A practical approach to implementing Continuous Delivery
A case study at the Swedish Board of Agriculture
This exam work has been carried out at the School of Engineering in Jönköping in the subject area Computer Engineering. The author takes full responsibility for opinions, conclusions and findings presented.

Examiner: He Tan
Supervisor: Magnus Schoultz
Scope: 15 hp
Date: 2018-03-09
Abstract

This thesis has been carried out at the Swedish Board of Agriculture. The client wants to orientate towards a development environment and architecture that allows for more frequent software deliveries than in the current situation, to increase the business benefits of the development work carried out.

The purpose of this thesis is to develop a process to iteratively move towards an architecture and development environment that enable Continuous Delivery. Continuous delivery aims to a very high extent to treat a developer's code as part of a potential release candidate. This in turn causes high demands on being able to secure the reliability of both the infrastructure and the individual developers’ contributions.

The work has been carried out in cooperation with developers, infrastructure engineers, architects and team leaders on The Swedish Board of Agriculture. Theories have been tested within the IT organization to ensure their reliability and applicability in the organization.

A process has been developed with the limitation that it has not been tested in a production environment because of the limited timeframe available. However, it has been demonstrated that the process is feasible for the systems that acted as the main testing candidates during the thesis.

Keywords

IT architecture, Continuous Delivery, Continuous Integration, System Development, Software Quality, Testing Strategy
Sammanfattning

Detta examensarbete har utförts vid Statens Jordbruksverk. Uppdragsgivaren önskar att orientera sig mer mot en utvecklingsmiljö och arkitektur som möjliggör tätare leveranser än i dagsläget, för att öka verksamhetsnyttan av det utvecklingsarbete som genomförs.

Syftet med detta examensarbete är att ta fram en process för att iterativt kunna gå mot en arkitektur som möjliggör för Continuous Delivery, eller kontinuerlig leverans. Kontinuerlig leverans syftar till att i mycket hög mån behandla en utvecklarens kod som en del av en potentiell relesekandidat. Detta för i sin tur med sig höga krav på att kunna säkra tillförlitligheten av både infrastruktur samt den individuelle utvecklarens bidrag.

Arbetet har utförts i samarbete med utvecklare, infrastrukturtekniker, arkitekter samt teamledare på Jordbruksverket. Teorier har testats inom IT-organisationen för att se dess tillförlitlighet samt tillämplighet på just Jordbruksverkets organisation.

Arbetet påvisar att det är möjligt att dela upp monolitiska system och gå närmare något som liknar kontinuerlig leverans, utan att behöva genomföra stora förändringar inom organisationen.

En process har tagits fram med begränsning att den inte testats i produktionsmiljö på grund av tidsbrist. Det har dock påvisats att processen är gångbar för det system som varit testkandidat genom arbetets gång.
# Contents

Abstract ................................................................................................................. 3  

Keywords .............................................................................................................. 3  

Sammanfattning ................................................................................................. 4  

Contents ............................................................................................................. 5  

List of Figures ................................................................................................. 8  

List of Tables ............................................................................................... 8  

1 Introduction ................................................................................................. 9  
  1.1 Background .............................................................................................. 9  
  1.2 Purpose and research questions ............................................................... 10  
    1.1.1 Goals and purpose .............................................................................. 10  
    1.1.2 Objective .......................................................................................... 10  
    1.1.3 Research questions ........................................................................... 10  
  1.3 Delimitations ............................................................................................ 10  
  1.4 Outline ..................................................................................................... 11  

2 Method and implementation ....................................................................... 11  
  2.1 Choice of methods .................................................................................... 11  
    2.1.1 Action Research ................................................................................ 11  
    2.1.2 Case Study ....................................................................................... 12  
    2.1.3 Methods used in conjunction ............................................................ 13  
    2.1.4 Timeframe ......................................................................................... 13  

3 Theoretical background ............................................................................. 15  
  3.1 Software Quality ....................................................................................... 15  
    3.1.1 Factors affecting software quality ..................................................... 15  
    3.1.2 Test types ......................................................................................... 15  
    3.1.3 Testing Strategy ............................................................................... 17  
  3.2 Continuous Delivery Building Blocks ...................................................... 19  
    3.2.1 The Deployment Pipeline ................................................................. 19  
    3.2.2 Version Control ................................................................................ 23  
    3.2.3 Configuration Management ............................................................ 26  
    3.2.4 Continuous Integration ................................................................... 27  

4 Findings and analysis ............................................................................... 31  
  4.1 The Swedish Board of Agriculture ............................................................ 31  
    4.1.1 Server environments ....................................................................... 31
5 Discussion and conclusions .................................................. 58

5.1 Discussion of method ................................................................. 59

5.2 Course of action ............................................................................. 59

5.2.1 Problem definition ................................................................. 59

5.2.2 Data collection and analysis ..................................................... 60

5.2.3 Practical application .............................................................. 60

5.2.4 Defining the process .............................................................. 61

5.3 Discussion of findings ............................................................. 61

5.4 Conclusions and recommendations ........................................... 62

5.4.1 Conclusions ................................................................. 62

5.4.2 Recommendations and further work ...................................... 62

6 References ................................................................................. 63

7 Appendices .................................................................................... 65

7.1 Appendix 1: Reasoning behind choosing Git ............................. 65

7.1.1 Original text ............................................................................. 65

7.1.2 Translated text ................................................................. 65

7.2 Appendix 2: Example of how the style guide operates .............. 66
Appendix 3: Deploy script for the style guide ................................................................. 66
Appendix 4: Karma testing configuration ........................................................................ 67
Appendix 5: Puppeteer test, screenshots ......................................................................... 68
Appendix 6: Scripting queues and databases .................................................................. 69
Appendix 7: Trends for SVN and Git .............................................................................. 69
Appendix 8: Jenkins Pipeline script for Jorden .............................................................. 70
Appendix 9: Jenkins Pipeline regression script for Jorden ............................................. 73
List of Figures

Figure 1 The Improvement Kata workflow .............................................................. 11
Figure 2 An overview of all methods used in conjunction ........................................ 13
Figure 3 Gantt schema of planned phases ............................................................. 13
Figure 4 Test automation pyramid. Adapted from [12] ............................................. 16
Figure 5 The four agile testing quadrants. Adapted from [18] ................................. 19
Figure 6 Stages in a deployment pipeline. Cloned from [49] .......................................... 20
Figure 7 VCS revision flow. Adapted from [27] ......................................................... 23
Figure 8 Depiction of CVCS and DVCS. Cloned from [26] ........................................ 24
Figure 9 Version control system usage. Adapted from [34] ......................................... 25
Figure 10 Google search trends from September 2016 to August 2017 [50] .................. 34
Figure 11 Suggested pipeline design for Jorden ....................................................... 41
Figure 12 Screen capture from SBAs CI job for the Design System ............................ 43
Figure 13 Suggested pipeline for the Style Guide .................................................... 46
Figure 14 Example of release notes published with a release ................................. 47
Figure 15 Webhook and violation plugin in action .................................................. 50
Figure 16 A violation showing a merge conflict ...................................................... 51
Figure 17 Transparency of Jorden pipeline ............................................................. 52
Figure 18 Transparency of Jorden acceptance and regression testing ....................... 53
Figure 19 A Checkstyle violation in a sample project .............................................. 53

List of Tables

Table 1 Deployment pipeline and testing strategy ...................................................... 20
Table 2 Version control vocabulary ........................................................................ 25
Table 3 Popular repository hosting sites ..................................................................... 35
Table 4 VCS feature comparison ............................................................................. 35
Table 5 Pipeline stages for Jorden ............................................................................ 40
Table 6 Jorden pipeline and corresponding testing quadrants ................................... 41
Table 7 Pipeline requirements for the Style Guide .................................................... 48
Table 8 Pipeline requirements for Jorden ................................................................. 54
Table 9 Outlined steps towards Continuous Delivery ................................................. 57
1 Introduction

This chapter will give you a background into the nature of this study, and why the subject was chosen.

1.1 Background

The management at the Swedish Board of Agriculture (henceforth referred to as SBA), do an annual review of the goals and budget set for the fiscal year. These goals are revised annually and are generally achieved. One of the goals, which has been difficult to achieve, is the so-called Emergency Deployments. The goal is to have as few of these as possible, since they are hard to perform and require extra work.

Emergency deployments are deployments that must be done in addition to regular scheduled deployment times. They are intended to quickly fix problems that sometimes arise in conjunction with scheduled deployments. These problems can be difficult to correct and bring a lot of extra work if left unattended, or simply make a system do something undesirable that hampers the end users experience. Errors can increase the number of calls to the helpdesk, which in turn increases their workload when it could have been avoided. It is therefore desirable to address these problems as quickly as possible, and this is done in the form of an emergency deployment.

Before an emergency deployment can be done, stakeholders are convened to discuss potential damages if the error is left in the production environment. Conversely, problems that might arise from an emergency deployment are also discussed. It is therefore of the utmost importance that the developer(s) delivering the correction are aware of the possible consequences.

A drawback of emergency deployments is that testing of the proposed solution might be lacking, since the fix has to be delivered quickly. Unit tests are always expected, but not all requirements can be tested with this method. Load testing and non-functional requirements might not be met, other bugs or problems could be introduced into the production environment. Some tests are downright not possible to perform due to the amount of time they take.

To get a picture of the current situation, the goal set for emergency deployments in 2016 was three or less each month. The actual number in 2016 was 108 emergency deployments, which is nine times per month. The goal has thus been passed by 300%.

At the time of writing, the IT department at SBA has approximately 16 scheduled deployments each year. Each planned deployment might lead to one or more emergency deployments. The number of emergency deployments varies and is dependent on how many systems encounter errors in conjunction with their scheduled deployment. The interval between these scheduled deployments varies depending on the time of the year. Some periods require more frequent deliveries than others do because the number of users who would otherwise be impacted is higher than usual. If an error, bug or other problem is introduced during this time, an emergency deployment as to be done quickly.

This relatively small number of scheduled deployments can cause problems because bugs or other hard-to-detect problems that are found after deployment may have to wait a long time if they are not critical. For instance, a simple spelling error or something that does not impact functionality or user experience would have to wait for the next scheduled deployment. If they are critical for the functionality of the system, they are put through the process outlined earlier.

The management at SBAs IT department wants to strive towards minimizing the number of emergency deployments and introduce an easier way to handle the deployments of new releases. Continuous Delivery aims to solve this problem by having each system in a production ready state and having a built and deliverable system ready at all times [1]. It does so by subjecting each system to a rigorous test process in which it has to pass a series of
stages, each testing various parts of the system, and each stage is intended to increase confidence in the quality of the release.

## 1.2 Purpose and research questions

### 1.1.1 Goals and purpose

This thesis will try to define a process to help move an organization towards a continuous delivery approach. While this is a case study, great care has been taken to develop a process that could be applied to any organization, independent of chosen technology stacks.

A secondary goal, birthed from the main goal, is to introduce change into SBA by conducting experiments and evaluating different technologies and approaches. The change has to be meaningful, and therefore a sound understanding of the organization and the systems within it is crucial.

The field of continuous delivery is well documented, although most sources cover what it is and how to make it possible, but not an actual, hands-on approach to how to get there. Many developers, architects and managers find it alienating and difficult to grasp.

By conducting interviews, collecting data and performing tests at SBA, the process developed has considered every layer of an organization, i.e. managers, developers, architects, operations and database administrators.

### 1.1.2 Objective

The objective is to develop a process that is easy to follow and to get started with. By dividing the practical work into phases, the proposed process has a structure that is straightforward and easy to navigate.

Since the field of system development has a great diversity of roles and skills, the author has deemed it important to take the opinions of everyone that would be impacted by implementing into consideration.

### 1.1.3 Research questions

This thesis has been guided by one major research question:

1. Is it possible to develop a reusable process with clear guidelines to implement Continuous Delivery, for any kind of organization of arbitrary size and what would such a method look like?

Through this thesis, the author will answer this question and related questions that might arise during the process.

## 1.3 Delimitations

The concept of continuous delivery touch on many aspects of an organization, both technical and organizational [1], but the process will only focus on the technical aspects.

Recommendations and ideas about organizational changes will be presented, but not tested, since this would make the scope too large for this thesis. Presenting the ideas is deemed important by the author, since this might influence how the proposed process is implemented and by whom it is implemented.

Because this is a case study, choices of products and software might be limited to the chosen technical stack at SBA, i.e. Java Enterprise Edition. However, great care has been taken to make the process applicable to other technology stacks too, even if the proposed product is dependent on a specific technology.

Due to the limited timeframe, the process developed in this thesis will not be tested in a production environment. However, an actual system being used within the organization will be subject to all tests.
1.4 Outline
This thesis starts with an in-depth description of methodology and implementation in the chapter Method and implementation.

In the following topic, Theoretical background, the literature that has been reviewed for this thesis is explained to give a deeper understanding of the subject.

In Findings and analysis, the author presents the findings, decisions and how the proposed process for implementation would look like.

The last section, Discussion and conclusions, addresses the authors concerns with the findings, how the work could be improved, and the problems that occurred. The thesis ends with a reference list and appendices.

2 Method and implementation
This section will describe and justify the choice of method as well as how this method was planned and realized in practice. How the work was planned and what techniques that were used to achieve the results of this thesis, are also presented.

2.1 Choice of methods
The overall methodology used in this thesis is a combination of case study, experimental and action research approach. The implications and thought behind this methodology are discussed in Methods used in conjunction later in this chapter.

The case study involves understanding SBAs current situation and environment, the experimental approach involves incrementally introducing key aspects of continuous delivery into the organization.

During the later stages, an iterative approach using the Toyota Improvement Kata has been used in conjunction with Action Research. This method has been used to develop the pipeline, and the knowledge gained from the first phase in this thesis has been used to establish target conditions.

2.1.1 Action Research
Since the area of Continuous Delivery is huge and for most companies a question of years of continuous improvement to implement [3], this thesis has been using an incremental approach to answer the research question stated in the introduction and to retain an achievable scope.

In addition to trying to answer the research question, a goal has been set for each increment. The purpose of these goals is to achieve a measurable result instead of an intangible opinion.

Lean Software Development (henceforth referred to as LSD), in which continuous delivery plays a large part, is adapted from the Toyota Production System [4], so the method used in this thesis is based on the process put forth by Mike Rother in his book Toyota Kata: Managing People for Improvement, Adaptiveness and Superior Results.

Mike Rother calls this process the Improvement Kata [5]. The model has been proven to work within the area of computer engineering and was first mentioned in this context by Tom and Mary Poppendieck in their book Lean Software Development: An Agile Toolkit. This model is a systematic, scientific pattern of working with continuous improvement and the workflow is illustrated in Figure 1.

Figure 1 The Improvement Kata workflow
This model lends itself well to an experimental approach and aims to achieve measurable goals quickly. Mike Rother boils the process down to the following:

...in consideration of a vision, direction, or target, and with a firsthand grasp of the current condition, a next target condition on the way to the vision is defined. When we then strive to move step by step toward that target condition, we encounter obstacles that define what we need to work on, and from which we learn [5].

Since one of the goals of this thesis is to implement change into SBA, the improvement kata aligns well with the goals and is suitable for introducing modern technology and concepts on a trial basis.

This method closely resembles that of Action Research (AR). AR contains five phases [6]:

1. Identifying problems
2. Action planning
3. Implementation
4. Evaluation
5. Reflection

When a satisfactory outcome has been achieved using the improvement kata, and the goal set for the iteration has been reached, phases four and five of the action research method will be used to evaluate and mediate the results to those who might be concerned within the organization.

The reason for using the improvement kata to tackle the experimental approach instead of the first three steps of the AR method are threefold:

1. It promotes the use of an experimental approach to reach the envisioned goal, compared to action research which promotes a more structured and planned approach based on the second step of the process, as shown above. The experimental approach will help to achieve greater external validity of the process being developed.

2. This thesis does not have a real implementation phase, since the experimental approach might not yield the desired result. In action research, a solution is implicit to the method.

3. The kata will be iterated over several times and is easier to adapt to a limited period.

2.1.2 Case Study

A case study approach will be taken in the initial stages of this thesis to get a better understanding of the needs of the organization.

To start to implement change that will be lasting and meaningful for those involved a sound understanding of the problems we set out to solve is crucial. We also need to get a better understanding of what caused these problems, so we do not put ourselves in the same situation again.

Using this methodology, the author hopes to get a clear picture of current obstacles within the organization and determine what cultural and organizational factors that has led to the current situation.

Implementing a continuous delivery approach is not only about making a developer’s job easier; it permeates every layer of an IT organization, and even beyond IT. It is therefore important to get a clear picture of the wants and needs of the many. This will also help narrow the experimental approach and set clear and concise goals.

Robert K. Yin has described the case study method as follows.
...an empirical inquiry that investigates a contemporary phenomenon with its real-life context; when the boundaries between phenomenon and context are not clearly evident; and in which multiple sources of evidence are used [2].

The above underlines the two main goals of this phase of the thesis; to get a better understanding of existing phenomena within the organization and how these phenomena arose.

Data gathering in this phase has been done primarily via interviews, discussions and observations within SBA.

2.1.3 Methods used in conjunction

The overall process used in this thesis closely resembles AR.

The case study of SBA correlate to step one and two in the AR cycle, the knowledge gained from these will then be used in several iterations of step three to find suitable solutions for the research question.

This is where the improvement kata is used to try to implement the knowledge gained and help form valid conclusions that will then be used to create the process. Iterating during this process will help identify obstacles and concrete ways to overcome these obstacles.

Finally, step four and five will guide the final phase of this thesis. Here, the author will determine the applicability of the process both internally at SBA and externally for other organizations. Figure 2 shows how these methods will be used.

2.1.4 Timeframe

The initial timeframe for each phase is displayed in the schema in Figure 3.
An additional eight weeks after the final report writing week above will be used to correct any errors pointed out during the final phase of the course.
3 Theoretical background

The theoretical background represents the basis for the result of this thesis.

3.1 Software Quality

To be able to deliver software continuously and with high frequency, it is important to secure the quality of said software. Quality is assessed through both automatic and manual verification to make sure functional and non-functional requirements are met.

To gain any advantage from Continuous Delivery, it is important to have a solid strategy for how and what to test.

3.1.1 Factors affecting software quality

According to Chemuturi, overall software quality has four dimensions [10]:

- Specification quality
- Design quality
- Development quality
- Conformance quality

This thesis will limit its scope to development quality, since it is the only applicable factor because of the nature of Continuous Delivery. Continuous Delivery can only guarantee development quality, but the development quality in turn is dependent on each of the other dimensions. Thus, it is important to understand the other factors and how they correlate to one another.

*Specification quality* determines the quality of the initial specification [10]. What is our system expected to do? This is the foundation of how our system should work and what the end user expects. The specification should act as a platform for a test suite; it outlines the expected outcome of these tests.

*Design quality* refers to the quality of the design of the delivered product [10]. This factor directly correlates to the specification. With a weak specification, there is no way for the designer of the system to determine how it should work.

*Development quality*, Chemuturi argues, is achieved by adhering to established coding guidelines of the programming language being used. [10]

Finally, *conformance quality* determines how well quality has been built into each of the other dimensions deliverables. It is the responsibility of everyone involved in the development lifecycle to ensure that that quality is sufficient.

Identifying key persons in the lifecycle and educating or retraining them on how to work with software engineering requirements is key. Education also increases the sense of responsibility and ownership of the people involved. [11]

3.1.2 Test types

To maintain development quality, running various kinds of tests against the software is crucial. There are many types of tests, performed by both developers and business experts. Tests can be visualized as a pyramid, as seen in Figure 4.
This is just one of many ways to visualize tests, but it goes hand in hand with the continuous delivery mind-set. If each contribution to the code base should start a test suite, running tests that are fast and does not heavily affect performance on the build server is essential. If the cheap and fast unit tests fail, continuing to run the suite is not necessary.

3.1.2.1 Unit tests
At the most granular level, developers write Unit Tests. The purpose of a unit test is to verify that a small piece of code, or unit, performs as expected. [13] For example, what would happen if a developer writes a piece of code that should divide candies between kids if the number of kids is zero, if it is null or something invalid, like a letter?

This would throw an exception or crash the program entirely, telling the programmer that they have made an error. The developer then fixes this error, by telling the user that a valid number of kids should be more than zero, only numbers, and that it has to be set (not null). The developer then writes a unit test for each of these cases to verify that the intended behavior is enforced.

Now, if another developer unintentionally circumvents this functionality, the unit test will break the build and notify the developer.

3.1.2.2 Regression tests
Any kind of test that has passed becomes part of a regression test suite. The purpose of this suite is to guard against unintentionally introducing bugs [13]. This way, teams can make sure previously implemented functionality has not been broken in the process of implementing new functionality.
3.1.2.3 Component tests

Component tests are like unit tests, but with a higher level of integration. These tests are usually run in the context of the application, ensuring it works as intended. A component is a larger part than a unit, and might require mocking. For instance, a developer or tester might want to verify that several units form a cohesive component, and that this component acts as expected.

3.1.2.4 Acceptance tests

“Acceptance tests are tests that define the business value each story must deliver”. [13] These tests may verify functional or nonfunctional requirements, and may or may not be used as developer guidelines.

3.1.2.5 Smoke tests

Smoke tests are a subset of automated acceptance tests. The purpose of these tests is to quickly find severe failures and ascertain that the most crucial functionality works. [14] An example would be to simply check if the application started as expected.

3.1.2.6 GUI tests

According to Cohn, limiting GUI tests (graphical user interface tests) should be done for a couple of reasons [12):

- They are brittle. If the user interface undergoes layout changes, chances are these tests will break.
- They are expensive to write. The easiest way of writing a user interface test is by using a capture-and-playback software, but these tests are extremely brittle and prone to error even in slight changes to the interface. They are expensive to run, because you must have the interface available, i.e. headless testing is not possible.
- They are time consuming. User interfaces have to be reloaded to try different states, and the number of tests to try different scenarios will quickly build up.

3.1.2.7 Manual tests

Manual tests are necessary for a few reasons [12]:

- Every test-aspect of a system is impossible to automate
- Tests might be too resource intensive to automate
- Tests might require physical intervention

Manual testing is a way of doing exploratory testing. This is an interactive form of testing and focuses on learning the application and finding errors through normal use cases. It can also help in identifying missing use cases, missing test cases and even functionality that does not serve any purpose in the finished product [15].

These tests are placed on top of the pyramid indicating it is balancing on the GUI tests. This is because there is a need to strike a balance between what should be automated without being too brittle, and what should be done manually. This can only be decided with knowledge of the system. For instance, of a user interface is expected to change often, it is cheaper to verify the functionality within it manually.

3.1.3 Testing Strategy

A testing strategy is vital to maintaining good health and quality of any system, to keep non-technical stakeholders informed about the progress of the development and to clearly signal to a developer when they are done. Developers should write their unit tests, but this only confirms that a developer’s code works in isolation. This is not enough, even in projects of moderate size. To battle the issues of integrating with others, a solid testing strategy of how to do so is needed.
To make sure that software is performing to according to all specified requirements, a test suite is run against the software. To ensure the requirements are met, the test suite should be written in collaboration between testers and developers before any implementation begins. This way, it can be used to confirm the code base each time a contribution is made; it will act as an executable specification and therefore facilitates automatic testing [15]. A Continuous Integration Server is responsible for the execution of the suite.

Brian Marick is the author of an approach that separates testing into four distinct areas [16] [17].

Marick also coined the term technology-facing tests, which are tests described in a developer’s jargon. He gives an example of this in the form of the question:

Different browsers implement JavaScript differently, so we test whether our product works with the most important ones [16].

On the opposite side of the spectrum are business-facing tests, which describe the systems functionality from the point of view of a business expert. Marick also gives an example question that would engage a business expert:

If you withdraw more money than you have in your account, does the system automatically extend you a loan for the difference [16]?

These test categories are further separated by defining if a test should critique the product or support programming. The former looks at a finished product to discover inadequacies, and the latter serves the purpose of telling a programmer when their work is finished. These tests put together also serve the important purpose of easily detecting errors when refactoring. The programs behavior might change and that is detected by the technology-facing tests, but if our business-facing tests are not fulfilled anymore, something vital broke along the way.

Having defined these four categories, the strategy can be visualized in a matrix, as seen in Figure 5.
3.2 Continuous Delivery Building Blocks

Continuous Delivery is a concept in System Development and IT Architecture. It is an umbrella term and consists of many different technologies to form a unified approach to delivering software. The goal of all these technologies is that they should work in tandem to make deploying new artifacts quick and painless by having the code base in a constant production ready state [2].

Ultimately, the goal is to be able to deliver continuously. This is done by having the system in a constant deliverable state where you are entirely sure of its quality and stability.

In this chapter, the building blocks that make up Continuous Delivery are explained, and their purpose in the ecosystem is expanded upon.

3.2.1 The Deployment Pipeline

The Deployment Pipeline is the backbone of CD. It is also known as a staged build or build pipeline. The pipeline utilizes all the previously discussed principles and orchestrates them to work together. Its purpose is to automate large swaths of the workflow; from build, to test, to deploy and finally the release. A sample of a pipeline can be seen in Figure 6.

---

Figure 5 The four agile testing quadrants. Adapted from [18]
Figure 6 Stages in a deployment pipeline. Cloned from [49]

As can be seen above, every commit to the VCS triggers the pipeline automatically. As discussed in the chapter Continuous Integration, this automatic triggering would replace the need for a rubber chicken and a bell entirely.

As each step is passed, the confidence that the revision is without bugs increases. As the confidence increases, the artifact is put through more stages of testing, each more production−like and resource intensive than the last.

Because of this increase in resource consumption in later stages of the pipeline, it is desirable to perform the least resource intensive tests first, as seen in the test automation pyramid in Figure 4.

By doing this, the pipeline optimizes the build by trying to fail as early and fast as possible, thereby speeding up the feedback loop. Quicker feedback means developers can quickly begin to correct the error and bring value to the product.

The stages of the deployment pipeline are mirrored in the testing quadrants discussed in chapter 3.1.3. This shows the importance of a well thought out testing strategy, and how it mitigates a lot of hard-to-detect errors in production environments. Without a testing strategy, the deployment pipeline will be lacking vital steps.

Steps in the pipeline graph above are reflected in the testing quadrants, as can be seen in Table 1.

<table>
<thead>
<tr>
<th>Table 1 Deployment pipeline and testing strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Testing Quadrant</strong></td>
</tr>
<tr>
<td>Q1, unit &amp; component tests</td>
</tr>
<tr>
<td>Q2. (automated) functional tests</td>
</tr>
<tr>
<td>Q2. (manual) functional tests</td>
</tr>
<tr>
<td>Q3, exploratory testing</td>
</tr>
</tbody>
</table>


It is important be aware of the word *manual* in the figure and table above. The deployment pipeline is by no means meant to eliminate all manual intervention, but the steps that require a lot of work that are easily automated instead.

3.2.1.1 Best practices
Humble and Farley outline six practices that the pipeline should follow to give the best value as quickly as possible. [49]

First, a system should only build its executables or binaries once. Building the executables with different system settings might introduce errors, and building once allows us to eliminate these risks. For instance, the version of the compiler installed might differ or it might have a different configuration entirely, you might get a different version a third-party dependency and so on. We run the risk of deploying something that was not tested in its entirety to production.

When the build is performed once, there is one executable and a single source of truth as to how the system will act through the test stages. If you have to rely on many binaries or executables aimed at different environments, you are introducing a lot of unnecessary overhead on your CI server. Chances are that you are not handling your configurations in a correct way, and that dependencies to environments are hard coded. To top it off, rebuilding also violates the efficiency of the pipeline.

Second, "deploy the same way to every environment." When you can successfully deploy using the same script to any of your environments, you can rule out your deployment pipeline as a source of error. This also forces you to adhere to configuration management practices. Using the same pipeline effectively eliminates "it works on my machine" errors.

Third, smoke test your deployments. Smoke testing is crucial, and it is an efficient way to quickly give feedback to developers if any of the tests fail. If these tests fail, something major is out of order and the failed test should give basic diagnostics as to what went wrong. As soon as the unit tests pass, this should be the next suite to run.

Fourth, "deploy into a copy of production." Production is the environment where the least number of deploys are actually made, and as such it won't get tested as frequently. Because of this, it is important that the testing environment that the pipeline utilizes is as close to a production clone as possible. Ideally, we would want full control over operating system patch levels, infrastructure such as firewalls, the entirety of the application stack and its dependencies.

Fifth, each change should trigger the pipeline instantly. If many developers are working and utilizing the same pipeline and each commit triggers every phase, a queue will form quickly, and outdated code will be tested, in lieu of the latest.

Imagine four commits occur simultaneously, a race condition occurs and the first commit to pass the unit tests will then initiate the next stage and block it from the others. We now have 3 commits waiting to utilize a single phase of the pipeline while another one finishes. During this time, even more commits might be queued up against the pipeline. To mitigate this scenario, your choice of CI server should have a smart scheduler. The first build triggered the first phase (unit testing) and finishes. The CI server now checks if there are several commits to process and if that is the case, it will choose the latest and initiate the unit testing with that commit. This will repeat until there are no commits queued up, and the latest build can progress through the pipeline.
Lastly, *stop when failure occurs*. This is extremely important, as the whole purpose of the pipeline is to give stakeholders quick and reliable feedback. It is also a waste of resources to continue testing if a step has failed, since that means the software is broken.

### 3.2.1.2 Stages of the pipeline

A pipeline is split into several stages, each more resource intensive than the last.

The first of the stages is the *commit stage* [49]. The commit stage is responsible for compiling, running commit tests, creating executables and performing analysis on the code. If compiling fails, the issuing developer probably has not tried to merge their changes before committing. If this is the case, the build should fail immediately and notify the developer.

If the compilation is successful, a specialized test suite call *commit stage tests* should run. This suite is optimized to run very quickly, and is a subset of unit tests. The purpose of these tests is to be quick and give a greater certainty that the code we propagate forward indeed works.

Non-functional requirements that are not met should also make this stage fail. Examples of non-functional requirements to test for are test coverage, duplicate code, cyclomatic complexity and afferent and efferent couplings [49]. These types of statistics can quickly give developers an idea of the state and health of the code.

Lastly, if all the aforementioned tests pass, an executable should be created, and all unit tests associated with it should pass. This executable is the basis which all other tests will be performed on as the pipeline progresses through its stages. This executable is what is referred to as a *release candidate*. It is not yet a release, but it has the potential to be.

The second stage is running *automated acceptance tests*. Unit tests are great and cheap to run, but there’s lots of errors that they won’t catch. It is common practice for developers to make sure their code works by simply running unit tests, and then committing. [49] However, unit tests do not ensure that the application will start and perform as expected.

This is where acceptance tests come in to play. The purpose of these tests is to identify that the product meets customer’s specification and to verify that no bugs have been introduced into existing behavior. This test suite should be at the heart of the development process and not be maintained by a single team, but rather cross-functional team of many disciplines that know exactly how the system should behave.

If a test in this stage fails, it might not be anything wrong with the software. From case to case, it is up to the developers to determine if the breakage was caused by a regression, an intentional change of behavior or if something is wrong with the test. [49] If your software should target many different environments, the acceptance tests should also run in clones of each environment.

Finally, when a release candidate has passed the commit stage and the automated acceptance test stage, it is placed in a staging area. Or, for the simplest of pipelines, the release candidate is now a release and can be released into production. However, in most cases, some manual verification is usually needed.

When the release candidate is in the staging area, other stakeholders can choose to deploy it to other environments to continue testing. The stages that follows here depend entirely on the requirements on the system, but some form of manual verification and verification of non-functional requirements take place. Testing for non-functional requirements here usually require some tools. Testing for security and capacity isn’t trivial and is highly related to the system, and hence it won’t be covered here.
3.2.2 Version Control

The practice of storing revisions in a central repository is referred to as Version Control, Revision Control or Source Control. This is facilitated by using a Version Control System (VCS) like Git, Subversion (SVN), Concurrent Versions System (CVS) or Team Foundation Server (TFS).

However, the simplest form of version control could be to simply have a folder on the file system with a unique name, and then copy the project to that folder as a way of saving different revisions. However, this is not something that should be used since relying solely in the file system can be dangerous and files might be lost forever [26].

Having version control is a fundamental building block to achieve Continuous Delivery. The VCS will act as a single source of truth, and every developer should contribute to the same source of truth. With this, there is no way to keep track of which version is deployed where, and there is no simple way for developers to collaborate.

All the aforementioned tools work differently but perform the same job: handling revisions of files. A revision contains information like a unique identifier, a timestamp and the changes made to the file. A VCS “allows you to revert files back to a previous state, revert the entire project back to a previous state, compare changes over time, see who last modified something that might be causing a problem, who introduced an issue and when, and more.” [27]

The flow of a VCS is commonly represented as a tree, as seen in Figure 7, with branches (orange) merging back to trunk (blue). With this information, tags (green) that contain information from trunk in that specific revision can be created and stored as artifacts.

![Figure 7 VCS revision flow. Adapted from [27]](image)

A number that specifies its chronological order depicts each revision above. In reality, this would most likely be a revision number or a timestamp, depending on the VCS being used.

Over time, VCSs have developed and adapted to new requirements. The two most commonly used approaches today are Centralized Version Control Systems and Distributed Version Control Systems [26].

Centralized VCS uses a centralized server that contains every revision for every file. Any number of developers on any number of machines can then fetch these files and start working from that revision. SVN is a well-known CVSC. Figure 8 shows how this works in practice.
Theoretical background

Distributed VCS works by having each developer clone the entirety of the repository they want to work on. This means that it combines the functionality of a local VCS that is independent of any connection to a server, and that of a CVCS that stores revisions on a central server.

Whether the system is distributed or centralized, a branching strategy is needed. Without a branching strategy, a project’s repository might quickly become unwieldy. A branching strategy dictates how each developer works with hotfixes, features and releases and how these get pushed and merged to the mainline.

3.2.2.1 Noteworthy differences between Git and SVN

As mentioned, the two major players in the different schools of version control are Git and SVN. Besides being distributed and centralized, respectively, there are a few other noteworthy differences.

In Git, anyone can work on any branch, and thanks to its decentralized nature, can utilize version control in their local development environment. [28] A developer does not have to wait to commit a substantial chunk of changes if they don’t currently have the right permissions for the repository.

Compared to SVN, a developer would have no version control of their contributions until the repository owner gives them access to commit to the repository.

Git also handles storing very differently. In the moment of writing, the entire repository for the Linux kernel is only 1939563 kb\(^1\), or about 1.9 GB for years of development and hundreds of thousands of commits. For instance, the entirety of the Mozilla project takes up about 12 GB in SVN, and only 420 MB in Git. [28]

Another key difference is speed. Since the whole repository is fetched to the developers’ local environment, operations like diff\(^2\), commit and merge is nearly instantaneous since all network latency is avoided. Independent tests show that on average, Git is at least twice as fast, on average, independent of the operation being carried out. [29]

However, there are a few drawbacks to Git that might be worth considering. It is complex and has a lot of command line syntax [30], making the learning curve steep. Repositories that are

---

\(^1\) https://api.github.com/repos/torvalds/linux
\(^2\) An operation that shows differences between commits
truly huge, like the GNOME project [31], become unmanageable in a single repository in Git because of its sheer size.

SVN also has superior authorization tools. SVN provides path-based authorization [32], whereas with Git the developer has access to the whole repository and all its branches once it has been pulled.

Git is also not very friendly to binary files, since it works under the assumption that all files should be able to merge. In SVN, users can set locks for certain files which would prevent merging of binary files [33].

### 3.2.2.2 Proprietary version control systems

There are many proprietary version control systems available, but the far most well-known and widely used is Perforce, as can be seen in Figure 9.

![Figure 9 Version control system usage. Adapted from [34]](image)

Out of the other version control systems in the chart above, only Perforce is proprietary and has a market share according to the survey.

Perforce is used by giants like NASA, CD Projekt Red, NVIDIA and Homeland Security [35].

Google relied on a single Perforce server for over 11 years before developing their own system called Piper. Before this server was replaced by Piper, it handled an impressive 12 million commands on average day, had a changelog of over 20 million and hosted projects with over six million files [36] [37].

### 3.2.2.3 Version control vocabulary

Git and SVN share concepts, but some of the vocabulary is different between the two technologies. In Table 2 the terms are explained, and differences between the two are highlighted.

<table>
<thead>
<tr>
<th>Term</th>
<th>Meaning</th>
<th>Git</th>
<th>SVN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>A healthy revision from which all branches should be cloned</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Branch</td>
<td>A set of files that can be worked on individually from other branches</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Theoretical background

<table>
<thead>
<tr>
<th>Merge</th>
<th>Including changes from one branch into another</th>
<th>rebase / merge</th>
<th>merge</th>
</tr>
</thead>
<tbody>
<tr>
<td>Working copy</td>
<td>The local copy on the developer’s machine</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Checkout</td>
<td>Getting files from the server</td>
<td>clone</td>
<td>checkout</td>
</tr>
<tr>
<td>Fetch</td>
<td>Retrieve files from the central server</td>
<td>pull</td>
<td>update</td>
</tr>
<tr>
<td>Commit</td>
<td>Commit code to the repository.</td>
<td>commit → push</td>
<td>commit</td>
</tr>
</tbody>
</table>

- The “trunk” from which all branches are branched from

| Mainline       |                                             | master         | trunk |

3.2.3 Configuration Management

Configuration Management (CM) “refers to the process by which all artifacts relevant to your project, and the relationships between them, are stored, retrieved, uniquely identified, and modified.” [38]

By adhering to the above quote, it will be possible for any developer to setup a system in their own development environment. This is an important aspect of Continuous Delivery as it allows a system to be checked out and configured without having to know where to look for all resources to set up the system correctly. This in turn allows a system to be set up for automatic tests without any human intervention to do so.

The approach of storing configurations like this is common, not only in the field of Computer Sciences (CS), but also within the military and in other engineering professions. [39] [40] To achieve this, a way of tracking changes to our configurations is necessary. This is commonly achieved by using a VCS, as discussed in the previous chapter.

The minimal criteria to begin to undertake CM is having version control in place. Version control allows an organization and its teams to store, organize and control revisions of their software. Having this in place allows an organization to not only have their code revisions under control, but also place all environment configuration in one place. Without version control, collaboration between teams becomes a very arduous process.

It is not a requirement to host configurations in the same location as application code. To make use of the pattern, a specific version of the application code has to correlate some way to the configuration of its environment [38].

Every little detail about applicable environments should be visible and stored in version control. If a crash occurs unexpectedly, a copy should be able to be up and running in as little time as possible, a so-called Phoenix Server [41].

If all available configuration is stored in version control, a server can then be restored to any time for which a revision exists. By using CM, we also eliminate so called Snowflake Servers that are difficult to reproduce and modify. This makes clustering and scaling of environments and systems difficult [42].

Storing the configuration in version control servers another vital purpose; it allows rolling back entire environments to known states that work.
Theoretical background

Jez Humble and David Farley argue that you have good CM if you can reproduce any environment, including associated version of the operating system and its patch level, the entire software stack, every application deployed and finally each application’s unique configuration, essentially a phoenix server. This process should, however, in no way hinder efficient delivery of code. If this is the case, the benefits that CM gives are lost since it hampers the ability for quick feedback [38].

3.2.3.1 External libraries
Managing externally fetched libraries is also extremely important. When utilizing external repositories, there is always a risk that it won’t be accessible when it is needed or that it is removed entirely.

In March 2016, a single developer managed to break thousands of projects in npm by simply unpublishing eleven lines of code. This broke a lot of major projects in the JavaScript ecosystem, emphasizing the importance of managing external libraries responsibly [43] [44].

3.2.3.2 Key parts of configuration management
To be successful with configuration management, there are three main kinds of configuration that needs to be considered.

Software configuration refers to any configuration that can alter the behavior of software. Having flexibility is good, but it comes at a cost. It can be hard to predict the behavior of highly configurable application. More corner cases arise, and the testing has to become more flexible.

Having many configurable parameters also comes with risk. It might be forgotten in a check in, there might be misspellings, and encodings might be off, to name a few reasons. It is as easy to break an application by modifying its configurable parameters as it is to break it when accessing its source code.

Server configuration refers to any parameters that can be changed on the server. This includes what ports are open, what IP address the application can be accessed at and database and messaging server accessibility.

Environment configuration should store everything related to the applications environment. This includes OS, patch level, the software that stack that is needed for the application function and users that can access the application.

3.2.4 Continuous Integration
Continuous Integration is an integral part to Continuous Delivery. The purpose of the practice is to ensure that the code that is produced integrates nicely with other modules, systems and units within the environment. With many developers contributing to the same project, this becomes very important.

The idea behind continuous integration is that, if regular integration of your codebase is good, why not do it all the time? By using this principle, a few unpleasant scenarios are more easily avoided. As is apparent from the name, it is the practice of continuously integration other developers code into your own. This is done by either pulling one branch into another. In SVN this entails a merge, and in Git it is either a rebase or a merge.

Integration Hell refers to the scenario when developers have been working on separate parts of a project and then want to integrate these parts [45]. Each developer knows that their code works in isolation, which might be true, but as a whole, the parts do not fit. Continuous integration forces us to find these problems as soon as they arise, and handle them accordingly.

---

3 Node Packet Manager, a dependency management tool for JavaScript
A developer checks into mainline, and this in turn should trigger a build of their artifact. This artifact should then be tested and make sure it integrates correctly with already existing artifacts. If a developer breaks backwards compatibility with another artifact, their changes are rolled back, and they will be notified about what went wrong so that they can fix the problem. This encourages a behavior to check in your changes frequently and continuously and catch bugs early.

These problems are usually identified when a project initiates its Big Bang Testing phase. As the name reveals, this is when a team of developers perform all of their testing in one sizable chunk. Testing is not something that should be done once, but continuously.

The benefits of adopting continuous integration are many [46]:

- Avoid integration problems
- Promotes well written code
- Improved software quality
- Software is always in a deliverable state
- Easily locate bugs to specific builds and versions
- Easily redeploy earlier versions if a bug is introduced

To adopt this approach; a version control system is a prerequisite and a CI server will make life easier for all people involved. There are many options out there, Jenkins, TeamCity, Circle CI and Travis CI to name a few.

Even if the development process is on a shoestring budget and needs an enterprise edition CI server, James Shore has introduced a way to get started on next to nothing. James calls this process “Continuous Integration on a Dollar a Day” [47], and it explains the practice of CI in a nice and concise way if it seems a bit much.

The gist of this approach is to get and old development computer that is able to run a build, a rubber chicken and a desk bell. Apart from buying these, James puts emphasis on build automation and group agreement. The rubber chicken rests on top of the build computer, and the bell somewhere everyone in the team can hear it.

When all the prerequisites are in place, the workflow is as follows:

1. Before checking out the latest code, check if the chicken is on top of the build computer.
2. If it is not there, wait for the person that has it to return it.
   a. In the meantime, run your test suite locally and make sure it passes.
3. When the chicken is back, get the latest code. You should now be in possession if the chicken. When the chicken is on top of the build computer, no one is actively working on the code.
   a. Build the code you just checked out, make sure it runs. If it does not, put the chicken back and notify the person who last checked in to fix the problem.
4. When the latest code runs, and your test suite passes, it is time to check in your code. Your code is now integrated with everyone else’s, and you can be sure it works.
5. Walk over to your designated build computer, fetch the latest code (i.e., your own code), run the build scripts and make sure it passes. If it does not pass, remove your contribution. This probably means that an environment variable or something else from the development computer is different from the build computer. When you’ve reverted your code, give the chicken to someone else if they need it.
6. When the build computer successfully builds the software, ring the bell. You now have a complete, functioning codebase.
7. Put the chicken back.
Today, there is no reason to use this approach since there is many alternatives to CI servers that are completely free. However, it clearly illustrates the benefits of CI.

Based on James archaic approach, it is easier to grasp the concept of CI. There are many manual steps in the process, but many of these are unnecessary with some further automation. The rubber chicken and the bell become obsolete with the use of a CI server.

The CI server is responsible for triggering the build when code is checked in, and notifying a developer if the build breaks. If the build does not break, a release can be built, provided the last step in the deployment pipeline has been reached. Building a full release can be very resource consuming, and depending on the size of the system being developed, might be done once a day or several times a day.

Notice that none of this would be possible if weren’t for the previously discussed version control and configuration management.

Secondly, all tests should be runnable via a command line. This is not necessary in the manual approach described above since someone initiates the process manually, but to save the valuable time of a developer, a machine must be able to initiate the build. Most tooling and programming languages today support this, but if the build process is dependent on a graphical interface, automating will be a problem.

3.2.4.1 Fowler’s practices for successful CI

According to Martin Fowler, there are a couple of practices that make up effective CI [48].

1. Maintain a Single Source Repository

Every file that is needed to run any given project should be maintained in a single repository. This means that not only the source code should be checked in, but also build scripts, test scripts, property files, database schemas and third-party libraries. The goal is reached when it is possible to take the code from source control and run it on any computer.

Every developer on the project should be aware of where to get the source code, and where to commit it when they have contributed.

2. Automate the build

Getting all code from source control and create a running program can sometimes be a complicated process that is prone to error. But as many other aspects in programming, this too can be automated since it is simply a repetition of steps each time.

The build script should compile all code, fetch database schemas, configure its environment and run the program. When the build script can do this on any (newly installed) computer, the build can be considered automated.

3. Make your build self-testing

When a project can be built automatically, it is time to make sure it performs according to specifications. Just because a program runs, does not mean it does what it is intended to do. Modern statically typed languages remove some easy-to-detect errors, but fare nastier bugs can crop up if code isn’t well tested.

A popular way of creating self-testing code is by following TDD, but there is no requirement to write tests before the code in CI [48].

A test suite with various tests should be included with the code, and it should be able to run with just a simple command. If the suite detects any errors, anyone concerned should get notified.

4. Everyone commits to mainline every day

To successfully integrate, developers should commit daily. By doing so, everyone is always up to date with the latest changes and the chance for merge conflicts aren’t as big.
Before committing to mainline, a developer must update their working copy from the VCS, resolve any conflicts that arise, make sure it builds and, finally, make sure the test suite passes. When these prerequisites are met, the developer can commit to mainline. By following this cycle, no broken code is submitted to the VCS.

The more frequent the commits, the less space for errors to hide, and the quicker they can be fixed if they arise. Frequent commits also encourage developers to break down their work into smaller chunks, and it provides a sense of progress [48].

5. Every commit should build the mainline on an integration machine
With frequent commits, there’s a greater chance that the mainline of the project stays in good health. However, things can still go wrong, be it forgetfulness by not updating the working copy before committing, or differences between developers’ environments.

To combat these errors, every commit should be built on a dedicated integration machine. A commit is not considered done until the build on this machine has passed. If the build breaks, there might be a missing configuration the development environment, or a developer has committed broken code. If it breaks, the commit should be reverted and the developer responsible should be notified, preferably automatically. The triggering of this build can be handled either manually or automatically.

In the manual approach, it would be equivalent to the analogy discussed earlier in this chapter. In the automatic approach, a CI server is responsible for fetching the latest code when a change is detected.

6. Fix broken builds immediately
The main benefit of CI is the knowledge that the code being worked on is always in a stable state. If a developer breaks a build, it is important to fix the problem immediately, especially if no automated rollback of that commit is in place.

If a build breaks, fixing it should be the highest priority. No more development should take place until the build passes on the dedicated integration machine.

7. Keep the build fast
Rapid feedback is an important aspect of CI. If the feedback takes too long, a broken build might stay on the mainline for extended periods of time, resulting in many developers fetching broken code.

It is possible to get fast feedback by splitting the testing into separate phases, and running the faster tests first.

8. Test in a clone of the production environment
The point of testing is to find problems, but in safe environment that won’t affect end users or customers. A significant factor is the environment in which the tests are executed. Every slight difference in the environment might introduce unexpected behavior.

Because of this, it is important that the testing environment is as close to the production environment as possible.

9. Make it easy for anyone to get the latest executable
Making the executable easily available facilitates demonstrations and exploratory testing. If the end user can see the finished software, it is often easier to get feedback on what needs to change.

10. Everyone can see what’s happening
Because communication is key in CI, it is important to ensure that everyone involved in the project can easily see the state of the system and the changes. The health of the build should be easy to see and understand, not just for developers but for all stakeholders.
11. Automate deployment
Since it is useful to have various stages of testing, it’s also useful to have different environments for each of these stages. Moving the executable between these environments should also be automated and easily performed by a simple command.

This brings with it the benefit that deploying to production should be equally straightforward and easy.

4 Findings and analysis
This section of the thesis summarizes what has been learned through the practical application phase and attempts to answer the research question stated in the introduction.

The chapter begins with an overview of the environment at SBA and a presentation of the systems that were chosen as test systems, and why they were chosen.

Every key aspect of continuous delivery will be examined in its own chapter, together with any findings from interviews, literature and observations.

Each part of the deployment pipeline implementation will be presented together with the reasoning behind the choice and examination of where SBA is at this stage, and where they are heading.

4.1 The Swedish Board of Agriculture
Understanding the conditions under which this thesis has been performed is a must to understand the final result. The Swedish Board of Agriculture has many important aspects already in place, or in the works. Here, the author presents an overview of the infrastructure and environments at SBA.

A more thorough examination of each part of continuous delivery is presented in the following chapters.

4.1.1 Server environments
SBA is using virtualized hosts to run servers and host applications. This has the benefit of easily scaling up and accommodating requirements for more resources quickly. However, it also requires a new operating system to be installed when a new server is provisioned, in contrast to containerized environments.

Some applications are running on several nodes in clusters, and the traffic to these nodes are then load balanced and have failovers to provide nearly 100% uptime.

Currently there are three main development environments. In order of least production like to most production these are:
- Development
- Test
- Volume

The development environment is a sandbox environment for developers. Here developers are free to test different configurations, deploy at any time and reconfigure the servers. These servers have the least resources available to them, but they are a great resource to test that the locally built executable works in another environment.

Next comes the test environment. This environment has more resources available than development, but developers aren’t as free to fiddle with it, since others rely in these servers to operate.

Lastly, there is the volume environment. This environment has clustered and load balanced servers, and the most resources of the three, making it the most production like environment available.

The final destination for any deployable is the production environment.
To deploy an executable or change any configuration here, developers must first write deployment specifications according to a Word template and send them to the operations team. The operations team then reads the specification and performs the deployment of the requested change. Any change performed must be linked to an issue in JIRA, SBAs issue management tool.

4.1.2 Current production deployment procedures

Deploying to production at SBA is done a few dozen times each year, as discussed in the Introduction chapter. Each system that has a release due must be accompanied by a deployment specification. The deployment specification contains information about:

- The deployable
  - Version
  - GroupId (Specific to Maven)
  - ArtifactId (Specific to Maven)
- What server(s) to deploy to
- Environment configuration
- Possible dependencies to other, simultaneous deployments

Maven\(^4\) has a specific convention for uniquely identifying software and its versions. This identifier consists of a groupId, artifactId and version, henceforth referred to as GAV.

The groupId is used to distinguish artifacts with the same artifactId from each other, and the version is used to separate executable versions.

The specification also contains information about which servers to deploy to. Some deployments can only be deployed to one server in the cluster, especially if they have some database write privileges. If a deployment that writes to a database on a schedule is deployed to two or more servers, that might cause errors like race conditions and events triggering twice in different systems, overwriting date or just running twice unnecessarily. This is a problem that will make it difficult for SBA to deploy software automatically, since human intervention will always be needed to determine which server(s) the system should be deployed to.

Deploying to several nodes in a cluster also means more manual steps for the operations team, meaning each node has to be identically configured or the deployment might fail. This is another problem with current procedure and has led to almost every server/environment being snowflake environments. There is no straightforward way to issue another server or scale horizontally. Instead, SBA has relied on vertical scaling, meaning the increase of resources (CPU, disk space, memory etc.) on a single node.

If the deployment needs new or updated environment configuration, this also has to be added to the deployment specification. This has caused errors, because pasting a command line to Word might change encoding or alter some characters. It won’t be immediately visible, but it will render the command unusable, or in a worst-case scenario, malignant.

Finally, some systems have dependencies that other systems are deployed before they can function or start correctly. They have to be started in a specific order, or they might not start at all.

4.1.3 Systems under test

The author has chosen two systems with distinct features and needs to test each principle on. Two systems were chosen to show that the research question is attainable. The first system described is also easier to grasp due to its relatively small size. This effectively makes it a stepping board to a more complex implementation in the latter system.

Here, a brief overview of each system and the technology it is based on is presented.

\(^4\) Maven is a build management tool
Findings and analysis

4.1.3.1 Style Guide
SBA is currently developing a style guide to easily maintain a unified front of external systems and services.

The style guide contains an automatically built documentation site, custom HTML markup that generates components or elements when a browser page is loaded, and any associated HTML these components might need. An example of how the system operates can be seen in Appendix 2: Example of how the style guide operates.

The reasoning for choosing this project is threefold:

- It is unit and component tested.
- The system is written entirely in TypeScript, a superset of JavaScript. Other systems at SBA are mostly Java or C#, making this system the odd one out and unexplored territory for SBA. It also differs from the second system under test chosen, which is written in Java.
- It behaves differently depending on the environment, meaning configuration management will be applicable

The project is currently only in its crib and consequently a perfect candidate for elaborating the concepts discussed in this thesis. It has not been deployed in production as of writing this.

4.1.3.2 Jorden
Jorden, meaning Earth, is the biggest project at SBA. The repository in Git rests at 1.4GB and at the time of writing it consists of 38 modules. It is written in Java with a user interface written in AngularJS. This thesis will limit itself to the back-end part of the application, since front end is already covered by the style guide.

Jorden is a management system. Its purpose is to handle applications from farmers and then split these into distinct application types, e.g. subsidies for animals, equipment or environmental measures. The system then applies a lot of rules to these applications, all of which affect the outcome of the final payment. All rules are based on laws in the EU or Sweden. To apply all these rules, Jorden has dependencies to five other systems, each with their own responsibility either directly related to agriculture or payments related to agriculture. When a decision has been made, a controller determines if the application is correct, and then another system at SBA handles the final payment.

When one of these rules must be rewritten, many farmers might need to have their rules recalculated and this can be error prone and require a lot of testing and evaluation.

The reasoning for choosing this system is threefold:

- It is the system that would benefit the most from a deployment pipeline
  - The development team is under pressure and always on tight deadlines
- It is the biggest system, meaning it has the most requirements to fulfill
- It has many contributors each day, making integration a must

4.1.4 Continuous Integration
SBA is currently using Jenkins as a CI server, and have been for some time. Developers can create their own items that can listen for changes on branches and then trigger builds.

For some systems, this is used to deploy automatically to certain environments, but not to production. However, this automatic deploy will fail if a system introduces new properties or environment variables that it needs to run properly. An even bigger issue is that environment variables aren’t version controlled, so many servers are so called snowflake servers.
All properties\(^5\) are version controlled, but they are not deployed as part of the executable and need to be added manually in a different directory than the actual executable. Currently, there is no scripted way to do so and this introduces a few problems.

The first of these problems being, many properties are environment dependent. For instance, even if the properties file would be fetched from the VCS at the time of deployment, the properties in it might not correlate the environment it is being deployed to. This could easily be solved by maintaining a properties file in VCS that is unique for each environment.

The final blocker is systems that cannot be deployed to more than one server because they would cause race conditions or overwrite each other’s data. This makes it impossible to scale systems or services horizontally and they need human intervention to know where they can be deployed.

### 4.2 Choice of version control system

The chapter is structured by observations and statistical analysis first, then literature findings and finally the situation at SBA.

The target condition for this implementation phase is to determine if:

1. SBA has the right tools in place already
2. There is a right or wrong choice when no VCS is in place yet

#### 4.2.1 Observations and statistics

Choosing a version control system that all developers are comfortable with is tricky. Many developers are comfortable with Git thanks to the widespread use of GitHub. Google Trends provide a clear answer to the question of which technology is more popular. In Figure 10, the search trends for the terms *Git* and *SVN* are displayed.

![Google search trends from September 2016 to August 2017](image)

Figure 10: Google search trends from September 2016 to August 2017 [50]

The data (available in Appendix 7: Trends for SVN and Git) is formatted as follows:

Numbers represent search interest relative to the highest point on the chart for the given region and time. A value of 100 is the peak popularity for the term. A value of 50 means that the term is half as popular. Likewise a score of 0 means the term was less than 1% as popular as the peak [50].

---

\(^5\) Properties control how a system behaves
When comparing number of users and projects on the world’s top three version control repositories, the results also speak volumes. Take into consideration when viewing Table 3 that GitHub only provides Git as a version control system, whereas their main competitors have more alternatives.

Table 3 Popular repository hosting sites

<table>
<thead>
<tr>
<th>#</th>
<th>Name</th>
<th>Users</th>
<th>Projects</th>
<th>VCS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>GitHub</td>
<td>~32.4m [51]</td>
<td>~105.4m [52]</td>
<td>Git</td>
</tr>
<tr>
<td>2</td>
<td>Bitbucket</td>
<td>~6m [54]</td>
<td>N/A</td>
<td>Git, Hg⁶</td>
</tr>
<tr>
<td>3</td>
<td>SourceForge</td>
<td>~3.8m [55]</td>
<td>~430,000 [55]</td>
<td>Git, Hg, SVN, Fossil⁷</td>
</tr>
</tbody>
</table>

4.2.2 Literature findings

A thorough literature review has found no way of saying that there is a clear-cut choice of what version control system is the best, it all boils down to what needs your organization has.

Each VCS has its own benefits and drawbacks and will suit different organizations with unique needs. The two are even so diverse that one organization might need them both, or decide to not use an open source alternative and opt to pay for Perforce instead.

If your team and/or organization consists of five developers or less, Perforce is free [56]. Be aware however, that if the organization grows beyond that it could become an expensive choice. The most major differences are outlined in Table 4.

Table 4 VCS feature comparison

<table>
<thead>
<tr>
<th>Tool / Factor</th>
<th>Git</th>
<th>SVN</th>
<th>Perforce</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centralized</td>
<td>Red</td>
<td>Red</td>
<td>Green</td>
</tr>
<tr>
<td>Distributed</td>
<td>Green</td>
<td>Red</td>
<td>Green</td>
</tr>
<tr>
<td>Forking</td>
<td>Red</td>
<td>Red</td>
<td>Grey</td>
</tr>
<tr>
<td>Gigantic repositories</td>
<td>Red</td>
<td>Yellow</td>
<td>Red</td>
</tr>
<tr>
<td>Local branching</td>
<td>Red</td>
<td>Red</td>
<td>Green</td>
</tr>
<tr>
<td>Locking</td>
<td>Green</td>
<td>Red</td>
<td>Green</td>
</tr>
<tr>
<td>Path-access authorization</td>
<td>Red</td>
<td>Yellow</td>
<td>Red</td>
</tr>
<tr>
<td>Rebasinf</td>
<td>Green</td>
<td>Red</td>
<td>Green</td>
</tr>
<tr>
<td>Speed</td>
<td>Free</td>
<td>Free</td>
<td>$740 per seat</td>
</tr>
</tbody>
</table>

Green means that the technology is suitable, yellow means partially suitable or up to a limit, and red means not suitable. Grey means no information was found.

There are many more factors than the ones outlined in the table above, but they are the most major points. Deciding what version control system should be used should not be rushed.

If your organization has no version control system, you might be interested in trying out a cloud hosted option like GitHub, GitLab or SourceForge before deciding, if your organizations security allows for code being hosted off premises. To host private repositories, most sites require subscription.

---

⁶ Mercurial (Hg), a versioning system written in python.
⁷ A distributed version control system
If off premises hosting is not an option, GitLab has a free community edition that can be used and easily set up as a Docker container. GitHub is also possible to host on premises, but it requires an enterprise account and is not free.

Getting up and running with version control should be trivial for any organization, even if no version control has been in place since earlier. There is a plethora of alternatives, both free and proprietary, and something that will suit the needs for everyone. There is never a reason to rely solely on a shared folder on a hard drive.

### 4.2.3 Version control at SBA

SVN has been used successfully at SBA since 2011, and before that was CVS. SBA strives to be at the front of technology, but adopting too early in such a large organization might cause a lot of problems down the line if these decisions aren’t researched properly.

Git has had a lot of time to mature since its release in 2005, and SBA has now begun migrating repositories from SVN to Git.

The reasoning behind choosing Git were many, and among them the previously discussed speed and its distributed nature (for full decision, see Appendix 1).

There is also an ongoing venture at SBA to enable developers to work remotely and being able to access more tools. Git’s distributed nature makes it a perfect candidate to remote working, since you don’t always need a network connection.

Software that helps the developer often runs multiple commands and does much behind the scenes and obscures what it does. Because of this, developers are encouraged to use Git from the command prompt to get a better understanding of how it works.

SBA has some need for path-based authorization, but the systems that do need these capabilities can instead be hosted in SVN where that option exists. However, when this need has arisen, SBA has chosen not to use path-based authorization. Instead, these permissions are handled via an internal system that grants a user access to the specific server where the code is hosted.

There is rarely any need to host binary files, and if there is a need SharePoint is mostly used for that purpose. Repository sizes aren’t that problematic (yet) either, the biggest repository as of writing rests at ~1.4 GB. Compared to the Linux Kernel at GitHub that is 2 GB, that is big. However, if taking all forks of the kernel into account, we are reaching a couple of terabytes in size.

Overall, the right path for SBA to be on right now is to keep SVN and Git side by side, and simply migrate the projects that are suitable to migrate one by one. This way the work can be done incrementally, and the organization gets the best of both SVN and Git, since both are already in place. If money was no issue, Perforce would have served their needs better, but at about 100 developers and lots of consultants, it simply isn’t an option.

Unfortunately, SBA currently has three co-existing VCSs which will likely to a problem in the future. Some code related to older systems is still in CVS, and the code is sometimes 10 years or older. This is problematic since there is no easy way to integrate into a modern workflow.

### 4.3 Tackling configuration management

This chapter discusses the findings of the configuration management phase of the practical application. Since SBA has next to no configuration management in place, a solid strategy for how to handle this is needed.

The author has interviewed some key people from the operations team to get a better understanding of their work. The result from these interviews is presented first. These interviews laid the ground work for the action plan, i.e. how to implement configuration management.
Findings and analysis

The author has chosen to only focus on the configurations that concern the systems under test, not those that belong to operating systems or server as this is not something that can be applied in other organizations. It will depend on if a hypervisor is being used and which hypervisor, if the servers are clustered, if the application is being run in a container and so on. The operations team at SBA handle these, and it is out of the scope for this thesis.

Next, the literature findings are presented, and finally the outcome of the implementation phase at SBA.

The target condition for this implementation phase is to determine if:

1. SBAs current infrastructure is sufficient
2. It is possible to manage the configuration of the chosen systems under test

4.3.1 Interview findings

According to the interview findings with the operations team at SBA, configuration management is lacking in many areas.

For server provisioning, some of the work getting the virtual host up and running is manual. A new host is cloned from a VMware® template, and then it is assigned an IP address manually. Work is being done to automate the provisioning of virtual hosts, and different technologies are currently being evaluated.

When the virtual host is up, another layer of virtualization is added in the form of Docker. New servers being provisioned use a Docker base image to avoid any configuration issues or snowflake servers. This way, it is possible to pinpoint specific configurations, versions and settings for specific needs on top of the hypervisor.

All Docker images are hosted in an internal repository and maintained by technical test leaders. Folders from the virtual host are then mapped to the Docker image, effectively connecting the two virtualized layers.

The work of automating hosts and application environments is ongoing, but some parts of the infrastructure are lacking. For instance, it is not possible to automatically provision databases and messaging services. For full adherence to current configuration management standards, the provisioning and configuring of these should preferably be automated.

There is partial management of database properties. For instance, there is a standard version used when installing databases and then some user configuration. However, these databases and their schemas have changed over time, effectively making the current configurations obsolete. There is no way to recreate an environment’s database as is.

Messaging services is another key piece of the infrastructure that is lacking in this aspect. The configuration of these servers has evolved over time, and provisioning a new server or configuring a development environment is done by hand.

The last important piece of the puzzle is OS configuration and other software used. Since SBA uses Java EE, controlling what versions are run on each server would be desirable.

The team is not interested in giving up all control to automated scripting due to, in part, security concerns.

Properties is another key piece that are being worked towards solving. Properties are often written in the deployment specifications that are in Microsoft Word format. This can cause encoding problems, and some characters are changed altogether when pasting them to a new source. For instance, the back tick (‘) is not friendly to copy paste and has in more than one instance caused errors.

This problem would be solved by having all properties in version control. As of now, only properties needed to setup a service or system to communicate against the development environment are checked in.

---

8 A virtualization software
4.3.2 Literature findings

Information about technology agnostic configuration management in literature is very sparse. Most literature found during the initial literature study only covered specific tools coupled with specific programming languages and/or environments.

If an organization has a VCS, chances are all requirements to begin with configuration management are already met. However, having a solid strategy for where to place the configurations and how to fetch and update them is the most difficult part of getting started.

4.3.3 Result of implementation

If all properties are checked in to Git, retrieving them would simply be a matter of knowing which service or system is being deployed, standardize the folder structure and standardize the way the properties are handled. This should be up to each system or service, since the needs are rarely the same from instance to instance.

4.3.3.1 Style Guide

The style guide has a few distinct aspects that must be considered when managing its configurations. For instance, it has image and font dependencies that are not dependent on specific environments, and properties that are environment dependent.

4.3.3.1.1 Handling environment dependent properties

When working on the style guide, the author opted to version control a plain text file with a key-value format for each environment. This format is a very common way of configuring Java applications, and it uses the .properties extension. It can be handled intuitively from Java code, and it’s easy to handle with bash scripts as well.

Even though this system is not written in Java but in Typescript, the author deemed it preferable to have a coherent management and structure of the files containing properties. The resulting file is as basic as follows:

```plaintext
CDN_BASE_URI=vl-cdnutv01
CDN_USER=[REDACTED]
CDN_BASE_FOLDER=/opt/nginx/sjv/designsystem
CDN_ASSET_FOLDER=/assets
```

There are four files, each with a name corresponding to the environment to which it will be deployed. The files were placed in repository at /environments/development.properties, and so on for each unique environment.

Using a plain text file minimizes the risk of encoding errors and other hard-to-diagnose errors. Beware of making the system too configurable, as that too will make it very brittle to changes.

Only one server is set as base, but there is one additional server that is load balanced. This server is mirrored from the first node, so it does not have to be taken into consideration.

4.3.3.1.2 Handling static content per environment

The style guide has more dependencies than properties, and these too must be managed. For instance, it is desirable that SBAs logotype, chosen fonts and icon packs are available in the environment that is being published to.

Previously, these files were just dragged and dropped to an internal CDN\(^9\) and weren’t managed anywhere. The author has chosen to version control these dependencies. For instance, some icon material is handled inline in files since these will rarely change, e.g. they are bundled with the deployable of the system. Others, like SBAs logotype, might change and

---

\(^9\) Content Delivery Network
then it would be desirable to have every system implementing the style guide reflect this
without updating their version.

To facilitate this, content that is subject to change is pushed to the CDN during the deploy job
on the CI server. This makes it possible for UX designers to redeploy a logotype for every
system that has implemented the style guide without making them upgrade to newer versions.
We could also, in theory, change the font and icon set of every system with a single source of
truth.

The style guide now has all its configurations version controlled in its own repository in Git. It
is possible to backtrack errors through tags, and see who changed any configurations.

4.3.3.2 Jorden
Jorden has a many more dependencies than our previous system. It is dependent on
properties to change its behavior, database connections, message queues and connections to
other systems.

It is helpful to immediately classify these dependencies into two types; environmental
dependent and static, as we also did for the design system although not explicitly mentioned.

An example of a dependency that is dependent of environment would be, for instance, a
database connection. We would not want to
read from the test database when we deploy to the
development environment.

A static variable would be something that is consistent between environments, for instance a
cron job that should initiate at a certain time or the name of a message queue to listen or
send to.

To make configuration easier, let’s split these up in different files. Static content belongs in
one file that should rarely be changed, and environment dependent variables in one file for
each environment.

In this case, connections to databases and message queues are handled by our application
server and has to be added by scripts when the server is running. To facilitate this, we version
control our scripts too (see Appendix 6: Scripting queues and databases). As before,
the .properties extension and syntax is used. The code base for this system is Java, so here it is
a natural choice. Due to security reasons, the configuration of the system cannot be shown
here. However, it follows the same key-value as the previous example.

The final structure for handling configuration looks like this:

```
Jorden
|-- static.properties
|-- scripts
  |-- add_queue.sh
  `-- add_database.sh
|-- env
  |-- dev.properties
  |-- test.properties
  |-- volume.properties
  `-- prod.properties
```

Everything is in place to utilize the configurations in the upcoming pipeline.

---

\(^{20}\) A time-based job scheduler
4.4 Implementing Continuous Integration

This chapter discusses the findings of implementing continuous integration into the systems. Since SBA already has a CI server in place, this part of the thesis will mainly focus on how to utilize this server effectively to shift the workload from the developers to the CI server.

Some unstructured interviews have been conducted with developers, testers and operations teams to get a better understanding of their work. The result from these interviews is presented first. These interviews laid the ground work for the action plan, i.e. how to implement continuous integration.

Next, the literature findings are presented, and finally the outcome of the implementation phase at SBA.

The target condition for this implementation phase is to determine if:
1. SBAs current infrastructure is sufficient
2. It is feasible to continuously integrate the chosen systems under test

4.4.1 Interview findings

The style guide is developed by a small group of developers that have a lot of stakeholders. The components developed by this team is going to be used by every system and external service that SBA provides.

New requirements are constantly cropping up from many sides, and the team developing the product needs to be able to focus on bringing value to the organization at a steady pace. Most of the team members are UX designers and their knowledge is better suited to developing easy-to-use and reusable components.

The UX designers wish to have:
1. An automated test suite
2. An automated deploy job
3. A way for stakeholders to verify in a safe environment
4. Release notes for each release

Point 3 and 4 are somewhat out of scope to this thesis, so they won’t be covered in-depth.

Jorden, on the other hand, has steeper requirements. Because of the size of the system and number of contributors, they need a pipeline that will fail fast if anyone broke the build. If the pipeline does not fail fast in case of errors, we are wasting precious resources on our CI server. The system already has a solid suite of automated unit tests, but lacks in other areas, like code analysis and acceptance testing.

After interviewing members of the QA team, a few developers and a tester involved in the system, a pipeline was agreed upon. The following process should exit immediately if a step fails. It should be initiated from a merge request. The merge request will not be committed to the target branch if any of the steps fail. Table 5 illustrates the proposal in an orderly way.

Table 5 Pipeline stages for Jorden

<table>
<thead>
<tr>
<th>#</th>
<th>Action</th>
<th>On pass</th>
<th>On failure</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Commit</td>
<td>Trigger pipeline</td>
<td>No fallback, changes still only on developer’s machine</td>
</tr>
<tr>
<td>2</td>
<td>Merge</td>
<td>Trigger compilation</td>
<td>Notify contributor(s), revert merge</td>
</tr>
<tr>
<td>3</td>
<td>Compile</td>
<td>Trigger unit tests</td>
<td>Notify contributor(s), revert merge</td>
</tr>
</tbody>
</table>
As a flow chart diagram, this pipeline can be visualized as seen in Figure 11. For brevity, the feedback has been left out of most stages, but it is implied in every stage, as can be seen in the table above.

![Figure 11 Suggested pipeline design for Jorden](image)

All developers are expected to write unit tests and attempt to merge and then run their unit tests against their local copy before committing. However, safety nets and fallbacks are needed in case someone forgets.

When designing the initial pipeline, the testing quadrant discussed in Testing Strategy has been adhered to. Each stage’s associated testing quadrant can be seen in Table 6.

<table>
<thead>
<tr>
<th>Testing Quadrant</th>
<th>Pipeline Phase</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1, unit &amp; component tests</td>
<td>Merge, Compile, Unit tests</td>
<td>Quickly discover errors</td>
</tr>
<tr>
<td>Q2, (automated) functional tests</td>
<td>(Automated) regression tests</td>
<td>Ensure customer needs are met. Ensure no regressions were introduced.</td>
</tr>
</tbody>
</table>
Findings and analysis

<table>
<thead>
<tr>
<th>Q4, Performance &amp; Load</th>
<th>Static code analysis</th>
<th>Ensure code adheres to best practice standards</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q2, (automated) functional tests</td>
<td>Acceptance tests</td>
<td>Ensure customer needs are met.</td>
</tr>
</tbody>
</table>

As you may have noticed, no stage in the pipeline corresponds to quadrant 3, Exploratory Testing. The reason behind this is that the team did not want automatic deployment to environments, but instead want to manually initiate each release. This would correspond to a built release being in a staging area. They wanted full control of where and when the release got deployed into any given environment, and hence, that responsibility is outside the responsibilities of the pipeline. Without this control, they were at risk of accidentally interrupting crucial load testing or manual testing being performed.

4.4.2 Literature findings

Even though they were written as far back as 2006, Martin Fowler's principles regarding CI still hold strong. If a system manages to get all these in place, it has come a long way. They serve as a solid baseline for what to aim for, and they have guided this implementation and the talks with teams.

The main take away from all literature, is that of having an automated test suite that is automatically triggered on commits or merge requests, and checking in regularly to detect incompatible code.

Getting the team into a mindset and getting them to agree to this new workflow is important. In the beginning, they might feel like they are not contributing because initially, the time to develop is drastically increased because of the new mindset and rigorous testing. Talk to the team, explain to them how the process works, and show them the benefits.

The key part to get any of this to work is to have every single piece of code, configuration or other dependency to be runnable via command line. Without this requirement, it would not be possible to run a headless CI server, and every step that does not support command line initiation would have to be triggered manually.

Finally, any artifact that is produced should be deployable to any environment, provided it gets the correct configuration files. We absolutely do not want our binary data to differ between environments.

4.4.3 Result of implementation

For continuous integration to work, it is dependent on that everything that is configurable is version controlled. If this is not the case, the build will be tightly coupled with a single environment, meaning that we could only utilize the pipeline for this specific environment. That would defeat the purpose of this approach entirely.

4.4.3.1 Style Guide

The deploy job is initiated manually and an environment is chosen. The reasoning behind the manual start is that there is no way to know beforehand which environment needs the new deployment, the need might arise from a feature request or bug report from another developer.

Most commonly, though, it is first deployed to the development environment to be evaluated manually by another developer or end user. Automating all testing for such a visually dependent system would be very fragile, hence the manual review.

The deployment job is initiated from SBAs CI server. A branch and a release version are specified, an environment chosen and then the job is triggered. Through scripting, the system is built and deployed to the chosen environment. The entirety of the script can be seen in Appendix 3: Deploy script for the style guide.
The main thing that is happening in this script that also utilizes the configuration management that was implemented before this are the following lines:

```
scp -r $WORKSPACE/dist/release
${map["CDN_USER"]}@${map["CDN_BASE_URI"]}:${map[CDN_BASE_FOLDER]}/$sjvVersion

scp -r $WORKSPACE/dist/site
${map["CDN_USER"]}@${map["CDN_BASE_URI"]}:${map[CDN_BASE_FOLDER]}/$sjvVersion/docs

scp -r $WORKSPACE/assets
${map["CDN_USER"]}@${map["CDN_BASE_URI"]}:${map[CDN_BASE_FOLDER]}${map["CDN_ASSET_FOLDER"]}
```

This takes the properties from the file and makes sure each part of the system is deployed at the right place.

Developers do not need know-how of the underlying infrastructure any longer. They can instead focus on bringing functionality to the system, and deploy when they need it.

The screen for issuing a deployment job can be seen in Figure 12.

![Figure 12 Screen capture from SBAs CI job for the Design System](image)

This pipeline is as basic as it can get, yet it removes many headaches and manual steps from the developer's workday. Optionally, we could even have job on the server poll or VCS at regular intervals and automatically deploy to one or more of our environments. As discussed earlier, this requires every team member to be in a “CI mindset”, where every single commit is treated as a potential release.

### 4.4.3.1 Tests

To gain more value from this deploy job and not deploy anything broken, we need an automated test suite.

The test suite in the style guide tests the components using a real DOM in Chrome running in headless mode. The author chose to run in headless mode since it is easier to automate testing against.

Testing is done through a framework called Karma. Through Karma it is possible to configure which browsers to test against (see Appendix 4: Karma testing configuration). The only limitation is that the browser needs to be installed on both the developer’s machine and the CI server.

If a test fails the developer responsible for the breakage is notified via mail, and a screenshot of the breakage is submitted so it can be viewed and recreated using the screenshot and the

---

1 Document Object Model, the HTML representation
failed test case as help. This is done using a framework called Puppeteer which facilitates headless testing (see Appendix 5: Puppeteer test, screenshots for usage).

4.4.3.2 Jorden
When a developer issues a merge request, a web hook is triggered from GitLab. The web hook is pointed to a specific job on the CI server. The job on the CI server, in turn, contains a pipeline script.

By default, the CI server did not possess capabilities that were sought after. Some skillful developers from Jorden took it upon themselves to develop the needed functionality. The author had no part in the development of these plugins. What resulted was two plugins for Jenkins, a generic web hook trigger and a violation comment to GitLab [57] [58]. These two solutions are tightly coupled to the tools that SBA has chosen, but they are available as open source for anyone to use.

The first plugin is responsible for reacting to the issued HTTP request from GitLab, and extracting any information needed from the request. The second plugin is responsible for publishing a comment in the merge request that initiated the job, should a stage fail for some reason.

The biggest hurdle to overcome when starting this was handling dependencies to all other systems. Jorden acts as a hub at SBA, and thus it depends on many other systems and many other systems depend on it. To get a test environment that we were able to recreate, we had to automate it. To facilitate this, one of the technical test leaders at SBA has set up several Docker containers to effectively emulate other systems. The author has not been involved in the development of these containers, all requirements have been put forth by the system owners and implemented by SBAs technical test leaders.

The containers are advanced and have a very high cohesion with the infrastructure at SBA, and as such they will not be presented here. The problem we aimed to solve with this approach, was to quickly set up an environment and emulate all other systems. When all tests are done, we are able to tear down this environment and stop consuming server resources. This enables us to continuously and safely set up the same test scenario repeatedly.

4.5 Designing a deployment pipeline
The following chapter aims to be an amalgamation of all knowledge acquired through the process of writing this thesis.

First, the knowledge gained from the literature study will be presented.

Lastly, the actual results of the implementation at SBA will be presented.

The target condition for this implementation phase is to determine if:

1. A pipeline solution would bring the desired value

4.5.1 Literature findings
The de facto book on the subject is Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation by Humble and Farley. It presents common pitfalls and best practices that the author has largely based further research on.

The author has referred mainly to the chapter about the deployment pipeline, since the other chapters aren’t very pragmatic when approaching this subject from a background where none of these aspects have been in place previously.
The authors offer examples from pitfalls they themselves have encountered and present examples for how to overcome these hurdles. They state that without the ability to run on any developer machine, developers might start to ignore errors from the acceptance test suite and try to bypass it, whereas if they had the ability to fix them, they would instead contribute to maintaining them. The author strongly agrees with this, but due to the current architecture of Jorden, this simply isn’t possible at the time of writing.

4.5.2 Result of implementation
The author has, together with several invested stakeholders, developed deployment pipelines for both the Style Guide and Jorden. Here, the author presents how these were put together, what underlying mechanics are in the works and how it is all pieced together.

4.5.2.1 Style Guide
The style guide is a simple project to write a deployment pipeline for. The pipeline must do the following:

1. Transpile\(^{12}\) to ES5
2. Test transpiled outcome
3. Configure URLs to CDN where static resources will be fetched
4. Build and publish documentation
5. Build and deploy a demo site for (future) Selenium tests
6. Minify content, deploy to CDN
7. Deploy an installable package to internal npm registry

Out of the six stages above, stage 3-6 are dependent on which environment it will be deployed to. As of now, there is no way to automatically target a specific environment, this must be done manually as shown in chapter 4.4 and Figure 12.

During the writing of this thesis, the script responsible for deployment and the surrounding configurations have been continuously improved and now resemble something that is almost production ready.

This has been done using only bash scripting, a static file server, a version control system and a free, open source, CI server. The confidence with each release has increased, and the UX designers can now release to any environment with just the push of a button. It is important to note, however, that the evaluation has been done solely by developers at SBA, and the style guide has not been deployed to production yet.

The flow is fully automatic except for in its initial stage, as can be seen in Figure 13. Some steps above have been sidestepped, both due to the nature of the tools used and what was learned along the way.

\(^{12}\)Transforming a programming language to another, similar, language
A scenario where the pipeline would be used is when a developer performs some hotfix or maybe an entire release. When the work is deemed done, the developer (or anyone, for that matter) can trigger the pipeline. This is done as previously shown, by selecting an environment and triggering it manually.

As soon as the pipeline is triggered, it fetches the entire project from Git and the given branch. When the project is fetched to the local workspace on the CI server, an install occurs. This is necessary due to the nature of the project. It uses npm and Typescript, and Typescript needs some dependencies to be able to transpile to the lowest common denominator of JavaScript, which is currently ES5. Without this, we would not be able to target older browsers like Internet Explorer 8.

Next, we tag and publish the project as a package to npm. For instance, if the build was given number 1.0, this would create a tag (snapshot in time) in Git, and give the same version to the release we are going to publish. During the time of writing, publishing the entire source code was highly sought after from participating frontend developers. The reason being that when the source code is available, they can easily fetch any variable used in the style guide and use it as they like, without having to resort to copy and pasting content. If they would have to copy and paste, the value might be out of date for the next release.

The author has set up a hook in npm that responds to the publish event. As soon as this event is triggered, the project will first build a production ready version of itself before publishing occurs. Finally, the entire project is packaged as a tar ball and sent to SBAs internal npm registry with the given release number.

Following this, the documentation is built. The documentation is both written by hand and automatically generated from source code. At the same time, release notes are generated and pushed to GitLab. The release notes are also written by hand, because there is a need for fine grained information in each release. This way we are able to reference solved issues and structure the notes in a better way using markdown, as seen in Figure 14.
Figure 14 Example of release notes published with a release

If all of this succeeds, we publish the documentation and the transpiled release to the given CDN. Wrapping up, this is what occurs in the pipeline:

1. Fetch from Git
2. Install dependencies
3. Tag release
4. Build production ready, transpiled output
5. Test transpiled output
6. Publish source code and transpiled output to npm
7. Publish release notes
8. Build and publish documentation to CDN
9. Publish transpiled output to CDN

All work done in this pipeline can be seen in Appendix 3: Deploy script for the style guide.

As you may remember, the initial specification also required a demo site. This has been implemented in the upstream of this project, and is triggered as soon as a build has passed. In this job, the author fetches the latest version of the style guide, and updates the demo site’s dependencies. When this is done, the site is built and published to CDN. Finally, the newly updated project is pushed back to Git. The script looks as follows:

```bash
# Setup

# Fetch from Git
git checkout master

# Get version number
version=`npm view @sjv/component-library version`

# Install dependencies
# npm install @sjv/component-library@latest --save

# Build production ready, transpiled output
npm run build -- --prod --base-href=./

git push --tags

# Build and publish to CDN
ssh ${map["CDN_USER"]}@${map["CDN_BASE_URI"]} rm -rf /opt/nginx/sjv/designsystem/example

scp -r ${WORKSPACE}/dist
${map["CDN_USER"]}@${map["CDN_BASE_URI"]}:/opt/nginx/sjv/designsystem/example

# Publish version
version=`npm view @sjv/component-library version`

git status

git add .

git status
```
Diving into this, it does the following:

1. Check out master
2. Get the latest version number of the style guide
3. Install and build site
4. Remove previous site (only the latest release needs a demo site, it would bloat quickly)
5. Publish newly built site to CDN
6. Push go Git. This release will have the exact same version as the style guide to facilitate debugging and finding potential errors.

The pipeline was built with the practices of Humble, Farley and Fowler in mind. The final result of how this measures up against the reviewed literature can be seen in Table 7.

<table>
<thead>
<tr>
<th>#</th>
<th>Requirement</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Single repository</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Automatic build</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Self-testing</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Daily commits</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Automatic pipeline trigger</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Quickly fix broken builds</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Fast build times</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Test in production clone</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Easily retrievable executable</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Transparent process</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Automated deployment</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>Build binary once</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>Deploy the same way to every environment</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>Smoke test</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>Deploy into copy of production</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>Stop when failure occurs</td>
<td></td>
</tr>
</tbody>
</table>

Some of these are not directly applicable to the pipeline but more so to the habits of the developers contributing. However, this is a key part for this to function and the author has chosen to incorporate it.

Unfortunately, requirement number 5 has not been met. The first is partly due to the infrastructure at SBA. There are four total environments, and there is really no way of knowing beforehand where a release is needed. With this project, however, it would have been possible to simply deploy to all environments except production upon every push to mainline. Versions are not forced on anyone, except for the statically served content (to preserve a unified outward appearance). To fully migrate, development steps has to be taken.

Requirements marked as orange are partly met. Requirement number 8 is hard to define, as in this case a production clone would simply mean a browser. We do run browser tests, but these do not target all available browsers. The author also deems requirement number 10 as partly met due to it not being “transparent” to a layman. To a developer however, it doesn’t
take much to understand. Lastly, number 10 is also partially met. This is due to the same reasons as requirement 8, but also because the style guide isn’t used in itself, other systems utilize it. By extension, this means that the style guide is deployed into a production copy if the system utilizing it is tested that way.

At the time of writing, the pipeline has been started 88 times and has an average build time of 70 seconds.

4.5.2.2 Jorden

Jorden was a massive undertaking. The author began with the simplest possible approach, sketching a skeleton of a pipeline based on interview findings and discussions with developers involved in the project. The result of this can be seen in Table 5 and Figure 11. The suggested pipeline design for Jorden. The result is based on knowledge gained from the literature review, and is based in best practices from Humble, Farley and Fowler.

With the table and pipeline diagram as a baseline, development of the pipeline began. Some pieces were in place from the beginning, some had to be developed. Thankfully, Jenkins has out of the box support building pipelines which made life easier for everyone involved. However, there was no existing way to trigger this naturally from a developer's workflow. A developer from Jorden, who has a solid background in CI on various platforms and tools [59], Jenkins included, took it upon himself to provide the missing functionality.

The goal was to not cause delays or awkward workflows for developers, and as stated in the table and diagram, it should have a hook to trigger on commit. This requirement, however, was a bit sidestepped to better facilitate Jorden’s code review policy. Instead, the plugin triggers when a developer issues a merge request via GitLab. We also want the fast feedback that is the main benefit of this approach. For this, he also developed a plugin that writes a comment back to GitLab. The plugin can be seen in action in Figure 15 and Figure 16.
Figure 15 Webhook and violation plugin in action

The implementation in Jenkins Pipeline DSL looks as follows. The full pipeline script can be found in Appendix 8: Jenkins Pipeline script for Jorden.

```plaintext
stage('Merge') {
    sh "git init"
    sh "git fetch --no-tags git@jbr:jorden.git +refs/heads/*:refs/remotes/origin/* --depth=200"
    sh "git checkout origin/${env.gitlabTargetBranch}"
    sh "git merge origin/${env.gitlabSourceBranch}"
    sh "git log --graph --abbrev-commit --max-count=10"
}

stage('Compile') {
    try {
        sh "${mvnHome}/bin/mvn -U -q package -DskipTests -Dmaven.test.failure.ignore=true -Dcheckstyle.skip=true -Dcheckstyle."}
```
Findings and analysis

Dfindbugs.skip=true -Dsurefire.skip=true -Dmaven.compile.fork=true -Dmaven.javadoc.skip=true
        sh "mvn jorden-ear/target/jorden-ear.ear ${friendlyNameEar}"
archiveArtifacts artifacts: '***/target/*.ear', fingerprint: true
    //commentMr(env.gitlabMergeRequestIid, "Ear-filen för
$gitlabSourceBranch kan nu laddas ner från ${BUILD_URL}")
    catch (e) {
        commentMr(env.gitlabMergeRequestIid, ":x: Kompilerar inte
            worried: $gitlabSourceBranch ${BUILD_URL}"
            throw e
        }
    }

If any of these two steps fails, this is what it looks like in GitLab:

![Screenshot of GitLab merge request](image)

Figure 16 A violation showing a merge conflict

When a merge request has been successfully merged and compiled, it is time to unit test it. This is done by simply issuing a command via Maven, `mvn test`. In a pipeline, it looks a bit more advanced:

```java
stage('Unit test') {
    try {
        sh "${mvnHome}/bin/mvn test
            -Dcheckstyle.skip=true -Dfindbugs.skip=true -Dsurefire.skip=false
            -Dmaven.compile.fork=false -Dmaven.javadoc.skip=true"
        commentMr(env.gitlabMergeRequestIid, ":heavy_check_mark: Tester ok
            :smiley: $gitlabSourceBranch ${BUILD_URL}"
        junit "**/target/surefire-reports/**/*.xml"
    } catch (e) {
        commentMr(env.gitlabMergeRequestIid, ":x: Tester **ej ok**
            :worried: i $gitlabSourceBranch ${BUILD_URL}"
        junit "**/target/surefire-reports/**/*.xml"
        throw e
    }
}
The additional flags sent to Maven here is just to speed up the process of running the unit tests. We are only interested in how the merged and compiled source manages to handle all unit tests. Notice that in the catch clause, we catch errors in each stage and have specific error messages for each that we supply to the violations plugin.

Now that we have the least resource intensive tests out of the way, it’s time to move on to the heavy guns; acceptance and regression tests. The author has had little involvement in this part of the pipeline, as it requires deep knowledge of the system, deep knowledge of all systems involved and some serious Docker knowledge.

The script for this part of the pipeline is heavy to run and difficult to maintain, so it is maintained in a different job in Jenkins. It is triggered from our main pipeline as follows:

```plaintext
stage('Regression test') {
    build job: 'jorden-regressiontest-pipeline', wait: false,
    parameters: [
        [ $class: 'StringParameterValue', name: 'gitlabSourceBranch', value: env.gitlabSourceBranch],
        [ $class: 'StringParameterValue', name: 'gitBranch', value: env.gitlabTargetBranch],
        [ $class: 'StringParameterValue', name: 'gitlabMergeRequestId', value: env.gitlabMergeRequestId]
    ]
}
```

Thanks to Jenkins pipeline support, all these stages are nicely visualized as can be seen in Figure 17.

The above code snippet will trigger another job on Jenkins, which does a lot of heavy lifting:

1. Start a test container
2. Build image of the system under test (Jorden)
3. Start infrastructure dependencies as containers
4. Run acceptance and regression test suite
5. Tear down temporary testing environment

The script is advanced, and in a nutshell, it runs commands against an external Docker host which in turn is responsible for fetching Dockerfiles from SBAs internal repository and executing them. The latest commit from Jorden is deployed here, and when the environment is set up, the tests start. The pipeline script in its entirety can be seen in Appendix 9: Jenkins Pipeline regression script for Jorden.

Again, this is visible and fully transparent thanks to Jenkins pipeline functionality, as shown in Figure 18.
Second to last, we perform static code analysis using Checkstyle\textsuperscript{13}. This step uses the Violations to GitLab plugin to output nicely formatted errors back in GitLab, as can be seen in Figure 19.

Lastly, we run acceptance tests using FitNesse. FitNesse allows developers to document functionality and the functionality acts as acceptance tests at the same time. These tests are plain text files written in a style similar to markdown. The files act as both documentation and test specification, thus making sure that the documentation is always correct for the given release. If a test fails, the documentation is outdated. This makes sure the documentation is always correct.

As with the Style Guide, this pipeline too has been developed with practices of Humble, Farley and Fowler in mind. The result can be seen in Table 8.

\textsuperscript{13} A tool that helps developers adhere to best coding practices
Two, unfortunately major, requirements are not met here. This is due to culture and inveterate habits at SBA. Production is treated differently than the other environments and it is not allowed to deploy anything automatically. Everything that is released into production has to be documented and saved in Word documents.

This is a true anti pattern in the realm of continuous delivery. It creates snow flake servers, spreads documentation out where it shouldn’t be, hinders configuration management and makes the process of deployment very time consuming, error prone and brittle. This is related to culture and habits and partly mistrust in the technology that enables continuous delivery.

Requirements marked as orange are partly met. Requirement number 1 is partly met but the repository that hosts Dockerfiles used in the regression and acceptance test stage is not bundled with the application code. Jorden also utilizes a caching solution and an event store, both developed in house and only utilized by Jorden, but not hosted with it. The reasoning behind this is that they both could potentially be used by other systems, but alas they are not, and according to the author they are tailor-made to meet the needs of Jorden.

Requirement number 7 is very subjective, but for a project as big as Jorden, this could be considered fast. Build times average 30 minutes, all stages included.

Requirement number 8 is partly met due to the testing environment being set up using Docker containers, and in production the application is deployed to a virtualized host. The Docker images are also manually maintained by technical test leaders, so there is no guarantee that these images are a true reflection of the production environment. Due to the differing technologies, environment variables are also sure to differ, although this is hard to verify. Requirement number 15 is highly related to this, and the same reasoning applies.

4.6 Defining a process

Thanks to having two systems to test on, it is possible to outline the common steps used in the development of both pipelines. Without this, it would not have been possible to find common denominators and outline the following guidelines.
Findings and analysis

An unspoken prerequisite for each of the following steps is that the application build process should be able to start via command line. Of course, to gain any advantage from this, a solid test suite is also a must. This must also have the option to be triggered via command line. Your test suite should optimally contain the following:

- Unit tests, 85% coverage
- Commit test suite, consisting of a small number of unit tests
- Smoke tests
- Acceptance and regression tests
- Exploratory testing, if needed

When creating a test suite, involve the necessary people (i.e. not just developers) and adhere to a testing strategy. Code coverage really isn’t that interesting, although it can be a useful metric. For instance, if you already have a well-tested code base with high coverage, it might be a good idea to enforce that this percentage does not drop below that.

To begin, version control is indeed a must. In today’s development practices, it would be very surprising to find an organization that does not have some version control system already in place, be it public, private or both.

However, just having version control is not enough unless the organization possess the knowledge to utilize it correctly. For instance, adhering to the single repository principle put forth by Fowler will make life easier for everyone involved. There is rarely any reason to sidestep this rule, unless the project is as huge as GNOME. Be aware, however, that this will give huge overhead and be hard to manage. Having everything in a single repository also significantly facilitates the introduction of configuration management.

Your organization’s definition of done for version control implementation may vary, but at the very least all source code should be placed there and have their own corresponding repositories. Whether you choose Git, CVS, SVN or Perforce is up to the needs of your organization. However, Git is a major player and chances are developers are more comfortable using it.

You should also decide on a branching strategy to facilitate working on features, releases and hotfixes. It is important that all developers are on board with the given branching strategy to keep the repository uniform. Naming conventions also make it easier for other developers to get on board and contribute faster.

A minimum definition of done for version control:

- Decide on which version control system is right for your organization
- Check everything into your version control system
  - Source code
  - Configurations
  - Everything needed to build/run the application. More on this later in this chapter.
- Decide on a branching strategy

As a quality of life improvement, it is also a good idea to decide on a code formatting tool. This will increase readability and ensure that all contributing developers adhere to the same coding standard. At the same time, choose a tool for static code analysis, like Checkstyle for Java or ESLint for JavaScript. These two simple things will greatly improve code quality and help developers write better code.

- If code formatting is not already implemented, decide on a suitable tool and let developers incrementally format code as they work on features or hotfixes
- If static code analysis isn’t already in place, choose a suitable tool. Let developers incrementally fix errors that the tool points out
If version control is in place, it is time to start sorting out what needs to be a managed configuration and what shouldn’t. Hopefully, your application already uses some kind of third party packet manager like npm, Maven or NuGet. Do not bundle third party libraries with your application, these should be managed by a packet manager. If you have bundled binaries or executables, find the corresponding packet in your corresponding manager and resolve it from there.

If your source code is tightly coupled to any one environment, that’s a red flag. This would immediately make it impossible to deploy the same way to every environment. Instead, use properties to fetch values dynamically from your source code. In Java, this is done via .properties files, in .NET via settings like appSettings or userSettings. At the very least, you should be able to deploy the same executable to any environment. When that is possible, configuration management for your application can be considered complete.

However, there is a lot more to configuration management than managing it for your application. Depending on how you provision environments, this should be a consistently repeatable task that should produce the same outcome. Scour your applications existing environment for settings that it depends on to run. A fairly simple way to do this, is to deploy your application to a freshly started server. If the application fails to start, look at the log files and determine what is missing. Fix the error, via command line and make sure this configuration is in the same repository as the project. This ensures that your application manages its own configuration and can be deployed anywhere.

The last piece of the puzzle is to have reproducible server environment and application stack. Automating server provisioning should be a fairly easy task, whereas the application stack might be harder. There is no guarantee that everything you need to configure is doable via a command line interface, but strive to do as close to 100% as possible. There are a lot of tools that can help with this, for instance Puppet, Ansible or Chef. Unfortunately, this is a whole other area and does not fly under the development flag and would make a whole other thesis in itself.

A skilled operations team should fix this issue, in cooperation with developers. Decently sized organizations should form a cross-functional DevOps team. The ultimate goal is to be able to have so called Phoenix Servers, but that might be a stretch.

Strive to meet the following when implementing configuration management:

- Identify third party dependencies your application relies on
  - Ensure these are handled by a packet manager
- Identify your application environment configuration
  - Ensure this is runnable via a CLI
- Identify your applications stack
  - When recreating the applications stack, install everything via CLI and use a packet manager
- Make server provisioning a consistently repeatable task
  - If needed, form a DevOps team

This does not have to be done all at once, but keep in mind this is a blocker before you can continue with any of the other steps. CI and CD is not possible to its full extent without well thought out configuration management.

Having version control and configuration management in place, you should feel comfortable and knowledgeable about your application, its dependencies, its stack and its environment. There should be no surprises. This knowledge is needed to develop a strong test suite and test the correct things in your pipeline. Before beginning with a pipeline, ensure that all developers are on board with the chosen branching strategy and that they are willing to adhere to the practices of CI.
A CI server software should be chosen with care. It should have pipeline support and it should be able to run pipelines with a smart schedule to allow for organizational growth and continually increasing contributions. Jot down the usual build process, make a sketch, and make a flow chart. Visualize what the pipeline should do and achieve before getting down technical details. This mindset should also persist when you do get technical. Use big screens to show the process of a build, where it fails, what takes time and so on. All stakeholders and investors should be able to know the status of the application quickly.

When designing your pipeline, do it with fail-fast principle in mind. As discussed earlier, this means that cheap and fast tests should run first. This will increase throughput and feedback speed and result in more value from the pipeline. Before starting expensive tests, like acceptance and regression tests, make sure to smoke test the application. Simply make sure it even starts, before wasting valuable resources on a broken environment or build.

- Ensure developers:
  - understand the chosen branching strategy
  - commit to mainline frequently
  - do not cause merge conflicts
  - write unit tests
  - fix problems quickly, this is top priority
- Design the pipeline analogously before getting technical
- Choose a CI server that is proven and battle hardened
- The pipeline should
  - fail fast and provide quick feedback
  - only build executable once
  - notify developers
  - be transparent
  - run its test on a (at least close to) production clone

Before starting to implement this, it might be a good idea to actually use the rubber chicken example in real life. This will prepare developers and get them in the mindset that is required for CI.

The author does not deem it necessary to actually deploy to an environment automatically, although this goes against the title of this thesis. Simply having a rigorous test suite and a pipeline that delivers an executable to a staging area or repository would be a huge step forward for many organizations.

If you take away one thing from this, it should be to get started with CI. This would also force you to manage all your configurations, but in the long run you will see a tremendous increase in throughput, system stability and developer happiness.

### 4.6.1 Implementing Continuous Delivery Step by Step

Continuous Delivery will be a huge step forward for any organization. It can be done incrementally, and it does not have to be expensive. To show the relative simplicity, the process can be presented in a checklist, as seen in Table 9.

**Table 9 Outlined steps towards Continuous Delivery**

<table>
<thead>
<tr>
<th>#</th>
<th>Step</th>
<th>Priority</th>
<th>Definition of Done</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><strong>Code Quality</strong></td>
<td></td>
<td>Every developer uses the same formatting tool.</td>
</tr>
<tr>
<td>1.1</td>
<td>Code formatting</td>
<td>1</td>
<td>This will reduce potential merge conflicts later on</td>
</tr>
</tbody>
</table>
Discussion and conclusions

This section will address the author’s discussion of the method and conclusions reached. First, the author will address the method and how it worked in practice. Second, findings from the implementation will be presented and lastly, any recommendations from the author for further research is presented.

1.2 Static code analysis 1 A tool has been chosen, implemented and is being used by every developer.

2 Test Suite 1 You would feel certain to deploy into production

2.1 Unit tests 1 85% test coverage

2.2 Commit tests 3 You can run a small, useful, test suite separated from the main unit tests

2.3 Smoke tests 3 You can ensure that the test guarantee basic functionality of the application

2.4 Regression tests 1 Your old, but still valid, acceptance tests guard from introducing unwanted behavior

2.5 Acceptance tests 1 You can fully satisfy the needs of the customer

3 Version Control 1 When nothing concerning an application, or its environment resides on a single machine

3.1 Branching strategy 1 Consensus has been reached about strategy, and everyone involved understands why it is used

3.2 Source code 1 Any developer can check out and build from a single source

4 Configuration Management 1 Everything needed to configure and deploy an application is checked in

4.1 Application configuration 1 Any developer can setup a testing environment locally

4.2 Application stack configuration 2 All application dependency (databases, message brokers) configuration is checked in

4.3 Server configuration 3 A server can be provisioned automatically, and the application and its stack can be deployed to it

5 Continuous Integration 1 Application is always in a production ready state

5.1 Developer agreement 1 Developers understand the need to write tests, check in frequently and solve errors quickly

5.2 Analogously sketch 2 Everyone involved can clearly visualize the life cycle of the application

5.3 CI Server 2 A standalone server which satisfies your organizations needs

5.3 Fail fast 1 The fastest tests you have run first

5.4 Quick feedback 1 Developers get instantly notified if they break a build

5.5 Production clone 2 The testing environment is (close to) a production clone

5.6 Visualize 3 Anyone can understand and follow the build process

This can and should be done incrementally and continuously improved as you learn more about these practices, your application, and its needs.

5 Discussion and conclusions

This section will address the author’s discussion of the method and conclusions reached. First, the author will address the method and how it worked in practice. Second, findings from the implementation will be presented and lastly, any recommendations from the author for further research is presented.

---

14 Should be decided from case to case
15 Depending on complexity, exploratory and/or manual testing may also be needed to satisfy
16 Tightly coupled with version control, although separate practices
17 Depending on how complex the application life cycle is, this may not be needed
5.1 Discussion of method

My goal for this thesis was to implement lasting change and get a fundamental understanding both of SBA, Continuous Delivery and the key pieces that enables it. To achieve this, it was natural to initially approach the problem as a case study to learn more about the current situation and what problems needed to be solved. This also enabled me to more easily find suitable literature.

During the case study phase, I observed and documented current practices as well as collected data through semi-structured interviews, and speaking to people as they went about their work. This approach was very beneficial in the long run, since it enabled a running start to the rest of the thesis and enabled me to focus on bringing a working solution to SBA, thanks to having a helicopter view of many areas.

With the knowledge from the initial data gathering and literature review, examining each building block for Continuous Delivery using the target condition was very eye opening and provided a lot of knowledge of the systems involved and about CD. I managed to get a lot of involvement from many different stakeholders; technical test leaders, system developers, UX designers, managers and even the head of the IT department.

To use the Improvement Kata in the implementation phase of the AR approach enabled me to involve many different roles, get many different perspectives and adapt the solutions to better fit the organization. Communication was easy and clear, thanks to having a target condition to work towards.

Using this amalgamation of approaches, I believe to have reached the requirements of validity and reliability, and developed a process that can be adapted to any organization, even on a shoestring budget. All theoretical findings have been put to use and enabled a fully working solution, with two next-to-complete pipelines. Some fine tuning is required to increase stability, but they are not far from production ready, should need be. Thanks to using two systems to test the theory on, external validity is ensured to a greater extent. Without this, it could be the solution was too tailor-made to fit only one system and only within the context of SBA.

5.2 Course of action

The course of action consists of four distinct phases; problem definition, data collection and analysis, practical application and definition of a process. Structuring the work into these phases greatly helped with planning.

However, the initially allocated time to each phase was not enough, particularly in regard to the practical application and the definition of the process. Both of these phases have run in parallel, although they have different purposes. In the end, this seems natural instead of the initially planned waterfall type of work that was planned. I believe this has resulted in a more accurate end result, and has also helped contribute to greater validity.

5.2.1 Problem definition

The overall aim of this phase was to grasp the whole problem stated in the research question. I have observed during several occasions how the operations team do a deployment to production. I chose to do semi-structured interviews, as I had no clue of what kind of answers to expect from the interviewees. This type of interview format gave me the opportunity to ask follow-up questions.

These interviews were a great help when looking for literature concerning the subject, since many of the terms were not known to me beforehand. It also gave me a chance to get an understanding of the organization and the people usually involved in the deployment procedure.
5.2.2 Data collection and analysis

In the second phase, I examined existing literature and practices and tried to correlate these to SBA. I wanted to identify which practices were commonly outlined between different literatures, and draw parallels to the way SBA worked. The literature study has mainly revolved around books, but I have found it useful to dive deep into the few scientific papers that I found.

What I found most useful was papers and books that described common anti-patterns. These anti-patterns were easy to identify within SBA thanks to the knowledge gained during the problem definition. They were also an excellent starting point for further research. From the problem definition, I also learned of many companies that use the practices outlined in this thesis, but unfortunately there is little to no information available from these companies about how they actually did it. A common denominator that I noticed among those who already adopted this, is that they are mostly giants in the computing industry.

Because Continuous Delivery is an umbrella term, finding literature about it has been quite difficult with no prior knowledge. Instead, I found it better to focus on the integral parts of the subject and used keywords such as software quality, software testing, continuous integration and configuration management. Even then, many of these do not give sufficient hits. My main trouble was that many books concerned themselves with CI or configuration management using a specific technology, whereas I have only been interested in technology agnostic solutions.

My main problem during this phase has been finding sources other than blogs. Many sources are very well written, but verifying their validity has not always been easy. During the process of reading I did find several frequently occurring names, which I have used as a baseline to find more information. These names are major contributors to modern development practices, and I found that their books and articles was a great source of knowledge. They would be what is referred to as innovators in Error! Reference source not found..

Being critical of this stage, many sources I have used are from textbooks or straight documentation of some APIs (e.g. Git). However, I have applied this in later phases and drawn conclusions of my own. I also deem the sources and their authors to be highly reliable.

5.2.3 Practical application

Based on the previous phase, I had identified several anti-patterns within the organization. With these as a starting point, I got to work trying to contrast my new knowledge and find the weakest spots in current procedures.

Before I began with anything, I talked to members of the development teams of each system I had chosen to get an understanding of how they worked. With this, I had a textual representation from which I could easily visualize a proposed pipeline, and using the knowledge from the literature review it was easy to convey the importance of all aspects involved.

During the literature review and data gathering, I found a lot of tools that would be very useful to SBA. Unfortunately, there has not been much time to evaluate these. Instead, I chose to use the tools that were either already implemented or evaluate only one tool if it met the following criteria:

- It measures up against competitors
- It fulfills requirements within SBA
- It has been around for some time
- It is technology agnostic
- It is open source

These requirements have been validated thanks to having two systems to test with, both written in different languages and both built and processed using the same tools.
A lot of new and complementary knowledge was gained during this phase, and laid the ground for the process I aimed to lay out. Through each stage; version control, configuration management, continuous integration and pipeline design, I have either done the work from scratch or improved on something existing. For instance, version control was already in place, but from how it is being used currently I came to several conclusions that made their way into the process, one of them being that there is no coherent branching strategy between projects.

Each of the four parts in this phase has not been done in isolation, but rather continuously improved alongside each other. For instance, when I had successfully placed the application configuration in version control, I was able to further refine my pipeline. From just being able to run unit tests in it, I was now able to deploy the systems to pre-existing environments if all their tests succeeded.

Seeing this gradual increase in usefulness in the pipelines has been a true motivator, and it has garnered the attention from many stakeholders. I’ve had huge amounts of help through technical test leaders and the QA team from Jorden who enabled Jorden to run acceptance and regression tests.

The style guide was a significantly smaller undertaking, and as such acted as a springboard into the next, more complex system. Thanks to having a simple outline of the necessary steps, this undertaking was possible during this thesis. When using the initial process in Jorden, it became clear that it had shortcomings. These were overcome during the implementation, and the process refined to meet higher requirements.

5.2.4 Defining the process

This phase aimed to combine all previous phases into something tangible and easy to grasp. I wanted to make it easy to access and to get started with, and thanks to the incremental nature of the chosen method I have found many pitfalls, helpful tips and come to conclusions that I believe can fit nearly any organization or team of arbitrary size. I say nearly, because I am unsure of some performance aspects when dealing with truly huge organizations like Google, but unfortunately, I’ve had no way to verify this. Still, the method outlined in this thesis is as general as can get.

However, I would recommend beginners to read more about each aspect of the pipeline. The theory in this thesis combined with the process outlined in Table 9 is sufficient to get a running start, although it is impossible to go into detail because details between organizations and systems will differ.

5.3 Discussion of findings

Here, I will discuss the findings of the thesis with respect to the purpose and the stated research question. To reconnect with this, my aim was to demystify the concept that this thesis concerns, and bring a more streamlined way for the uninitiated to get comfortable with the concepts and get started more easily. In addition, I also wanted the process to be applicable regardless of the organizations size.

From the literature study, it became clear that continuous delivery is a mature approach to software development and it requires discipline from everyone involved. It also became evident that it is a major paradigm shift, both for how developers think and how organizations deliver their software. With this in mind, I believe that the choice to only focus on technical aspects, as outlined in chapter 1.3, was the wrong decision.

I had no particular problems introducing these concepts to involved developers, but I fear that culture and misconceptions will hinder this going forward to its full extent. In hindsight, I would have limited my practical phase and instead focused on providing this knowledge to key people in the organization. This way it is more likely to be a continuous work of improvement that would also permeate to other systems, not just those I’ve been involved with.
Implementing continuous delivery will permeate through entire organizations, not just development teams. To reap all the benefits, it is therefore important to not only involve developers but also the operations team, managers and even customers. Those writing specifications will have to be taught to see from a developer’s point of view as there should be little to no room for interpretation of a feature. Developers need to think more like testers. Initially, this will probably feel like it slows your work down, and it is slowing it down, but only for a limited period of time. This “lost” time will be regained several times as you do not have to concern yourselves with many of the manual steps you did previously.

I found it to be extremely valuable to involve other roles, as it is easy to get stuck in your own bubble and miss problems that might be obvious to an outside viewer. Once again, my delimitations in chapter 1.3 are something that I would like to have revised. It is nearly impossible for a single individual to possess all knowledge needed to get value out of a pipeline. Developers, testers, UX designers and managers all need to be involved. This makes sure test suites are correct and that we are testing for the right things, it increases confidence in the approach and will yield better results overall.

5.4 Conclusions and recommendations

The following sections presents the conclusions of this thesis, followed by recommendations and proposed further work.

5.4.1 Conclusions

By using a Lean approach through each practical application stage throughout this thesis, I have been able to obtain common steps needed to reach your goal, divided into separate phases. Using existing software testing practices, I have implemented test strategies, test suites and pipelines on real systems in the organization.

Tools will help greatly with every aspect of this, and I recommend that you use them extensively. There is no need to reinvent the wheel, though you should be careful to introduce unnecessary complexity. If you set up a tool and only have one key person that can operate it, chances are you are going to put yourself in a precarious situation if this person disappears.

Ensure every developer can understand the tools used, and ensure everyone can contribute to the test suite. If everyone cannot contribute, chances are high they will find a workaround to not run these tests. Tests should not be maintained by a separate team, they should be maintained by each application and its developers.

If you feel that your development process has slowed down for a significant amount of time, and you can identify the bottleneck, make sure to dispose of this immediately and revise your solution.

5.4.2 Recommendations and further work

In order to approach this subject in a more organized way, I highly recommend forming a cross-functional team dedicated to explore options for continuous delivery in your organization. Ensure to form this team from the onset and that the team has a passion and makes this a top priority. Do not include someone simply because they have the right credentials as this might do more harm than good.

For increased validity, I would like to test this process outside of SBA. Validity is fairly high thanks to using several systems, one in its beginning and one fully mature and complex. However, I cannot be entirely sure that there aren’t organizational aspects that need to be considered elsewhere. Since I have been tasked with writing this thesis, I have had fairly free reins to do as I wish.

The majority of continuous delivery revolves around having a robust and easily maintainable test suite. If this is already in place and runnable via command line, getting this into a pipeline should be trivial.

When implementing this, a good metric to use is deployments to production and time from feature request to deployment in production. Whether you choose to use continuous delivery or continuous deployment, you should see a significant increase in your delivery speed.
References


References


7 Appendices

7.1 Appendix 1: Reasoning behind choosing Git

7.1.1 Original text

**Bakgrund**
Vi har behov av att byta vårt versionshanteringsverktyg för vår kod på JUP. SVN som är vårt nuvarande verktyg börjar bli omodernt och vi behöver annan funktionalitet som finns i GIT.

Effekter som vi vill uppnå med att byta till GIT:
- Öka prestanda för utvecklarna
- Spara tid för utvecklarna
- En enhetlig process för hantering av kod
- Minska fel genom en bättre överblick
- Förutsättningar för bättre versionshantering
- Snabbare hantering av brancher
- Enklare att merge – färre konflikter
- Öka kvalitén på releaser till våra miljöer
- Snabbare task-switching
- Underlätta distansarbete
- Använda moderna verktyg

GIT är ett modernt och det starkaste verktyget just nu på marknaden.

Vi har genomfört en kort omvärldsbekakning och inte funnit några andra aktuella kandidater.

**Beslut**
Införa GIT för JUP.

**Konsekvens och Risk**
- Att vi inte kommer att sätta upp GIT på rätt sätt.
- Att vi inte avslutar SVN och kommer att leva med två versionshanteringssystem för JUP.

**Kostnad**
Det kan uppkomma licenskostnader, uppdraget ska ta fram ett kostnadsförslag.

Investering av vår tid och kompetensutveckling. Varje projekt som ska gå över till GIT ska bekosta det själva.

7.1.2 Translated text

**Background**
We need to replace our version management tool for our code on JUP. SVN, which is our current tool, begins to become outdated and we need other functionality found in GIT.

Effects that we want to achieve by switching to GIT:
• Increase performance for developers
• Save time for developers
• A uniform process for code management
• Decrease errors by better overview
• Prerequisites for better version management
• Faster branch management
• Easier to merge, fewer conflicts
• Increase quality of releases to our environments
• Faster task switching
• Facilitate remote work
• Using modern tools

GIT is modern and the strongest tool at the market right now. We have performed a short analysis and not found any suitable candidates.

**Decision**
Introduce GIT for JUP.

**Consequences and risk**
- Incorrect setup of GIT
- Not fully migrating from SVN and having two co-existing version control systems

**Cost**
There may be licensing costs. The group will make a quotation.

Investing our time and skills development. Every project that will go over to GIT will pay for it using their own budget.

### 7.2 Appendix 2: Example of how the style guide operates

A developer may choose to simply apply a style to a generic HTML markdown button. The HTML would then be written as follows:

```html
<button class="c-btn">Default</button>
<button class="c-btn--primary">Primary</button>
<button class="c-btn--secondary-alt">Secondary Alternative</button>
```

When navigating to the page containing the markdown, the following will be displayed:

[Default] [Primary] [Secondary Alternative]

### 7.3 Appendix 3: Deploy script for the style guide

```bash
# Perform checkout of given branch
git checkout $source
# Update package.json to given version
npm version $sjvVersion
# Install latest dependencies in temporary workspace
npm install
# Publish package to internal npm repo
npm publish
```
# Build release files and documentation

```bash
npm run build:all
```

# Update release notes with given version

```bash
sed -i "s/{tag}/${sjvVersion}/g" $WORKSPACE/release.md
```

# Create json body to update tag in git

```bash
node -e "require('./create-notes.js').createReleaseNotesBody('${sjvVersion}')"
```

# Push the tagged release to git

```bash
git push --tags
```

# Send json to newly pushed tag

```bash
curl --header "Content-Type: application/json; charset=UTF-8" --X POST --data @$WORKSPACE/release_${sjvVersion}.json http://git/api/v3/projects/539/repository/tags/v${sjvVersion}/release?private_token=REDACTED
```

# Declare a value map

```bash
declare -A map
```

# File path to properties for chosen environment

```bash
file=$WORKSPACE/environments/$ENV
```

# Read each line into map

```bash
while IFS=' ' read -r file || [[ -n "$line" ]]; do
 IFS='=' read -r a keyValue <<< "$line"
  map[keyValue[0]] = keyValue[1]
done
```

# Send release and documentation site to node

```bash
scp -r $WORKSPACE/dist/release
scp -r $WORKSPACE/dist/site
scp -r $WORKSPACE/assets
```

# Declare a value map

```bash
git push
```

## 7.4 Appendix 4: Karma testing configuration

/* Requirements */

```javascript
const path = require('path');
/* Webpack configuration */

let webpackConfig = require('./webpack.prod');
// Entries are handled by karma in 'files' below
webpackConfig.entry = undefined;
// Enable quicker source maps
webpackConfig.devtool = 'inline-source-map';
// Library config in webpack conf breaks tests:
/*
Uncaught TypeError: Failed building the required modules.
at karma/index.ts:1
*/
// So override library config while testing
webpackConfig.output = {
  path: path.resolve(__dirname, 'dist/release'),
  filename: 'sjv-[name].js',
  chunkFilename: 'sjv-[name].js',
}
/* Karma configuration */

let customLaunchers = {
  /* Run tests against installed version of Chrome
   * ChromeInstallationHeadless: {
   *   base: 'Chrome',
   *   flags: [
   *     '--no-sandbox',
   *     '--headless',
   *     '--disable-gpu',
   *     // Without a remote debugging port, Google Chrome exits immediately.
   */
module.exports = function (config) {
    config.set({
        basePath: '',
        frameworks: [ 'mocha', 'chai', 'karma-typescript' ],
        files: [
            // polyfill for accessibilityjs
            "spec/karma/polyfill/closest.polyfill.ts",
            // accessibilityjs
            "node_modules/accessibilityjs/dist/index-umd.js",
            // For coverage
            "src/**/*.ts",
            // For performance, use only one entry point
            // and include tests from there
            "spec/karma/index.ts"
        ],
        exclude: [],
        // https://github.com/angular/angular-cli/issues/2125#issuecomment-247395088
        mime: { 'text/x-typescript': ['ts'] },
        preprocessors: { 'src/**/*.ts': [ 'karma-typescript' ],
            'spec/karma/index.ts': ['webpack', 'sourcemap'] },
        webpack: webpackConfig,
        reporters: [ 'mocha', 'karma-typescript' ],
        port: 9876,
        colors: true,
        logLevel: config.LOG_INFO,
        autoWatch: false,
        browsers: [ 'ChromeHeadless' ],
        customLaunchers: customLaunchers,
        singleRun: true,
        concurrency: Infinity
    })
}

7.5 Appendix 5: Puppeteer test, screenshots

describe('screenshot test', function () {
    it('should take a screenshot', async () => {
        await window.puppeteerCommand({
            type: 'screenshot', // The page command to run
            args: []
        })
    })
})
### 7.6 Appendix 6: Scripting queues and databases

Adding a database:

```
<subsystem xmlns="urn:jboss:domain: WildFly:22.0.0"
    subsystem="datasources"
    xa-data-source="utskicksbegaran"
    jndi-name="jndiName",
    enabled=true,
    driver-name="driver",
    xa-data-source-class="oracle.jdbc.xa.client.OracleXADatasource",
    user-name="user",
    password="password",
    valid-connection-checker-class="org.jboss.jca.adapters.jdbc.extensions.oracle.OracleValidConnectionChecker",
    background-validation=true,
    stale-connection-checker-class="org.jboss.jca.adapters.jdbc.extensions.oracle.OracleStaleConnectionChecker",
    exception-sorter-class="org.jboss.jca.adapters.jdbc.extensions.oracle.OracleExceptionSorter",
    same-rm-override=false,
    no-tx-separate-pool=true
/>
```

Adding a message queue:

```
<subsystem xmlns="urn:jboss:domain: WildFly:22.0.0"
    subsystem="resource-adapters"
    resource-adapter="activemq-rar.rar"
    admin-objects="ActiveMQQueue.\$queueName"
    jndi-name="java:/$queueName",
    enabled=true,
    use-java-context=true
/>
```

<table>
<thead>
<tr>
<th>Date</th>
<th>Git</th>
<th>SVN</th>
</tr>
</thead>
<tbody>
<tr>
<td>2016-09-18</td>
<td>84</td>
<td>14</td>
</tr>
<tr>
<td>2016-09-25</td>
<td>83</td>
<td>15</td>
</tr>
<tr>
<td>2016-10-02</td>
<td>83</td>
<td>14</td>
</tr>
<tr>
<td>2016-10-09</td>
<td>84</td>
<td>14</td>
</tr>
<tr>
<td>2016-10-16</td>
<td>88</td>
<td>15</td>
</tr>
<tr>
<td>2016-10-23</td>
<td>89</td>
<td>15</td>
</tr>
<tr>
<td>2016-10-30</td>
<td>86</td>
<td>13</td>
</tr>
<tr>
<td>2016-11-06</td>
<td>86</td>
<td>14</td>
</tr>
<tr>
<td>2016-11-13</td>
<td>94</td>
<td>15</td>
</tr>
<tr>
<td>2016-11-20</td>
<td>82</td>
<td>13</td>
</tr>
<tr>
<td>2016-11-27</td>
<td>89</td>
<td>14</td>
</tr>
<tr>
<td>2016-12-04</td>
<td>89</td>
<td>14</td>
</tr>
<tr>
<td>2016-12-11</td>
<td>87</td>
<td>13</td>
</tr>
<tr>
<td>2016-12-18</td>
<td>80</td>
<td>12</td>
</tr>
<tr>
<td>2016-12-25</td>
<td>53</td>
<td>8</td>
</tr>
<tr>
<td>2017-01-01</td>
<td>73</td>
<td>11</td>
</tr>
<tr>
<td>2017-01-08</td>
<td>91</td>
<td>13</td>
</tr>
</tbody>
</table>
7.8 Appendix 8: Jenkins Pipeline script for Jorden

def commentMr(gitlabMergeRequestIid, String comment) {
    // Not allowed to use java.net.URLEncoder.encode
    def body = comment
        .replaceAll(" ", "%20")
        .replaceAll("/", "%2F")
    sh "curl http://git.intern.jordbruksverket.se/api/v4/projects/jbr%2Fjorden/mer"
node ('java1.7-maven-3.3.9') {
    def mvnHome = tool 'Maven 3.3.9'
    def mrUrl = "http://git.intern.jordbruksverket.se/jbr/jorden/merge_requests/$gitlabMergeRequestId"
    def friendlyBranch = env.gitlabSourceBranch.replaceAll("[^a-zA-Z0-9]","")
    def friendlyName = friendlyBranch + "-" + env.gitlabMergeRequestLastCommit.substring(0,10)
    def friendlyNameEar = "jorden-ear/target/jorden-ear-$friendlyName.ear"
    def builtEarUrl = "${BUILD_URL}artifact/"+friendlyNameEar;
    currentBuild.description = "$gitlabMergeRequestTitle
$gitlabSourceBranch
$mrUrl
$builtEarUrl"
    commentMr(env.gitlabMergeRequestIid, ":spy: Verifierar $gitlabSourceBranch i ${BUILD_URL}"")
    currentBuild.displayName = env.gitlabSourceBranch + " i " + env.gitlabTargetBranch + " " + gitlabMergeRequestTitle
}

stage('Merge') {
    sh "git init"
    sh "git fetch --no-tags git@jbr:jorden.git +refs/heads/*:refs/remotes/origin/* --depth=200"
    sh "git checkout origin/${env.gitlabTargetBranch}"
    sh "git merge origin/${env.gitlabSourceBranch}"
    sh "git log --graph --abbrev-commit --max-count=10"
}

stage('Compile') {
    try {
        sh "${mvnHome}/bin/mvn -U -q package -DskipTests -Dmaven.test.failure.ignore=true -Dcheckstyle.skip=true -Dfindbugs.skip=true -Dsurefire.skip=true -Dmaven.compile.fork=true -Dmaven.javadoc.skip=true"
        sh "mv jorden-ear/target/jorden-ear.ear ${friendlyNameEar}"
        archiveArtifacts artifacts: '*/target/*.ear', fingerprint: true
    //commentMr(env.gitlabMergeRequestIid, "Ear-filen för $gitlabSourceBranch kan nu laddas ner från ${BUILD_URL}")
    }
    catch (e) {
        commentMr(env.gitlabMergeRequestIid, ":x: Kompilerar inte :worried: $gitlabSourceBranch ${BUILD_URL}"")
        throw e
    }
}

stage('Unit test') {
    try {
        sh "${mvnHome}/bin/mvn test -Dmaven.test.failure.ignore=false -Dcheckstyle.skip=true -Dfindbugs.skip=true -Dsurefire.skip=false -Dmaven.compile.fork=false -Dmaven.javadoc.skip=true"
commentMr(env.gitlabMergeRequestId, ":heavy_check_mark: Tester ok :) smiley: $gitlabSourceBranch ${BUILD_URL}"
junit "**/target/surefire-reports/**/*.xml"
} catch (e) {
  commentMr(env.gitlabMergeRequestId, ":x: Tester **ej ok** :worried: i $gitlabSourceBranch ${BUILD_URL}"
junit "**/target/surefire-reports/**/*.xml"
  throw e
}
}

stage('Regression test') {
  build job: 'jorden-regressionstest-pipeline', wait: false,
  parameters: [
    [ $class: 'StringParameterValue', name: 'gitlabSourceBranch',
      value: env.gitlabSourceBranch],
    [ $class: 'StringParameterValue', name: 'gitBranch', value: env.gitlabTargetBranch],
    [ $class: 'StringParameterValue', name: 'gitlabMergeRequestId',
      value: env.gitlabMergeRequestId]
  ]
}

stage('Static code analysis') {
  sh "[-f build-tools/checkstyle.xml ] || ${mvnHome}/bin/mvn -q
  pmd:check findbugs:check package -DskipTests -Dmaven.test.failure.ignore=false -Dsurefire.skip=true -Dmaven.compile.fork=true -Dmaven.javadoc.skip=true -P ci -Dpmd.failOnViolation=false"
  sh "[-f build-tools/checkstyle.xml ] && ${mvnHome}/bin/mvn -q
  pmd:check findbugs:check package -DskipTests -Dmaven.test.failure.ignore=false -Dsurefire.skip=true -Dmaven.compile.fork=true -Dmaven.javadoc.skip=true || echo ok"
  step([
    $class: 'ViolationsToGitLabRecorder',
    config: [
      gitLabUrl: 'http://git.intern.jordbruksverket.se/',
      projectId: "393",
      mergeRequestId: env.gitlabMergeRequestId,
      commentOnlyChangedContent: true,
      createCommentWithAllSingleFileComments: true,
      minSeverity: 'INFO',
      useApiToken: true,
      apiToken: 'REDACTED',
      useApiTokenCredentials: false,
      apiTokenCredentialsId: 'id',
      apiTokenPrivate: true,
      authMethodHeader: true,
      ignoreCertificateErrors: true,
      violationConfigs: [
        [ pattern: '.*/checkstyle-result\.xml$', parser: 'CHECKSTYLE', reporter: 'Checkstyle' ],
        [ pattern: '.*/findbugsXml\..*\.xml$', parser: 'FINDBUGS', reporter: 'Findbugs' ],
        [ pattern: '.*/pmd\..*\.xml$', parser: 'PMD', reporter: 'PMD' ],
      ]
  ]
}
def commentMr(gitlabMergeRequestId, comment) {
    // Not allowed to use java.net.URLEncoder.encode
    def body = comment.replaceAll(" ", "%20")
    if (currentBuild.result != 'UNSTABLE' && currentBuild.result != 'FAILURE' && currentBuild.result != 'ABORTED') {
        commentMr(env.gitlabMergeRequestIid, ":heavy_check_mark: Fitnesse ok :smiley: $gitlabSourceBranch ${BUILD_URL}")
    } else {
        // commentMr(env.gitlabMergeRequestIid, ":x: Fitnesse **ej ok** :worried: i $gitlabSourceBranch ${BUILD_URL} , de har nyligen börjat köras på feature-brancherna så behöver inte vara ditt fel! https://utv.sjv.se/xwiki/bin/view/JUP/FitnesseTesterIJordenMaven")
        // currentBuild.result = 'FAILURE'
    }
}

7.9 Appendix 9: Jenkins Pipeline regression script for Jorden

def commentMr(gitlabMergeRequestId, comment) {
    // Not allowed to use java.net.URLEncoder.encode
    def body = comment.replaceAll(" ", "%20")
    if (currentBuild.result != 'UNSTABLE' && currentBuild.result != 'FAILURE' && currentBuild.result != 'ABORTED') {
        commentMr(env.gitlabMergeRequestIid, ":heavy_check_mark: Fitnesse ok :smiley: $gitlabSourceBranch ${BUILD_URL}")
    } else {
        // commentMr(env.gitlabMergeRequestIid, ":x: Fitnesse **ej ok** :worried: i $gitlabSourceBranch ${BUILD_URL} , de har nyligen börjat köras på feature-brancherna så behöver inte vara ditt fel! https://utv.sjv.se/xwiki/bin/view/JUP/FitnesseTesterIJordenMaven")
        // currentBuild.result = 'FAILURE'
    }
}

Appendices
def remoteCommand(command):
    sh 'ssh docker@vl-dockertest03 "' + command + '"'

def getRemoteFile(remotePath, localPath):
    sh 'scp -r docker@vl-dockertest03:' + remotePath + ' ' + localPath

def getRemoteCommandStdout(command):
    def out = sh script: 'ssh docker@vl-dockertest03 "' + command + '"', returnStdout: true
    echo(command + ' gav ' + out)
    return out

def cleanUp(workingDir, projectName):
    try {
        remoteCommand('cd ' + workingDir + '; ./farm -p ' + projectName + ' down')
        getRemoteFile(workingDir + '/docker-app-image', '.')
        archiveArtifacts artifacts: '**/log/server*', fingerprint: true
        remoteCommand('docker network prune -f')
    } catch (e) {
        echo('Fel inträffat när container skulle tas ner.')
    }

    remoteCommand('cd ' + workingDir + '; ./farm -p ' + projectName + ' status')
    try {
        remoteCommand('rm -rfv ' + workingDir)
    } catch (e) {
        echo('Fel inträffat när working dir skulle tas bort.')
    }
}

node ('java1.7-maven-3.3.9') {
    def isBuildingMr = env.gitlabSourceBranch?.trim()
    if (isBuildingMr) {
        currentBuild.displayName = env.gitlabSourceBranch + ' i ' + env.gitBranch
    } else {
        currentBuild.displayName = env.gitBranch
    }
    def projectName = currentBuild.displayName.replaceAll("[^a-zA-Z0-9]"apers.replaceAll("/", "%2F")
    def workingDir = 'jorden-' + projectName
    def templateDir = '/home/jorden/docker-template_v6'
    remoteCommand('cp -rv ' + templateDir + ' ' + workingDir)
deleteDir()

remoteCommand('ls')
remoteCommand('git --version')
remoteCommand('docker --version')
remoteCommand('docker-compose --version')
remoteCommand('docker ps')

try {
    stage('Start test container') {
        if (isBuildingMr) {
            remoteCommand('cd ' + workingDir + '; ./farm -p ' + projectName + ' build-ear --ci origin/' + env.gitBranch + ' origin/' + env.gitlabSourceBranch)
        } else {
            remoteCommand('cd ' + workingDir + '; ./farm -p ' + projectName + ' build-ear --ci origin/' + env.gitBranch)
        }
        remoteCommand('cd ' + workingDir + '; ./farm -p ' + projectName + ' status')
    }

    stage('Build Jorden image') {
        remoteCommand('cd ' + workingDir + '; ./farm -p ' + projectName + ' build-image jorden.ci.properties standalone-full.ci.xml')
    }

    stage('Start Jorden, AMQ and DB containers') {
        timeout(time: 30, unit: 'MINUTES') {
            remoteCommand('cd ' + workingDir + '; ./farm -p ' + projectName + ' start --ci && ./farm -p ' + projectName + ' healthcheck 15m')
            remoteCommand('cd ' + workingDir + '; ./farm -p ' + projectName + ' status')
        }
    }

    stage('Perform tests') {
        try {
            if (isBuildingMr) {
                timeout(time: 30, unit: 'MINUTES') {
                    remoteCommand('cd ' + workingDir + '; ./farm -p ' + projectName + ' test-exec')
                }
            } else {
                timeout(time: 10, unit: 'HOURS') {
                    remoteCommand('cd ' + workingDir + '; ./farm -p ' + projectName + ' test-exec regtest')
                }
            }
            getRemoteFile(workingDir + '/docker-compose', '.')
            junit "**/failsafe-reports/**/*.xml"
        } catch (e) {
            getRemoteFile(workingDir + '/docker-compose', '.')
            junit "**/failsafe-reports/**/*.xml"
            throw e
        }
        if (currentBuild.result == 'UNSTABLE') {
//junit sätter UNSTABLE om det finns testfel, slänger exception så att vi failar.
    currentBuild.result = 'FAILURE'
    throw new RuntimeException("Tester failar")
}

} catch (e) {
    if (isBuildingMr) {
        commentMr(env.gitlabMergeRequestId, ":x: Regressionstester **ej ok** :worried: i $gitlabSourceBranch ${BUILD_URL}"
    }
    cleanUp(workingDir, projectName)
    throw e
}