MASTER'S THESIS 2020

Continuous Delivery: Challenges, Best Practices, and Important Metrics

Anders Klint, Vilhelm Åkerström

ISSN 1650-2884
LU-CS-EX: 2020-22

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2020-22

Continuous Delivery: Challenges, Best Practices, and Important Metrics

Anders Klint, Vilhelm Åkerström

Continuous Delivery: Challenges, Best Practices, and Important Metrics

Anders Klint
dat15akl@student.lu.se

Vilhelm Åkerström
dat15vak@student.lu.se

June 18, 2020

Master's thesis work carried out at Robert Bosch AB.

Supervisors: Lars Bendix, lars.bendix@cs.lth.se

Examiner: Elizabeth Bjarnason, elizabet.bjarnason@cs.lth.se

Abstract

With the rise of agile methods in the software industry, automating the delivery process using Continuous Delivery (CD) has become a desirable goal. However, achieving a healthy CD-pipeline is not an easy task, let alone maintaining one. This thesis aims to investigate what challenges are related to CD, best practices to mitigate these, as well as to analyse which metrics could be important in monitoring the performance of a CD-pipeline. In order to answer these questions a combination of literature studies, interviews with developers, and an implementation of a proof of concept for displaying potentially useful metrics from a chosen team's pipeline was utilised. Additionally, some practitioners were interviewed and surveyed in order to gather their opinion about which of these metrics were useful for improving a CD-pipeline.

This resulted in a list of challenges and best practices in relation to CD, as well as a list of metrics considered important when looking to improve the efficiency of a CD-pipeline. These metrics were used to find two major bottlenecks in a chosen team's pipeline; one caused by limited licenses to a unit testing tool, and one caused by generating documentation unnecessarily often. In terms of challenges the majority of them were related to automation and human or organisational factors such as a lack of resources or discipline. Some of the important best practices were having shared responsibility between developers of the CD-pipeline, and utilising automation by making sure that it is a priority.

These challenges, best practises, and metrics could assist practitioners or fellow researchers in finding potential improvements of CD-pipelines, and the investigation of the chosen team could serve as an example where these findings had a positive effect by finding bottlenecks of a team's pipeline.

Keywords: Software Engineering, Continuous delivery, Continuous integration, Process improvement, Delivery Pipeline Feedback

Acknowledgements

We would like to thank our supervisor Lars Bendix for his substantial support and valuable feedback throughout the thesis. We would also like to thank our supervisors at Bosch, Axel Franke and Peter Walls, for their engagement and helpful support of our thesis work, and for facilitating the creation of this thesis. We would also like to thank Marek Pola Loethman and Erik Stenlund for letting us work with their teams, and Jari Kuusisto for helping us with implementing the proof of concept. Finally, we'd like to thank Flavius Gruian for providing us with the LaTeX template used to create this report. The template was altered slightly to account for the change of the cover photo to one which Anders Klint owns the full rights to.

Distribution of work

The work load was distributed evenly among the authors. Since Vilhelm Åkerström didn't have access to a computer connected to the case company's intranet for the later stages of the implementation, most of the coding and implementation was done by Anders Klint. However, Åkerström dedicated this time to further research and planning the writing of this report. Regarding the report writing, some sections were written cooperatively, and some divided between the authors. Generally, Åkerström had a focus toward challenges and best practices, as well as interviews, and Klint had a focus toward the implementation and its result, as well as the survey and the topic of metric-based feedback.

Contents

1 Introduction **7**
1.1 Problem Statement . . . 7
1.2 Research questions . . . 8

2 Background **9**
2.1 Theory . . . 9
2.1.1 Continuous Integration . . . 9
2.1.2 Continuous Delivery . . . 11
2.2 Case Company . . . 13
2.3 The advantages of a fast feedback loop . . . 14

3 Method **17**
3.1 Planned Method . . . 18
3.1.1 Literature study . . . 18
3.1.2 Interviews . . . 19
3.1.3 Deciding which metrics to gather . . . 21
3.1.4 Gathering the Metrics . . . 22
3.1.5 Evaluating the effects of the metric-based feedback . . . 22
3.2 Problems during thesis work . . . 22
3.2.1 Lack of suitable teams . . . 22
3.2.2 Working remotely due to covid-19 . . . 22
3.2.3 Lack of time to observe the effect of metric-based feedback . . . 23
3.3 Changes to the method . . . 23
3.3.1 Split into interview and implementation . . . 23
3.3.2 Gathering feedback from Developers . . . 23
3.3.3 Survey about what metrics are useful for each role . . . 24

4 Results **25**
4.1 Challenges and best practices of CD (RQ1) . . . 25
4.1.1 Literature studies . . . 25

4.1.2 Interviews . . . 28
4.1.3 Results for challenges and best practices . . . 30
4.2 Metrics for improving a CD process (RQ2) . . . 32
4.2.1 Literature studies . . . 33
4.2.2 Interviews . . . 34
4.2.3 List of metrics used . . . 35
4.3 Proof-of-concept: implementation and analysis (RQ2) . . . 36
4.3.1 Implementing metrics gathering and visualisation . . . 36
4.3.2 Analysis of the gathered metrics . . . 38
4.3.3 Metrics for Optimising Lead Time . . . 44
4.4 Feedback from developers (RQ3) . . . 45
4.4.1 Presentations to the teams . . . 45
4.4.2 Evaluation survey . . . 46
4.4.3 Results from interviews and survey . . . 49
4.5 Summary of results . . . 50
4.5.1 Challenges of CD (RQ1) . . . 50
4.5.2 Best Practices of CD (RQ1) . . . 50
4.5.3 Important metrics for improving CD (RQ2) . . . 51
4.5.4 The effect of giving developers metric-based feedback (RQ3) . . . 51

5 Discussion & Related Work 53
5.1 Evaluation of method . . . 53
5.2 Discussion of results . . . 55
5.2.1 Threats to validity . . . 55
5.2.2 Generalisation of results . . . 57
5.2.3 Overall judgement of the results . . . 58
5.3 Related work . . . 58
5.3.1 Defining metrics relevant to CD . . . 58
5.3.2 How and why metrics are used in the agile industry . . . 61
5.3.3 Best practices, benefits and challenges of continuous delivery . . . 64
5.3.4 Adopting continuous delivery: problems, causes and solutions . . . 65
5.4 Future work . . . 66

6 Conclusion 67

References 69

Appendix A First Interview Questions 73

Appendix B Further interview with the E2 team 75

Appendix C Metrics from literature 77

Appendix D Informational document included in the Survey 79

Chapter 1
Introduction

Continuous delivery (CD) is an agile software practice used by teams wanting to cut down on release cycle time, and get faster feedback on their changes [1]. This gives many software projects an incentive to adopt the practice. However, the ways that this is done can differ a lot from project to project depending on the circumstances such as developer experience, distribution of development resources, and management [2]. This can lead to sub-optimal results consisting of half-hearted CD pipelines with little maintenance that eventually starts to slow down development, or it can lead to teams successfully finding ways to work which helps them the most [3]. In order to succeed with CD there are a lot of processes that need to work together. Both internally in the team, but also when it comes to managers, product owners, etc. On top of this if there is little experience in the team when it comes to working with CD then there is a need to find good information about how to do it well, and what the challenges along the way are [4]. In this thesis we set out to discover which things are considered "best practices", both in terms of research and in the industry that helps teams to get the most out of working with CD. This will also include finding out what parts of working with CD present challenges to teams, and how these best practices can help them overcome these. On top of this we will also look at which metrics of the pipeline can be useful for developers and practitioners to help them when working with CD. To do this there is a need to look at what research already exists on the topic, as well as to find how people work with CD in practice.

The report will start by giving a background on the topic of CD. Then the method will be discussed followed by the results. Finally a discussion of the results and method as well as potential future work and the conclusions of the thesis will be presented.

1.1 Problem Statement

With the aim to explore this topic there are two avenues of research. The first being, studying literature on CD to discover what practices, challenges, and important metrics could be

found based on earlier studies. The second being, performing interviews with developers of different levels of skill when it comes to CD, as well as working in different projects at a company in order to see how they work with CD and what parts of this work well. This includes different types of projects, both in terms of scope, work processes, and target platforms. It also involves discussing what challenges they have had working in this way, and what improvements they would like to make given the time and resources.

Additionally, there is also the question of how to determine if the way a team works with CD is good or not. For this data is required. Modern tools used for developing software often have a lot of data available for their users if they can find it. However, practitioners are not always able to easily access this data, especially if it is scattered among multiple tools. Even if they are willing to go looking for the data, it can often be hard to parse it since there is so much of it, and the way the data is presented does not help with understanding it. There is also the problem of the uncertainty what data is relevant to look at, and the reason for looking at it can be misguided. Therefore, the second part of this thesis was to test a number of metrics gathered from a project, that are displayed in a coherent and user friendly way in order to see how helpful they can be to help developers in optimising their CD pipeline in terms of lead time and general code quality.

1.2 Research questions

This problem statement leads to research questions regarding what constitutes best practices for CD, what metrics are relevant for measuring the quality of a CD pipeline, and how these metrics can help developers optimise their workflow.

- **RQ1:** What are the challenges and best practices for continuous delivery?
- **RQ2:** What important indicators and metrics are relevant when looking to improve the continuous delivery process?
- **RQ3:** How can these indicators & metrics support developers to optimise for lead time and code quality?

To study this, multiple methods were used. Apart from the literature studies and interviews mentioned earlier, one additional method was used to help answer the second and third research questions. This was a proof of concept that collects relevant metrics from various tools that are part of the CD pipeline in a team, and which also displays this data in a way that can be easily interpreted by the developers. All of this was to help find ways to evaluate the approach which different teams take to achieve Continuous Delivery, and help developers new to the process to avoid common pitfalls. Giving guidelines to what some best practices are can both help evaluate existing CD processes and be helpful when creating new ones. The proof of concept could help improve the efficiency of the team that was used for the implementation, and potentially be generalisable by giving the developers better feedback and clearer goals in a more focused process, in addition to eliminating or decreasing bottlenecks in the pipeline, which can be done by noting where the developers get stuck in the development process and trying to minimise this time.

Chapter 2
Background

What is Continuous Delivery? Since this concept is still rather new, and not something everyone is all that familiar with, this question can have a lot of answers. This makes it hard when trying to discuss the concept. It is also hard to discuss challenges and best practices if there is no clear definition. Therefore, this chapter will provide context for the thesis, as well as an overview of the core concepts required to get the most out of this research. The chapter will give the reader an overview of the context and important definitions that will make the rest of the thesis much easier to follow. In the context part, the case company where the thesis was conducted will be discussed. In terms of core concepts, Continuous Integration (CI) and Continuous Delivery (CD) will be described, and given definitions that will be useful going forward. These are central to the research done and have varying definitions, so it is important to describe what constitutes CI and CD in this thesis. On top of this the concept of metrics in relation to these concepts will be discussed, and why they are considered useful.

2.1 Theory

For the purpose of preparing the reader for the concepts in this thesis this section will bring up the fundamental ideas of Continuous Integration and Continuous Delivery, as well as the definition of them used in this research. This includes describing these concepts, what problems they help solve, and differentiating them, as well as showing how CD is an extension of CI.

2.1.1 Continuous Integration

In many software projects it is common that the product is in a non-working state for extended periods of time during the development process. This is mostly the case for software projects using more traditional development processes such as a waterfall approach to software development [1]. When working like this it means that teams seldom integrate code,

which could mean weeks or even months between each integration. Working in this way leads, among other things, to a problem called the double maintenance problem, which is when multiple identical copies of software is kept [5]. Keeping multiple copies of software means that all these copies needs to be maintained. If for example a bug is found all copies needs to be fixed. When developing these copies further in different directions they will start to differ more and more. There will be different bugs and features in each copy which means that when these copies are integrated it will require a lot of effort to make sure that they work together. And the longer they have been worked on separately the more they will differ, and the more work will be required when integrating them. Traditionally what has been done by software companies is to avoid this integration for as long as possible, leaving as the final step of developing software: a "big bang" integration. However, this leads to very long integration times that can be unpredictable since there is no telling how many errors there will be or where they will come from. Bugs from many different copies can create headaches for the people integrating since there is no telling which copy they came from, or if it is a new bug created when integrating different versions. This can lead to a software project being worked on for years, and then taking months to integrate where it is not known how long it will take to finish the integration [6].

An answer to this problem is Continuous integration, a concept that was first introduced in 1999 in Kent Beck's book "Extreme Programming Explained", as one of 12 core practices for working with XP [7]. Out of these 12 CI is one of the more influential practices that have been widely adopted, along with for example Test driven development (TDD) which is the practice of creating tests before implementing a new feature or fixing a bug that tests the behaviour that is desired by the new change. The book also brings up a concept that is central to the practice of CI [7]:

> "If it hurts do it more often."

Continuous integration aims to solve the issue of painful integration by integrating more often - or actually all the time. The idea is that integrating small changes is often quick and painless compared to taking weeks or even years' worth of work and trying to integrate it all at once. Therefore, integration should happen continuously - the developers work with a local version of the software and integrate their changes to a version control system as soon as they are done with them. There are several things that are required for CI to work in a project. Some of these practices are now common among most software development companies, such as having a single source repository. For CI to work properly it requires more than just a single source repository though. It also requires that upon each integration the entire application is built and tested with a comprehensive suite of automated tests, and if these tests fail, fixing the issue should be a priority [1]. Additionally, these tests must be of a reasonable length. If it takes too long to build and test the application, it will lead to unnecessary wait times or the developers will not run the tests at all. Testing should be done quickly, ideally within 5-10 minutes [1]. More time consuming tasks like building the entire application and test that take longer to run can be run for example during the night. What is considered a working state needs to be defined as well. This is something the whole team needs to agree of. Usually this is when all the automated tests pass, and the application can be run in a production like environment, although this can differ slightly. Ideally tests should be run in the production environment, but this can sometimes be hard to achieve if certain hardware is required. CI also requires a high degree of automated tests, to make sure that

the quality of the software is kept up as new changes are added to it [6]. It is also important for developers to keep the latest version of the software locally. This means that before they can commit their changes they have to make sure that they have the latest version, and that their changes are working with it [7]. It also means that everyone must make sure to commit their new changes as often as they can, as soon as it is working it should be committed to the mainline. Working in this way will create a situation where the software is in a constant working state, which satisfies the goal of Continuous integration as stated by Jez Humble and David Farley [1]:

> "The goal of continuous integration is that the software is in a working state all the time."

This is the definition of Continuous Integration we will use in the rest of this paper. If a team can satisfy the requirement of keeping the software they are working with in a continuously working state using automated builds and tests then they are practising CI. In order to satisfy this definition the team needs to make sure that the software is working after each new change has been committed. This requires that the application be built on each committed change and a comprehensive set of automated tests are run to make sure that it is working.

There are many benefits to CI. Perhaps the biggest one is reduced risk of long integration processes [6]. Continuously integrating means that there is no process near the end of development when it is unclear if the software will be done, or if it is even going to work at all. It also makes bugs easier to find, since the code is built and runs every day meaning that there is never an insurmountable amount of new code to search through to discover the source of the problem.

Although the benefits are evident, they come with a price. The developers need to stay disciplined in maintaining a comprehensive test suite with suitable test coverage, otherwise they cannot trust the validity of the tests run on each integration, and thus, not trust that the software is in a working state. However, according to Humble and Farley, the teams that can perform continuous integration effectively will not only encounter less bugs, but also deliver software faster [1]. This does require the whole team to be on board, as well as management, otherwise it can very easily break down if developers choose not to integrate as often as they should, or management prioritises development of new features over creating tests to make sure that everything is always working.

2.1.2 Continuous Delivery

Continuous Integration is about continuously integrating software and making sure it runs. This gets rid of a lot of issues of software development, as mentioned in the previous section. However, there are still problems with software development when using CI. Humble & Farley explain that common shortcomings of CI include the following [1]:

- Operators waiting for documentation and fixes.
- Testers waiting for builds of the software that are worth testing.
- Teams receiving bug reports weeks after the context of the implemented code is fresh.
- Discovering late in the development process that the target environment will not support the implemented system's requirements.

These problems create a situation where the software take a long time to get into a production-like environment and ready for release, and makes it buggy because of long feedback cycles between developers, testers, and operators.

Continuous Delivery (CD) aim to eliminate these problems by automating the release process. CD extends CI by not only making sure that the application is always in a working state, but also that it is in a releasable state at all times. CD is the act of being able to deliver software at will, i.e. the software should be in a deliverable state at any stage and time in the development process [1][8]. Martin Fowler summarises the act of doing CD with the following bullet points [9]:

- The software should deployable throughout its lifecycle.
- The team prioritises keeping the software deployable over working on new features.
- Anybody can get fast, automated feedback on the production readiness of their systems any time somebody makes a change to them.
- It is possible to perform push-button deployments of any version of the software to any environment on demand.

Traditionally, in addition to the shortcomings of CI mentioned at the start, there would be a certain amount of time before a delivery dedicated to ensuring that the product was ready, e.g. by performing extensive bug testing, integrating development branches to a release branch, excluding certain changes not yet ready for delivery, and making sure the software runs on different production environments. Teams could take days or even weeks to test and build the release [1]. Furthermore, after a delivery new bugs could be found which needed patching, and everything in the build process had to be done again, in the right way and in the right order. In short, there would be many manual steps in the release process, making it not only prone the human factor of making mistakes, but also making it very time consuming and expensive. Additionally, the software would mostly not be tested on a production like environment until it was time for release, since building the release and setting up an environment would be too time consuming. Therefore, any environment specific failures would not be recorded until it was too late. [1]. This tedious release process would also naturally tempt the product owners to release less frequently, but since releases often give valuable customer feedback, frequent releases are desirable.

Thus, to counter these problems, Continuous delivery is used as an extension of Continuous integration [1] [8]. Again, while CI focuses on automating the development process, CD focuses on automating the whole process, including the release process. Utilising CD makes the release process quick and painless, which in turn makes it possible to release more frequently and therefore get faster feedback from the release. In addition, since the release process will be run more often, it will lower the risk of failures and that various bugs or environment specific configurations might be setup incorrectly [1]. Lianping Chen demonstrates how making the change to CD in their company had several benefits, including accelerated time to market, improved productivity and efficiency, reliable releases, improved product quality, improved customer satisfaction, and ensuring the right product is built (due to frequent releases which generates rapid feedback from the customers) [10].

In this context, it is important to distinguish between a deployment and a release/delivery. While the definitions may vary, a common distinction is that deploying is the act of configuring and installing the software on a target environment, while a release is the act of making

the product available to its entire audience. Due to these similarities, the definition of Continuous Delivery and Continuous Deployment should not be mixed together. As CD is an extension of CI, Continuous Deployment is an extension of CD [8] [1], see Figure 2.1. Humble & Farley cleared this misconception up shortly after publishing their book [1] on CD in an extended blog post on the official website of the book [11]:

> Continuous deployment is the practice of releasing every good build to users - a more accurate name might have been "continuous release". [...] Implementing continuous delivery means making sure your software is always production ready throughout its entire lifecycle - that any build could potentially be released to users at the touch of a button using a fully automated process in a matter of seconds or minutes.

In short, the goal of Continuous Delivery is to always be able to release with a push of a button, while the goal of Continuous Deployment is to automatically release every working build [8] [1].

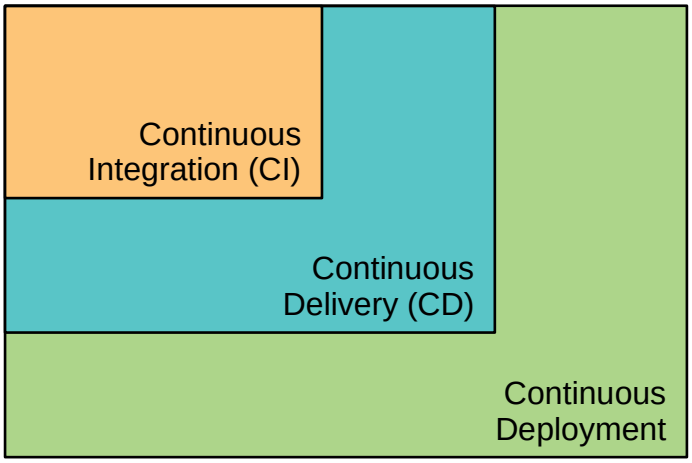


Figure 2.1: Continuous Deployment extends Continuous Delivery, which in turn extends Continuous Integration.

2.2 Case Company

The research was conducted at an office focused on software development at the company Robert Bosch AB, situated in Lund. Although Bosch is an old and well-established company, they are primarily a hardware company, and software is something they started focusing on in later years. It is also a very large company with around 400 000 employees worldwide, and about 150 in Lund. The company is known for their diverse range of hardware: Washing machines, refrigerators, power tools, etc. But they also have a large role as a supplier to the automotive industry and have a lot of other products, both in terms of hardware and software. In the Lund office they develop software products mostly related to the automotive industry and electric bikes (eBike). This takes the form of some embedded development when it comes to both the automotive and eBike projects, as well as apps for the eBike system.

The office has several products that they are working on. Each product can have multiple projects, and each project is split into teams. The teams usually work on different parts of a project, with varying amounts of overlap between teams. The teams work with the development process called Scrum [12]. Each team has a Scrum master and there is regular contact between scrum masters that are part of the same project, as well as "Scrum of Scrums" for each product. In each team there are developers that also work with testing and tool maintenance, but no set roles other than a technical lead. Each project has other members such as testers, requirements engineers, and a project lead. Some of these projects are confined to Bosch Lund, but most interact with other teams at Bosch sites all over the world, and in some cases have team members at other offices. In terms of deliveries, some projects develop a sub-part of a bigger project and as such do internal releases, while others create products that are released to actual customers. All the teams aim in some capacity to work with continuous delivery, and have progressed different amounts when it comes to reaching this goal. Their goal when it comes to working with CD is to reduce their “time to market” and improve their code quality. However, this workflow varies from project to project, since each team are independently hosting and maintaining the CD-tools of their choice, in addition to having varied experience and time to invest in setting up a CD-pipeline. There are no internal rules for exactly which tools or processes they must use in each project, so the approach they take to CD can vary a lot between teams.

2.3 The advantages of a fast feedback loop

An important advantage of using CI/CD is the ability to gain feedback at any stage in the pipeline. This means that as soon as something goes wrong, the change will be stopped and cancelled at the current stage in the pipeline, and some sort of feedback will be presented regarding what went wrong. Figure 2.2 shows a simple CD-pipeline with its feedback loop. One metric that can be important to look at when it comes to CD is lead time. This is defined as the time from when a change is first committed to when it is released [1].

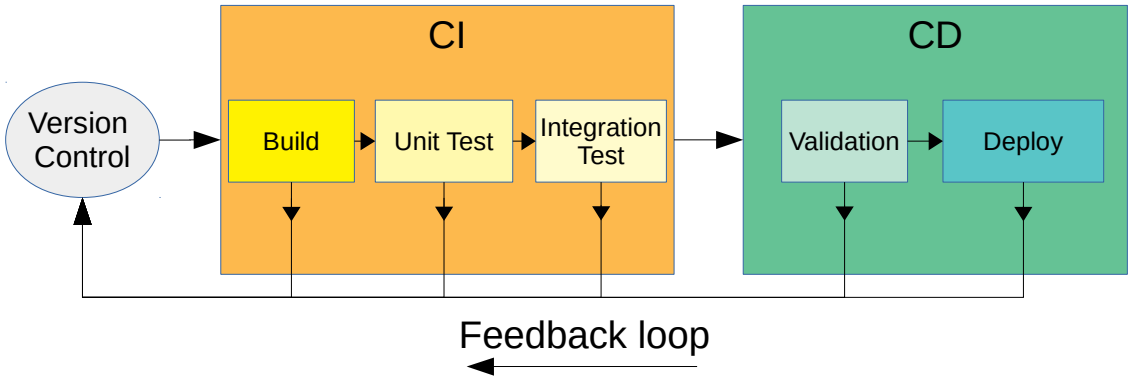


Figure 2.2: An overview of a simple CD-pipeline, visualising the feedback loop. If an error occurs at any stage, the pipeline stops and gives appropriate feedback.

Once a problem in the pipeline is fixed, the change can go through the pipeline again, starting at the first stage. This is usually referred to as a feedback cycle [1]. Additionally,

it is important to keep this feedback cycle as short as possible. The reason is that with a shorter feedback cycle, the errors can be caught and dealt with while they are still fresh in the developer's head and the context has not changed. Optimally, most faults should be detected in the automated test suite at the first stage of the pipeline, so that the feedback can be given quickly. It is therefore important that this test suite does not take too long to run. Humble & Farley recommend that this process should be shorter than 5 minutes, and in order to achieve this only the most important tests that capture the most common faults in the code should be run at this stage, and the other more time consuming or less critical test should be moved to a later stage [1]. Additionally, they also argue why the fast feedback in this stage is important for the developer:

> Errors are easiest to fix if they are detected early, close to the point where they were introduced. This is not only because they are fresh in the minds of those who introduced them to the system, but also because the mechanics of discovering the cause of the error are simpler.

However, as getting feedback from the pipeline is very important, it is equally important to present the feedback such that the people involved can actively see it. So, the feedback loop needs to be both short and visible. Additionally, actively presenting feedback and metrics can have a positive influence on the performance of a project, for example, by motivating the developers further and giving an increased ability to detect critical faults introduced to the pipeline [1]. This feedback could, for example, be presented by a big display in the office, with big and easy to see indicators of various metrics. This method could create fast, visible feedback that will motivate the developers to achieve the goal of the feedback, i.e. if code coverage is seen on the screen, the developers would be motivated to write test such that the code coverage does not decrease over time. Lehtonen et al. demonstrated how a team at their case company, with the help of a big screen in the office showing the status of the pipeline jobs, could quickly identify a problem of a release and start working on the hot-fix as soon as possible. They could also easily monitor the status of the hot-fix once it was in the pipeline. This resulted in a critical fault in the release being patched within only a few hours [8].

Of course, metrics is not only interesting for the developers, but could also be interesting for other roles in a team or company, such as operators, testers, product owners etc.

Chapter 3
Method

In this chapter we will discuss the steps involved in the research done for the thesis, divided into three sections. This includes an overview and motivation of the methods that were used, such as literature studies and interviews. Section 3.1 discusses the planned methodology at the outset of the study, Section 3.2 discusses the problems that were encountered in the planned methodology, and Section 3.3 motivates how these problems were tackled, as well as how the research was conducted instead to get around these problems. In the beginning of the project the plan included the following steps:

- Getting an overview of how the teams worked with CD at Bosch in order to find teams suitable for working with.
- Choosing two teams to study further to discover common problems and useful metrics for CD in order to help answer RQ1&2. This includes deeper interviews and gathering metrics from their CD-pipeline.
- Implementing a way to gather metrics about the CD-pipeline in these teams as a proof of concept.
- Study how displaying these metrics to the developers could potentially help improve code quality and time to market in order to help answer RQ2&3.
- Concluding and giving feedback about the findings to the teams.

However, this changed somewhat during the process due to several factors that will be discussed in this chapter. A general overview of the final method can be seen in Figure 3.1.

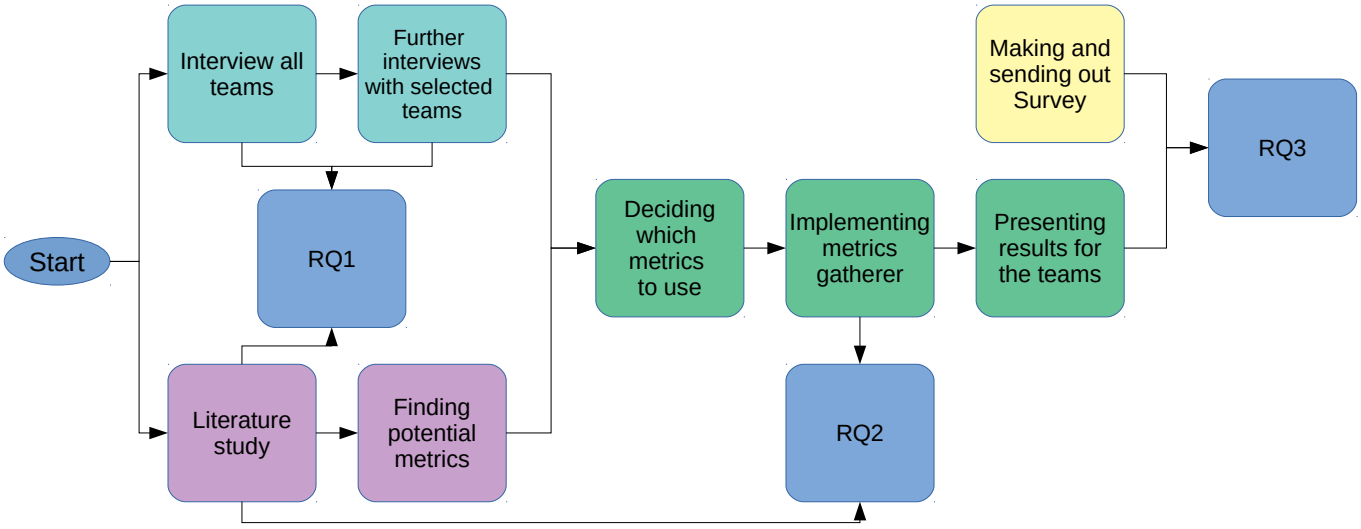


Figure 3.1: An overview of the final workflow throughout the study.

3.1 Planned Method

This section will discuss the overall methodology as planned at the start of the thesis in order to give an overview of what the original plan was and how it would answer the research questions.

3.1.1 Literature study

To get a firm understanding of the subject of CD, and to help answer the first two research questions, a study of some kind was necessary. This could, for example, mean interviewing developers at different companies who have experience with CD and ask about challenges, best practices, and what metrics are useful. However, this would require connections to these people, as well as a lot of time, so while this could be a study on its own, it is not in the scope of this thesis. Instead it was decided that a literature study would be conducted at the start of the project. This would allow us to gather a lot of knowledge in a reasonable amount of time from a lot of different contexts. We prioritised peer reviewed literature, but sometimes exceptions had to be made, where we in these cases were very critical about the authors' claims. Additionally, one other thesis with a similar scope to ours was analysed for inspiration and comparison.

There were two topics that were focused on when filtering the literature, and which were related to the research questions. Firstly, there was the topic of challenges and best practices when it comes to working with CD (RQ1). Secondly there was the question of what metrics were useful to aid in working with CD (RQ2). This could be both in terms of developer feedback, as well as metrics that could help find bottlenecks in a CD-pipeline. For this we used Google Scholar as well as Lund University's LUB-search. We used a combination of the keywords "Continuous delivery", "Continuous integration", "metrics", and "software" to find interesting papers. These were then read in different stages, and sorted out if they seemed

to not be relevant. These stages were: Read the title, then abstract, then introduction and conclusion, and finally the whole paper. The literature analysed was also discussed between the authors at two occasions. Initially, when just the abstract had been read, and finally when the whole paper had been read. There was one source that was not found this way, and it was the book called "Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation" [1] which was recommended from the start by all of our supervisors and was provided to us by Bosch, since it is the foundational text for CD.

To find data that was relevant in each of the chosen texts both of us read through them and took out data relevant to the study separately. Then these were compared and discussed if there were any deviations between the two of us. These data summaries were used when discussing the results of the literature studies in terms of challenges, best practices, and useful metrics.

We did not document how many papers we actually found and considered reading, but as for the ones we actually read further than the abstract there were a total of 20. Not all these were used, since some turned out to be less interesting than we expected after reading further.

3.1.2 Interviews

Interviews were used in two steps of the process. First, they were used as quick interviews when we needed to get an overview of all the projects at Bosch, and establish contact with members of each project who were knowledgeable about CD. Then they were used again when doing deeper interviews of selected teams to discover what data could be extracted from the pipeline, to help assist with answering RQ2.

Orienting interviews at Bosch

To see how the projects worked at Bosch and find teams that could be suitable to study further, we needed a good way to get an overview. It was decided that in order to achieve these goals interviews would be conducted with representatives for each project, and sometimes multiple people in the same project who were knowledgeable about their team's CD-pipeline. It was decided to interview each project rather than each team since it would take a lot more time to do it on a team level, and it was assumed that the teams within a project were working in a similar way. Interviews were also used instead of for example a survey because it would, in addition to gathering the data we needed, help us establish contacts with each project and get a feeling for how willing they were to work with. In finding the right people to interview we were supported by one of our company supervisors who had contacts in all the projects and as such could point us to the right people to talk to. He gave us a list of people to contact, and notified them in an email about us and what we wanted to do.

The interviews were discussion focused and had a number of questions, although additional questions were asked depending on what information the interviewee shared. They were performed by having one researcher assigned as interviewer and the other taking notes, backing up with more questions and discussion when needed. The interview questions used for this can be found in Appendix A. Each interview took about thirty minutes.

After each interview, the notes taken during them were constructed into a text of about one page summarising the key information gained from the interviewee. When all interviews

had been concluded, the notes and summaries were analysed by both researchers, discussed, and compared in order to find interesting correlations and key points. Thereafter, a summary of the projects at the office was made, discussing how they worked, problems they encountered and if there were any metrics that they thought were important for CD in some way.

From the interviews data was gathered about a total of ten teams, which covered all the projects, and in the case of the two biggest projects in Lund more than one team from the same project was interviewed. In order to determine which teams were suitable in terms of how well they worked with CD we gathered some information about their CD-pipeline, such as which tools they use, what processes they have in place, and how well they feel that they are working with CD. There is also some other useful information that was gathered from them such as how easy the teams were to contact, and how willing they were to work with us that were important factors for picking the teams.

In-depth interviews

After the first round of interviews were done it was time to pick two teams that would be studied further. There were several factors that were considered when picking the teams. First there was the scope of the thesis; there was a time and resource limit on how many teams we could study, and so it was at first decided that two teams would be about what could be managed. There were a few other factors as well that were gathered from the earlier interviews such as:

- How willing they were to work with us.
- The availability of a contact person in the team.
- How mature their CD-pipeline was.
- What kind of data was available from their CD-pipeline.

Since there was data about all these factors it was through discussion between us and our supervisors that it was decided which teams would be studied. Using the four criteria mentioned in this section for what we were looking for in potential teams to study further, there were four teams that were of interest. This was two of the teams that were doing CD, and two others that were on their way, but not quite there yet. Three of these were teams working on embedded projects, here referred to as E1, E2, and E3. Then there was one team working on an app, referred to as A1. E1 and E2 were part of the same project so in order to cover the most ground we wanted to have teams from different projects. After discussing this at length, both with our supervisors and between ourselves but it was decided that the best thing would be to look at different projects since this could give a broader range of problems and solutions for CD. At first, we considered using the A1 team since their CD-pipeline was the most mature. A less developed pipeline would already have some obvious improvements that could be made since research how to build a developed pipeline has already been concluded. Choosing a team with a well-maintained pipeline which does not have any obvious room for improvement that has already been explored a lot makes for interesting research. However, there were several problems with choosing this team though. The app team was hard to reach, had a long process to acquire the necessary access rights, and a lot of the people who knew the most about the CD-pipeline were not working in Lund. All of this was factors that made us decide to not use the A1 team.

Sorting out A1 left the E teams. Both E1 and E2 were working very well with CI, and in the case of E2, were making very good progress towards CD. For this case that meant that they were automating some parts of the release process such as building and parts of the test process, but not others like creating release notes. These teams were proven to be very easy to work with since we from the previous interview with them knew that they were easily contactable and very interested in our work. E3 did have a somewhat less mature CD-pipeline but the contact person was very knowledgeable on the topic of CD. At first using E2 and E3 seemed like a good idea, but there was one issue that made us hesitant to use E3. The person we had contacted in that team were about to leave the company, and since he was the developer who had the most experience with their CD-pipeline by far it would mean that we would need to re-establish contact with the team and hope that the next person knew enough about the pipeline to be helpful for us. This risk was a factor when we decided to not use E3, as well as the fact that they were using different tools compared to the E1 and E2 teams which would mean a lot of extra work to extract data from two different tools.

Having these issues with the other teams led us to decide that even though they were part of the same project we would work with E1 and E2.

When the teams had been selected it was time to get further information from them. In order to keep up contact with the teams, and to get the more in-depth information about their CD-pipeline that we wanted it was decided to use interviews again. These interviews were conducted in order to determine what data was available from the teams CD-pipeline, as well as getting access to the tools of their CD-pipeline so that we could explore them as well to see what data was possible to gather. The interviews gave us a deeper understanding of their pipeline, as well as an opportunity to as for access rights in person to ensure that we would get them as quickly as possible. At this point we also asked the developers if there was any data that they were currently looking at regularly as indicators, how much they are currently using metrics as part of their work, and which of the metrics we had found they would be interested in. The questions used for these interviews can be found in Appendix B.

3.1.3 Deciding which metrics to gather

There had been some previous research on the topics of useful metrics when looking to improve a CD workflow which we could make use of. From literature studies we found useful sources that were directly discussing metrics useful for CD in some way. Using these, combined with metrics that were recommended from experienced developers at Bosch during our interviews, a list was created of metrics that were potential candidates to be extracted from a CD-pipeline that can be found in Appendix C. These were all metrics that could potentially be of use, but it was not certain if they were all available. They had to be metrics that could currently be extracted from their pipeline, since implementing any new metrics would mean that a certain amount of time would need to pass before there would be enough data so that they could be studied, and this was not within the scope of this thesis. The interviews in Section 3.1.2 were used here to filter out the metrics that were unavailable, which narrowed down the list of metrics that could be extracted. When these had been filtered out the remaining metrics were all gathered.

3.1.4 Gathering the Metrics

Once two candidate teams to gather metrics from were selected, and what metrics to prioritise had been discussed, it was time to implement a way to gather metrics from their pipelines. Luckily, the case company had done something similar before for another team, leaving some infrastructure for us to use. They had through trial and error, in addition to some expertise and previous knowledge and experience, come up with a solution of gathering metrics from that team's pipeline. Thus, we decided to evaluate their solution and explore if we could use parts of it for our own. This turned out to be true, and we could reuse some of the code to setup the infrastructure more quickly.

The metrics gathered from this process were later analysed and used as one part in answering RQ2.

3.1.5 Evaluating the effects of the metric-based feedback

The metrics gathering process had left us with a lot of data to analyse. Ideally, an interesting study would be to experiment with showing this data to the teams as a form of feedback and then keep on monitoring them to see if it would have a positive impact on the health of their CD-pipeline. However, this was not possible in our case. Why this was the case will be discussed in Section 3.2.

3.2 Problems during thesis work

Due to several circumstances the planned method was not the same as the final one. This section will discuss how the problems encountered during the thesis work caused deviation from the initial plan.

3.2.1 Lack of suitable teams

Out of the ten teams we interviewed there four that were at the outset considered suitable for further studies because of the type of project that was intended. We looked at E1 and E2, but E1 was just about to start up a new project and as such they did not have much data or a fully functional CD-pipeline set up for the new project yet. This meant that only E2 was suitable for implementing metrics in, and E1 would instead be used to do deeper interviews with. This meant that some changes had to be made to the method, since the original plan was to implement metrics in two teams.

3.2.2 Working remotely due to covid-19

During the 12th week of work on the thesis the Bosch office in Lund was closed almost completely because of precautions taken due to the covid-19 virus. At this point we had decided which teams to work with further, and had started doing interviews in order to determine

what data was possible to gather from the team's pipeline. This meant that all the implementation of the metrics gatherer, as well as some of the interviews had to be done remotely. After about three weeks the office started to open up at about 10% capacity again. However, the major challenge was that after about two weeks of working remotely Bosch needed one of the laptops that we were using back. This meant that after this point some work that required the Bosch network could only be performed by one of us at a time.

3.2.3 Lack of time to observe the effect of metric-based feedback

Due to the fact that the original interviews took longer than expected, and the extra work transitioning to working at home with just one laptop, we did not have time to observe how the metrics we had gathered could affect the performance of the team. We expected a gradual change as these metrics were introduced, since the developers needed time to partly get used to them, and partly fix any problems they might find due to them.

3.3 Changes to the method

While the early stages of the thesis were conducted as planned, the problems discussed in Section 3.2 did change the methodology of the thesis somewhat. This section will discuss our changes made to the method to deal with all the problems.

3.3.1 Split into interview and implementation

In order to still gather enough information it was decided that while we would only implement the proof of concept in E2, we would still do further interviews with E1. E1 were about to change their pipeline and as such they would be asked what was good about their old pipeline and where it could be improved. On top of this they were shown the results of the E2 implementation as well in order to get some additional feedback. In this way we could still gather data about RQ1 & RQ2 from E1 through interviews, and get some additional feedback about our results from the E2.

3.3.2 Gathering feedback from Developers

Since there was not enough time for us to monitor the E2's pipeline after we had given them feedback about the result, we chose to interview them about the topic instead.

Once the findings from the gathering process were analysed and discussed, we wanted to see if the target teams could give some valuable feedback on them, as well as sanity check that everything seemed to be in accordance; if the target team would see some very unexpected results, we could have a closer look at them and double check if a misunderstanding or bug in the code could have caused it. This also gave opportunity for discussion with the teams, as they might have some valuable thoughts about the result we could not think of.

This process was conducted by making a PowerPoint presentation showing the most important and interesting graphs, as well as some of our conclusions about them. This was done

in a Skype meeting with some members of the team. Mainly it was our contact person in the E2, who also was the person responsible for their pipeline, who discussed with us. A few more team members listened in as well. We showed them a dashboard view of all the metrics, and asked if this was something which was potentially useful for their team. Additionally, since the two teams that were chosen were closely connected, we took the opportunity to show our results to our contact person at the other team as well, since this could help expand our results somewhat. He was very interested about the results and even agreed that our data helped him see some issues with the pipeline which could easily be fixed.

The information gathered in this process was used to partly explore RQ3.

3.3.3 Survey about what metrics are useful for each role

As another counter measure to not having enough time to observe how the metrics-based feedback affected the target team, we needed something short and easy which still could give some valuable data about the results, in addition to try to still answer parts of RQ3. Thus, a short survey was constructed. This survey consisted of all the metrics we had gathered accompanied with a graph displaying their result from the team, as well as a description of what they were. These metrics including their graphs description as given on the survey sheet can be found in Appendix D. Following were two simple questions:

- What role do you have in the company? (DevOps, developer, product owner etc.)
- Which of these metrics do you find most interesting for your role, and why?

This survey was sent out via email to personnel at the case company, as well as a description to why their answers were interesting to us. The estimated time to answer the survey was 10 minutes.

Chapter 4
Results

To help answer the research questions we analysed the results of the study in this chapter. This included presenting the data gathered from the study, then analysing and discussing it, and finally presenting and motivating the results. This structure was used for each section. First, our findings from the literature study and deeper interviews will be discussed to answer our first research question regarding challenges and practices for CD (RQ1). These findings will also be discussed further in Section 4.2 to help answer our second research questions regarding relevant metrics for improving the CD process (RQ2), as well as to motivate which metrics we decided to use for our proof of concept solution. Finally, to help answer our third research question regarding how the metrics found in the second research question can aid developers to optimise their CD process (RQ3), the feedback we got from developers on the proof of concept will be discussed.

4.1 Challenges and best practices of CD (RQ1)

In order to answer RQ1, literature studies and interviews were conducted. However, this was not the only reason for doing this, as will be discussed in the next section. This section will present and discuss the results of the literature studies and interviews that are related to RQ1, which will then be summarised in Section 4.1.3.

4.1.1 Literature studies

Many of the best practises and challenges of CD were presented in Humble & Farley's book on CD [1]. This book contained plenty useful information on how to succeed with CD - all being well argued and motivated for, and backed up by the authors' expertise and previous experiences on the subject.

There are some best practices discussed in relation to CI that are also useful for CD, since CD build on CI. This includes:

- Test driven development (TDD).
- The importance of automation of tests and builds.
- Checking in regularly and never checking in on a broken build.
- If a broken build is checked in it should be fixed by the person who did it.
- Not to comment out failing tests.
- Keeping builds and test processes quick.
- Having a comprehensive automated test suite.
- Never going home on a broken build.
- Be prepared to revert to previous revision.
- Giving fast feedback to developers.

These were all considered essential practices for CI, and are also mentioned as essential for CD. There were also some CD specific best practices, such as always deploying into a copy of the production environment, doing smoke tests on deployment, that the whole team own the CD pipeline and can make changes to it, and that each change should propagate thought the CD-pipeline instantly, and if any part of it fails it should stop the pipeline.

When describing some of these practices there is also a discussion of the problems that can arise if they are not followed. Without a comprehensive and fast test suite for example there could be issues that get into the code that are not discovered. If the test suite is not fast it could mean that developers lose a lot of time, start to integrate less often, or start ignoring the tests. If someone checks in a broken build it should be fixed by the person who did it, since they have the best chance of fixing it, and no one else should check in since this will take much longer for it to be fixed since they no longer have a clear run at fixing the issue. This could lead to the build being broken for a long time, and developers getting used to it being broken. Not going home on a broken build is important because otherwise it could mean that it is someone else who will have to fix it the next day, creating tensions and extra work in the team.

While this book was a great source of information, there was still a need of different studies on the subject, to get different perspectives on the topic [10][13]. The book was mostly theoretical, therefore we deemed it important to get some practical information of CD being implemented at a company.

Chen's article discussing their success moving to CD in their company proved to help fill this gap [10]. He discussed some challenges adapting a CD workflow, the biggest of them being organisational challenges, e.g. gaining access rights or root access to certain resources. Chen argued that this problem was partly solved once some organisational barriers were broken. He also discussed about process challenges. An example of a process challenge might be that a release must be approved by a board before being released, delaying it with days. Finally, he argued that there is no robust out of the box solution for CD at the time of writing, and every team/company/project must build their own pipeline. For this, many different tools might be used, and avoiding vendor lock-ins can be a challenge.

Furthermore, Neely & Stolt described their full journey towards CD in their company, from eight-week time-boxed Scrum sprints to fully functional CD [13]. Most importantly, they documented all their challenges and how they were solved during the process. From this the biggest takeaways are that there are two very important practices when doing CD. The first is automation, automate as much as possible in the build, test, and delivery. Automated tests should be "fast, solid, and reliable". The other important factor is diligence from everyone involved in the development process. This includes making sure everyone is committed to doing CD, educating developers on CD, making sure to keep the code tested by writing new tests when writing code, that everyone has a responsibility for the CD-pipeline, and respecting quality checks and gates when integrating and releasing. There were also some practices that were considered important, such as frequent commits, feature toggles, and test planning. Feature toggles is a method of building features behind a toggle so that they can be turned on or off at will for specific customers, giving the company more control of the release features. Test planning is when developers sit down with QA to plan the test that will be written for stories. According to the authors this can create a fuller understanding of the automated test suite, which means that most of the manual regression testing is no longer necessary.

In a master thesis project done at Malmö University the benefits, obstacles, and best practices of CD were studied through interviews and a literature study [4]. There were three categories of challenges that were discovered in this study:

- Lack of test automation - Tests that are hard to automate like interface tests, as well as ambiguous test results and a time consuming testing process.
- Complex environment & technical challenges - Maintenance and configuration of for example build scripts and other tools.
- Human, Organisational & resource problems - Lack of discipline, motivation, or experience among developers or management.

There were also five categories for best practices:

- Creating a reliable & repeatable process - Keep the merged code working at all times, and make the process of releasing easy to repeat.
- Using automation - In order to avoid human error, and to speed up the process, automation is key.
- Using version control - keep all code in version control, so that every change is easy to trace.
- Sharing responsibility & build in quality - Use test driven development to build quality into the code, and make every team member responsible for all code.
- Releasing Frequently - Releasing often will mean that problems will be brought forward early and can then be fixed.

There was also a paper by Laukkanen et al. describing a literature study of challenges, causes, and solutions when adopting CD [2]. In this study they did a paper review and discussed problems, causes, and solutions (some of which could be considered best practices)

when adopting CD. These were deemed relevant to our study as well. This included problems with integration, such as large commits, testing problems such time-consuming testing, and problems with hardware and UI testing. It also brought up human and organisational factors like insufficient hardware resources, a lack of motivation or discipline, and lack of experience with CD. Important best practices included: rejecting bad commits, test parallelisation in order to avoid long test times, monitoring the build time to make sure that it does not get too long, instituting social rules to make sure that developers have discipline, and training developers on the practices of CD in order to make sure that they understood why they were doing it and how they could do it best.

In a paper by Leppänen et al. benefits and obstacles of continuous deployment is discussed [14]. Since continuous deployment builds on continuous delivery it is relevant to look at what some of these challenges were. They mention manual and non-functional testing, test duration being too long, having different development and product environments, and developer trust and confidence as potential challenges that are also relevant to CD.

4.1.2 Interviews

At the outset, the quick interviews were not meant to do more than help us decide which teams to work with. However, it was quickly discovered that there were some problems at Bosch when it came to CD that seemed to be present in almost all the teams we interviewed. Therefore, it was decided to include these in the results, as well as the results from the deeper interviews.

Quick interviews

Having done the interviews there were important information for making the choice of teams that we took away from these. Firstly, it was noted how few of the teams were actually following CD as outlined in section 2.1.2. Out of the ten teams that were interviewed, only three were following (or in one case very close to following) CD. The team that was close to CD had automated parts of their release process. However, there were some small parts that they had not felt were worth focusing on at the moment that were done manually. Then there were five teams that were not currently doing CD, but were doing CI in some capacity by our definition. Another interesting thing that we took away was that a lot of the projects, and teams in the same project, were using different tools for example version control and testing. Each team also had responsibility when it came to what tools they were using, and were for the most case maintaining it themselves.

While the motivation for doing the quick interviews at the start was not to help answer the research questions there were still some useful data that came out of them. In particular, there were some trends that were noticed among the teams regarding common problems they were having which is of use when answering RQ1. Since we were looking for common problems of CD it was decided to use the information from these interviews as well. This gave such a broad range of teams where a lot of them had the same problems. There were three challenges that were brought up a lot during the interview, these were:

1. The team has to perform manual work which could be automated.
2. The team has non-sufficient test coverage.

3. The code review process takes a long time.

The reason for these issues could vary from team to team, and not all teams have these problems. The first of these problems was present in all teams. For some teams this was a big problem, with manual steps that took a lot of time, while for others it was more of an annoyance. Many of the teams seem to waste time on a lot of manual work which can be automated. The reasons vary between having lack of experience in automation, to having limited resources and time. All teams have at least basic build scripts so building the application is not a problem. The problem mostly lies in specific time-consuming tasks that have not been prioritised when it comes to automating tasks.

The second problem was brought up by five of the teams as something they were struggling with. These teams reported that they had lacking test coverage, meaning that they could not trust their test suite well enough to evaluate their code. How much of a problem this was differed among the teams. Some were doing a fair amount of automated testing, but it was not a priority in the team so there were falling behind on some of the code. Others had very few tests at all since it had not been a priority and instead the focus was on creating new features. On another note, one of the interviewees who had a lot of experience in CD faced the problem of a very bare-bones automated test suite in a team they had been assigned to working with. The team had previously been assigned to work with a legacy system, lacking unit tests. The interviewee's response to this issue was: "At this stage it is hard to change things even if someone was brought in (like a Unit test expert/QA person/Motivational speaker) since a technical debt has already been built up" It should be noted that test coverage in itself is not a good metric, since it can provide a false sense of code quality. As an example of this the same team used to have more automated tests and better test coverage but this turned out to be unit tests they had imported from a module they did not create. In other words, useless unit tests had offset the test coverage score.

As for the third problem this differs from team to team as well. Almost all teams agreed that the way that they get notified about code reviews by email is annoying. Many people add rules to filter these messages away to another folder, making them harder to notice. In addition, most teams lack a review policy other than that everyone in the team should be added as reviewers and that a commit should be looked at by two other developers. This could mean that code reviews can take a very long time, as seen in a previous study of a team at Bosch done partly by one of our supervisors that had an average review time of five days. There were two teams that saw little to no problems with the code reviews. They used two different systems to solve this problem. These were the E1 and E2 teams. E1 had a system where the one who wrote the code often sat down together with the reviewer to answer any questions. This team found out that having the author of the code there solved a lot of minor issues and sped up the review time considerably. E2 used a different method, where the commits were integrated after 24 hours if they had passed the automated tests even if they had not been reviewed by two other developers. They found that this helped increase productivity while not compromising quality in terms of more bugs or breaking the build from what they could tell.

There were also one issue that was found in two teams and it was the problem of having technical barriers to automation, such as a lack of proper hardware in the case of one team, and issues of automating tests on the custom hardware in the other.

In-depth interviews

As the E1 team was initiating a new project from scratch, we had through a discussion with our supervisor decided it would be a good idea to interview this team about their process in setting up a new CD-pipeline. Particularly interesting was which changes they would make to this pipeline compared to their last one and why. Thus, two interviews with this team were conducted. At the time of the first interview, the team had just gotten started implementing a skeleton for their pipeline, and not much actual work had been done yet. Instead they had discussed it, and had come up with a rough idea of what they wanted to implement. Therefore, a few weeks later, another interview was conducted. At this time more of the infrastructure was implemented and the team could show us some more details about their implementation and strategy. First, we discussed the positives of the old pipeline. This was the fact that it was simple so anyone in the team could change or add things as needed. Everything in the pipeline was also defined as code, including for example Jenkins jobs, so it was all version controlled and reviewed. Overall E1 was very happy with the old pipeline, but there were some things that they wanted to improve. The biggest thing was that they would have liked for there to be "more" of it. This includes adding static code analysis to each commit, more automated tests (it was a bit too bare bones in the old project he felt, especially application tests) and quality gates like memory use or time for a function to run. These were not new problems, as they had been brought up in some sense in the quick interviews. One important concept that was discussed was that of shared responsibility for the code. This was very important to the main maintainer of E1's CD-pipeline. Even though he did most of the maintenance it was important that everyone understood it and could change it if necessary.

There was also an interview done with E2 which gave a deeper understanding of their CD-pipeline, such as what tools they were using and why. While the early interview with E2 had shown some of the problem they were having this interview also showed that they were using some of the best practices that were found during the literature study. This included using automation tools to build for commit and release, and testing in a production like environment. They also had a robust automated test suite, and a shared responsibility for the tools they were using.

4.1.3 Results for challenges and best practices

There were a number of challenges and best practices that were brought up, both in the literature study and in the interviews. Some of these returned multiple times, both in the literature and when discussing CD with developers at Bosch. Here the most important takeaways will be summarised to give an answer to RQ1.

Challenges of CD

Table 4.1: Overview of challenges of CD by source

Source	Issues related to automation	Issues related to human and organisational factors	Issues related to lack of experience	Technical challenges
Interviews	x	x	x	x
Humble & Farley [1]	x	x	x	x
Chen L. [10]		x		x
Neely & Stolt [13]	x	x	x	
Hansson B. [4]	x	x	x	x
Laukkanen et al. [2]	x	x		x
Leppänen et al. [14]	x	x		x

In terms of challenges there are a some that are strictly related to CD while others are challenges for CI as well. There are two challenges which are brought up directly or indirectly in every paper and interview: Lack of automation, and the problem of not committing fully to CD. In table 4.1 the sources are summarised based on which challenges they included. The challenges have been boiled down to four essential categories that all smaller challenges can be categorised in.

The first challenge, which is brought up over and over as a common problem the teams were having at Bosch, was that automation was not always prioritised. As previously discussed, it is essential to automate as much as possible to effectively be able to integrate or deliver continuously. When it comes to automation it should be considered essential for CD, and that is true for all stages of the pipeline such as testing, building, and delivering.

The other challenge consists of a lot of smaller problems. For example, a challenge could be developers not having enough knowledge to do CD properly as is described as one of the early problems in the paper by Neely & Stolt [13]. Or it could be management not wanting to commit enough resources, which could lead to the first problem again. It could also be developers not looking at the feedback that they get from the pipeline, ignoring warnings, or bypassing quality gates.

Then there were some smaller challenges that were not present in every paper or interview, but were still found to be important. For the interviews at Bosch this was the code review process taking a long time. This meant that integration could take a long time, and that new changes pile up. There were some teams that had found ways around this, as was discussed above, but it can be a problem if it is not handled well. Then there are technical challenges as described by Hansson [4], and by some of the teams at Bosch. Problems with testing on certain hardware, or doing for example interface testing can slow down the work of automating tests, or if there is not enough experience in a team it could hinder them from doing CD altogether. The speed of the automated tests can also be a technical challenge. As discussed earlier, at commit the test suite take no more than ten minutes to run, preferably shorter. To keep this down some technical solution might be necessary.

Best practices of CD

Table 4.2: Overview of best practices of CD by source

Source	Committing to doing CD	Utilising Automation	Shared responsibility of the CD-pipeline	Test driven development
Interviews		x	x	x
Humble & Farley [1]	x	x	x	x
Chen L. [10]	x	x		
Neely & Stolt [13]		x		x
Hansson B. [4]	x	x	x	x
Laukkanen et al. [2]	x	x		
Leppänen et al. [14]		x		

A lot of the best practices could help avoid or solve the problems discussed above. In table 4.2 each source and what best practices they recommend can be seen. Overall, making sure that everyone is on the same page when it comes to wanting to do CD is essential to making sure that it will work, as discussed in [13][10][4][2]. This is a part of a solution to many problems, such as lack of automation, and a lack of discipline. There are other important best practices as well such as keeping and maintaining comprehensive test suite. One way to do this though test planning was described, and while it has not been brought up in any other literature that was used it does seem like an interesting concept that could be studied further. In order to keep everything running smoothly it is important to make sure that automation is prioritised. Test driven development is seen as a given and essential practice in order to keep automation up to date by all the sources we looked at, and all the teams doing CI and CD at Bosch were practising TDD. Having shared responsibility for the CD-pipeline is also an important practice, since it empowers developers to make changes and fix problems themselves according to the interviews we had with E1. This is also brought up in two of the papers [13][4], and in the Continuous delivery book [1]. One more thing that every source agrees on is to never commit broken code, or remove failing tests. Making sure that quality gates are respected in order to make sure that the quality of the code stays high is essential to doing CD well. The importance of fast feedback to developers is also discussed as an essential practice in the CD book, and is touched on in the papers, but not as an essential factor. This is also true for always deploying into a copy of the production environment, and smoke testing deployments. Two ways of helping with cutting down on code reviews were found during the interviews, and this is not something that is discussed in the articles, partly because code reviews are not considered an essential practice for CD. However if code reviews are a part of the CD process, as it is at Bosch, then some ways to help shorten these could be useful. How well these work could be a potential topic for further study.

4.2 Metrics for improving a CD process (RQ2)

To determine metrics that could be useful for improving the performance of a CD-pipeline a literature study and multiple interviews were conducted. Doing this was a way of answering RQ2, which was concerning relevant metrics of improving a CD process. Thereafter, a solution for gathering and displaying data for these metrics form an industry team was explored. This backed up the findings regarding RQ2, and later aided in answering RQ3, which was addressing how the metrics found in RQ2 could support developers to optimise their CD process. This section will discuss all of these steps in detail.

4.2.1 Literature studies

Although a bit limited, some previous research about useful metrics in the context of CD had been performed. Following is a summary of information gathered from studying three different sources: a study by Lehtonen et al. [8], a discussion from Humble & Farley's book on CD [1], and finally a systematic literature review performed by Kupiainen et al. [15].

Lehtonen et al. conducted a study where they defined a number of metrics which could be useful for improving the performance of a CD-pipeline [8]. Their findings were the following:

- Development time: the time taken to develop a new feature.
- Deployment time: the time it takes to deploy a new feature once its implementation is complete.
- Activation time: the time it takes for a user to activate the new feature after deployment.
- Oldest done feature: the time a feature has been developed, but not yet deployed to a production environment.
- Features Per Month: the number of new features released per month.
- Releases Per Month: the number of releases per month.
- Fastest Possible Feature Lead Time: the amount of time a feature spends in the build and test phase.

The usefulness of these metrics were backed up by both a literature study of various papers related to the topic, as well as a practical single case study at a company. Therefore, their validity seemed high enough for us to include them among desirable metrics to monitor.

Humble & Farley suggested the following useful metrics in their book on CD [1]:

- Cycle time.
- Test coverage.
- Amount of duplicated code.
- Cyclomatic complexity.
- Afferent and efferent coupling.
- Number of warnings.
- Number of defects.
- Number of commits/builds/build failures per day.
- Duration of build, including automated tests.

The authors argued well and motivated for how these metrics are important in a CD context. These augmentations and motivations were well in line with optimising a CD workflow. Additionally, the authors have a vast amount of real-world experience on the topic, and while this does not mean we should trust them blindly, we should at least consider their claims with the fact that they are based on experience. The key here is that the motivation for using these metrics makes sense, therefore we should consider using them.

Kupiainen et al. performed an extensive systematic literature review (SLR) on the topic of how metrics are used in the industry [15]. This resulted in the following representation of categories for metrics used in an industrial agile software context:

- Iteration Planning: metrics which helps to choose which tasks to prioritise for the next iteration.
- Iteration Tracking: metrics used for tracking the status of the selected tasks during an iteration.
- Motivating and Improving: metrics used for motivating people and improving practices and performance on a team level.
- Identifying Process Problems: metrics used to spot problems in processes and workflows.
- Pre-release Quality: metrics used to assure and evaluate the quality of the software before it has been released.
- Post-release Quality: metrics used to assure and evaluate the quality of the software after it has been released.
- Changes in processes or Tools: how applying metrics led to changes in the processes and tools used.

In their SLR they put high demands on the fact that the literature should contain findings from actual industry examples with empirical findings, as well as why and how they use them. Although the scope was not CD, but agile software development in general. However, since CD requires agile software development, these findings should be at least somewhat useful in CD.

4.2.2 Interviews

There were some metrics that were recommended to us by experienced developers at Bosch. This happened during the quick interviews at the start, and during the deeper interviews following. At this occasion, we didn't present any of the metrics we found in the literature study to the developers. These metrics were recommended by them without any directions from our side, except us asking them if their pipeline currently presented some metrics they found useful. From the quick interviews we were recommended the following metrics:

- Mean time to restore
- Delivery lead time

- Delivery frequency
- Review Times
- change failure rate
- Which reviews have waited the longest
- Intermittent errors (number of failed builds/number of builds)

And from the deeper interviews there were a few more that were suggested:

- Cyclomatic complexity
- Split errors into environment/compilation errors
- Test pyramid- Where are the errors found in the pipeline

While the importance of these metrics only were opinions from developers experienced in the field, their experience is not to be taken lightly. Together with the fact that that these people find these metrics useful, and that for example cyclomatic complexity were found in the literature as well, must mean that these metrics have at least some significance. While some of these metrics might be highly personal, some of them might also be very general regarding the context of personnel involved in the CD-pipeline. For this reason, we put some trust in these peoples experience in the field and considered these metrics as potentially useful.

4.2.3 List of metrics used

The literature study and interviews resulted in a list of potentially useful metrics which could be used for improving a CD-pipeline. This list can be found in Appendix C. These metrics can be regarded as a partial answer to RQ2. However, to separate the potentially important ones and the actual important ones, we considered it necessary to do further studies. Implementing monitoring of these metrics on E2's pipeline, followed by evaluating the result, served as a way to explore the importance of these metrics. If the metrics proved to be important for this team, it might also be important for other teams. However, as mentioned in Section 3.1.3, the metrics had to be possible to gather from E2's pipeline. This meant that some of the metrics were not usable for this study. This narrowed down the original list to a total of 11 different metrics, all of which were implemented. There were a number of reasons that the other metrics were removed, such that they have fixed two week sprint so metrics like delivery frequency is almost always the same, and that some metrics were not available such as cyclomatic complexity. Ultimately, metrics related to code quality were either very hard to implement or unavailable. This resulted in the chosen metrics being solely focused on lead time and not code quality. Thus, the final list of metrics used were the following:

- Number of commits per day
- Number of successful builds per day
- Number of failed builds per day
- Build on commit time (as well as sub-build time)

- Intermittent error ratio (number of failed builds / number of builds)
- Review time (uploaded change -> approved code review)
- Merge time (uploaded change -> merged), also known as Development lead time
- Delivery time (merged -> delivered)
- Oldest done feature
- Mean time to restore
- Features Per Month

4.3 Proof-of-concept: implementation and analysis (RQ2)

When the potentially useful metrics from the literature study and interviews had been established, we wanted to implement them on a team to work as a proof of concept on how use these metrics to improve an existing pipeline. This process along with it's result will be discussed in this section.

4.3.1 Implementing metrics gathering and visualisation

Employees at the case company had performed a pilot study gathering metrics from a team before. Their idea was simple and straight forward. A limitation was that they did not want to disturb the performance of the team they were monitoring, either by touching their code base to implement metrics gathering directly in their software, or by affecting the performance of their CI/CD-pipeline. With this in mind, their implementation had become a simple Python script which contacted the APIs of the pipeline tools, and gathered metrics over time as a job in Jenkins. In turn, this script saved its result to a MySQL database, which was connected as a data source to an open source application called Grafana. Grafana was used to visualise the data graphically.

Since we had the same limitation regarding affecting the performance of the monitored team (i.e. by not touching their code-base or throttling their pipeline tools), in addition to a lot of the infrastructure already being set up and ready to be reused, we chose to adopt a similar approach to theirs. That is why we used their Python script as a starting point. After extending this script a bit for gathering some of our specified metrics, in addition to getting it to work with our target team, the visualisation process was yet to be implemented. For this, we reused the MySQL and Grafana server instance setup by the previous project. Figure 4.1 shows an overview of the gathering process: relevant data from Git, Gerrit and Jenkins are processed in the local machine and then uploaded to MySQL, which in turn is used by Grafana to visualise the metrics. As a starting point, a new database was setup in the MySQL instance. In addition, SQL-scripts to make and populate the tables were created and ready to

be reused. Since this was only a small-scale scripting project, there was no need to contemplate an advanced optimal structure for the SQL-tables. At this point, there were two types of structures of data to gather: data on a per-day basis and data on a per-change basis. Thus, two simple tables were created for each of these data structures. Both these tables mainly consisted of various timestamps of events happening in Jenkins and Gerrit. The timestamps were chosen in accordance to extracting the desired metrics. These timestamps were later synthesised in the SQL-queries to form durational data for the metrics, i.e. the difference between two timestamps gives the duration of a process; e.g. for review time, this would be the difference between the change was uploaded until it was reviewed. Unfortunately, there was no data to see when a developer started doing a review, but only when they finished the review by giving the change a score (+2 for accepted). Therefore, the idle time, being the time between a change was uploaded until someone started to review it, could not be excluded. This left the review time being the idle time plus the time spent on reviewing.

Once the tables were set up, the Python script was modified to output a text file containing the data for each table. These text files could then easily be loaded into the tables in MySQL. The reason we used text files instead of directly connection to a MySQL cursor via Python was because the server machine the MySQL and Grafana instances were running on had some limitations. Firstly, since the server was not owned by us, and the employees at the target company were using it frequently for various tasks, we could not install the required Python version on it to run our script. Secondly, the server did not allow connection remotely to the MySQL instance on it. Therefore, we had to manually copy the table data to this server using SCP, and locally populate the tables with it.

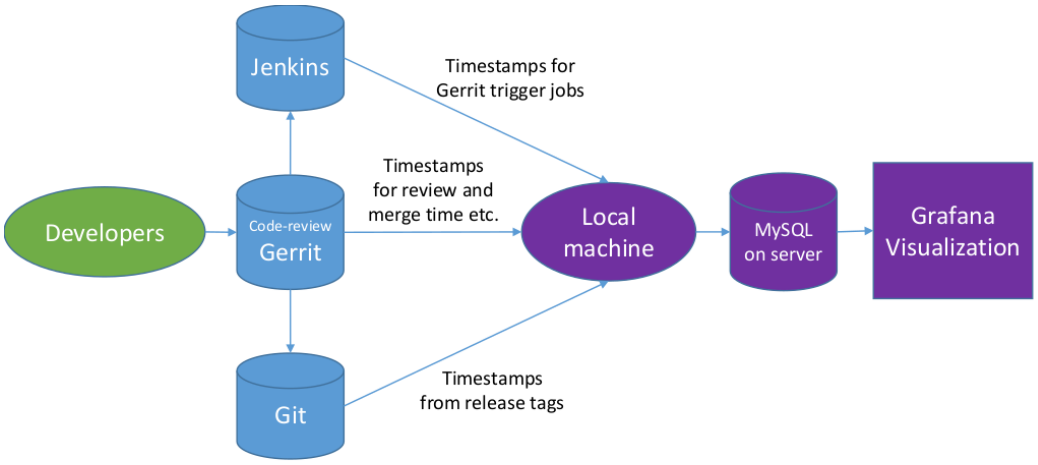


Figure 4.1: An overview of the metrics gathering process.

Once the tables were populated, the MySQL database was connected to Grafana, where each graph was querying the database for specific data, as well as being configured to our liking. The configuration of the graphs included choosing how the data should be visually represented, for example as a bar or line graph, or if multiple Y-axis should be used etc. Grafana proved to be very straight forward and easy to use, and the process of adding a new graph was simply to make a query to the database and then tinker around with the settings in the application until the visual representation of the data was, in our eyes, easy to interpret and gave a good overview.

After this Python-MySQL-Grafana workflow had been set up, it was time to extend the

functionality by implementing more metrics to be gathered. This was done by extending the Python script to find and output more data. The script then updated the tables in the database, and finally configured Grafana to show the new metrics, in our opinion, in a pleasing way. Once all the desired metrics had been implemented, two Cron-jobs were created to update the database with new data each day. One job running on the server with the MySQL instance, and one job running the Python script on a machine owned by us. The Cron-job on our local machine simply ran the Python script and saved the result to files. The job on the server machine ran the SCP command to copy the results from the Python script, and then opened a connection to MySQL and updated the tables. This script was run each night at 2:00 am and gathered the data from the previous day.

Finally, when everything was set up, it was time to analyse and evaluate the results by looking at the various graphs visualised in Grafana. However, we quickly noticed that we needed to have a deeper understanding of the Build Time metric, which measured the time taken for all the Jenkins jobs to run at commit. The reason behind this being that the metric did not give us enough information; if one of the jobs accumulated in this metric were causing a bottleneck, we would not be able to track it down to this specific job. To solve this issue, we created a new table in the MySQL database with each of these jobs and their timestamps for start and end time as unique elements. In this way, we could create a graph for each job displaying their duration taken, and had effectively split the Build Time metric into smaller parts.

When everything was set up, and all desired metrics to monitor were implemented we knew there was no need to monitor the metrics at a daily basis and wait for result, because all the data needed for the metrics was already there. This meant that we did not have to wait for results, as they were already available. Instead, we could get metrics from when the project was first started until today. We decided that gathering metrics 7 months back until today would suffice, therefore we gathered data from 2019-10-01 until present (2020-05-01). A longer time period could introduce variation in the team's workflow - for example, they could have changed a few team members or introduced or tried a new workflow strategy. On the contrary, a too small time period would limit the amount of data that could be found, and eliminate the chance to see an overall trend in it. We deemed about 7 months was a sufficient time period for this team. This amount of time would include a good amount of releases while still being small enough to limit some risk with variations of the team's process.

At last, we analysed the results by looking at various trends in the graphs in addition to comparing certain metrics to others to see if there were any correlations. We explain our findings further in Section 4.3.2 below.

4.3.2 Analysis of the gathered metrics

With the metrics gathering process complete, the graphs and data could finally be analysed. As recommended by Humble & Farley [1], as well as one of our supervisors at the target company who had years of experience with this field of work, the best way to find bottleneck in a CD-pipeline is to first look at the overall cycle time to see which part is throttling. When we had an overview, we could have closer look at smaller parts. This method would be used as opposed to having a closer look at all the parts, and thus save a lot of time. With this in mind, an overview of the E2 team's pipeline was made, where each step showing its average time taken, or for the case of Nightly builds and Release, estimated time taken (see Figure

4.2).

The reason Nightly build time was only estimated was that the team did not report any problem with nightly builds running too long, and the time spent gathering data from all nightly builds in order to get an average was not worth it. After all, since they are run in the night when no one is working, if they are completed until morning there is no need to optimise them for efficiency. Instead, a few nightly builds were manually looked at to estimate an average.

The reason behind the release time being estimated was that it was partly a manual process with the manual release note creation as a bottleneck, so the information about how long this took was taken from the interviews with the team.

However, the time a change spends in Gerrit (mainly deduced by time it takes to code review), as well as the time a change spends being built in Jenkins are actual averages of the data gathered.

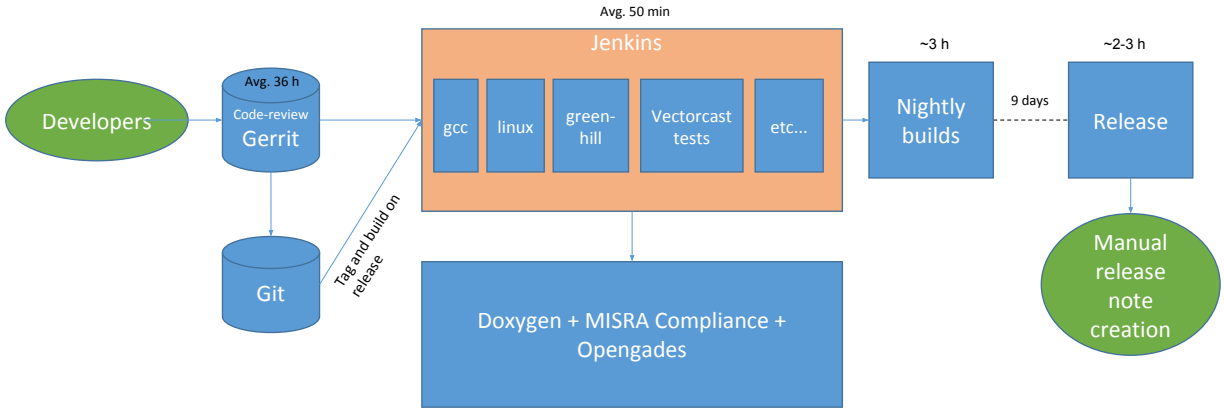


Figure 4.2: An overview of the pipeline of the E2 team, including duration taken for each stage.

Now, having an overview of how much time each stage in the pipeline takes on average, it was time to find a bottleneck. We began looking at the stages taking the longest time. First, since this team are doing Continuous Delivery and not Continuous Deployment, delivering once every fortnight, the average release time of 9 days should not be optimised. Simply because the team nor their customers want to release more frequently. Following in longest duration taken is the time spent in Gerrit. This is the time between a change being uploaded until being merged to master, thus including the review time. There are three requirements for merging a change to master:

1. The build must be successful.
2. It must have been given a +2 in code review.
3. Someone must press the "merge" button once the previous two requirements are fulfilled.

The graph in Figure 4.3 shows the average time taken between a change was uploaded until given +2 in Gerrit, in this case 1.369 days, or 33.504 hours. The graph in Figure 4.4 shows the review time in addition to how long it took for it to be merged, in this case 1.499 days, or 35.976 hours (idle time included). Since a change cannot be merged until it has been given +2

in Gerrit and the build has passed, the average time of both processes should be subtracted to the merge time of 35.967 hours. This in order to find out how long it takes for the change to be merged once it can be merged. With an average build time of 50 min, and a review time of 33.504 hours, this leaves us with an effective merge time of 35.967 hours - 33.504 hours - 50 min = 1.629 hours. In other words, it takes on average 1.629 hours for a change to be merged once it can be merged. However, coming this far, is this something worth to optimise? It could be very possible that this average is greatly affected by the idle time a developer stays at home, e.g. imagine the scenario where developer A uploads a new change. After passing the build process, developer B reviews their code and gives it a +2. However, this might be at the end of the day and developer A might already have gone home for the day. This means that developer A will not merge the change until the next day, causing the effective merge time to take hours. These instances are very likely to happen since the personnel at the case company has flexible working hours, and might therefore have a big effect on the average value of the effective merge time. Furthermore, one could argue that there is little room for improvement for the average merge time being 35.967 hours. This time is both skewed by developers not actually being at work, and longer lasting holidays and weekends. Additionally, when asked about it the target team agreed that they did not have any problems with changes taking too long to review and merge.

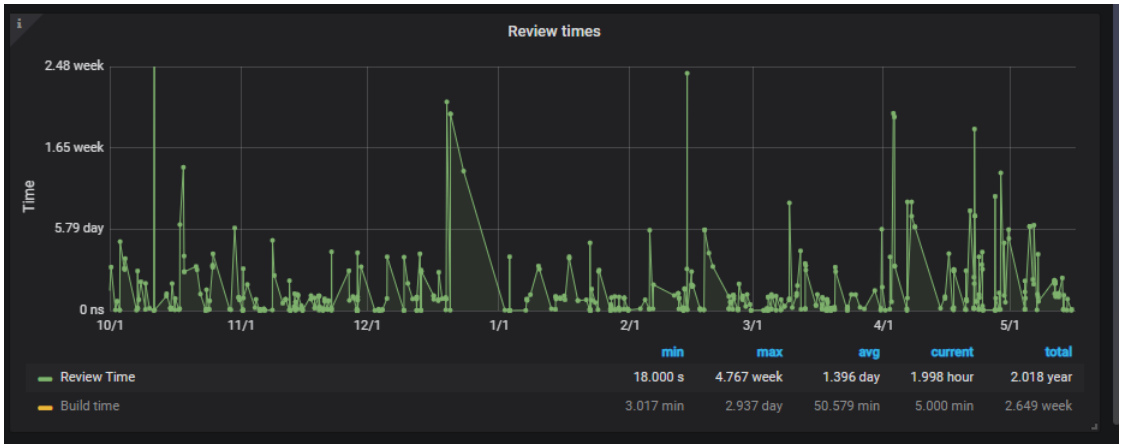


Figure 4.3: A graph showing the time taken from a change was uploaded in Gerrit until it was approved in a review.

As mentioned previously, optimising the nightly build would not be necessary either, since they all ran in the night and it did not affect the efficiency of the pipeline. Furthermore, the release time was mostly manual, and the team did not have any interest in automating it at the moment.

This left the built time of 50 min to be optimised. As remembered from the literature study, Humble & Farley recommend a build time lower than 5 minutes [1]. This is because it is important for the developers to get feedback as soon as possible, which was argued for in Section 2.3. Looking at Figure 4.5, a clear correlation between failed builds and an increased number of patch sets could be seen. We argue that this might mean that when a build fails, the developer uploads a new patch set, and with more failures, more patch sets will be uploaded. This would cause a developer to have to wait for each build to complete before they know if their new patch set will pass the build process or not. In some cases, it could take multiple failed patch sets until the fault was fixed. Additionally, having an average build success ratio

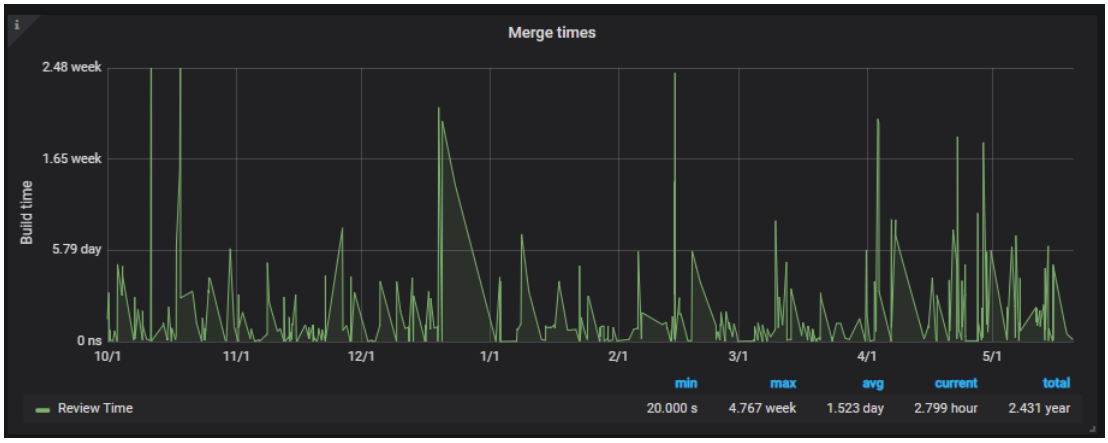


Figure 4.4: A graph showing the time taken from a change was uploaded in Gerrit until it was merged to master.

of 60% (see Figure 4.6), mean that the build fails 40% of the time. Of course, it might also be the other way around, i.e. more patch sets cause load on the building process which causes it to fail. We were open for discussion about this topic and decided to ask the E2 team if this is something they have noticed developers doing. Details and conclusions about this meeting will be discussed in Section 4.4.1.

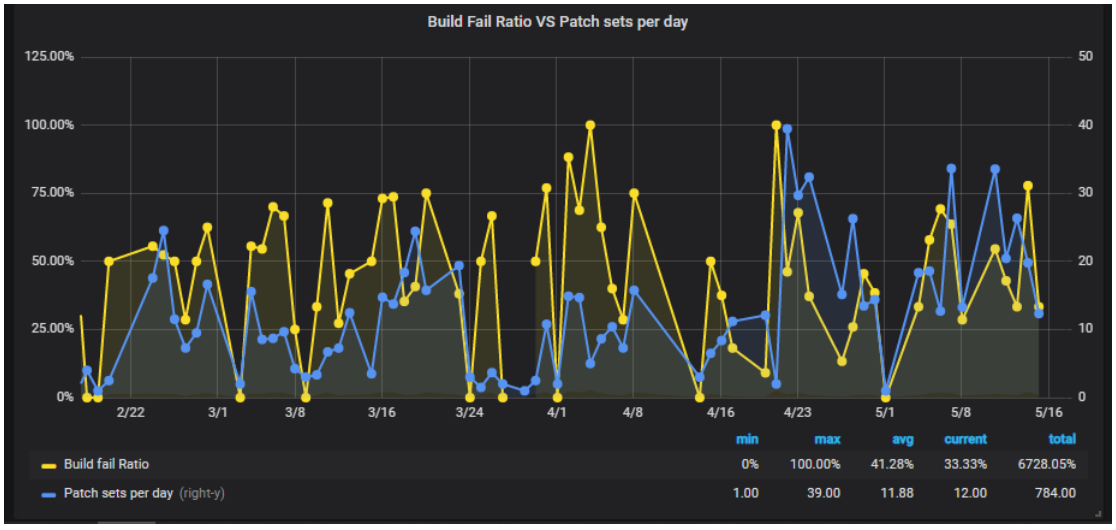


Figure 4.5: A graph comparing the build fail ratio (left Y-axis) with the amount of patch sets per day (right Y-axis).

To optimise the build time, we decided to sort the sub jobs of the build in terms of total time taken. The reason for this would be that the builds with the longest total time, would have the biggest effect on the average time taken, since they either are run often or take a long time to run, or both. Figure 4.7 shows a graph highlighting the two sub-builds which had the longest total time; Opengades and Vectorcast. Opengades is used to generate documentation, and Vectorcast is used to run unit tests. Looking closer at these two sub builds, the issue with them were not apparent. We therefore decided to discuss this with the E2 team while presenting the data to them. This is discussed further in Section 4.4.1.

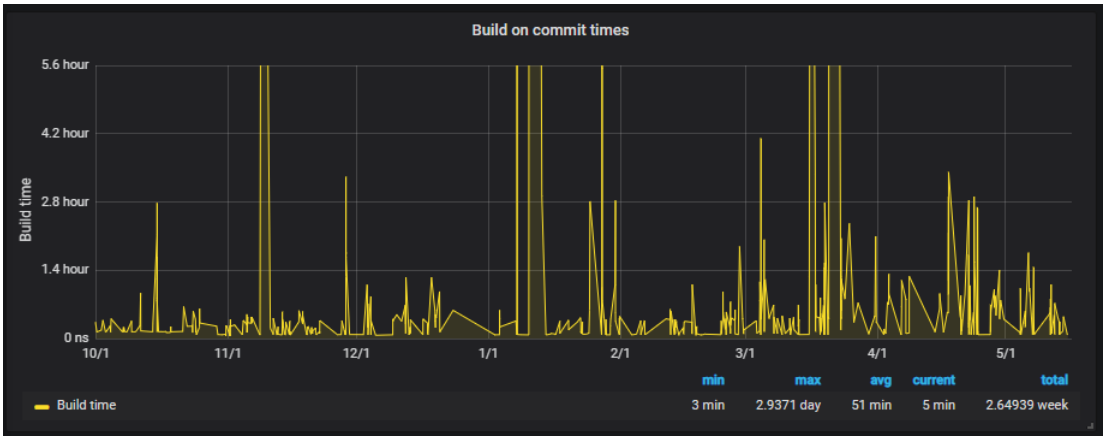


Figure 4.6: A graph showing how long it took to run the on commit build process for each change.

We chose to limit ourselves to two sub builds due to time limitations, both from our side and the team's. If we wanted to analyse more sub builds, we would have to either go deep in their toolchain and do some time-consuming detective work, or interview the team about them. Since we could not bother the team too much, as they had a lot of work to do and deadline to meet, we chose to limit the study to two sub-builds.

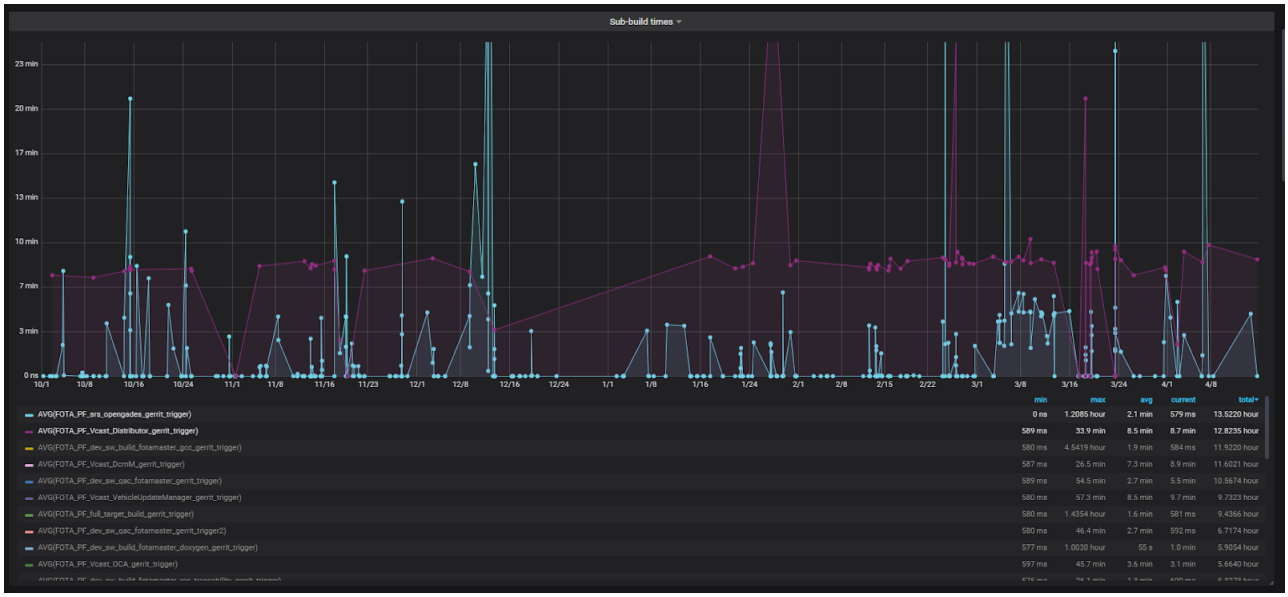


Figure 4.7: A Graph highlighting the two sub-build with the highest amount of total time taken. The table shown is sorted on total amount of time taken.

Another interesting find was a correlation between the number of patch sets per day and build time. As seen in Figure 4.8, which plots the amount of patch sets per day against the average build time, there is a correlation where the average build time increases with the number of patch sets. This means that the build times increase with more load to the system. This possibly mean that there is an idle time in the process where a job waits for access to

a resource. Again, this is something we deemed necessary to discuss with the team. The conclusions from this discussion will be explored in Section 4.4.1.

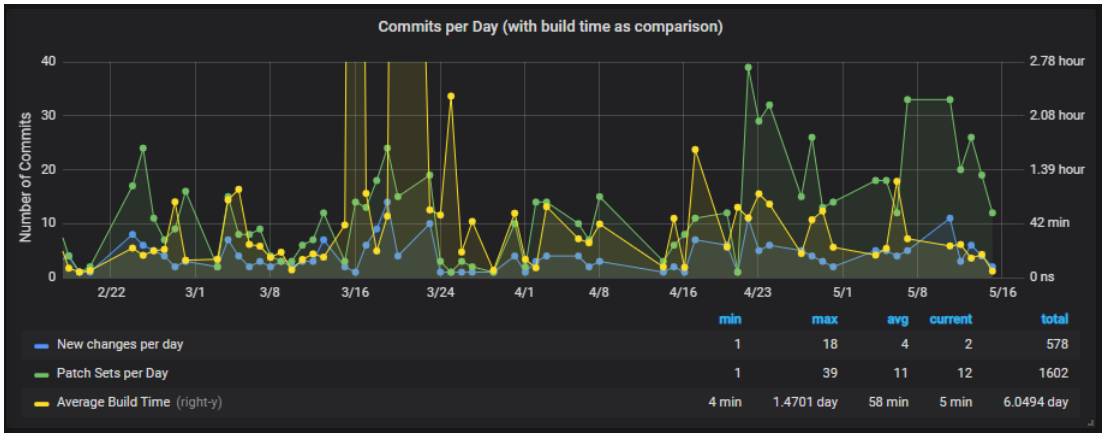


Figure 4.8: A graph showing the correlation between build on commit times, patch sets per day, and amount of new changes uploaded per day.

Outliers in graphs

There are two graphs with some clear outliers in them; Review time (Figure 4.3) and Build time (Figure 4.6). As for Review time, it is not always apparent what these outliers could be. Some of them might be caused by developers uploading unfinished changes to later continue work on them, while others might be caused by national holidays. Other possible reasons could be caused by the team having a high workload, and some of them might just be general outliers. However, there are two cases where we know that most of the or all the employees at the case company are having holiday. The first occurrence is around Christmas - we can see that barely any changes have been uploaded around the end of December, in addition to a big spike in review time for the ones that very uploaded right before people went on vacation. Another known holiday is around Easter, where many employees utilises their vacation days in addition to the already work free days. Thus, the same spike around Christmas can be seen around Easter (in the beginning of April), but smaller. As for the other outliers, there is not enough data for us to draw any conclusion.

However, it is apparent that from the beginning of April and onward, there are a lot more spikes in review time compared with before. One explanation is that the developers started working from home in March in accordance to preventing the spread of the Covid-19 virus. Working from home might have limited their communication capabilities, since they no longer worked at the same office and had to communicate digitally. This in turn can make it harder for them to make their colleagues notice when they want their code reviewed. While working at the office, they were seated closely together, and it is possible they could, for example, just nag a bit on the developer sitting next to them to review their code. Working remotely like this naturally becomes a bit harder, unless they have a tin can telephone connected to their friendly neighbourhood developer, which is quite unlikely considering the outdated technology. However, when we confronted the team about this, they said that they had recently hired 6 new consultants as developers. This meant that the prolonged review

times was because these new employees had yet to get acquainted with their code-base and coding style.

The other graph showing outliers, Build time (Figure 4.6), may have a more apparent reason as of why. Most of these outliers can be cross references to Figure 4.7, showing the two most critical sub-builds. However, these sub-builds do not take nearly enough time to justify the big 5h+ spikes of the Build time. This is an indication that there are more bottlenecks in the building process, and more sub-builds must be analysed to find them. Unfortunately, analysing this further would take some time as we would need to discuss more sub-builds with the E2 team, since we do not have enough information to draw any conclusions about them, and there simply was not enough time for more interviews to be made. Another possibility may be unrecorded idle time. Though, this is unlikely since we use timestamps to record the build and sub-build times, so the idle time would be included in the duration between two timestamps.

4.3.3 Metrics for Optimising Lead Time

To answer RQ2, the metrics to be regarded as important as of our study on the teams would arguably be the metrics used for finding and recording the bottlenecks. First, all the metrics used for narrowing the point of optimisation of the pipeline down to the Build time should be regarded as useful. These were the metrics which gave an overview of all stages of the pipeline, and helped us understand which of these stages needed optimisation the most. Simply put, they are useful because they helped us find the bottlenecks. These metrics would therefore be the ones covered by Development lead time and Deployment time. Additionally, Sub-build time should also be regarded as important, as it was used to find bottlenecks in the Build time. This left us with the following list of important metrics gathered from the practical part of the study:

- Development lead time
  - Duration of build, including automated tests
  - Sub-build time
  - Review time
- Deployment time
  - Nightly build time
  - Release time

To strengthen this result, these metrics can be correlated to a study by Kupiainen et al. [15] discussed in Section 4.2.1, where they explored which metrics are actually used in the industry by doing a systematic literature review, resulting in a seven categories of metrics actually used in the industry. Among these categories, one of the categories is related to the above metrics, being:

- Identifying Process Problems: metrics used to spot problems in processes and workflows.

All the metrics above should belong to the Identifying Process Problems category, as they can be used to spot problems in processes wan workflows, e.g. if the Duration of build or Nightly build time metrics are too high, there is a problem with the building process, and if the Release time is too high, there is a problem in the delivery process.

Finally, to answer RQ2, these metrics above proved to be relevant in improving a CD process by our case study. The metrics gathered from the literature study, shown in Appendix C, may also be relevant in improving a CD process.

On another note, how the metrics gathered in the metrics gathering process were used in aid of answering RQ3 will be discussed in Section 4.4.

4.4 Feedback from developers (RQ3)

When the metrics had been gathered and analysed it was originally planned to show the E2 team the gathered metrics, and use them to give the team feedback from the CD-pipeline. Thereafter, how this metric-based feedback affected them would be observed. This would aid in answering RQ3, regarding how the metrics found in RQ2 can support developers to optimise a CD process. However, due to time limitations this was not possible. Instead, a presentation of the findings held for the team, as well as a survey was conducted to record opinions about usefulness of the metrics. Additionally, these interviews gave some further insight of the sub-build bottlenecks, which were relevant to answer RQ2.

4.4.1 Presentations to the teams

As mentioned in Section 3.3.2, the results and conclusions from the metrics gathered from the E2 team was presented at two occasions. Firstly, to the E2 team themselves, and secondly to the E1 team closely connected to the E2 team.

At the presentation with the E2 team, we proposed the idea that their build time might need to be optimised. We motivated this with the following two reasons, previously argued for in Section 4.3.2:

1. Increased patch sets cause higher build failure (or the other way around).

2. The average build on commit time of 50 minutes is too long.

Regarding the first reason, the spokesperson (who also is the main maintainer of their CD-pipeline) agreed that there is a correlation that build failures causes more patch sets. After all, this is very logical since a build failure indicates that something has gone wrong, and a way to fix this is to patch the code by uploading a new patch set and build again. He also agreed that it is pretty common that developers have had failing build which had caused them to upload new patch sets over and over again until it was fixed, each time waiting for the build to finish. He agreed that a lower average build time was desirable when we backed this claim up with the second argument.

Now agreeing that the build time should be optimised, we gave the team a closer look at the build process by showing the two sub builds (Jenkins jobs) which had run for the longest total time. One of them was a job generating documentation with Opengades. The other one was a unit test job, running test with the Vectorcast software. Unfortunately, there

was not much discussion about the Opengades job, so no valuable information was gained regarding it. However, the Vectorcast job was discussed thoroughly. It turned out that they only had one licence for the Vectorcast software, which resulted in only one job at a time being able to use it. This caused a bottleneck where jobs were constantly in queue waiting to get access to the resource, which in turn naturally increased the average time for this job. The job itself did not take too much time to run, it was the waiting process which increased its duration significantly. It is important to remember that we did not record the time taken for a process in our metric gathering process, but instead recorded the timestamps from between a job was initialised until it was done, in other words the duration of the job. With this technique, we also recorded the idle time of the job, i.e. the time it had to wait until it was ready for processing. If we had used this method, we would never have noticed the bottleneck caused by this Vectorcast job having to wait in "queue". Therefore, it seems like recording the whole duration was a fruitful decision. Otherwise there would be gaps in the pipeline with unrecorded data. Nevertheless, the spokesperson from the team admitted he was aware of this limitation, but he did not know it had this big of an impact as shown by our results. Therefore, he said he would be considering a proposal to get more licenses for the Vectorcast product.

The shorter version of the presentation was also given at a later occasion to our contact person in the E1 team (who was responsible for his team's pipeline). He reacted to the Opengades build, which had barely been discussed at the presentation for the E2 team, and said he was surprised it took so much time to run. When thinking further about the issue, he exclaimed that he could not think of a reason why it would be necessary to generate this documentation with Opengades on each build. Instead, it would probably suffice to do it on the nightly builds instead, as it was highly unlikely somebody would require the documentation the same day it had been implemented.

4.4.2 Evaluation survey

While interviewing the teams was a good way to get quick and valuable feedback, we still had a lot of gathered data and graphs of which we had not gotten any feedback on. The Survey we sent out had the goal to fill these gaps. However, getting answers from the developers seemed troublesome. We had initially only sent the Survey out to a more specific range of people, estimated to be around 20, but after more than two week we had only gotten 4 replies. We thought that our explanation of the Survey which was sent in the initial email might be a bit too much to read. Therefore, we rearranged it to be a lot more concise and not get too much into details. Additionally, we thought the developers might be too busy to answer, and they might think this Survey would take a long time to answer. To counter this, we simply included in the explanatory email that the survey would only take about 5 to 10 minutes. Furthermore, this time the survey was sent to the whole office in Lund.

As a result, an additional 4 answers were given. However, these answers ranged a bit in quality, as two of the recipients had misunderstood the context of the survey. They answered in the context of general software development improvement, instead of metrics to improve a CD-workflow. This misunderstanding was caused be the vague description given in the survey this time. While the description in the first send-out had seemed to be too detailed, this description seemed to have too few details making the recipients confused.

While these 8 answers may seem somewhat unsatisfactory, it is important to mention

that there is a big limitation of people who understand the topic of CD well enough to give answers. It is highly possible that a lot of people did not answer because it was not in their skill set, and they felt like they could not give any useful answer. Nevertheless, some interesting opinions were given to us through the survey. Which metrics were useful for each role will be summarised on a per person and role basis. Each unique answer is marked with their role and a counting number if there are more than one. The following results show their answer to which metrics they find most useful for their role, including a motivation (in most cases):

DevOps 1:

- Build on commit time: in order to optimise for build and test time, and to find out if different stages need optimisation.
- Review time: to ensure developers does not get stalled too much.
- Build success ratio per day: may hint that more testing should be done on the client side.

DevOps 2:

- Development lead time (time between the first patch set was uploaded until it was merged): to give an overview; if this time is good enough there is no need for optimisation, otherwise bottleneck should be presented by ore more metrics we presented.
- Deployment time (time between a change was merged until it was released): same as above.

Scrum Master:

- Wants to see trends overall, e.g. if the review time increases day by day, they want to find the root cause of this.

Group Manager:

- Same as the Scrum master, wants to see trends.

Software Architect:

- Build success ratio per day
- Build on commit time (including automated tests, with filtered outliers)
- Review time
- Development lead time
- Deployment time
- Features per week
- Feedback of recent review efficiency

The Software Architect above unfortunately did not motivate their answer further than to say that they found the listed metrics useful.

Team Leader:

- New changes per day (Number of patch set 1 uploaded per day): important.
- Patch sets per day: important, but not as important as new changes per day.
- Failed builds per day: indicates Jenkins trouble.
- Build success ratio per day: same as above.
- Build on commit time (including automated tests): very useful, indicates equipment suitability.
- Review time (time between the first patch set was uploaded until given +2 in Gerrit): useful, indicates team internal communication and spirit.
- Features Per Month (a completed feature = a change that has been merged): important.
- Feedback of recent review efficiency: Yes, useful. An alarm should be set for 24h.

Developer:

- Build on commit time (including automated tests): although arguing that if this time takes more than 30 min they tend to work on other issues instead.

Architect and Team Leader:

- Build success ratio per day: if this is low the developers needs better feedback loop in the local development environment.
- Build on commit time (including automated tests, with filtered outliers): Efficiency of the CI feedback loop.
- Review time (time between the first patch set was uploaded until given +2 in Gerrit): Review efficiency.
- Deployment time (time between a change was merged until it was released): Efficiency of the "customer" feedback loop.
- Average time to restore (time it takes from when a once successful build fails until it is successful again): How easy it is to understand and correct the given CI feedback. (Do you need to read long log files etc.).
- Development lead time (time between the first patch set was uploaded until it was merged): Interesting, but more information would be necessary since there might be many causes of a long development lead time.
- Oldest done feature (the feature which have been done the longest amount of time but not yet included in a release): Same as for Development lead time, as there might be many reasons of why a change has been uploaded.

With this data, it is unfortunately hard to draw any hard conclusions. However, the data is still enough to open up for discussion.

Overall, it seems like Build on commit time is an interesting metric for most roles, being directly present in 5 answers. This is followed by Review time and Build success ratio per day; both being directly mentioned in 4 answers. The reason of the popularity of the Build on commit time, might be that it has an affecting presence in most roles. Developers are directly affected since they must wait for the build to complete to get feedback, and Operators are affected since they may use it to monitor the health and status of the pipeline. Likewise, Review time is also noticeable for the Developers since they are directly involved in the review process - faster feedback is of course desirable for them because they can then get done with their feature quicker and move on. Additionally, this is an interesting metrics for Operators and Architects, since it plays a big part in the cycle time, i.e. the general efficiency of the pipeline. Furthermore, build success ratio per day might also be deemed useful due to the same reason as Build on Commit Time; more failing builds will mean more waiting for the Developers, in addition to a less effective pipeline.

Furthermore, both Deployment time and Development lead time is regarded as important in 3 answers, present in both Architect roles and by one of the DevOps roles. Although there was not much argumentation why the metrics were interesting in most roles, one DevOps role claimed that these metrics can be used to get an overview of the status of the pipeline. This DevOps person gave the motivation that if deployment and development lead times are good enough there is no need to optimise the pipeline.

Additional useful information gathered from the survey might be some responses to the Feedback of review efficiency metric. Specifically, the Team Leader, who suggested that an alarm would go of if something has not been review within 24 hours. This might be a viable policy which can be used to decrease long review times.

Conclusively, these finding can once again be strengthened by the study about which metrics are being used in agile teams in the industry, performed by Kupiainen et al. [15]. Already discussed in Section 4.3.3, the metrics related to Development lead time and Deployment time, being Duration of Build, Sub-build time, Review time, and Nightly build time, could all be related to the Identifying Process Problems category find in the study. Additionally, the Build success ratio metric found useful by some in the survey, can also be placed in this category, since it could be used to identify process problems; if the ratio is low, many builds are failing, therefore there might be a problem with the building process. Furthermore, the Feedback of review efficiency metric can be placed in the Motivating and Improving category, since it might be used to motivate developers to keep the review time low.

4.4.3 Results from interviews and survey

While RQ3 could not be answered explicitly, the interviews and survey gave some insight into it. The interviews suggested that the Build time and Sub-build time metrics were useful to the E2 team, since these metrics actually made the team aware of the degree their limited license for Vectorcast throttled the pipeline, as well as making them aware of how much time the generation of documentation with Opengades accounted for. This might cause them to get more licenses for Vectorcast, and reconsider to move the Opengades job to a later stage in the pipeline (maybe in the nightly builds). To summarise, these metrics supported the developers in optimising their pipeline by opening their eyes to how big of an effect the

limited license of Vectorcast and documentation generation with Opengades had on their Build time.

Additionally, the survey also gave some insight about what metrics are important. Although the answers were limited, they were still valuable since they came from people with knowledge and experience about CD (as an assumption, since many of the roles were connected to CD, and all teams in the case company are working, or at least trying to work with CD). The survey showed that metrics like Build time, Review time and Build success ratio were important in their context. To summarise, these metrics were regarded by a few employees related to CD to optimise for lead time and code quality. Additionally, the survey also yielded that using the Review time metric to give feedback to the developers (the metric called Feedback on review efficiency in the survey), was desirable.

4.5 Summary of results

This section summarises our most important findings, as well as answers to the research questions.

4.5.1 Challenges of CD (RQ1)

To summarise, the following can be considered challenges of CD:

1. Issues related to automation - This can be a lack of automation in any sector like test, build or release, or it could be technical issues when for example automating certain tests. Included here is also the problem of having automated tasks that run too slow, and as such lose their meaning.

2. Issues related to human and organisational factors - This is some part of the organisation not committing to doing CD, this could be developers not following the correct guidelines or management prioritising work on new functionality rather than maintaining a good CD pipeline. It can also lead to other problems like a lack of automation. Included here is also the issue of long review times that was found at Bosch, but is of course a problem that can only come up if the organisation is doing code reviews.

3. Lack of experience - This is related to the second issue, but deserves a mention since it can be crucial. Not having enough experience means that developers might not know why they are doing the things they are doing, and without meaning it is easy to start ignoring important tasks.

4. Technical challenges - This was found in two interviews, as well as in the paper by Hansson [4]. It can be problems with setting up automation in a complicated environment, or hardware limitations.

4.5.2 Best Practices of CD (RQ1)

To summarise, the following can be considered best practices of CD:

1. Committing to doing CD - Having a commitment from everyone involved in the development process is crucial to making sure that practices that are used will be followed, such as not checking in a broken build or not removing out failing tests.

2. Utilising Automation - Automation is key to CD, having manual processes introduce human errors, and take a lot of time. This is something that every source agrees on, as well as everyone that was interviewed.

3. Shared responsibility of the CD-pipeline - Sharing responsibility will make sure that everyone is on broad with CD, and that there is a constant exchange of knowledge between developers. This also includes having social rules to make sure that everyone agrees on what is expected of them.

4. Test driven development - Working with TDD makes sure that new code is always tested, and is a good way to make sure that the necessary automation of testing is done.

4.5.3 Important metrics for improving CD (RQ2)

A list of potentially useful metrics derived from literature is presented in Appendix C.

Out of these metrics, the ones related to lead time were explored in a proof-of-concept, which proved the following metrics in the categories of Development lead time and Deployment time to be important for this study:

- Development lead time
  - Duration of build, including automated tests
  - Sub-build time
  - Review time
- Deployment time
  - Nightly build time
  - Release time

4.5.4 The effect of giving developers metric-based feedback (RQ3)

Unfortunately, the effect of giving developers metric-based real time feedback couldn't be fully explored due to time limitations. However, a few interviews and a survey gave some insight to this which noted the following metrics as important:

- Build on commit time
- Build success ratio per day
- Review time

- Deployment time
- Development lead time
- Feedback of review efficiency

Additionally, by exposing the monitored team to our findings in the proof-of-concept, they became aware of tasks in their build on commit process which took longer to run than they were aware of, and which was also bottle-necking their pipeline. One of these tasks was a bottleneck since it had too few licenses and couldn't be parallelised due to this, and the other task was a documentation job which could be run more seldom.

Chapter 5
Discussion & Related Work

In this chapter we will discuss our method, what worked well and what did not, and why. We will also discuss our results, and compare this study to related work. Then we will discuss the potential for future work that has come out of the research done in this thesis. This is to evaluate the thesis method and result, and point out potential weaknesses in order to learn from the mistakes made. We will also help put the thesis in context to relevant work.

5.1 Evaluation of method

To evaluate how we worked in this thesis, this section will discuss which parts of the thesis work went well, which did not, and why. It will also give some further discussion on why we made the changes to the method that we did, and possible alternate solutions.

At the outset of the thesis the plan was clear, and there where good reasons for each step we were going to take. However, there were some problems early on that could have been mitigated somewhat. Doing interviews with each project for example took more time than anticipated. The reason behind this was mostly because of getting in contact with all the correct people and booking all of them at times that worked for the interviewee was an issue as it was often not seen as a priority from their side. Additionally, most interviews were booked via e-mail, and despite great support from our supervisors the impersonal touch of an e-mail might get the developer to overlook it. There were some people we wanted to interview that took a lot of effort to get a hold of, so sometimes we found out where in the building they worked and introduced ourselves in person. Asking these people in person should perhaps have been done sooner, since it could have sped up the process somewhat. However, it was still considered worthwhile to do the interviews in person instead of using other methods. This was because it gave us a good indication of how willing they were to work with us, as well as some additional results when it came to common problems they were having with CD that might not have been brought up in another type of data gathering. Considering these benefits using interviews for this step was a good choice. It also helped

that the literature studies could be done in the downtime so that the work was still effective.

Another issue was the lack of suitable teams to work with. At the outset we wanted two teams to implement the proof of concept in order to help answer RQ3. Having conducted the quick interviews it was clear that choosing teams to work with would be difficult. A lot of the teams did not have a mature enough pipeline that it would be worth doing our type of study on them, while other were in the middle of intensive periods of work that made them hard to contact and work with. On top of this the fact that the office shut down due to the Covid-19 pandemic, meant that we needed teams that were easy to contact online. This was another problem with the app team since the person we had contact with there was using a different network for his work and as such rarely checked his emails. Had the Covid-19 pandemic not led to this shutdown it is possible that the app team could have been used and more of the original plan could have been executed since we would have had access to the team physically. However, this was not a problem that was possible to see coming, and as such we had to adapt in the moment. This led to making some changes to our process. It was important to try and keep the project on track, and try to answer the research questions as well as we possibly could with the change of method. This led to us deciding to implement the proof of concept in one team while using another for further interviews, as well as doing a survey at the end. In this way we could still gather metrics and ask developers and other employees which metrics they found useful, while also getting some more data to answer RQ1 in a better way. Since the teams were closely related it was also useful to ask for feedback on the metrics from both teams which was useful for RQ2.

However, there is a shortcoming in doing a survey, as it might not be an optimal way of getting information about useful metrics. The reason behind this is that which metrics are important and useful for which role is highly personal and project dependant. Thus, to get more accurate and interesting results, a more discussion-based approach, like doing interviews, would be preferable. In this way, we could adapt the discussion based on the interviewee, as well as ask about potential uncertainties in answers. Additionally, we could go further into detail as we would not have to keep the interview as short as the survey. Keep in mind that the survey had to be kept short and concise in order to get more people interested in answering; if it would take too long, few people might have the time or energy to answer it properly.

Therefore, doing interviews instead would lead to more developed answers. However, this would take a lot of time, since booking interviews with a satisfying amount of people is a much longer administrative process than simply sending out a survey. Additionally, since doing an interview would require more time of the interview than answering a survey, it is possible less people would be eligible to this. Therefore, we concluded that doing a survey would be the best thing regarding our limited time, as well as getting more answers.

However, we had one big shortcoming when sending out the survey. This was that we initially made a too long description of the survey, as well as failing to mention that the survey would only take a short amount of time to answer. Consequently, we sent out a new, shorter description a few weeks later, as well as sending it out to more people. This time we mentioned that the survey would be quick to answer, but unfortunately the description proved to be a bit too vague this time, making a lot of recipients confused about how to answer the survey. In retrospect, we should have spent more time making a carefully crafted description of the survey, both being short and getting the point across without possible misunderstandings.

Another possible shortcoming of this study was that the Review time metrics could be misleading. The measurement of the duration of Review time was from the first patch set was uploaded until it was approved. However, this does not take account for possible reviews not approving the patch set prior to being approved. For example, a developer could upload a patch set, get feedback on it, change the code in accordance to the feedback, upload another patch set, get feedback again, and so on. Finally, when there is no more feedback to give and the reviewer(s) are happy with the change, it gets approved. There could be a lot of variance from change to change of how many of these feedback cycles would have to be done. In some cases, the change can get approved right away, and in some after several patches.

The Review time used in this study includes all these patch sets up until the change was approved. While this still might be an interesting metrics to analyse review efficiency, the name given to it can be somewhat misleading. Review time could be interpreted as the time taken for a developer to do a review, and not the time taken for a change to be approved. Additionally, there might be interesting data found in looking at the time taken for all reviews in a single change, to greater understand if longer time to get a change approved might be because of reviewer's taking a long time before starting to do a review, or if the uploader is writing code which needs a lot of feedback from reviews.

Therefore, it would be interesting to have a metric for the actual review time too, and not only the time taken to get a change approved. However, our limited experience made us miss this issue at the time of implementation.

Overall, while there were some problems encountered during the thesis work they were considered to be handled by the best of our ability. Especially the ones we could not have foreseen such as Covid-19 shutting down the office, together with most of the world. In hindsight there were some problems that could have been handled better with the experience we have now, but on the whole we are happy with how we tackled the method.

5.2 Discussion of results

Some of the results might have needed some stronger data to strengthen the conclusions about them, as well as make the finding more generally applicable. However, this was not always possible due to the limited scope of the study. This section will analyse the method and results in terms of validity, as well as how general they could be considered.

5.2.1 Threats to validity

Due to certain limitations, regarding time, resources, and experience, the validity of the results might not be entirely trustworthy. Challenges threatening this validity will be discussed in this section.

Quick interview scope change

The original plan for the quick interviews was to merely get an overview of all the projects to pick teams that could be used for further studies. However, since there were some issues that were brought up during the interviews related to challenges and best practices for CD (RQ1), it was decided to use these findings as well when helping to answer what challenges

exists when doing CD. Since this was not planed from the start it was not as structured as it could have been. This could mean that there were teams that were having the same problems that we only found out about in one or two teams. However, to make sure that the results were trustworthy we only used the problems that were found in most of the teams, or were backed it up by our literature studies.

Lacklustre data to answer RQ3

Due to the fact that some changes had to be made to the method it gave some problems when answering how the metrics found in RQ2 could assist developers to optimise their CD workflow (RQ3), since there was not enough hard data to support any solid conclusions. There are some things that can be said about it, as described in Chapter 4, but in order to answer if the metrics can actually help optimise for lead time and quality there would be a need to look at these values over a longer period of time as was originally planned. This meant that we just had some indications to help answer RQ3 in the form of answers to the survey, but no hard data as in actual measurements.

The metrics were only gathered from one team

While the literature studies served as a good way to find potentially useful metrics, the gathering of these metrics and analysis of the result served to empirically determine which metrics were useful. However, the scope of this was only a single team (E2), limiting some of the results for being specific for only this team. Although, it helped that we also discussed the result with the E1 team as well. Additionally, many of these metrics could not be gathered due to certain limitations in their pipeline, as well as in our workflow regarding time, resources, and interference with the team.

On top of this, since only 8 people answered the survey, the conclusions drawn based on these answers are not very robust. In many cases there were only one person giving answers per role. Optimally, this survey would be extended to a lot more people to see some trends in the results. However, as it is now it may serve as an indication to which metrics are desirable for which roles.

The E2 team had manual steps in their delivery process

The analysis of the delivery part in the E2 team's pipeline was very limited. There was a manual part in this process; the creation of release notes, and the other parts of the release was done by a manually executed script. This script froze some changes in Git, and copied the ones who had passed the nightly build to their internal customers repository. Considering this it would have been hard to gather data about the releases without interfering with the productivity of the team too much. Therefore, we simply asked the employee responsible for the releases to estimate how long this process took and settled with that.

Ultimately, this made us focus a bit more on the CI part of the pipeline, and a lot of potential findings which could have been explored was it that more data would be available from the delivery part of their pipeline, would have to be omitted.

Limited deeper analysis of the Build time

Since only two of the sub-builds in the build on commit time were analysed, it is possible there were more causes to the high build time. Although the result of these two sub-builds was promising, this might merely be a smaller portion of a bigger part. If more sub-builds were analysed, there is a possibility that a few more problems with the build time would have been highlighted. In other words, it is possible we only found a smaller part of the bigger problem.

5.2.2 Generalisation of results

The study was limited to only one company, and we only worked closely with two teams at this company. This might have caused some specific findings which may only be relevant to the case company or the targeted teams. On the contrary, there might be some similarities with the target company and teams, which could motivate for some results being relevant in a more general context. Which of these results that can be used for a more general case will be discussed in this section.

General challenges and best practices

There were a lot of challenges and best practices found to answer RQ1 that could be considered general since they were found in most literature studies and interviews at the case company. The importance of automation is one of these that is brought up repeatedly as a best practice, and that a lack of automation leads to a lot of problems when trying to do CD. Human and organisational issues are also mentioned a lot and that it can be solved by committing fully to CD, by training developers, setting up rules that can be agreed on, having a shared responsibility for the CD-pipeline, and keeping CD as a priority.

Measuring idle time could be very important

One of the bottlenecks found in the E2 team's pipeline was that jobs had to wait in queue in order to get access to the unit test tool Vectorcast, since the team had a limited amount of licenses to it. Meanwhile, once a job got access to Vectorcast, the unit tests did not take long to run. If only the actual time spent doing the job, and not the idle time spent waiting for the resource would be measured, this bottleneck would not have been found. Therefore, measuring the idle time proved to have a big positive effect on finding bottlenecks in a CD-pipeline. This conclusion could be generalised on all kinds of measurements since idle time could be found almost anywhere in a similar way to this example.

Licenses of certain tools might cause bottlenecks

One of the conclusions draw from analysing the pipeline of the E2 team was that the limited license for the unit test tool Vectorcast throttled the build time. While teams in the industry using Vectorcast may be limited, this finding should not be limited to just this. Instead, the exact software used can be omitted, and the conclusion generalised to licensing limitations in any software used in a pipeline. Vectorcast's requirement was that only one instance of the software per license can be run at the same time. However, this could be generalised to

any software with the same requirement. Licensed software should be handled with care in a CD-pipeline, as they might very well be a bottleneck if too few licenses are bought.

5.2.3 Overall judgement of the results

The literature study presented a list of potentially useful metrics, which should be an interesting look for anyone trying to measure the performance of a CD-pipeline, as well as a good starting point for future researchers. Furthermore, some of these metrics proved to be useful monitoring the E2 team's pipeline by not only giving an overview of the general status of the pipeline, but also finding two major bottlenecks.

With these findings we are satisfied with the results, and are happy we could help the target team to improve their pipeline. The generalisation factor of the results discussed above also indicates that more teams in the industry, as well as curious researchers, might regard our findings as helpful. The only part of the result that is considered a flaw is the survey, and the lack of hard data to answer RQ3.

5.3 Related work

During our literature study there were many papers that were relevant to our thesis. Here we have picked four pieces of related work determined to be relevant for this thesis that will be discussed in relation to our thesis. They gave an overview of the current landscape of research into CD and metrics, and were used as a part of the literature study as well.

5.3.1 Defining metrics relevant to CD

Lehtonen et al. defined metrics which could be useful for CD in their paper "Defining Metrics for Continuous Delivery and Deployment Pipeline" [8]. A summary to this paper followed by a discussion, as well as this paper's relation to this thesis follows.

Summary

The motivation of the research was to explore the importance of measuring the performance of the CD-pipeline, and what metrics gathered from the already automatically created data from the tool chain can be useful for this. Despite this topic being researched before (in 2010), the implementation of the Lean principles has already undergone major changes (as of 2015), which motivates further research. Additionally, the authors motivate their study further by arguing that Lean creates an environment where every step in the pipeline can easily be traced, and therefore measured in a way which was not possible with more traditional methods. They also argue that since a CD-pipeline is continuously improving, valuable metrics on it will be the first step of improving it further, as the first step of improvement is identifying the fault. Metrics allow for this. They also back up these claims with various related work. Thus, they justify performing a contemporary analysis of what should be tracked in a CD-pipeline. The research questions are the following:

- RQ1: Which relevant data for practical metrics are automatically created when using a state-of-the-art deployment pipeline?

- RQ2: How should the pipeline or associated process be modified to support the metrics that escape the data that is presently available?
- RQ3: What kind of new metrics based on automatically generated data could produce valuable information to the development team?

The study was performed on a project at a mid-size Finish software company. The project in question was a website responsible for handling municipal authorisations and permissions. As a first step, the authors analysed research about metrics in Lean methods in manufacturing. The motivation being that lean methods originates from just manufacturing. With this information they converted these metrics to be applied in a CD-context. Additionally, they also conducted the previous research from 2010. As a result, they proposed metrics divided into two categories: *Metrics on the Implementation Level*, and *Metrics on the Pipeline Level*. The former are metrics dependant on how the pipeline was implemented; what tool-set and practices were used. The latter are metrics independent of the implementation of the pipeline.

The metrics on the implementation level were the following:

- Development time: the time taken to develop a new feature.
- Deployment time: the time it takes to deploy a new feature once its implementation is complete.
- Activation time: the time it takes for a user to activate the new feature after deployment.
- Oldest done feature: the time a feature has been developed, but not yet deployed to a production environment.

The metrics on the pipeline level were the following:

- Features Per Month: the number of new features released per month,
- Releases Per Month: the number of releases per month.
- Fastest Possible Feature Lead Time: the amount of time a feature spends in the build and test phase.

Particularly, the oldest done feature metric was deemed as the most useful metric by the team.

Following are the authors' conclusion to the research questions:

- RQ1: Both development and deployment time data can be gathered from the toolchain. However, it is dependent on what tools are used. As for activation time it is a lot less clear, since there are many different features and what can be regarded as activating them might depend heavily on the context. For visual elements maybe the first time a user looks at the app, or for a feature that can be activated, the first time of activation.
- RQ2: While metrics for the development processes are automatically stored in the version control system, metrics for the end user are not. It was concluded that more work on an improved toolchain must be done to regard for this. However, they also argue that this might not even be in the scope of Lean process management.

- RQ3: Metrics based on similar research (from manufacturing) was proposed, as well as metrics based on old research about the same subject.

The authors argue that the results should be mostly generalisable. The metrics on the pipeline level could be applied to other CD-projects, since they are not dependant on what tools are used. The metrics on the development level, however, are tool specific, but it is highly possible most modern tools would support at least most of these metrics, otherwise this might be solved with workarounds or add-ons.

Discussion

As motivated by the authors, the significance of the research problem was that the practice for implementing a CD-pipeline is constantly changing and being modernised, and since the last research on this subject was performed back in 2010, where the concept of CD still was pretty young in the industry, further research in 2015 was deemed necessary. These claims were well argued for and had reasonable data backing them up, making the reason of the study apparent. The paper should be a valuable read for anyone related to lean principles, and particularly interesting for anyone doing research about metrics in lean practices. Furthermore, not only is the importance of metrics motivated well by the authors, but the result of the research is also valuable for anyone trying to improve their CD-pipeline. Therefore, it is highly relevant to our thesis study.

In addition of taking claims from related research an applying them to lean principles in CD, they also evaluated new and unique information in the shape of new metrics. However, what is somewhat lacking is data backed up proof that these metrics work. For the metrics taken from manufacturing, their claim is that because they work in lean methods in manufacturing, they should also work in lean methods in computer science. Although the metrics are generalisable enough for some readers to obviously be transferable, there might still be subtle differences in a CD-pipeline compared to a lean manufacturing one. Furthermore, the authors also claim:

> Based on discussions with the team [...], visualising the data regarding features on the pipeline was found very useful, **and exposing developers to it actually led to faster deployment and to less uncompleted work in the pipeline**.

Empirical data of showing how much this sped up the deployment time and how much less uncompleted work it caused, would be very interesting.

Additionally, the research was only performed at one single team working with one project, vastly limiting the generalisation factor of the results. However, we counter this in some regard in our thesis by using some of the metrics on additionally one team and getting feedback on it.

Relation to the thesis

This study was found very useful when looking for interesting metrics to monitor. Several of the metrics resulting from the paper were implemented while monitoring the E2 team. These metrics were Development time, Deployment time, Feature per month, and Oldest done feature. Both Development time and Deployment time helped with locating the bottlenecks of the pipeline and proved valuable in our study, as well as being prominent in the survey's

answers. Features per month also showed some popularity in the survey. As for Oldest done feature, one of the roles in the survey argued that they would find it useful, as long as it could be split up in smaller parts in order to locate more specific error, as there may be many reasons as of why a feature has not been merged.

As for the other metrics found in the paper, as well as the ones we implemented, they all were marked as potentially useful by our literature study.

5.3.2 How and why metrics are used in the agile industry

Kupiainen et al. explored metrics used in agile teams in the software industry in their paper "Why Are Industrial Agile Teams Using Metrics and How Do They Use Them?" [15]. A summary to this paper followed by a discussion, as well as this paper's relation to this thesis follows.

Summary

The authors list several examples of literature reviews which have been published studying software metrics. However, they argue that all these examples have been written from an academic viewpoint, and to their knowledge there has been no systematic literature review on metrics used in the actual software industry. Furthermore, they claim that agile software development is increasing in popularity and that the use of metrics in more traditional software development methods cannot be directly transferred to agile methods. This motivates them to perform a study on agile software development metrics to find out which metrics are used, in addition to why they are used and what actions the use of these metrics can trigger.

The study was performed by doing a systematic literature review (SLR) on relevant literature on the topic of agile software metrics. As a first step, the authors familiarise themselves with the concept of SLR by reading literature on the topic and using a guide on how to make an SLR as a basis. Additionally, they iterated the protocol in weekly meetings, as well as making a pilot study on the topic. Once familiar with the concept, the SLR was performed in the following steps:

1. **Search and Selection Process**: papers were selected in accordance with the main selection criteria, being: "papers that present empirical findings on the industrial use and experiences of metrics in agile context", in addition to other criteria such as quality of results and agile and industrial context. With these criteria in mind the selection process was conducted in the following stages:

   (a) All literature relevant to the topic was found using automated search methods. Specifically, certain search strings with relevant keywords and synonyms to agile software development and metrics were used to find the first batch of literature. In this way 774 papers were found.

   (b) Papers were filtered based on their title and abstract by one of the authors, leaving 163 papers. The validity of this selection was then analysed by letting another author filter a random selection of 26 papers, which resulted in the level of agreement being substantial.

(c) Papers were filtered based on their full text, leaving 29 papers. Again, the selection process was quality assessed by another author, reading 7 randomly selected papers.

2. **Data extraction**: Integrated coding was used as a data extraction strategy. This made it possible to make a sample list of the following code categories: "Why is the metric used?", "How is the metric used?" and "Metrics". The code was built by the first author marking interesting quotes in the full texts, and then letting a second author review the code regarding them. Throughout weekly meetings, the authors built the following rule set for data extraction:

    - Collect a metric only if the team or company uses it.
    - Do not collect metrics that are only used for the comparison and selection of development methods.
    - Do not collect metrics that are primarily used to compare teams.
    - Collect metric only if something is said about why it is used or what actions it causes.

3. **Data synthesis**: in order to make higher level categories for the codes, each code was first described on a higher level, then grouped by similar codes, and lastly these groups were given a high level code in the form of a category.

These steps resulted in the following representation of categories for metrics used in an industrial agile software context:

- Iteration Planning: metrics which helps to choose which tasks to prioritise for the next iteration.
- Iteration Tracking: metrics used for tracking the status of the selected tasks during an iteration.
- Motivating and Improving: metrics used for motivating people and improving practices and performance on a team level.
- Identifying Process Problems: metrics used to spot problems in processes and workflows.
- Pre-release Quality: metrics used to assure and evaluate the quality of the software before it has been released.
- Post-release Quality: metrics used to assure and evaluate the quality of the software after it has been released.
- Changes in processes or Tools: how applying metrics led to changes in the processes and tools used.

For each of these topics, the authors explained specific cases extracted from the papers and how they correlated to the category. In addition, the authors portrayed the validity of the data by showing the distribution of agile methods, the distribution of domains (fields of

work), and a distribution of the publishers of the primary studies. Furthermore, the authors claim that this paper will help give future researchers and practitioners an overview of the metrics used in agile software development, as well as documented reasoning behind the use of them.

Discussion

The authors argue that a solely literature-based study is satisfactory for their desired result since there's already existing literature that could be synthesised. However, this might not be entirely true, as it comes with two major drawbacks which are showcased in the study. The first one being that the researchers are not able to control the scope of the study, i.e. they have limited control of which type of companies and which type of agile methods they want to include in it. This is specifically proven by the distribution of domains presented in the result section, showing a clear bias towards the telecom industry. The second drawback is that, in addition to making the research biased to a certain domain, it could also make it biased to a certain company. In this case, the study was biased towards Ericsson, since they had published many papers on the topic. While both drawbacks mentioned are explained in the limitations section, the authors do not provide a solution to eliminate them, or what they could have done differently to eliminate them. This bias is somewhat of a threat to our thesis, since our case company was not in the telecom industry. However, it is still only a bias, and there might very possibly be some overlap where this bias is not important, as all software development in the study was based on agile methods.

When presenting the results, the authors truncate some data by categorising it by "other". This occurs both in the distribution of domains, and the distribution of methods. In both distributions they only showcase the three most popular domains/methods and label all the other domains/methods as "other". This causes a loss of valuable information which could be useful for future researchers. Specifically, for our case we wanted to know how many of these agile methods were using CD, but since it was a minority it had been truncated to "other". This makes us unaware of if no study using CD was accounted for in the result, or if a few or one was accounted for.

Relation to the thesis

Despite not knowing if any on the agile methods reviewed in the study were using CD or not, as well as the bias towards the telecom industry, the generalisation factor of agile methods should be great enough to at least consider these findings useful for CD; agile methods in the telecom industry might not differ that much to agile methods elsewhere, and findings from agile methods in general might nullify their appliance to CD.

Additionally, in our study we regarded some metrics as useful or potentially useful by our findings from implementing them on the E2 team's pipeline, as well as from our findings in the survey. However, as the validity of these metrics were not as strong as we desire, they were strengthened by correlating them to the categories found in this study, effectively binding our results to a more industrial and practical context.

5.3.3 Best practices, benefits and challenges of continuous delivery

Hansson studied challenges causes and solutions in his paper "Best Practices, Benefits and Obstacles When Conducting Continuous Delivery in Software-Intensive Projects" [4]. A summary to this paper followed by a discussion of this paper's relation to this thesis follows.

Summary

This is a thesis from Malmö University that addresses the "stairway to heaven" software project model, and the obstacles to reach the continuous delivery (CD) step, which is the fourth step. This model consists of five steps starting at traditional development and ascending to R&D as an experiment system with different levels of continuous activities in between. The thesis also explores benefits and best practices of CD. This is to build on earlier research into these topics which left some questions about further obstacles to reach the CD step in the stairway to heaven, and beyond. The goal is to summarise and extend previous research into the topic of benefits, obstacles, and best practices of CD. This is done by connecting the literature on CD with industry knowledge. To do this a study combining literature reviews on the topic, as well as multiple case studies of four different companies were conducted. The intent is to put the problems of the companies into the theoretical framework from the literature that was studied before.

Literature was found using google scholar and Malmö University Libsearch with several keywords such as "continuous delivery". The case studies were done in the form of semi-structured interviews that included a list of questions with room for discussion with representative developers from the four companies. These interviews were done either in person or over the phone depending on what was most convenient, and if the authors could get permission they were also recorded. On top of this some field work was conducted at one company to better understand their workflow and try to improve it. From the literature studies they concluded that there are many benefits to CD, and that these are fairly well documented already so there are no obvious gaps in this research. The largest and most general obstacles were lack of automation and complex development environments, but there were others and it depended on the company and the type of work they were doing. To fix these the authors gave some examples of best practices which differs from organisation to organisation, depending on their needs. Test driven development (and extreme programming practices in general) as well as finding easier tools for the CD pipeline were two that were important. The problems of the different companies were addressed separately, but parallels were drawn in an effort to see which problems were general and had generalisable solutions and which required more specific solutions such as the issues of custom hardware and UI testing.

Discussion

This thesis was very relevant to ours, since it brought up challenges and best practices of CD. Overall, there were some similar results between this thesis and ours, and it was used as a source during the literature study. In terms of challenges many of the same ones were found, such as a lack of automation and problems related to the human and organisational elements. It also discussed complex environments which were also a problem found in our literature

study, and in some teams at Bosch. It does not bring up long review times as an issue, which is a problem that we found at Bosch. Therefore, it is possible that this is not a general problem.

There were also many of the same best practices discussed. Using automation is a big one that comes up a lot in our and their thesis, as well as shared responsibility for the CD-pipeline. Their thesis does bring up some other topics which were not mentioned much in ours, such as using version control, but this is considered to be such a basic step of CD that it was not included as a best practice since it should be assumed. The same can be said for releasing frequently, as it is brought up in the background of this thesis.

5.3.4 Adopting continuous delivery: problems, causes and solutions

Laukkanen et al. studied challenges causes and solutions in their paper "Problems, causes and solutions when adopting continuous delivery — A systematic literature review" [2]. A summary to this paper followed by a discussion of this paper's relation to this thesis follows.

Summary

This paper addresses the lack of adoption in the industry when it comes to Continuous Development (CD). Although instructions on how to adopt CD have existed for a long time, it is not yet widely adopted and those who have adopted it have found numerous challenges according to the authors. To ease the adoption process, this paper tries to find the problems, causes, and solutions of adopting CD. The hope is to gather data that can be used to help the adoption of CD, showing on which problems can come up, as well as their causes and solutions. This is specifically on the topic of adopting CD and does not deal with what problems can arise once a team has adopted it and are working with CD. It can also help the reader understand the benefits and limitations of CD to evaluate if it is a worthwhile practice for them. It also fills a niche since there was some previous research on the problems of CD, but not on adoption problems. To get this data a systematic literature review was conducted. It included only empirical studies from major bibliographic databases. Since there was not a lot of literature on the topic of CD specifically in the context that the authors were looking for they did not use a strict definition of CD, and as such included articles mentioning continuous integration (CI) and continuous deployment as well. They used several filtering criteria to find the articles that were related to the subject, which at the end of this process was a total of 30 articles. These were then read in full and problems, causes, and solutions were extracted from the text. In total there were 40 problems, 28 causes, and 29 solutions. Problems and solutions were then categorised into themes (7 problem themes and 6 solution themes). Of these problem themes the most frequent were build design, system design and testing. In studying these problem themes, it was also shown how they were interconnected, and that the solutions to some problems were a step towards solving others, where for example system design solutions could help solve testing problems as well.

Discussion

Some of the problems and solutions in this article can be considered challenges or best practices of CD, while others are less clear, or are part of what we consider to be the same problem.

This paper also focused on adoption problems of CD, which means that some of the problems and solutions brought up was not relevant to our work, such as system design problems like unsuitable architecture. There were many problems that were relevant since they had been found in our other studies, or were related to CD both at adoption time and when maintaining a CD-pipeline. These included build and test automation issues, hardware and UI testing problems, and lack of experience or motivation in the organisation for doing CD. It also brought up some best practices that helped strengthen our results. This included test parallelisation to cut down on test times, training, and social rules on CD for developers to help mitigate developer-related challenges, and monitoring build length. There were some things that were not brought up explicitly such as automation of tests, Test driven development, and having a shared responsibility of the CD-pipeline. However, automation is touched on when testing problems but is not brought forward as heavily as we have in this thesis.

5.4 Future work

As the subject of CD is relatively new in the software development market, there is to our knowledge still a lot of research which could be done. Interesting topics we would have liked to explore but was out of scope of our time limited and resource limited research will be discussed following, as well as other interesting topics concluded from our results.

First there is the gap left in this research. As discussed in section 5.2.1 there was a lack of hard data to answer RQ3. This could be a topic for further studies, as there is an opportunity for researchers to pick up where we left of by observing what effects actively presenting the implemented metrics could have on the E2 team, maybe in the form of a dashboard displayed on a big screen in the office. In this way, RQ3 would be explored further, and thus be given a more solid answer. Furthermore, since a number of the metrics seemed promising from our results, a topic for a future study could be to implement these and visualise them for a team so that they can easily keep track of them in a similar way to our proof of concept, an then looking at different performance metrics over a period of time to see what happens to these.

It could also be interesting to look at some of the metrics we did not have the resources to look at. This could be by implementing them in a team, or find a team that already have them available through some tool, and then do a similar study to see if these metrics can help improve the performance based on some criteria. This could be trying to see if the metrics improve over time, such as cyclomatic complexity decreasing, test coverage increasing, or review times decreasing. Of course, this should be with the added perspective of looking at why these metrics change. Test coverage increase does not mean that the tests are quality test, and review times decreasing should cut down on the time before someone looks at the review, not on the time spent reviewing which could mean that the reviewer misses something.

There could also be an opportunity for further understanding by interviewing teams at other companies that are working with CD. This could strengthen or contradict the results of this thesis in terms of how general challenges and best practices are, as well as what metrics are considered important in different fields of software engineering. It could also help answer if the duration of each review could be an important metric to look at.

Chapter 6
Conclusion

Creating and maintaining an efficient CD pipeline can be a very difficult task, but if done correctly it can also be very rewarding. We set out to explore how this can be achieved by finding challenges and best practices of CD, as well as with a proof-of-concept demonstrating how metrics can be gathered and analysed from an existing pipeline in order to make it more efficient.

The challenges and best practices were found through doing literature studies and interview with experienced practitioners and developers. Some common challenges were lack of automation and human and organisational factors. Some best practices were automating as much as possible, and sharing responsibility of the CD-pipeline.

Additionally, through a literature study, a list of potentially useful metrics was created. Among these metrics, a few were implemented on a targeted team's project. This resulted in us finding two major bottlenecks in their pipeline: one dependant on limited licenses to a unit testing tool running on each uploaded change, and one dependant on generating documentation too often.

Furthermore, this also led to an unexpectedly useful finding: while monitoring a pipeline with metrics, it is important to include idle times between processes. In our case, the bottleneck regarding licenses would never have been found if this idle time wasn't measured.

Finally, we wanted to observe the effects of giving metric-based feedback to the team. However, our time limitation did not make this possible. Interviews with practitioners as well a survey gave some insight to this, but not enough to draw any strong conclusions. Therefore, we determined that this would be an interesting topic to study further.

Overall, we are satisfied with our findings. The case company can use our proof-of-concept to monitor more teams in the future in order to effectivise their pipeline. Furthermore, practitioners of CD and future researchers can use our challenges and best practices while looking to improve a CD-pipeline.

References

[1] J. Farley, D. Humble, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley, 2010.

[2] J. I. E. Laukkanen and C. Lassenius, "Problems, causes and solutions when adopting continuous delivery—a systematic literature review," in *Information and Software Technology*, vol. 82, pp. 55–79, 2017.

[3] S. Bellomo, N. Ernst, R. Nord, and R. Kazman, "Toward design decisions to enable deployability: Empirical study of three projects reaching for the continuous delivery holy grail," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pp. 702–707, 2014.

[4] B. Hansson, "Best practices, benefits and obstacles when conducting continuous delivery in software-intensive projects," 2017. Master thesis, Malmö University.

[5] W. Babich, *Software Configuration Management - Coordination for Team Productivity*. Addison-Wesley, 1986.

[6] M. Fowler, "Continuous integration." `https://martinfowler.com/articles/continuousIntegration.html`, 5 2006. Accessed: 2020-06-15.

[7] C. Beck, K. Andres, *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2004.

[8] T. Lehtonen, S. Suonsyrjä, T. Kilamo, and T. Mikkonen, "Defining metrics for continuous delivery and deployment pipeline," in *Proceedings of the 14th Symposium on Programming Languages and Software Tools*, CEUR Workshop Proceedings, pp. 16–30, 2015.

[9] M. Fowler, "Continuous delivery." `https://martinfowler.com/bliki/ContinuousDelivery.html`, 5 2013. Accessed: 2020-06-15.

[10] L. Chen, "Continuous delivery: Huge benefits, but challenges too," *IEEE Software*, vol. 32, 03 2015.

[11] Jez Humble & David Farley, "Continuous delivery vs continuous deployment." https://continuousdelivery.com/2010/08/continuous-delivery-vs-continuous-deployment, 2010. Accessed: 2020-06-15.

[12] M. Fowler, "What is scrum." https://www.scrum.org/resources/what-is-scrum, 5 2020. Accessed: 2020-06-15.

[13] S. Neely and S. Stolt, "Continuous delivery? easy! just change everything (well, maybe it is not that easy)," in *2013 Agile Conference*, pp. 121–128, 2013.

[14] M. Leppänen, S. Mäkinen, M. Pagels, V. Eloranta, J. Itkonen, M. V. Mäntylä, and T. Männistö, "The highways and country roads to continuous deployment," *IEEE Software*, vol. 32, no. 2, pp. 64–72, 2015.

[15] E. Kupiainen, M. V. Mäntylä, and J. Itkonen, "Why are industrial agile teams using metrics and how do they use them?," in *Proceedings of the 5th International Workshop on Emerging Trends in Software Metrics*, WETSoM 2014, (New York, NY, USA), p. 23–29, Association for Computing Machinery, 2014.

Appendices

Appendix A
First Interview Questions

1. What's the name of your team? How many people work in it? What do you do?
2. Do you have any documents or a wiki about your development process?
3. Do you have any experience with CD?
4. Are you following CD as far as you know?
5. What tools are you using for code management / CD (Gerrit, jenkins)
6. How do you work with code reviews? Policy?
7. How long does it normally take before getting your code reviewed?
8. Do you have automated build scripts?
9. Do you use commit hooks?
10. How often do you build?
11. How often do you do a release?
12. How do you test your code? Automated tests, manual tests, GUI tests, smoke tests?
13. Do you have metrics for test coverage, code analysis, build duration or number of failures etc.? (Sonar cube)
14. How often do you release and what's your release process? Automated delivery?
15. Where do you deliver code? Customer, company, internally?
16. Are there any steps in the code management process you feel takes too much time or could be improved?

17. What are the parts that work best in your CD pipeline according to you?
18. Anything else you would like to add?

Appendix B
Further interview with the E2 team

1. Can you give us an overview of your pipeline from commit to release-build. Processes, tools, etc.
2. How do you handle built binaries?
3. Are there any rules / guidelines for your processes such as code reviews, commit messages etc.? If not, is this something you want?
4. What kind of feedback do you currently receive from your pipeline?
5. How do you get access to this feedback (different tools, spread out/assembled)?
6. How do you use the feedback you receive now?
7. Is all the feedback you receive now useful? Which metrics do you use most often? Why?
8. Is there any metric you feel you are missing at the moment? Why?
9. What do your tests look like, which ones do you perform as a team and which are passed on to others?
10. Which tests have you automated and which are manual? Can the manual be automated? Why / why not?
11. How confident do you feel about the code quality when your automated tests have run?
12. Do you run smoke tests? When in that case? Why / Why not?
13. Do you run integration tests? When in that case?
14. Who is responsible for your CD-pipeline? (team / person / DevOps teams / different for different parts)

15. How much experience does the team as a whole have of CD?
16. Do you build the system the same way it is tested as it does for release? If not, why not?
17. Are there any manual steps in your release process? Can they be automated?
18. How long does a release take?
19. Do you have the opportunity to roll back a release?
20. Which parts work best / worst in your pipeline?

Appendix C
Metrics from literature

The metrics recommended by the book Continuous Delivery:

- Cycle time.
- Test coverage.
- Amount of duplicated code.
- Cyclomatic complexity.
- Afferent and efferent coupling.
- Number of warnings.
- Number of defects.
- Number of commits/builds/build failures per day.
- Duration of build, including automated tests.

The metrics recommended by experienced Operations developers at Bosch:

- Intermittent errors (number of failed builds/number of builds).
- Jenkins job status.
- Review times
- Which reviews have waited the longest.
- Change failure rate.
- Mean time to restore.

- Delivery lead time.
- Delivery frequency.
- Mean time between failures.
- Mean time to recovery.

Metrics from the research of Lethonen et al.:

- Development time: the time taken to develop a new feature.
- Deployment time: the time it takes to deploy a new feature once it's implementation is complete.
- Activation time: the time it takes for a user to activate the new feature after deployment.
- Oldest done feature: the time a feature has been developed, but not yet deployed to a production environment.
- Features Per Month: the number of new features released per month.
- Releases Per Month: the number of releases per month.
- Fastest Possible Feature Lead Time: the amount of time a feature spends in the build and test phase.

Categories of metrics defined by a systematic literature review performed by Kupiainen et al.[15]:

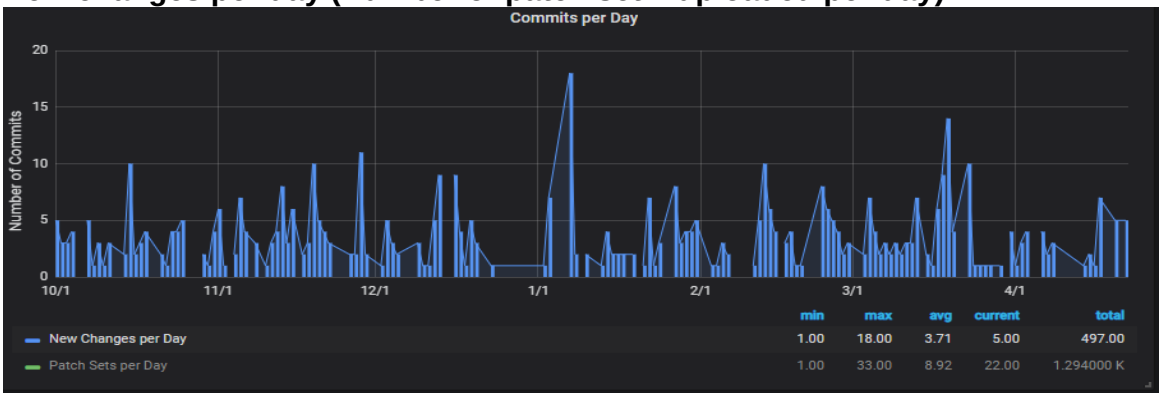
- Iteration Planning: metrics which helps to choose which tasks to prioritise for the next iteration.
- Iteration Tracking: metrics used for tracking the status of the selected tasks during an iteration.
- Motivating and Improving: metrics used for motivating people and improving practices and performance on a team level.
- Identifying Process Problems: metrics used to spot problems in processes and workflows.
- Pre-release Quality: metrics used to assure and evaluate the quality of the software before it has been released.
- Post-release Quality: metrics used to assure and evaluate the quality of the software after it has been released.
- Changes in processes or Tools: how applying metrics led to changes in the processes and tools used.

Appendix D

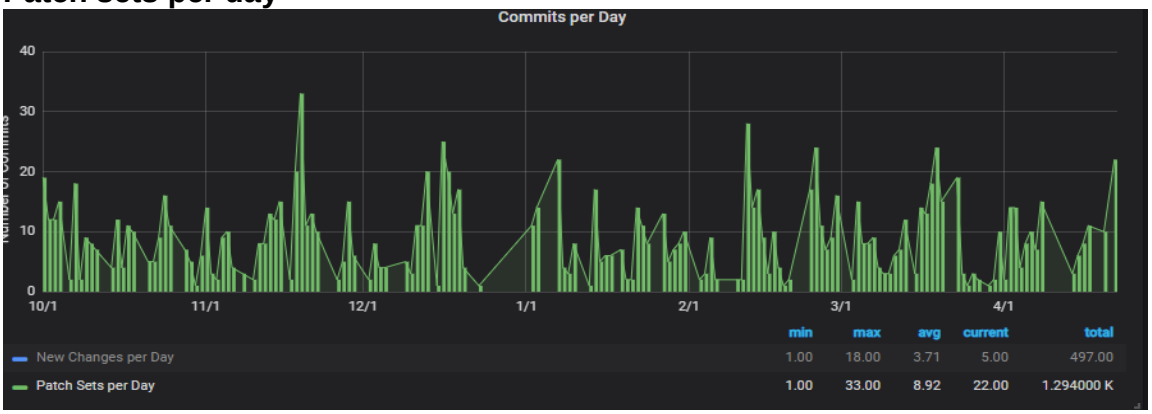
Informational document included in the Survey

Timespan: 2019-10-01 – 2020-04-22
Please also have a look at the aggregated data in the tables of each graph, for example “*avg*”, “*current*” etc.

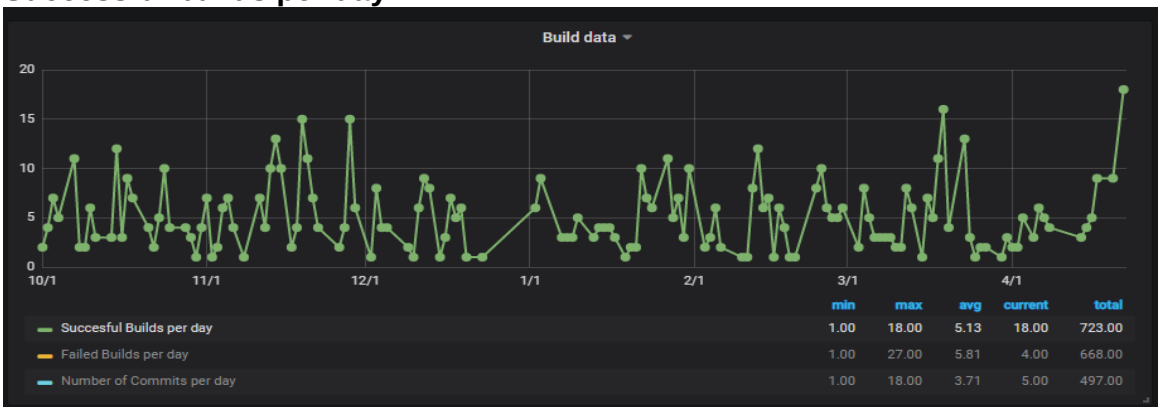
New changes per day (Number of patch set 1 uploaded per day)



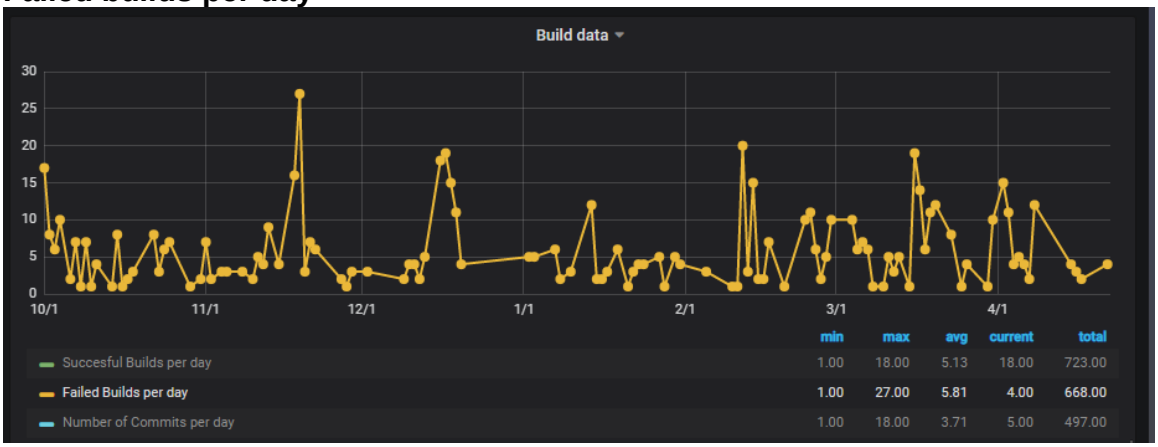
Patch sets per day



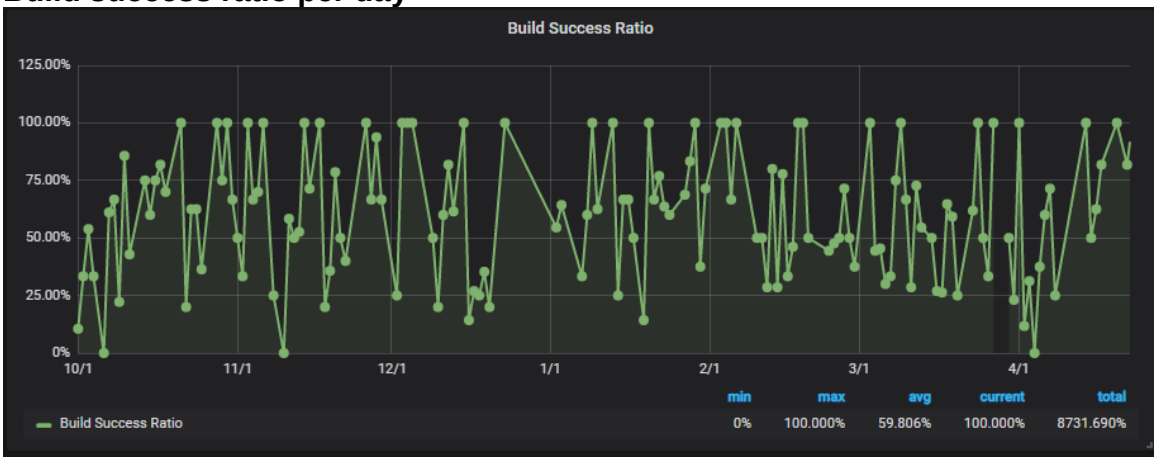
Successful builds per day



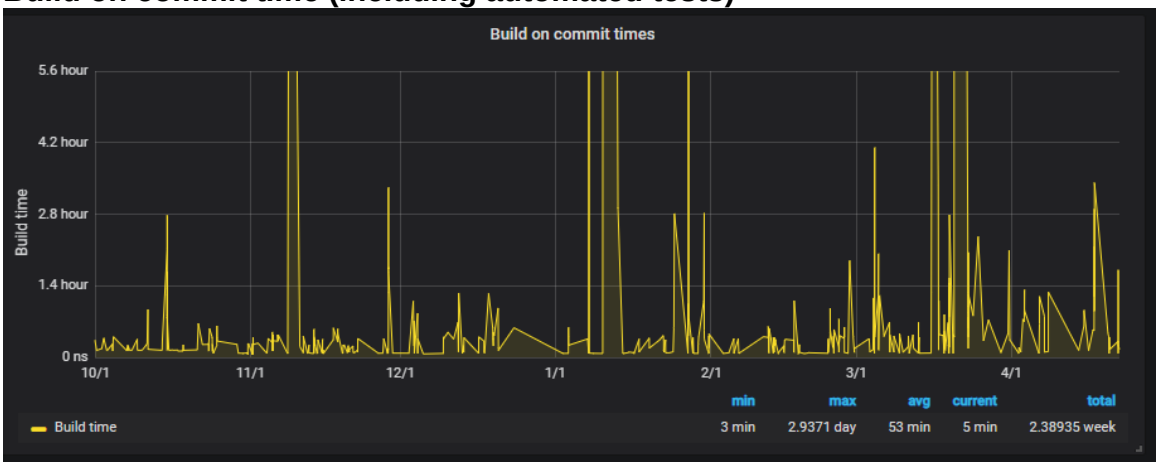
Failed builds per day



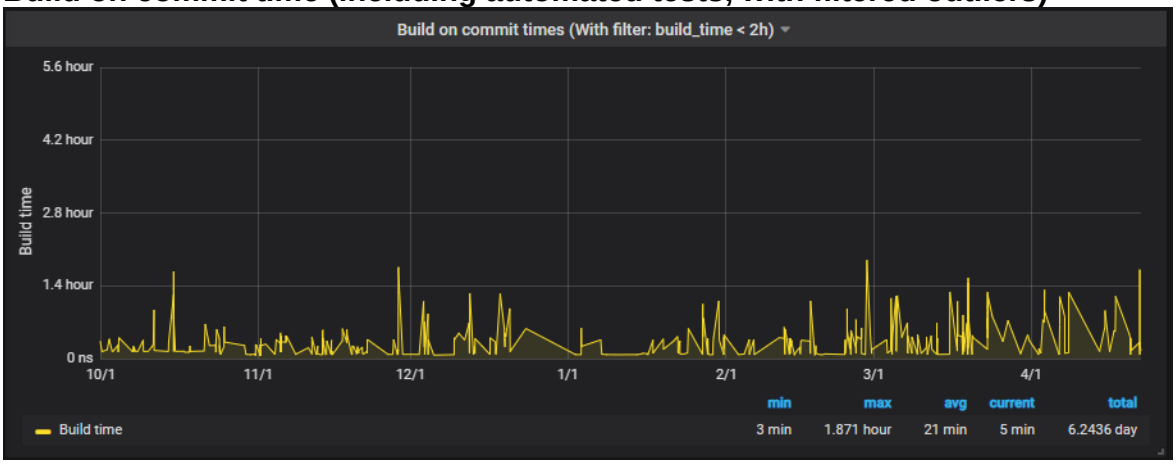
Build success ratio per day



Build on commit time (including automated tests)

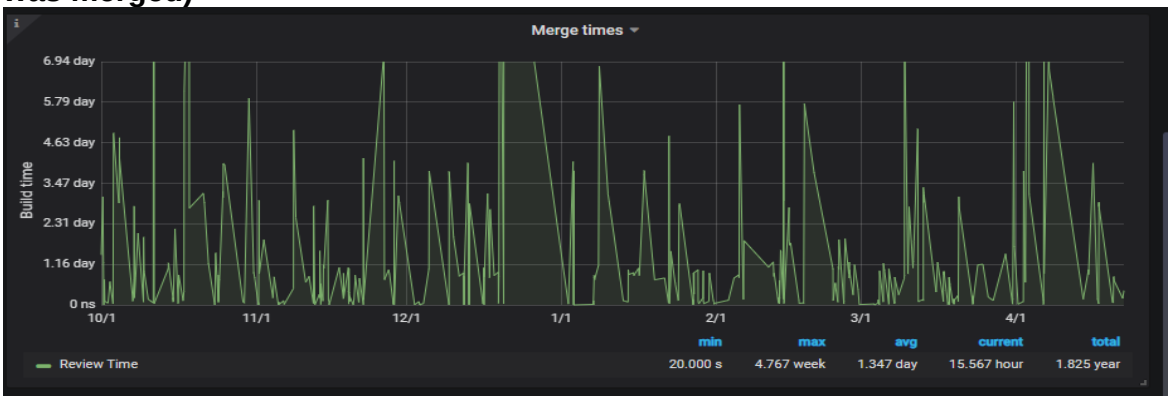


Build on commit time (including automated tests, with filtered outliers)

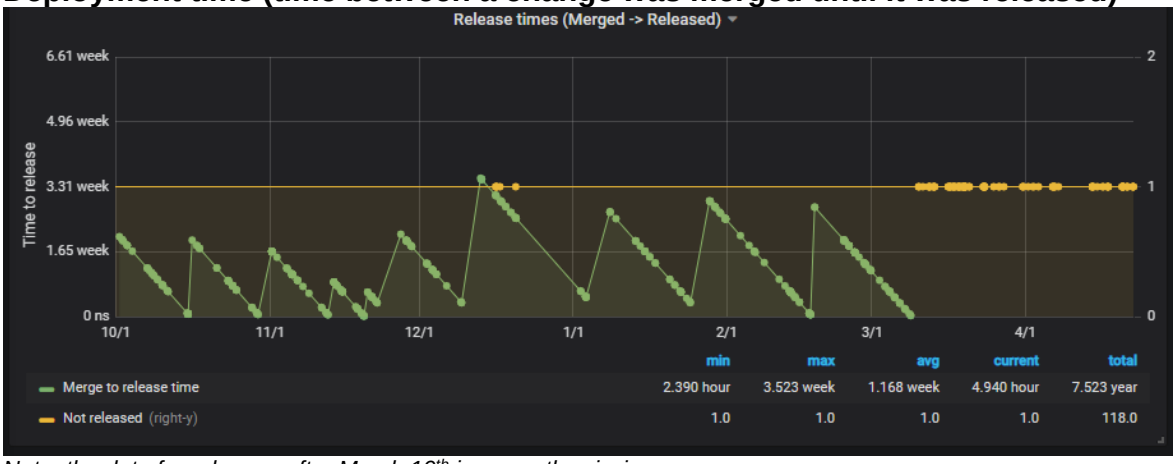


Review time (time between the first patch set was uploaded until given +2 in Gerrit)

Development lead time (time between the first patch set was uploaded until it was merged)

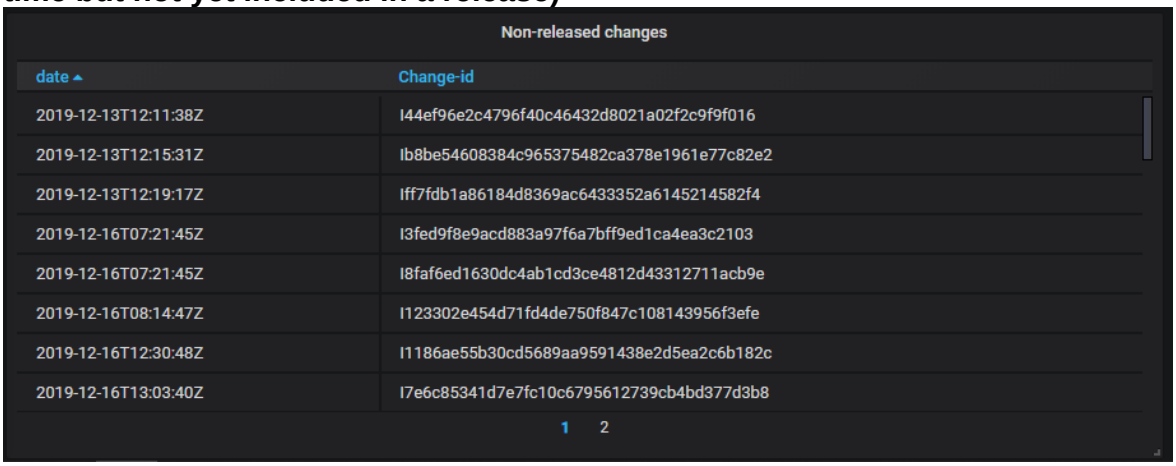


Deployment time (time between a change was merged until it was released)



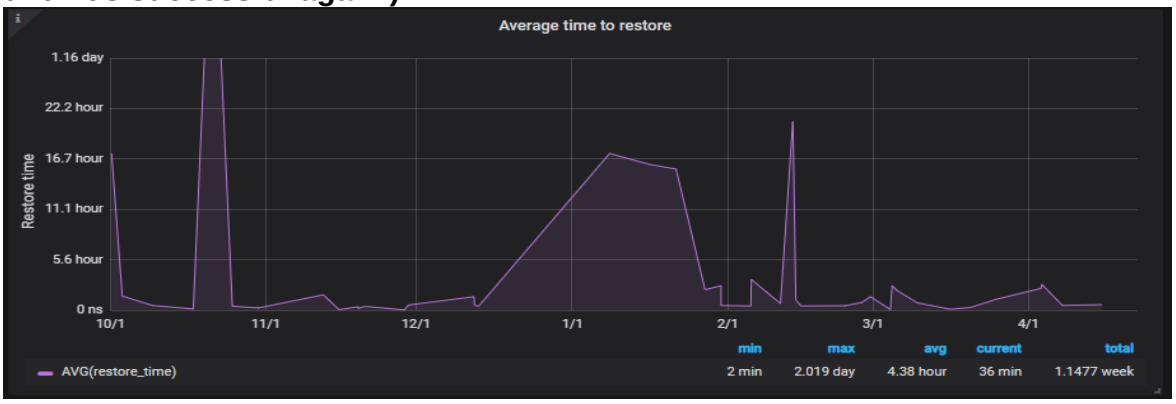
Note: the data for releases after March 10th is currently missing.

Oldest done feature (the feature which have been done the longest amount of time but not yet included in a release)

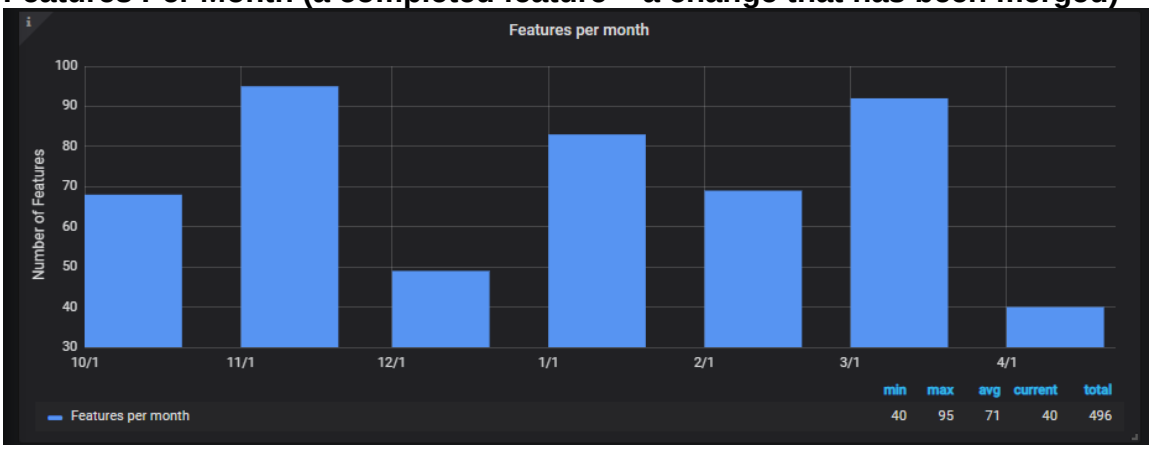


Note: the features shown here are in a test branch and should actually be excluded from the data. This graph is only for demonstration purposes.

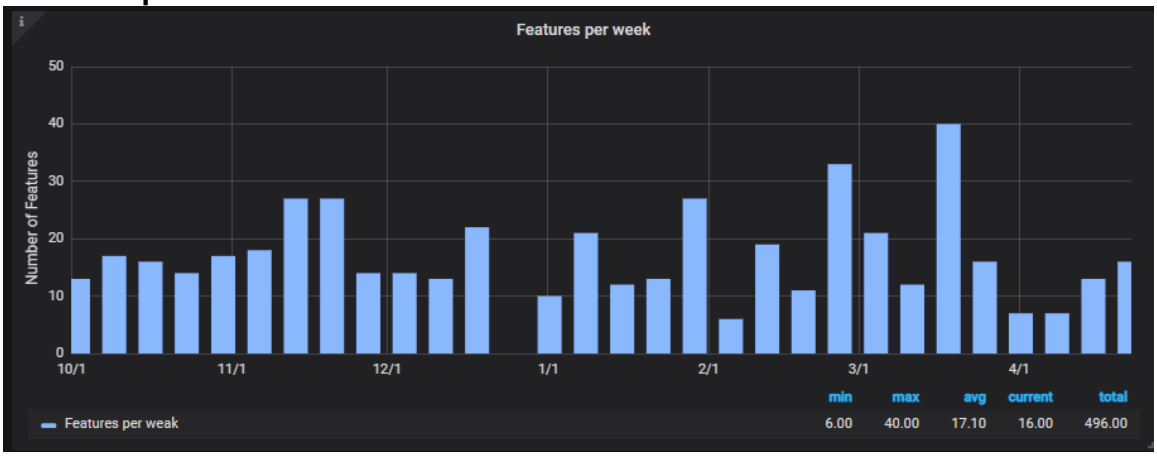
Average time to restore (time it takes from when a once successful build fails until it's successful again.)



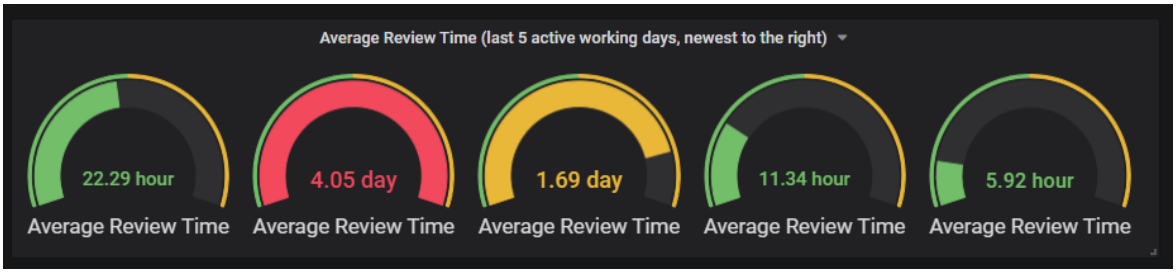
Features Per Month (a completed feature = a change that has been merged)



Features per week



Feedback of recent review efficiency



Summary of shown metrics

- New changes per day (Number of patch set 1 uploaded per day)
- Patch sets per day
- Successful builds per day
- Failed builds per day
- Build success ratio per day
- Build on commit time (including automated tests)
- Build on commit time (including automated tests, with filtered outliers)
- Review time (time between the first patch set was uploaded until given +2 in Gerrit)
- Development lead time (time between the first patch set was uploaded until it was merged)
- Deployment time (time between a change was merged until it was released)
- Oldest done feature (the feature which have been done the longest amount of time but not yet included in a release)
- Average time to restore (time it takes from when a once successful build fails until it's successful again.)
- Features Per Month (a completed feature = a change that has been merged)
- Features per week
- Feedback of recent review efficiency

DEPARTMENT OF COMPUTER SCIENCE | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2020-06-11

EXAMENSARBETE Continuous Delivery: Challenges, Best Practices, and Important Metrics
STUDENTER Anders Klint, Vilhelm Åkerström
HANDLEDARE Lars Bendix (LTH), Axel Franke (Bosch), Peter Walls (Bosch)
EXAMINATOR Elizabeth Bjarnason (LTH)

Challenges, Best Practices, and Important metrics for Continuous Delivery of Software

POPULAR SCIENCE SUMMARY **Anders Klint, Vilhelm Åkerström**

Software teams being able to frequently and painlessly deliver new updates of their software to their customers has many benefits, however accomplishing this is not always easy. We have explored challenges and best practises of this type of workflow, as well as which metrics are valuable when evaluating the effectiveness of the process.

Releasing software can often be a tedious process requiring many work hours and quality checks required for a satisfactory result. Continuous Delivery (CD) is a software development workflow which aims to eliminate the tediousness of this process, and which has become very popular in recent years. To achieve this, the goal of CD is to automate steps such as building, testing, and releasing the software such that quality software can be released with a push of a button at any time in the development process.

However, with CD still being a new way of working there is a lot of uncertainty of how to do it well. The change to CD from other software development methods can not be done in just a day or a week. This is a gradual process which faces many different challenges. We performed a study to discover what these challenges are, in addition to some best practices to aim for when looking to achieve this workflow.

Some of the common challenges were keeping up with automation, due to technical challenges such as tests taking too long to run, or a lack of commitment to automate from developers or managers wanting to focus on developing new features. In order to combat this, it is important for everyone involved to be on board with the processes and work necessary to do CD, in addition to sharing responsibility for the CD-pipeline, as well as making sure that the team has proper experience of CD through training and hands-on experience.

Additionally, some metrics gathered from literature were used to monitor a team having achieved CD at the case company. This resulted in three major findings: Firstly, that limited licenses to software tools used to quality check newly written code can throttle the process if the number of instances of the software is limited by the number of licenses. Secondly, that a tool used to generate documentation on each code change took longer to run than anticipated, and could thus be run less frequently. Thirdly, that when measuring duration, it is important to not only measure the time taken for each process to run, but also the idle time between processes.

These findings of the monitored team could inspire other teams working with CD of how to improve their workflow. Additionally, the challenges, best practices, and important metrics found could be useful for practitioners and future researchers.