Benefits and challenges of Continuous Integration and Delivery - A Case Study

Axel Wikström

Helsinki February 22, 2019

UNIVERSITY OF HELSINKI
Department of Computer Science

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Faculty of Science	Department of Computer Science

Tekijä — Författare — Author
Axel Wikström

Työn nimi — Arbetets titel — Title
Benefits and challenges of Continuous Integration and Delivery - A Case Study

Oppiaine — Läroämne — Subject
Computer Science

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
	February 22, 2019	33 pages + 1 appendix

Tiivistelmä — Referat — Abstract

Continuous integration (CI) and continuous delivery (CD) can be seen as an essential part of modern software development. CI/CD consists of always having software in a deployable state. This is accomplished by continuously integrating the code into a main branch, in addition to automatically building and testing it. Version control and dedicated CI/CD tools can be used to accomplish this.

This thesis consists of a case study which aim was to find the benefits and challenges related to the implementation of CI/CD in the context of a Finnish software company. The study was conducted with semi-structured interviews.

The benefits of CD that were found include faster iteration, better assurance of quality, and easier deployments. The challenges identified were related to testing practices, infrastructure management and company culture. It is also difficult to implement a full continuous deployment pipeline for the case project, which is mostly due to the risks involved updating software in business-critical production use.

The results of this study were found to be similar to the results of previous studies. The case company's adoption of modern CI/CD tools such and GitLab and cloud computing are also discussed. While the tools can make the implementation of CI/CD easier, they still come with challenges in adapting them to specific use cases.

ACM Computing Classification System (CCS):
Software and its engineering → Software creation and management → Software verification and validation
Software and its engineering → Software notations and tools → Software configuration management and version control systems

Avainsanat — Nyckelord — Keywords
continuous integration, continuous delivery

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — övriga uppgifter — Additional information

Contents

1 Introduction

Continuous integration (CI) and continuous delivery (CD) may nowadays be seen as essential parts of the software development process. These practices have gained a widespread popularity in the software industry during the recent years (Fitzgerald and Stol, 2017). In contrast to traditional development methods where planning, development, testing and deployment are performed in sequence, continuous practices allow software organizations release software more quickly while still maintaining quality. (Humble and Farley, 2010)

In a nutshell CD/CD consists of always having software in a deliverable state. This is archived by automating the testing and deployment phases of the development workflow. Continuous integration also implies that developers' code is continuously integrated into the main codebase. The main benefits of CI/CD is that software can be developed more rapidly, while quality is still maintained. By making deployment a common task, mistakes, such as configuration errors, are less likely to occur. (Fowler, 2006; Humble and Farley, 2010)

Even though CI/CD is supposed to make rapid software development easer, adopting CI/CD can be challenging. This can especially be the case if the organization is not used to such practices from before. CI/CD practices are also relatively new, and thus research on the topic is fairly recent. Even though high level aspects of continuous practices haven been agreed upon by research, individual organizations may apply these practices differently. (Leppänen et al., 2015; Mäkinen et al., 2016)

The tools for CI/CD are also evolving rapidly together with facilitating technologies, such as containerization and cloud computing. Not much research has been done in how the evolution of these tools relate to the ease of implementing CD.

This thesis consists of a case study of a software organization where CI/CD is employed. The goal is to identify benefits and challenges in the specific context of the case company, and to suggest ways to mitigate the identified challenges.

2 Terminology

There is yet a complete consensus to be formed about the definition of the terms continuous integration, delivery and deployment (Luke and Prince, 2017; Shahin et al., 2017a). Here follows the definitions which are used in this thesis, which seem to be the most common ones used in academic literature.

Continuous Integration (CI) commonly means that a developer continuously integrates their changes with the main codebase (Fowler, 2006). In practice this can be achieved though using the branching and merging features of a version control system (VCS). Successful integration implies that the code compiles and works as intended. The concept of continuous integration also commonly includes the existence of an automated build and testing process. This definition is also used in this thesis. There are many CI tools available for this purpose that integrate with the VCS. Some examples include Travis, Jenkins and GitLab.

Continuous Delivery (CD) can be seen as an extension of CI. CD is also mentioned in first principle of the Agile Manifesto from 2001: "Our highest priority is to satisfy the customer through early and continuous delivery of valuable software." (Beck et al., 2001). Some may refer to CD as an overall part of the Agile practice, but in conjunction with CI, it usually means that the software is always ready to be deployed, which requires an automated deployment process to be in place (Fowler, 2013). This is also the definition commonly used by vendors such as Atlassian and Amazon (Atlassian; Amazon Web Services, 2017).

The final term; **Continuous Deployment** (CDep) refers to the software being continuously deployed to a production environment without manual intervention (Fowler, 2013; Atlassian; Amazon Web Services, 2017). Deployments can happen several times per day. This practice is however not suitable for every kind of software (Leppänen et al., 2015; Shahin et al., 2017a).

Other software development practices that are involved in CI are described below. It is to be noted that the practices here are described in a general way, and organizations adapt their own specific way of working. Since continuous practices are still relatively new, a definite set of "best practices" do not exist. It is also to be noted that practices are to be adopted in a way that fits the domain and nature of the software.

2.1 Version control

A version control system (VCS) allows developers to effectively collaborate on writing code. Using version control is an essential pert of CI. Git is a commonly used VCS that was initially developed for the Linux kernel project. Other VCS include Subversion, Mercurial and CVS.

The set of files that are version controlled are called a repository. A specific revision of the repository is called a commit. Many VCS, model the history of commits as a directed acyclic graph. This allows developers to work on separate branches independently. When the work is done, branches can be merged together. Most repositories have a master branch (also called trunk or mainline).

Merge conflicts may occur when many people are working on the same file. One has to manually decide which changes should be taken into use. A code base with well separated modules can make merge conflicts less likely.

In a distributed VCS (DVCS) (e.g., Git), the developer has a local copy of the repository on their machine. In contrast to a non-distributed VCS (e.g., Subversion), committing and pushing the changes to a remote repository are separate operations. Because of their flexibility DVCS are commonly used today.

Due to this flexibility, there is no standard workflow or branching model. A development organization can use the VCS in a way which fits their needs. There exists some common patters for working with branches. Typically developers work on separate feature branches before merging them to master. The releases of the software can also be reflected in the branching model.

Two common branching models are *git-flow* (Driessen, 2010) and *trunk-based development* (Hammant, 2017). The crucial difference between them are how releases are managed. In git-flow (Figure 1), development happens on a specific development branch. A new branch is created for each major release. If bug fixes are made on a release branch, they have to be merged back to the development branch.

In trunk-based development, development happens directly on the master branch. Bug fixes are also committed to the master branch instead of the release branches. The commits are then cherry-picked to the release branches. Cherry-picking means that the changes of a commit are reproduced on another branch. Feature branches may be used in trunk-based development, but they are advised to be short-lived in order to avoid merge conflicts. They are mostly just used to allow code review and automated testing to be performed. (Hammant, 2017)

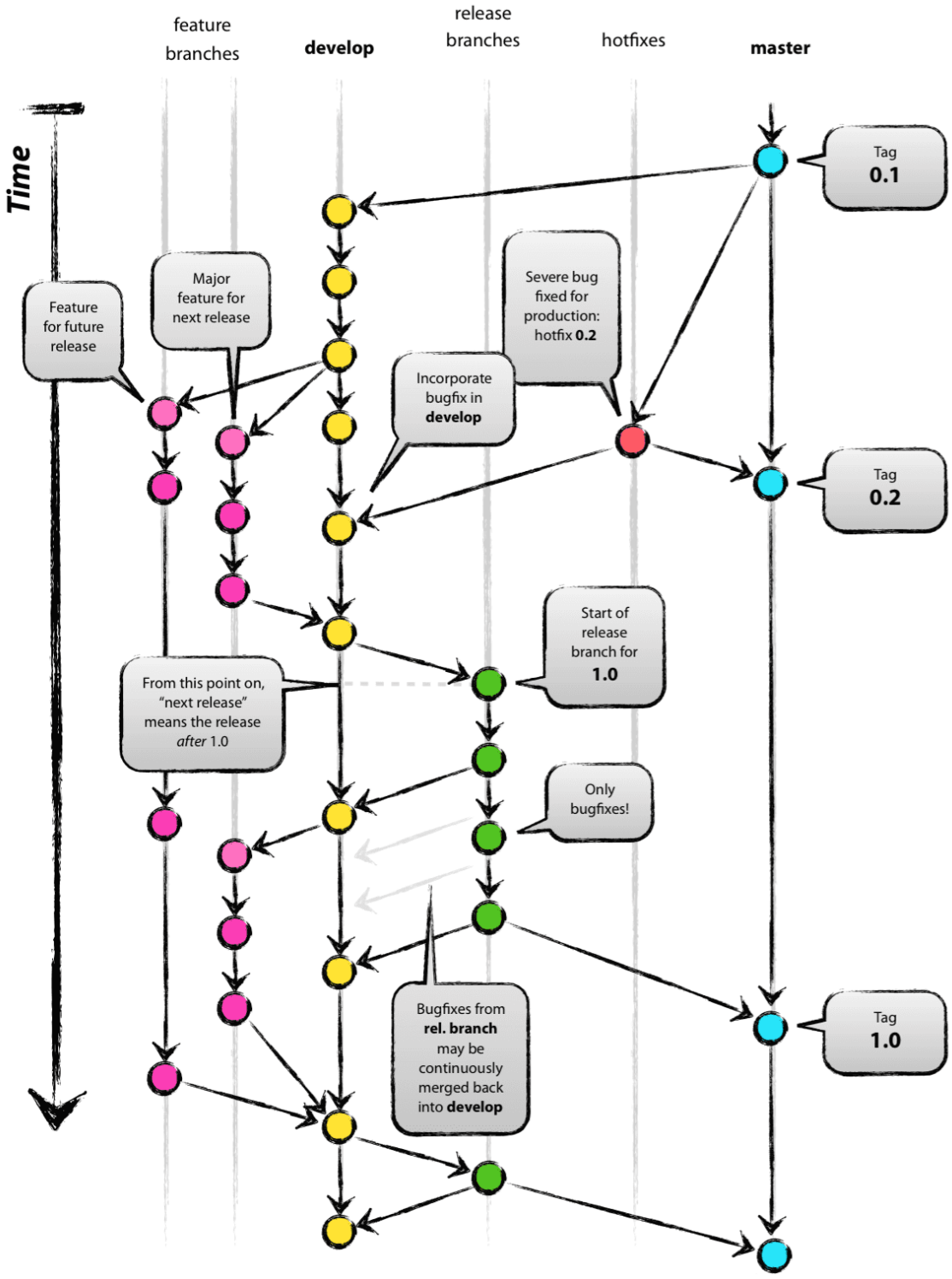

Figure 1: The Git Flow branching model (Driessen, 2010)

Git is commonly used with a web-based git platform. Some examples are GitHub, GitLab and Bitbucket. These platforms provide a user interface to browse the repository along with tools for code review. Most git platforms can integrate with CI platforms.

2.2 Testing

Software testing is a central concept to continuous development. Especially automated unit and integration tests can be seen as an essential part of a successful CI implementation. As the complexity of the software increases, the risk of regressions may also increase (Islam and Zibran, 2017). Continuous testing can be used to prevent regressions. Much of the software testing process can be automated and integrated into the CI pipeline.

Unit testing comprises of testing individual functions or classes in the codebase. Unit testing is commonly done with unit testing framework. Examples of these are JUnit for Java and RSpec for Ruby. Unit tests can be seen as a contract that the implementing code has to fulfill.

While unit tests test the code in isolation, *integration tests* test multiple parts of a system, such as an application and a database. Some refer to integration tests as unit tests as well, because they can be made using a unit testing framework. Integration tests usually take longer to run than unit tests.

Acceptance tests verify that the software can fulfill the required business needs, which are high-level requirements which may be defined by, e.g., user stories. This is done by testing a complete deployment of the system. The tests can be performed against the user interface or a backend API.

2.3 CI pipeline and tools

These practices come together in a CI pipeline. Each revision of the software goes through the pipeline. Typically this is accomplished using a CI tool, such as Jenkins or GitLab. The CI tool integrated with the VCS so that the pipeline gets run for each commit. At least the pipeline includes build and testing stages. If CD or CDep is employed, the pipeline may have a manually or automatically triggered deployment stage.

In some cases the pipeline may involve manual verification stages, such as accep-

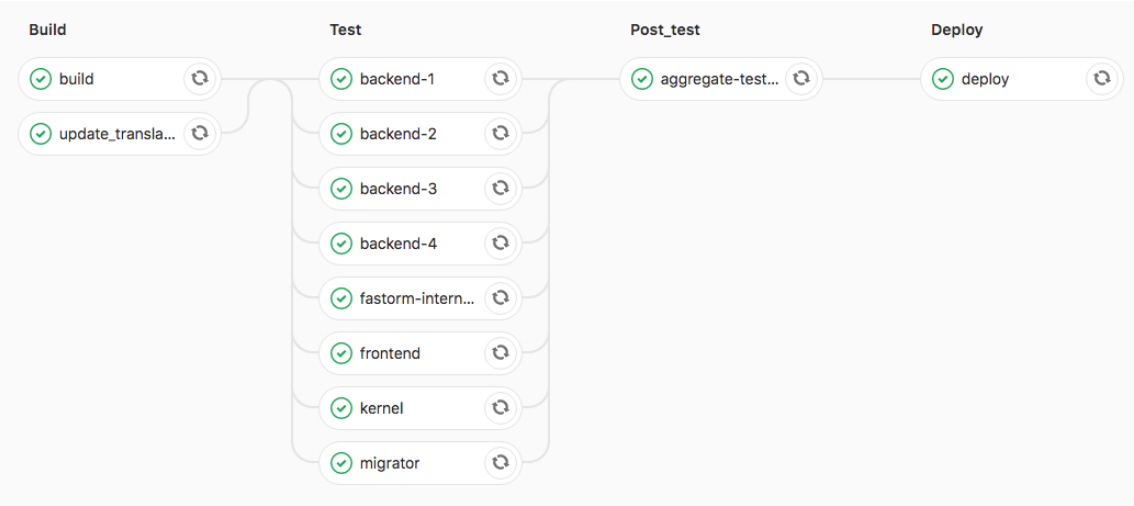

Figure 2: Example of a CI pipeline

tance testing. In cases like this, the "pipeline" may be considered to extend outside the CI tool. As an example from this case study, deployments to customers is done with a custom tool, which is separate from the main CI tool. In addition to tests, the pipeline can also include other stages. Dependency checks scan for known vulnerabilities and unapproved software licenses in the dependencies. Style checks aka. "linting" can verify that the source code adheres to a commonly agreed code style.

Another tool in CI is information radiators (also known as build monitors or dashboards), which indicate the status of the pipelines on a visible monitor in the office (Figure 3).

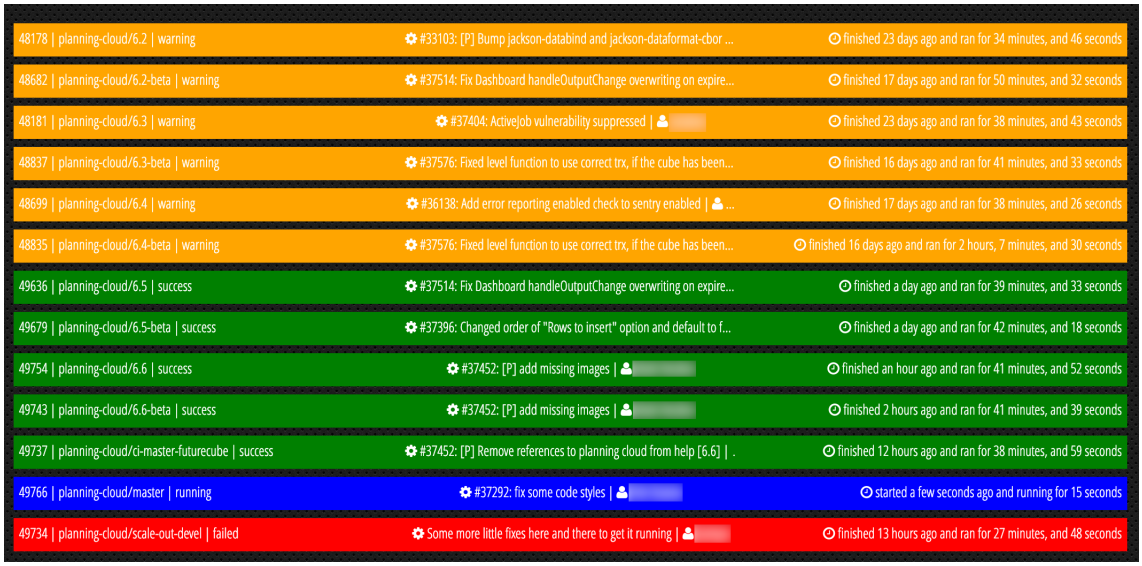

Figure 3: CI radiator used at the case company. The latest commit and build status for each version branch is shown. Blue color indicates a currently ongoing build, while green and red indicate passing and failing builds respectively. The yellow color indicates a "warning" status which might for example mean that a security vulnerability has been found in a dependency.

3 Methods

3.1 Case background and motivation

The case company is a Finnish software company which develops supply chain optimization software for retail businesses. During the last years, it has seen a significant growth in its number of employees. For software development, the collaboration may become more challenging as the development teams grow. The increasing number of customers also provides a challenge, as the software needs to be deployed to the customers' specific environments along with their custom configuration and integrations.

This thesis concentrates on the development process of the *planning-cloud* project, which takes place in Helsinki, Finland. Planning-cloud is a web application implemented with various technologies including Java, JRuby on Rails, Ember and React. The software is deployed as a monolithic WAR file running on the Tomcat server.

The project is developed by multiple teams. The main application is supported by a custom in-memory database and deployment tool, which are developed by their respective teams. In addition, a release management team manages the different versions of the software. The release management team also does nightly black-box and performance testing.

Multiple versions of planning-cloud are maintained simultaneously. Four to five different versions are supported at the same time. A form of trunk-based development is used. Main development happens in the master branch which correspond the current unreleased version. Feature branches are squash-merged[1] into single commits on the master branch. If a bugfix needs to be implemented into a previous version, the commits are cherry-picked[1] into the version specific branches.

When a new version is created, an alpha branch is created from master, and a *feature freeze* is introduced for that version. When the alpha branch is deemed stable enough, a beta and stable branch is created. The alpha branch is then deprecated. This is shown in Figure 4.

At the time of starting this thesis, Jenkins was used for running the CI/CD pipeline. At the time of writing, a transition of the CI pipeline from Jenkins to GitLab is in progress.

[1] *squashing* and *cherry-picking* are features of the Git version control system.

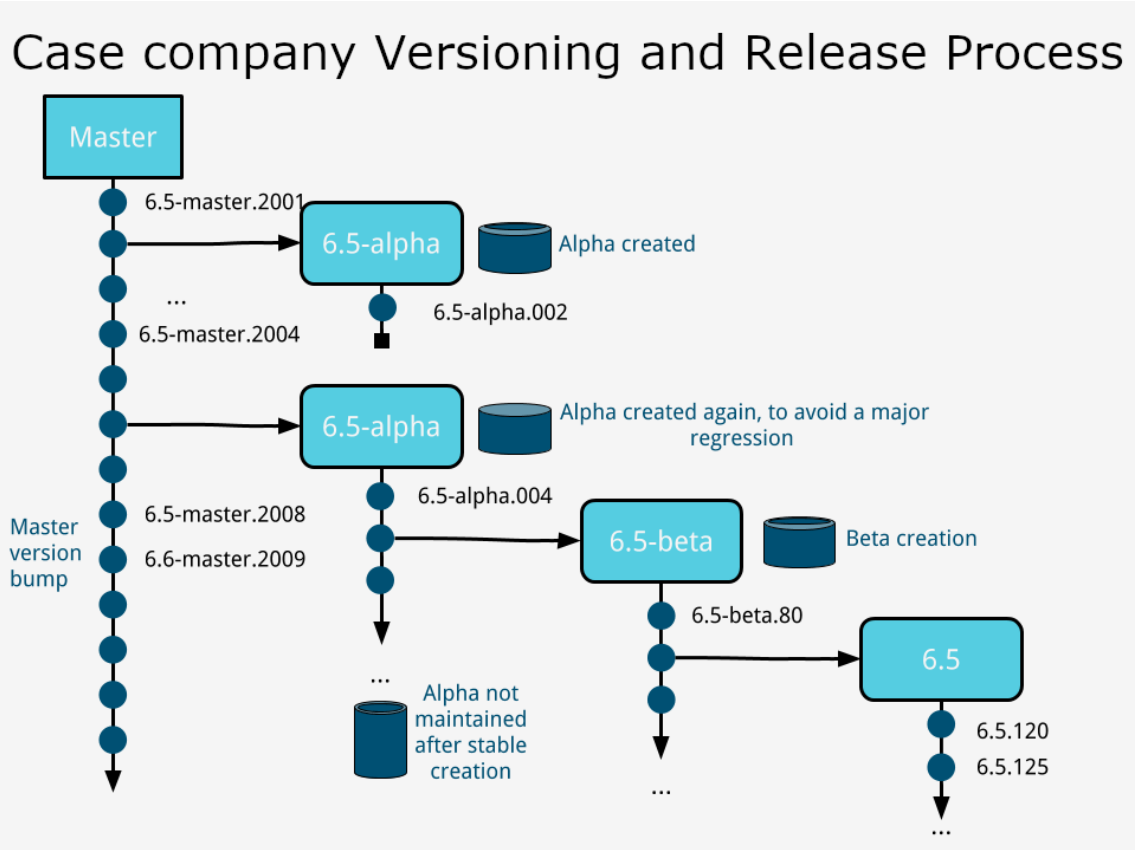

Figure 4: A form of trunk-based development is used at the case company.

The CI pipeline produces a WAR file for each commit passing all tests. The files are stored in an artifact repository (Sonatype Nexus).[2] The custom deployment tool deploys these to the staging and production environments and makes the appropriate configurations to the software and runtime environment.

As the case company has grown, the teams and development processes have evolved. The author was employed to look over the CD process and implementation at the case company. The goal of this thesis is to conduct a case study in order to identify the benefits and challenges of the CI process, and to find possible ways of addressing the identified challenges.

3.2 Research methods

The main part of the study was done with semi-structured interviews. Semi-structured interviews are common in case studies are suitable for qualitative research (Gray, 2009). The questions used for the interview can be found in Appendix 1.

Team leads from the development teams and Release Management were chosen for the interviews. In total 5 people were interviewed (referred to as S1-S5). The interviewees' experience at the company ranged from 3 to 8 years. The interviews were conducted in Finnish.

The interviews were recorded. Challenges and benefits were identified by listening to the recordings several times. The benefits and challenges were categorized as the structure of this thesis. The categorization was changed multiple times during the study, as new potential cause relationships between the challenges were discovered. Transcripts and their English translations were only made for relevant sections to be included in this thesis.

[2]The artifact repository is being replaced with a custom solution at the time of writing.

4 Identified benefits

The amount of benefits reported by the interviewees were relatively small compared to the challenges. The focus of the semi-structured interviews quickly shifted to the challenges during all of the interviews.

According to one of the team leads, Continuous Integration and its benefits is something that should be taken for granted in the software development process. CI can be seen as a standard practice for a modern software organization, which is one reason for it being implemented at the case company.

> *I don't know how any software company could state that they do good work without automatic testing. There has probably been automatic testing for as long as I've been at [the case company]. I think not having tests isn't even an alternative.* – **S1**

4.1 Faster iteration

A software development team is able to iterate on their project faster, as the CI pipelines gives continuous feedback about possible defects in the code. The possibility to automatically deploy the software also saves time. Automatic deployment to staging environments also makes it faster to manually catch e.g., UI bugs.

4.2 Assurance of quality

A well maintained set of automated tests makes it more likely that regressions would be found. From the perspective of the team developing the custom deployment tool and the Release Management team, they can be assured that the deployment packages delivered by the CI pipeline had passed their automatic tests. The customers are also more satisfied when there are less bugs. Developers also have started to take more responsibility of their work.

4.3 Easier deployment

Previously Capistrano[1] was used for both building and deploying the software. It was slow, and there was often issues with it. The development of a separate custom-made deployment tool has significantly decreased deployment issues.

> *If we compare to the situation where everything was done by hand, the errors and problems related to deployment have decreased significantly. Before the question "Why doesn't my Capistrano deploy work?" was asked almost daily, nowadays almost never because that tool is barely used.* [...] *We do less useless work when we don't build the packages repeatedly. When we have a web-based tool anyone can utilize it.* – **S1**

[1] Capistrano (`https://capistranorb.com/`) is a deployment tool written in Ruby.

5 Identified challenges

The challenges identified from the interviews were divided into four main categories.

5.1 Testing

5.1.1 Slow tests

Most of the challenges the interviewees identified had to do with the technical implementation of the automated tests. The biggest issue is the time it takes to run the tests. A problem with the test suite is that there is no distinction between unit tests and integration tests. While unit tests are supposed to be fast, integration tests take a long time to run since they interact with the database. Some tests also have sleep statements in their code, adding several seconds to the execution time. Because of this, it normally takes over one hour to run all the tests, which is way above an acceptable duration.

Due to some technicalities with the JRuby platform, it take a log time to execute single tests separately on a developer's local machine. This is due to the time it takes for the Java Virtual Machine to start and the Ruby code to be parsed. This has been looked into by an interviewee (S3), but a way to significantly alleviate this issue has not been found.

To alleviate to the tests taking a long time to run, they are run on multiple machines in parallel. Cloud providers such as Amazon Web Services makes it easy to start new test runner machines when needed. Using a queueing service makes it possible to run the tests in parallel across multiple machines. The drawback is that this requires work to implement. At the case company this has been successfully implemented using a custom Rake task in combination with the NSQ[2] queue service.

5.1.2 Non-deterministic (flaky) tests

Using sleep statements and concurrency features may result in the tests being non-deterministic. This means that the tests will fail on random occurrences. When the CI platform and the server the tests were run on was switched, some tests started to fail more frequently. The solution to this is simply to write correct tests.

[2] `https://nsq.io/`

This however requires a thorough understanding of the concurrency features of the programming language and testing framework used.

5.1.3 Test coverage not good enough

A consensus of the interviewees was that the current automated tests cannot sufficiently assure the quality of the software and its readiness to be released.

S3 suggested that the proprietary database could be tested more thoroughly, as much as commonly available database engines (such as SQLite) are tested. There could also be teams dedicated to improve the tests.

5.2 Infrastructure management

The CI solutions needs servers to run on. Acquiring this infrastructure can be difficult if the team managing the infrastructure is busy. The CI platform also needs to access external services, such as an artifact repository. Getting the connections to work between the CI servers and the external services has at times been difficult as it has required efforts from the infrastructure team.

The use of cloud providers such as AWS alleviates these issues to some extent, since it allows the person managing the CI software to also manage the infrastructure it runs on. This makes it easier and faster to get new servers when needed.

The cloud does not however come without challenges, when concerns like security has to be taken into consideration. The infrastructure team still manages the user accounts, permissions and network architecture. This may slow down the development.

> *During the last months we have been doing this transition to GitLab, and it has been kind of difficult on an administrative level since there isn't that one person who is deciding what we should do.* [...] *When there are separate entities, and it sometimes feels that they don't work well together and things move very slowly. Generally when we want to get something done in this organization, someone has to dedicate almost all of their working time to get the change pushed through from everywhere.*
> – **S4**

Infrastructure also needs to be actively maintained. Crucial internal services, such

as Git were running on old servers which warranty has expired. Transitioning internal services to AWS is being worked on, but it has been slow. An improvement suggestion was to dedicate one or a few persons to drive the needed infrastructure changes. It was also suggested that the SaaS versions of internal services (such as GitLab) could be used instead of their self-hosted version.

5.3 Pipeline complexity

At the case company, there are many different services running that are involved in the CD process. These services include multiple Git platforms and Jenkins instances, a chatbot and a custom test deployment service. The complexity of maintaining these was found to be a challenge.

The complexity of the planning-cloud software itself also makes it more challenging to maintain the CI pipeline. Multiple modules in the repository have to be built and packaged together. The modules are developed with different technologies and languages, requiring the invocations of their respective build tools and package managers (e.g., Maven, npm, bundler) to be orchestrated. The custom deployment tool also is also responsible for specific parts of the deployment process.

5.3.1 Culture issues

One if the interviewees noted that the lack of people dedicated to maintain the CD pipeline has led to a culture where issues are solved in a quick and unplanned way. The CD architecture consisting of many small pieces, and the lack of a vision of the big picture has also made it harder to maintain.

> *Many of these are made with kind of and ad-hoc approach, where nobody had really thought of the big picture. Now we have the situation where we have lots and lots of bits and pieces involved in the CI/CD process. Reasoning about the big picture is difficult when nobody remembers what pieces exist and what they are needed for. And now when we have started moving them to AWS, there has been a lot to think about, in what order and how they should be moved, so the complexity has been a bit of a challenge here. And hopefully we are beginning - or actually have already begun fixing this so the whole thing would be a simple as possible.*– **S1**

S3 thought that there is a firefighting culture in delivering new features to customers, and resolving problems related to them. S3 thinks that the developers spend too much time on solving customer issues, and that reducing the feedback cycle from tests could alleviate this.

> *Another thing which is especially appreciated in addition to making new features is that you put out fires. In a way there is kind of an incentive to not produce the best quality - but okay - to make new features, and then when we get problems we try to solve them, and whoever solves them, as well as the others get to feel good.* [...] *We have been discussing this for a while, and I can't say for sure, but some things that may help to some extent is when all feedback cycles get shorter. Then it's easier to test and verify things.* – **S3**

5.4 Lack of Continuous Deployment

Due to the nature of the software and contractual reasons, a full Continuous Deployment process cannot currently be employed at the case company. The software is used in business critical processes, and thus updating the customers' environments is considered to involve too much risk. The customers do not want the software to be updated without their permission, which is often also specified in their contract. Because of this, some customers are still using over two year-old versions of the software.

> *Some of our customers use our software for making nightly orders for thousands of stores with at least over ten million euros. Updating those systems requires a lot of validation work for a specific version in customer projects. The software coming from Development and Release Management is at the moment not of good enough quality in order for us to get by with generic validation.* [...] *We still have to use customer specific validation.* – **S5**

5.4.1 Maintaining multiple versions

This leads to additional challenges since multiple versions of the software have to be maintained simultaneously. Since the CI configuration and build scripts is stored

in the same repository together with the code, it has to be updated separately for each version when changes are made to it. The build process and environment also differs, so new features in the CI scripts have to be backported to older versions.

Often bug fixes can directly be cherry-picked to older versions, but sometimes additional implementation is necessary. The way the software is configured also differs between versions, which further complicates deployment. There may also be a lot of manual work (e.g., solving database migration issues) required when updating customer environments.

There is also sometimes a mismatch releases and customers' feature requests. This causes customer specific branches to be created in addition to the stable version. Often bug fixes are not cherry-picked to the customer specific versions.

> *Many customers have the case where there is some kind of schedule mismatch between our versioning and [the needs of the customer] They want some nice business features before they are in some release, [which leads to] the features going into a custom branch, and bug fixes won't be picked into it. Some fixes may be picked, but something might be missed or hard to adapt to the new features, or then there goes too much stuff which hasn't been tested, which leads to problems.* – **S5**

5.4.2 Customer specific configurations make automation harder

Since the software is highly configurable, it can be hard to verify that changes, such as updates to the database schema do not break functionality for the customers. Moreover, it can be difficult, if not impossible to make automatic database migrations that would cover every customer use-case.

> *Most of them [customer specific configurations] are such that migrations cannot be decided in advance. For example if some database schema is changed so that some data is not stored on product level anymore, but is moved to product-location level. We cannot always say what would be the reasonable default for all customers for some database field value, so we need to go though it in the project and check what it needs to be in that specific case, and then the changes are made by hand. It is things like this we can probably never automate.* – **S2**

The custom deployment tool is also lacking in functionality. It is too easy for users to make invalid configurations. Sometimes there are also manual changes (e.g., changes in Apache configuration) that need to be made with version updates that are not handled by the custom deployment tool.

5.5 Summary of interview findings

The findings from the interviews are summarized in Table 1. The solutions listed in the table are both ones that were suggested by the interviewees, and ones that have been suggested by the author.

Table 1: Summary of interview findings

Benefits	**Mentioned by**	
Faster iteration	S2, S3	
Assurance of quality	S1, S3, S5	
Easier deployment	S1, S2, S4	
Challenges		**Solutions**
Slow tests	S2, S3	Parallelization, selection, prioritization
Flaky tests	S2, S3, S5	Write better tests
Test coverage not good enough	S2, S5	Improve testing strategy Make acceptance tests continuous and improve visibility of their results
Infrastructure management	S1, S4	Outsourcing, increase collaboration
Pipeline complexity	S1, S4	Dedicated persons, clearer responsibilities
Culture issues	S3, S4	Improve collaboration
Lack of Continuous Deployment	S2	
Maintaining multiple versions	S1, S2, S5	Update customers faster, shorter release cycles, feature toggles, splitting the monolith
Customer specific configurations	S2, S4, S5	More controlled changes to configuration schema, improve deployment tool UI

From the table it can be seen that in most cases, each topic was mentioned by multiple interviewees. However, there was no topic that came up in all of the interviews.

6 Discussion

6.1 Related work

Most research seem to focus on going from a non-CD to a CD process. This is different from the context of this case study, where an existing CD process is improved.

A few literature reviews discuss the subject of this thesis. There findings of those are similar to the ones in this case study. (Debbiche et al., 2014; Rodríguez et al., 2017; Laukkanen et al., 2017; Hilton et al., 2017) In overall, the results of this study do not seem to conflict with the results of existing studies.

6.1.1 Benefits

In his article, Chen (2015) identified six main benefits of CD. In a systematic mapping study by Rodríguez et al. (2017), a similar set of benefits is listed.

Some of the benefits mentioned by Chen (2015) can be directly related to the interview findings. Those benefits are *Improved productivity and efficiency*, *Reliable releases*, *Improved customer satisfaction* and *Improved product quality*.

However, the benefits *Accelerated time to market* and *Building the right product* were not mentioned in the interviews. While CD can shorten the time between releases and the feedback loop between the company and the customers, this has not been the case at the case company, as the software is not continuously deployed. The lack of continuous deployment was identified as a challenge in itself. Moreover, some of the interviewees suggested that release cycles could be shorter. This is discussed further in section 6.6.2.

6.1.2 Challenges

All of the challenges identified in this study could be matched with corresponding challenges found in the case studies.

In a systematic literature review (Shahin et al., 2017b) of 69 papers, 20 challenges related to continuous practices were identified. Ten out of those 20 challenges could be matched to the ones found in this case study.

Some of the challenges mentioned by Shahin et al. (2017b) were not found in this study. It can be difficult to state whether a specific challenge is explicitly present at

Table 2: Challenges mentioned in (Shahin et al., 2017b) compared to this study.

Challenge	**Present at case company**
Lack of awareness and transparency	x
Coordination and collaboration challenges	x
Cost	
Lack of experience and skill	x
More pressure and workload from team members	
Lack of suitable tools and technologies	
General resistance to change	
Skepticism and distrust on continuous practices	
Difficulty to change established organizational policies and cultures	x
Distributed organization	
Lack of proper test strategy	
Poor test quality	x
Merging conflicts	
Dependencies in design and code	
Database schemas changes	x
Team dependencies	x
Customer environment	x
Dependencies with hardware and other (legacy) applications	
Customer preference	x
Domain constraints	x

the case company. Therefore the challenges seen at the case company, listed in Table 2, are ones that were directly inferred from the interviews. Some of the challenges have been an issue at the case company earlier, but later resolved. An example of this is "Lack of suitable tools", which has been resolved by developing own tools in-house.

"Skepticism and distrust on continuous practices" and "General resistance to change" were observed by some studies. However, none of the interviewees in this study seemed to show any skepticism towards continuous practices. Neither were cost issues observed as a challenge.

Shahin et al. (2017b) mention that CD in some cases may cause "increased workload and pressure from management" While the interviews didn't mention the increase of pressure as significant, it was mentioned that customers may want new features to be released as early as possible. This has led to custom releases for specific customers being made, and additional maintenance work is needed for those releases.

Supporting legacy software was not either mentioned by the interviewees, even though integrating with other systems is a significant part of the planning-cloud application. One reason for this might be that the integration work is mostly done by technical project teams separated from the main development teams which were interviewed.

6.2 Testing

The challenges related to testing are not unique to the case company. Multiple literature reviews and case studies on CI/CD have identified the same challenges related to testing (Debbiche et al., 2014; Laukkanen et al., 2017; Hilton et al., 2017). Some of the literature reviews discuss the following mitigations to testing related challenges.

6.2.1 Test prioritization

Test prioritization is a way to shorten the feedback time. This can be achieved by e.g., prioritizing tests if they have failed recently, or based on the code changes. However it seems like test prioritization has not been widely adopted in practice. (Chen, 2017)

Humble and Farley (2010) recommend that the unit tests should be in the "commit stage" of the CI pipeline, which with including the build stage preferably should take under ten minutes. In the case project the testing stage is around 25 minutes when the tests are parallelized. Since the test suites are run in parallel, a failure in a single suite can be detected earlier. It is still advisable to keep the test stage as short as possible, since the complete set of tests have to be run in any case.

Humble and Farley (2010) also suggest that the acceptance tests should be run in a separate stage after the commit stage. At the case company, the equivalent would be the black-box tests, which are run nightly. Humble and Farley (2010) assume that acceptance test environments are a limited resource, and thus, not every commit is necessarily tested. At the case company, a more continuous acceptance testing could be implemented, which also means that the testing environments would be used more efficiently.

6.2.2 Test selection

Test selection is another way to shorten the feedback time by omitting tests completely. This would be especially useful in a code repository with multiple modules. If test results from different modules are known to be independent, the results for each revision of a module could be stored. If a commit does not contain changes for a specific module, the tests for that module could be skipped since it has already been tested. The process of implementing this would however be manual, and one would have to manually assure that the modules' tests would be completely independent from each other. In other words, this approach would not work well for integration tests.

The are however ways to automate the test selection. A systematic review of test selection techniques (Engström et al., 2010) could not conclude that any test selection technique would be vastly superior. While there has been a lot of research about test prioritization and test selection, neither of them seem to be widely adopted in current field of software engineering.

6.2.3 Flaky tests

Flaky tests are caused mostly by concurrency problems in the test code. Luo et al. (2014) list three main reasons for flaky tests.

- **Async wait:** Checking for a result may happen before it becomes available, which is usually caused by `sleep` statements. This can be resolved by replacing the `sleep` statements with a `waitFor` mechanism.
- **Concurrency** issues with multiple threads, which may be caused by absence or incorrect use of guarding features, such as mutexes. The issue can lie either in the test code, or the code that is being tested:
- **Test order dependency:** differing ordering of the tests by the unit testing framework may cause tests to fail.

It is possible to develop automated tools for detecting flaky tests (Luo et al., 2014). Currently at the case company, only manual strategies are used to deal with them. When the CI pipeline is run more frequently, flaky tests are more likely to be detected. Solving flaky tests should be highly prioritized by the development team.

6.2.4 Parallelization

For running tests in parallel, some ready-made solutions are available e.g., Knapsack [3] (for Ruby) and Test Load Balancer [4]. Others have also had success with a similar parallel approach (Gopularam et al., 2012). However there does not seem to exist any widely used tools for parallelization.

6.2.5 Availability of test results

Rodríguez et al. (2017) points out that lack of transparency in testing activities can be a problem. An improvement for the case company could be to unify the test results from both the CI pipeline and the testing done by the Release Management team. While it was not explicitly mentioned in any of the interviews, the author did experience that the communication of testing efforts between the different team could be improved. The black-box and user interface tests could also be integrated into the CI pipeline, so that faulty build would be rejected and unable to be deployed, as recommended by Humble and Farley (2010)

[3] https://github.com/ArturT/knapsack
[4] http://test-load-balancer.github.io/

6.3 Infrastructure

Challenges related to CI infrastructure has been mentioned in other studies (Shahin et al., 2017b). Effort is required to set up the servers needed for the version control system and CI platform. This is often done by a separate infrastructure or operations team.

6.3.1 SaaS vs. self-hosted services

Services that support CI are often hosted on the software organization's own servers, but some companies offer their solutions as Software as a Service (SaaS), which is usually backed by a cloud provider. Using SaaS means that the providing company takes care of hosting the service. The clients do not need to think about managing servers and network infrastructure.

For SaaS there is trade-offs with cost and suitability for the use case at the case company. A more in-depth cost-benefit analysis on SaaS vs. the self-hosted version of GitLab would be needed. One potential drawback is that the CI pipeline is quite complex and heavily used. At first glance, it seems that the CI capabilities provided by the SaaS version of GitLab are not sufficient for the case company's use case. The most limiting factor is that the SaaS license only includes a limited number of minutes per month for which the CI pipeline can run.

6.3.2 Containerization

Containerization tools such as Docker have however made it easier to deploy applications, both in the cloud and on on-premise servers. Containerization allows applications to be deployed without having to care about the underlying operating system's installed packages.

Cloud providers also have their own container services. One example is Amazon's Fargate, which takes care of provisioning the underlying infrastructure for running Docker containers. The user only needs to specify the containers they want to run.

Many CI tools are available as readily deployable Docker images. This can make it easier for organizations to set up their CI infrastructure. Docker is also utilized by CI platforms to quickly set up environments for running builds and unit tests.

6.4 Culture

Most of the related research mentioned that collaboration and the lack of expertise in CI/CD are two main organization related challenges in implementing CI/CD.

Shahin et al. (2017c) surveyed how CD adoption may impact development and operations team structures. In most of the organizations surveyed, CD adoption had led to increased collaboration between development and operations teams. It was found that co-locating teams was one of the more common ways for organizations to enable collaboration. The challenges at the case company related to infrastructure management are mostly caused by limited collaboration between the Dev and Infra teams. One cause for this might be that the Infra team is physically located on a different floor, and thus makes collaboration harder. Shahin et al. (2017c) also mention that some companies have separate Dev and Ops teams, with a facilitating team between them. A similar pattern can be said to have emerged at the case company.

The difficulties in collaboration may also be caused by unclear responsibilities between the teams. As mentioned in the interviews, the lack of dedicated persons to maintain internal services has been a problem. The author suggests that grater efforts are made to clarify the responsibilities in regard of managing internal development infrastructure.

6.5 Pipeline complexity

CI/CD tools have to trade-off between simplicity and configurability (Hilton et al., 2017). This shows itself as added complexity to the CI pipeline. In this section, the author discusses his experiences with implementing GitLab as a CI tool.

6.5.1 GitLab as a CI tool

GitLab is a relatively new tool for supporting development teams. In addition to providing Git repositories for software projects, it can also function as a CI server. It also includes a simple issue tracking system. The author only found very few mentions of GitLab in previous academic work.

Hilton et al. (2017) recommend that the CI process should be made as simple as possible. They also mention that both simplicity and configurability is desired from

a CI tool, even though the two properties sometimes may conflict each other.

To mitigate the complexity of the CI pipeline at the case company, abstractions have been made in the form of Python scripts that automate the common tasks related to building and testing the application. Moreover, the switch from Jenkins to GitLab has provided a simpler, but more flexible way to configure the CI pipeline. While Jenkins is based on plugins that abstract away common tasks, GitLab is based on running arbitrary commands and relying on their exit code to determine the status of the pipeline. Jenkins' Groovy-based scripting language was hard to maintain and debug, and relies on global variables to determine the pipeline status. With GitLab, the custom Python scripts can be called. The Python scripts can also be run with a local debugger, which eases the development significantly.

Another aspect that add complexity is the caching of dependencies and build artifacts. For example, when a Java application is built, a set of .class files are produced. These files are when running the unit tests. The application may use dependencies from the Maven repository, which are stored in a local cache. The challenge comes when these files have to be passed from a finished build stage to a newly started runner machine which runs the unit tests. In GitLab, the files to be preserved have to be specified manually. Experimentation and knowledge of the build tools is needed to determine the files that need to be kept, although some CI solutions provide ready-made configurations for specific technology stacks. A fast network is also needed to keep the pipeline at a reasonable speed.

There has also been some reliability problems with the automatically scaling GitLab runners, that have not yet been fully resolved. The current solution is based on docker-machine and AWS, so the problem might lie in either those or in the GitLab software itself. CI solutions in the cloud are sill relatively young and developing rapidly, so hopefully in the future more stable solutions will be available.

Instead of waiting for the current tools to be updated, another option is to evaluate other corresponding tools, or have developers at the case company look into resolving issues with the tools currently used. It would be possible for the case company to make contributions to the tools, since many of them are open-source.

6.5.2 Preventing broken builds

GitLab supports merge requests (similar to GitHub's pull requests) which allow code reviews to be performed before a branch is merged. Merge requests can also

integrate with the CI pipeline, so that only passing builds are allowed to be merged. This avoids the commonly mentioned problem of "breaking the build" which prevents new releases from being made until the failing pipeline is fixed. This happens from time to time with the current pipeline implementation at the case company.

Git supports custom hook scripts which can be used to validate commits. For example it can be enforced that each commit in the master branch has to include an issue ID. Hook scripts have been extensively used in the case company's current git implementation to enforce a common workflow. For example, merge commits are not allowed and thus prevented. Integrating the hook scripts in GitLab has been a challenge. GitLab supports custom hooks, and even integration with merge requests, such that the hook errors would be shown if a merge failed. There has however been a bug that stopped this feature from working. A similar functionality to the hook scripts could also be implemented with GitLab's CI pipeline.

6.6 Lack of Continuous deployment

It is a common finding in related studies (Claps et al., 2015; Rodríguez et al., 2017; Shahin et al., 2017a), that CDep cannot be employed due to customer or business constraints.

Continuous Deployment is easier to implement when only a single version of the software product has to be maintained. This is usually the case for public facing web applications. Business-critical applications have different requirements, which makes CDep harder to implement.

6.6.1 Splitting the monolith

Continuous deployment could be implemented for less business-critical parts of the application, such as the user interface. This would however require the UI module to be split from the main codebase, and the existing deployment tools to be updated to accommodate that. Having separated modules also requires an API contract to be defined between them. According to one interviewee, such approaches have been discussed. Re-architecting the application would require significant effort from the development teams.

6.6.2 Rapid releases

Mäntylä et al. (2015) and Vesikivi (2016) found that automatic deployments is an enabler for having rapid releases. When there are less changes between versions, updates may become less painful. On the other hand it also requires having customers willing to update their software. Rapid releases may however conflict with the goal of having high reliability (Mäntylä et al., 2015).

In an internal workshop study at the case company aimed at improving the time between releases, it was found that poor QA and testing practices was the main perceived risk associated with a more rapid release cycle. At the same time, QA was also identified as the most important area to improve in. Splitting the monolith was also suggested as a way to make iterations faster.

7 Conclusions and recommendations

This case study aimed to identify challenges and benefits related to the practices of continuous integration and continuous delivery. The challenges and benefits that were identified were similar to what has been found in previous research.

Implementing CI/CD has had numerous benefits for the case company. Having a team dedicated to developing an automatic deployment tool has made deployment faster and reduced problems related to deployment. The introduction of CI/CD has also led to developers taking more responsibility in the quality of the product.

The identified challenges were mainly related to testing and collaboration within the company. Automated tests may be slow and fail randomly if they are not written with care. More effort should be put into ensuring the quality of the test code, especially in how waiting for asynchronous results is handled. Moreover, time should be spent on creating a testing strategy in order to improve the automated tests, so that they can better verify the quality of the software.

Collaboration was seen as a challenge by other studies in addition to this case study. In the case of the case company, the main challenge is a lack of dedicated persons to maintain the CI/CD pipeline and infrastructure. There are also technical challenges related to the complexity of the build process as whole, and the stability of the CI/CD tools used. Additionally, the collaboration between the development and infrastructure teams could be improved.

The critical use case of the software itself makes it hard to employ full continuous deployment. Customers do not want to update their version of the software when there is a risk of it breaking after the update. As multiple versions of the software are maintained simultaneously, applying bug fixes and maintaining the CI pipeline for each version requires additional effort. It is also hard to automate updates between versions when customer specific configurations have been applied to the software. Introducing a more rapid release cycle may alleviate this challenge by reducing the amount of changes between software versions, and thus making updates easier.

References

Amazon Web Services. Practicing Continuous Integration and Continuous Delivery on AWS: Accelerating Software Delivery with DevOps, June 2017. URL https://d0.awsstatic.com/whitepapers/DevOps/practicing-continuous-integration-continuous-delivery-on-AWS.pdf.

Atlassian. Continuous integration vs. continuous delivery vs. continuous deployment. URL https://www.atlassian.com/continuous-delivery/ci-vs-ci-vs-cd.

Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. *Manifesto for Agile Software Development*. 2001. URL http://www.agilemanifesto.org/.

L. Chen. Continuous Delivery: Huge Benefits, but Challenges Too. *IEEE Software*, 32(2):50–54, March 2015. ISSN 0740-7459. doi: 10.1109/MS.2015.27.

Lianping Chen. Continuous Delivery: Overcoming adoption challenges. *Journal of Systems and Software*, 128:72–86, June 2017. ISSN 0164-1212. doi: 10.1016/j.jss.2017.02.013.

Gerry Gerard Claps, Richard Berntsson Svensson, and Aybüke Aurum. On the journey to continuous deployment: Technical and social challenges along the way. *Information and Software Technology*, 57:21–31, January 2015. ISSN 0950-5849. doi: 10.1016/j.infsof.2014.07.009.

Adam Debbiche, Mikael Dienér, and Richard Berntsson Svensson. Challenges When Adopting Continuous Integration: A Case Study. In Andreas Jedlitschka, Pasi Kuvaja, Marco Kuhrmann, Tomi Männistö, Jürgen Münch, and Mikko Raatikainen, editors, *Product-Focused Software Process Improvement*, Lecture Notes in Computer Science, pages 17–32. Springer International Publishing, 2014. ISBN 978-3-319-13835-0.

Vincent Driessen. A successful Git branching model, January 2010. URL http://nvie.com/posts/a-successful-git-branching-model/.

Emelie Engström, Per Runeson, and Mats Skoglund. A systematic review on regression test selection techniques. *Information and Software Technology*, 52(1):14–30, January 2010. ISSN 0950-5849. doi: 10.1016/j.infsof.2009.07.001.

Brian Fitzgerald and Klaas-Jan Stol. Continuous software engineering: A roadmap and agenda. *Journal of Systems and Software*, 123:176–189, January 2017. ISSN 0164-1212. doi: 10.1016/j.jss.2015.06.063.

Martin Fowler. Continuous Integration, May 2006. URL `https://martinfowler.com/articles/continuousIntegration.html`.

Martin Fowler. ContinuousDelivery, May 2013. URL `https://martinfowler.com/bliki/ContinuousDelivery.html`.

B. P. Gopularam, C. B. Yogeesha, and P. Periasamy. Highly scalable model for tests execution in cloud environments. In *2012 18th International Conference on Advanced Computing and Communications (ADCOM)*, pages 54–58, December 2012. doi: 10.1109/ADCOM.2012.6563584.

David E. Gray. *Doing Research in the Real World*. SAGE Publications Ltd, Los Angeles, second edition edition, March 2009. ISBN 978-1-84787-337-8.

Paul Hammant. Trunk Based Development, 2017. URL `https://trunkbaseddevelopment.com/`.

Michael Hilton, Nicholas Nelson, Timothy Tunnell, Darko Marinov, and Danny Dig. Trade-offs in Continuous Integration: Assurance, Security, and Flexibility. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 197–207, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5105-8. doi: 10.1145/3106237.3106270.

Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Professional, Upper Saddle River, NJ, 1 edition edition, August 2010. ISBN 978-0-321-60191-9.

Md Rakibul Islam and Minhaz F. Zibran. Insights into Continuous Integration Build Failures. In *Proceedings of the 14th International Conference on Mining Software Repositories*, MSR '17, pages 467–470, Piscataway, NJ, USA, 2017. IEEE Press. ISBN 978-1-5386-1544-7. doi: 10.1109/MSR.2017.30.

Eero Laukkanen, Juha Itkonen, and Casper Lassenius. Problems, causes and solutions when adopting continuous delivery—A systematic literature review. *Information and Software Technology*, 82:55–79, February 2017. ISSN 0950-5849. doi: 10.1016/j.infsof.2016.10.001.

M. Leppänen, S. Mäkinen, M. Pagels, V. P. Eloranta, J. Itkonen, M. V. Mäntylä, and T. Männistö. The highways and country roads to continuous deployment. *IEEE Software*, 32(2):64–72, March 2015. ISSN 0740-7459. doi: 10.1109/MS.2015.50.

Emily Luke and Suzie Prince. No One Agrees How to Define CI or CD | GoCD Blog, May 2017. URL `https://www.gocd.org/2017/05/09/continuous-integration-devops-research/`.

Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. An Empirical Analysis of Flaky Tests. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 643–653, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3056-5. doi: 10.1145/2635868.2635920.

Simo Mäkinen, Marko Leppänen, Terhi Kilamo, Anna-Liisa Mattila, Eero Laukkanen, Max Pagels, and Tomi Männistö. Improving the delivery cycle: A multiple-case study of the toolchains in Finnish software intensive enterprises. *Information and Software Technology*, 80:175–194, December 2016. ISSN 0950-5849. doi: 10.1016/j.infsof.2016.09.001.

Mika V. Mäntylä, Bram Adams, Foutse Khomh, Emelie Engström, and Kai Petersen. On rapid releases and software testing: a case study and a semi-systematic literature review. *Empirical Software Engineering*, 20(5):1384–1425, October 2015. ISSN 1382-3256, 1573-7616. doi: 10.1007/s10664-014-9338-4.

Pilar Rodríguez, Alireza Haghighatkhah, Lucy Ellen Lwakatare, Susanna Teppola, Tanja Suomalainen, Juho Eskeli, Teemu Karvonen, Pasi Kuvaja, June M. Verner, and Markku Oivo. Continuous deployment of software intensive products and services: A systematic mapping study. *Journal of Systems and Software*, 123: 263–291, January 2017. ISSN 0164-1212. doi: 10.1016/j.jss.2015.12.015.

M. Shahin, M. A. Babar, M. Zahedi, and L. Zhu. Beyond Continuous Delivery: An Empirical Investigation of Continuous Deployment Challenges. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and*

Measurement (ESEM), pages 111–120, November 2017a. doi: 10.1109/ESEM.2017.18.

M. Shahin, M. Ali Babar, and L. Zhu. Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. *IEEE Access*, 5:3909–3943, 2017b. ISSN 2169-3536. doi: 10.1109/ACCESS.2017.2685629.

Mojtaba Shahin, Mansooreh Zahedi, Muhammad Ali Babar, and Liming Zhu. Adopting Continuous Delivery and Deployment: Impacts on Team Structures, Collaboration and Responsibilities. In *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*, EASE'17, pages 384–393, New York, NY, USA, 2017c. ACM. ISBN 978-1-4503-4804-1. doi: 10.1145/3084226.3084263.

Matti Vesikivi. Release management process in a software service company. June 2016.

Appendix 1. Interview structure

- What are your responsibilities
  - Regarding the delivery of our software?
- What do you see as the benefits of CD (having the product in a deliverable state, e.g., by using automated tests)
- What are challenging aspects of CD?
  - Implementation
  - Organizational
- How has the CD process evolved over time?
- Do you see any problems with the current implementation?
  - Do you have any suggestions on how to solve them?
- Any other comments about CD?