Kim Rejström

Implementing Continuous Integration in a Small Company:
A Case Study
Release engineering is the process responsible for taking individual code contributions of developers and bringing those to the end user in the form of high quality software releases. This process encompasses code change integration, build system specifications, infrastructure-as-code, deployment and release. Organizations have started automating various parts of their release pipelines resulting in the practices of continuous integration (CI), continuous delivery (CD), continuous deployment, DevOps and continuous value delivery.

The goal of this thesis is to produce an updated release pipeline for the case company and evaluate the proposed solution from a CI/CD perspective. This is done through design science research, where an artifact (the new release pipeline) is constructed based on a series of feasibility studies aimed at finding an optimal solution to the elicited needs of the company.

The study implemented a modern, automated release pipeline that solved the problems targeted. The solution includes many of the typical methods and tools documented in existing research: automated builds, test automation, configuration management and automated deployments. The CI/CD adoption was not without problems, as challenges related to resource allocation, security and QA efforts emerged. The transition was however successful and as a result of the implementation, perceived benefits included improved release planning, infrastructure-as-code and close tie-ins with the DevOps culture.

Keywords: Continuous Integration, Continuous Delivery, Single Case Study, Release Pipeline, DevOps, Design Science Research, CI assimilation

Language: English
SAMMANDRAG AV
DIPLOMARBETET

Utfört av: Kim Rejström
Arbetets namn: Implementering av Kontinuerlig Integration i ett litet företag: En fallstudie
Datum: Den 27 Oktober 2016
Sidantal: vii + 100
Huvudämne: Programvaruutveckling och affärsverksamhet
Kod: T-110

Övervakare: Professor Casper Lassenius
Handledare: Maria Paasivaara, Juuso Pesola

Publiceringsutveckling är processen ansvarig för att individuella kod bidrag från programvaruutvecklarna levereras till slutanvändarna i form av högkvalitets-programvara. Denna process omfattar integration av kod, specifikationer för build-system, infrastruktur-som-kod samt distribution och publicering av programvara. Organisationer har börjat automatisera olika delar av sina publiceringlinjer, något som resulterat i införandet av kontinuerlig integration (CI), kontinuerlig leverans (CD), kontinuerlig publicering, DevOps och kontinuerlig värde leverans.

Målet med denna avhandling är att producera en uppdaterad publiceringslinje för fallstudie företaget och utvärdera den föreslagna lösningen från ett CI / CD perspektiv. Studien görs som design science forskning, dvs. en artefakt (den nya publiceringlinjen) konstrueras på basen av ett antal förstudier ämnade för att finna en optimal lösning på företagens problem.

I studien implementerades en modern, automatiserad publiceringslinje som löste problemen den addresserade. Lösningen innehåller många av de typiska metod och verktyg dokumenterade i befintlig forskning: automatiserade build-system, testautomatisering, konfigurationshantering samt automatiserade installationer. CI/CD övergången var dock inte problemfri – utmaningar relaterade till resursallokering, säkerhet och QA insatser uppstod på vägen. Övergången var trots detta lyckad och som följer av ett framtågsrikt genomförande uppstod förbättrat publiceringsplanering, infrastruktur-som-kod samt nära band till DevOps kulturen.

Nyckelord: Kontinuerlig Integration, Kontinuerlig Leverans, Fallstudie, Publiceringslinje, DevOps, Design Science Forskning, CI assimilation

Språk: Engelska
I want to express the deepest of gratitude to my fellow colleagues at The Button Corporation for supporting this endeavor from day one. I also want to thank my professor Casper Lassenius together with Maria Paasivaara for guiding me throughout the process. A big thank you goes to Juuso Pesola for all the time he devoted to this project as well.

Espoo, October 27, 2016

Kim Rejström
# Abbreviations and Acronyms

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CI</td>
<td>Continuous Integration</td>
</tr>
<tr>
<td>CD</td>
<td>Continuous Delivery</td>
</tr>
<tr>
<td>CSA</td>
<td>Current State Analysis</td>
</tr>
<tr>
<td>CN</td>
<td>Company Needs</td>
</tr>
<tr>
<td>VCS</td>
<td>Version Control System</td>
</tr>
<tr>
<td>DSR</td>
<td>Design Science Research</td>
</tr>
</tbody>
</table>
Contents

Abbreviations and Acronyms v

1 Introduction 1
  1.1 Problem statement ........................................ 2
  1.2 Structure of the Thesis .................................. 2

2 Background 4
  2.1 Characteristics of Continuous Integration and Delivery . 5
  2.2 Practices of successful CI/CD ........................... 10
  2.3 Challenges in adopting CI/CD ............................ 13
  2.4 Proposed benefits in adopting CI/CD ...................... 15
  2.5 Organizational implications of CI assimilation .......... 16
  2.6 SMB experience reports .................................. 18
  2.7 Integration flow model .................................. 20

3 Research Method 22
  3.1 Research Motivation ..................................... 22
  3.2 Design science research ................................. 23
  3.3 Data Collection ......................................... 24
  3.4 Research Phases ......................................... 26
  3.5 Environment ............................................. 26

4 Current State Analysis 30
  4.1 Requirements ........................................... 31
  4.2 Development ............................................. 33
  4.3 Operations .............................................. 35
  4.4 Testing .................................................. 36
  4.5 Quality .................................................. 37
  4.6 Original Release pipeline ................................ 38
  4.7 Company Needs ......................................... 41
Chapter 1

Introduction

Software engineering is about the creation and delivery of value to its intended customers and users. During the past two decades the software development practices have transitioned from document-driven, rigid waterfall models to lean and agile development practices. These development practices have introduced a new way of looking at value delivery. Value is defined by the users, which means getting useful feedback on whether or not the software under development satisfies the actual requirements of the users is the only way to know if the right thing is being developed. Reducing the time it takes to develop new features, thus shortening the feedback loop between developers and users, decreases the risk of developing the wrong product.

Release engineering is the process responsible for taking the individual code contributions of developers and bringing those to the end user in the form of a high quality software release [Adams and McIntosh, 2016]. This process encompasses code change integration, continuous integration, build system specifications, infrastructure-as-code, deployment and release. Organizations have started automating various parts of their release pipelines [Ståhl and Bosch, 2014b] resulting in the practices of continuous integration (CI), continuous delivery (CD), continuous deployment and continuous value delivery [Dingsøyr and Lassenius, 2016].

CI is a software development practice where software is integrated continuously during development. All the code that is produced must be checked in to a version control system, integrated and built into executables which are then automatically tested [Fowler and Foemmel, 2006]. CD is an extension of CI in the sense that all CI practices are fully included in the CD process as shown in Figure 2.1. CD is defined as a software development practice in which software can be released to production at any time [Fowler, 2013]. This means the CD pipeline not only integrates and tests the code, but moves the executables to production-like environments for acceptance and release.
CHAPTER 1. INTRODUCTION

testing. Continuous deployment further automates CD as no manual action is needed to deploy the product increment.[Fowler, 2013]

1.1 Problem statement

Prior to this study, the company has identified and need to improve its development process. They have decided to adopt agile software development methods within the company. Implementing a CI system was seen as the fundamental first step needed to start a transition towards more agile practices overall within the company. Thus, the problem this thesis aims to solve is how to improve the release pipeline of The Button Corporation. This is performed as design science research, where in addition to evaluating the current state of the company’s release pipeline a CI/CD tool-chain will be implemented and introduced into their release pipeline. The thesis intends to address this problem by answering the following research questions:

RQ1 What are the motivation factors driving the adoption of CI/CD practices within the company?

RQ2 What are the most suitable solutions to the identified company needs?

RQ3 How well does the implemented release pipeline solve the company needs?

This research is conducted as design science research [Hevner et al., 2004] in the context of a single case study [Yin, 2013]. The data used in the study includes quantitative data collected from systems and tools in use within the company, along with qualitative data gathered through workshops held with employees at the company.

CI and CD are in the context of this study often referred to as CI/CD. Because of the interconnected nature of these two practices coupled with the goal of implementing a new release pipeline for the company it is relevant to examine practices related to the entire release pipeline. The CI/CD abbreviation accurately captures the practices related to the release pipeline examined in this study.

1.2 Structure of the Thesis

This thesis is divided into nine chapters. Following this chapter, the core concepts of CI and CD are presented along with a series of case studies relevant to this case. The chapter reviews the existing literature on the topic
of CI and CD, as well as presents the proposed benefits and typical characteristics of successful CI/CD implementations. The third chapter defines the research method and outlines the timeline of the case study as well as describes the context and company studied. The fourth chapter describes the current state of the company’s release pipeline. Based on those results the company needs are identified and described later in the same chapter. In the following chapter the solution proposals and their accompanying feasibility studies are presented. The sixth chapter documents the implementation process. In the following chapters the research questions are answered and the results are evaluated and discussed. The last chapter summarizes the most important findings and conclusions drawn from the study.
Chapter 2

Background

This chapter reviews Continuous Integration and Continuous Delivery practices based on previous literature. These two practices are at the core of the research questions presented in chapter 1 and the focus point of the entire study. The defining characteristics of these two practices are presented in the next section, with focus on the different stages of the release pipeline and the tools involved. As context for the implementation phase of the study the principles and practices that characterize successful CI/CD are also examined. Furthermore the challenges of adopting CI/CD are presented in section 2.3 followed by the proposed benefits of CI/CD in section 2.4 and an overview of the common themes related to assimilation of CI/CD practices in an organization in section 2.5. In addition to the existing academic research on CI/CD, relevant case studies related to CI/CD adoption in small and medium sized businesses are also examined.

The search criteria for identifying relevant literature was two-fold. For the core concepts of CI/CD: widely cited and relevant, up-to-date, established academic research was used. To find these sources (which include: [Fowler and Foemmel, 2006; Fowler, 2013; Humble and Farley, 2010; Humble et al., 2006; Neely and Stolt, 2013; Leppänen et al., 2015; Chen, 2015; Rodríguez et al., 2016]) the bibliographies of existing academic literature containing any of the keywords: continuous integration, continuous delivery, continuous deployment, release engineering, devops, rapid release were evaluated. For the case studies a similar approach was tried, were the keywords case study, case-study, company, organization, small company, small organization were added to the mix. Unfortunately this yielded unfavorable results as existing scientific research on CI/CD adoption in companies have largely targeted big companies and organizations such as Mozilla [Mäntylä et al., 2015], Facebook [Mäntylä et al., 2015; Feitelson et al., 2013], IBM [Bakal and Althouse, 2012], Amazon [Forsgren and Humble, 2015] and Google [Mäntylä et al., 2015; Fors-
CHAPTER 2. BACKGROUND

gren and Humble, 2015]. To find more relevant literature targeting CI/CD implementations in small to medium sized businesses (SMB), blogs, seminar and conference presentations etc. had to be used. There is a substantial amount of SMBs sharing their experience of CI/CD adoption available online. It is however not scientific research and is thus subject to a number of validity threats. In the context of this study, the findings presented in these reports are considered circumstantial and anecdotal. Nonetheless, these reports provide insights into the specifics of implementing CI/CD in SMBs and can help establish a frame of reference for this study. These reports are presented in section 2.6.

2.1 Characteristics of Continuous Integration and Delivery

The first mention of CI was in 1994 in the context of developing micro processes [Grady, 1994]. The term was adopted by Kent Beck in his definition of Extreme Programming (XP) in 2000 [Beck, 2000]. Martin Fowler, a leading figure in making CI popular and Chief Scientist at ThoughtWorks\(^1\), a company making CI and CD tools, is a CI/CD evangelist credited with establishing the current definitions of the practices. Fowler defines CI as:

Continuous Integration (CI) is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible.[Fowler and Foemmel, 2006]

Fowler extended his views on CI to CD, which he defines as:

Continuous Delivery (CD) is a software development discipline where you build software in such a way that the software can be released to production at any time.[Fowler, 2013]

Figure 2.1 shows the relationship between the CI pipeline and the CD pipeline as described by Humble and Farley: a stage-gate process consisting of a commit stage, acceptance test stage and other tests stage [Humble and Farley, 2010]. Outlined is the CI part of the pipeline, containing the Version Control System (VCS) and the commit stage. Within the commit stage the typical elements of a CI system can be found: build, unit test and integration test. In the next section the workflow and tools involved in CI are presented.

\(^1\)https://www.thoughtworks.com/
CHAPTER 2. BACKGROUND

Continuous Integration

The CI workflow requires developers to integrate early and often to enable teams to detect problems early [Fowler and Foemmel, 2006]. When code is checked in to a VCS it is automatically built, unit tested and finally integration tested. The results are reported back to the developer. With CI, working software is the default, and a failure at any point in the CI pipeline renders the build broken. Until the developer receives a notification that the integration was successful, he or she is not done with the commit [Fowler and Foemmel, 2006]. Humble and Farley further refined the idea, saying that broken builds should redirect the team’s focus towards fixing the issue instead of developing new features to ensure the software is kept in a always releasable state [Humble and Farley, 2010].

CI Workflow

The CI workflow can thus be broken up in to the following tasks [Fowler and Foemmel, 2006]:

- Developers check out code into their private workspaces.
- When done, commit the changes to the repository.
- The CI server monitors the repository and checks out changes when they occur.
- The CI server builds the system and runs unit and integration tests.
- The CI server releases deployable artifacts for testing.
- The CI server assigns a build label to the version of the code it just built.
- The CI server informs the team of the outcome of the build.
• If the build or tests failed, the team fixes the issue at the earliest opportunity.
• Continue to continually integrate and test throughout the project.

This straightforward workflow provides continuous feedback on the quality of the code, ensuring developer attention is kept on the current code change until the change has been verified to work.

CI Tools

CI is largely tool agnostic. There exists many different tools aimed for use within the CI pipeline, but there is no optimal tool-chain solution, only opinionated answers. Each solution depends on the project, frameworks in use, skill-set of the stakeholders and other factors. There are however two must-have tools of any CI system: the version control system (VCS) and the CI server. Popular VCSs include Git\textsuperscript{2}, SVN\textsuperscript{3} and Mercurial\textsuperscript{4}. VCSs are often coupled with file browsers, issue trackers and code reviewing tools in to VC Platforms such as Github\textsuperscript{5}, Bitbucket\textsuperscript{6} and Gitlab\textsuperscript{7}.

CI servers range from open source solutions like Jenkins\textsuperscript{8}, Hudson\textsuperscript{9} and GoCD\textsuperscript{10} to commercial solutions like TravisCI\textsuperscript{11}, CircleCI\textsuperscript{12}, CodeShip\textsuperscript{13} and TeamCity\textsuperscript{14}. Each of these solutions offer a similar core tool-set with additional features aimed at setting them apart from the competition. Choosing the appropriate tools is about finding a balance between price, setup and configuration effort, ease-of-use, integration capabilities between the selected tools, framework suitability and maintainability in respect to current codebase.

\textsuperscript{2}https://git-scm.com/
\textsuperscript{3}https://subversion.apache.org/
\textsuperscript{4}https://codeship.com/
\textsuperscript{5}https://github.com/
\textsuperscript{6}https://bitbucket.org/
\textsuperscript{7}https://about.gitlab.com/
\textsuperscript{8}https://jenkins-ci.org/
\textsuperscript{9}http://hudson-ci.org/
\textsuperscript{10}http://www.go.cd/
\textsuperscript{11}https://travis-ci.com/
\textsuperscript{12}https://circleci.com/
\textsuperscript{13}https://codeship.com/
\textsuperscript{14}https://www.jetbrains.com/teamcity/
Continuous Delivery

The CD workflow aims to assure that the software is deployable throughout its lifecycle and that the team prioritizes keeping the software deployable over working on new features. Fowler suggests that the key test to determine if an organization is doing CD successfully is:

A business sponsor could request that the current development version of the software can be deployed into production at a moment’s notice - and nobody would bat an eyelid, let alone panic. [Fowler, 2013]

CD is enabled through the deployment pipeline [Humble et al., 2006]. The purpose of the deployment pipeline is threefold: add visibility, provide feedback and continually deploy.

Visibility - All aspects of the delivery system including building, deploying, testing, and releasing are visible to every member of the team.

Feedback - Anybody can get fast, automated feedback on the production readiness of their systems any time somebody makes a change to them.

Continually deploy - Through a fully automated process, you can deploy and release any version of the software to any environment. [Fowler, 2013]

Ideally the deployment pipeline should be as automated as possible. In addition, all manual work pertaining to the release is to be minimized. [Humble and Farley, 2010]

CD Workflow

The CD workflow, as seen in Figure 2.2 is largely automated and feedback driven. It is a direct extension of the CI workflow, extending further towards releasing the software. The deployment pipeline is a stage-gate process, where a set of activities are performed at each stage, evaluated and if deemed OK, allowed to proceed to the next stage. Feedback from each stage is communicated to the developer. This way tracking down a problem is made easier as knowledge about which stage failed is readily available.

CD Tools

In addition to the tools used in the CI pipeline CD makes use of provisioning and deployment tools. As moving executables towards production-like environments is part of the pipeline, managing those environments becomes a major task in CD. The DevOps movement, a trend of close collaboration between development and operations teams has established configuration man-
These tools include IT automation and configuration management tools such as Chef\textsuperscript{15}, Puppet\textsuperscript{16}, Ansible\textsuperscript{17} and Capistrano\textsuperscript{18}. In addition to these, virtualization tools like Vagrant\textsuperscript{19} and Docker\textsuperscript{20} are also common. Because of the frequent use of these tools together with CI tools, out-of-the-box integrations and well-documented set up instructions for linking CD tools with CI tools are widely available online. Successfully orchestrating these tools into a well functioning release pipeline is a difficult task. In the next section the common practices for achieving this is presented.

\textsuperscript{15}\url{https://www.chef.io/chef/}
\textsuperscript{16}\url{https://puppet.com/}
\textsuperscript{17}\url{https://www.ansible.com/}
\textsuperscript{18}\url{http://capistranorb.com/}
\textsuperscript{19}\url{https://www.vagrantup.com/}
\textsuperscript{20}\url{https://www.docker.com/}
CHAPTER 2. BACKGROUND

CI/CD Practices

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Automate the build</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Make your build self-testing</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Every commit should build on an integration machine</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Keep the build fast</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Keep the build green</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Test in a clone of the production environment</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Everyone can see what’s happening</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Automate deployment</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

Table 2.1: A summary of successful practices in CI/CD in existing literature

2.2 Practices of successful CI/CD

Successfully adopting CI or CD practices goes far beyond just setting up the right tools. It requires the organization to rethink the way it develops its products. Based on the evaluation of the 4 studies presented in Table 2.1, a set of 8 themes were identified as commonly documented successful practices. What follows is an overview of these 8 practices and how they impact the successful adaptation of CI and CD.

Automate the build. At the core of CI and specifically the commit stage of the pipeline is triggering the build. Each commit should automatically trigger the pipeline and the first step is to build the binaries. The executable being built in this stage makes its way through the entire pipeline to become a verified, deployment ready version at the end. The when and what in the triggering of the build are significant questions to answer as builds and test suites can be time-consuming processes.

Make your build self-testing. As the pipeline is largely dependent on a trigger-feedback control-sequence, testing is at the heart of the pipeline. The trigger for moving on to the next phase of the pipeline is a positive test result.
CHAPTER 2. BACKGROUND

from the previous phase. This way it is possible to halt the process at the first sight of problems. This means the source code contains not only code for the software itself, but unit and integration test code as well. With each increment, tests are added along with the new features. Code health-checks or semantic tests are often included in the automated test suites, linting the code to check for syntax errors and "code smells". Code smell is a surface indication that usually corresponds to a deeper problem in the system [Fowler and Foemmel, 2006]. They are not inherently faulty but act as indicators for violations of fundamental design principles that negatively impact design quality.

Every commit should build on an integration machine. To ensure homogeneity and to eliminate environment caused problems, CD makes use of the principle "Build binaries once, deploy the same way" [Humble and Farley, 2010]. When all binaries that pass through the pipeline are built in the commit stage we know the executable we end-up with at the end of the pipeline is in fact the one that was tested.

Keep the build fast. The speed of the pipeline, the builds, the tests and the feedback loops are cornerstones of any CI/CD process. To ensure all of these remain fast, CD relies on increased release frequency to ensure small code changes per release. This is important as it means that less errors will be introduced per release. Each commit will thus be easier and faster to test and consequentially also to fix [Fowler, 2013].

Keep the build green. In CI/CD, development should always take place on a known and stable base [Fowler, 2013]. That means focus needs to be on deployable software over new features. Meaning, if a code change breaks the build it becomes a high priority task to fix it. Automated tests ensure the developer receives constant feedback on the state of the build. With frequent commits fast test suites are of high importance as they need to be comprehensive enough to guarantee high quality, but simultaneously not limit the speed of the release cycle [Leppänen et al., 2015].

Test in a clone of the production environment. A significant part of finding any problems the system may encounter in production is testing the environment within which the production system will run. Testing in a different environments introduces a risk that what happens under test conditions will not happen in production. Thus, the test environment should be as exact a mimic of your production environment as possible.[Fowler, 2013] To achieve this configuration management, virtualization tools and infrastructure-as-code play a vital role. They allow you to specify recipes or blueprints for provisioning environments. This eliminates the manual work from setting up servers and creates identical environments automatically.

Everyone can see what’s happening. Communication and transparency
are vital to continuous methods. A key success factor is the introduction of information radiators and metrics in the development environment [Humble and Farley, 2010]. In its simplest form a red and green light is enough to communicate the status of the build. However the information radiators can be much more sophisticated showing not only the status of a build, but also measurements like cycle time, test coverage and build success percentages [Humble and Farley, 2010]. In addition to providing feedback from the deployment pipeline the transparency from the development process has been proved to heighten team moral and have a positive effect on developers’ sense of accomplishment and motivation [Leppänen et al., 2015].

Automate deployment. Multiple environments are a part of any deployment pipeline - from environments to run commit tests, secondary tests and acceptance tests to staging and production environments. Since executables are being deployed to these environments multiple times a day, doing this automatically significantly reduces the workload. Thus, the importance of having scripts that will allow deploying the application into any environment easily is significant. A sign of a good CI/CD practice is that the newest software increment to come out of the deployment pipeline can be deployed at any time. Thus, a well working CD practice is constantly in a state where deployment by the push of a button is possible [Fowler, 2013]. A pre-requisite for this is: a green build and automatic deployment scripts. As the deployments are being done in clones of the production environment, the deployment scripts can largely be the same as well. Exclusively using scripts for deployment eliminates the chance of manual error and provides a reliable and traceable way of doing it. Tools like Ansible and Capistrano often provide a rollback feature in addition to deployment capabilities further encouraging frequent deployments as reverting to the previous version is quick and easy.

These 8 principles should guide the CI/CD implementation within organizations. The principles have been established as best practices through the validation of these concepts through several literature reviews [Rodriguez et al., 2016; Mäntylä et al., 2015; Ståhl and Bosch, 2014a]. They can be viewed almost as a design pattern for adopting CI/CD, offering a generic solution adaptable to most cases. Although following this design pattern, a CI/CD adoption is a challenging task not without problems. The following section goes over the most common challenges in adopting CI/CD.
2.3 Challenges in adopting CI/CD

As with the successful practices, the challenges in adopting CI/CD presented in this section are established, commonly occurring themes from existing research. The seven challenges are typical problems, tripping-stones that propose a threat to the successful adoption of CI/CD. They should be evaluated from a risk assessment perspective, when implementing CI/CD practices. Not all challenges apply to every case, but they are frequent enough to put some effort into mitigating the risk of them being realized.

*Change resistance.* Transforming towards more continuous deployment practices is an evolutionary process that requires investments and involvement from the entire organization [Rodríguez et al., 2016]. As with any larger changes to established behavior, people’s mindsets and the general organizational way of working needs to adapt to the change for it to be successful. Resistance to change is a common issue that usually manifests on both a personal level and on a decision level within the organization. A stiff organizational culture challenges the adoption of CI/CD as management may be unwilling to risk losing productivity over new, experimental processes [Leppänen et al., 2015]. In addition to a change in organizational culture, successfully adopting CI/CD practices requires a buy-in from all key stakeholders and transparency within the organization [Neely and Stolt, 2013].

*External constraints.* All software is part of larger contexts where external constraints may affect the delivery and deployment of said software. Especially customer preferences and domain imposed restrictions have been identified as challenges when transitioning towards continuous delivery. Customers may be reluctant to deal with frequent releases. A big inhibitor from a customer point of view is the learning curve that continuous changes request from the end-users along with the increased workload of taking the new version into active use [Rodríguez et al., 2016]. The domain is another challenge. In highly regulated domains, such as telecom and medical, regulations may require extensive testing before new versions can be allowed to enter production [Rodríguez et al., 2016]. These restrictions can make the full adoption of CI/CD almost impossible.

*QA effort.* To ensure quality is being built-in, the automated test suite needs to be exhaustive enough to provide developers confidence in that the software is working as it should. This can lead to increased QA efforts due to difficulties in managing the test automation infrastructure [Rodríguez et al., 2016]. The lack of automated testing is a major barrier when transitioning to more continuous software releases. Especially at first, when the test suite
is being developed, trust in defect-free builds is easily lost as automated tests have relatively low coverage. This makes developers feel exploratory testing is needed before the release [Leppänen et al., 2015].

**Legacy Code.** Projects often carry technical debt in the form of legacy code. The legacy software has not been designed to be automatically tested and may cause integration failures inhibiting the continuous deployment flow. Furthermore the legacy software might be owned by another team or even company, shifting the testing responsibility to other developers, thus delaying the deployment process [Rodríguez et al., 2016].

**Complex software.** Software projects vary widely in size, complexity and interdependency. The more complex the software is, the more challenging setting up a deployment pipeline has proven to be [Leppänen et al., 2015]. The size of the codebase affects both the testing suite and deployment tasks. Developing the tests takes considerable effort and running large test suites with good coverage is time consuming. Building executables with complex, interdependent components is challenging to automate and can result in very large binaries that are slow to transfer.

**Environment management.** One challenge in software development and deployment is the different environments used in development, testing, and production systems. Often the environments differ from each other an keeping all the environments similar and in sync can be challenging. Differences in environment configurations used in development, testing and production can result in undetected issues making their way to the production environment. Good configuration management enabling automated provisioning of environments is essential to the success of CD [Leppänen et al., 2015].

**Manual Testing.** Not all quality attributes can be verified using automated tests. Automated tests do not necessarily cover all essential aspects of the software. Non-functional requirements such as performance, UX/UI and security often need additional manual testing before the software can be released. Although the manual testing stage is part of the deployment pipeline [Humble and Farley, 2010], cumbersome manual testing poses a challenge to the overall speed and smoothness of the process [Leppänen et al., 2015].

When implementing CI/CD it is important to analyze the existing software to try and identify if any of these challenges apply. Any potential red flags should be further explored and adequately addressed. Furthermore it is worth noting that this collection is in no way exhaustive and companies will always face case-specific challenges not present in existing literature. However, when successfully dealing with these challenges, the rewards of CI/CD will start to present themselves. In the next section the proposed benefits of adopting
2.4 Proposed benefits in adopting CI/CD

The proposed benefits presented in this section have been frequently reported as directly caused by the CI/CD adoption in previous research. These benefits appear in existing literature under a variety of synonymous names, here grouped together in five themes. It is worth noting that these benefits are fairly generic, common also within other software development method improvements such as agile transformations [Laanti et al., 2011] and lean software development [Poppendieck and Poppendieck, 2003]. Organizations adopting CI/CD can expect improvements in these areas, although more case-specific results are likely to appear as well.

**Shorter time-to-market.** Shorter release cycles enable organizations to constantly develop, learn and improve their offerings. With fast and frequent releases the organization gains a better understanding of its customers and the market [Neely and Stolt, 2013]. Focus can thus be directed towards deploying relevant functionalities that meet customers’ expectations. In addition, shorter release cycles have been linked to making release planning slightly easier [Khomh et al., 2012]. Delivering frequently can be seen as a reducing waste, as features can be deployed as soon as they're done [Leppänen et al., 2015]. Moreover, frequent releases have also been seen as enablers of experimentation, as new ideas can be tried out easily and with reasonably low risk [Neely and Stolt, 2013].

**Rapid feedback.** Frequent releases mean customers can evaluate the enhancements and provide feedback immediately and in a continuous way, which improves communication between the organization and its customers [Neely and Stolt, 2013]. Rapid feedback has been linked to several CI/CD related benefits. It allows teams to work only on useful features, as features deemed not useful can be quickly identified and abandoned early on [Chen, 2015]. Thus, the organization is more likely to develop the ”right” product [Fowler, 2013]. As CI/CD allows continual product enhancement and immediate access to new features and bug fixes, it has been linked to increased customer satisfaction [Mäntylä et al., 2015].

**Improved software quality.** The heavy reliance on automated testing combined with smaller more manageable releases have been linked to an increase in software quality [Leppänen et al., 2015]. Studies have also reported a decrease in the number of open bugs and production incidents after adopting CI/CD practices [Mäntylä et al., 2015]. Static code analysis tests have been
proved to be low-effort high-reward additions to the test suite as they only require an initial setup of rules. Linting can then automatically be done on each commit.

*Improved release reliability.* A working deployment pipeline along with intensive automated testing and fast rollback mechanisms positively affects release reliability and quality [Neely and Stolt, 2013]. Automated deployment coupled with information radiators provide safer environments for deploying code to production. Smaller, more rapid releases means that fewer things can go wrong in a release [Fowler, 2013]. When the deployment pipeline makes it possible to automatically rollback to a previous version, the amount of release related stress amongst developers and other stakeholders is reduced [Neely and Stolt, 2013][Chen, 2015].

*Improved developer productivity.* Automated deployment processes have been linked to improved developer productivity. Significant time savings for developers have been observed as CI/CD practices eliminate most of the manual work related to environment management and other non-value adding tasks [Rodríguez et al., 2016]. Although the setup costs of a deployment pipeline can be quite high and require vast amounts of effort, when setup they free up developers time, allowing them to focus on actual, value-adding software development work [Humble and Farley, 2010].

These benefits have one thing in common, they are incremental improvements upon the status quo. All of these proposed benefits improve upon their respective metric, without bringing anything new to the mix. It is likely that an organization adopting CI/CD is doing it to solve some concrete problem they are facing. In those cases the benefits of CI/CD will likely extend to more problem-specific contexts, possibly adding completely new benefits to the mix. However to achieve any benefits the adoption of CI/CD needs to be successful. In the following section the question of how to plan this adoption, in other words the organizational implications of CI assimilation are examined.

### 2.5 Organizational implications of CI assimilation

A transition to using CI/CD imposes a disruptive change on any organization. To deal with the change successfully a plan for how to introduce CI in to the organization is needed. CI/CD adoption is a development-centric task, but it affects a large ecosystem consisting of *development, operations,*
testing, product management, sales, marketing, customers, users and management. Forming a plan for how to effectively manage the change: how to communicate with all stakeholders, how to deal with potential performance dips, how to foster customer involvement etc. becomes vital to the success of the adoption [Eck et al., 2014].

<table>
<thead>
<tr>
<th>Purposeful assimilation Processes &amp; organization</th>
<th>Acceptance</th>
<th>Routinization</th>
<th>Infusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Devising an assimilation path</td>
<td>4</td>
<td>A,D</td>
<td>CI assimilation metrics</td>
</tr>
<tr>
<td>Overcoming initial learning phase</td>
<td>3</td>
<td>C,D</td>
<td>3</td>
</tr>
<tr>
<td>Clarifying division of labor</td>
<td>5</td>
<td>A,B,D</td>
<td></td>
</tr>
<tr>
<td>CI and distributed development</td>
<td>5</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>Testing</td>
<td>Dealing with test failures right away</td>
<td>4</td>
<td>A,B,C,E</td>
</tr>
<tr>
<td>IT infrastructure</td>
<td>Introducing CI for complex systems</td>
<td>7</td>
<td>A,C,E</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>6</td>
</tr>
</tbody>
</table>

Figure 2.3: Assimilation of CI/CD practices [Eck et al., 2014]

Figure 2.3 shows the 14 themes related to CI/CD assimilation discovered through a concept-centric literature review [Webster and Watson, 2002] in which 34 relevant CI/CD case studies were analyzed [Eck et al., 2014]. Each theme was defined to belong to one of the three post-implementation assimilation stages: acceptance, routinization, or infusion [Cooper and Zmud, 1990]. The stages describe the change process related to CI/CD adoption and are defined as:

- **Acceptance stage**: Organizational members start to accept the innovation and use it in their daily work.
- **Routinization stage**: Usage is perceived as a normal activity and organizational processes are adjusted accordingly.
- **Infusion stage**: Usage is characterized as sophisticated and extensive.

Realizing that CI/CD adoption has organizational implications that extend beyond development is extremely important for the success of the transition. To what extent other stakeholders are affected is largely company-specific, but assessing the impact the change has on the whole life-cycle of
the product will not only better the chances of the change succeeding, it will bring attention to more remote potential benefits otherwise easily overlooked. Furthermore outlining the stages of the change provides a frame of reference for how far along the adoption has come. This also makes it easier to relate your specific case to other case studies.

In order to establish a frame of reference for this study, the next section presents the anecdotal findings of CI/CD adoptions in SMBs.

### 2.6 SMB experience reports

As stated in the beginning of this chapter, scientifically documented and published articles relating to the adoption of CI/CD in small companies or organizations were not found. Instead a large number of experience reports published as blog posts, videos, presentations and marketing materials online were found. After evaluating the material 6 experience reports that displayed thorough analysis: explaining challenges, practices and benefits of the corresponding CI/CD adoptions were included in this study.

All 6 studies report a successful adoption of CI/CD. The companies range from 50-1000 employees total, with development departments at roughly a 100 employees or less. Their definitions of CI/CD correspond with that of existing literature and with what has been presented earlier in this chapter.

Common challenges reported that have not been covered in this chapter already included: *backwards compatibility, handling of data schema changes, managing test runtimes and insufficient test coverage*. Managing external dependencies and schema changes, basically any change affecting a part of

<table>
<thead>
<tr>
<th>Company</th>
<th>Report</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMVU</td>
<td>Scaling Continuous Deployment at IMVU</td>
</tr>
<tr>
<td>Wealthfront</td>
<td>DevOps Cafe on Continuous Deployment and Operations Dashboards</td>
</tr>
<tr>
<td>Etsy</td>
<td>Quantum of Deployment</td>
</tr>
<tr>
<td>Disqus</td>
<td>Pitfalls of Continuous Deployment</td>
</tr>
<tr>
<td>Box</td>
<td>Continuous Deployment in Desktop Software</td>
</tr>
<tr>
<td>HubSpot</td>
<td>How We Deploy 300 Times a Day</td>
</tr>
<tr>
<td>Wix</td>
<td>Continuous Delivery at Wix</td>
</tr>
</tbody>
</table>

Table 2.2: CI/CD experience reports from SMBs
the system not in the code repository posed a challenge. This was closely related to maintaining both backwards and forwards compatibility and being able to test different versions of the system on the same machine. Solutions included

”When a build runs it locks in the versions of every dependency. This means that if that build is deployed today, or in a year, it will still point to the correct dependencies.” [HubSpot21]

”Each component has to function with latest, next or prior version of other components (including DBs)” [Wix22]

Furthermore test coverage of legacy code was seen as a big challenge [IMVU23, Etsy24, Box25]. To mitigate the effects, several companies suggested Test-driven development (TDD) or simply requiring full coverage on all new code produced. This however did not address the problem of untested legacy code which reportedly caused problems through regression. Suggestions included investing time in writing tests for the critical parts of the legacy code and incrementally writing tests for the parts failing through regression [IMVU, Disqus26].

These challenges are not specific to SMBs and could affect a differently sized company just as well. There were however some challenges more directly linked to the size of the company that are worth mentioning.

”The biggest challenges come with growth of the engineering organization. As the team grows, we’re more likely to have test failures” [IMVU]

”No matter how much we grow, there is always a single team of three or four developers who own and are deploying any part of the product.” [HubSpot]

Several of the companies mentioned growth related challenges with CI/CD [IMVU, Hubspot, Wix, Wealthfront27]. Sharing knowledge of how the release pipeline functions with new employees, establishing a testing culture and ensuring that quality is built-in when the development team expands beyond

---

21http://product.hubspot.com/blog/how-we-deploy-300-times-a-day
22http://www.slideshare.net/yoavaa/continues-delivery-at-wix
23http://bit.ly/2dC2m1A
24https://codeascraft.com/2010/05/20/quantum-of-deployment/
26http://www.slideshare.net/zeeg/pitfalls-of-continuous-deployment
27https://wlth.fr/2dC2rWm
the ones that implemented the CI/CD system were seen as challenges related to the size of the company. Employee training and knowledge sharing sessions together with an organizational culture supporting CI/CD were suggested solutions. Also modularizing the software and ensuring teams owned manageable sized independent modules was suggested as a solution to scaling organizations.

Furthermore all reports called upon future adopters to prepare themselves for system outages, frustration and dips in performance during the change. All companies emphasized that the change is most definitely worth it, just not an easy endeavor.

### 2.7 Integration flow model

As the last part of this background chapter the integration flow model is presented. The integration flow model was created as a way of modeling the software integration flow of any arbitrary software organization. Thus enabling the direct comparison of different release pipelines [Ståhl and Bosch, 2014b]. The model was successfully used to describe the release pipeline in five out of five cases, in a study carried out by the authors [Ståhl and Bosch, 2014a]. The integration flow model was chosen for this study to easily create comparable models of the company’s release pipeline. One model depicting the initial pipeline, and another depicting the CI/CD incorporated model. The reason for using an established model notation is to enable comparisons to existing research and to allow future research to easily compare implementations to this study.

The model consists of six elements used to describe the integration flows. Figure 2.4 shows the elements and there relations to each other. The model consists of three types of nodes: *Input nodes* (triangles) are sources that provide input to the model, *Activity nodes* (rectangles) perform actions on data input and *Trigger nodes* (circles) describe possible external triggering factors. In addition to these nodes, there are three more elements in the model: *Input edges* (dashed arrows) map the flow of data between nodes, *trigger edges* (solid arrows) describe the conditions for triggering activity nodes, and finally, node attributes describe the scope and characteristics of the nodes.[Ståhl and Bosch, 2014b,a]

The literature reviewed in this chapter provides a context for the study when moving forward. In addition to general CI/CD industry best practices, more specific data relevant to a small company have been presented. Any and all decisions and artifacts produced in this study will in addition to being
evaluated against the company needs also be evaluated against this context. The next chapter describes the research method and motivation for this study in more detail.
Chapter 3

Research Method

In this chapter the research motivation along with the timeline and corresponding research phases of the case study are outlined. The method used in this thesis: design science research as defined by Hevner et al. [2004] is also explained. Furthermore the environment, along with the case company and the context it operates in are introduced. Throughout this study different workshops were held to gather information, create actionable plans and evaluate implementations related to the implementation of a new release pipeline. The workshop as a data gathering method is also presented in this chapter.

3.1 Research Motivation

As seen in the previous chapter, several studies on the subject of CI and CD practices exist. The scientific field is however lacking in empirical, industry-based studies on CI/CD implementations and adoptions within organizations. Existing research has largely focused on the theoretical ideal solutions [Humble and Farley, 2010; Fowler, 2013] or on the perceived benefits and challenges of CI/CD implementations [Leppänen et al., 2015; Rodríguez et al., 2016; Neely and Stolt, 2013; Mäntylä et al., 2015; Ståhl and Bosch, 2013].

The case company, subject for the case study in this thesis, is investigating ways to better their release pipeline and remove bottlenecks from development. They have identified CI/CD as a possible solution to some of the perceived problems and are interested in knowing whether CI practices can bring about positive change within their release pipeline. Because of that, the aim of this study is to produce an artifact: a new release pipeline, which is built and evaluated according to the guidelines of design science research, presented in more detail in section 3.2. In chapter 7 the effects of the
implementation are evaluated in detail. The secondary goal of the study is to build on the existing body of academic research within the field of CI/CD. To do this, the study aims to achieve generalizable results related to perceived benefits, challenges and assimilation of CI/CD practices relevant to similar sized organizations. In chapter 8 the results are discussed in the context of previous research.

To efficiently address these needs the following research questions are to be answered:

RQ1 What are the motivation factors driving the adoption of CI/CD practices within the company?

The aim is to identify the crucial needs within the company based on a current state analysis of their development practices and their release pipeline. The goal is to not only elicitate these needs, but categorize, estimate and prioritize them to form a holistic view of the drivers of change in CI/CD adoption.

RQ2 What are the most suitable solutions to the identified company needs?

In the context of best practices, tools and processes, there is no absolute correct answer. The best suitable solution is context specific and often also opinionated and bound by external restrictions like budget, time etc. The aim is to find the best possible solution for this specific case, not a generalizable blue-print for all CI implementations. The artifact produced in this phase of the study is the updated release pipeline.

RQ3 How well does the implemented release pipeline solve the company needs?

Does the proposed solutions solve the problems and needs identified in RQ1? Were the proposed solutions from RQ2 the right ones, or did unforeseen problems emerge? What problems related to adopting the new release pipeline emerged? The success of the implementation is evaluated based on the above questions. Detailed results can be found in chapter 7 and chapter 8.

3.2 Design science research

The main purpose of design science research (DSR) is achieving knowledge and understanding of a problem domain by building an application of a designed artifact [Hevner et al., 2004]. Through the evaluation of artifacts intended to solve identified organizational problems, insights and information
can be gained. Artifacts within DSR are perceived to be knowledge containing. This knowledge ranges from the design logic, construction methods and tools to assumptions about the context in which the artifact is intended to function [Markus et al., 2002].

Design science research is defined by the 7 guidelines presented in Table 3.1. To effectively document and evaluate the artifact produced in this study, DSR in a business environment must be viewed as a mixed method research paradigm. This means both qualitative and quantitative data collection methods will be used in evaluating the artifact.[Venkatesh et al., 2013]

Qualitative research is primarily exploratory research. It is used to gain an understanding of underlying reasons, opinions, and motivations. It provides insights into the problem or helps to develop ideas or hypotheses for potential quantitative research. Moreover, qualitative research is used to uncover trends in thought and opinions, and dive deeper into the problem. Qualitative data collection methods vary from unstructured to semi-structured techniques. Some common methods include focus groups (workshops), individual interviews, and participation/observations.[Stake, 1995]

Quantitative research is used to quantify the problem by generating numerical data or data that can be transformed into useable statistics. It is used to quantify attitudes, opinions, behaviors, and other defined metrics. Quantitative research uses measurable data to formulate facts and uncover patterns in research. Quantitative data collection methods are much more structured than qualitative data collection methods.[Stake, 1995] The two approaches should be regarded as complementary rather than competitive in the context of mixed method research [Wohlin et al., 2003].

In this study the adoption of CI/CD practices within the case company are evaluated through the design and implementation of a new release pipeline. The release pipeline is the artifact produced by this research.

### 3.3 Data Collection

The primary source for gathering the qualitative data for this study will be through a series of workshops. As a vital part of the change itself, stakeholder inclusion in the entire process is essential. Thus, the series of workshops conducted in this study can be regarded as a form of action research: a reflective process of progressive problem solving led by individuals working with others in teams or as part of a "community of practice" [?]. The research in itself does not qualify as action research as this thesis is by its definition individual work. However, the collaborative nature of the research can be viewed as a form of collaborative learning in the workplace [Tynjälä, 2008]. Collabora-
Design-Science Research Guidelines

<table>
<thead>
<tr>
<th>Guideline</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design as an Artifact</td>
<td>Design-science research must produce a viable artifact in the form of a construct, a model, a method, or an instantiation.</td>
</tr>
<tr>
<td>Problem Relevance</td>
<td>The objective of design-science research is to develop technology-based solutions to important and relevant business problems.</td>
</tr>
<tr>
<td>Design Evaluation</td>
<td>The utility, quality, and efficacy of a design artifact must be rigorously demonstrated via well-executed evaluation methods.</td>
</tr>
<tr>
<td>Research Contributions</td>
<td>Effective design-science research must provide clear and verifiable contributions in the areas of the design artifact, design foundations, and/or design methodologies.</td>
</tr>
<tr>
<td>Research Rigor</td>
<td>Design-science research relies upon the application of rigorous methods in both the construction and evaluation of the design artifact.</td>
</tr>
<tr>
<td>Design as a Search Process</td>
<td>The search for an effective artifact requires utilizing available means to reach desired ends while satisfying laws in the problem environment.</td>
</tr>
<tr>
<td>Communication of Research</td>
<td>Design-science research must be presented effectively both to technology-oriented as well as management-oriented audiences.</td>
</tr>
</tbody>
</table>

Table 3.1: Design science research: Guidelines [Hevner et al., 2004]
tive learning is related to action research and is regarded as a method for facilitating the progressive problem solving in teams. The concept of collaborative learning is that individuals are able to achieve higher levels of learning and retain more information when they work in a group rather than individually. [Tynjälä, 2008] In order to facilitate this, each workshop will make use of the collaborative learning method *think-pair-share* [Lyman, 1987]. Think-pair-share is the basic concept of when dealing with a task, first evaluating it individually, then forming pairs or small groups and collaborating on the task to gain new insights and lastly sharing the results of the pair work with the rest of the workshop participants. This ensures that the workshops are highly participatory and that all members of the workshop get to voice their opinions. Furthermore, in the *think* and *share* parts participants are encouraged to take notes. This makes it easier to transcribe the data and results from the workshops later on.

### 3.4 Research Phases

The research is stretched out over a 6-month period as seen in Figure 3.1. The phases can be grouped into two logical categories: Group 1 represents the starting point or initial state of the company. Furthermore it encompasses the literature review and background research relating to CI/CD. Group 2 contains all phases of the study pertaining to the solution proposals, to all new elements and practices introduced into the company’s release pipeline as a result of the Current State Analysis (CSA). All research phases are presented in further detail in their corresponding chapters.

### 3.5 Environment

This section introduces the environment of the case study. The case company, its business area and product offerings are presented. This section also describes the organization, its compound and distribution as well as the core application suite. This is to better understand the problem domain and establish a frame of reference for the coming chapters. Furthermore the author’s role within the company is explained.

#### Case Company

The Button Corporation Ltd. is an IoT start-up company, that in their own words, provide ”The simplest internet user interface in the world” to their
CHAPTER 3. RESEARCH METHOD

Project Phases (2 week sprints)

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Group 1</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Current State Analysis</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Workshop (CSA)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Workshop (CNs)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exclusion criteria</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Workshop (Scope)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Group 2</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Feasibility Studies</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Solution Proposals</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Workshop (Solution proposals)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Implementation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data collection</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Workshop (Release Pipeline)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Evaluation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Workshop (Evaluation)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Final Draft</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.1: Research Phases
customers. Founded in 2013, they are based in Helsinki and have offices in both Finland and The United States. At the time of writing (6/2016) The Button Corporation employs 15 persons, 12 in Finland and 3 in The United States. Their largest customers include Taxi companies and dispatch centers from around the world, including Les Taxis Bleus and YellowCab. Other customers include digital marketing agencies and their clients, such as Lenovo, ClearChannel, Finnmkino and Pernod Ricard.

Their product offering consists of internet-connected buttons called bttns. These hardware devices are physical control interfaces to anything connected to the Internet. The company develops the hardware devices as well as the software responsible for controlling the bttns.

The organization mainly consists of a sales and marketing department and a development department. Manufacturing is outsourced, so the focus is on sales and product development. The development team is organized around the company’s products; meaning there is a two person hardware team responsible for the development of the bttn and a three person software team responsible for the my.bt.tn server application along with all other software related to the company.

The development team works in a pull-based fashion, with new requirements being generated by sales, the Product Manager, customers and management. There is no defined structure for the pull-based working practices, but on average the team deploys larger releases every 4-6 weeks. The development team has been active in trying different methods for bettering the quality of the code produced, improving work practices as well as collaboration with sales and management. The development practices are presented in greater detail in Chapter 4.

The author (Kim Rejström) works as a product developer for the company. He has been part of the software team since early 2014. His main responsibilities are front-end development, UI/UX design and graphic design throughout the company’s application suite. In addition to this he manages the company’s web store. The topic for this case study was chosen by the author, together with the instructors and supervisor, as a means to facilitate process improvement within the organization. Continuous integration provides a solid foundation to build upon, as the development teams aim to move towards shorter, more consistent release-cycles in the future.

**Application Suite**

The application suite developed by the company is directly related to the bttn, its supporting functions or to sales. The whole product offering consists of several applications and services. The main components being:
The bttn comes in 3 versions, a WiFi model, a Mobile Data model and a SIGFOX model. The different connectivity versions affect the firmware on the device but not the server. The bttn service which is responsible for the communication and control of the bttn device communicates with the physical bttns through the Event API, which abstracts away any differences in the communication protocol. The my.bt.tn configuration tool is a web application for configuring the bttns and their associated actions, as well as viewing usage statistics, event logs and connecting to 3rd party services. The admin.bt.tn management tool is an admin interface for managing users, permissions, bttns and statistics. Used exclusively by the development team. The bttn REST API provides endpoints for retrieving status information about buttons and counters. The source of status information may either be 1) a button (a physical or a virtual button) or 2) a counter. The difference from client’s perspective is that a button API endpoint is always mapped to one button, but a counter API endpoint may return status information that is a combination of one or more buttons. The bt.tn website is the company’s homepage as well as the storefront for their web shop.

The application suite effectively splits the development in to two categories: embedded software development and web software development. The teams are formed around these two categories. The two categories and their corresponding applications are heavily interdependent and interconnected adding to the complexity of their development, testing and releasing process. The main programming languages used throughout the applications are C, PHP and Javascript. The main applications are around 450K to 600K lines of code in size.

The following chapter marks the start of Group 2 in the research phase timeline shown in Figure 3.1. The rest of this thesis will focus on the tasks related to producing the updated release pipeline for The Button Corporation.
Chapter 4

Current State Analysis

This chapter describes the starting point for the study, a current state analysis which aims to thoroughly explain the development process in-use at The Button Corporation Ltd prior to this study. We examine the then-current process through the elements of the modern software development release pipeline presented in chapter 2. Furthermore the current release pipeline is presented using the extended Ståhl and Bosch notation for modeling software integration flows [Ståhl and Bosch, 2014a].

The solution proposals presented in chapter 5 as well as the scope of the accompanying implementations carried out as part of this study are based on this analysis.

The material for the Current State Analysis (CSA) was gathered through a workshop organized by the author. The workshop methods used throughout the study were presented in detail in chapter 3. An outline for the analysis was used to facilitate discussion within all relevant areas of the development process. This outline grouped the activities related to modern software development into the following areas: Requirements, Development, Operations, Testing and Quality. This categorization is based on a thematic analysis of common tool-chains within software organizations.[Mäkinen et al., 2016] As seen in Figure 4.1 each area has a set of related key activities. These key activities acted as discussion topics in the workshop. The CSA results, presented in the next section, are grouped according to this outline. Some of these activities extend beyond the scope of CI/CD, but were included in the CSA to achieve a holistic view of the situation and provide context for the identified problems.

The workshop included the CTO, both the hardware and software development teams, as well as the PO. The workshop was organized as a collaborative problem solving session utilizing the think-pair-share method. Each key activity presented in Figure 4.1 acted as an open question or discussion
In the next section we look closer at results of the workshop. How the company handled the activities associated with modern software development, as well as what problems were identified with the current process, through the context of Requirements, Development, Operations, Testing and Quality. The findings presented in the next sections are direct results from the workshop and describe the development activities as they were defined by the workshop participants. Quotes included in the following sections were gathered from notes, post-its and whiteboard scribbles produced during the workshop. Workshop artifacts of this kind will be referenced, throughout the thesis, using the following notation:

"This is a post-it note quote", "So is this!"

4.1 Requirements

Requirements drive and guide the development. Whether the organization works in a pull- or push-based fashion, access to clear requirements is guar-
Requirements Elicitation

There was no defined process for requirements elicitation in use at the company. Varying requirements came from several different channels: large business critical features were discussed and agreed upon in company wide workshops, forming the basis for the roadmap. These requirements were often large and complex and would guide the development effort for the next quarter. Often smaller but time-wise more critical requirements would come directly from the sales department, especially ”show-stopper” features that were needed to close deals were frequent and prioritized highly. This often resulted in disruptive changes to schedules and timetables as new requirements superseded ongoing ones. End-user feedback and customer demos were another source for requirements elicitation. Often end-users, re-sellers or their customers would bring forth use-cases which required new features or changes to existing features in the system. These requirements were evaluated based on how well they aligned with the roadmap features and their potential affect on revenue. In most cases customer demos were created as prototypes of new features, creating a minimum viable product, testing it with the customer and if successful made part of the product.

Backlog Management

"There is no backlog", "Too many tools no-one uses"

Backlog management was kept on a high-level, showing proposed features for the next quarter, next year and ”down-the-line”. The granularity of these feature were very large as can be seen in Table 4.1. The roadmap features were the responsibility of the Project Manager and were maintained using Excel.

Finer grained development tasks were planned and discussed within the development team during weekly planning meetings. These tasks were not documented in any way. As described in the previous section, new requirements were frequent leading to the development team adopting a pull-based way of working. The high-level features in the roadmap guiding overall development with new (often smaller) requirements superseding them frequently.

Most of the backlog item details were implicit knowledge within the company, often not documented in any other way than shown in Table 4.1. No unified backlog management tool was used leading to requirements coming
CHAPTER 4. CURRENT STATE ANALYSIS

from different channels ending up in different tools. Sales and marketing was using Asana¹ for managing ongoing projects and related development requirements were often also added there. Customer generated requirements were handled through the customer support channel Freshdesk² leading to some development requirements ending up there. In addition requirements were frequently communicated using Slack³ and face-to-face discussions. This lead to a situation were a lot of critical information was implicit knowledge within the organization.

Bug Tracking

Bug tracking is closely related to backlog management as bugs are special form of development task, but a development task nonetheless. There was no dedicated bug tracking tool in use at the company, with bugs being reported through the same channels as other development tasks.

4.2 Development

Development work is focused on the writing, managing and testing of code. Development focuses on the work done by a developer in his local devel-

Table 4.1: Backlog Management

1https://asana.com/
2https://freshdesk.com/
3https://slack.com/
opment environment with emphasis on version control, build systems and continuous integration.[Mäkinen et al., 2016]

Version Control

"Branching", "git vs. hg"

A distributed version control system has been used within the company since the start of development. The software in use - RhodeCode\(^4\); an on-premise source code management tool for Mercurial, Git and Subversion hosted all code repositories on the company’s internal network.

Most repositories were using Mercurial for version control, although Git repositories were also used. There were 10 repositories actively receiving new commits, although there were only 3 distinct code bases: bttn server, bttn fw and bt.tn website. Discrepancies between the branching logic used within repositories, as well as when new repositories were created were present. The bt.tn website repository was using the Git Flow\(^5\) feature-driven branching model, whereas the bttn server repository used a single master branch. The bttn fw repository was using a repository based branching model where each developer had their own repository, in addition to a development repository and a release repository.

Commit frequency was on average 2-3 commits per developer per day. The teams strove for high commit frequencies to keep integration problems within the same code base low. The commit history for the largest and most active repository bttn server can be seen in Figure 4.3.

Build Systems

The organization had several different build systems in use, as their products differed in programming language, deployment method and production environment. The build systems were built around makefiles, proprietary shell scripts and task-runners. All builds had to be manually triggered, but all builds had accompanying scripts or makefiles to run all necessary steps of the build.

The bttn server repository contained several CodeIgniter\(^6\) (a lightweight PHP framework) projects. The code base was setup so that all projects shared the common CodeIgniter core components, which were compiled to full-featured working projects at the build stage. In addition to the makefile

\(^4\)https://rhodecode.com/
\(^5\)http://nvie.com/posts/a-successful-git-branching-model/
\(^6\)https://www.codeigniter.com/
that handled the builds, there were shell scripts for database initialization, deployment package creation and documentation generation. The repository also contained the code for bttn service which was written in C, as well as the front-end code written in Javascript and SASS.

All web-based projects with a GUI utilized the same build scripts for compiling the front-end assets. Grunt.js\(^7\) a Javascript task runner was used to compile and minify assets, optimize images check coding standards, run tests and create production versions of the GUI.

**Continuous Integration**

"No resources", "Hire more people?"

No CI system was used within the organization. Build tasks, tests and deployment tasks were carried out manually by the developers at appropriate times.

### 4.3 Operations

Operations is about managing technical infrastructure and thus requires attention to be paid to provisioning and deployment tools.[Mäkinen et al., 2016]

**Provisioning**

"Vagrant", "Configuration management, Docker?"

There was no standard development environment used within the organization. The developers were free to choose any suitable environment for development. Varying provisioning solutions, ranging from no configuration management to automatic provisioning of new production servers, were in use. Each product (or code repository) had its own way of handling provisioning. Vagrant\(^8\), a tool for building and easily cloning complete development environments was used by some developers, but there were no standard Vagrant boxes in use.

Although not enforced to be identical clones, staging servers and preferably also local development environments were required to mimic the production servers. Keeping the environments in sync was a manual task.

---

7[http://gruntjs.com/](http://gruntjs.com/)
8[https://www.vagrantup.com/](https://www.vagrantup.com/)
Deployment

Deployment tools were not used within the organization. All of the production servers were hosted on Amazon EC2 instances. Deployment to production was carried out manually, utilizing custom shell scripts to do the heavy lifting. Exact deployment procedure varied from product to product on the software side.

Deployment of the bttn Firmware through OTA (Over-the-air) updates followed the same procedures as on the server side, manually deploying the new Firmware version to Amazon.

4.4 Testing

"No tests”, ”4 distinct testing needs”

Testing is a fundamental part of modern software development. It splits up in to unit testing, UI testing, acceptance testing and performance testing.\[\text{Mäkinen et al., 2016}\]

Unit Testing

Unit tests were used exclusively within the low-level parts of the software. Meaning only C-code had unit tests. The entire web application built on a PHP back-end and a JS front-end were without unit tests. There were no test-runners in use, so all tests were triggered manually. Unit test reporting was confined to the environment it was run in.

UI Testing

"Too complicated"

UI testing was completely manual, with focus on compatibility only with modern browsers (Chrome and Firefox). All GUIs were sporadically tested for correct responsive behavior on smaller screens. Testing on handhelds was exclusively done through device virtualization.

Acceptance Testing

Acceptance testing was done exclusively within the organization. End-users were not directly involved in the release cycle. There were no tools in place for the testing. Acceptance testing was seen as the last tests to decide if the
system was ready to be deployed. To achieve the level of certainty needed, the acceptance testing was done during a code-freeze where no new commits were allowed to the repository. Criteria for the acceptance testing varied from development team decisions, product owner decisions and manual testing to management approval.

There were no concrete acceptance metrics in use, instead an ad-hoc process of manual "sanity checking" that things work was used for each release.

**Performance Testing**

Performance testing was not part of the development process within the organization. There were no explicit performance requirements defined and no scheduled performance tests. Testing was carried out on smaller components of the system when deemed necessary.

"Live-server health checker", "Monitor"

Both on the *bttn server* and *bttn website* actual production loads on the deployed software had unearthed performance related bugs in production.

### 4.5 Quality

Quality can be seen as the end result of a good development process. Each step supports the notion of quality-built-in and there is no need to add quality to the developed software at a later stage. In this section we look closer at *code quality* as well as *code review*, two practices not directly included in the previous sections.[Mäkinen et al., 2016]

**Code Quality**

The company was not using any linters in their development process. If linting was done it was carried out by the various IDEs the developers were using. No code style nor coding standard was defined as a company standard for any of the development languages. There was also no commenting standard in use within the organization.

This meant indentations, spaces, variable declarations etc. were not standardized and the code style varied based on IDE specific settings as style was not checked by any build job.
CHAPTER 4. CURRENT STATE ANALYSIS

38

Code Review

There was a practice within the organization to do sporadic code reviewing. Most developers went through commits to see what changes had been made by the others. There was however no formal code review, which meant no pull request process was used. The reviewed code was thus already committed to the master branch at the time it was reviewed.

Code ownership was not defined explicitly, although implicit ownership existed within some parts of the system. Although several parts of the system were subject to collaborative efforts code ownership within the organization was closer to individual owners than collective ownership. This also meant testing responsibility was unclear, which contributed to the lack of testing overall.

The CSA outlined above extends beyond the scope of the new release pipeline. The slightly larger scope of this workshop was deemed valuable for the company, and included in this chapter to provide some context to the problems not related to CI/CD although present within the company. In the next section the then-current release pipeline is visualized.

4.6 Original Release pipeline

This section aims to establish a reference point for the study; an accurate visualization of the release pipeline before any changes were introduced. A rough version of this model was created during the CSA workshop, and later re-created for this paper by the author using Ståhl & Bosch notation.

Integration Flow Model

As seen in Figure 4.2 the current release pipeline is fully manual. All trigger edges are derived from the external trigger node 'Manual'. Only input edges connect the activity nodes to each other. This means that data flows through the model from the VCS to the production server by manual invocation at every step. The release pipeline has two main branches, the hotfix branch and roadmap branch.

All change-sets pass through one of the branches on their way to production. Most changes are small, incremental updates to existing features - these changes are deployed without extensive testing, code freezes or blocking of the master branch, through the hotfix release path. The live system may receive updates several times per day this way.
CHAPTER 4. CURRENT STATE ANALYSIS

Figure 4.2: Current release pipeline (Ståhl & Bosch notation).
Changes going through the roadmap path are more extensive, potentially braking changes and new features. These are features documented in the roadmap, that require more extensive work to complete. These changes are accompanied by a code freeze and the blocking of the master branch. No new changes are allowed to be pushed to the repository during the code freeze. After the code freeze, manual testing is started. Testing is carried out on the staging server: it involves manually testing the new feature, testing old features, UI testing and performance testing. The testing phase varies from 2-5 days. The live system receives updates from the roadmap path about once per month.

Release cycle

A normal release cycle is about one month in calendar-days. Time-boxing is not used within the company, so the actual release cycle lengths vary. During this one month period, changes are pushed to production through both paths (hotfix and roadmap). As seen in Figure 4.3 the large commit peak happening about once per month is due to the compound effect of finishing and testing a new feature, together with the pile-up of unrelated commits pending addition due to the blocked master branch. When the blocked repository is re-opened all pending commits are added to the VCS. This often leads to some integration problems in the form of merge conflicts.

"RC testing too slow", "No planning of tasks"

Based on the current release pipeline it was obvious that there was room for improvement. As seen in Figure 4.2 most of the trigger edges coming from the ‘Manual’ node should be moving from one activity node to the next in the same way as the input edges. Furthermore, a more automated pipeline would harmonize the commit graph seen in Figure 4.3 by enabling more frequent, reliable changes.

Figure 4.3: Commit statistics from bttn repository.
To achieve something actionable and concrete based on the CSA, a second workshop was organized where the CSA acted as input for the next phase of the study. The results of that workshop is presented next.

4.7 Company Needs

Based on the CSA results a second workshop was held to elicitate the company needs and form the scope of the study. The workshop consisted of the same persons as in the CSA workshop. At the start of the workshop the results of the CSA-workshop were presented. The results consisted of a cleaned up and organized version of the things discussed at the CSA-workshop. Meaning the practices and identified problems relating to the key activities in-use within the company grouped and organized in to: Requirements, Development, Operations, Testing and Quality. The goal of the workshop was to have these key activities act as themes for identifying the improvement areas related to CI/CD. After going over the results of the CSA-workshop, the participants formed smaller discussion groups with the goal of listing the most crucial needs related to the company’s release pipeline and testing practices. The discussion groups were as multi-disciplinary as possible, mixing hardware and software development teams with product management. Each discussion group was assigned two areas to analyze. The results were then discussed and finalized together with the other discussion groups. In the following section the identified needs are presented.

As stated in the research question, the purpose of this study is to provide the customer with both implementations and solution proposals for the identified crucial needs related to the company’s release pipeline. It is however worth noting that the CSA was conducted with a slightly larger scope, addressing issues outside CI/CD related practices. Some of the company needs presented in this section are thus also not directly related to CI/CD.

The answer to RQ1: What are the motivation factors driving the adoption of CI/CD practices within the company? is presented in Table 4.2.

There were severe problems identified that need addressing within all five categories examined in the CSA. These problems are concretized as 19 company needs (see Table 4.2) as elicited in the workshop. Together with the company representatives the scope of the study has been limited to address only a few of these needs.

The exclusion criteria were formed by the author and company instructor based on the recommendations of the supervisor. Each CN was evaluated from two viewpoints: Does it fit in with the topic of the case-study? and
CHAPTER 4. CURRENT STATE ANALYSIS

Company Needs

<table>
<thead>
<tr>
<th>Description</th>
<th>ID</th>
<th>Cat.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Define process for Backlog Management and setup supporting tools</td>
<td>CN1</td>
<td>1</td>
</tr>
<tr>
<td>Define process for Bug Tracking and setup supporting tools</td>
<td>CN2</td>
<td>1</td>
</tr>
<tr>
<td>Define branching logic</td>
<td>CN3</td>
<td>2</td>
</tr>
<tr>
<td>Define process for code review</td>
<td>CN4</td>
<td>2</td>
</tr>
<tr>
<td>Define code ownership</td>
<td>CN5</td>
<td>2</td>
</tr>
<tr>
<td>Create build scripts for CI server</td>
<td>CN6</td>
<td>3</td>
</tr>
<tr>
<td>Define process for creating FW binaries</td>
<td>CN7</td>
<td>3</td>
</tr>
<tr>
<td>Create CI Server for testing, building, reporting</td>
<td>CN8</td>
<td>1</td>
</tr>
<tr>
<td>Feasibility of artifact repository for built binaries</td>
<td>CN9</td>
<td>2</td>
</tr>
<tr>
<td>Feasibility of Configuration Management</td>
<td>CN10</td>
<td>3</td>
</tr>
<tr>
<td>Create automated FW OTA update process</td>
<td>CN11</td>
<td>3</td>
</tr>
<tr>
<td>Define unit testing process and implement test frameworks for use on CI server</td>
<td>CN12</td>
<td>2</td>
</tr>
<tr>
<td>Implement UI testing framework for responsive testing and browser specific testing</td>
<td>CN13</td>
<td>3</td>
</tr>
<tr>
<td>Define recipe checklist for manual acceptance testing of deployment-ready-software created by release pipeline</td>
<td>CN14</td>
<td>3</td>
</tr>
<tr>
<td>Define critical performance tests (db queries, traffic spikes etc.) to be run as nightly performance testing</td>
<td>CN15</td>
<td>3</td>
</tr>
<tr>
<td>Create automated alert system that periodically checks that mail servers, databases etc is running as they should in the production environment</td>
<td>CN16</td>
<td>3</td>
</tr>
<tr>
<td>Create testing environment for testing physical buttons against a “identical to production” sandbox server</td>
<td>CN17</td>
<td>3</td>
</tr>
<tr>
<td>Define code style specifications and coding standard</td>
<td>CN18</td>
<td>1</td>
</tr>
<tr>
<td>Implement linters into development workflow and CI</td>
<td>CN19</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 4.2: Company needs
How important is it for the company?. The idea was to keep the scope narrow enough to enable both an implementation and analysis phase of the core processes involved in CI, while also addressing the most important CNs from the company’s viewpoint. Because of discrepancies between these two viewpoints the categorization and prioritization of CNs presented here correspond only to this case-study, not to the over-all prioritization of the CNs within the company. In the following section the identified company needs are presented and the extent of their inclusion in the study is defined.

It is worth noting that due to limitations regarding virtualization possibilities for the bttn hardware, the embedded software side requires physical on-premise testing equipment, which falls outside of the software release pipeline for the rest of the products within the company and was thus deemed to be out of scope for this study. That being said all proposed solutions, where it was relevant, considered the feasibility of the solution on the embedded software development side. This was due to the fact that many of the solutions are directly applicable on the embedded software side as well.

The scope of the study was thus limited to a cloud-based release pipeline with implementations and solution proposals for the needs related to bttn server and bttn website product suite. Furthermore the inclusion of requirements and code quality related CNs in the scope of the study although strictly not related to CI/CD practices are motivated by their importance to the company along with their close tie-ins with the CI system.

Each of the 19 company needs have been categorized according to the following three categories:

- Category 1: Implementation
- Category 2: Solution proposal
- Category 3: Out of scope

Categories 1 and 2 both include a feasibility study with an accompanying solution proposal. Category 1 needs were in addition to this implemented as part of the study. Category 3 items were deemed out of scope and were not studied further. The categorization was done based on the exclusion criterial mentioned earlier. Category 1 needs were the ones that fit well with the topic of the case-study and were of high importance to the company. Category 2 needs either fit well with the topic and were of moderate importance to the company, or of high importance but too large to implement within the scope of the study. Category 3 needs were of low importance or too far from the topic of the study to justify inclusion. For a full overview of the identified crucial needs and which needs were included see Table 4.2. The included CNs
Results

In this section the company needs included in the study (Category 1 and Category 2) are presented in more detail. Category 1 classified needs were fully implemented meaning there was a mapping of available solutions made, followed by a decision and implementation of the identified optimal solution. Category 2 needs were assessed and a solution proposal was made but not implemented.

Requirements

Define process for Backlog Management and setup supporting tools: **Category 1.** Identify on what granularity level the requirements (backlog items, non-functional requirements) need to be presented. Define a requirements elicitation / specification process suitable for the company and setup an appropriate tool for managing the backlog. Backlog management is not within the scope of CI/CD but related to the release pipeline nonetheless. Scheduling releases, managing version numbers and deployment decisions all have close tie-ins with the CI system. Together with the high priority ranking from the company, a Category 1 classification was assigned to this CN.

Define process for Bug Tracking and setup supporting tools: **Category 1.** Specify what kind of bugs, error reports and malfunctions should be handled by which department, how they should be tracked and how they should be integrated in to the development process. Setup necessary tools for bug tracking. Bug tracking is also not a CI/CD activity, although integrations with VCSs and the widespread use of commit hooks to manage issues and bugs land it in close proximity to CI/CD activities.

Development

Define branching logic: **Category 2.** Define a standardized way of handling branching in the version control system that is the same for all repositories. The branching model also needs to support CI build triggering and eliminate large commits and work blockages. At the heart of the CI pipeline as branching logic orchestrates the triggering of the CI system.

Define process for code review: **Category 2.** To support the branching logic,
a formal process for code reviewing needs to be created. The combination of CI build triggering and pull request acceptance needs to be considered in the code review solution proposal. A necessary part of a more automated release pipeline, code reviewing ensures only quality code makes it way downstream.

Define code ownership: **Category 2.** To enable the code review process, code ownership must be defined. Decide whether to use collective, individual or other form of code ownership. This is also related to how testing and the CI system is viewed: will there be a tester that handles QC or is it a shared responsibility?

Create CI Server for testing, building, reporting: **Category 1.** Define what jobs the CI server needs to carry out, identify the optimal CI solution - set it up and configure it. This is the single largest implementation task of this study. The results for each CI area (testing, building, reporting) is presented in detail in chapter 7.

Feasibility of artifact repository for built binaries: **Category 2.** All build process in use within the company produce build artifacts. These artifacts could potentially be used in a Continuous Delivery / Deployment pipeline as well as in acceptance and performance testing in the CI environment.

**Testing**

Define unit testing process and implement test frameworks for use on CI server: **Category 2.** Setup a process for creating and running unit tests. The process should be defined in a way that ensures the necessary creation of tests. Implement and configure the optimal test runners and testing frameworks that allow the CI system to carry out the same tests automatically on the CI Server.

**Quality**

Define code style specifications and coding standard: **Category 1.** In addition to choosing the styles and standards the configuration files need to be set up for the IDEs to enforce the chosen coding standards and code styles. Code styles are not within the context of CI/CD, although code quality and static code analysis are concepts related to CI. Classified as a low-hanging fruit CN with immediate returns it was included in the study.

Implement linters into development workflow and CI: **Category 1.** Config-
ure linters to test that the code follows the coding standards and code styles. Linters provide reporting on the health of the code, and are run in concession with automated unit tests.

The first two workshops (the CSA and CN) have formed the basis for the research in this study. In this chapter the problem domain has been defined and RQ1 has been addressed through the elicitation of the CNs. The next step is to start forming the solution proposals and prepare their implementation. This is addressed in the following chapter.
Chapter 5

Solution Proposals

In this chapter the solution proposals for the CNs included in this study (CN1, CN2, CN3, CN4, CN5, CN8, CN12, CN18, CN19) are presented. Each solution proposal is based on a feasibility study to determine the optimal solution or tool for the company. The feasibility studies were done by combining industry best practices from existing literature with the concrete needs of the company. Many of the solutions presented in this study are based on comparing and choosing suitable tool-chains or application stacks, from ready-made available software. This means there is no absolute truth, no correct tool-chain solution, only opinionated answers.

Furthermore it is worth noting that most solution proposals could have been implemented with other tools or methods and still achieved the same results. As a basis for the proposals we considered the following feasibility factors: price, setup and configuration effort, ease-of-use, integration capabilities between the selected tools, framework suitability and maintainability in respect to current code-base.

The following sections present the results of two separate feasibility studies. The first one focused on the CNs related to requirements, development and testing, while the second one focused on the CNs related to quality. The first one represents the core of the new release pipeline, while the second one provides supporting functions for improving quality built-in through the CI system.

5.1 Feasibility study: Core tool-chain

The core tools needed to deal with the CNs in Requirements (CN1, CN2), Development (CN3, CN4, CN5, CN8) and Testing (CN12) can be grouped in to a 3-part tool-chain consisting of a Version Control System, a Continu-
uous Integration Server and a Backlog Management Tool. The tool-chain must in addition to the feasibility factors mentioned also support CN18, CN19 directly or through extensions. All proposed solutions must enable implementation of the best practices identified in existing literature. Ideally they should also mitigate the risks related to common challenges in CI/CD adoption presented in chapter 2.

The proposed tool-chains can roughly be divided into four categories:

- Category 1: Cloud-hosted Business-grade
- Category 2: On-premise Community-grade
- Category 3: On-premise Business-grade
- Category 4: On-premise Enterprise-grade

As pricing was an important factor, Category 4 was completely excluded from the proposals after reviewing the pricing models of the enterprise versions of Github, Bitbucket, Bamboo, Travis CI, Zenhub and JIRA. Each piece of software in the Enterprise tier was priced in the thousands and largely exceeded the budget for this project.

It is worth noting that discrepancies in naming conventions were common, with many tools naming their paid solution *Enterprise Edition* yet pricing it as the more commonly used *Business or Growing Team* tier of the competitors. Both RhodeCode EE and Gitlab EE correspond to the middle tier pricing of Github and Atlassian.

Moreover, *cloud-hosted* is here defined as a SaaS solution where the software and hardware is maintained by an entity external to the company. In this case this means utilizing github.com or gitlab.com directly. *On-premise* refers to any solution where the software and hardware is maintained from within the company. This includes both actual on-premise solutions where the servers are physically located in the office as well as cloud-infrastructure solutions such as Amazon Web Services (AWS), where the software is technically cloud-hosted but under the full control of the company.

Based on this categorization, four tool-chain candidates were identified. The grouping of tools is not absolute, and other mixes than those presented in Table 5.1 were also considered. The groupings presented here aim for a consistent and working solution with as few integration problems as possible. However most of the VCS, backlog tools and CI servers can be viewed as interchangeable components within the presented tool-chains. These tools can be considered industry leaders within CI/CD and as such displayed very similar feature-sets. Not all tools however, actively enabled all the best practices from existing literature, although none of the candidates explicitly hindered them either. The largest discrepancies were related to monitoring a
CHAPTER 5. SOLUTION PROPOSALS

<table>
<thead>
<tr>
<th>Tool-chain Comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td>Category</td>
</tr>
<tr>
<td>Cloud-hosted Business-grade</td>
</tr>
<tr>
<td>VC System</td>
</tr>
<tr>
<td>CI Server</td>
</tr>
<tr>
<td>Backlog</td>
</tr>
<tr>
<td>Popularity</td>
</tr>
<tr>
<td>Documentation</td>
</tr>
<tr>
<td>Integrations</td>
</tr>
<tr>
<td>Version Control</td>
</tr>
<tr>
<td>Issue tracker</td>
</tr>
<tr>
<td>Code Review</td>
</tr>
<tr>
<td>Pull Requests</td>
</tr>
<tr>
<td>Wiki</td>
</tr>
<tr>
<td>CI reporting</td>
</tr>
<tr>
<td>Hosting</td>
</tr>
<tr>
<td>Price</td>
</tr>
</tbody>
</table>

Table 5.1: Proposed Toolchains

configuration management. Everyone can see what’s happening [Humble and Farley, 2010] and test in a clone of the production environment [Fowler, 2013] require dedicated tools often not included in these tool-chains. The solution proposals that did not provide built-in support for these were judged on their integration possibilities with third-party alternatives.

In Table 5.1 the results of the feasibility study for the core tools of the release pipeline are presented. A workshop was held were the results of the feasibility study were discussed with the team. The development team, together with the COO and PO went over the pros and cons of each tool-chain, arriving at a decision to go forward with Tool-chain 3 as it was deemed the most suitable alternative. The main pros being price, a unified UI with
all parts of the tool-chain available through the same interface as well as support for Google Apps accounts and easily configurable hosting.

5.2 Solution: Tool-chain 3 – The Gitlab Suite

Gitlab Enterprise Edition hosted on Amazon AWS. The Gitlab suite provides a unified, one-stop place for Version Control, Project Management, Bug Tracking, Code Review, Continuous Integration and Wikis. It is profiled as a self-hosted alternative to github.com, with a feature rich VCS packed with supporting tools. In addition to the VCS, Gitlab have created their own CI server, Gitlab CI. Gitlab CI is not directly part of the VCS but fully integrates with it out-of-the-box. The VCS is however CI agnostic, and the CI server can easily be replaced with ie. Jenkins or Bamboo.

This solution gives the company the possibility to bring together all the development tools under one roof and utilize centralized user management at the same time. Instead of having 5-6 different tools, all with their own user management systems, the existing bt.tn Google Apps accounts used for e-mail, file sharing and calendar tools can be leveraged within the development work as well. As Gitlab supports OmniAuth, Google logins work out-of-the-box.

As a frame of reference for the in-depth presentation of the new tool-chain, the old tool-chain in use at the company, prior to this study, consisted of:

- RhodeCode (VCS) with its own user management
- Trac (Wiki) with its own user management
- Excel, Pivotal Tracker, Asana (Project Management) all with their own user management

There is nothing inherently wrong with the tools in the above tool-chain. They were chosen at the start of development because of previous familiarity with the products. However, as the company grew, the needs changed and these tools no longer provide the best way of addressing those needs.

In the following sections each part of the new tool-chain is presented in more detail, followed by a review of the tools related to code quality. The last section in the chapter is a discussion related to the identified possible problems and pitfalls that pose a threat to the implementation and adaption of this specific tool-chain.
Version Control

Gitlab’s VCS comes bundled with a set of features addressing the CNs. In addition to the standard activity log and file browser included in all VCS systems, Gitlab has a graph module for in-depth statistics and reports about the repository. It also has version and tag handling for easy maintenance of releases. As opposed to the Rhodecode VCS currently in use, Gitlab only supports git repositories. Currently most repositories are mercurial repositories within the company. To account for this the fast-export\textsuperscript{1} tool is proposed for migrating the mercurial repositories to git. All branches, history and tags are kept intact, so no information loss should occur. The tool was successfully tested on the largest of the company repositories without any issues. In addition to the source code the repositories should in the future also include tests, CI config files as well as .editorconfig and possible linter config files to comply with the CNs.

Project Management and Bug Tracking

Gitlab’s VCS includes an issue tracker capable of both road-mapping and bug tracking. With the issue tracker it is possible to maintain a project backlog as well as plan and schedule releases. As the issue tracker is built around the concept of tags and milestones both upcoming features as well as known issues and bugs can be tracked through the same system. It is possible to assign weights and deadlines to the issues as well as group them in to milestones for release planning. Gitlab offers an easy way to connect development with project management as there is full integration support between the VCS and the issue tracker. It is possible to automatically reference, update and close issues based on commit messages. Using any of the defined trigger words within the commit message updates the status of the referenced issue. Furthermore the issue tracker supports discussions regarding the issues directly within the issue tracker UI. In the future all relevant information regarding an issue should be added to the issue description or in the related discussion.

Code Review

To address CN3, CN4, CN5 as well as CN18 and CN19 a process for code reviewing needs to be present in the development workflow. Gitlab has support for pull request (PR), also referred to as merge request, driven code reviewing. The major benefits of PR driven code reviews are the introduction of

\textsuperscript{1}https://github.com/frej/fast-export
collective code ownership, peer reviewing and direct integration with the CI server. Code reviews ensure that only high quality working code is merged to the master branch, thus keeping the code in the main branch always deployable according to CD principles. Code reviews are directly related to the best practice of added transparency as everyone can see what’s happening [Humble and Farley, 2010]. The way PR driven code reviews work are quite straightforward: at the completion of a new feature, instead of directly pushing the changes to the master branch of the repository a code review process is initiated. During this process the software is built and tested by the CI system, the source code is reviewed and discussed by the development team, any identified problems are fixed and re-reviewed before finally accepting the PR and merging the reviewed changes with the master branch.

For PR driven CI triggering and code reviewing to work a defined branching model needs to be in place. Gitlab is branching agnostic, but has built-in support for the popular git-flow\(^2\) and github-flow\(^3\) branching models. These models define branching flows aiming to provide clean repository structures with easily readable timelines and always deployable master branches. To harmonize the branching models, or lack thereof, within the company the Github-flow model is suggested as starting point for handling branching consistently in the future.

Github-flow is a simple master-feature branching model that emphasizes short-lived branches and frequent commits. Merging everything into the master branch and deploying often means minimizing the amount of code in ”inventory” which is in line with CD best practices. A typical development workflow, utilizing the github-flow branching model, would be the following:

- Create new feature branch
- Develop the feature
- Create unit tests for the feature
- Run (at least) the new tests locally
- Create PR
- CI Server runs entire test suite and all code checks, then reports back
- In case no problems were found, code is reviewed by another team member
- Fix possible issues found in review
- In case no problems were found, a team member accepts the PR

\(^2\)http://nvie.com/posts/a-successful-git-branching-model/
\(^3\)https://guides.github.com/introduction/flow/index.html
• The feature branch is merged into the master branch
• The feature branch is automatically closed and optionally deleted after the PR is accepted
• Related issues in the Issue Tracker are automatically closed based on the commit or PR messages
• The master branch has thus been updated with the latest changes and is ready for deploying

**Continuous Integration**

Gitlab’s CI is integrated with the VCS, thus enabling testing, building and deploying of code through the Gitlab UI. The integration means that the CI is configured, managed and monitored through the Gitlab interface. It also provides views for reporting on the status of ongoing and completed jobs and provides an overview of all configured pipelines and jobs through the Gitlab UI. It automatically checks the box for the following best practices from chapter 2: *automate the build, make your build self-testing, every commit should build on an integration machine, everyone can see what’s happening and automate deployment.* Furthermore it encourages the practices of *keep the build fast, keep the build green* and *test in a clone of the production environment* although not strictly enforcing them. Through monitoring and reporting of the CI pipelines and jobs developers are informed of the build speeds, build successes and statuses of the builds.

Gitlab CI is a full-featured and scalable, self-hosted continuous integration server. The main features are:

• Multi-platform: execute builds on Unix, Windows, OSX, and any other platform that supports Go
• Multi-language: build scripts are command line driven and work with Java, PHP, Ruby, C, and any other language
• Stable: builds run on different machines than the GitLab instance
• Parallel builds: GitLab CI can split builds over multiple machines
• Realtime logging: integration with PRs and code reviews for dynamically updated reporting
• Pipeline: define multiple jobs per stage and programmatically trigger other builds
• Build artifacts: automatically upload binaries and other build artifacts to GitLab after successful builds
• Test locally: reproduce tests locally

Architecture. Gitlab CI is part of the Gitlab VCS, it acts as a configuration and reporting tool only. All tests are handled by the Gitlab Runners. These are separately deployed build processors that communicate with Gitlab CI through an API. The Gitlab runners are responsible for setting up the test environments and actually running the tests. As seen in Figure 5.1 the runners can be distributed over several servers, each running 0...n runner processes. This way the main Gitlab instance is not affected by heavy test suites and the CI capabilities can easily be scaled to match the testing demands.

This architecture addresses the challenges of complex software and environment management [Leppänen et al., 2015] as the modularized CI runner setup allows dealing with several operating systems and repository specific environments through the use of dedicated runners.

Concepts. The CI server uses a.gitlab-ci.yml configuration file\(^4\). The.gitlab-ci.yml file is where you setup your pipelines. A pipeline is a group of builds that get executed in three stages: build, test, and deploy. All of

\(^4\)http://docs.gitlab.com/ce/ci/yaml/README.html
the builds in a stage are executed in parallel (if there are enough concurrent runners), and if they all succeed, the pipeline moves on to the next stage. If one of the builds fails, the next stage is not executed and the pipeline is halted. Builds are individual runs of jobs. Job is defined by a list of parameters that define the build behavior. This hierarchical structure of jobs, in stages, in pipelines is defined in the .gitlab-ci.yml configuration file. This file should be added to the root directory of the repository. This way the CI configuration is also under version control and easily maintainable.

Wiki

Gitlab also comes with a Wiki feature. Wiki is a separate system for documentation, built into GitLab. It is a source controlled alternative to the current Trac system currently in use in the company. To reduce the number of separate systems and accounts within the company a transition from Trac to Gitlab Wiki is proposed. Gitlab Wiki has the possibility to completely replace Trac without any loss of features. There is tools available for handling the migration: Trac to Gitlab migration converts all documentation written in WikiFormatting to the more widely used Markdown format supported by Gitlab Wiki. The tools was successfully tested on the existing Trac documentation without any issues.

The Gitlab suite appears to be a very good fit for the company. It provides concrete solutions to the CNs while promoting industry best practices. It also adds some additional benefits in the form of centralizing services, a welcome streamlining of processes that was not actually part of the CNs. The process of configuring and implementing the Gitlab suite in to the new release pipeline is described in the following chapter. Before that, the second feasibility study, targeting code quality related CNs is presented.

5.3 Feasibility study: Code quality

The solutions addressing the CNs in Quality (CN18, CN19) pertain to coding standards, code styles, linters and formatters. In addition to supporting the frameworks in use, the tools also needed to be compatible with the toolchain presented in section 5.1. These are mostly framework dependent tools with heavily opinionated best practices and rulesets. The goal of this feasibility study was to find the tools necessary to enable monitoring of code

---

5https://trac.edgewall.org/
6http://eric.van-der-vlist.com/blog/2013/11/06/from-trac-to-gitlab/
quality within the company. Enforcing very strict rules and standards was not deemed necessary nor optimal, instead an easily configurable code health checker was preferred. The idea is to automatically review the checked-in code for syntax, security, style and other problems to be able to merge with confidence.

That means, at the very least, that the checks need to be carried out on the CI server after every commit. However, to minimize failed CI builds it is preferable that the checks can be ran locally as well. The health checker consists of three parts: linters, formatters (or beautifiers) and configuration files. The motivation for adding static code analysis to the release pipeline is to improve software quality [Leppänen et al., 2015] and mitigate risks related to code smells and bad coding practices as outlined in chapter 2. Next the three parts of the health checker are presented.

**Configuration files.** These are very common files in any computing context. Configuration files provide initial settings or parameters for a given application. Config files are often prefixed with a dot, making them hidden files, and thus often referred to as ‘dotfiles’. In the context of code styles and coding standards these files define rules for the structure and format of files.

**Linters.** Linters perform static analysis of the source code. Originally lint was the name of a program that flagged suspicious and non-portable constructs in the C language source code. The term is now applied generically to tools that flag suspicious usage in software written in any computer language. Linters check the source code against the rules and style-guides provided in the configuration files. The linter run results in a report highlighting any potential problems, errors and warnings found.

**Formatters.** Formatters and beautifiers can be seen as an extension to linters. Some linters offer beautifying as part of the tool, others make use of stand-alone formatters. The idea is that when problems are found during the linting, they can automatically be fixed using the beautifiers. Beautifying a language is not trivial, as the formatter needs to accurately identify code-blocks, control flows and various loops and conditional blocks. Beautifiers are often included in IDEs and also exist for most languages as stand-alone tools. For the purpose of easy and maintainable configuration, only beautifiers that worked with the same configuration files as the corresponding linters were considered as viable alternatives.
5.4 Solution: Full-stack code health checker for bt.tn application suite

As explained in chapter 3 the main application suite consists of a web application with a PHP back-end and Javascript, HTML5 front-end. In addition to this there are several web services built primarily in C.

The primary goal of the health checker is to catch actual syntax errors, in addition to this it should check all files for correct indentation, trailing whitespace, line-endings and other generic style-rules. These are the two general tasks performed on all non-third-party code within the repository. The health checker also performs a variety of language specific checks on C, PHP, JS and SCSS files related to code smells [Fowler and Foemmel, 2006].

All linters and formatters considered have command-line versions of the tools available, as the primary use is to run the checks on the CI server. Many of the tools reviewed are however also embedded in popular IDEs or can be installed as extensions for on-the-fly code checking. The following tools form the base of the code health checker proposed for the bt.tn application suite:

Editorconfig. EditorConfig\(^7\) helps developers define and maintain consistent coding styles between different editors and IDEs. The EditorConfig file sets global and language specific rules concerning indentation, line-endings, charsets and whitespace. Coupled with ECLint, a tool for validating and fixing code that does not adhere to settings defined in the .editorconfig files, it becomes easy to maintain consistent styles throughout the project. An .editorconfig file should be committed to the repository and all IDEs used by the developers configured to automatically read the settings from that file. In addition to that, the CI server checks all files against the .editorconfig rules on commit.

PHP Code Sniffer. PHPCS\(^8\) is a set of two PHP scripts; the main phpcs script that tokenizes PHP files to detect violations of a defined coding standard, and a second phpcbf script to automatically correct coding standard violations. PHPCS checks all php files against the PSR-2 coding standard. PSR-2 adds style guidelines on top of the PSR-1 standard. PSR-1 is the basic coding standard defined by PHP-Fig\(^9\), the PHP Framework Interop Group, designed to reduce cognitive friction when scanning code from different authors. The phpcs linter is run by the CI server on commits, no automatic fixing of files is done.

\(^7\)http://editorconfig.org/
\(^8\)https://github.com/squizlabs/PHP-CodeSniffer
\(^9\)http://www.php-fig.org/
PHP Mess Detector. PHPMD\textsuperscript{10} compliments PHPCS, as it focuses on finding possible bugs, suboptimal code, overcomplicated expressions, unused parameters, methods and properties within the code, while ignoring code style. The PHPMD checks are run in addition to the PHPCS checks by the CI server.

ECMAScript Linter. ESLint\textsuperscript{11} is an open source JavaScript linting utility. ESLint checks all .js files against a set of rules defined in the .eslintrc configuration file. Like with the other pluggable configuration files, the .eslintrc file should be committed to the code repository. The ruleset is fully configurable but as a starting point the default ESLint configuration file was used. The default configuration emphasizes penalization for complexity and disables style checks. Style checks are disabled so that PRs will not error on style choices the team has yet to decide.

SCSS Linter. SCSS-lint\textsuperscript{12} is a customizable tool with opinionated defaults that helps enforce a consistent style in SCSS files. The tool checks .scss files against a ruleset defined in the .scss-lint.yml configuration file. The SCSS-lint recommended default configuration file was used as a base for the configuration. Checks are run both locally and on the CI server in conjunction with processing the SCSS to CSS.

The solution proposal for code quality presented above differs radically in size from the tool-chain proposal. Code health checkers are essentially a configure-once use-everywhere solution, where the largest effort is spent on configuring the rule-sets in the set up phase. They provide a low-hanging fruit possibility to improve upon the software quality of the overall system. Implementation should be relatively quick and straight-forward. However, before moving on to the implementation phase, the next section describes the context specific threats to implementing the presented solution proposals successfully.

5.5 Threats to successful implementation

The proposed solutions impose a complete overhaul of the systems and tools in use currently within the company. Big changes are never risk free, and in this section the most crucial threats to the successful implementation of the solution proposals are presented.

\textsuperscript{10}https://phpmd.org/
\textsuperscript{11}http://eslint.org/
\textsuperscript{12}https://github.com/brigade/scss-lint
There are two types of threats involved in this transformation. The first type is the context independent challenges associated with adopting CI/CD in any organization, detailed in chapter 2. These include for example change resistance within the organization [Rodríguez et al., 2016] and complex software suites [Leppänen et al., 2015], as well as effort needed to implement the change and managing external constraints on the software in the context of adopting CI practices [Rodríguez et al., 2016].

The second type of threat is context dependent; the proposed tool-chain requires a certain form of implementation when it comes to hosting, back-ups, version control, code reviewing and testing. Within this context the most crucial threats have been identified as:

- Mercurial to Git migration
- Source code hosted on Amazon AWS
- Back-up architecture
- CI server suitability

**Mercurial to Git migration**

Migrating the source code from mercurial repositories to git repositories is a potential threat as the migration must keep the complete repository history intact, as well as retain all tags and branches present in the mercurial versions. In addition to the actual migration, transitioning to git also affects some of the custom build scripts currently in use. These scripts utilize mercurial commands that need to be updated to their git equivalents after the transition. No further threats were identified related to the migration, as git and mercurial are reasonably similar in their capabilities as version control systems.

**Source code hosted on Amazon AWS**

There are a number of security concerns associated with cloud computing and cloud hosting. These issues fall into two broad categories: security issues faced by cloud providers (organizations providing software-, platform-, or infrastructure-as-a-service via the cloud) and security issues faced by their customers (companies or organizations who host applications or store data on the cloud). Krutz and Vines [2010] As the company already utilizes the Amazon AWS services for a large number of their servers and tools, transferring the VCS system to the cloud poses little new risk from the cloud provider point-of-view. However, AWS systems within the company, have
largely been maintained by a few developers with dev-ops knowledge. Moving the VCS to Amazon means distributing ssh keys to all developers and thus a need for proper protocol for managing and securing ssh keys needs to be put in to place. Furthermore, all API keys, application secrets, database passwords and other security sensitive information currently hard-coded in to the source code needs to either be extracted and maintained outside of the repository or moved to an encrypted configuration file. There are popular tools such as Ansible Vault\textsuperscript{13} and git-crypt\textsuperscript{14} available for managing encrypted files within a repository.

\textbf{Back-up architecture}

There needs to be a fail-safe in place for protection against potential data loss. Currently the VCS is backed up daily to a network-attached storage device (NAS) located physically in the company office. This solution is however only viable if the VCS is hosted within the same network, as the outgoing internet connection at the office is non-fiber and not suited for large data-transfers. As the VCS will be hosted on Amazon, a different approach is needed. Gitlab has a built in back-up system that backs up the entire application for full recoveries. It is however not reasonable to back-up the VCS to the same instance it is running on, so to mitigate the risk of data loss the proposed solution is to utilize the built-in integration to Amazon S3 from within Gitlab. This enables secure transfer of the back-ups within the Amazon cloud at very low cost.

\textbf{CI server suitability}

The Gitlab CI is extremely configurable as each runner can be independently hosted and managed. It theoretically supports all necessary operating systems for building the software components developed by the company. But discrepancies between development, build and production environments currently exist. These must be unified and cloned throughout the pipeline to achieve consistent results. The environments used on the web development side are almost completely unified already, but the embedded software side poses larger problems.

Ideally the environments should be provisioned based on configuration management tools or complete virtual machine images. For the web development side, which uses a linux environment, this is easily achieved. The

\textsuperscript{13}\url{http://docs.ansible.com/ansible/playbooks_vault.html}
\textsuperscript{14}\url{https://github.com/AGWA/git-crypt}
embedded side uses a Windows environment with proprietary build tools not
directly accessible via the command line, thus posing a greater challenge.
The implementation of automated testing on the embedded side is outside
of the scope of this study. However, based on the current understanding of
the CI system, automated testing and continuous integration of embedded
software is not made impossible by the CI server and should be configured
outside the scope of this study.

The upcoming chapter describes the process of implementing the above so-
lution proposals. The outcome of the implementations is the new release
pipeline, which in the terms of design science research is the artifact pro-
duced in this study. The following chapters after that evaluate and analyze
the new release pipeline in the context of the CNs, the research questions
and existing literature.
Chapter 6

Implementation

In this chapter the implementation of the new release pipeline is presented. Based on the solution proposals from chapter 5 the Gitlab tool-chain was implemented and now serves as the core of the new release pipeline. The following section goes over the overall architecture of the components involved in the company’s updated pipeline. In addition, the updated integration flow model is described in detail in the following section. Lastly the problems related to the implementation itself are discussed. Any problems and possible restrictions related to the solutions, not the implementation itself are discussed in chapter 7. Before proceeding with the implementation, it is worth noting that the solution proposals presented in chapter 5 were reviewed and agreed upon in the Workshop (Solution proposals) shown in Figure 3.1. The implementation details were not discussed in that workshop, only the solutions that were to be implemented. Any decisions related to implementation details were made by the author. What follows is an overview of the implemented release pipeline.

6.1 Architecture

Based on the solution proposal, the decision to utilize Amazon Web Services\(^1\) (AWS) was made as a means to enable centralizing of services. As production, staging and testing servers were already hosted on AWS, introducing a second hosting provider in to the mix, made little sense. Thus, as can be seen in Figure 6.1 all components relevant to the release pipeline are confined to either the company’s local network environment or AWS. The selected architecture also adheres to the suggestions in existing literature by providing a structure that enforces that *every commit should build on an integration*

\(^1\)https://aws.amazon.com/
AWS is composed of a large number of services and tools. The definition from the AWS website, defines their product offering in the following way:

Amazon Web Services is a secure cloud services platform, offering compute power, database storage, content delivery and other functionality to help businesses scale and grow.

The AWS Cloud provides a broad set of infrastructure services, such as computing power, storage options, networking and databases, delivered as a utility: on-demand, available in seconds, with pay-as-you-go pricing.

The company utilizes AWS computing power in the form of virtual servers, storage options in the form of S3 buckets, as well as CDNs and database services throughout their product suite. Adding Gitlab to the existing infrastructure was a straightforward process. As seen in Figure 6.1 the Gitlab components added to the AWS architecture were:

- Gitlab (main instance)
- Gitlab CI Runner(s)
- Gitlab back-up

In addition to the Gitlab components there are development, staging and production environments present in the pipeline. Next the role or function of each component in the system is described. The motivation for including configuration management in this setup was based on the identified best practice of testing in a clone of the production environment [Fowler, 2013] coupled with the added benefits of easy development environment set ups for new employees.

Development environment(s). The local development environment of each developer. This is the environment where new features are developed. In addition to having a local copy of the code repository, all local environments run a local version of the production environment, complete with local versions of the web server and databases. Automatically provisioned using Ansible for configuration management.

Staging environment. A copy of the production server, hosted on AWS, not locally. Mirrors the production environment as closely as possible. Automatically provisioned using Ansible for configuration management.

Production environment. The live environment accessed by the customers. Differs from the staging server only in computing capacity. Automatically provisioned using Ansible for configuration management.
CHAPTER 6. IMPLEMENTATION

**Gitlab instance** The VCS system hosted on AWS. Runs the Gitlab and Gitlab CI instances. Hosts all code repositories, roadmaps, issues, CI pipelines and wiki articles.

**Gitlab CI Runner(s).** CI testing environments, currently hosted on AWS. CI runners execute tests, builds and deploys according to the rules defined for each code repository. Runners can be configured to run on any machine, and are not limited to AWS hosting. CI runners communicate with the Gitlab CI instance using an API.

**Gitlab back-up.** A complete back-up of the Gitlab instance, hosted on AWS S3. The Gitlab instance is configured to automatically create fully restorable back-ups and offload them to the S3 bucket for storage.

![Figure 6.1: Architecture: Release pipeline components](image)

Information and data flows through the system in the following way: code is checked out from the Gitlab instance to the development environments. New features are developed and tested locally. Changes are then committed and pushed to the Gitlab instance. The Gitlab CI instance monitors all repositories and initializes the corresponding CI pipelines on new commits. The CI instance informs the CI runners about the changes. The runners test and build the updated code. On success the changes are automatically deployed to the staging environment. The changes are then manually acceptance tested in the staging environment. When the changes are accepted the code is deployed to the production environment.

The entire pipeline is Linux based, meaning all environments are running some linux distribution. In the Ansible enabled web store repository, configuration management is configured to provision Ubuntu 16.04 OSes on the development, staging and production servers. The Gitlab instance is, in-line with the installation instructions from their website, also running a Ubuntu
6.2 Integration Flow Model

To further explain the different phases of the release pipeline Figure 7.1b shows the integration flow model of the updated pipeline using Ståhl & Bosch notation. Instead of visualizing the pipeline based on the components involved, as in the previous section, the activities taking place are examined...
in the following section.

All in all the model has nine activity nodes, one input node and one external trigger node. Input flows from one node to the next in a linear fashion. Trigger nodes follow suite with the exception of the acceptance testing activities which are manually started and ended. At the core of the integration flow model is the Github branching model discussed in chapter 5, it defines when and what happens within the release pipeline.

Everything starts with a developer checking out the code, creating a new feature branch and implementing the new feature. After local development and testing, changes from the feature branch are pushed to the VCS. The commit triggers the build phase where the code is compiled and packaged. On success the test and analyze phases are triggered. A series of static code analysis tests and linting rules are run, checking the code for syntax errors and style violations. In parallel, the automated test suites are started, running the unit tests and integration tests. If both phases complete successfully, the code is deployed to the staging server for further testing. The following phases: internal and external acceptance testing are about manually checking the software for bugs not caught by the earlier processes. UI, performance and
other non-functional requirements are tested and reviewed in these phases. When the software is deemed accepted a pull request (PR) from the feature branch to the development branch is created. The PR is peer reviewed, if no critical errors are found in the code, the PR is accepted.

This triggers a release, there are two types of releases depending on what kind of branch is being merged. Delivery and deployment releases. Delivery releases are potentially deployable software created when merging a feature branch with the development branch. Deployment releases are automatically deployed releases created when merging the development branch with the master branch. In delivery releases any referenced issues are automatically closed and the feature branch is automatically closed. In deployment releases the version number is automatically updated and the changes are deployed to the production environment.

This concludes the implementation phase, in the following chapters focus is shifted to evaluating the produced artifact and tying back to the research questions. However, before moving on to the evaluation, the encountered implementation problems are briefly discussed in the next section.

### 6.3 Implementation problems

Larger challenges and problems related to the adoption of CI/CD are discussed in chapter 8, this section discusses encountered problems related to automatic deployments using Gitlab CI in conjunction with Ansible along with the actions taken to mitigate the risks related to the potential threats discussed in chapter 5. None of those threats were realized during the implementation. To recap the identified potential threats were:

- Mercurial to Git migration
- Source code hosted on Amazon AWS
- Back-up architecture
- CI server suitability

Mercurial to Git migration was not a problem. The fast-export\textsuperscript{3} tool worked without problems on all of the repositories, successfully migrating each one from mercurial to git with all tags, branches and the full commit-history intact.

\textsuperscript{3}https://github.com/frej/fast-export
Source code hosted on Amazon AWS was about the security concerns related to security sensitive information stored in the repositories. Utilizing Ansible Vault\footnote{http://docs.ansible.com/ansible/playbooks_vault.html} all passwords and keys were encrypted before being committed to the repository. The tool worked as documented and no issues emerged from using it.

Back-up architecture for the Gitlab instance. Gitlab comes packaged with a set of administrative tools for maintaining the instance. Back-ups are supported through the `sudo gitlab-rake gitlab:backup:create` command which creates an archive file that contains the database, all repositories and all attachments. Gitlab can then be further configured to automatically upload the back-up to Amazon S3 using the Fog library\footnote{http://fog.io/}. This configuration worked without problems as soon as the S3 api keys were added to the Gitlab configuration file.

CI server suitability turned out to be a non-issue as the CI runners can be configured on any OS that supports the GO language. This means that even for the difficult hardware side, CI runners can be used as the current build machines can be configured to act as CI runner environments as well. As long as the runner is set up with a shell executor all tools available on the host machine can be accessed by the runner.

Automatic deployment using SSH keys

What did cause problems however was configuring automatic deployments as part of the release pipeline. Gitlab handles access to checking out repositories through user management and SSH keys. The system is set up to accept any verified Google Apps user from the bt.tn domain as a new user within Gitlab. Each user then associates a SSH key-pair with his account. This enables push and pull access to the repositories. The problem however arises when the CI runners are set to deploy code automatically to the staging and production environments. There are two approaches to solving this issue. One is through SSH agent forwarding\footnote{https://developer.github.com/guides/using-ssh-agent-forwarding/} which allows the usage of a users local SSH keys on a remote server. The other approach is actually adding the SSH keys needed to connect to the production server to the build server. This poses a larger threat than the first approach, as a breach on the build server would potentially yield access to the production server through unprotected SSH keys.

For this reasons the first approach, SSH agent forwarding, was chosen as the best way to deal with remote deployments. There were however several
potential stumbling blocks along the way that caused the set up to be relatively difficult. Although not exhaustive, this list covers the most common problems:

- You must be using an SSH URL to check out code
- Your SSH keys must work locally
- Your system must allow SSH agent forwarding
- Your server must allow SSH agent forwarding on inbound connections
- Your local ssh-agent must be running
- Your key must be available to ssh-agent

Furthermore OSX forgets any keys added to the ssh-agent on a restart, so for persistence the keys must be added to the OSX keychain using `ssh-add -K your-key`. Once properly configured, automatic deployments from the CI runners to the staging and production environments started working.

Overall, the implementation was easier than anticipated; very few problems were encountered during the set up and configuration of the tools. Most things worked out-of-the-box and configuration was easy thanks to extensive and well written documentation. After actually going through with the implementation, we were left with the perception that implementing the new release pipeline was surprisingly enough the easiest part of the entire transition. The real challenges lie in the actual adoption of the new release pipeline. These challenges, along with the experienced benefits and contributing success factors are further discussed in chapter 8. Before delving in to that, the release pipeline is evaluated based on how well it solves the CNs in the next chapter.
Chapter 7

Evaluation

In this chapter the effects of the implementation of the new release pipeline are evaluated. Each of the Company Needs (CNs) included in the study are assessed, their role and impact on the new pipeline evaluated and discussed. Furthermore, the integration flow models from before and after the implementation are compared to each other. This chapter consists of two distinct parts. The first evaluation is grounded in actual use of the entire updated release pipeline, as experienced by the author in his day-to-day use of the implemented systems. The *bt tn website* repository was successfully migrated to the new tool-chain and has utilized the new release pipeline since August 2016. The entire development department has adopted the code health checker and started adding configuration management to existing repositories, but a full transition to the new pipeline has yet to happen for the other repositories.

Thus the second part of this chapter describes a second evaluation of the updated release pipeline from the perspective of the other stakeholders. In order to gain a better, and more unbiased understanding of the release pipeline a final workshop targeting the evaluation of the produced artifact was organized by the author. The workshop included all stakeholders from the previous workshops. The idea was to assess, evaluate and gather feedback on the new tool-chain and release pipeline from all stakeholders involved in the adoption. The main findings of this workshop are presented in section 7.3.

7.1 Removing the bottlenecks

The largest problems with the old release pipeline were related to long testing streaks, accompanied by code-freezes and blocking of the master branch. Thus, often followed by integration problems and an uneven commit pace.
CHAPTER 7. EVALUATION

Addressed Company Needs

<table>
<thead>
<tr>
<th>Description</th>
<th>CN</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Define process for Backlog Management and setup supporting tools</td>
<td>CN1</td>
<td>1</td>
</tr>
<tr>
<td>Define process for Bug Tracking and setup supporting tools</td>
<td>CN2</td>
<td>1</td>
</tr>
<tr>
<td>Define branching logic</td>
<td>CN3</td>
<td>2</td>
</tr>
<tr>
<td>Define process for code review</td>
<td>CN4</td>
<td>2</td>
</tr>
<tr>
<td>Define code ownership</td>
<td>CN5</td>
<td>2</td>
</tr>
<tr>
<td>Create CI Server for testing, building, reporting</td>
<td>CN8</td>
<td>1</td>
</tr>
<tr>
<td>Define unit testing process and implement test frameworks for use on CI server</td>
<td>CN12</td>
<td>2</td>
</tr>
<tr>
<td>Define code style specifications and coding standard</td>
<td>CN18</td>
<td>1</td>
</tr>
<tr>
<td>Implement linters into development workflow and CI</td>
<td>CN19</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 7.1: Company needs

The lack of automatic steps involved in the old release pipeline paired with the absence of automated tests provided little trust in the quality of potential releases. To address these problems properly, changes in all areas related to the development cycle were introduced. The CNs included in this study can be seen in Table 7.1. The inclusion criteria Category 1 and Category 2 defines wether the CN was fully implemented (Category 1) or addressed only as solution proposal, not implemented (Category 2).

The new tool-chain provided a way to centralize the tools used within the company to one location. It reduced the number of user accounts per developer from 4 to 1 through the use of OmniAuth and the grouping of all services to the Gitlab instance.

Define process for Backlog Management and setup supporting tools (Category 1). The connection between project management and development was formally completely missing. Development was fully based on tacit knowledge being passed around by word of mouth within the company. The reason for this was part due to the small size of the company, and part due to the scattered nature of the existing tools. Backlog management tools and issue trackers weren’t consistently used, largely because they were completely disconnected from the development flow. With Gitlab, road-mapping, backlog management as well as issue tracking is not only found in the same place as
all development related tools, they are inherently part of the development process. Effectively connecting project management to development in a formal way.

Define process for Bug Tracking and setup supporting tools (Category 1). Bug tracking is handled as a special instance of normal task management as described above. Bugs are tagged with the appropriate tags and documented in the same manner as other tasks and features, from within Gitlab.

Define branching logic (Category 2). The branching model is at the heart of triggering the pipeline. The feature-development-master branching model is a simple, easy to understand model that differs as little as possible from previous branching conventions used within the company. It is designed to keep a clean commit history, with a low number of concurrent branches, while still covering the complexity of deploying to different testing and staging environments based on the branch the code was committed to. This conditional flow is further explained in Figure 7.2.

Define process for code review (Category 2). The process of peer assessment through code reviews impacts both quality and knowledge sharing within the organization. From a quality point of view fewer errors end up in production as at least two developers have checked the code for potential problems. Peer reviewing has another indirect effect on quality as well: the knowledge that another developer will review the code induces extra effort to ensure everything works before inspection [Radigan]. As the company strategy is geared towards growth, the importance of code review grows with the size of the development team. Code reviews not only increase transparency but work as a way to familiarize new developers with parts of the system previously unknown to them. The biggest challenge was to find the right place in the pipeline for the review process. Not every commit needs to be reviewed, nor should the reviews be done only once per release as the reviewable content then grows unmanageable. The solution to this problem was found in the branching logic. Assigning each branch their own set of rules allows for fine-grained control over what takes place where and when. Code reviewing can then be ignored on feature branches, done by any developer on pull-requests to the development branch and, as defined by the code ownership rules, by the domain expert on pull-requests to the master branch.

Define code ownership (Category 2). As the developers in the company collaborate on all repositories collective code ownership seemed the logical solution. However, as each repository has a distinct domain expert the solution became slightly altered. Feature branch creation is unrestricted, so is merging new features to development through the process of pull requests. Only the master branch, and thus the production environment is restricted.
This convention was put in place as a measure of control, as the company moves to outsource some of the development work to off-shore developers in the near-future. The repository administrator, in this case the domain expert has sole rights to accepting pull requests to the master branch. This two-fold code ownership was also the solution to the timing problem related to the code reviews. As any developer can accept pull requests to the development branch, reviewing is distributed but frequent, with the domain expert reviewing only the release candidate. This way code reviews target only completed features and completed releases respectively. Nothing unfinished needs to go through the code review process.

Create CI Server for testing, building, reporting (Category 1). The most important of all CNs in the context of this study. Adding the CI server to the release pipeline improved several areas: increased trust in quality of the software, removal of several manual steps from the workflow, a decrease in possible human errors and streamlining of the release pipeline. In addition to being at the core of this study, directly affecting several of the other CNs, the CI server provides a foundation for future extensions to the release pipeline. As covered in chapter 4 several of the testing needs identified within the company relate to the testing of the bttn device as well as server-device communications. The CI system provides a base to build this testing suite upon. It imposes a drastic change on the way of working, and the migration of all repositories turned out to be more problematic than anticipated. The largest obstacles were security related and are discussed in detail in section 7.3.

Define unit testing process and implement test frameworks for use on CI server (Category 2). As testing was close to non-existent within these repositories and there did not exist much of testing culture within the organization imposing radical changes was deemed likely to fail. This meant test-driven development (TDD) was not a feasible solution. Instead a more incremental and gradual approach to introducing testing in to the workflow was adopted. All new changes are going through the updated release pipeline which means they will pass through the CI system as well as code reviewing before ending up in production. This makes new changes visible to the development team in a new way, and thus subject to scrutiny. Paired with the effects of code review and reporting from the CI system writing tests becomes increasingly visible and incentivized. In addition to creating tests for all new changes, a code analysis is to be held to determine the most crucial parts of the existing system to create unit tests for. It was deemed too big a task to write unit tests for the entire existing system, instead the key parts of the system will receive tests. Other parts of the system will receive tests if/when development alters them. This way the testing suite will grow with time, gradually covering more of the system.
Define code style specifications and coding standard (Category 1). All repositories suffered from confusing indentation and varying uses of line endings and file endings due to the use of different IDEs amongst the developers. The consensus was to fix this problem without introducing extremely strict code styles that were seen as causing more harm than good. As discussed in chapter 5 the editorconfig tool is used for managing indentation, whitespace and other language agnostic parts of the code. It was configured for all IDEs used by developers and a config file was added to the repositories. Although drastically improving the situation there were some IDE specific differences in the execution of the editorconfig file. Editorconfig support was based on third-party plugins, unique to each IDE and especially indentation handling differed to some extent. The main problem was related to how different IDEs handled existing files. A batch task needed to be run on the entire repository to normalize the indentation and to make the IDEs behave correctly.

Implement linters into development workflow and CI (Category 1). A linting task was added to the CI system, which checks all code for syntax errors and editorconfig level checks. The linting task is configured for stricter, language specific checks for PHP, Javascript and SASS incase they are deemed necessary. Running linting checks in the CI system makes builds fail for code not adhering to the guidelines. Developers not adjusting to the new guidelines are unable to merge changes further up the pipeline. This has been proven an effective way to introduce a new doctrine as long as the organization is in favor of it [Fowler, 2013].

Each of the CNs outlined in chapter 4 have been addressed by the implemented release pipeline as described above. This has caused drastic changes to the entire system. To further concretize the differences between the new and old release pipelines the integration flow models constructed for each pipeline are compared to each other in the following section.

7.2 Automating the pipeline

By removing the bottlenecks discussed in the previous section, the release pipeline changed. Moving from a strictly manually triggered model to a more automated one, made the model less complicated and more linear. In addition to being more automated overall, some new activity nodes were introduced: test automation, pull request and release. The test automation activity is part of the CI system, handling the execution of tests and reporting of code coverage. The pull request activity is a way to add peer reviewing to the development process and impose control over the branching conven-


The pull request activity is closely related to the release activity. The release node handles activities related to deploying new software to production; incrementing the version number, generating a new changelog and triggering update notifications.

The manual triggers needed in the new pipeline are limited to the acceptance testing activities and the handling of the pull request. The entire model has shifted from a developer driven, series of manual steps to a VCS/commit driven automated pipeline. In the old release pipeline the role of the VCS was nothing more than a place to store and share code amongst developers. It had no relation to any other activities within the pipeline. In the new model it is at the heart of the pipeline, a commit to the VCS triggers the entire release pipeline. This is a drastic change to the importance of the VCS which is the most important tool in the new tool-chain.

![Diagram of release pipelines](image)

(a) Original release pipeline  
(b) Updated release pipeline

Figure 7.1: Release pipeline modeled with Ståhl & Bosch notation

The model seen in Figure 7.1b depicts the generic representation of the new release pipeline. It is modeled to show all activity nodes in-common for the projects within the company. In reality the flow differs slightly between projects and is often more complicated than shown here. There are iterative and conditional flows related to the model that are hard to express accurately with the Ståhl & Bosch notation. The next section explains these conditional flows in more detail.

### Conditional flow

The model is based on a set of conditional rules based on the branching conventions used. Commits to different branches cause different things to happen in the testing and deployment phases of the pipeline. The release
pipeline is built around the concept of \( n \) development environments, \( m \) testing environments and one staging and one production environment. The simple master-development-feature branching model controls how, when and where the code is moved within the pipeline. In comparison to the old model where there was an arbitrary division in to hotfix or roadmap releases, the new model has a two-phase deployment cycle grouping changes in to potentially deliverable and deployable releases. All changes go through both of these phases as potentially deliverable releases are grouped together to form a deployable release. Explained from a branching perspective this means changes in feature branches move through the CI parts of the pipeline but are not deployed to any other environment than the local development environment. Feature branches are then merged to the development branch which moves through the release pipeline ending up in the staging environment. This constitutes a potentially deliverable release. When all features planned for the following release have been successfully merged to the development branch a pull request from the development branch to the master branch is created. All potentially deliverable releases added to the development branch since the last actual deployed release are then merged to the master branch and the next deployable release is thus created.

This conditional flow ensures that all changes going to the production server have been tested, not only on their own in the feature branch, but also together with all other changes made to the system since the last release. All potential code conflicts stemming from two or more developers working on the same parts of the system are thus dealt with before the changes reach the master branch. This ensures that there is a deployable version of the system available at all times.

The first part of the evaluation ends here. In this first part the new release pipeline, consisting of the implemented solution proposals from chapter 5 were explained and evaluated from the point of view of the author. Their connection to the CNs and the reasoning behind the implementation decisions were also explained. As the owner of the \textit{bt.tn website}, the only repository fully utilizing the entire updated release pipeline, the author had most experience with the system and was as such best suited for evaluating it. In order to triangulate and to some extent validate this evaluation a second evaluation was organized. This evaluation is further explained in the next section.
7.3 Evaluating the pipeline

A workshop about the implementation was organized by the author on September 14th, 2016. The workshop was held in order to gather feedback and thoughts on the implementation of the Gitlab tool-chain and the new release pipeline from stakeholders from development, project management and management. All developers, from both the hardware and software side participated, along with the Product Owner and CTO of the company. The goal of the workshop was to evaluate the produced artifact as well as possible, without access to extensive usage experience reports from the participants. Due to misalignments with the project schedule and the CI/CD adoption timeline within the company, it was not possible to gather experience data based on extensive use of the system from all stakeholders as the system had yet to be taken fully in to use. Thus the following evaluation is based exclusively on the thoughts gathered from the participants during the workshop.

The workshop was structured to include a brief recap of the CNs and the corresponding solution proposals, followed by an introduction and walkthrough of the features available in the Gitlab tool-chain. A live demo was then held demonstrating how a new code change moves through the release pipeline in the context of Gitlab. Each step seen in Figure 7.1b was explained through its corresponding action in the Gitlab interface, linking each node of the release pipeline model to its concrete interface within the tool-chain.

The participants were then asked to evaluate the implementation. To effectively engage all participants the think-pair-share method was once again utilized. To achieve a structured response, the participants were tasked with evaluating each step in the integration flow model. After a 10 minute session of working individually on this, the participants paired up and discussed their findings further. Finally, the results were discussed together with the entire group. This discussion phase raised some valid concerns, pointed out some sub-optimal decisions in the model and validated most of the decisions made by the author in the implementation phase. In the following section the main findings and insights of the workshop are presented.

Workshop results

“Too generic”, ”What about the not happy cases?”

The first concern was related to the completeness of the release pipeline: how well does the integration flow model cover other than the ”happy cases” were everything goes according to plan? The model itself covers only generic cases and is too simplified and restricted in expression power to accurately
describe all possible cases. The Ståhl & Bosh notation is limited in accurately
describing finer details like conditional branching, component dependencies
and user-level controlled triggering that are part of the new release pipeline.
Together with supporting documentation and training however, these short-
comings can be addressed outside of the model. The problem was not related
to the pipeline itself, rather a concern pertaining to the notation used for
modeling it.

"What about schema changes?", "Not all code changes
are equal"

The second concern was related to the expected behavior of the CI system and
the developer when dealing with code changes affecting components external
to the system. Some code changes are reflected in environments outside of the
repository i.e. changes to databases, external APIs, updated dependencies
etc. This leads to the fact that not all commits are equal, some commits will
contain tests or changes that undoubtably will fail as long as the external
components remain unmodified. The most important feature addressing this
problem is the ability to manually trigger the CI pipeline without the addition
of new commits. This means the ability to repeat failed test and build
stages within the CI pipeline for old commits. This way external components
affecting the test outcomes can be updated independently of repository code.

"How does the system know when to test what?", "Which
commits go to staging?"

An important insight was how big of a role the CI configuration file turned
out to play in the pipeline. It was unclear how crucial the gitlab-ci.yml file
was to the orchestration of flow within the pipeline before the actual imple-
mentation. Most logic is defined within this file, which means the same CI
system can easily be tuned very differently depending on repository. The
gitlab-ci.yml provides the possibility to define per branch and even per com-
mit behavior of the CI system. As it is version controlled, changes to CI be-
havior can be commit specific without breaking builds targeting other code
versions. This flexibility turned out to be the key reason the generic release
pipeline could successfully be applied to the entire product suite within the
company.

"Development should be included in the model and code
checks and unit tests should run locally as well"

As seen in Figure 7.3 some alterations to the proposed pipeline were made as
a result of the workshop. There was a need for a development node prior to
the VCS node. Actual development, writing tests and running tests locally as well as testing in local testing environments was seen as crucial tasks related to the release pipeline. The decision to include these activities in the model, make them a formal part of the pipeline. This was to ensure that linting your code, writing tests and running tests locally would stop being an ad-hoc activity carried out sporadically by some developer. Instead a more structured approach where each developer is directly responsible for his commits was chosen. Through added transparency, test coverage reporting and other information radiators the current quality of the system is constantly visible which provides positive pressure on the developers to improve or at least maintain the status quo with each commit.

"Performance testing shouldn’t be done for all commits"

Furthermore, the proposed model does not address performance and "release candidate" testing efficiently. Not all tests are suitable to run as part of the test automation suite triggered on every commit. There is a need for a larger, more thorough and potentially a lot slower "nightly" testing suite triggered based on a schedule or manually. These tests would focus on large database queries, high number of concurrent connections, bttn device to server communications, bttn press tests and other costly tests not sensible to run for every commit. In addition to running these on a schedule, a need for triggering these tests for release candidates, i.e. deployable releases was also identified. As a result, the tests triggered for all commits were kept fast. Figure 7.2 shows the build durations for the last 30 commits. The three spikes present in the graph, are related to network timeouts during the deploy stage of the CI pipeline. Otherwise, test and build phases consistently take under 30 seconds in total to complete after caching of external dependencies was configured for the CI system. Automated deployment adds another 40 seconds to the total build time, resulting in an overall runtime of 60-70 seconds for moving code from a development environment through the CI pipeline to the staging environment. These values are specific to the bt.tn website repository, but shows that by optimizing the test suite and moving more costly tests downstream keeping the build fast for standard commits is easily achievable.

A smaller issue was that the model doesn’t address artifact handling and conventions in its current form. Gitlab supports artifact handling out-of-the-box, which means it is configurable to upon a successful build upload the built version of the software to a predetermined location. The company however lacks an agreed upon convention for dealing with artifacts. There is no guideline specifying how many artifacts should be stored, for how long old versions should be supported, where they should be stored and what
Figure 7.2: Build duration in minutes

Figure 7.3: Modified release pipeline (Ståhl & Bosch notation).

constitutes a release artifact. This is closely related to the handling of the btn device firmware and outside the scope of this thesis. The problem was however raised during the workshop and the possibilities provided by Gitlab explored and deemed sufficient.

"This architecture constitutes a breaking change in our security policy”

However, the largest concern was not with the release pipeline itself, but with the architecture, or rather hosting solution suggested. The proposed cloud-based model is a drastic shift from the status quo. Currently all code repositories are hosted at the company premises, available only from that
internal network. From a risk assessment point of view, the environment has been deemed secure and thus allowed sensitive information to exist within the checked-in code. This means the repositories are inherently, in their current state, a security risk and can not be moved to the cloud without addressing this issue. However, a cloud-based solution is more in-line with the company strategy than sticking with the internally hosted one.

"This is all great but a change like this is too big”

The entire adaptation of, and migration to the new release pipeline was deemed too big of a task to handle as one project. Instead a step-wise solution was chosen over the suggested solution proposal. The Gitlab tool-chain is to be setup and configured in the company internal network, removing the initial security concern regarding any sensitive information present in the repositories. Then, work related to securing the repositories and removing hard-coded sensitive information can be carried out risk-free. When all repositories are secured and the process for securing, handling and storing the sensitive information is in place, can the pipeline be moved to AWS.

As a corner-stone in the successful execution of the above plan is the incorporation of configuration management in the process. All components related to the new release pipeline are to be configured using Ansible. So that once everything is ready, tested and deemed safe to host in the cloud, there will be no need for further set up phases. With the help of Ansible all necessary servers will be provisioned in the exact same way in AWS as they were internally. In addition to mitigating the migration risks and eliminating additional manual work phases, the inclusion of Ansible in this project is meant to familiarize as many developers as possible with the tool. The idea is to, in the future, use Ansible to handle provisioning and managing of the development, staging and production environments in all projects within the company.

The two evaluations presented in this chapter focused on the produced artifact of this research in the context of the company needs. In the following chapter the results from this study are discussed from a more general perspective.
Chapter 8

Discussion

In the previous chapter the updated release pipeline was evaluated based on how well it solved the CNs and how well the implementation suited the company. In this chapter the pipeline is further discussed in the context of existing research; mainly how well the implementation correlates with established success factors, proposed benefits and potential challenges outlined in existing literature.

In addition to that, the research questions and their corresponding results are summarized, followed by a brief discussion on the potential threats to the validity of the research. Furthermore the produced artifact together with the research itself are evaluated against the design science research guidelines presented in Table 3.1.

8.1 The ideal solution

Existing research within the field of CI/CD is limited, especially closer to the CD and DevOps end of the spectrum. Continuous integration as a concept has been around longer than the others and thus been subject to a larger number of studies. In a modern setting it is however difficult to talk about CI as an isolated phenomenon, as it has grown into an integral part of larger whole. This study aims to validate the existing body of knowledge through careful generalization of the achieved results and to add to it through previously undocumented findings. To achieve results that have significance outside of the company context the next section validates the findings against existing literature.
CHAPTER 8. DISCUSSION

Success factors

As outlined in chapter 2 there are a number of key factors that have been identified in existing literature as common success factors in enabling successful adoptions of CI/CD practices. The implemented release pipeline checks the box for all of the aforementioned success factors, albeit only partially for some.

Both make your build self-testing [Fowler and Foemmel, 2006] and every commit should build on an integration machine [Humble and Farley, 2010] are directly handled by the CI server. Modern CI tools push you towards setting up the release pipeline in such a way that these concepts are automatically addressed. Creating a pipeline that differed in the fundamentals from this would have required significant effort and time spent re-inventing the wheel.

The following two concepts are easier to get wrong: keep the build fast [Fowler, 2013] and keep the build green [Fowler, 2013] are related to what you test, how you test it and how you deal with errors. In the implemented release pipeline, testing has been split up in to separate activities to ensure the build is kept fast. Slower, heavier tests are run in nightly batches for larger change sets. Faster, per-commit unit tests are run whenever new changes are added to the repository. Keeping the build green is addressed in two ways: developers test locally according to their own judgement in order to eliminate completely unnecessary failed builds on the CI system, and fixing newly introduced errors is always a priority. Non-working code is not allowed to move forward in the pipeline.

Test in a clone of the production environment [Fowler, 2013] and automate deployment [Fowler, 2013] are factors that go far beyond the requirements of a normal CI system as they are more related to DevOps and CD. Cloning environments across the release pipeline as well as deploying code to these environments are handled by Ansible in the new setup. Utilizing this tool not only the code, test suite and CI configuration are put under version control, but the server infrastructure as well. Although Ansible is highly extensible the main functions in use in this particular system are provisioning and deployment. Strictly speaking Ansible does not clone environments, it provisions them to be exactly identical. The difference is small but significant: copying an entire environment is time-consuming. File transfers are slow as virtual machine images often grow to several GBs in size. Changing the approach to provisioning the servers based on a common recipe eliminates these problems. Initial-state servers are spun up and provisioned based on the Ansible playbooks in the code repository, resulting in identical, always up-to-date servers.

The last concept is about communication, transparency and feedback:
everyone can see what’s happening [Humble and Farley, 2010]. In the implemented system Gitlab provides CI reporting inside the UI, along with insights in to the repositories in the form of commit graphs, contributor statistics and language analysis. While these provide potential for everyone to see what is going on, on their own they are not enough. Information radiators have been shown to have a positive effect on how development teams approach things like keeping the build green and addressing pull requests in a timely fashion [Humble and Farley, 2010; Neely and Stolt, 2013]. Although not yet implemented in this system, setting up information radiators showing at least the following: build times, test coverage, build status and open pull requests, are essential to successfully adopting the new practices [Neely and Stolt, 2013].

This information should be visible at all times, on a screen or monitor physically located in the office. This way attention is immediately drawn to failing builds, changes to test coverage and new pull requests. As the team utilizes Slack for all non-verbal communication, setting up notifications from Gitlab to Slack is a good way to further increase awareness of changes happening in the release pipeline.

Challenges

In much the same way as key success factors were identified in existing literature, common challenges in transitioning towards more continuous releases are also present in existing literature. The problems and challenges faced in this case-study are now compared to the common tripping-stones.

Some of these challenges are unrelated to CI/CD and present in most transitions, changes and introductions of new elements in old settings. They are directly related to the human response to change. As Figure 8.1 shows a stage of chaos follows the introduction of a foreign (new) element in to the existing reality. Only after growing accustomed to the change, learning to deal with it and figuring out how to successfully make it part of your reality can a new status quo be reached.[Satir and Banmen, 1991]

Because of the chaotic nature of change a common challenge in adopting CI/CD practices is change resistance [Rodríguez et al., 2016; Leppänen et al., 2015] A valid threat to the implementation, that turned out to be unfounded. A mitigating factor to the risk of change resistance in this study, was the fact that this change was something the company needed. The new release pipeline was introduced as way to address the CNs, meant to solve the biggest problems related to the company release pipeline. During the project no direct resistance was noticeable, although other related problems emerged. The real-life realities of being a small-company severely hindered the implementation. In line with the reports from IMVU, Box and Wix,
presented in chapter 2 the limited resources of the company lead to a shift in priorities, leaving the development team, and especially the author constantly occupied with something more pressing than furthering the adoption of CI/CD. The general consensus within the company was that updating the release pipeline was important, as it would solve a number of problems and needs, just not so important that much time could be dedicated to it. It was also viewed as an extremely large and time-consuming task that could not be done with the existing resources. This meant that, although no direct resistance was present, there was lack of support for the endeavor. Thus based on this case-study the following can be said about change resistance: lack of resistance does not equal support. Organizational support only in theory, is not enough to successfully implement a new release pipeline. With the lack of explicit support from management, it becomes increasingly difficult to gather support from the development team as well. The importance of the change agent or “champion” becomes extremely high in these situations – when change must be pushed from the bottom to the top [Laanti et al., 2011]. In retrospect, as the change agent for this project, a more active role from my side would have been needed to grow the support for the project and get it moved to a higher priority task within the company.

Other challenges covered in existing literature, more directly related to CI/CD implementations are: QA effort [Rodríguez et al., 2016; Leppänen...
et al., 2015], legacy code [Rodríguez et al., 2016] and manual testing [Humble and Farley, 2010]. As mentioned earlier, putting together a testing suite that is complete enough to induce confidence in the quality of the software without the need for manual testing is a big challenge. The QA effort needed to create this testing suite is estimated to be larger than the resources available. The company has decided to hire additional developers to make sure this is handled. A large part of the system has not been written with unit tests in mind, and is considered legacy code from a CI point of view. Getting the testing suite to the level that the need for manual testing is significantly decreased should be regarded as the largest remaining threat to successfully moving to the new release pipeline. Insufficient test coverage of legacy code was seen as a big challenge as the legacy code had regression bugs that were time-consuming and costly to fix [IMVU, Etsy, Box].

Lastly, existing literature mentions complex software and environment management as potential challenges as well [Leppänen et al., 2015]. In this particular case, the software is not too complex to effectively make use of the CI system. Any risk related to environment management has been mitigated through the use of Ansible.

The study unearthed security as a central theme in challenges related to CI/CD adoption. Previous research has not explicitly mentioned security in this context. The security issues were related to both infrastructural concerns about cloud hosted systems and security sensitive information within the repositories. The security of information committed to the repository turned out to be the bigger challenge. A common theme in CI/CD adoptions is extending CI beyond source code [Eck et al., 2014]. This means adding configuration management files, CI configuration files and artifact files to the VCS. These files contain extremely sensitive information like database passwords, API-keys, SSH-keys etc., which are necessary for achieving an automated release pipeline. The rapid emergence of tools designed to solve these issues through repository encryption solutions, suggest that security is becoming a central theme that needs to be carefully considered when designing CI/CD solutions.

8.2 Answers to the research questions

The research questions have successfully been answered in chapter 4, chapter 5 and chapter 7 respectively. What follows is a summary of the key insights for each question.
CHAPTER 8. DISCUSSION

RQ1: Drivers of change

RQ1 What are the motivation factors driving the adoption of CI/CD practices within the company?

The aim was to identify the crucial needs within the company based on a current state analysis of their development practices and their release pipeline. The goal was to not only elicitate these needs, but categorize, estimate and prioritize them to form a holistic view of the drivers of change in CI/CD adoption. The needs are summarized in Table 4.2. Categorizing and prioritizing them were done by assessing their impact on the CI/CD adoption as a whole. When deciding on how to drive CI/CD adoption there are three typical approaches to choose from:

- The low-hanging fruits approach: the aspects that are easily implemented and require the least behavioral changes should be introduced first. As the organizational members get used to the change, more sophisticated practices can be introduced.

- The extended nucleus approach: starting with a core CI system that automates integration and testing, the ambition and reach of CI is extended continuously, until CI is used to leverage operational software for experimentation.

- The challenge-oriented approach: after prioritization of current challenges, the organization should implement those agile practices first that are likely to have the highest return.[Eck et al., 2014]

When prioritizing the CNs and forming the inclusion/exclusion criteria, aspects of both the bottom-up low-hanging fruits approach and the top-down challenge-oriented approach were utilized. Implementing a CI system was seen as the fundamental first step needed to start a transition towards more agile practices overall within the company. Furthermore, decisions specifically related to CI/CD practices were assessed based on their potential return, ease-of-implementation and potential change resistance in respect to the transition as a whole.

The drivers of change identified in this study can be seen as a result of the CSA, concretized in the CNs described in chapter 4. Although described on a very detailed level and anchored in the company context, the largest motivators identified fall under improved software quality, improved release reliability, shorter time-to-market and improved developer productivity, as defined in existing literature.
A crucial need, not mentioned in existing literature, was improved release planning. This need was for the company about the improved collaboration and integration of project management related tasks with ongoing development tasks. The company was experiencing a large gap between daily development work and product roadmapping. Some of the CNs included in the study appear to be on the very outskirts of what can be considered CI/CD practices, namely CN1 and CN2 which are about backlog management and issue tracking. Although heavily related to roadmapping and requirements engineering, backlog management plays a fundamental part in release planning as well. The introduction of the CI system effectively reduced the gap between product management and development, as the centralized solution brought the two activities significantly closer to each other through a more unified approach to release planning. In the new solution release planning encompasses roadmapping, backlog management, issue management, versioning and deployments.

RQ2: Centralizing services

RQ2 What are the most suitable solutions to the identified company needs?

The aim was to find the best possible solution for the specific needs of the case company. The company’s product suite is fairly complex as it contains embedded systems, multiple programming languages and several communication protocols. When dealing with complex systems like this, a suggested approach is to have a common service bus architecture around which the various elements are brought together. Complex systems also tend to affect the CI infrastructure itself, often making it heterogeneous and complex.[Eck et al., 2014]

Thus, the centralized solution provided by the Gitlab suite is a perfect fit. It brings together all the necessary services through a very flexible architecture built around the modularity of the CI runners and the configurability of the gitlab-ci.yml files. Paired with configuration management and DevOps, each part of the complex product suite can move through the same release pipeline without problems.

Furthermore, learning a complex practice such as CI takes time – previous studies show that it is very likely that productivity decreases before any positive effects materialize [Leppänen et al., 2015]. A key success factor in easing the transition is by facilitating learning – both on a group and individual level.[Olsson et al., 2012]

The value of a one-stop service such as Gitlab is emphasized especially at the early stages of learning. A unified UI provides a clearer overall picture
of the system than a fragmented solution would. Moreover, the different services are more likely to be used when grouped together.[Hartson and Hix, 1989] The users need to familiarize themselves with only one new system which further lowers the threshold for adoption [Box, Disqus].

**RQ3: Perfect fit**

RQ3 How well does the implemented release pipeline solve the company needs?

The updated release pipeline adequately solves the problems and needs identified in RQ1. The proposed solutions were by all definitions the right ones, although similar results would most likely have been achievable with a different tool-mix. As discussed in chapter 7 some alterations to the proposed solution were suggested after implementation. The value of multidisciplinary input in the context of assimilation of CI/CD practices needs to be emphasized here. A release pipeline contains activities that span organizational silos, involving stakeholders from development, management and sales to customers and end-users. Valuable insights that led to concrete changes in the proposed model were gained from workshops involving stakeholders from these groups. An important factor in the successful implementation of the CI system turned out to be wide-spread stakeholder involvement throughout the process.

The implemented solution had another unforeseen effect, that positively affected the whole. It turned out that the CI system displayed very close ties to the DevOps culture. DevOps is a cross-cultural, no-silos approach to development and operations that emphasizes building, testing, and releasing software rapidly, frequently, and reliably. DevOps promotes a set of processes and methods for thinking about communication and collaboration between development, QA, and IT operations.[Httermann, 2012]

This was not directly a problem within the company, as the development team handled all areas related to DevOps already before the CI implementation. The important difference is that the sole reason for doing this used to be lack of resources. The introduction of CI/CD practices exposed the benefits of explicitly choosing a DevOps culture. Instead of looking at operations as a necessary evil that is best outsourced as soon as possible, DevOps culture gives concrete reasons for the company to not separate operations from development at all. DevOps strongly supports the continuity of development and is a vital part of the release pipeline. It is heavily intertwined with CD practices, especially in release pipelines utilizing configuration management to manage server infrastructure.
8.3 Threats to validity

Design science in information systems research can be viewed as a mixed method research paradigm encompassing both qualitative and quantitative aspects [Hevner et al., 2004]. When evaluating the validity of mixed method research the quality of the study should be assessed from the following perspectives:

- Design validity: identifying correct operational measures for the concepts being studied.
- Inference quality: the accuracy of inductively and deductively derived conclusions in a research inquiry.
- Reliability: the degree to which collected data (results of measurement or observation) meet the standards of quality to be considered valid (e.g., trustworthiness) and reliable (e.g., dependable). [Venkatesh et al., 2013]

Any results presented in the study are conclusions derived from both qualitative and quantitative data. These types of results can be described as meta-interferences defined as:

Inference in mixed methods design is defined as a researcher’s construction of the relationships among people, events, and variables as well as his or her construction of respondents’ perceptions, behavior, and feelings and how these relate to each other in coherent and systematic manner. [Venkatesh et al., 2013]

It is worth noting that observations related to causal effects and relationships presented in this thesis have not been argued as facts, as triangulation and generalization of the results would require additional research to enable verification of the data. All results presented in this thesis holds true only for the context of the study. Any generalizations and extensions of results outside of the company context should be regarded as speculations.

The following section evaluates the quality and validity of this research based on design validity, inference quality and reliability.

Design validity

There are no direct errors or inconsistencies in the correlation between the research questions and the results presented in this study. The study captured what it intended to examine. However, there are a few threats to construct
validity that should be examined. First, the chosen format for qualitative data gathering was workshops. These are free-form, non-standardized methods of gathering information that potentially allows for an uneven distribution of opinions gathered from the participants. As opposed to one-on-one structured interviews, where each interviewee gets a chance to answer each question personally, the workshop format facilitates open discussion which might favor the more outspoken individuals participating. What this means in terms of validity is that some of the results may be circumstantial or opinionated, and all perspectives on the adoption are unlikely to have been fully covered.

What concerns the quantitative data used in this study, a set of threats to construct validity exists for the following data: estimated release dates, inconsistent version numbering, uncertain development cycle times and varying testing conventions. The lack of reliable metrics is due to inconsistent development practices and lack of monitoring tools. For example the server install log did not correspond with the commit history as deployment was a completely external, manual activity. Versioning existed, but version numbers where changed without a clear protocol, and did not correspond with releases. Because of the unreliability of the quantitative data, qualitative data was primarily used when evaluating the implementation of the new release pipeline. In order to improve the validity of quantitative data of this kind in the future, better monitoring should be put in to place along with standardized procedures in favor of ad-hoc processes. Furthermore, additional input sources should be integrated or linked if possible to enable cross-referencing the data. For example, commits and CI build data can be linked to issues and milestones, which can identify versions and releases.

**Inference quality**

All results presented in this study are meta-inferences; constructed from both qualitative and quantitative data by the author. The results are accurate and hold true for the specific context of the study. Most results pertaining to success factors and challenges in CI/CD adoption strongly correspond with existing literature and can be considered to further validate that research. Findings not present in previous literature have been thoroughly evaluated and explained and deemed valid for the specific context of implementing CI/CD practices in a small software company, which means they can act as input for future studies aiming to generalize common themes present in CI/CD adoption and assimilation. However, the results can not directly be regarded as generalizable, but can with caution be extended to cases similar in nature. The insights related to the CI/CD drivers of change,
the unexpected challenges related to security and the strong connection to DevOps can be seen as adding to the existing body of knowledge on CI/CD adoptions.

Reliability

Most of the main results regarding the success and benefits of the implementation can be repeatedly deduced by using the methodology of this study. The findings related to RQ2: What are the most suitable solutions to the identified company needs? are however more difficult to reproduce. The solution proposals are temporally bound to the timespan of the study, as the tool-chain alternatives will change over time, and the motivations made in this study may no longer hold true in the future. Furthermore, the solution proposals are to some degree opinionated by the authors preferences as no objective truths exist in these matters. To minimize the reliability risk, the selected proposals were chosen by the other stakeholders, not by the author.

In addition, as discussed in Inference quality, many of the findings are circumstantial in the sense that they were constructed meta-inferences that could not be explicitly investigated and validated in the context of this study. It is left to future research to validate these findings.

Design Science Research

In extension to validating the findings of the study, DSR provides a set of guidelines shown in Table 3.1 for evaluating the study itself [Hevner et al., 2004]. In this section we examine how well the research conducted in this study adheres to these guidelines.

Design-science research must produce a viable artifact in the form of a construct, a model, a method, or an instantiation. The produced artifact was, as mentioned, the new release pipeline consisting of the Gitlab toolchain and the code health checker.

The objective of design-science research is to develop technology-based solutions to important and relevant business problems. The proposed solution adequately solves the relevant business problems it targeted. These business problems have throughout the study been referred to as the CNs.

The utility, quality, and efficacy of a design artifact must be rigorously demonstrated via well-executed evaluation methods. The implementation of the artifact, along with the artifact itself have been evaluated both from a company specific context and from a more general perspective.

Effective design-science research must provide clear and verifiable contributions in the areas of the design artifact, design foundations, and/or design
methodologies. The artifact has been modeled using the established Ståhl and Bosch notation to ease future research contributions. The results of the study have been validated against existing literature and all new findings have been validated and triangulated as well as possible.

*Design-science research relies upon the application of rigorous methods in both the construction and evaluation of the design artifact.* The research rigor of this thesis has been evaluated from the point of view design validity, inference quality and reliability.

The search for an effective artifact requires utilizing available means to reach desired ends while satisfying laws in the problem environment. The search process has been founded in evidence from the start. A review of existing literature formed the basis for the study, followed by a detailed analysis of the current state of the company. These findings acted as input for forming the CNs and the subsequent solution proposals ultimately leading up to the construction of the release pipeline.

*Design-science research must be presented effectively both to technology-oriented as well as management-oriented audiences.* The communication of this research is two-fold. The concrete result of the study is this thesis, while communicating the research results to the company extends to training sessions, workshops and knowledge sharing on a more practical level.

The findings of this study should ideally be seen as a contribution to research within the field of CI/CD. The following section elaborates further on the identified concepts for potential future research.

### 8.4 Future research

Previous research and published accounts of successful adaptations of CI/CD practices have established a generic model of an ideal release pipeline. This proposed model provides a good reference point and idea of what theoretically should be achievable with modernizing the release pipeline. It is however only a reference, too generic to be of direct use in CI/CD adoptions. With an increased focus on the needs of CI, categorizing the specific drivers of change could help form a design pattern for CI/CD adoptions. Previously undocumented needs such as *improved release planning* and *improved configuration management* emerged as potential themes for future research within this area.

*Security* turned out to be a major challenge in the successful adoption of CI/CD practices. It remains unclear how context specific this insight is and thus it presents an excellent opportunity for future research to validate the
Another unexpected result was the close tie-in with the DevOps culture. Existing literature on CD and DevOps are similar in their meanings and are often conflated in existing literature. They are however two different distinct concepts [Swartout, 2014]. Because of the close correlation and overlap of these concepts future research should make an attempt to more explicitly connect the two.

An aspect that emerged several times during the course of the study was the psychological effects of CI/CD. Software engineering is a people business, which means human emotions are a vital part of it. Even in an automated release pipeline there are human elements affected by it. Currently the research on how the different activities related to the release pipeline affect the different stakeholders involved is practically non-existent. These questions are however related to existing concepts like change resistance, developer confidence and information radiator effects, and could thus be natural extension to these concepts. Extending future research to include behavioral psychology in the context of CI/CD could answer questions like: What effects does the increased transparency have on developer motivation? How can a "champion" effectively drive and enable cultural change within the organization? How does code reviewing affect developer confidence? How do information radiators affect the experienced confidence in the quality of the product? How do customers perceive more frequent releases?

There was no way of accurately gathering data related to these questions within the scope of the study, but the case company remains a valuable source for future research. These questions are best answered by companies in a mature stage of CI assimilation and future research should ideally target previous case-study companies that have successfully adopted CI/CD practices.
Chapter 9

Conclusions

As a result of the design science research carried out in this thesis an artifact in the form of a new release pipeline has been created for the case company. From identifying the largest problems and the most crucial needs related to the way the company delivers software to eliciting the most viable solutions and implementing them, this study has brought the case company closer to its goal of transitioning to agile practices. The adoption of CI/CD practices was identified as the crucial first step towards that goal. This study has introduced a series of changes that have brought development further away from an ad-hoc, largely manual process with no automation to speak of and closer to a more modern world with development supporting tools and collaborative value driven development. CI/CD assimilation is a long process, and the case company is at the time of writing in the first stage of post-implementation: the acceptance stage, where organizational members start to accept the innovation and use it in their daily work. A disruptive change is most likely to fail before it properly gets going, once it gains momentum it becomes likelier to succeed. Although the most severe obstacles related to CI/CD adoption have been overcome, several challenges remain. Prioritizing CI/CD assimilation high enough to allow resources to be allocated to furthering the adoption remains of utmost importance. Only through a commitment to the change will the transition be successful. As a recommended improvement and next step for the case company: migrate the remaining repositories to the new system and start building the testing suites as soon as possible. Do not let the pace of the adoption stagnate.

In addition to implementing a new release pipeline, the main findings of the thesis validated existing research and brought to light some interesting aspects of CI/CD adoption not previously documented. Security, tie-ins with DevOps culture, improved release planning and the central role of configuration management are concepts that should be further researched in the
future. The company remains a potential candidate for a follow-up study; where the assimilation of CI/CD is further examined, and the benefits of the change can be more accurately measured.

In conclusion, when done right CI/CD practices can impose a value driven mindset on to the developing organization. Quality built-in and focus on continuity will shorten the gap to the users as software can be deployed often and easily. This leads to shorter feedback-loops and thus an overall improvement of the software. A state worth striving for.
Bibliography


Nicole Forsgren and Jez Humble. The role of continuous delivery in it and organizational performance. Available at SSRN, 2015.


Daniel Ståhl and Jan Bosch. Experienced benefits of continuous integration in industry software product development: A case study. In *The 12th*


