the test selection at the commit, others would like to extend the static selection with more quality tests, and others would instead preferably see a full test suite instead of a selection. But they were not interested in increasing the test times to make this possible, instead, they would prefer further optimisation or increasing hardware to be able to execute more tests in the same time as now or even faster if possible.

Additionally, they have some smaller general improvements they would like to see to increase the ease of use and their workflow productivity even further. This includes the addition of static test analysis tools, to only generate documentation for changes to comments, easier to find and view the change in Gerrit, easier to get a list of failed tests if it is very long, accessing the builds for longer without them getting overwritten to make it easier for reviewers.

The benefits reported from just successfully building automatically were something that we did not expect to be so appreciated. We believed the most important benefit would be from gaining feedback faster from the test results. But with our test selection being not optimal and the test environment being so unstable they could only see the potential from the faster testing but not experience it. Therefore we believe that after more work is done with improving the test part of the pipeline the developers would then be able to perceive further benefits related to feedback from testing.

Observations

By doing some observations of how the developers use our implementation we wanted to see how they were interacting with our pipeline.

Looking at data from our pipeline inside Jenkins reveals some information but not much. Based on what we have seen it seems like it is 50/50 if all test cases pass or if some fail due to the unstable nature of the environment and the tests. Figure 5.4 shows three unstable runs (marked yellow) that should have passed but failed due to the environment. One of the developers tested our pipeline by committing code that would not be able to build. The pipeline did fail to build and reported back to Gerrit that the build failed.

Analytical Data

We wanted to use some data from the tests to investigate how the failure rates differ between our pipeline and the nightly build to be able to see the effectiveness of the testing.

Unfortunately, this was not as easy as we had thought. The reasons why we could not do this were the low rate of commits, the difficulty to track the tests in the nightly build, and the unstable test environment as it was difficult to know which tests actually had failed.

This is unfortunate as this metric would be important to strengthen any benefits from test feedback discovered from the interviews. But since the developers reported in the interviews that they could not use the test results in their current form because of the instability, we see this as potential possibility to investigate further when the test environment is more stable. As this would make it easier for the developers to see the benefits of faster feedback and to investigate how the test selection detects failures compared to the nightly build.

6.2 RQ2 - Build Times

This section aims to provide the reader with a summary of our findings reported in Chapter 5 related to RQ2. This section does not present any new information, but it incorporates a discussion concerning the previously presented findings.

Our pipeline's build time is approximately ten times faster than the nightly build's build time. The nightly build requires 22 minutes to build all three available platforms, whereas we discovered that building only the developer platform sufficed for our objective of building and testing. The average build time for the developer platform using the build tool make is two minutes, and occasionally, it is even quicker since make can recognise if a section does not require rebuilding. Our approach to building and the resulting build times remained constant throughout all design and implementation iterations.

The building process is hard to optimise further. If we want to make it even faster it would probably take some heavy code changes and changes to the whole product structure. Something that we do not have the experience and time to do. However, shaving off approximately 20 minutes and staying within the hardware limitations is a huge improvement.

6.3 RQ3 - Test Times

The purpose of this section is to offer a condensed overview of our Chapter 5 findings regarding RQ3. It should be noted that this section does not introduce any new information, but instead includes a discussion of the findings already presented.

Our pipeline started off with a subset of 48 tests from the available 2000 tests. Those 48 became the baseline for our pipeline. Before our pipeline, developers usually ran 3 test scripts. These 3 are included in our baseline. In the first iteration where all tests were executed in sequence, and it took around 68 minutes. It was then changed to execute the 48 tests in parallel. This change brought the total time for executing all 48 tests to an average of 6 minutes. A huge difference. Due to time restraints, we had to exclude manual test selection and also exclude adding more test cases to the fixed selection of tests that are currently used. If this compromise had not been made it would have strengthened the quality insurance of the feedback from the pipeline.

The first area where we think improvements could be made is in the test selection. The current static test selection could be further improved upon with a more dynamic selection instead. A dynamic selection could choose test cases from the modules where the change took place or chose tests depending on other factors such as coverage, time from the latest execution, or failure rate. Using a dynamic test selection with the same amount of tests as the static selection would provide users with more quality feedback while still being executed in roughly the same amount of time. A smarter and more precise selection could likewise be used to reduce the required test times further, while still providing users with the same accuracy of feedback.

A noteworthy observation concerning the hardware limitations is as follows: If, for example, there were three times the amount of VMs on the test server, it would be feasible to execute all 2000 tests in roughly 10 minutes. In our pipeline that would bring the total time of building and testing all 2000 tests to around 15 minutes. Running all tests within that time frame would provide developers with the same level of assurance as the nightly build currently provides. It would represent a significant improvement and may even question the need for the nightly build. As then that procedure would also be a lot faster. Perhaps this would prompt the practice of building and testing all platforms after each integration to the mainline, which is more in line with the principles of continuous integration than what is presently followed.

Chapter 7
Discussion and Related Work

This chapter serves several purposes. Firstly, it aims to provide the reader with our reflections on the entire work, how the results matched against our initial assumptions, and an analysis of the methods used. Secondly, it aims to analyse the generalisation of our findings, if and in what situations our findings can be applied. Thirdly, to analyse the validity of our claims, and if our methods and data collected can provide valid results. Fourthly, to relate our findings to other works, how our results compare, and what other methods we could potentially have used. Finally to provide the reader with our input for possible future work, how our implementation could be further improved, and what further research could be done with more time.

7.1 Reflections

In this section, we provide a reflective analysis of our work. Here, we delve into the key takeaways, lessons learned, and what we would like to have done if we had more time.

This has been a bumpy ride with obstacles around every corner. We anticipated it to be a challenging task. However, we did not foresee the amount of time that it took to understand the organisation and the systems. But after getting to know our way around it became reasonable that it took some time.

We are proud that we managed to create a working pipeline in the real production setting. In the beginning, it seemed we would not be able to do that due to the Jenkins being poorly maintained and key plugins for our work being outdated. It feels good that we managed it and that the total time is below the 15 minutes that the developers requested. Based on the received feedback it is safe to say that it will continue to be used.

If there would not have been any hardware limitations. It would have been interesting if we then were able to get our hands on some newer hardware than they currently have. Because then we could have compared the performance of our pipeline on the new versus the old hardware. But also explore what new possibilities new hardware would open up. Potentially it could have made it possible to execute a lot more tests in the same duration of time or to reduce the time to execute tests even further. This would have made it possible to see what further benefits an improved and faster testing step in our pipeline could have yielded for the case company.

It has been brought up earlier that we did not have time to implement some sort of dynamic or smart test selection. Implementing build and parallel test execution in our pipeline was a steeper climb than we first imagined. The main obstacle we see for constructing a better test selection is that the test suite is not organised in such a way that it would enable dynamic test selection easily. Getting it organised would take time and a lot of domain knowledge. We believe it would be beneficial for the case company to invest further time into this. There are a lot of capable employees with enough domain knowledge to make it happen faster than it would take two thesis students with a fixed time budget.

There will always be unexpected problems, we will now bring up some that we encountered in the hope of facilitating anyone looking into conducting a similar endeavour. We had some problems with the general IT environment at the case company. We have explained the situation with the poorly maintained Jenkins server, which if it was updated more regularly would probably have saved us around four weeks of time. There was a new firewall installation which caused a few side effects which limited access to things. During our recent attempt to collect the final results from the developers, we encountered an issue with the LDAP server. Unfortunately, this server refused access to anyone trying to log in to either Gerrit or Jenkins, causing an unexpected setback. As a consequence, some developers did not get time to fully test our pipeline. This was a further setback to our final evaluation which already had a problem with the development life cycle being in a sub-optimal phase for us.

Furthermore, the overall system environment had its own set of problems, such as hard navigation due to insufficient documentation on how components were connected. The scripts used for build and tests in the nightly build were complex, nestled together with other scripts, making it difficult to comprehend their purpose and the rationale behind certain steps. Thankfully, the people at the office were incredibly friendly and helpful. Their assistance played a vital role in helping us overcome these obstacles and make progress. With-

out their support, we would not have been able to achieve as much as we did.

One factor that played a role in why we did not get enough time to look into test case selection was that we only had access to one company-configured computer. Jenkins, Gerrit, and the file system could only be accessed through a computer configured by the company. Which meant we could only have a single person working with Jenkins and the system code base at a time. This was a problem for us as due to some factors we worked remotely for the most part. If we had access to one laptop each, we could have worked on two different things related to our pipeline implementation at the same time. This could have made it possible for us to implement further improvements and features at the same time.

To work with the design and implementation in iterations we believe worked very well. As it was very hard to be newly introduced to the company's system and everything around it. Just getting something that works and then addressing bottlenecks in an iterative manner helped us understand the complexity of the product.

Using interviews with developers and stakeholders as a method to gain a deeper understanding of their needs was an effective method. It provided us with valuable knowledge that was used to formulate our design choices. In hindsight, it could have been preferred to repeat the interviews after each iteration or major design step to gain valuable feedback on users wishes and thoughts about the latest changes. Instead, we relied on direct communication with the Development Lead, the Test Lead, and the CM to aid us with our problems and to gain some feedback on our current progress in their respective area. Ultimately, why we chose not to gather a broader set of feedback after each iteration than to coordinate with all the developers were due to time limitations. We weighed the added value of more feedback against time and decided that the time saved by not asking all the developers for feedback was more valuable to us and them as it would let us have more time to improve the pipeline further.

7.2 Generalisation

In this section, we will highlight and analyse which of our findings from RQ1, RQ2, and RQ3 that can be generalised and in what circumstances. This analysis is valuable to readers interested in which of our findings are applicable to their situation.

7.2.1 RQ1 - Pipeline

The most valuable benefit that developers at the case company could see from our implementation was that they could get a fast and consistent confirmation that the changes they made build properly and will not cause problems with the nightly build. This can be generalised for other companies introducing a faster and continuous integration in an agile context, as it will let developers receive build feedback faster.

This benefit from using a build on every commit can be generalised for anyone that also uses only nightly builds or builds starting at specific times instead of every time a change is made. Another benefit that also can be generalised is when building before putting a change up for a code review lets reviewers save time and gives them further security that the change will not cause build problems if they will not find small mistakes that could cause a build failure at a later stage.

From our results, we can see that the introduction of automatic build and test on commits has benefits. In our situation, it helped save time for developers when wanting to build and test their changes. Companies that want to introduce automatic processes for similar purposes can therefore expect similar results. Another benefit reported was also that it created a uniform step that made it easier for developers to know that changes made by other developers had passed through the pipeline and therefore builds correctly. This perceived security is harder to generalise as it depends heavily on the development processes, the developers in question, and the general development mentality at the company.

7.2.2 RQ2 - Build Time

Only a single improvement was made for the build optimisation. It gave a significant reduction in time and more focus was required on test optimisations to bring down the pipeline to a reasonable execution time. The improvement made to reduce the build optimisations was to only build the developer platform instead of all 3 platforms built during the nightly build. This optimisation is very specific to the case company's build setup, but it is possible that other similar setups could use a similar approach to reduce their build times. The degree of improvement that this would yield is therefore uncertain as it depends heavily on the context in question. However, the general perspective can be put as *Build only the minimum of what is necessary to fulfil your goal.*

7.2.3 RQ3 - Test Time

To reduce the test times we performed both a static test selection from the entire night suite and then parallelised the test execution on 90 VMs. As we were limited to not increasing the available hardware we could not scale the number of VMs to further reduce the test times or increase our test selection with more tests. Therefore our optimisations techniques, especially a test selection, static or dynamic can be a relevant technique when faced with the same or similar hardware limitations.

Using test selection as a technique to reduce test times while still maintaining the ability to detect failures to a high degree is a matter of balance. Selecting test cases carefully to maximise the ability to detect failures under the time allocated for testing. Therefore we conclude that using a test selection to decrease the test times can be generalised for any software project, but the time reduced while maintaining good failure detection depends on the project in question. One key aspect we believe is important to more easily improve test times is to have a well-organised test suit. Organised in such a way that it can easily be known which part(s) of the code/system the tests relate to as this would make it easy to implement a test selection that chooses tests depending on where changes have been made in the code. Another aspect that is more specific for the case company is refactoring and optimising tests, this can make them execute faster and also help with making flaky tests stable.

7.3 Threats to Validity

The credibility and applicability of any research findings heavily rely on the validity they possess. As such, it is essential to identify and address any factors that could undermine the

validity of the study. In this section, we discuss the potential threats to validity in the context of our work. To substantiate our claims, it is necessary to analyse the validity of the findings. If the methods used were correct and could generate valid results. If the results and quantity of collected data are valid and solidify our claims.

When evaluating the results for RQ1 we conducted a second set of interviews after some time, to let the developers have some time to familiarise themselves and use our pipeline. One problem that we encountered here was that they were at the end of the product life-cycle, which meant that they were not focusing on writing code as much. This meant that they had fewer opportunities to use the pipeline than we had planned. The limited usage of our implementation by the developers during this evaluation period means that our conclusions drawn from them are less certain. With more time, preferably a longer evaluation period should be performed to discover all the benefits our implementation will have on the development team at the case company.

Although the unfortunate situation where the developers could not use our implementation as much as we would have liked. The results from RQ1 can still be considered valid, because of the multiple interviews performed where almost all of them perceive the same benefits. The only problem was about the testing parts of the pipeline where the perceived benefits were few. This was because of the few commits made during our evaluation period and the instability of the test results. Therefore further evaluation needs to be done to see all the benefits possible from the faster testing.

Our results for RQ2 were simply that we only build the platform that we required to execute the tests instead of all of the existing platforms. As this is very specific to the situation at the case company we do not see any threats to the validity of this optimisation.

As a result for RQ3 we introduced a simple test selection that reduces the test time in the pipeline, the test was chosen through a collaboration with the Test Lead. While a smaller test selection than the entire test suite will always lead to faster test execution, the failure detection rate from the selected tests depends on the chosen tests. To ensure as high fault detection rate as possible during a short time, tests need to be chosen smart and this depends entirely on what tests are available and what product is being tested. Since we do not have results on this we would have liked to invest more time in exploring the tests chosen and improving this test selection, as it would have ensured the effectiveness of our test selection. Without any clear data on this, the validity of our test selection cannot be ensured.

7.4 Related Work

This section will present other works that are related to this thesis. The goal is to describe and compare these other works with our research. Discuss similarities and differences and reflect if some of the findings they make are interesting to explore with our work. We chose to discuss works that focus on test selection as it is the area that we would have liked to explore more in our work. By comparing these works with ours we hope to explore possibilities for future ideas and improvements for our work. These works will be referred to as P1 to P4 and their full titles can be seen in table 7.1.

ID	Title
P1	DevOps Improvements for Reduced Cycle Times with Integrated Test Optimizations for Continuous Integration
P2	Shortening Feedback Time in Continuous Integration Environment in Large-Scale Embedded Software Development with Test Selection
P3	Software Testing - Test Suite Compilation and Execution Optimizations
P4	Understanding and Improving Regression Test Selection in Continuous Integration

Table 7.1: List of Related work

P1

Dusica Marijan, Marius Liaaen, and Sagar Sen [14] propose and investigate a new design implementation for test selection to reduce cycle times. This is relevant for our work as we previously discussed that test selection would be beneficial for our pipeline and is something we did not have time to fully explore and implement. If the case company decides to explore test selection in the future this research could be a good place to start exploring.

Summary

Long build times have been reported as one of the top barriers that developers encounter when using CI. In DevOps you want to be able to deploy fast while guaranteeing high quality. The challenge when it comes to building and testing then is the trade-off between low test run-time and fault detection. The goal is to reduce cycle times by reducing long test run-times by finding a optimised test suit that maintains the following: Low run-time, high fault detection, and high-risk coverage. First, they proposed a solution for selecting an optimally reduced test suit. Then they validate the proposed solution in an experimental case study. It can be summarised to *find a subset that has the following properties: test cases with the shortest execution time, test cases with the highest consecutive failure rate for the given number of executions, and test cases with the highest coverage of uncovered risk scenarios.* To keep in line with the practices of DevOps the test selection optimisation not only utilises the test results from previous test runs but also production and risk assessment indexes obtained by domain analysis.

The case study was divided up into 5 research questions. Three of the investigated the effectiveness of the proposed solution. While the last two were about comparing the proposed solution with random test selection with the same time budget. The test data set used in the validation was 10 test suites, each containing historical test execution results of tests used in regression testing for 6 consecutive iterations. As a baseline to evaluate the effectiveness, they used change impact analysis on the code modification to find which areas of the system should be tested. This approach is stated to be popular in the industry. The results were

that the optimised test suites required 18% - 35% less execution time. Averaged seemed to be around 25%. It created a minor impact on fault detection effectiveness and risk coverage. In both cases up to 3%. Fault detection effectiveness was preserved for 20% of the test suites. Risk coverage was preserved for 30% of the test suites. On average, the test suites generated by random selection showed a 40% lower fault detection efficacy and 33% lower coverage of high-risk test scenarios. It is then concluded that the proposed test optimisation solution is a significant improvement over the other methods that it was evaluated against.

Discussion

The problem they addressed is real, and exploring new design strategies to tackle this problem has great value. However, there exist factors that threaten the solution's credibility. The proposed solution that was implemented and tested was only tested at one company. Hence for it to be more reliable and to enable wider industry insights to be derived, it should be tested across multiple companies. The authors acknowledge the necessity for additional and extensive testing and have expressed their intent to carry out such testing in the future. It will be intriguing to observe the outcome of this subsequent research.

One prerequisite for incorporating the approach is to have well-organised test suits in terms of what tests are related to which area of the system. It is also important to save x amount of test result data. Attempting to apply this approach at the case company would require a considerable amount of time as the test suites are not clearly organised, making it difficult to identify the system area being tested. Furthermore, it would necessitate the storage of more test results, which is not feasible due to hardware limitations. Despite these challenges, it would be worthwhile to experiment with this approach since it has the potential to positively impact our pipeline. The results obtained through this approach seem promising in theory, and it is an improvement to the common test selection approach they compared against. It is highly likely that it would be a better option than our fixed sub set of tests.

P2

Jarmo Koivuniemi [11] completed a master's thesis similar to ours, which was conducted in a comparable setting. Their initial problem is the same as ours. However, their thesis focused solely on improving test times through test selection.

Summary

The topic of this thesis was to investigate how to shorten feedback time for developers in large-scale embedded software development. The focus is on optimisations for test times as it is a common bottleneck for companies with large code bases. Because then testing everything every time becomes expensive. The three most popular test optimisation techniques are test suite minimisation, test suite optimisation, and test case selection. Test case selection was found to be the best suited for the company where the thesis was conducted. Almost everything in the thesis is viewed from a continuous integration point of view. The method used was to mix literature and real implementation experiment.

The case company had started implementing a new tool from scratch that this theses then took over and continued implementing. The need for constructing its own tool was due to

the instrumented code being executed in a memory and CPU-constrained environment. By creating it on their own they would have easier control over the tool compared with using an existing tool. The tests are first selected by coverage so it selects the tests related to what changes are made in each commit. Then they add test cases that failed in the previous run. However, because the environment is unstable there are a lot of flaky tests. So most of the time the tests that were added from previous runs most likely fail due to the environment and not a real fault.

Literature was used to justify decisions used throughout the design implementation. The regression test suit was the suit that saw the most benefits with test selection in regard to time. From 193.1 minutes, down to 75.3 minutes with test selection. In most cases, the sub set of tests found the same number of faults as the full test suit did. However, there was some uncertainty due to the test environment being unstable.

Discussion

This thesis is very related to our thesis and it is the most similar work compared to ours that we have found. The problem it addresses is in its foundation the same problem as we address. They shared a couple of obstacles with us, an unstable test environment, flaky tests, limited hardware, and limited test result storage in Jenkins. These all added to uncertainty regarding their results. Their work was carried out at one company, which also adds uncertainty to the validity. We also face this problem with our validity. Unlike us, they solely focus on improving test times. If put aside the threats to the validity of their results, it looks very interesting. That they manage to find the same number of faults with test selection in 39% of the time it took to run all tests. Therefore this further points towards test selection as highly beneficial for the pipeline design we purpose for the case company. It would have been interesting if we could have made a comparison between their results from the fault detection investigation they did on their prototype solution. However, we did not have time to investigate that and if we had, those results would be very untrustworthy due to flaky tests. If we were to have had trustworthy results we could compare them with their results and that might have given us a better indication of the performance of our static test selection and how efficient a similar solution to theirs would have worked for us.

P3

Stratis Panagiotis [18] completed a Ph.D. thesis where he investigated the possibilities to reduce test times without removing test cases or changing the hardware through 3 novel approaches. Since we also have a limitation on increasing hardware the techniques proposed were therefore interesting.

Summary

Software has through the years increased in complexity and importance, making testing a crucial part of software development. It is a difficult and time consuming task to ensure that software is free from failures and meets all its requirements, which requires large test suites. Most optimisation techniques used today create a test selection by removal of redundant tests to reach a high code coverage as the metric. With growing software projects the amount of

essential test cases grows too. Therefore in this thesis, they explore techniques to reduce the test times without removing any test cases or changing the computation hardware.

They propose 3 novel ways to optimise the test times. Firstly, through a data transformation that reduces the number of instructions in individual test case code to optimise the overall test suite compilation. This is made by restructuring test inputs so that it reduces the number of required calls to the software. The first proposal finds that the speedup depends on the size of the test suite, with less than 100 tests the improvement is minimal. The speedup also varies heavily on the project, for some projects it found speedups of 69x times while for others it was only by 1.8 times. Secondly, an approach that schedules test cases in an order that minimises the distance between instructions for neighbouring tests. Improving this distance between test instructions reduces the test execution times. It found speedups of 8%-30% depending on the subjects. Thirdly, through another scheduling technique that improves the load balancing between multiple devices of a diverse system with different types of computing cores to reduce the test execution times. It found an improvement of 20% on heterogeneous systems.

The first proposal for optimisations were evaluated through an empirical study using programs from industry, after using the code transformation on the subject code it was then evaluated on compilation, speedup, execution time, scalability, and correctness. The second proposal was empirically evaluated using 20 subject programs from a repository with software related artefacts meant to support experimentation and through industry standard benchmarks. On these programs, the execution times and cache misses were compared using the proposed scheduling technique versus a traditional scheduling that maximises code coverage. The third proposal was evaluated on a large industry test suite.

With the three techniques investigated it was found that all of them could improve the test times but it depends heavily on the software in question to what degree. The techniques proposed can also be used together with other optimisation techniques. Their thesis confirms their belief that there is a need to utilise low level optimisations before reducing the number of tests in test suites.

Discussion

What this thesis has in common with ours is the limitations on increasing the hardware available to reduce test time further. In contrast to us, they decided to limit themselves to not be able to reduce the test suite to reduce the test time. Because of this a comparison in time saved between their techniques and ours can't be made directly as they do not compromise on the number of tests executed while we do. Meaning the time reduced in our implementation is possible due to less security in the testing, while theirs do not compromise on the security at all and only looks to optimise the time spent on testing.

Therefore their techniques would be interesting to investigate further for us if we had more time since it would let us reduce the test time in our pipeline further. With a faster test time we could either provide users with feedback faster or to extend the test selection to provide more precise feedback while still maintaining the same test time.

One problem for us to incorporate the techniques proposed in the thesis is the validity of their findings. The degree of improvement is very varied over the different subjects they evaluated their techniques on, for some subjects the improvement gained was also very minimal. As their techniques are quite complicated and could take some time to investigate further

and implement, the cost versus the benefits is therefore something that needs to be taken into consideration.

P4

Shi et al. [17] wrote a conference paper investigating ways to perform test selection for regression testing in CI. This is interesting for us as it gives us some ideas for potential ways to improve our test selection.

Summary

Regression testing is a common practice in continuous integration(CI), but it can be costly to run the entire test suite during regression testing. Therefore regression test selection(RTS) can be used to reduce the cost. In industry, module level RTS is commonly used while in academia class level is instead more often proposed. Therefore this paper compares the two options in a cloud based CI environment. A hybrid RTS technique that combines both module and class level is also proposed and evaluated.

It is found that all the RTS techniques investigated save testing time compared to running all the tests, around 77% of time saved compared to running all the tests with selections around 30% of the entire test suite. When evaluating test failures it is found that RTS techniques can miss to select tests that failed, but these are almost always flaky test failures. Avoiding flaky tests can provide additional value as it lowers the time required by developers to debug failures that were not actually related to the code change. From the comparison, the technique that combines both class and module level test selection is preferred as it is faster than module selection and although slower than class selection it is a safer choice that detects failures to a better degree.

22 cloud based CI projects that matched certain filters were chosen from over 1000 open source projects on GitHub. The filters were: projects using Travis, where some testing was performed in Travis, average build times, and projects that can use all 3 tools. The 22 projects had an average testing time of 10 minutes. Then for each project the different techniques were used on the same commits to generate testing time and recording which test cases were selected.

RTS can be used to reduce costs during regression testing in CI, although it provides less speedup in cloud based projects compared to local ones due to more overhead in cloud based projects. In industry module level RTS is the most popular, it is a more safe choice that although slower than class level RTS can detect more failures. The proposed hybrid module that combines both class and module selection is found to give a good trade-off between the best traits from the module and class RTS. It is faster than module selection but slower than class selection, it is also safer than class selection but not as effective in finding failures as the module selection. It is also found that RTS techniques often miss selecting flaky tests that fail in a “run all” regression test. This was another benefit of using RTS as this lets developers avoid spending time debugging flaky tests.

Discussion

In comparison to our test selection where only around 2-3% of the entire test suite is chosen, their selection is much larger at around 30%. With their selection, they save 77% of the total time while in our case the amount of time saved is around 85%, which was calculated by 7.1.

$$\frac{Time\ for\ test\ selection}{Time\ for\ entire\ suite} = Time\ saved \tag{7.1}$$

The reason that we do not manage to save that much more time with a much more restrictive selection is due to the overhead when setting up the test environment and that the tests selected are quite long and extensive tests. Therefore we could gain numbers closer to the other study if we were able to compromise a little on the total test time.

As our current test selection is quite simple and is definitely a point that could be improved further with more time, the proposed hybrid technique is an interesting technique that could be beneficial for our pipeline.

The thesis also highlights that using a test selection for regression testing is preferred over executing the entire test suite. Since our pipeline basically works as a regression test it strengthens our decision to use a test selection to reduce the test times.

The thesis also mentions that using a good test selection can help avoid flaky tests, which would give developers more time to investigate real faults and not spend time debugging false failures. Since the case company's current test environment suffers from instability and they have a few flaky tests, an improved test selection for our pipeline could therefore also have further benefits except better failure detection.

7.5 Future Work

This section aims to present our thoughts and ideas for future research and possible improvements in our implementation that could have been done with more time. It will reflect on the findings from the related work in contrast to our findings if there are techniques or methods used in the related works that would be beneficial to add to our work. What parts of our research need future investigation or points where improvements need to be made. How the threats to the validity of our claims can be investigated further.

In the future, it would be interesting to evaluate the fault detection of our pipeline in relation to the nightly build and test. Due to the unstable nature of the test environment and the flaky tests, we deemed the time and effort it would take us to investigate this was not currently worthwhile with the small amount of time that we had left. As the validity would be too uncertain under the current circumstances. Further investigations into this could be well-suited for new thesis work. Evaluate the fault catching capability of our pipeline compared to the nightly build and investigate potential improvements to further its capability.

Because of the very limited evaluation period of our implementation, it made it difficult to prove long term effects on the developers at the case company's ways of working. With more time, we would have liked to investigate this further and possibly come back after a few months to compare and see if there have been any changes, both in the mentality and in the commit behaviour. Therefore we request future work to be done in the area to find long term effects of introducing a faster pipeline. To further investigate the validity of our findings and claims, we recommend conducting a new study at the case company, after 6-12

months of time after our work is done, with a focus on the effect our pipeline had on the developers and what other improvements can be done to improve the ways of working.

Here we present a list of improvements for the case company as a guide for where to start when continuing work on the pipeline. The suggestions are a mix of things we did not have the time to include, things we believe are beneficial to have and suggestions developers have brought up with us.

Test selection. The current test selection in the pipeline is static, simple, and more of a placeholder. Therefore it should be replaced by a more dynamic selection that selects tests depending on the change committed. From the interviews, we can see that excluding a smarter selection they would also like to be able to manually choose test cases to be added to the remaining automatically chosen tests. From the related studies, we see further support for using a test selection and that the tests should be carefully chosen to maximise fault detection. One study presented a hybrid technique that chooses tests depending on what module and what class have changed with the latest change, therefore we recommend a similar approach for implementing a more dynamic test selection at the case company.

Content based steps. Based on the content within the commit/change is it of interest to do different steps. For example, if you were to commit changes made in test cases or added new test cases you are not interested to see if the system is able to build. Instead, you want it to execute those tests on an already stable build to see if they work. This is something we already added support for but it needs some further improvements. For example, if you upload a commit only containing changes to the documentation, you do not want to build and test anything.

Set up test environment. Another thing we would have liked to explore further is the stage where our pipeline sets up the test environment. We strongly believe that the process can be done faster than the average 3 minutes and 20 seconds it currently has. Because when we ran the tests in sequence the time it took to set up the test environment was way faster. Sadly we never clocked the exact time it took then. It would be interesting to figure out why there is a remarkable difference.

Flaky tests. This is both a problem for our pipeline and the nightly build, but it is also a developer mentality. The pipeline gets a few unstable results due to tests that are failing and most failures are due to flaky tests. The nightly build has to spend a long time every night where it reruns test that has failed. But more importantly, we have seen that it adds to the mentality "just rerun it and hopefully, it will pass". This is a bad practice as they do not seem to take test results so seriously. This also causes distrust in test results, where it can never really be trusted that it actually was a fault.

More hardware resources. If the case company were to move things into the cloud. We highly recommend ensuring that they then have the capability to run multiple pipeline jobs at the same time instead of having the queue that is currently in place because the resources are not enough. It is also recommended to also ensure that it can handle all 2000 test cases in the given time span that they want. This will increase the value of the pipeline. As it could give better and faster feedback to the developers, which would increase the team's produc-

tivity.

Look over existing scripts. Our pipeline uses the same scripts as the nightly build also uses. We discovered when we first looked into these scripts that they in turn use further scripts. Most of these scripts were very hard to understand what they are doing and why they are doing the things they do. We highly recommend that they go through these scripts and update them so that it becomes clear what their true purpose is and how they achieve it.

Look over Test result location. We have gotten feedback that people would like it if the logs from the test results from our pipeline were more easily accessed. Currently, they are stored the same way as the logs from the nightly build and it takes that you know where to look to be able to find them. The best way is if the team sat down and figured out where they want the logs to be accessible from and how. It could be nice if they were somehow accessible from the review page inside Gerrit.

Chapter 8
Conclusion

The initial problem that this work aimed to solve was that developers at the case company suffered from long feedback loops that forced them to multi-task while waiting for feedback. Thus the goal became to shorten the feedback loop for each code change the developers make. From interviews with developers, the goal became to bring it down to 15 minutes or less. This resulted in the following research questions:

- **RQ1** What are the key considerations and practices for designing a suitable pipeline for the case company?
- **RQ2** What strategies for reducing build times are suitable for the case company?
- **RQ3** What strategies for reducing test times are suitable for the case company?

For RQ1 we managed to design and implement an automatic pipeline that integrates well into the current development process. The developers feel comfortable using it and it brings the feedback loop down to less than 15 minutes.

For RQ2 we found that the best and easiest way to reduce build time was to only build the developer platform instead of all three available platforms. Building the dev. platform takes an average of two minutes while building all three takes the nightly build job an average of 22 minutes.

We found for RQ3 that some sort of test selection and parallel test execution to be the best suited options. Our prototype utilises a static test selection. However, we have not been able to evaluate if it ensures the desired quality that the developers seek. Instead of getting feedback the next morning developers now get it after 10-11 minutes after committing a change to the repository. This is a huge improvement and the work that comes after is to ensure that the feedback meets the desired quality and if the total time can be brought down to 3-4 minutes.

References

[1] Zahra Shakeri Hossein Abad, Mohammad Noaeen, Didar Zowghi, Behrouz H Far, and Ken Barker. Two sides of the same coin: Software developers' perceptions of task switching and task interruption. In *Proceedings of the 22Nd International Conference on Evaluation and Assessment in Software Engineering 2018*, pages 175–180, 2018.

[2] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, Boston, 2004.

[3] Lars Bendix and Torbjörn Ekman. Software configuration management in agile development. In *Software Applications: Concepts, Methodologies, Tools, and Applications*, pages 291–308. Igi Global, 2009.

[4] Vinod Kumar Chauhan. Smoke testing. *Int. J. Sci. Res. Publ*, 4(1):2250–3153, 2014.

[5] Adam Debbiche, Mikael Dienér, and Richard Berntsson Svensson. Challenges when adopting continuous integration: A case study. In *Product-Focused Software Process Improvement: 15th International Conference, PROFES 2014, Helsinki, Finland, December 10-12, 2014. Proceedings 15*, pages 17–32. Springer, 2014.

[6] Santiago Delgado. Parallel testing techniques for optimizing test program execution and reducing test time. In *2008 IEEE AUTOTESTCON*, pages 439–441. IEEE, 2008.

[7] Tomas Fernandez. 5 ways to run faster ci/cd builds, June 2022. `https://www.bairesdev.com/blog/how-to-speed-up-your-ci-cd-pipeline/` (Retrieved: 23/01/10).

[8] Martin Fowler and Matt Foemmel. Continuous integration, May 2006.

[9] Gerrit, January 2023. `https://www.gerritcodereview.com/about.html` (Retrieved: 23/01/15).

[10] Jenkins, January 2023. `https://www.jenkins.io/` (Retrieved: 23/01/15).

[11] Jarmo Koivuniemi. Shortening feedback time in continuous integration environment in large-scale embedded software development with test selection. *Master's Thesis University of Oulu*, 2017.

[12] Eero Laukkanen and Mika Mäntylä. Build waiting time in continuous integration – an initial interdisciplinary literature review. In *2015 IEEE/ACM 2nd International Workshop on Rapid Continuous Software Engineering*, pages 1–4, 2015.

[13] Dusica Marijan, Arnaud Gotlieb, and Sagar Sen. Test case prioritization for continuous regression testing: An industrial case study. In *2013 IEEE International Conference on Software Maintenance*, pages 540–543. IEEE, 2013.

[14] Dusica Marijan, Marius Liaaen, and Sagar Sen. Devops improvements for reduced cycle times with integrated test optimizations for continuous integration. *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC), Computer Software and Applications Conference (COMPSAC), 2018 IEEE 42nd Annual, COMPSAC*, 01:22 – 27, 2018.

[15] Microsoft. Incremental builds, October 2021. `https://learn.microsoft.com/en-us/visualstudio/msbuild/incremental-builds?view=vs-2022` (Retrieved: 23/01/16).

[16] Jiantao Pan. Software testing. *Dependable Embedded Systems*, 5, 1999.

[17] August Shi, Peiyuan Zhao, and Darko Marinov. Understanding and improving regression test selection in continuous integration. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, pages 228–238. IEEE, 2019.

[18] Panagiotis Stratis. Software testing: test suite compilation and execution optimizations. *The University of Edinburgh*, 2020.

[19] Alexey Tregubov, Barry Boehm, Natalia Rodchenko, and Jo Ann Lane. Impact of task switching and work interruptions on software development processes. In *Proceedings of the 2017 International Conference on Software and System Process*, pages 134–138, 2017.

[20] Mubarak Albarka Umar and Chen Zhanfang. A study of automated software testing: Automation tools and frameworks. *International Journal of Computer Science Engineering (IJCSE)*, 6:217–225, 2019.

[21] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. Quality and productivity outcomes relating to continuous integration in github. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, page 805–816, New York, NY, USA, 2015. Association for Computing Machinery.

[22] Wikipedia. Regression testing, January 2023. `https://en.wikipedia.org/wiki/Regression_testing` (Retrieved: 23/01/10).

[23] Wikipedia. Software testing, January 2023. `https://en.wikipedia.org/wiki/Software_testing` (Retrieved: 23/01/10).

Appendices

Appendix A
Interview questions

Interview questions for developers

1. Name, work title/responsibilities/area?
2. How long have you been working at the company? Earlier roles?
3. How is the division of labor in the team?
4. Do developers have areas of responsibilities? Do multiple developers work in the same area or not? Do you normally work on a single task at a time or multiple parallel tasks?
5. How is the internal team communication?
6. How would you describe the current development strategy?
7. What is the biggest driving force behind how you work today?
8. What are the benefits and disadvantages of your current development ways?
9. What are some common problems that you face in daily workflow?
10. How do these problems affect yours or the team's development pace?
11. How are these problems usually solved or mitigated?
12. What do you think should be changed about the current ways of working?
13. How does cybersecurity affect your daily work?
14. How comfortable are you with using git?
15. What does your repository structure look like?
16. How often do you test your changes and how?
17. Why do you test? Do you test before sending your change to the repo?
18. Do you build locally before sending changes to the repo?
19. Is it hard to test? How does the process work?
20. Do you know what the tests test?
21. How often do you push your changes to the repo?
22. How often do you "pull" down the latest changes from the main branch?
23. How often would you say it is optimal to test?
24. Do you think something needs to change about the testing?
25. If the nightly build fails how long does it take to fix the problem?
26. How long does it take before you are sure that the problem is completely fixed?
27. How often does the nightly build fail?
28. How long would you say it takes before you receive feedback on a change?
29. What is your preferred time for this? What would be optimal?
30. How would you like an automatic process that tests your changes to look like?
31. How should it start? What kind of feedback? How long should it take? How precise should it be?
32. What does your release structure look like? How often? How does it work?
33. Is releasing a stressful element? Does it take a long time to be finished with a release?
34. Other thoughts?

Interview questions for higher stakeholders

1. Name, work title/responsibilities/area?
2. How long have you been working at the company? Earlier roles?
3. What does your release structure look like? How often do you have releases?
4. How does the release process work? Is it a stressful element in development?
5. What are your thoughts about the current release structure? Something you would like to change?
6. What is the driving force behind the way you currently work?
7. What are the most commonly reported problems?
8. How do these problems impact development speed and productivity?
9. What are the largest organisational problems with your development process? Other organisational problems?
10. What are the advantages to the current ways of working?
11. What do you think should be changed about the current ways of working?
12. What do you think about the nightly builds? advantages and disadvantages? Other alternatives?
13. What do you consider a reasonable trade off between increasing hardware or reducing test cases to increase the time to test and build?
14. What limitations exist in increasing the current hardware?
15. How would you like an automatic process for testing changes to look like?
16. How should it be started? What kind of feedback would you like to see? How long should it take?
17. How precise should it be? Optimal level of preciseness?
18. How do you think the team and organisation at large will be affected by faster build and test times?
19. What would you say are the biggest issues for shortening both build and test times?
20. Other thoughts?

EXAMENSARBETE Two-step continuous integration
STUDENTER Andreas Erlandsson, Hannes Lantz
HANDLEDARE Lars Bendix (LTH)
EXAMINATOR Emelie Engström (LTH)

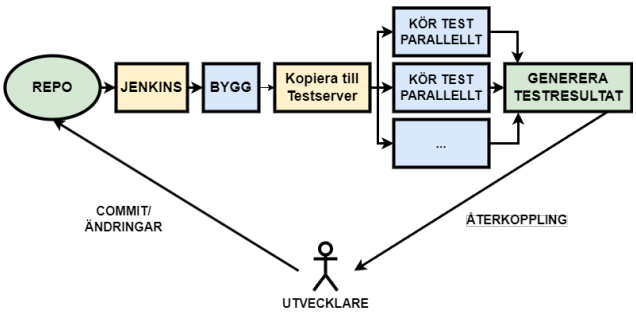
Tvåstegs kontinuerlig integration

POPULÄRVETENSKAPLIG SAMMANFATTNING **Andreas Erlandsson, Hannes Lantz**

Att behöva vänta länge för att få återkoppling på sina kodändringar gör att utvecklare ofta behöver arbeta med ett antal olika saker parallellt. Detta leder till att deras produktivitet minskar. Detta arbete undersöker hur man kan ge återkoppling snabbare.

Med vår prototyplösning har vi lyckats göra så att mjukvaruutvecklare på ett företag kan få återkoppling på deras kodändringar mycket snabbare än tidigare. Detta var möjligt genom att introducera en process som automatiskt bygger och testar varje kodändring. För att denna processen skulle vara tillräckligt snabb behövdes en del tidssparande optimeringar göras. Genom att endast bygga det minsta som krävs för att kunna köra de testfall som önskas, sparas en hel del tid. Lika så sparas mycket tid genom att vi använder ett mindre antal testfall. Genom att bara kontrollera grundfunktionalitet istället för att köra alla testfallen. En annan teknik vi utnyttjar för att spara tid vid testningen är att utföra testfallen parallellt istället för en efter en i serie. Eftersom den totala tiden för att köra alla testfall då blir markant mindre. Som snabbast kan den totala tiden för att köra testafallen bli tiden det tar att köra det testfallet som kräver mest tid. För att det ska vara möjligt ställs det dock större krav på ens hårdvaruresurser.

Arbetet utfördes på ett företag som tidigare endast byggde och testade sina kodändringar under natten. Detta ledde till att utvecklare var tvungna att vänta till nästa dag för att få återkoppling på sina ändringar. Denna långa väntan gjorde att utvecklarna fick arbeta med flera uppgifter parallellt. Med hjälp av vår prototyp får utvecklarna återkoppling mycket snabbare. Detta gör att de slipper arbeta med flera uppgifter samtidigt och kan fokusera på en uppgift i taget.



Det steg som utförs av vår automatiska process beskrivs i bilden. Den startar sig själv när en utvecklare lägger upp sina kodändringar på deras versionshandteringsserver (repot). Resultatet visar att utvecklarna känner sig tryggare i att gå vidare till en annan arbetsuppgift efter de har fått snabb återkoppling på att sina kodändring lyckats bygga och att resultatet från det mindre testurvalet ser bra ut. Utvecklarna får nu återkoppling på sina ändringar inom 15 minuter och väljer själva när, vilket är en stor förbättring. Den förbättrar även för de som granskar kodändringarna innan de godkänns eller nekas. Eftersom de nu kan känna sig tryggare att småfel fångas av vår lösning och kan istället fokusera på de övergripande detaljerna som rör varje ändring.