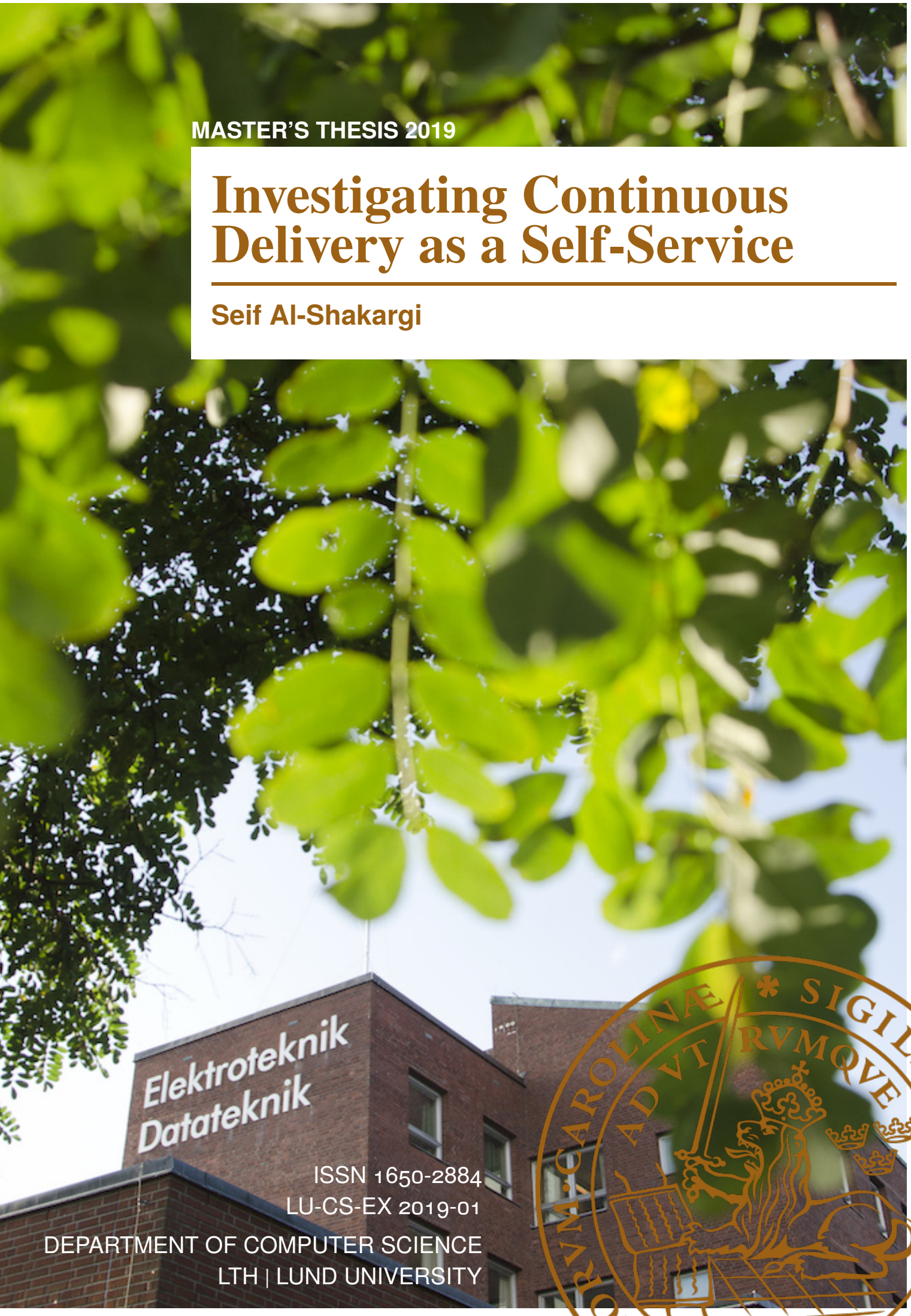

MASTER'S THESIS 2019

Investigating Continuous Delivery as a Self-Service

Seif Al-Shakargi

ISSN 1650-2884
LU-CS-EX 2019-01

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2019-01

Investigating Continuous Delivery as a Self-Service

Seif Al-Shakargi

Investigating Continuous Delivery as a Self-Service

Seif Al-Shakargi
seif.al-shakargi.086@student.lu.se

February 17, 2019

Master's thesis work carried out at Softhouse Consulting Öresund AB.

Supervisors: Lars Bendix, lars.bendix@cs.lth.se
Fredrik Stål, fredrik.stal@softhouse.se

Examiner: Ulf Asklund, Ulf.Asklund@cs.lth.se

Abstract

Continuous Delivery (CD) has today become an essential part of software development for a reliable and automated quality assurance process. However, the challenge of introducing CD starts with setting up a CD pipeline and providing the necessary infrastructure for it, which may take a considerable amount of time and require expertise knowledge. This complex and costly implementation will also require maintenance once in use. The expertise required for CD is not always available in smaller teams making the QA process suffer. That is, why this master thesis has researched how software development teams can setup and maintain CD without needing expertise with a Self-Service. By look into usability, maintainability and cost aspects, a requirement specification for a Minimum Viable Product was set. A proof-of-concept design of a service was drawn up and validated with a small team by testing Software as a Service tools.

Keywords: Continuous Delivery, Usability, Maintainability, Cost-benefit, as a Service, non-experts

Acknowledgements

First, I would like to thank my supervisors Lars Bendix, LTH, and Fredrik Stål, Softhouse, for their continuous support and useful feedback during the work process of this master thesis. Also, I would like to thank the stand-in supervisor Jon Nessmar, Softhouse, for guiding me in the beginning of this thesis. Next, I would like to thank my examiner Ulf Asklund for his useful feedback on this master thesis report.

Furthermore, I would like to thank everybody in Softhouse Malmö for welcoming me and I would like to specially thank everybody who has been a part of my research. Lastly, I would like to thank my family for their support and patience throughout my university studies.

Foreword

As way to help the reader, a target group is specified, here in the foreword, which will set expectations on the details presented in this thesis. The target audience of this master thesis are developers, with moderate knowledge of Continuous Delivery, in software development teams considering to implement or change their current approach to Continuous Delivery. This thesis serves specially developers working in smaller teams as this thesis has studied one.

Contents

Chapter 1

Introduction

Continuous Delivery, CD, is essential in a development environment for quick feedback and to shorten the release cycle of products. Over the past decade, the benefits of CD have been well studied and are well known today, giving software development teams an incentive to adopt CD. However, not all software development teams have access to the expertise knowledge needed to implement CD in their team or from external partners. This master thesis has studied how to implement and maintain Continuous Delivery in a simple and cost-efficient way with a Self-Service. A part of the research has been assessing a small team's working environment, and their current CD system, as well as how CD delivered as a Self-Service can facilitate and sustain a CD pipeline for the team. This small team is specialised in delivering mobile ticket solutions and works with several customers as a part of their work. Introducing a service to their quality assurance workflow will be both beneficial and cost effective, as this approach will not be needing expertise knowledge and can be quickly used.

In software development teams Continuous Delivery is handled by either an expert in the team or an outside team responsible for several teams CD pipelines [7]. These approaches create a reliance and dependency on a number of individuals. The experts are often sent out to customers, in IT-consulting companies, making smaller teams loose the CD knowledge.

The setup of CD pipeline involves orchestrating an infrastructure base to build upon for computing capabilities, networking and storage. This time-consuming task, and often repetitive, requires knowledge within system, application and network configuration, which most developers do not possess [7].

An implemented pipeline, without the needed expertise, results in a not properly maintained pipeline due to the complexity behind maintaining it. The complexity of maintenance originates from configuration issues regarding the tool chain implemented, that the

developers are not knowledgeable about.

The goal of this master thesis has been to research and define a service in the area of Continuous Delivery to serve for future implementation.

The main purpose of this master thesis is to facilitate easier implementation of CD in a cost-effective manner, making it available to software developers with moderate knowledge of CD. In order to do that a reduction of time in setting up the CD pipeline, a minimisation of the CD maintenance, an abstraction of the outside factors relating to CD has to be achieved. In addition, making CD a priority and responsibility for the whole team.

Initially, two research questions and a hypothesis were formulated:

RQ1. What kind of infrastructure is needed for implementing CD as a service?

RQ2. How can this service provide known quality?

Hypothesis: This service will cut the initial costs of implementing CD.

As the research progressed new and interesting aspects were discovered. The focus of this thesis instead pointed to usability and maintenance aspects to Continuous delivery from a user's perspective as well as cost factors relating to CD.

The contents of this thesis are listed below as well as a short description of them.

Background - provides a short history on the topic of CD, the context of this thesis, an explanation of CD and the concept of as a service and lastly the used research methodology.

Analysis - presents an assessment of the target team, requirements of a Minimum Viable Product for the service, testing of tools in the area of CD, interviews with CD experts and lastly final requirements.

Design - discusses functionalities of a future service, proposals for implementing the Minimum Viable Product, a cost-benefit analysis and infrastructural components of Continuous Delivery as a Self-Service, CDaaSS.

Discussion & related work - presents a discussion about the thesis in general, validation of results, related work, reflections & limitations and future work.

Conclusions - presents a summary of results from this thesis.

Chapter 2

Background

This chapter will help the reader in understanding the purpose and motivation behind this thesis and set the scene going forward. The aim of this thesis has been to research how teams can be made CD self-sufficient by identifying current challenges in a small team's CD system. In small teams, CD experts are not always available that is why maintainability and usability aspects of Continuous Delivery are explored in this thesis. For a team to set up and maintain Continuous Delivery, it should be delivered as a Self-service.

First, a short history about research in the area and a description of what Continuous Delivery entails will be presented. Next, the context of this thesis and definitions of Continuous Delivery and the concept of as a Service will be clarified. Lastly, to better understand the results of this thesis, the research methodology is discussed and motivated in detail.

2.1 History

To better understand Continuous Delivery, this section describes how it emerged and has developed over time. It all started when a number of intellectuals, within the area of software development, were gathered to discuss development methods. The end result of the conference was the Agile manifesto and 12 principles for supporting it, which can be found here [28]. The first principle listed was "Our highest priority is to satisfy the customer through early and continuous delivery of valuable software". This principle became the starting point for Continuous Delivery. In order to provide instant feedback and to shorten the release cycle when developing software, tools and processes started to emerge to support agile software development.

Fast forward from year 2001 when the Agile Manifesto was published, Continuous Deliv-

ery's benefits and challenges have been thoroughly researched since then, when switching from traditional waterfall to agile development. One example is an article by Lianping Chen highlighting the following benefits when adopting Continuous Delivery [6]:

- Accelerated time to market
- Building the right product
- Improved productivity and efficiency
- Reliable releases
- Improved product quality
- Improved customer satisfaction

In the study a dedicated team of 8 people worked for 2 years to implement Continuous Delivery in a large company. Similar work, highlighting the experience of implementing CD, can be found here [29].

However, these resources, available in a large company, are not always at hand in smaller teams and companies. Therefore a study into a user's perspective to Continuous Delivery in a smaller team might bring new insights.

2.2 Context

To help the reader understand and apply the results found, this section describes the working environment related to the company and team where this thesis was conducted.

This master thesis was conducted at an IT-consulting company called Softhouse, which provides their expertise within Continuous Delivery among other areas. The expertise is provided to customers through their in-house development and IT-consultants. Based on customer demands, Softhouse finds the adequate in-house team, with the necessary expertise, to take on projects. Their adopted way of working with software is Agile in particular SCRUM.

The studied software development team is an in-house team dedicated to application development specialised in mobile ticket solutions. Their activities lie in new development and maintenance of software. The team consists of 5 members, 4 developers and 1 agile coach. The developers are split into front-end and back-end development. The team works with several projects in parallel, that is why flexibility is important. They use a CD pipeline, but do not possess the expertise to set up and maintain a pipeline.

2.3 Continuous Delivery & as a Service and Self-Service

Since there are many definitions, this section clarifies the definitions of Continuous Delivery, as a Service and Self-Service, used throughout this thesis, and how these combined form Continuous Delivery as a Self-Service. Two definitions of Continuous Delivery are

discussed in this section. A Continuous Delivery pipeline is also discussed and examples of tools are given that can form a pipeline.

2.3.1 Continuous Delivery

The first definition of Continuous Delivery given by Lianping Chen is the following [6]:

> "Continuous Delivery (CD) is a software engineering approach in which teams keep producing valuable software in short cycles and ensure that the software can be reliably released at any time."

Another definition of Continuous Delivery given by Jez Humble is the following [19]:

> "Continuous Delivery is a set of principles, patterns, and practices designed to make deployments—whether of a large-scale distributed system, a complex production environment, an embedded system, or a mobile app— predictable, routine affairs that can be performed on demand at any time."

The definition given by Chen is specifically referring to teams and producing software in short cycles, whereas Humble's definition is more general. That is why moving forward we will be using Chen's definition of Continuous Delivery. In our context, the focus is on a software development team working in short cycles.

To be able to achieve Continuous Delivery software development teams need to have the software ready for deployment at all times. Software is made reliable by automating the steps in the release process, which form the CD pipeline, also called the deployment pipeline. The steps involves building, testing, integrating new code into the system continuously and delivering [20][p.3]. Continuous Delivery puts an emphasis on releasing at any time.

An illustration of the steps in a deployment pipeline is given in figure 2.1. The pipeline progresses as the steps are completed, and if a step fails the user is provided feedback. The tool stack can be different depending on the organisation implementing the pipeline [20][p.3].

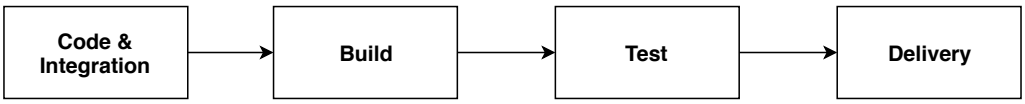

Figure 2.1: An illustration of the steps in a Continuous Delivery pipeline.

In the first step Code & Integration, git can be to version control code and integrate different commits by developers. In the build step, depending on the application being built, applications can be built with Gradle or Maven, where build parameters can be specified. In the test step, unit testing, functional testing and acceptance testing can take place with tools like JUnit or Selenium, where these tools provide test reports as a result. Lastly, in the delivery step Heroku or Pivotal web services be used to deploy the application to the desired environment. More about the tools that can be used in a pipeline can be found in this systematic literature review [31].

Tools like CircleCI, GitLab CI and Jenkins, tie these steps together forming a pipeline and support for the different tools used in the stack. In these tools the data about the progression is visualised.

2.3.2 As a Service and Self-Service

The concept of as a Service is widely used in Software, and referred to as Software as a Service. It is related to computing capabilities and technologies offered through the web [36]. The purpose of as a Service is to provide the customer with services directly usable without having to install software and provide the necessary infrastructure for it. Some of the advantages are cutting costs as you pay for what you use and providing expertise on-demand.

Combining the concepts of Continuous Delivery and as a Service, we get Continuous Delivery as a Service, CDaaS. This service makes Continuous Delivery directly usable without having to install tools and providing the necessary infrastructure.

Building on the concept of as a Service, a Self-Service for Continuous Delivery addresses usability and maintainability aspects, focusing on breaking the initial adoption hurdle of CD by guiding users in the right direction by automating tasks and minimising configurations. The users are software developers without the technical expertise for setting up the infrastructure involving Continuous Delivery. Using as a Service tools can still be complicated from a user's perspective. That is why Continuous Delivery as a Self-Service is suggested, that is adjusted to software development teams with the advantages of as a Service tools. This definition is explored throughout this thesis.

The difference between as a Service and Self-Service is that the Self-Service takes into account usability aspects for making implementations and configurations of a pipeline easier, so that developers can do it by themselves.

2.4 Research methodology

To understand the outcomes of this thesis, this section discusses the research methodology and work process used. First, this section describes the approaches used and later on discusses alternative approaches that were disregarded.

Below the steps taken in the work process are discussed. The first step was an assessment of the team, that was used to define requirements. Later, tool testing took place with the requirements as a base. The requirements were further investigated by interviewing CD experts. After the feedback gained from testing and interviews, a validation of a design took place with the team. This thesis was summarised by defining components of a future service. In parallel, a literature study was conducted.

Assessment

In order to get a complete overview of the target team's Continuous Delivery system and practices, an assessment was done. The overview was intended to help identify current challenges in the system and how a service might help and how the team could be made CD self-sufficient. This assessment was composed of an interview with two developers and the team filling out an electronic form distributed through Google Forms, which can be found in Appendix A. The interview questions asked were based on the company's maturity model [32], since it was relevant in the context and clear and concise. However, two categories were added general and team, where the purpose was to understand the team and system infrastructure better. To get initial understanding of the team's system and knowledge the form was handed out. The form was distributed before the interview for insight and preparation as well as a base plate for further discussions when conducting the interview. An important note is that the form also provides privacy in comparison to the interviews, which might bring forward other interesting things. A central part of the assessment was also informal discussions with the team.

Requirements

To address the challenges found in the assessment, requirements were extracted. The requirements were based on the target team's system and practices and how a service could help developers. To be able to differentiate the characteristics of the extracted requirements, three categories were introduced: Absolute Minimum, Minimum Viable Product CDaaSS and Target team. The first category was essential to have a CD system. The second category was requirements specific to the service to show its value. The last category was requirements related specifically to the target team, based on their wishes. The majority of the requirements was high-level requirements of CD and the service. The goal of the requirements was so that they can be used as a base when testing different tools and studying the concept of CDaaSS.

Tool testing

To explore current available and suitable tools in the area of CD that can be set up as a service, three tools were tested. The tool tested were CircleCI, Jenkins and GitLab CI and where chosen based on their popularity. A part of the exploration was to find tools sufficiently simple so that the team can set up and maintain a CD pipeline on their own. Firstly, the tools were tested through tutorials found on their respective website for building simple pipelines. Secondly, the tools were evaluated and tested against the requirements set. This was done in order to find potential candidates for a service as well as find out how well today's tools are in the aspects of usability and maintenance. The end result was each tool mapped against the requirements. Two potential tools were found adequate to be used as a service, CircleCI and GitLab CI.

Interviewing experts

To create a wider perspective on the technical aspects of CD, two CD experts were interviewed. In the interviews, questions were asked regarding current available tools and the requirements set, found in Appendix B. The discussions helped clarify already existing requirements and also add new ones.

Validation with team

The purpose of testing with the team was to validate and get domain knowledge and feedback by the intended users of CDaaSS as this was a new area. Each member of the team was handed a project to build a pipeline for, with an accompanying problem formulation where what steps to perform were written, found in Appendix D. To show initial value of the tools, the testing was time limited to 2h. The two tools, CircleCI and GitLab CI, found in the testing phase were selected to be validated. In order to get feedback about the tools and a future service, a Google form was distributed to be filled out after the testing.

Defining components

To break down the complexity of CDaaSS, the studied components of CDaaSS were written down in a mind-map. This can help a future implementation of a service adhering to usability and maintenance aspects. Lastly, a methodology for working with a service in a small team was proposed as well as proposals for a future implementation.

Literature study

To find inspiration and similar work and to get a better idea of what has not already been covered by research, a literature study was conducted in parallel. In this study Google Scholar, LUBsearch and LUP student papers were used to find relevant work. A part of this study was to use relevant keywords when searching for the papers. Relevant literature was examined by reading their abstract, introduction and conclusion in that order, thereafter a decision was made if the literature was to be added into the literature list. Addressing the initial hypothesis written in the synopsis, a cost-benefit analysis was done with a literature study. For inspiration, relevant blogs and websites were visited connected to CD.

2.4.1 Alternative approaches

This section highlights alternative approaches that were disregarded during the work process.

A complete literature study was disregarded since this thesis explores a new area and it is not suitable in a company context.

After having done the assessment, many questions regarding the team's work process emerged that could have been deeply investigated. However, in-depth interviews were disregarded as some of the findings were out of the scope of this thesis. Another reason was the time-consuming task to conduct and schedule interviews fitting the team's schedule.

Implementing a CD pipeline in the team's system was disregarded as it will work against the purpose of this thesis. There would have been a possibility of me becoming the new CD person in the team, by implementing the whole pipeline and maintaining it.

Implementing a proof of concept was also disregarded as this was not the goal of this thesis, which was the team should be able to do it on its own. If an implementation was done, the team would not had the possibility of testing tools usability aspects .

When choosing a project for the team to set up for the validation, there was the possibility of choosing a project the team was working on. However, doing that would take the validation to unknown territory and a project not set up before. That is why it was disregarded, as it can take more than 2h.

Chapter 3

Analysis

To identify usability and maintenance issues of Continuous Delivery regarding infrastructure and cost, an analysis was done. In order to analyse how a software development team can be made CD self-sufficient, requirements on a future Self-Service were set. The initial requirements were worked with iteratively to gain feedback and further clarify them by analysing tools and interviewing experts.

The analysis was composed of assessing a software development team, setting initial requirements for a future Self-Service, testing different CD tools and lastly interviewing CD experts. The final section of this chapter lists the final requirements.

3.1 Assessment

For a complete overview of the target team's Continuous Delivery system and practices, an assessment was done. The overview was intended to help understand the team's CD needs and how the team can handle their own release process. As part of the assessment, an interview was conducted with two team member about their CD system. To get an initial understanding of the team's CD system and knowledge, a form was distributed before conducting the interview. A part of the assessment has also been informal discussions with the team members. The assessment and form questions can be found in Appendix A.

The outcomes from the interview and the filled out form, are analysed below. To understand the status of the CD system, the structure follows the categories from the assessment questions: General, Code & Integration, Build, Test & Quality Control, Delivery and Team. Thereafter, reflections from the interview and the form are discussed. Lastly, concluding thoughts are presented.

General

As a starting point in the interview, to create a general view and understand the structure of the system, the tools used and the development process were discussed. Some of the tools used in system are GitLab for version control, Xcode for the iOS platform and Jenkins for Continuous Integration. However, Jenkins is disabled today, adhering to the lack of expertise to maintain it. The IDEs used in the team are Xcode, Android Studio and Visual Studio.

The steps taken by the team in the development process are: develop code, commit, build, unit tests, manual acceptance tests, delivery. After the release is delivered to the customer, the project can re-enter the development process depending on the customer's notes.

Code & Integration

As part of understanding the team's practices, the next category of questions were discussed. Adhering to the different nature of the projects taken on by the team, the frequency of integrating code varies. In the work process, the team has adopted GitFlow as their branching strategy, as it makes feature development possible. In GitFlow, two branches are used, one stable master branch for deliveries and one develop branch for feature development [3].

Documentation on the different projects have been mainly driven by customers, as some see the value of it. Details in the commit messages and the architecture of the system, have been documented by the developers as they see fit. To understand the system, team members have helped each other. To communicate with the customer, JIRA has been used to handle stories and bugs. JIRA provides a visualisation of a Kanban board and other relevant information like the latest releases and statistics, needed in agile software development.

During the development phase of software an agile mindset is adopted within the team with sprints, tasks and frequent releases. However, a less agile mindset has been present as the software products evolve, since they have less activity and enter the maintenance phase of development. Since customers have different experiences with agile software development and have worked with deadlines rather than sprints, there has been a project where the team has been working on for 1 year without a release.

Build

In the next category of questions related to the CD system, the build process was discussed. The build processes in the team are automatic and triggered by commits, where build logs and binary build files are stored at a central build computer. There are three builds in the system: internal, customer and release builds. They are built in the same way but with different configurations. In the iOS system, to take part of the CD functions, Xcode has been used for building, an automatic static code analysis and display dependencies. The Android system uses a Gradle plugin as a build tool in Android Studio.

Test & Quality Control

In the next category of questions, Test & Quality Control were discussed. In the system the team works on, there are automated unit tests and UI tests. UI tests are in the form of happy path testing, which follows one path to test basic functionality with no regards to error-handling [20][p.85]. A lot of happy path testing may slow down the build process of the application, which is the reason behind less automated tests according to one interviewed developer.

Test results from builds are available to developers from the central build computer, and also at GitLab and Xcode. Tests are conducted for different configurations like different operative systems such as iOS and Android and different mobile phones. However on both the Android and iOS side, manual testing is present to a great extent, as there is no tester in the team. The tool Fabric is used for crash statistics and helps the team reproduce bugs, but hardware errors are more difficult to reproduce. Bugs found from the customer is also well enough documented to be able to reproduce.

The team's quality assurance were better before when there was a tester in the team, as there was a cooperation between developers and testers and well defined roles. This also serves as a explanation why manual testing is present. Today, the team members have adopted two roles, as developers and testers.

Delivery

In the next category of questions, the delivery process was discussed. The deliveries are automated in the system, by specifying a target name. Delivery dates are primarily decided by the customer, which controls the workload for the team. Short delivery dates change the team's focus to the assigned project entirely, according to a developer in the team.

In the beginning of a certain project, deliveries have been made every two weeks. However, when the project went into maintenance only new operative systems or external updates have forced new releases, or of course bugs too. Releases notes are included with deliveries, but they are not automatically done.

When releasing the applications to market there is no downtime. There are two identical servers to reroute the traffic in order to avoid downtime. The traffic is rerouted incrementally to the other server. It is possible to follow bugs/issues regarding different releases in JIRA. However, the approach to reporting bugs differs between customers.

Team

In the last category of questions, the team structure was discussed. The team consists of 5 members, 1 agile coach and 4 developers. 2 of them work with backend, 1 with Android and 1 with iOS. One of the developers works from another office, however there is a well established communication channel, with Slack, to help the team communicate.

The latest assessment that was done of the team was 3 years ago and it was also done by a master thesis student. One thing found by the student is how waterfall testing takes place in the final stages of development as well as highlighting difficulties with automated GUI tests when developing mobile applications. The difficulties of automated GUI testing lie in changing system features such as the system clock and turning on flight mode, which is prohibited according to the student [35].

There is not an appointed CD responsible in the team, but one developer has been handling some things regarding CD out of interest. Sometimes the work done by the developer within CI/CD has felt like a part time job for him, since some projects has been needing special configurations.

All parts of CD are implemented at the start of projects. One example is a function disabled today, where changes had to pass tests to be incorporated into the main branch. After the tester and the CD expert, left the team and lack of time and priority to fix issues, the CD system started to deteriorate and functions were disabled. The team's CD system has not been maintained properly ever since the responsible CD expert left the team. This has made the developers look for ad-hoc solutions to their CD needs. For instance Xcode has been used to fill many of the CD functions, that before were not up to date.

3.1.1 Reflections

Interview

The CD system in the team has been great at the start of projects, however as the projects are done and enter the maintenance phase the CD system has not been prioritised. Lack of time from developers has been one of the reasons behind prioritising other tasks. An in-depth discussion about costs for a team of 5 can be found in 3.3.4 and 4.3. This has resulted into ad-hoc solutions by the developers and sometimes not into a complete CD pipeline. A future service will address this challenge.

A lot of time is spent on manual testing in the team. For instance, the Android developer manually tests new functions on 3 applications across 7 mobile phones. In a systematic literature review, it was found that despite automating GUI tests, the QA members still manually checked tests citing reliability concerns [31]. The team iterated that they miss a tester and CD expert in their team. My general assessment is that the software the team is working on is stable, which is why they can manually test with confidence to a high degree.

It has happened that the developers wanted to set up a CD pipeline, but did not have the knowledge how to and time. A future service can provide an easier adoption to Continuous Delivery, by addressing the set up and maintenance and not requiring expertise knowledge.

Form

The trust in the CD system is average, 3.2/5, and the knowledge of CD is below average, 2.4/5, in the team. It seems like the most difficult parts to understand are Test & Quality Control and Delivery. The members in the team point out that the most important things within CD from a developer's perspective are automation, security, smoothness, automated tests, an easy to understand pipeline and high trust in the system.

There is an understanding within the team that there is not an appointed CD person in the team. When asked about complications, most team members point to Test & Quality Control. The average score of how common complications are in the CD system scores 3.2/5, which is slightly above average. However, the speed of fixing these complications are close to average, 2.6/5. The interesting question of how often do the developers fix these issues, scores fairly low, 2.2/5.

The team members point to different things regarding difficulties with maintenance and usability. Sharing knowledge is one of the things pointed out and that there is not a clear picture on who does what and how it is done on the different platforms. An agile mindset is suggested in regards to deliveries. Another thing pointed out is that it can be messy to deal with different systems instead of one, e.g. there are one for integration/test and one for deployment. Another member pointed to the fact that it might be annoying to switch between maintenance and development of software.

The maintenance is a task requiring time the developers do not have when developing software, according to one team member. While saying that usability is not an issue, since everybody can learn for instance how to use Jenkins. Lastly, a member points to the difficulties within involvement and understanding when it comes to usability and maintenance.

3.1.2 Concluding thoughts

The assessment points to the challenge of how to keep the CD pipeline at work as the projects evolve or go into maintenance. A future Self-Service will address this, by providing easier maintenance and infrastructure of the CD pipeline. So that once the CD pipeline is set up, it stays up by the service making it available.

The manual task of providing the infrastructure involves creating, installing and configuring a Virtual Machine as well as setting security and load balancer configurations. To simplify this process, an automatic procedure is required and can be achieved by making use of the concept of as a Service [7].

The CD system in the team need improvement in the area of CI and test, in order to cut the time spent on manual testing. An improvement in these areas can increase the trust in the system and make the system smoother than it is today. This can also reduce the complication occurrences in the CD system.

The form answers point to complications in Test & Quality Control in the CD pipeline and experienced difficulty to understand the same area by the developers. The team would

greatly benefit of more CD knowledge, that could be accomplished by education and sharing of knowledge within the team. The answers also point to the benefit of improving the Test & Quality Control. Another interesting aspect that the team would greatly benefit from is one system that is easy to understand.

3.2 Initial Requirements

To address the challenges found in team's CD system from the assessment, initial requirements were set for a future service. Aspects of usability and maintainability, are discussed in this section. In addition, the initial requirements include wishes by the team to simplify their work process. As a starting point, the initial requirements purpose was to be used for further analysis. To be able to differentiate the requirements, three categories were created: Absolute Minimum, Minimum Viable Product CDaaSS and Target team. The majority of the requirements are high-level of CD and the future service.

First, in this section, the different categories of the initial requirements, Absolute minimum, Minimum Viable Product and Target team, are motivated and discussed. Second, the requirements are summarised in a list.

3.2.1 Absolute minimum

The requirements listed in this category, are the first building blocks of CD and without them, the deployment pipeline is not possible. An illustration of the deployment pipeline can found in [20][p.4] and in 2.3.1. The category includes build, code & integration, test, delivery and feedback, however at their absolute minimum for a working CD system and to facilitate a continuous chain of events, i.e. a pipeline. To be able to resolve issues, feedback from builds are included. The requirements are listed below.

1. Automatic builds are triggered by commits/changes.
2. Integrates with SCM, software configuration management, tools such as Bitbucket, GitHub, GitLab, (Gerrit).
3. Run ”scripts” for test and deploy.
4. Create a collection of stage blocks, i.e. a pipeline.
5. Report results of builds/runs, i.e. feedback.

3.2.2 Minimum Viable Product

To show the value of a service and address challenges found in the assessment, the minimum viable product, which includes the absolute minimum requirements, is presented below. The following paragraphs will be motivations and discussions behind each requirement in the product.

Req 1: No appointed CD maintainer in the team. *Category: maintenance, quantitative.*

To address the challenge, found in the assessment, of needing expertise when setting up and maintaining a CD system, there should not be a reliance and need of an expert to use the service. This means the service needs to facilitate easy maintainability and break the initial hurdle of adopting CD, by automating expert related tasks such as infrastructure provisioning. A CD platform should be easy to use for a team to implement a CD pipeline by themselves [7].

Req 2: Team only concerned of code, tests and pipeline configuration code. *Category: usability.*

To address usability aspects and the expertise needed by the service, the team's main concern should be the code, the tests and the pipeline configuration code when developing software. The stages in the pipeline should be the team's concern to CD. To be able to manage the pipeline and reuse it, the configuration code should be version controlled.

Req 3: Reliable and trustworthy:(a) CPU, RAM, storage are provided based on need. (b) High availability -> No downtime. *Category: infrastructure, functional.*

For the service to be reliable and trustworthy, resources and availability should be provided. Infrastructure is mentioned as one of the technical adoption challenges of CD [9], that is why resources like CPU, RAM and storage, relating to not needing expertise from Req 1, should be easy to scale.

Two quality requirements listed by Chen for a CD platform are reliable and scalable, to support the development process and the growing number of applications [7].

To not disturb the work process and to deliver software fast, the service should provide high availability, meaning no downtime when deploying or using the service. The team members should not have to wait to test their builds.

Req 4: Secure to manage secrets. *Category: infrastructure, functional.*

A part of having a CD system, requires using sensitive data and keys. To protect production systems, the CD platform needs to be secure [7]. That is why a functionality for managing secrets should be provided.

Req 5: Service updates are provided. Tools/plugins updates are available on demand. *Category: infrastructure, maintenance, functional.*

To combat compatibility issues, that may arise, and part of making the service easy to use, the service has automatic updates and with tools/plugins updates available on demand. The choice behind tools/plugins updates being available on demand, is to tackle compatibility issues from other tools/plugins and to not disrupt the CD system suddenly.

Req 6: Error detection: (a) Traceable from failure to origin of error. (b) Reason for failure clear and directly visible to user. *Category: usability, functional.*

To be able to debug the pipeline, error detection need to be provided. That is why the error should be traceable to the stage or file where the failure occurred. To save time and to adhere to user friendliness, reason for failure should be extracted from script logs and clearly visible to the user.

Req 7: Lower cost than appointed maintainer(s). *Category: whole product, cost.*

For the service to be beneficial, the cost of this service should be less than the cost of appointing a CD maintainer. This should also serve as a motivation for using the service.

Req 8: Should not take longer than 2 days to start using service. *Category: usability.*

To be able to configure the system easy and fast, a time limit is set on 2 days for education and setting up the pipeline for the service. Also, if the CD system needs changing and the service no longer can provide the customisation needed, it should not be felt like a lot of time has been wasted. This also might give the team an incentive to return to the service after encountering complex CD systems that take long time to set up.

3.2.3 Target team

The requirements in this category, found in 3.2.4, are related to the target team studied. The first requirement is a requirement the team suggested would help their work. As for the second and the third requirements are completely tied to their working environment. The last requirement is a requirement which would hugely benefit the team secure their working process and save them time from manual testing.

3.2.4 List of initial requirements

This is a summarised list of requirements.

Absolute minimum

1. Automatic builds are triggered by commits/changes.
2. Integrates with SCM, software configuration management, tools such as Bitbucket, GitHub, GitLab, (Gerrit).
3. Run ”scripts” for test and deploy.
4. Create a collection of stage blocks, i.e. a pipeline.
5. Report results of builds/runs, i.e. feedback.

Minimum Viable Product Continuous Delivery as a Self-Service

1. No appointed CD maintainer in the team.
2. Team only concerned of code, tests and pipeline configuration code.
3. Reliable and trustworthy:
   (a) CPU, RAM, storage are provided based on need.
   (b) High availability -> No downtime.
4. Secure to manage secrets.
5. Service updates are provided. Tools/plugins updates are available on demand.

6. Error detection:
   (a) Traceable from failure to origin of error.
   (b) Reason for failure clear and directly visible to user.
7. Lower cost than appointed maintainer(s).
8. Should not take longer than 2 days to start using service.

Target team

1. Visible to both team and customer that the software is in a working state.
2. Support for multiple platforms(.Net, iOS, Android)
3. Integration to Jira and Slack.
4. Support automatic tests on physical phones.

3.3 Testing tools

A part of the analysis, to explore today's tools and how well they can meet the requirements of a Self-Service, three tools were tested. Based on their popularity, CircleCI, Jenkins and GitLab CI/CD were tested. The tools were tested through tutorials found on their respective websites, for building simple pipelines, and were evaluated and matched against the requirements set. This was done in order to find tools sufficiently simple so that the team can set up and maintain a CD pipeline on their own. A difficulty level is specified for each tools tested. This experienced difficulty of the tool is based on my experience as a beginner in the area of CD. The intended users of the future service are users with similar experience and knowledge of CD.

First, each tool is discussed in the following order: CircleCI, Jenkins and GitLab CI/CD. In the analysis of each tool, a general impression, matching requirements and a summary are presented. Second, reflections about the tools are presented. Lastly, what is not sufficiently simple with the tools and new requirements are presented.

3.3.1 CircleCI(Cloud)

Version: 2. Difficulty level: 2/5.

CircleCI [21] is fast and easy to get started with for implementing CI/CD. There is a smooth integration to GitHub and BitBucket. Unfortunately, it does not support GitLab at the moment. Since CircleCI is a cloud-based system, it requires no maintenance and needs minimal configurations to get started with. There is the possibility of having an on-premises solution, which is only available for the enterprise version. However, you can run jobs locally on your computer with CircleCI CLI and Docker, but without a GUI. This results in some functions not available compared to the cloud solution.

For customisations you have to integrate with third party software, for instance for deploying or for special testing of your application. There are support for a significant amount of programming languages through docker images maintained by CircleCI. You could use private docker images if the language you use is not supported. However, CircleCI does not support a Windows environment at the moment, only Linux and MacOS.

Requirements

Absolute minimum

Req 1: Once you connect your git repository to CircleCI and have committed the .circleci/config.yml file, thereafter every commit will trigger an automatic build. CircleCI listens to every change in your repository. There is the possibility of skipping a build, by adding [skip ci] in the commit message.

Req 2: In order to sign up to CircleCI, you either use your BitBucket or GitHub account. I signed up with BitBucket. As of now, CircleCI works only with BitBucket or GitHub. After signing up you will be able to see which projects you have in your account and choose which one you would like to setup with CircleCI, by following a project.

Req 3: The only file needed for CircleCI is .circleci/config.yml file, where you can specify which scripts to run for the different stages. The configuration file is written in the YAML language, which is very indentation sensitive.

Req 4: In the configuration file, it is possible to create different stages and a chain of events easily.

Req 5: You will receive instant feedback after running CircleCI, if your build succeeded with a clear colour indication, green or red. The project build status will also be visible on your SCM tool. When a build has finished, the running time of the pipeline and each stage is available in CircleCI.

Minimum Viable Product

Req 1: Using CircleCI needs minimal to no maintenance at all, since it is easy to setup and your main concern is the pipeline configuration code. CircleCI provides detailed tutorials and templates to get started. Based on my experience with CI/CD configuration, I was able to setup my own project within an hour.

Req 2: During my testing I was only concerned with the code, the tests and the pipeline configuration code. The pipeline configuration code is version controlled and a part of the repository. I was able to reuse the configuration code several times when testing with minor changes.

Req 3: For the projects tested, CPU, RAM and storage was not an issue and was nothing I was concerned about. CircleCI provides better CPU and RAM if needed for a cost. During my proof-of-concept testing I did not experience any queuing of builds. All my builds ran immediately.

Req 4: There is the possibility of storing environment variables, in the Project setting -> Build settings -> Environment variables. This is better than storing the variables directly in the repository, to save them from getting leaked or in the wrong hands. This way the secrets are encrypted [2].

Req 5: In CircleCI you don't have to specifically update anything, you only specify which version of CircleCI to use in your configuration code and the docker images for the languages used in the project. This is done once in the header of the configuration file.

Req 6a: In the workflows tab, a clear view of the pipeline is visible and also which stage succeeds with a colour indication. In the builds tab, the reason for failure is traceable and clear. There is the possibility of SSH into the container to debug. This was helpful for me to understand paths in the beginning.

Req 6b: Failing builds due to configuration errors are clearly visible. However they can be prevented by testing the configuration file locally with CircleCI CLI. Also, failing builds due to tests are directly visible without having to dig deep into the log, if you have uploaded your test reports to CircleCI.

Req 7: My initial impression is that CircleCI is of a lower cost than appointing a maintainer, since you don't need expertise to get it started or to maintain it. If you need to upgrade for better CPU, RAM and storage you can do that for a price. Keeping in mind, that other non cloud-based tools require the same thing with the additional cost of maintainers.

Req 8: In order to start and set up the tool, it takes hours. It is very easy and fast. With education considered it may take half a day to a day to create a complex pipeline, based on my experience.

Target team

Req 1: There is a clear indication of the software's state with colour indications. The pipeline status can also be directly visible with a status badge in the readme file in your repository. An API key has to be configured for this purpose.

Req 2: CircleCI supports a lot of programming languages. It all depends on the available docker images for the languages. Both Android and iOS environments are supported, however iOS costs money. A Windows environment is not supported.

Req 3: Integration to Slack and Jira is possible through the project settings by going into Permissions -> JIRA Integration and for Slack by going into Notifications.

Req 4: Configuration of automatic testing on physical phones is not a built-in feature. Keeping in mind that this is a customisation requirement and seen from CDaaSS perspective, it is not fulfilled. However it is possible to set up a third party software like Firebase or Amazon AWS for automatic testing on physical phones.

Summary

The requirements for Absolute minimum and Minimum Viable Product are fulfilled, but 2 out of 4 are not fulfilled for the Target team, which are the requirements 2 and 4.

3.3.2 Jenkins(On-premises)

Version: 2.138.2. Difficulty level: 3.5/5.

Jenkins [23] is an open-source CI/CD tool and was tested on-premise. Its features mainly depend on the available plugins, which means it is highly customisable. It took some time to understand paths, when running both locally and in Docker. Everything is possible with Jenkins however the solutions will be custom and non-reusable. Jenkins has a startup time compared to the other tools, as there is a waiting time for Jenkins to load and to be usable in the web interface.

In order to get started, you have to first install Jenkins on your computer. Then to start Jenkins, you run a runnable file, where Jenkins is accessible on localhost:8080, if run locally. Thereafter you will encounter a GUI to setup Jenkins, where you install plugins and create an account. For me to get started I had to do some configuration, because it did not work initially on my MacBook. I had to change a line in /.jenkins/hudson.model.UpdateCenter.xml, from https to http. It took some time to find the fix and the file.

However, Jenkins can also be set up with Docker, which was easier for me in the beginning. You would first run the Docker application, then from your terminal run the docker command with additional flags specifying image, port, home library etc. There is the possibility to create a dockerfile, to start Jenkins with only one docker command, docker-compose up. The docker setup would have been impossible if not for the provided tutorials. One disadvantage experienced with Docker, is that it takes a lot of storage.

Requirements

Absolute minimum

The recommended plugins make Jenkins fulfil the absolute minimum requirements.

Req 1: It is not automatically set up. You have to configure triggers for automatic builds by yourself. I was able to setup a scan of the repository every 5 minutes and if changes are present run the pipeline. There is a better solution, but it was time consuming to find for a beginner. I tested another solution with a Github plugin but it did not work.

Req 2: There is an integration to SCM tools with the help of a Git plugin. I was able to integrate with both GitHub and BitBucket as well as with a local git repository. There is the possibility of using other SCM tools, like SubVersion, Clearcase and Perforce. However, their respective plugins have to be installed.

Req 3: The file needed is called Jenkinsfile, where scripts and stages can be specified. It is version-controlled and a part of the repository. The configuration file can be in two styles, either in declarative or scripted syntax.

Req 4: Stage blocks are created in your pipeline configuration code, the Jenkinsfile. It is possible to create a pipeline, with its stages, through a GUI provided by the BlueOcean plugin.

Req 5: In the project view it is possible to get status of your build, and if it has failed in which stage. When a build has finished, the running time of the pipeline and each stage is available. In the BlueOcean plugin, there is a nice overview of the pipeline.

Minimum Viable Product

Req 1: Using Jenkins requires a lot of steps in order to get it started. There is a lot of settings to configure and understand. It requires a lot of ad-hoc solutions, which means something is bound to fail. Based on my experience with Jenkins, one would have to appoint a CD maintainer or have someone in the team with Jenkins expertise to maintain the pipeline.

Req 2: When using Jenkins, you are concerned with more than the code, the tests and the pipeline configuration code. There are external things like project and configuration settings, account login, the runnable Jenkins file, storage, Docker if you are using it and additional servers configured to Jenkins.

Req 3: I did not experience Jenkins as reliable and trustworthy as I had to find ad-hoc solutions, for instance to get Jenkins running and to run automatic builds. CPU, RAM and storage are not provided, it is your own concern to provide. Also to ensure high availability you would have to set up a server or keep your computer running at all times.

Req 4: Jenkins is secure to manage secrets, as it stores passwords as secrets which are encrypted, and the decrypted secrets are stored in a secrets directory with the highest protection [25]. There is a possibility of configuring more advanced security with a Credentials plugin, which encrypts credentials on the Jenkins master instance by their ID, protecting it from other users [26].

Req 5: There is a plugin panel in Manage Jenkins -> Plugin Manager, where you can install and update plugins. If there are new essential plugin updates regarding security issues, you will get a warning to update. There is also the possibility to configure Jenkins to automatically update itself [24]. One interesting aspect found when searching on plugins, you might find several plugins of the same name, which is confusing.

Req 6: In the project view, the errors are traceable to the different blocks in the pipeline and clearly visible with colour indicators. The reason for failure is also visible without having to dig into the script log with the BlueOcean plugin.

Req 7: My impression is that it costs more to use Jenkins, because you would have to educate the team with the added costs of CPU, RAM, storage and connected servers. Relating to requirement 1 and 2, there is a lot of maintainability like plugins, servers and compatibility within the whole system.

Req 8: There is a risk that it takes longer than 2 days to get Jenkins fully working for complex projects, with the added education time. A lot of ad-hoc solutions are required to get started, seen from as a Self-Service perspective.

Target team

Req 1: There is a clear colour indication of the software's state, in the project view and in the BlueOcean plugin.

Req 2: Jenkins supports multiple environments and is highly customisable. It can support all languages with the added cost of installing them or using a docker image.

Req 3: It is possible to configure JIRA and Slack with the help of their respective plugins, which you have to install. Their settings will be found under Manage Jenkins -> Configure System.

Req 4: Automatic testing on physical phones is not a built-in feature, but can be configured since it is run on-premise. Alternatively, Jenkins can be linked to servers like Amazon AWS. This requirement is seen as fulfilled since in the nature of Jenkins everything has to be configured.

Summary

Jenkins fulfils both the Absolute minimum and Target team requirements, with some added effort. However, it fulfils only requirements 4,5 and 6 for Minimum Viable Product.

3.3.3 GitLab CI/CD(Cloud)

Version: 11.4.4-ee. Difficulty level: 2/5

GitLab [22] was tested after CircleCI and many similarities were found between them. One of the greatly admired properties is that GitLab is both an SCM and a CI/CD tool, which allows for more integrations and statistics. You can also integrate with other repositories that is not on GitLab, by providing the Git URL. The tutorials provided by GitLab are not as comprehensive and detailed as the ones provided by CircleCI. I tested GitLab CI/CD with GitLab and GitHub.

GitLab has more customisation integrations available than CircleCI, like Kubernetes integration. However, for customisations you have to still integrate with third party software, for instance for deploying or for special testing of your application. There is also support for a larger number of programming languages in GitLab than in CircleCI. Similar to CircleCI, there is the possibility of testing the configuration file before committing as well as running jobs on your machine with GitLab Runner. One thing that disappointed me was how slow the builds were in GitLab compared to CircleCI. The main reasons behind this are the shared runners and caching build data, that was not straightforward to do as in CircleCI.

GitLab provides a function called Auto DevOps, that creates pipelines automatically. However some preconditions have to be fulfilled. You would have to integrate to a Kubernetes cluster, set an ip-adress, and your language has to be supported. I tested this function with predefined templates, because not all projects can be used with it. One thing I noticed is how slow it is compared to when using a pipeline with manually written configuration code.

Requirements

Absolute minimum

Req 1: Once you add the configuration file .gitlab-ci.yml into your repository, GitLab will trigger automatic builds for every new commit. There is the possibility of skipping a build, by adding [skip ci] in the commit message.

Req 2: You can use your GitLab account directly to start with CI/CD. There is also the possibility to set up CI/CD for external projects, from other hosting service like BitBucket, GitHub or by providing a Git repository URL. GitLab CI/CD has integration with GitHub that is free function now, but will only be available for subscription members after March 22, 2019 [12].

Req 3: The only file needed for GitLab is .gitlab-ci.yml file, where you can specify which scripts to run for the different stages. The configuration file is written in the YAML language which is very indentation sensitive.

Req 4: In the configuration file, it is possible to create different stages and a chain of events easily.

Req 5: You will receive instant feedback after running the pipeline in GitLab, if the build succeeded or not, with a clear indication with the colours of green or red. When a build has finished, the running time of the pipeline and each stage is available.

Minimum Viable Product

Req 1: This requirement is fulfilled exactly in the same way as in CircleCI. Look at requirement 1 in Minimum Viable Product CircleCI.

Req 2: This requirement is fulfilled exactly in the same way as in CircleCI. Look at requirement 2 in Minimum Viable Product CircleCI.

Req 3: For the projects tested, CPU, RAM and storage was not an issue and was nothing I was concerned about. GitLab provides better CPU and RAM if needed for a cost. During my testing I did experience queuing of my builds. It seems like there is a lot of traffic in GitLab after 4 pm Swedish time. Errors with the shared runner occurred also concerning docker images.

Req 4: There is the possibility of storing environment variables in the project settings -> CI/CD -> Variables. This is better than storing the variables directly in the repository.

Req 5: In GitLab you don't have to specifically update anything, you can't even specify which versions to run of GitLab. This can cause worry as it can affect old and new releases when new functionalities appear in the updates. Therefore, adaption to the new updates must be done. However you can choose which docker images to use for the languages used in the project. So you don't have to worry about updating.

Req 6a: A clear view of the pipeline is visible and which stage succeeded with a colour indication, in the CI/CD tab - > Commit. There is the possibility of SSH into the container to debug and using an integrated web terminal to the deployed cluster in Kubernetes.

Req 6b: The reason for failure is not clear and directly visible even if you have uploaded your test reports to GitLab. However a test summary shows up when merging branches. This is a requested feature in the GitLab community and is underway to be

implemented [13]. On the other hand configuration errors are clearly visible and can be tested before committing.

Req 7: This requirement is fulfilled exactly in the same way as in CircleCI. Look at requirement 7 in Minimum Viable Product CircleCI.

Req 8: This requirement is fulfilled exactly in the same way as in CircleCI. Look at requirement 8 in Minimum Viable Product CircleCI.

Target team

Req 1: There is a clear indication of the software's state with colour indications. The pipeline status can be directly visible with a status badge in the readme file.

Req 2: GitLab supports a lot of programming languages, including Android and iOS. Note using available docker images decides support for the language. The available GitLab shared runners on GitLab.com, do not support an iOS or Windows environment. With GitLab runner, on-premises solution, iOS, Linux and Windows environments are supported. However, there is the possibility to configure to third party servers to run an iOS environment, like MacStadium.

Req 3: Integration to Slack and Jira is possible through the project settings by going into Settings -> Integrations, where you will find JIRA and Slack.

Req 4: This requirement is not fulfilled exactly in the same way as in CircleCI. Look at requirement 4 in Target team CircleCI.

Summary

The requirements for Absolute minimum are fulfilled, but MVP requirements are partly fulfilled 7,5/8. The requirement 6b is not filled in GitLab, but will in the future be implemented as it is a part of their issue board. GitLab fulfils 2/4 of the target team's requirements, where Requirements 2 and 4 are not fulfilled.

3.3.4 Reflections

The results from the testing can be found below, 3.1, and all the tested tools fulfiled the Absolute minimum requirements. The requirements of 1,2,7 and 8 MVP were difficult to evaluate, and were based on my experience with the tools. The intended users of CDaaSS will have similar experience and knowledge. These requirements are also the most central for the Minimum Viable Product. One requirement was more difficult to test than others, requirement 4 Target team. I instead read up on the possibility of fulfilling the requirement.

Table 3.1: A summary of the results, where X signifies a fulfilled requirement.

	CircleCI	**Jenkins**	**GitLab CI**
Abs. Minimum Requirements	X	X	X
MVP Requirement 1	X		X
Requirement 2	X		X
Requirement 3	X		X
Requirement 4	X	X	X
Requirement 5	X	X	X
Requirement 6a	X	X	X
Requirement 6b	X	X	
Requirement 7	X		X
Requirement 8	X		X
Target team Requirement 1	X	X	X
Requirement 2		X	
Requirement 3	X	X	X
Requirement 4		X	

Rule out Jenkins

Jenkins is a tool for customisations and require special attention. When used on-premise, it is dependable on your own resources. Based on my evaluation of the requirements 1, 2, 3, 7 and 8 in MVP, Jenkins is not a candidate to help facilitate CD as a Self-Service and can be ruled out. As Jenkins was found time-consuming and difficult, in the requirements 1 and 3.

Compare CircleCI & GitLab

GitLab and CircleCI are very similar as they are both cloud-based and fulfil almost similar requirements. Both are candidates for CDaaSS, as they fulfil all or almost all requirements from MVP CDaaSS. Regarding customisation, both have to use third party software for deployment and testing.

Updates

As CircleCI provides on demand updates, req 5 MVP CDaaSS, some users might want to wait to update, for different reasons like being in the middle of a sprint. However, users should not wait to long, as support for older versions can be made unavailable, like for CircleCI version 1.0 [8]. As new versions are released, there are more new things for the user to cope with. That is why incremental version changes, 1.0 to 1.1 to 1.2 etc, are recommended instead of bigger version jumps like from version 1.0 to 2.0. This will give the user a smoother transition through updates. In the other hand, GitLab has automatic updates which can require adaptions, if new functionality appear, req 5 MVP CDaaSS.

SCM integrations

The two cloud-based tools tested CircleCI and GitLab, are limited to their integrations to other SCM tools. However, you can import projects to the supported SCM tools and create pipelines for them. This might create a double maintenance problem, but one can configure two git repositories to be synchronised, with repository mirroring [14]. However, this needs to be configured and in which direction(s) the mirror should go between the repositories, pull or push.

Cost

CircleCI provides for private projects 1000 build minutes per month, 1 container and 1 concurrent job on a Linux container for free. One container consists of 2 CPU and 4GB RAM, with a limitation of 3GB limit on the uploaded artefacts. You can upgrade for a price based on your needs of containers and/or concurrent jobs, where each extra container costs 50$ per month. A MacOS container is not included for free. There is another kind of pricing plan for MacOS environments, where there are 3 kinds of subscriptions seed, startup and growth for 39$, 129$ and 249$ per month respectively.

GitLab provides for private projects 2000 CI minutes per month per group. The shared runners that run the CI/CD jobs run on n1-standard-1 instances on Google Cloud Platform. The instances have 1 virtual CPU, 3.75GB RAM and 25GB of HDD disk space. There are no shared runners for MacOS environments. There are 3 different pricing plans Bronze, Silver and Gold, at the prices of 4$, 19$ and 99$ per user per month.

CircleCI cost evaluation

From a cost perspective in order to use CircleCI as a Self-Service for your team, you would have to pay for a subscription, since the container provided for free are not enough and you don't have access to a MacOS container. Doing an estimation calculation on the build minutes, let's say you have 5 team members that check-in 10 times a day each, and where each build has a 3 minute duration. This would require at least 3000 build minutes, when counting 20 work days in a month, and more than one container without considering if concurrence jobs are needed. If you subscribe for at least two Linux containers, you are offered unlimited build minutes on these containers. You would have to pay at least 50$ for the Linux containers and 129$ for the MacOS containers per month, where 1800 build minutes are included on the Mac side. With this subscription plan, you will exceed the needed 3000 build minutes. This would amount to a total cost of 179$ per month.

GitLab cost evaluation

GitLab would also require a subscription plan to use GitLab as a Self-Service since the

build minutes are not enough. You need at least Bronze subscription for the team members plus an additional cost for a third party Mac environment, 20$ plus 69$ [27], based on the calculation of 5 team members. Bronze, which includes 2000 CI minutes, will be sufficient for the team if they need 3000 CI minutes per month, since the minutes are divided between the two environments. This would amount to a total cost of 89$ per month.

Chosen tool

The target team uses GitLab as their SCM tool, which is why it is the best choice for an implementation of CD as a Self-Service. If CircleCI had GitLab support, I would have recommended it instead of GitLab CI/CD, since it is faster, has a simpler UI and built-in MacOS support.

3.3.5 What is not sufficiently simple?

Below, things that were not sufficiently simple are listed:

- Manual initial setup.
- Indentation sensitive syntax.
- Integrating to third party services for deployment.
- Finding the right path in the containers.
- Optimising speed of builds.
- Uploading test reports and artefacts manually.
- Finding support for your language.
- Choosing the right subscription plan for your needs.

3.3.6 New Requirements

Below, are the new requirements listed:

- An automatic initial setup with built-in templates.
- Integrate to third party services, for testing and deploying, on demand.
- Non-reactive syntax for pipeline configuration code.
- Automated uploading of test reports and artefacts.
- Use images for supporting different platforms.
- Automated optimisation of builds.

3.4 Interviews with experts

In order to create a wider perspective on maintainability and usability regarding Continuous Delivery, two CD experts were interviewed. The questions asked were based on the initial requirements and the tested tools functionalities, and can be found in Appendix B. This was done in order to clarify already existing requirement and also to add new ones.

After having done the interviews, two perspectives to a future service are a part of our analysis, the users' and the experts'.

First, discussions regarding tools and requirements are presented. Second, reflections based on the interviews are presented.

3.4.1 Tools

Similar vs customised pipelines

The answers point to customised pipelines. However, there are similar aspects to the pipelines the CD-maintainers have worked with. The pipelines seem to be similar within a company. Also, the pipelines seem to have similar middle steps. The differences are how to get the code, where to store the artefacts and where to deploy. How to get the code and where to deploy, are usually directed to a number of tools depending on the project.

Most common CI/CD tools

The most common CI/CD tool is Jenkins. Other tools used that are connected to CI/CD, are Git for version control, Gerrit for code review and Artifactory for binary files. The main reasons behind Jenkins' popularity are habit, convenience, open-source and previous experience. The bigger companies have set a standard, and as the employees change jobs or quit, they bring their Jenkins experience with them to their new job. This has somehow set a standard for CI/CD tools. Another interesting factor is the cost perspective, and how the companies look at costs. As Jenkins is open-source, it is viewed at first sight as free. However, the costs of maintaining Jenkins, or other open-source tools, are built up over time, compared to cloud-based tools where the costs are direct.

Using Jenkins involves maintaining the hardware for running the instance, updating Jenkins and plugins, configurations for new updates if needed and external tools used for developing software [30].

Existing tool or create a new tool for CDaaS

The answers point to a new cloud application, but there are concerns to this approach with security and costs. Today, big companies have a tools team, that handles the "as a Service" part. On the other hand, smaller companies have more freedom to test other application like GitLab. The "as a Service" can be a set of tools connected together which are provided to the customer. One CD-maintainer pointed to a homogenous tool stack, because it is easier to integrate the tools together.

3.4.2 Requirements

Cloud-based tools

There are several reasons why cloud-based tools are not chosen as CI/CD tools. The reasons are: comfortability with other tools, previous experience with other tools, ownership, control, security, complexity and company culture.

Bigger companies want to control the details and with cloud-based tools you will have to give up a certain amount of control. In order to make a change in a bigger company to cloud, a decision must be taken from the company's management team. As for security, there is a mistrust to cloud-based tools and the following question is posed: how secure is it? Another question asked by a CD-maintainer, what is the difference in security between a self-configuration with for instance Microsoft Azure and with a "as a Service" solution?

One CD-maintainer pointed out that cloud brings complexity, when it comes to configuration and adaptation. Cloud-based tools can be compared to other cloud services in regards to security, like Google Photo.

Team's perspective

It is important that the service can be adapted to the team's line of work. One CD-maintainer pointed about that this service might be difficult to use for big companies, as changes are difficult to implement there and they have special needs. One example pointed out, there is a big company that is still using a two year old version of Jenkins. This is to point out that it takes time to make changes in big companies.

Other things the team have to think about other than the code, the tests and the pipeline configuration code, are the scaling of their needed resources and the costs and the time for these. As different teams have different requirements for resources such as CPU, storage and RAM, it is difficult to know how much you need and their costs. You would not want to waste resource by buying too much or too little.

Automatic vs on-demand updates

There were two different answers to this question. One CD-maintainer would prefer to work in an environment with automatic updates, but with the condition of getting notifications for new updates to be able to debug easier.

The other answer was mixed. It is the same thing as saying how often should we integrate with the main branch, which depends on the size of the changes, big vs small. The CD-maintainer pointed to bigger companies having tools teams and they can have automatic updates to use the new features, after a while they can send the updates forward to the team. Also, testers would prefer a baseline to work from and would not like constant changes. However, it would be best with incremental changes for bigger teams. Lastly, the CD-maintainer pointed to the importance of having release notes and traceability for the updates.

Speed

Faster builds are preferred, because you want feedback as fast as possible. Companies do not consider UI performance when choosing CI/CD tools. One CD-maintainer pointed that Jenkins requires a lot of effort to maintain, and that easier maintenance is possible with Groovy scripts. However, a lot of work is need for the Groovy scripts.

One CD-maintainer pointed that it is a matter of cost, to be able to build faster versus waiting and how long the waiting is. You would not like to be queued, like for example in GitLab. The most important thing is that you know that your build has started. You would rather have a slow build than a stack of queued builds, as it is more annoying. A slow

build can still provide feedback as the stages in the pipeline complete. UI performance is important for fast feedback.

Time for setup and education

This question has been the most difficult to answer and was left without a specific time. However, some factors were mentioned like it depends on the projects size, duration and company. Smaller companies can faster get started with new tools, unlike big companies. Developers would rather skip doing the setup or receiving education within CD. They want it to just work and to be able to send their code to the repository.

One CD-maintainer noted that it is important to have descriptive syntax, that is more readable compared to Groovy scripts. Factors to consider, are planning what the pipeline should consists of, where to deploy, how to get the code and an iterative implementation of the pipeline. It is also important that your pipeline consists of the same tasks you would manually do on your local environment, to avoid the dilemma of ”it works on my computer”.

Other

One CD-maintainer iterated the importance of adaptation to work. CDaaS will have difficulties to fulfil everybody's needs, and should niche towards certain development like web, embedded or application development, said the same CD-maintainer.

Big companies have difficulties to change, because of culture, management structure and existing solutions. One example put forward by one CD-maintainer, is that it took two years for a big company to switch from ClearCase to Git. It is difficult for bigger companies to change to a new tool since they have built so much around one tool. Smaller companies have more freedom to choose and change, as they have a smaller management structure.

One thing discussed with one CD-maintainer is that the developers have to learn the importance of CI/CD tools and to document their code better. When documentation is lacking, traceability is compromised. As a configuration manager you can not know every detail of the implementation, that is why documentation is important. Some companies have a test automation departments that may help solve this dilemma when the documentation is lacking.

3.4.3 Reflections

General

The interviews provided great feedback for new requirements. Requirements regarding updates, resources, syntax type of pipeline configuration code, integration to services and start time for the service, will be clarified. New requirements will be considered, such as cost function for needed resources, speed of UI for builds and way of working.

Pipelines

The pipelines used seem to have similar middle steps, that can be abstracted for use. The different aspects, such as where to deploy or where to store artefacts, can be solved in

the service by providing integrations to the most common tools in these area to cover the whole pipeline and more users.

Open-source vs cloud-based tools

Open-source tools seem free to users at first hand but as the use of these tools increase so does the maintenance cost. For open-source tools you can custom build your own plugins for your needs, but with cloud-based tools you are dependent on the service's implementation of features.

As the implementation of CDaaS point to a cloud solution, costs are important. In order to convince others of our solution, a cost calculation of having a cloud versus on-premises solution has to be done, see 4.3. We have to also adress factors for not choosing cloud-based tools, mentioned above.

Culture & Helping features

A willingness to try new tools is important in order for cloud-based tools to be successful. As well as the way of working has to involve CI/CD in the first sprint of development. Help to the team has to be available like resource management and descriptive syntax for the pipeline configuration code. Functions like auto-scaling or suggestions of choosing the appropriate resources based on the teams usage, are options to consider.

Functions

Automatic updates are the best option for this service, however with the conditions of having notifications and release notes for traceability. The question of starting time for the service will have to be changed to as fast as possible, since it is difficult to set a time. However, new factors to consider in the starting time has been found.

3.4.4 New requirements

Below, new requirements are listed.

- UI provides fast feedback for builds.
- Automated cost function bills based on usage.
- An iterative implementation of the pipeline.

3.5 Final requirements

The initial requirements, in 3.2.4, set after the assessment were worked with iteratively by testing tools 3.3 and interviewing experts 3.4. This resulted into final requirements, found in Appendix C, where new requirements were added and others were modified.

Chapter 4

Design

Now that we have identified the requirements for a future service, in the Analysis chapter 3, we are ready to address the design of a possible solution. This is done to solve how software development teams can be made CD "self-sufficient" with usability, maintainability and cost aspects in mind. As a starting point, to set the scene for the future service, functionalities are discussed on how they can fulfil the requirements. This done first to highlight the building blocks of the future service and used as a base when suggesting proposals for implementing the Minimum Viable Product.

In this chapter, first, functionalities of the service are presented. Second, proposals for implementing the Minimum Viable Product are discussed. Third, a cost-benefit analysis, of using a service versus using an on-premises solution, is presented. Lastly, a complete infrastructural view of CDaaSS is presented.

4.1 Functionalities

In the analysis, challenges regarding usability and maintainability were found when testing tools as well as concerns were brought forward with a future service when interviewing experts. That is why, this section discusses functionalities which can address the challenges and concerns found. These functionalities are also discussed since they will bring value to the service and be different from other solutions in regards of usability and maintenance. To create a clear image of the service and its outstanding features, functionalities are mapped against the requirements.

First, this section discusses how the Cloud can fulfil the requirements and addresses concerns with this approach. Second, it presents specially explored functionalities in Updates,

Customisation, GUI and Automation and how they can be realised. Third, it suggests a way of working for a service.

4.1.1 Cloud

In the analysis, testing of two Software as a Service tools and interviewing experts suggested CDaaSS to be cloud-based. An on-premises solution with Jenkins was excluded as it would require manual maintenance of infrastructure and was ruled out in the analysis due to being costly, time-consuming and difficult.

Cloud computing refers to resources and computing capabilities available to the customer through the Internet [20][p.312]. The concept of ”as a service” is part of cloud computing and is about delivering a service to the customer. Cloud pricing models often involve paying based on usage. This can mean paying for the minutes used, like in Google Kubernetes Engine, or it can mean paying a monthly fee, like in CircleCI. It all depends on the chosen abstraction level.

This section is composed of two parts, Requirements and Addressing concerns. The first part discusses how the requirements can be fulfilled with the Cloud and the second addresses concerns of adopting cloud-based tools.

Requirements

To be able to meet essential requirements of having the resources, the technical expertise and costs to work with CD, this part discusses resources, technical knowledge and costs from a cloud perspective.

Resources [MVP requirement 3]

The cloud can fulfil MVP requirement 3a for the service, as resources, like storage, hardware and servers can be provided on demand. This way physical infrastructure is rented over the Internet and the amount of resources needed for the service can at any time be scaled up or down. This will give the service scalability capabilities, as well as manage specific peak loads on-demand, since it is the cloud provider’s responsibility [20][p.314]. The service should be connected to at least two cloud providers, to spare it from outages, if a cloud providers services are down.

The cloud provider takes care of capacity and availability for the service, to help fulfil requirement 3b MVP, and resources will be available at all times. The networking issues that may arise with having the resources available at all times, are handled by the cloud provider as well as the capacity of the servers [20][p.314].

Technical knowledge [MVP requirement 1 & 2]

To ease use of CDaaSS, a cloud service is connected to it. This way requirement 1 MVP can be fulfilled, as a cloud service will not require high technological knowledge. You can use the cloud service with all its benefits, such as scaling and provisioning, without having to manage the complexity of it [20][p.314]. This also applies to the maintainability of the

infrastructure of the cloud service [5]. Relating to requirement 2 MVP, cloud services can help the team only be concerned with the pipeline configuration code, since networking and infrastructure would be an automatic procedure compared to the manual procedure required on-premises.

Costs [MVP requirement 7]

For the service to be of a lower cost than having a maintainer in the team, requirement 7 MVP, with a cloud service you only pay based on your usage. In infrastructure services, like AWS or Google Cloud, payment is done by credit card, which means payments can be postponed until the service has started to receive an income [20][p.312]. This means that there is no need to invest any capital into infrastructure [20][p.312]. Time will also be saved by not having to set up the infrastructure, which means minimising startup costs and in turn increasing agility [5].

Less time will be spent on managing and monitoring underlaying infrastructure, which means IT staff can focus on other issues. Additionally, electricity costs are saved on hosting servers. These arguments applies especially for bigger companies [5]. From a cost perspective, there is not a worry about over or under provisioning, since the service has scalability capabilities. This will in turn make sure that the service does not fall short of resources for customers or lose capital if it is unpopular [11].

When using a cloud service at the infrastructure level, you pay based on use, comparable to how electricity is billed [20][p.312]. However, using a service like CircleCI, you pay for a subscription, which can mean paying for extra features that maybe are not used. That is why it is important to choose services that can be tailored to our needs, where we can choose our features. Another concern is paying minutes for an out of control process in vain. This concern can be address by setting max build minutes for different steps, like in Google Cloud [15].

Addressing concerns

The concerns pointed out from the experts and the literature study with cloud, are addressed here to convince of a cloud solution for the service. This part discusses security, vendor lock-in, performance and control.

Security [MVP requirement 4]

One of the concerns found when discussing with experts is security. Storing your data in the cloud at a third party place, raises security concerns for companies wanting to adopt a cloud solution. Companies are concerned about their technology being compromised or their data being stolen [20][p.313]. One important thing to bring up, is the cloud services themselves use their own technology. Examples are Amazon Web Services and Google Cloud Platform, which can serve as a proof of concept. With an on-premises solution companies are assured that no one can look into their data and listen to their traffic as they use their own infrastructure. On the other hand, data stored in Google Cloud is encrypted by default using AES-256 [16] and the communication is also encrypted using HTTPS [17].

Therefore, there should not then be any concerns to manage secrets in the service as it makes use of cloud services, requirement 4 MVP.

Vendor lock-in

Another thing raised by the experts considering to migrate to a cloud service, is customisation. As you integrate more and more with a cloud service and use their infrastructure, your work will be dependent on that service. This raises issues like custom features. As there are no standards in cloud, you will have a difficult time if you would like to change to another service. This leads to some sort of lock-in [20][p.316]. On the other hand, if you have an on-premises solution you will be locked-in to your own infrastructure.

Performance [New requirement 7 & MVP requirement 3b]

To help fulfil fast feedback for builds and no downtime, new requirement 7 and MVP requirement 3b, performance is discussed. As you solely depend on the service to provide you with the necessary infrastructure, the required performance has to be delivered from the cloud service. This requires you to do an assessment of your needs [20][p.314]. Today however, the performance of hardware has greatly improved. A way to increase performance is to use parallelisation. It all depends on the cloud services and what kind of performance they can deliver. In general, cloud services can deliver performance at least as good as the performance of on-premises if not better and on-demand. This due to technology advancements and the services being used by the providers as well. Information on the performance that can be delivered by Google Cloud, can be found here [18].

Control

As you start using a cloud service, you will lose control over the infrastructure. This is something that scares the companies, since they will be dependent on the cloud providing company. As long as the service makes sure the work is done, there should not be any worries. If the cloud providing company goes out of business, customers are worried what might happen to their infrastructure. However, it is highly unlikely that big companies, like Amazon or Google, to go out of business, as they are the foundation of what defines cloud computing. On the other hand, there is a probability that a new and small service like CircleCI to go out of business, but migrating to a similar service will not be difficult as the configurations can be reused and the setup time will be the same. A crashing on-premises solution will require more effort as the setup and infrastructure provisioning will have to be redone.

4.1.2 Updates

One of the explored functionalitites in the analysis, was automatic updates versus on-demand, for MVP requirement 5.

Automatic updates for the service is the optimal solution, with incremental changes, since the user would have one less thing to think about and not risk falling behind on updates and new functionalities. This way the user will have a smooth experience as few functionalities have to be coped with in comparison with bigger version jumps, like 1.0 to 2.0. On-demand

updates, on the other hand, would require the user to specify the version and know when a new version has been released. Waiting or forgetting to update can be problemtic, as support for older versions can be made unavailable.

To ease usability and maintainability aspects, notifications are provided for the automatic updates with accompanying release notes for traceability. In the pipeline view, stages that can be affected by new updates can be highlighted, if adaptions need to be done.

4.1.3 Customisation & Integrations

For the service to be used by a broader audience and fulfil requirements target team 2 and 4 and new requirement 2, customisations and integrations must be provided.

First, customisation can be provided by docker images categorised for different types of development, app, web and mobile, as docker images easily collect different tools for use. This way the customer can be provided tailored environments.

Integrations can be provided on-demand, to avoid complex configuration setups, when creating the pipeline. This function can be realised by CDaaSS, by providing integrations to the most popular services to meet customers different needs. When using this function, the user should only be prompted for a login to the third party service, to establish communication. The Self-Service handles the necessary exchange of information, such as an API key and the integration commands.

4.1.4 GUI

The GUI will ease the initial setup and not require expertise when creating the pipeline, MVP requirement 1. With a GUI implementation of the pipeline, the user does not have to worry about syntax errors, as the GUI produces the correct configuration code to run the pipeline. Additionally, the GUI guides the user with implementing the pipeline, which makes sure the different stages are included. This way the user is familiarised with the different steps in the pipeline as well as the options which can be added in the future. The overview provided by the GUI will also simplify for other users to acquaint themselves with the pipeline.

Chen found that giving developers a visual CD pipeline skeleton will help teams to adopt CD, where stages are visible but without an implementation of them. This way the developer are reminded of unimplemented parts of the pipeline and will give them an incentive to implement them [7].

To help make an iterative implementation which can be tested, new requirement 9, the implementation is made possible to do step by step for the different stages. With this approach the feedback on the implementation is given incrementally, as whole pipelines take long time to run.

4.1.5 Automation

To facilitate a Self-Service, manual tasks can be automated which were found not sufficiently simple. These tasks are initial setup, cost function, optimising builds and uploading artefacts, new requirements 1, 4, 6 and 8. Below, possible implementations are suggested on how they can be automated.

As a part of automating the initial setup, in the beginning of implementation, an SCM connection can be automated by only providing a login. After having established a connection with the SCM, the projects in the SCM are displayed and the project to be set up is chosen. When the implementation has started, the user is provided templates for the most popular environments, since many users have the same needs.

To avoid the dilemma of choosing the right subscription plan, an automated cost function is suggested. The functionality makes use of the information stored in the SCM, to set up old projects. To make an estimation of the build minutes to choose a subscription plan, the users can be known from the SCM as well as how often the project is checked into. The service then test runs a build, and an estimation is done based on users, check-ins and build duration and a subscription is chosen. For new projects, the parameters mentioned can be filled in.

To enable faster builds automatically, static parts of the pipeline can be analysed after a build has run. The optimisation can be automatised by storing the static parts and save the them for every build. This functionality can suggest to the user if parts of the build seems slow and on-demand the user can optimise the pipeline.

To automated tasks and ease the use of the service, artefacts such as the application and the test reports are automatically uploaded. The artefacts can be uploaded automatically, by having a function which locates them making use of their reappearing file extension. This can help the user avoid complications to search the different paths to find the test reports. If there are several test reports in the repository, the user can be given the option to choose which ones to upload.

The functionalities suggested above to automate tasks can be difficult to implement, since different kinds of projects have different environments. However, today three of these exist to do manually by the user. The user can be prompted about the kind of project, then the service makes use of predefined information about these to automate the tasks. The cost function should not be too difficult to implement as a simple computation has to be made. The difficult lies in extracting the relevant information from the SCM.

4.1.6 Way of working

To decide to work with a service needs planning. I propose a methodology to follow in order to have a Continuous Delivery system as a Self-Service in your team and which can help fulfil new requirement 9.

What are your needs? What is the life-time of the project?

In the first step it is important to know what you need and what is important for your pipeline. Another crucial factor is the project duration for the project you are going to work with, which will help you set a guideline on time to implement a pipeline. An important thing to consider is getting lost in the chosen third party tool's features, that is why you should know your needs from the beginning.

What is your budget? Do a preliminary cost calculation for an on-premises versus as a service solution.

When choosing an approach to introduce Continuous Delivery in your software development team, you will have to take cost in to consideration and what kind of resources you have available. The cost factors mentioned, in 4.3, can be used when doing your cost calculation. However, if you have not before had a CD system in place in your team or company, you will a hard time doing an exact calculation, since you do not have a cost history for implementing CD in your environment. My advice would then be to choose the as a service solution, since you can almost immediately start implementing CD. Another important factor is if your company already has an existing IT-infrastructure. This can be used in your cost calculation. There is an emphasis that an exact cost calculation is very difficult.

Experiment with existing tools

If you have chosen the as a Service solution, choose a variation of Software as a Service, SaaS, tools for your team to test for a day making it a team activity. Today there are a lot of tools to choose from in SaaS area. Then based on your teams responses of the tools, choose one you that can meet your demands for support of platform and hardware capabilities. Switching to a SaaS tool will require days, as infrastructure and platform is provided, you have to only adjust the service to your projects.

Get started with the tool

Start building your pipeline to your needs, as you have chosen a SaaS which is easy to start with. The beginning it is important the whole team knows how the pipeline is constructed and what its different stages contain. Implementing a pipeline into legacy systems will be difficult, since adaptions have to be made to a new way of working and releasing. That is why CD implementation is advised to start in the beginning of your development, already in the first sprint. Also, to get the know the workflows and infrastructure of CD, simple applications, like Hello World, are advised for developers to begin with [31].

After implementation

After introducing CD in your team with an implemented pipeline. It is important to keep your pipeline alive as new functionalities coming from the service and changes in your system will require maintenance. For instance, there could be a change on how your application is built or a function in the service can be disabled. I suggest making CD tasks a part of your development like any development. Sharing of the CD knowledge and of the pipeline is essential and can be achieved by monthly meetings updating the team of the progression. The CD tasks are encouraged to be done in pairs as a way to spread the knowledge. This will result in a collective responsibility.

Improvements

For continuous improvements in the area of CD, an employee, in a big enough company, can go to workshops and courses on the latest within the area. This employee can then share the new knowledge to the team, to keep them up-to-date with the latest in CD and tools.

4.2 Minimum Viable Product

The functionalities of the Minimum Viable Product were discussed in 4.1 and now to realise the product and facilitate Continuous Delivery as a Self-Service, proposals for implementation are discussed. The purpose of implementing a Minimum Viable Product is to show initial value and to get feedback for features that would help developers with the implementation of CD.

There are several ways to implement a service for CD and to reduce maintenance. Below proposals in which environment the service can be implemented are presented and discussed. Three proposals are discussed in this section of how an implementation can be made and delivered to the user, which can adhere to the requirements set. The proposals are Jenkins plugin, Integrations Software as a Service, Software as a Service.

4.2.1 Jenkins plugin

Due to the nature of Jenkins being customisable, an implementation of a service is possible with a plugin. When implementing this plugin, it is possible to try to match it to the requirements.

In this implementation connections to cloud services can be made in the way drawn out in the previous section functionalitites 4.1. However, this can be problematic as changes of functionalities in Jenkins, is something out of the control for the implementer. This means adaptions have to be made to the constant changes of Jenkins, in order to keep compatibility between the plugin and Jenkins. Additional, the initial setup will still require an effort, described in the analysis chapter. That is why this alternative is ruled out.

4.2.2 Integrations SaaS

To try to match the functionalities and the requirements, it is possible to integrate a set of tools into a service. The implementer can provide the service as a set of integrated CD tools. This service involves all the essential building blocks of CD, building, testing, integrating and delivery. The implementer makes sure the connections and compatibility between the tools are working, with the infrastructure and platforms. Any maintenance involving the tools is handled by the service. This implementation will require a lot of reliance and maintenance, as the tools a part of the service decides the features. This implementation will require handling a lot of connections, that is why this implementation is ruled out.

4.2.3 SaaS

In this approach the frontend facing the user is implemented, i.e. the CD tool for building, testing, integrating and delivery. This way the implementer is in charge and can create the necessary features of a Self-Service. The implementation is a web-like application available over the Internet, to provide easy access to users. However, the underlaying infrastructure can be setup to other services for storage and hardware to make use of the cloud benefits highlighted in the previous section 4.1. This approach is similar to already existing services like GitLab CI or CircleCI, however abstracted and adjusted for use for developers. This is the best solution as the implementer will be in charge of the front-end and will be easier to match the usability and maintenance requirements.

4.3 Cost-Benefit Analysis

To highlight the cost perspective and how a service in the area of CD is beneficial, a cost-benefit analysis discusses costs and benefits. Also, the analysis is done to address how a service can lower costs, requirement 7 MVP. This section highlights cost factors, benefits and disadvantages and discusses costs from a small teams perspective.

Cost factors for having CD in the cloud versus on-premises will be listed as well as the cost factors of not having CD at all in a team. In addition, benefits and disadvantages are listed about these approaches.

Cost factors for CD in the cloud, CDaaSS:

- Direct costs for SaaS, pay-as-you-go model [5].
- Pay based on usage for Infrastructure or Platform as a Service [20][p.312].
- No or low maintenance cost [5].
- Saving on electricity costs, especially true for bigger companies [5].

Benefits of CD in the cloud:

- No time wasted on coordination issues.
- No need for buying in technology expertise for the team [11].
- Renting physical infrastructure is cheaper than buying in short term.
- No need for capital investments into infrastructure [20][p.312].
- Scalability ensures not missing on potential customers [11].

Cost factors for CD on-premises:

- The cost of manual setup of the infrastructure (configurations, installations of software) [5].
- Maintenance of physical infrastructure (reparations for hardware, buying in new hardware) [5].
- Cost of CD maintainer(s).
- Running costs of CD in terms of electricity [5].
- Renting space needed for the infrastructure.

Disadvantages with CD on-premises:

- Reliance on expertise.
- Scaling needs an effort.
- You do not know the actual cost of CD maintenance, until later.

Cost factors for Software development without CD:

- Cost of manual configuration management [20][p.19].
- Cost of mistakes are more expensive and it is bound to happen with manual procedures [20][p.19].
- Cost of waiting on builds.
- Costs of not continuously doing QA (lower quality).
- Coordination within the team.
- Reliance.

Humble mentions the following factors when comparing cloud computing vs on-premises [20][p.317]:

- Break-even point of the two models
- Depreciation
- Disaster recovery
- Support
- Not spending capital

When having an on-premises solution, you either have one or more persons in the team maintaining CD or external maintainers. You can make a comparison of the costs of having an on-premises solution by adding up all the costs mentioned and dividing by the users of the CD system. The cost per user can then be compared to the cost per user at a SaaS tool, like CircleCI. This can help make an assessment for the right choice.

To sum up, a general cost-benefit analysis approach is presented, for choosing between a cloud solution or on-premises. First, take into account the team size and budget for the team, which can help know what is suitable and reasonable to have. Second, consider the setup time and the infrastructure and platform for the projects the team works on, to know where support can be found for them. Third, take a look at the expertise within the team and its changing environment, to know what is available in resources and what might change in the team. Fourth, analyse if scaling efforts are needed in the future, to have a picture if changes will occur to the system.

For a small software development team

Using the general approach above, for a small team of five, where time for CD is limited and should be setup as fast as possible. In addition, support can be found for the platforms the team works on in SaaS tools and the needed build minutes of 3000 is available. Since it is a small team with a changing environment where expertise is not available and the system can change, a cloud solution is the best choice, from a cost-benefit perspective.

In the analysis chapter, costs were found to be 179$ for CircleCI and 89$ for GitLab CI per month. In our discussion, we use CircleCI as an example as it has an integrated MacOS

environment and use the more expensive service to prove the benefits of using one. In a real-life example when using services, a cost of 250$ per month can used as an approximation, to take into account external integrated services for testing and deploying in a special environment. This is the total cost of using as a service solution in a small team.

The average pay of a software developer in Sweden is 43100 SEK per month [1], resulting into a day's wage of 2155 SEK when counting 20 work days. This cost is comparable to the monthly cost of using as a Service solution, where 250$ roughly equals 2250 SEK when counting 1$ as 9 SEK.

4.4 Infrastructural components of CDaaSS

To break down the complexity and facilitate a full future implementation of CDaaSS, a mind-map is presented in figure 4.1 on the next page. This mind-map can also serve as a summary of the studied components in this thesis. The leafs in the mind-map are the studied components and the sub-leafs areas suggested for a Self-Service. All the sub-leafs have been addressed and motivated for in the previous sections and chapters.

The purpose and starting point for this section was to answer what kind of infrastructure is needed for implementing CD as a service. When the work began, a list started to develop, however connections between items could not be visible. Therefore, a mind-map was chosen as a way to highlight different parts needed and their connections to each other. This mind-map has helped me reflect on my own work and summarise the thesis. The initial bubble was Continuous Delivery as a Self-Service, where the areas researched were usability, maintenance, cost, infrastructure and integrations.

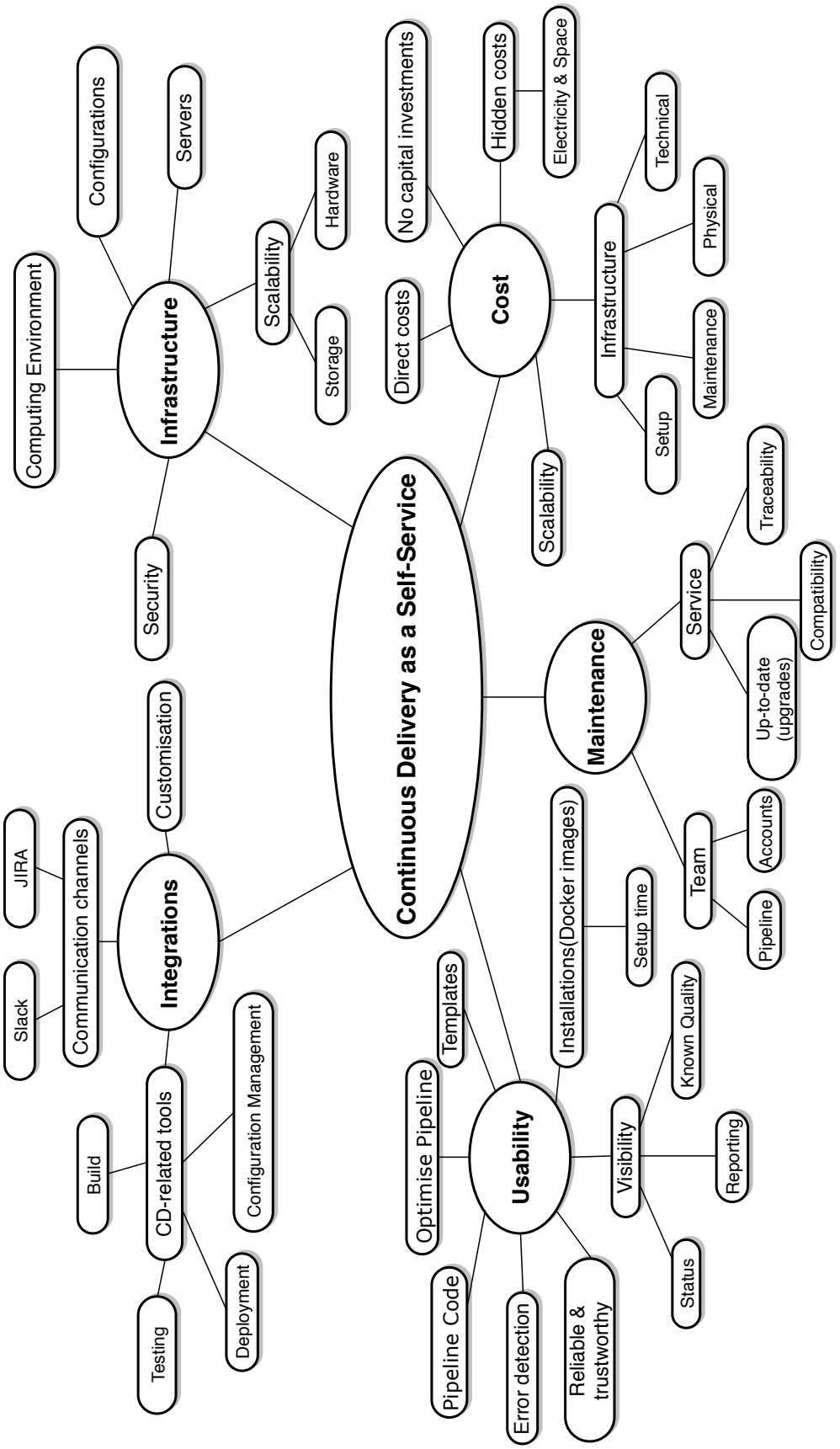

Figure 4.1: A mind-map of the infrastructural components of Continuous Delivery as a Self-Service

Chapter 5

Discussion & related work

After having done a thorough analysis of CD as a Self-Service, in chapter 3, and suggested a possible design for such a Self-Service, in chapter 4, it is time to reflect over the outcomes from this thesis. In order to validate the results and in particular the design, testing with the team was done with services found eligible in the analysis. Furthermore, to put this thesis into context, results from related research papers are discussed. To present the challenges and possible threats to the claims made in this thesis, a discussion about limitations on how general the results are is made. Lastly, it is discussed how the results from this thesis could be generalised, extended and applicable in a bigger context, suggesting possible future work.

In this chapter, first, a general discussion about the outcomes is presented. Second, validation of results is discussed. Third, related work is presented. Fourth, Reflections and limitations are discussed. Lastly, future work is discussed.

5.1 General discussion

After having conducted the research, looking back and reflections about our contributions are necessary to evaluate our process. This section is dedicated to discussing the thesis in general to reflect over the outcomes, the goals and the expectations.

In this section, first, outcomes are presented and, second, goals and expectations.

5.1.1 Outcomes

This part discusses how the outcomes can help software development teams with CD and how they can be used in the future.

After having done the assessment, difficulties with using CD in a small team were found. It was found that today's tools require maintenance and time the developers do not have as well as expertise. This has resulted into the team disabling functionalities and finding ad-hoc solutions to their quality assurance needs. The assessment process developed in this thesis can be reused for finding challenges in other teams.

In the researched context, a requirement specification of how a team can be made CD self-sufficient was made. A part of developing the specification, a lot of feedback was given on the studied tools. This feedback can help other teams when deciding on which tool to use for CD, by looking into the advantages and disadvantages listed. By addressing the concerns and the benefits of a service solution, we have presented motivations for other teams to use this approach. The cost analysis done for a small team, can also be used when choosing between approaches for implementing CD.

The design of the Self-Service, is a contribution which can be used for a future implementation. In this design, we have addressed how the requirements can be implemented and concerns with using a cloud solution. Our cost-analysis, argues how a service solution is cost-effective as well as listing cost factors, which can be used for other teams. The presented cost factors and benefits can help other companies make an assessment for the right approach for them.

The evaluation done when testing with the team, provided feedback on our design and on the tools from a user's perspective. This feedback is valuable for the team, since they have now been introduced to Software as a Service tools within CD and know what they can expected from a service.

5.1.2 Goals & Expectations

Initially, in the work process a goal document was written specifying our research problem, research methodology, expected contributions and project plan. That is why this section discusses goals and expectations.

The initial goal of this thesis was to research and define a service which can help teams become self-sufficient in the area of CD. This goal was achieved in our context, by presenting a Minimum Viable Product and a possible design of a Self-Service.

This thesis has presented a better understanding of the concept of as a Service within CD. Starting out, there was not a clear understanding of the concept and the issues of CD regarding usability and maintenance. We now know why CD at times are not properly maintained and how we can address these issues. The user's perspective to CD and a service has been presented and is now clearer, which was one of the expected contributions.

The research questions posed in the goal document have been answered, as we now know

what kind of infrastructure is needed for CD as a Self-Service and how we can cut initial costs of implementing CD. The second research question how this service can provide known quality has been answered by the feedback given, when testing the tools.

In the goal document, we expected to do a cost analysis of the service in order to cut initial costs of implementing CD by comparing with CD on-premises. This was done in our context however this turned out to be more difficult than expected, as the costs of an on-premises solution was unknown and cost approximations had to be done for a service solution. Instead, we presented more general cost factors to take into consideration when choosing an approach to CD and a cost-analysis based on a small team with a service.

5.2 Validation of results

The goals of the testing were to validate the results and get domain knowledge and feedback by the intended users of the future service. Each member of the team was handed a project to build a pipeline for, with a problem formulation and a survey to answer after the testing, both can be found in Appendix D. The testing was time limited to 2h, to show initial value and was found as an appropriate time to get feedback. The tools tested are GitLab CI and CircleCI, which were found eligible for setting up as a service in the analysis. The testers of the tools were 4 developers and 1 agile coach.

In this section, first, the validation is discussed and, second, the feedback is discussed.

5.2.1 Validation

This part discusses how the results were validated when testing with the team. All scores mentioned below are the average scores on the questions from the survey.

The testing validated that with the use of a service and without having to set up an infrastructure, 3 out of 5 testers can setup a pipeline within a reasonable time, 2h. The experienced difficult by the testers scored 2.6/4 and how fast the builds were experienced scored 3/4. The tools were found easy to get started with and graphically pleasing as well as with a lot of examples. How helpful the error tracing was in the tools scored 3/4 and the visibility of the status scored 3.4/4. Some of the things mentioned by the testers as being reliable with the tools were fast builds, fast to get started and the status of the builds. When the testers were asked if they could imagine maintaining the pipeline set up based on the experienced difficulty, 3 out of 5 answered that they could.

The results from the testing confirm the findings in the analysis, as the testers were able to set up a project that was used in the analysis. The fact that 3 out of 4 developers finished the testing in the given time and that they would consider maintaining the pipeline set up based on the experienced difficult, confirm the results of the tools being eligible for developers to use. The answers and scores in the survey confirm the results on how the tools match to the requirements, in regards to difficulty, build speed, easy to get started, visibility and error tracing.

5.2.2 Feedback

This part discusses the feedback given by the testers on the tools, which are things that can be considered in a future design of the service. The feedback given can also be used for setting new requirements and changing others.

When the testers were asked what could be improved in the tools, GitLab was found having slow builds, documentation could be improved and CircleCI should not need a setup with git to work. The things that the tools lacked in were better printout when tests fail, an easier get started step and speed. The least reliable things mentioned were error tracing in CircleCI, knowing if your languages are supported and the time for a build in GitLab CI. To make it easier for the team to maintain one tool, knowing if all the languages used are supported is important.

The domain knowledge acquired suggests users care about speed of builds, fast feedback, error tracing, support for languages, an easy get started step and tool should not need setup with git to work. The building in GitLab CI was one thing mentioned by both testers of the tool, as the time for a build was found unreliable and at times the build was stopped and jobs were cancelled, resulting in a slow feedback. For CircleCI, on the other hand, a faster get started step is suggested and not needing a setup with git.

The speed of builds in GitLab was also found slow in the analysis, and one tester pointed to be willing to maintain the pipeline given that a faster runner can be setup. The error tracing in CircleCI can be made easier by uploading test reports and is one function suggested to be automated in the design.

5.3 Related work

To put this thesis in comparison to other work and complement the results, this section discusses related work. Each paper presented has three parts: summary, critique and relation to thesis. This division is made to clarify the contents of the paper and the discussion about it. The two papers presented, where chosen based on the first suggesting a similar solution and the second trying to address the same problem domain. There were other papers that were considered, since they addressed sub-areas of this thesis. These papers were about the same context [35], scaling and cloud [4], studying students using CD [10] and challenges of CD [9].

5.3.1 ”End to End Automation On Cloud with Build Pipeline: The case for DevOps in Insurance Industry” [33]

Summary

This paper attempts to tackle the constant changes appearing in the application development related to the insurance industry. The need of faster time to market as well as new ways of customer interaction, are reason for looking into application lifecycle management practices. The use of Cloud computing has given several benefits such as agility, scalability and lower capital costs. That is why DevOps and Cloud computing is combined in this paper. The end goal of this paper is to create a proof of concept for an effective Continuous Delivery system making use of Cloud computing.

This paper contributes with a proof of concept implementation to tackle the challenges faced in the insurance industry. With the use of a DevOps culture and Cloud computing, benefits are highlighted with this approach. This paper also contributes with the chosen tools for this end to end automation as well as the design of it.

The method adopted is an implementation of a build pipeline in a cloud environment. This implementation is used as a proof of concept to tackle current challenges visible in the insurance industry relating to application development. The technology stack consists mainly of open source tools to achieve the end to end automation. There are several use cases listed a part of the proof of concept and to mention a few: integration of CI server, run pipeline for code changes, application assessment for Cloud environment, infrastructure provisioning in Cloud environment. A Cloud Acceleration Program was used to make the best use of Cloud computing.

The end result is an implemented build pipeline showcasing benefits to using a cloud environment in a DevOps setting. Some of the benefits listed are: end to end automation using open source tools for a deployment pipeline, automated infrastructure provisioning in Cloud environment, high availability of resources.

Critique

This paper lists several tools for the different areas in this implementation. However there is not a motivation behind the chosen tools and why open-source tools were chosen.

The benefits of the implementation are listed however there is not any mention on how they were verified or identified. The software environment where this implementation occurred and tested is not mentioned. That is why the results can not be applied in a general context.

Relation to Thesis

This paper is related with the thesis as both have investigated Continuous Delivery and Cloud computing. The benefits of Cloud computing are highlighted and can be an argumentation for the future service.

This paper confirms the results of this thesis as a cloud solution can be beneficial. Also, the implementation of combining the cloud and CD, extends the results as an implementation has been done. However, this thesis has results not covered in this paper, as motivations for the benefits are provided and a validation of a cloud solution with a team has been done.

5.3.2 ”Designing a Next-Generation Continuous Software Delivery System: Concepts and Architecture” [34]

Summary

This paper addresses two challenges faced in continuous delivery systems. These challenges have been highlighted in literature before, when adopting Continuous Delivery. The first challenge concerns the development of a software delivery system over time as requirements change in the system. Architectural changes or new testing stages are a part of the first challenge, that is why systems need to be flexible and maintainable. The other challenge addresses the usability aspect of creating a delivery system, where users today need to possess specific technical tool knowledge.

This paper contributes with its design of a new Continuous Delivery system tackling the issues of flexibility, maintainability and usability. A domain model is presented where the delivery process is central. The delivery process is described as a collection of stages which are composed of activities. These activities are described and classified into three categories: Transformation, Assessment and Quality Gate. This meta-model is then used to define a micro-service architectural software system.

The method used is to identify challenges and create a domain model, where the delivery process is central. This abstraction is then used and broke down in a micro service architectural design. The design has been implemented with JARVIS and validated in a case study. However, there are no presentation of this case study.

This paper presents a domain model and design of software delivery system tackling the mentioned challenges.

Critique

The paper does not address in which software developing environment the design can be used. Therefore the reader can not assume the design can be used in all settings. The

validation of the prototype is mentioned once and would have been of use for better understanding.

Relation to Thesis

This paper is related to the thesis as it has tried to address similar challenges, of maintainability and usability. However, the difference lies in the presented results as this thesis does not present a domain model and a detailed micro-service architectural design of a software delivery system.

This paper confirms the motivation of this thesis, as the need of a flexible and maintainable system and addressing usability with needing specific tool knowledge. However, the results of this paper provides a design of a solution different than in this thesis. The design from the paper creates a domain model from a general perspective, whereas this thesis addresses usability and maintainability and cost aspects with a cloud solution for a small team. Also, a validation of the design has given in this thesis.

5.4 Reflections & limitations

To look back on the challenges encountered during the work process, a discussion about the obstacles and limitations that formed this thesis is presented.

This thesis was first intended for two persons and the goal document was based on that. However, a sudden change resulted into one person conducting this thesis, which meant ambitions had to be scaled down to one person. The task of scaling down was difficult, as the research had to be limited to fewer tool tests and interviews to finish the thesis in the given time period.

In first step of this thesis, an extensive assessment was done with 47 questions. Looking back, a more concentrated assessment on the problem domain would have been better. As some of the results, pointed to other areas like branching strategy, development methods, customer relationships and testing, where some of these areas have already been researched in a previous thesis [35] and some are outside the problem domain. However, it is difficult to predict the results beforehand.

There was a challenge of scheduling time fitting other persons work schedule, as this kind of research conducted relied upon participation when interviewing and testing. Some appointments were postponed a few times, which meant the work process was slowed down. The testing with the team was an activity that was postponed more than once. However, the time was instead used on analysing and reading literature and preparing for the next step.

One thing that should have been done was assessing at least one more team, as the results could have been applicable in a wider context. Initially, three software development teams were supposed to be researched. However, one did not have a CD system and few participated in the initial survey from the second team. As the work process started to accelerate

after having done the assessment with the target team, assessing other teams was moved past. A solution to create a wider perspective was instead made by interviewing experts.

The threats to validity of the outcomes are that this thesis was conducted in a specific context and in a small team. The requirements set were based on this small team. In the analysis, three tools were tested and the things found which were not sufficiently simple might be simple in other tools. As this is a new area, no previous and similar work was found which could have helped the thesis. When assessing, testing and interviewing the threats to validity are presented as loosing focus of the main problem domain and the results showing no interest. The validation conducted with the team was more of a proof-of-concept testing and not on projects the team works on.

5.5 Future work

Now that the work is done, it is time to reflect on how extensions to work and results in this thesis can be made to help software development teams with CD.

Future work can look into different kinds of software development teams and see how the requirements set match to other teams. For instance, a different size of a team involved in a different kind of development can be researched.

To facilitate and further develop a Self-Service, an implementation of the Minimum Viable Product designed can be realised and compared to other tools that were not tested as well as letting a software development team test the product. This way the user's perspective to CD can be further developed.

To find new challenges and exact costs of implementing CD, an evaluation can be made with a team that implements one of their own projects. During the work process, costs can be assessed and after a period of time an exact cost-analysis can be made. This can help teams choose the right approach for CD in a cost-effective way.

Chapter 6

Conclusions

The tools used in Continuous Delivery require expertise to handle, which results into a reliance on outside actors in a team. Without the expertise, the CD system deteriorates since maintenance, infrastructure, integrations and compatibility between the tools need to be handled. This leads to developers seeking ad-hoc solutions to their CD needs by other means and disabling functionalities of the current CD system, to prioritise development. That is why one CD system adjusted to the user is preferred.

In order to address the need of expertise, a requirement specification is presented for our context, taking into consideration usability, maintenance, infrastructure and cost aspects from a user's perspective. A proof-of-concept design was made on how the requirement specification can be implemented where the cloud is the central component, since the infrastructure will be maintained by the provider and available on-demand. This together with the mind-map of components of a future Self-Service presented answer what kind of infrastructure is needed for a service. The cost-analysis for a small team and the cost aspects presented, answer how a cloud solution can cut the initial costs of implementing and maintaining CD. It was found that the cost of using a service per month is comparable to a day's wage of a developer.

The design was validated by the team testing Software as a Service tools and 3 out of 4 developers answered that they were willing to maintain the pipeline set up based on the difficulty of the tools. The feedback provided from the testing answers how this service can known quality, by easy error tracing, fast feedback, and reliable build time.

Bibliography

[1] Lönesök – hur mycket tjänar...? https://www.scb.se/hitta-statistik/sverige-i-siffror/lonesok/Search/?lon=utvecklare. Accessed: 2019-01-23.

[2] Sami Alajrami. Circleci vs google cloud build. https://www.praqma.com/stories/circle-ci-google-cloud-build/. Accessed: 2019-01-16.

[3] Atlassian. Gitflow workflow. https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow. Accessed: 2019-01-15.

[4] Cornel Barna, Hamzeh Khazaei, Marios Fokaefs, and Marin Litoiu. Delivering elastic containerized cloud applications to enable devops. *IEEE/ACM 12th International Symposium*, May 2017.

[5] Muzafar Ahmad Bhat, Razeef Mohd Shah, Bashir Ahmad, and Inayat Rasool Bhat. Cloud computing: A solution to information support systems (iss). *International Journal of Computer Applications, Volume 11– No.5*, December 2010.

[6] Lianping Chen. Continuous delivery: Huge benefits, but challenges too. *IEEE Software*, 2015.

[7] Lianping Chen. Continuous delivery: Overcoming adoption challenges. *The Journal of Systems and Software*, 2017.

[8] CircleCI. We're sunsetting circleci 1.0: August 31, 2018 is the final day for 1.0 builds. https://circleci.com/blog/sunsetting-1-0/. Accessed: 2019-01-17.

[9] Gerry Gerard Claps, Richard Berntsson Svensson, and Aybüke Auru. On the journey to continuous deployment: Technical and social challenges along the way. *Information and Software Technology*, 2015.

[10] Brian P. Eddy et al. A pilot study on introducing continuous integration and delivery into undergraduate software engineering courses. *The 30th IEEE Conference on Software Engineering Education and Training*, December 2010.

[11] Michael Armbrust et al. Above the clouds: A berkeley view of cloud computing. *Electrical Engineering and Computer Sciences University of California at Berkeley*, February 10, 2009.

[12] GitLab. Gitlab ci/cd for github. `https://about.gitlab.com/solutions/github/`. Accessed: 2019-01-17.

[13] GitLab. Junit xml test summary in pipeline view. `https://gitlab.com/gitlab-org/gitlab-ce/issues/49212`. Accessed: 2019-01-16.

[14] GitLab. Repository mirroring. `https://docs.gitlab.com/ee/workflow/repository_mirroring.html`. Accessed: 2019-01-16.

[15] Google. Build configuration overview. `https://cloud.google.com/cloud-build/docs/build-config`. Accessed: 2019-01-22.

[16] Google. Google-managed encryption keys. `https://cloud.google.com/storage/docs/encryption/default-keys`. Accessed: 2019-01-22.

[17] Google. How google protects your data in transit. `https://cloud.google.com/blog/products/gcp/how-google-protects-your-data-in-transit`. Accessed: 2019-01-22.

[18] Google. Machine types. `https://cloud.google.com/compute/docs/machine-types`. Accessed: 2019-01-22.

[19] Jez Humble. Continuous delivery sounds great, but will it work here? *Communications of the ACM*, April 2018.

[20] Jez Humble and David Farley. *Continuous Delivery - Reliable Software Releases Through Build, Test And Deployment Automation*. Addison-Wesley, 2010.

[21] Continuous Integration and Delivery. Circleci. `https://circleci.com`.

[22] Continuous Integration and Delivery. Gitlab ci/cd. `https://about.gitlab.com/product/continuous-integration/`.

[23] Continuous Integration and Delivery. Jenkins. `https://jenkins.io`.

[24] Jenkins. Automated upgrade. `https://wiki.jenkins.io/display/JENKINS/Automated+Upgrade`. Accessed: 2019-01-16.

[25] Jenkins. Storing secrets. `https://jenkins.io/doc/developer/security/secrets/`. Accessed: 2019-02-14.

[26] Jenkins. Using credentials. `https://jenkins.io/doc/book/using/using-credentials/`. Accessed: 2019-02-14.

[27] MacStadium. Pricing. `https://www.macstadium.com/pricing`. Accessed: 2019-01-16.

[28] The Agile Manifesto. https://agilemanifesto.org/principles.html. Accessed: 2019-01-09.

[29] Steve Neely and Steve Stolt. Continuous delivery? easy! just change everything. *2013 Agile Conference*, August 5-9, 2013.

[30] Nevercode. The maintenance side of jenkins ci. https://nevercode.io/blog/the-maintenance-side-of-jenkins-ci/. Accessed: 2019-02-16.

[31] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices. *IEEE Access Volume: 5*, 22 March 2017.

[32] Softhouse. Maturity model for continuous delivery. https://www.softhouse.se/wp-content/uploads/2016/02/mognadsmodellen_miniposter_liggande1.pdf. Accessed: 2018-09-19.

[33] Mitesh Soni. End to end automation on cloud with build pipeline: The case for devops in insurance industry, continuous integration, continuous testing, and continuous delivery. *IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*, 25-27 Nov. 2015.

[34] Andreas Steffens, Horst Lichter, and Jan Simon Lichter. Designing a next-generation continuous software delivery system: Concepts and architecture. *ACM/IEEE 4th International Workshop on Rapid Continuous Software Engineering*, 1-7 May, 2018.

[35] Jakob Svemar. Showstoppers for continuous delivery in small scale projects. Master's thesis, Lund university, 2016.

[36] Techopedia. Anything as a service. https://www.techopedia.com/definition/14027/anything-as-a-service-xaas. Accessed: 2019-01-09.

Appendices

Appendix A

Assessment

A.1 Assessment Questions

These questions are translated from Swedish. The questions in the categories are grouped linking them to the Maturity Model levels [32].

General

- Which tools are used in the system and for what purposes?
- Which IDEs are used?
- Which steps does the team follow to ensure deliverable code? (Checklist)

Code & Integration

- How is the code version controlled? (SCM)
- How is the code integrated and how often? (CI)
- How does the documentation of code changes occur? (Traceability)
- How does the team's branching strategy look like?
- How is system monitoring performed?
- How are bugs and major changes managed?
- Are there tools to visualise the architecture, changes and branches?
- Is there a way to track the changes or tasks?
- How are dependencies managed in the system? Which dependencies are there in the system?
- How is code developed? (Software development method)

Build

- How does the builds occur?
- Which building data is saved? Is it available to all developers?

- Is static code analysis a part of the build?
- How are binary dependencies dealt with?

- What is it that triggers the builds? Are there regular builds?

- How is the building environment generated?
- Is there a visualisation of trends from the static code analysis?

Test & Quality Control

- What types of tests are there in the team?
- Which tests run before commit?
- Which tests run as part of the build?
- How much of the tests are automated? Which are they?

- When do the tests happen?
- Are the test results available to all developers?

- How do functional tests occur?
- How detailed are the test results?

- Is the test code included in the release to the customer?
- Are all configurations tested?
- Is debug information documented?

- What kind of collaboration is there between developers and testers?
- Are the quality and performance visualised?
- How often are the tests re-visited?

Delivery

- In what way are deliveries made?
- When are deliveries made?
- What is included in the delivery?

- How is the test and production environment created?

- Is there a downtime?
- Are deliveries monitored and visualised?

Team

- How is the team organised?
- How is the communication handled in the team?
- Who owns the code?

- When was the last time an assessment was made for the system?
- Who is responsible for CD in the team?
- Are all parts of CD in the system?
- When was CD implemented in the team?

A.2 Form Questions

These questions are translated from Swedish.

1. What is your role in the team?
2. How trustful is the CD-system in the team, on a scale of 1-5?
3. How knowledgeable would you say that you are within CD, on a scale of 1-5?
4. Which parts of the CD pipeline do you find difficulties to understand? (Code & Integration, Build, Test & Quality Control or Delivery)
5. What do you think is particularly important, from a developer's perspective, within CD?
6. Is there a designated CD responsible in the team?
7. In which part of the pipeline do complications usually occur? (Code & Integration, Build, Test & Quality Control or Delivery)
8. How common are complications in the team related to the CD system, on a scale of 1-5?
9. How quickly are these corrected, on a scale of 1-5?
10. How often do you correct these, on a scale of 1-5?
11. Which difficulties are there usually when it comes to usability and maintainability of the system?

Appendix B

Interview Questions

These questions are translated from Swedish.

Tools

1. Based on your experience of setting up CD-pipelines, how often are the pipelines set up at the customer similar versus customised?
2. Which CI/CD tool do you usually encounter? What do you think is the reason behind it?
3. Do you think there is a tool that can be a solution to CDaaS or is it necessary to implement a completely new tool?

Requirements

1. Why do you think some cast aside cloud-based tools? How central is the safety aspect and special requirements when selecting tools?
2. Are there other things that the team will need to manage with cloud-based tools other than the code, the tests and the pipeline configuration code?
3. Which approach do you think is best, based on your experience, for tool updates, automatic or on demand? Why?
4. Is speed a requirement that customers set on tools that are chosen, in the form of builds and UI performance?
5. What do you think is a reasonable time for setting up and educating for CD in the team? Why?

Appendix C

Final Requirements

Absolute minimum

1. Automatic builds are triggered by commits/changes.
2. Integrates with SCM, software configuration management, tools such as Bitbucket, GitHub, GitLab, (Gerrit).
3. Run ”scripts” for test and deploy.
4. Create a collection of stage blocks, i.e. a pipeline.
5. Report results of builds/runs, i.e. feedback.

Minimum Viable Product Continuous Delivery as a Self-Service

1. No appointed CD maintainer in the team.
2. Team only concerned of code, tests and pipeline configuration code.
3. Reliable and trustworthy:
   (a) CPU, RAM, storage are provided based on need. Resources are scaleable.
   (b) High availability -> No downtime.
4. Secure to manage secrets.
5. Service updates are provided and traceable. Tools/plugins updates are available on demand.
6. Error detection:
   (a) Traceable from failure to origin of error.

(b) Reason for failure clear and directly visible to user.

7. Lower cost than appointed maintainer(s).
8. Start using service as fast as possible.

New Requirements

1. An automatic initial setup with built-in templates.
2. Integrate to third party services, for storing, testing and deploying, on demand.
3. Non-reactive and descriptive syntax for pipeline configuration code.
4. Automated uploading of test reports and artefacts.
5. Use images for supporting different platforms.
6. Automated optimisation of builds.
7. UI provides fast feedback for builds.
8. Automated cost function bills based on usage.
9. An iterative implementation of the pipeline.

Target team

1. Visible to both team and customer that the software is in a working state.
2. Support for multiple platforms(.Net, iOS, Android)
3. Integration to Jira and Slack.
4. Support automatic tests on physical phones.

Appendix D

Testing with team

D.1 Problem formulation

1. Build application.
2. Run tests.
   (a) Make a test fail.
   (b) Find the error in the tool.
3. Trigger a build with test for pull/merge request.
4. Built artefact should be available for download.

Extra

1. Optimise pipeline for faster builds.
2. Test reports should be available for download.
3. Test reports should be presented in the tool.
4. Explore SSH to container and available integrations with the tool.

D.2 Survey

General

1. Which tool did you test? (GitLab CI or CircleCI)
2. How far did you come?

Tool

1. How difficult did you experience the tool to be, on a scale of 1-4?
2. How fast did you experience the builds, on a scale of 1-4?
3. What do you think was characteristically good with the tool?
4. What do you think can be improved with the tool?
5. What do you think the tool lacks?

Testing

1. How helpful did you find the tracing of errors, when it comes to test and configuration errors, on a scale of 1-4?
2. How visible did you find the status of the programme, on a scale of 1-4?
3. What did you experience as most and least reliable with the tool?

Future work

1. Can you imagine maintaining the CD pipeline you set up based on the experienced difficulty? If no, why?

DEPARTMENT OF COMPUTER SCIENCE | LUND UNIVERSITY | 2019–01–28

MASTER THESIS Investigating Continuous Delivery as a Self-Service
STUDENT Seif Al-Shakargi
SUPERVISORS Lars Bendix (LTH) & Fredrik Stål (Softhouse)
EXAMINER Ulf Asklund (LTH)

Designing a Self-Service for Continuous Delivery

Popular Science Article **Seif Al-Shakargi**

Continuous Delivery is an essential component in software development for a reliable and automated quality assurance process. This master thesis has researched a possible design of a Self-Service to easily set up and maintain CD in a cost-effective way, making it available to users without CD expertise.

The process of Continuous Delivery, CD, has today become an essential part of software development as well as a necessity in development teams, both small and big. With CD we can shorten the release cycle and deliver quality assured code to our customers. This is made possible by automating the steps in the development process involving build, test, integration and delivery. The numerous tools available today can involve the whole CD pipeline or parts of it, which means it is your job to tie together the pipeline.

The challenge of introducing CD starts with setting up a CD pipeline, which may take a considerable amount of time and require expertise knowledge within this area. The resources required for this is not always available in smaller teams. This costly and complex implementation will also require maintenance once in use. At other times an implemented CD pipeline will not be properly maintained as the needed expertise is not available, which results in a non-functioning or disabled CD pipeline. The complexity does also lie in setting up an infrastructure for computing and storage capabilities to your CD system.

A new approach is suggested to tackle these difficulties to CD, by delivering it "as a Self-Service" to software development teams. Continuous Delivery as a Self-Service, CDaaSS, is abstracted and adjusted for use by developers without needing expertise. It is also directly usable in the same sense like Software as a Service tools, without needing to set up an infrastructure. The service provides the developers with guidelines, which helps the developers implement their own pipeline and processes.

This master thesis has researched a possible design of Self-Service by assessing a team, testing tools and interviewing experts. A requirement specification was developed by looking into usability, maintainability, infrastructure and cost aspects addressing expert related tasks.

A proof-of-concept design was drawn up for implementing the requirement specification, where the cloud is a central component as the necessary infrastructure can be provided on-demand without maintenance of it. A cost analysis based on a small team showed that the cost of using a service per month is comparable to a day's wage for a developer.

A validation of the design was made with Software as a Service tools, where the team implemented a pipeline. The end result was 3 out of 4 testers willing to maintain the pipeline based on the experienced difficulty with the tools.