A model for increasing the performance of an automated testing pipeline

Martin Nordsletten
Master’s Thesis Autumn 2017
A model for increasing the performance of an automated testing pipeline

Martin Nordsletten

August 1, 2017
Abstract

Projects that utilize a continuous integration pipeline will notice that the pipeline is a dynamic product which needs constant improvements and continuous support. As the main software project evolves the needs for the support infrastructure provided by the pipeline also evolves. The pipeline is not a static product that is set up once and left alone. With lots of moving pieces it is not always apparent what to prioritize when working on the pipeline.

The aim of this thesis is therefore to develop a model and terminology for developing efficient, automated and continuously improving testing pipelines based on a real life case study. The case study in question was performed over an extended period of time working with implementing a continuous integration pipeline for a real software development project. The need for the model came from seeing how the pipeline went through multiple larger revisions.

The model was created from the experiences gathered through the case study. Focused discussions with the supervisor helped form the clear parts of the model. The end outcomes from using the model are to create a framework which helps translate project goals into specific goals for the pipeline. Then use those goals to create a continuously improving pipeline where the most important goals of the project are satisfied first. This creates a dynamic pipeline which helps with transparency in the organization.

There is a need for future work to document usage of the model as the model was created after the case study was performed.
## Contents

1 Introduction

2 Background
   2.1 Historical aspect of DevOps and CI
      2.1.1 Why DevOps became necessary
      2.1.2 How does a company move towards DevOps?
      2.1.3 How to achieve higher quality and improved velocity
   2.2 Unikernels
      2.2.1 IncludeOS
   2.3 Continuous integration
      2.3.1 Starting to use continuous integration
      2.3.2 Importance of CI
      2.3.3 Pipeline examples
   2.4 Continuous integration tools
      2.4.1 Jenkins
      2.4.2 Openstack
      2.4.3 Git and Github
   2.5 Relevant research
      2.5.1 Including Performance Benchmarks into Continuous
            Integration to Enable DevOps
      2.5.2 Performance-oriented DevOps: A Research Agenda

3 Approach
   3.1 Defining the case study
   3.2 Planning the model
      3.2.1 Terminology
      3.2.2 Activities
      3.2.3 Quality assurance
   3.3 Analyzing the end result
      3.3.1 Properties for measuring success
      3.3.2 Assessing properties

4 Results 1: Case study: A solution for IncludeOS
   4.1 State of the code
   4.2 Start of automation
      4.2.1 Openstack as infrastructure
      4.2.2 Implementing Jenkins
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>A example fully implemented pipeline. (Chen, 2015)</td>
<td>12</td>
</tr>
<tr>
<td>2.2</td>
<td>The current implementation of a continuous integration system in IncludeOS</td>
<td>13</td>
</tr>
<tr>
<td>3.1</td>
<td>The tasks that have to be completed in the case study</td>
<td>19</td>
</tr>
<tr>
<td>3.2</td>
<td>Relationship between the case study and the model</td>
<td>21</td>
</tr>
<tr>
<td>3.3</td>
<td>Workflow when working with the model</td>
<td>22</td>
</tr>
<tr>
<td>4.1</td>
<td>State of the test system at the start of the project</td>
<td>28</td>
</tr>
<tr>
<td>4.2</td>
<td>Shows how there is a divide between test results after clearing environments</td>
<td>30</td>
</tr>
<tr>
<td>4.3</td>
<td>When a test run fails on the developers machine, but clears the test server the result should be trusted</td>
<td>31</td>
</tr>
<tr>
<td>4.4</td>
<td>The stages of the pipeline when jenkins is introduced</td>
<td>32</td>
</tr>
<tr>
<td>4.5</td>
<td>Flow showing how VM’s were spawned</td>
<td>33</td>
</tr>
<tr>
<td>4.6</td>
<td>Clear distinction between internet and private cloud</td>
<td>33</td>
</tr>
<tr>
<td>4.7</td>
<td>Shows the processes that are triggered when a pull request comes in</td>
<td>35</td>
</tr>
<tr>
<td>4.8</td>
<td>Expanding the testing stage with multiple operating systems and multiple test categories</td>
<td>36</td>
</tr>
<tr>
<td>4.9</td>
<td>An example build matrix for different operating systems</td>
<td>37</td>
</tr>
<tr>
<td>4.10</td>
<td>With an increasing number of tests the need of an overview increases</td>
<td>41</td>
</tr>
<tr>
<td>4.11</td>
<td>A new dedicated test tool, changes to the visualization and persistent VMs.</td>
<td>42</td>
</tr>
<tr>
<td>4.12</td>
<td>Before categorizing tests they had to be manually added to the various jobs</td>
<td>43</td>
</tr>
<tr>
<td>4.13</td>
<td>By introducing attributes to the tests they were much easier to control</td>
<td>44</td>
</tr>
<tr>
<td>4.14</td>
<td>The IncludeOS router is connected to two networks with two interfaces. The host machine needs to do the same.</td>
<td>45</td>
</tr>
<tr>
<td>4.15</td>
<td>There is a fixed compilation time regardless of the test. The test duration differs with the type of test.</td>
<td>46</td>
</tr>
<tr>
<td>4.16</td>
<td>Centralized build stage and distributed test execution.</td>
<td>46</td>
</tr>
<tr>
<td>4.17</td>
<td>Modular tests allows for dynamic reordering of the tests in execution</td>
<td>47</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>-----------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>5.1</td>
<td>Disconnect between management and goals and specifics of the pipeline.</td>
<td>53</td>
</tr>
<tr>
<td>5.2</td>
<td>By introducing goal based management to the CI pipeline the process becomes much more transparent.</td>
<td>54</td>
</tr>
<tr>
<td>5.3</td>
<td>Workflow when using POP showing the three main activities</td>
<td>56</td>
</tr>
<tr>
<td>5.4</td>
<td>Flow for the goal alignment activity</td>
<td>58</td>
</tr>
<tr>
<td>5.5</td>
<td>Process for setting up the tracking of the KPI variables</td>
<td>62</td>
</tr>
<tr>
<td>5.6</td>
<td>Process for going from goals and KPI variables into actionable steps</td>
<td>66</td>
</tr>
<tr>
<td>5.7</td>
<td>Process summary with included deliverables</td>
<td>70</td>
</tr>
</tbody>
</table>
List of Tables

6.1 Table evaluating the terms described in the terminology . . . 74
Preface

The thesis named "A model for increasing the performance of an automated testing pipeline" was created after having worked with a continuous integration pipeline for a year and experiencing first hand the issues of continuously updating it. The thesis is part of the masters program Network and System Administration at the University of Oslo (UiO). The work with the thesis started in January of 2016 and concluded in August of 2017.

The majority of the work is based upon the work I performed while working for IncludeOS, a startup creating a unikernel operating system. The experiences from this extended period of time helped create a thesis which is based on a real life scenario. The problem statement at the beginning of the project turned out to be difficult to answer, so a slight change of direction was done after knowing which problems were most pressing. This presented some problems to the structure of the thesis work as the normal order of writing a thesis where the background chapter comes first followed by the actual work was skewed.

Performing the change of direction and changes to the problem statement were only possible due to the help of my supervisor Kyrre Begnum. He helped identify which issues I was having when trying to answer my original problem statement. From these discussions grew what turned into the final model which is now the main part of the thesis. The discussions were invaluable to the final product and thank you also for all the great advice on structuring and writing the master thesis.

I would also like to thank the team at IncludeOS, especially Alfred Bratterud and Per Buer, who allowed me to have very free control over the development of the pipeline. Thanks also to Mike Long who read the model and provided feedback for my analysis.

Finally thank you to my wife Ingrid and my son Hugo who provided invaluable support for me to finish.

Martin Nordsletten Oslo, July 27, 2017
Chapter 1

Introduction

Over the past 20 years software has changed the way we live our daily lives, this growth and dependency will only continue in the years to come. If we look at the past 20 years the move to digital solutions for most things in our lives has been made. Services like banking, communication, shopping, planning trips and getting information are all used daily. People now expect to have all the information they require available at their fingertips. Services that do not offer a digital alternative will quickly become outdated and unused. If a competitor in your field of business starts offering an online shopping solution the marketing potential and draw on customers will be noticeable. This puts pressure on the development of software not only to be available, but to quickly adapt to the ever changing environment.

Software today is therefore expected to grow and adapt to the world around them. New competitors and security concerns will quickly make non-adapting software obsolete. In a world where a simple thought can go from idea to a prototype making money in a matter of days, having a short time to market for your product can mean everything. When a product is released today it is also expected to be supported and constantly updated over the coming years. Being able to push out updates and fixes to known problems will keep users content. There is also a security aspect to being able to push out updates quickly. Security vulnerabilities are discovered on a regular basis and if these are not fixed in your application they could potentially spiral out of control and cause harm in the way of exposed sensitive information. The monetary value and negative publicity of such exposure is immense.

In order to keep up in this increasingly dynamic environment one area that has received more attention is the field of release engineering. This entails the process from finished code to putting the code into production. It is an area that benefits greatly from automation. With a proper release pipeline the software gets built, tested and put into production automatically. If the pipeline is well built you can be sure that the production software lives up to the standard defined by the tests it goes through.
The pipeline is by itself a complex product which has to evolve over time. There are technological advancements to the tools used in the pipeline that need to be considered. The goals of the software the pipeline is supporting will also most likely change over time so the pipeline needs to adapt as well. It is not always apparent in what way the pipeline should change. The project could require very detailed testing which will take a long time to perform, or it might prioritize a very fast pipeline which provides feedback much more quickly. These design decisions mean that the software project and the pipeline need to have their goals properly aligned.

In order for the change in goals to quickly adapt over to the pipeline there is also a need for the pipeline to reevaluate its design. There is a need for a system that performs this evaluation in an organized cyclical manner. By having this system in place the whole process from designing software to testing to releasing is transparent and all working towards the same goal.

**Impact**

Creating a model which assists with the creation of a continuous integration pipeline will help guide through that initial phase of trying to figure out what the priorities should be. A pipeline is a dynamic product that needs independent goals and this model will highlight what these goals are in relation to the software it is supporting.

By implementing this model a structured continuously improving pipeline will be created which improves time to market, highlights goals explicitly, creates a link between a companies goals and the goals of the pipeline.

**Problem statement**

*Develop a model and terminology for developing efficient, automated and continuously improving testing pipelines based on a real life case study.*

The paper centres around an *automated testing pipeline*. A testing pipeline is predetermined set of tests that are executed in a specific order to ensure the quality of the tested software. When the entire pipeline is automated it will not require user interaction to start.

Basing the model and terminology on a *real life case study* means that a practical implementation will be made which again will give experiences that will help form the model. The case study will be made in a company that is creating an operating system. This means there is a wide variety of testing requirements.

One of the goals is to create a *model*. This means creating a high level framework that conceptually explains what is needed when creating the
proposed automated testing pipeline. The completed model will contain
definitions of terms used, recipes for activities that help accomplish the
goals of the model and ways of assuring the quality of the work done. The
model will generalize so that it can be useful to other situations that differ
from the case study that it is based upon.

The other goals is to define terminology used when developing the
automated testing pipeline. To enable easier future communication around
this topic some key terms will have to be defined. These terms will
help describe the model so that both the terminology and model work
together.

The task is to develop a model and terminology. By developing these end
products the experiences gained during the case study part of the thesis
will be used to shape and create the model and terminology.

The pipeline that is to be designed needs to be efficient. Being efficient is
defined as performing or functioning in the best possible manner with the
least waste of time and effort. This means that the pipeline will have to
both perform functionally and without wasting unnecessary time.

The pipeline also needs to be automated. This means that initiating and
running the pipeline should not require any human intervention.

A continuously improving pipeline has a system in place for a cyclical
evaluation of its goals, methods and results. When using the model there
needs to be a way of using previous runs as input for making better
decisions.

Summary of contributions

The thesis helps any continuous integration pipeline with:

- Align goals of the project with the goals of the pipeline.
- Identifying what the explicit goals of the pipeline should be.
- Provide a framework for continuously improving the pipeline over
  long periods of time.

Thesis organization

Chapter 2 will provide background information on the problem domain of
the thesis. Chapter 3 will describe the approach that will be used to answer
the problem statement. There are two results chapters one about the case
study and the second about the model. Chapter 4 is the first, it provides an
account of the case study and the linear progression of the work performed.
Chapter 5 describes the design of the model, the desired impact and the
actual model itself. Chapter 6 analyses the model figures out whether or
not the model accomplishes what the problem statement specifies. Chapter
Chapter 7 discusses the thesis as a whole and provides a detailed picture of all the accomplishments, issues and possible improvements. Chapter 8 draws the final conclusion.
Chapter 2

Background

The background chapter goes further into detail regarding the information introduced in the introduction. The information introduced there is put further into context.

Technologies and concepts introduced later in the thesis are also introduced in the background chapter.

2.1 Historical aspect of DevOps and CI

2.1.1 Why DevOps became necessary

Developing software before the digital distribution platforms we have today was a much slower and predictable process. Two big factors were completely different with the old process:

1. Development and operations were two different entities in the organization. When development work was finished the code was handed over to the productions team who would support it throughout it’s lifetime.

2. The time between releases could range from months to years, which meant that pushing out new features always came at a long delay.

The primary factor that made it clear that something had to change was the need for handling changes at a faster pace. With the rise of the internet and the digital software distribution platforms there arose a need for being able to output changes and functionality in software at a much higher pace. If a competitor was able to offer functionality to consumers at a much shorter time interval they would have a massive advantage in the market. User expectations for what what functionality should be available are continually rising. Another important factor to consider is the number of security flaws that a company will surely need to be ready to patch. Security holes are discovered regularly and are expected to be fixed as
quickly as possible. The time from when a user requirement arises to when it is expected to be in the application meant that a development pipeline that prioritizes speed and efficiency was required. The clear divide that had been prevalent between development teams and operations needed to be torn down in order for the time to market to be shortened. When a product launches, there are seldom plans to upgrade or deprecate functionality, and support is ongoing. (Roche, 2013)

The test requirements for software today have also changed. Before the software requirements specified the target platform where the software was expected to run. Today there are two different scenarios:

1. There are a huge number of software and hardware platforms that are expected to run your application.

2. The software runs as a server application and is expected to scale dynamically to the current number of active users.

No matter what the new requirements are they require a larger amount of testing. For this testing to be efficient and realistic it has to be done in an environment that resembles the operations environment. This means that during the test process temporary environments must be set up. To ensure that the requirements of speed and efficiency are withheld this process has to be automated. Automating this part of the pipeline ensures a level of consistency both in the development process, but also in the quality of the finished product.

Combining the requirements for faster releases, continued support and more complicated test requirements means that a new way of working was necessary. This is why we saw the rise of DevOps, it has become the way to keep up with the current market and lower expected time to market.

2.1.2 How does a company move towards DevOps?

When a company adopts a DevOps mentality there are a few challenges that have to be overcome. To meet the goals of faster time to market a pipeline that automates the delivery of software has to be implemented. Utilizing a continuous delivery pipeline requires that the company improves in some key areas; they need to increase communication amongst stakeholders, implement automation and improve agility in designing, delivering and operating software products and services (Lwakatare, Kuvaja, & Oivo, 2015). According to a literature review performed (Erich, Amrit, & Daneva, 2014) the main concepts related to DevOps are

- Culture
- Automation
- Measurement
- Sharing
Ideally you would want to master all of these principles, but for every organization this is a process where every area needs to be practiced in order for it to be mastered. The move to adopt DevOps is a gradual process where it is best to first implement the easiest factors and reap the early benefits. For a project this could mean implementing automation at an early stage. With an automated approach it is much easier to ensure a higher quality of the product. Seeing the benefits of this automation would surely impact the company culture as well.

2.1.3 How to achieve higher quality and improved velocity?

The overall goal is to improve the quality of the software produced and the velocity of production. The two main factors for achieving this goal are to improve the human and technical dimensions. Both of these two dimensions need to be taken into consideration in order for the overall process to be at its best.

Human dimension

The human factor in software development is always unpredictable. This unpredictability is partly why an agile development structure has been the prevalent way of developing software in later years. With work sprints consisting of focused work on a specific feature the teams can stand back and evaluate their direction and progress at the end of the period. This gives the team a framework for handling an ever changing market and can work on the feature that is most needed at any given moment. Having the possibility of evaluating what short term objectives to strive for is a great tool for development teams. If a direction or set of features turn out to not lead to anything it is easy to reevaluate and change direction at any time. Having a culture where this is not only possible, but encouraged will lead to a more dynamic and adaptable team.

If the teams ability to adapt to changes is missing then any major restructuring or change could either take too long to implement or it could leave the product in an unfinished state.

Technical dimension

The technical dimension deals with all the technical factors that can shorten the time from the initial code commit to the code is in production. Shorten-
ing this time will also help improve the workflow for the developers as they can know if they are producing production ready code at an early stage. There are a number of technical tools that can help with this work. Using containers to contain and deploy code, configuration management to provision and configure servers and deployment tools to automate merging and deployment.

2.2 Unikernels

Recent years has seen the rise of data centers renting out computing performance for customers to host their virtual machines (VMs). The VMs are usually single purpose in the way that there is a dedicated VM running a web server or a database. When there is a need to scale, multiple VMs are booted to serve the load required. This means having more instances each running a separate operating system. Each of these operating systems have a lot of unused features from drivers for different hardware devices ranging from Gigabit network interfaces to sound cards and last century printers. (Bratterud, Walla, Engelstad, & Begnum, 2015) With a large number of these machines running the power demands are also increasing. According to a report made by greenpeace in 2014 the total power demand by cloud computing platforms is the 6th largest in the world when compared to all the major countries. (Cook, Dowdall, Pomerantz, & Wang, 2014) This means that power saving in the future should be a major concern as cloud computing is still growing.

One possible solution to this issue comes from unikernels. Unikernels are specialized OS kernels that are written in a high-level language and act as individual software components. (Madhavapeddy & Scott, 2013) The recent interest in unikernels comes from a desire to specialize the actual operating system that is used in the virtual machines. When the application running in the cloud is a single purpose piece of software there is no need for the image to contain unnecessary and unused code.

2.2.1 IncludeOS

One such unikernel is IncludeOS. IncludeOS is a single-tasking operating system designed for virtualized environments. (Bratterud et al., 2015) The three main goals of IncludOS are:

- **Extreme resource efficiency and footprint** Every instance of an IncludeOS virtual machine is created to perform only a single task. Only the parts required by this task are included as part of running binary. This reduces the size of the images and improves the network and memory performance.

- **Efficient deployment process** When writing a c++ application the process of making it run with IncludeOS means adding `# include <os>`.
The build process returns an image file that can be booted as any other virtual machine.

**Virtualization platform independence** The image can be run on a variety of different virtualization platforms such as openstack and qemu.

There are also multiple security benefits to a unikernel operating system. There is a minimal amount of functionality thus reducing the attack surface. There is complete separation between independent IncludeOS instances running, as they do not share any common resources.

Developing an operating system like IncludeOS comes with a set of challenges:

- There is a lot of functionality that can only be tested by utilizing integration testing. Not everything can be unit tested.
- Even when running integration testing there are so many moving pieces that making sure every part of the code is touched during testing is difficult.

IncludeOS is being developed as an open source project. This means there is a potential for engagement on a whole new level. With this involvement the need for a proper testing pipeline is even higher. It is difficult to enforce a certain type of testing

The market sector that IncludeOS is trying to appeal to is a crowded space. A potential market for IncludeOS is running single service web applications in the cloud. One of the first projects that have been released using IncludeOS is their web server framework named Acorn. (IncludeOS, 2016) This market have a set of demands for the services that are in use:

- Long term reliability
- Performance
- Ease of use
- Security

For IncludeOS to even be considered as an alternative it will need to display superior qualities in all of these fields. The trade-off for setting aside time to adopt a brand new technology must come with the assurance of significant improvements.

### 2.3 Continuous integration

When a software development project moves from a single developer to multiple there must be an agreed upon way of integrating the code together to form a cohesive functioning product. Especially when both members will be working on the same part of the code conflicts can arise where
one change can break functionality for the other person. Traditionally a step in the software development cycle would be to "integrate" the code to form a fully functioning product with no conflicts. When you expand a team beyond two people the complexity of this step could increase drastically.

This is where the concept of continuous integration comes in. Continuous integration can be defined as a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day (Fowler, 2006). Many continuous integration implementations rely on a high degree of automation as actively managing the integration would take too much time.

Using a version control system is highly regarded as the norm these days. Git and similar version control systems automate parts of the integration process by making sure that code is ready to be merged with existing code without conflict. (Vasilescu, Van Schuylenburg, Wulms, Serebrenik, & van den Brand, 2014) This is only the first part of the solution, the second part is the automated testing. When a new piece of code is ready for integration and the version control system deems it ready to merge an automated test suite can initiate unit tests and integration tests. Ideally you want as high a test coverage as possible, as this ensures that the code that gets merged is solid and does not break existing functionality.

2.3.1 Starting to use continuous integration

When a software company that develop using one of the slower traditional development cycles decide to take the step towards a more modern approach there is a normal pattern which most companies follow. These steps are referred to as the "stairway to heaven" (Olsson, Alahyari, & Bosch, 2012) and these steps are:

A. Traditional development: Characterized by slower development cycles, large teams and sequential processes.

B. Agile R&D organization: In order to move the organization over to a more agile approach the testing of agile methods is made in smaller steps. Faster development cycles are first made in smaller teams.

C. Continuous integration: When a company starts using continuous integration in their process they are making frequent changes that get integrated quickly. In order to facilitate this automated merging, building and testing is utilized. When the development team embraces this workflow the testing team must follow the same patterns.

D. Continuous deployment: This step means that software is released to the customer more frequently. This allows the developers to see if the changes they made introduce real value at an early stage.
E. R&D as an ‘experiment system’: The final step relies on a stream of constant feedback from the product which allows the R&D team to base future features on instant customer feedback.

Continuous integration fits into these steps right in the middle. Agile development has been a natural goal for software development teams for a long time, but making the next move and utilizing continuous integration is not as straight forward in a large organization. According to the study done when writing the "stairways to heaven" (Olsson et al., 2012) the large companies that were interviewed had some issues when moving towards continuous integration. The issues experienced were dependency on suppliers who now also need to embrace a similar development philosophy, a cultural shift among the employees, tools and the transparency of the new process and finally the need for fully implemented and automated tests. In order to move past these issues the companies interviewed found that building a solid test suite which can be run as often as possible was the most important step in order to shift the culture and show the potential in this new way of working.

2.3.2 Importance of CI

When implementing continuous integration into your project there are certain benefits. As seen in a case study performed in companies that implemented continuous integration (Ståhl & Bosch, n.d.). They reported improvements to

- Communication
- Developer productivity
- Troubleshooting
- Predictability

The reports of improvements in these areas were not universal. Some of the participants did not report positive improvements in all of these areas. The reasons for this disparity fell outside the scope of this case study, but they theorized it could be caused by individual perception, differences in culture and process between the development projects, or inherent differences in the products being developed?

2.3.3 Pipeline examples

Fully implemented pipeline

An example of a pipeline that includes various integration steps is described in the article explaining how a large betting company moved all of the software development projects over to a continuous integration pipeline. (Chen, 2015) The steps included in this pipeline are shown in
The code advances to the next step only if the current step is passed. Moving from one part of the pipeline to the next can either be automatic, or require a manual approval. As the code moves through the various steps the confidence in the code's quality increases. The final step in this pipeline is production, which means that the code is deemed ready for users to be exposed to it.

**Figure 2.1:** A example fully implemented pipeline. (Chen, 2015)

The various steps of this pipeline are all explained below:

**Code Commit** This stage is triggered when the developer wants the code tested. Depending on the version control system and the CI tool in use this step can be either automatically or manually triggered. The snapshot of the code at this stage is what will go through the entire pipeline. Usually that means the code at this stage will be merged into the master branch of the project.

**Build** Before performing any tests the project is built and unit tests are performed on it. The unit tests are usually quick to perform and expose any minor flaws in the code. The build process generates an artifact which the later tests can be performed against. This ensures that the tests are performed against the same compiled code. The code is built once and tested multiple times. Once you move to production you can be sure that the code in production is the exact same as the one that was put through all the testing.

**Acceptance test** This stage is mainly concerned with checking that the software meets all the users requirements. In the example provided an environment that is the exact same as the production environment is set up and the code has to function under these circumstances. This step must be adapted to include any specific user or customer requirements. If your product advertises compatibility with a specific platform this is the moment to make sure that works.

**Performance test** This step has tests set up that measure performance for the specific software. The performance tests should be run in the same environments for this test as for production. The results from the performance test will then be fed into a tool that monitors the performance. If the performance falls below what is expected the developers should be notified and the progress through the pipeline should be stopped.
**Manual test**  Most of these tests are fully automated, but there might be instances where the developer wishes to perform exploratory testing. If that is the case the pipeline could be halted at this point and the developer could be given access to the environment where the code is currently running. Passing this step would then require a manual confirmation from the developer.

**Production**  The final step is to go into production. What this entails is different for every product, but if you have full confidence in the pipeline this step could also be automated.

**Current IncludeOS pipeline**

The current implementation of a continuous integration system in the IncludeOS team is shown in figure 2.2.

![Diagram of the current implementation of a continuous integration system in IncludeOS](image)

Figure 2.2: The current implementation of a continuous integration system in IncludeOS

The steps in this revision of the IncludeOS pipeline perform the following tasks:

**Pull request/Code Commit**  Git and Github is used as the version control system. As IncludeOS is an open source project this makes interfacing with other collaborators easy to manage. Code commits are performed to the central IncludeOS repository as pull requests. Performing a pull requests checks for merge conflicts and triggers the testing.
Testing/Build  The first step is the build step. The merged code is pulled, built and unit tests are run. If all is successful additional integration tests are run.

Production  There is no actual code that is pushed into production, but the code is marked as suitable for merging in github.

2.4 Continuous integration tools

In the world of continuous integration there are a number of tools that are used to achieve improved workflow and help integrate all the different services.

2.4.1 Jenkins

Jenkins is one of the leading automation servers available (“Jenkins.io,” n.d.). It is a tool that helps automate and visualize the pipeline that the code goes through. This is performed by having a large library of plugins that support a wide variety of different languages and technologies.

In Jenkins it is normal to create jobs that are limited to running and testing a certain application. A job can perform a number of tasks, such as:

- Pulling code from version control.
- Running when triggered from a remote trigger, such as a new update to version control or a timer.
- Preparing a run environment for the upcoming tests.
- Building the code.
- Reporting test results in a graphical manner. The test results can be transmitted back to the version control system.

Due to the plugin architecture there are endless possibilities for what can be added, and the active development community ensure good support.

2.4.2 Openstack

OpenStack is a cloud operating system that controls large pools of compute, storage, and networking resources throughout a datacenter (“Openstack.org,” 2017). To the end user Openstack is a cloud platform that offers on demand virtual machines that are easy to provision, configure and deploy.

The configuration of an Openstack cloud platform falls outside the scope of this thesis, but what it offers is:
• Fast and flexible provisioning of virtual machines.
• Easy installation of a number of different operating systems. The machines can be configured with ssh access.
• Varying flavors of hardware configurations. This allows for some machines to have more powerful hardware depending on the tasks they are expected to perform.
• Deleting and rebuilding machines is fast and simple. This ensures a fresh machine for each use.

2.4.3 Git and Github

Utilizing a proper version control tool enables proper tracking of development and allows for much easier cooperation. This is done through checking out code from a main repository where the master copy of the code is contained. If two people make separate changes to the same code then there will be a merge conflict which must be resolved before the code is submitted back to the master repository.

By using a version control system such as Git which is hosted on Github a lot of other possibilities arise. Github offers webhooks which are triggers that can send out alerts that there have been made changes to a certain repository. This allows for easy integration with a continuous integration pipeline.

Recent updates to Github allow for easy visualization of test results. When a user wants to submit code to a repository a pull request can be submitted which lets the maintainer of the repository review the changes before merging. The pull request system allows for external services to run tests and display their results. This can be utilized to automatically test any pull requests made and then either approve or fail the changes. Having the test results show up in the same interface as would be used to merge the code is very useful and means the developers will only have to use one interface.

2.5 Relevant research

2.5.1 Including Performance Benchmarks into Continuous Integration to Enable DevOps

A study (Waller, Ehmke, & Hasselbring, 2015) was made into what gains could be made by introducing performance benchmarking as a nightly step in the development of the monitoring tool named Kieker. The paper uses a benchmarking tool called MooBench created to measure the overhead for various monitoring frameworks (Waller & Hasselbring, 2013). The study uses the continuous integration tool called Jenkins to display the results
from a nightly benchmark of the performance. This allows the developers to see impacts on the performance within a day of committing faulty code. If the benchmark was to run less frequently there could potentially be too many commits that could cause the error, so a nightly execution fit this project well.

In order to verify that the benchmark would indeed pick up on faulty code they put an older version of the software through the benchmarking code. This resulted in a clear visual representation of the point in time where there was a loss in performance. The results were presented as a bar graph where each bar represented the performance results from a single day. This helps narrow it down to a commit from that day which caused the issue.

Prior to having this system many of the issues were never exposed until the software was put into production, the advantages of eliminating such issues at an early point are huge.

The paper concludes with some advice. They suggest implementing performance benchmarking at an early stage in the development process. Ideally they would have wanted an improved system of notifying the developers of issues exposed by the benchmarking process, as the current system needed a manual check of the results. Lastly the tools available for showing results in Jenkins are a little lacking.

2.5.2 Performance-oriented DevOps: A Research Agenda

This report (Brunnert et al., 2015) recognizes that DevOps traditionally has an advantage in helping the time to market for new functionality. DevOps principles have led to new features and bug fixes to be part of new releases. This research focuses on tools, activities and processes that help ensure the performance of this software.

Common metrics for performance are typically response times, throughput and resource utilization. In order to use these goals enterprises have set upper and lower targets for these metrics. These goals have then been individually translated to dev and to ops. By having these goals independent of each other the full transition to DevOps was not completed. The research tries to tie together these two performance testing methodologies.
Chapter 3

Approach

The first part of this thesis involved doing research which was the basis of the background chapter. Before doing this research the initial intended approach was to evaluate the current literature, create a model and terminology and use that to create an appropriate automated test pipeline. This way of constructing a project is a good approach in theory, but what became apparent was that the complexity of this field made gaining sufficient knowledge through theory difficult. Basing the model purely on theory proved difficult, and exposed the need for a practical approach when creating the model. This led to a slight change in the process, instead of basing the model on theory, the model will mostly be based on practical work performed over an extended period of time. The practical experience is gained through conducting a case study where a pipeline can be fully implemented.

When choosing a case study as the basis of the model this comes at a risk. A case study involves a lot of work, and when beginning this work there is no guarantee that it will lead to a meaningful result. After putting down hours of effort you could be left with the conclusion that it did not succeed, and building a model on such a conclusion is not possible. This risk factor means that at every stage of the case study there should be a focus on trying to see where newly gained practical knowledge can fit into a theoretical model. If it becomes apparent that this will be a difficult task then the project should be reevaluated.

Given that the case study does lead to meaningful experiences it is then possible to develop these experiences into a model. The practical knowledge needs to be generalized in order to create a framework that is useful to someone about to embark on a similar project. By defining key terminology, identifying activities and creating a way to ensure high quality the model will provide a useful tool for future automated pipeline projects.

There is a clear goal of what properties the end product should live up to. Such properties as automated and effective are part of the problem
This means that the end product must be held to this standard. A way of ensuring that the model will fulfill these properties needs to be planned. As part of the project an analysis will need to be conducted where these key properties are ensured.

3.1 Defining the case study

IncludeOS is a unikernel in active development. A team of 5 programmers are working on developing an operating system. As the project grows the need for a complete testing pipeline is becoming apparent. When developing an operating system there are a number of different tests that are necessary. There are isolated parts of the operating system that needs to be tested, but also full sections that integrate different functionality need to be tested. One of the goals of a unikernel is to eliminate as much overhead as possible and optimize for high performance. This means that performance testing is also required.

The team has a general approach towards writing both unit and integration tests. Unit tests are focused on testing the smallest possible functions and stand alone parts of the operating system. In this project the writing of unit tests falls upon the developers. The execution of the unit tests a step that will benefit from automation. Automating the unit tests will fit into the testing pipeline, and its main purpose is to quickly expose any errors. The main part that this project will focus on is the integration tests that attempt to test more complete operating system functionality. This part quickly becomes more complex than the unit tests as it requires writing specific test services and a system for booting services and checking for test success or failure. Before the start of this project there was no system for automating the execution of the integration tests. There was no automated pipeline defined that ensured code quality. In order to implement the efficient automated pipeline that is needed the case study will focus on the following processes:

- **Code life cycle** The stages that the code getting tested go through.

- **Infrastructure** What infrastructure is used to perform the tests.

- **Result visualization** How the results are presented to the developer or user.

- **Testing order** What order the tests are performed in.

- **Optimization** Any steps taken towards optimizing the performance of the pipeline.

Constructing an automated testing pipeline is such a project independent task that having real world experience is a valuable background for creating a model. The case study will describe the process and design choices made when implementing the automated pipeline for the IncludeOS project. This leads to a number of specific tasks that have to be
completed during the case study. The flow and relationship of the tasks are shown in figure 3.1. The optimization task can show that it is necessary to go back and redo a previous part of the project.

**Design code life cycle**  This first step is focused on the overarching design of the testing pipeline. The code life cycle is key when designing for high efficiency. In order for developers to get feedback as quickly as possible the code should be set up to report failure as early as possible. At this point it is important to design how the tests are to be executed in the pipeline. The execution of the tests is key when planning the infrastructure.

**Setup the infrastructure**  A central part of the testing pipeline is having an infrastructure that is able to facilitate the testing requirements. This means that the design of the infrastructure is highly reliant on the overall design of the test pipeline. If the pipeline relies a large number of servers to run on the infrastructure must be able to facilitate this. As the project grows the infrastructure should also be easily adaptable. Having a scalable infrastructure will make future
growth much easier to face.

**Design and develop the testing tool** At this point the overall design of the pipeline is decided upon and the possibilities provided by the infrastructure are known. This step focuses on creating the tool that will be used for executing the tests and utilizing the infrastructure. The tool must keep the focus on efficiency and future expansion. It is difficult to predict what future requirements will look like, but designing the tool in a modular fashion will ease future work.

**Integrate with front end tool** The testing tool will have to have an interface that the developers will interact with. At this point the test pipeline should be able to perform all the required tests. This step should focus on both easy access to information for developers and quick feedback.

**Perform optimisation steps** After a fully functioning automatic testing pipeline is completed this final task will focus on efficiency. As efficiency is the factor that can provide the company an edge compared to others it is a step that will provide value to the company as a whole. The optimisation step can require changes to be made to earlier tasks as well, which could lead to the whole pipeline needing to reflect these changes.

Documenting such a case study will be based on anecdotal evidence. That means it will be based on anecdotes which is defined as a short account of a particular incident or event. (“Dictionary.com Unabridged,” 2017) The evolution of the solution will be told in a way that explains the process. In addition to focusing on the evolution of the solution there will be technical discussions of the solutions implemented and developed. Due to the incremental nature of working with development and implementation of a pipeline there might arise instances where a major design decision leads to a change in the pipeline. If such a major shift is to happen then it is important for this to be emphasised. These revelations are valuable as they highlight a point in time where there was a realisation, and the thoughts around the process changed. These moments where the mindset shifts are what the model should attempt to convey without having to go through a long process of doing work inefficiently. Having a focus on highlighting these instances while performing the case study will help in creating a better model.

The documentation of the case study should also include a lot of technical discussions. Technical choices performed during the implementation of the pipeline helps form the basis of the model. When reading about how the creation of the model happened it will be easier to understand when having all the technical facts available.
3.2 Planning the model

The model is closely linked to the result of the case study. All the activities performed in the case study help form the terminology and the components of the model. As a first step it is therefore natural to discuss the case study and figure out if the goals for it were met. Reflecting around the case study will help form the experiences into a framework which will help form the model. The framework will be constructed in a way where the terminology and activities will fit into themes. This will help categorize the experiences from the case study and create a more generalized and easily accessible model. The relationship between the case study and the model can be seen in figure 3.2

![Diagram: Relationship between the case study and the model](image)

When crafting a model the focus will be on three main deliverables. In order to have a common ground for discussing the terms in the model the first step is to define the terminology used in the model. When using the model for creating an automated release pipeline a set of activities will be outlined. The final part is a number of steps that will assist in assuring a high level of quality in the pipeline produced. Ideally the steps that make up the quality assurance should be kept in mind during the entire design process. This will ensure that the factors outlined in the quality assurance step are factored in as early as possible. A workflow could then look like in figure 3.3 on the following page.
3.2.1 Terminology

Defining terminology is the first step of creating the model. The reason for defining a terminology is to ease any future communication around the subject. All discussion will be made on a common ground and will help avoid any misunderstandings that may arise from different understanding of terms. Part of defining a terminology is also to define the relationship between terms. The model should therefore make these terms extra clear and utilize venn diagrams or flow diagrams to visualize the relationships between them.

3.2.2 Activities

When using a model to implement a solution there is an expectancy to have a recipe to follow. Implementing an automatic testing pipeline is a very project specific affair so the process will be different for every implementation. There will be a lot of broader guidelines that can be followed so this model will focus on a number of activities. These activities will be independent, but have a relationship between them. This means that any project can utilize the activities that are needed and adapt the other activities to fit their project specific needs.
3.2.3 Quality assurance

The final part of the model is a set of steps that can be gone through to ensure that the product generated has a high quality. This could be routines to go through, processes to check or questions that help reflect on the pipeline that has been implemented. If the quality assurance step is good it will help expose parts of the pipeline that are lacking. As with any process it will help to make changes as early as possible. This means that the steps mentioned in this quality assurance part should be reflected upon at all stages in the process.

3.3 Analyzing the end result

3.3.1 Properties for measuring success

Before knowing if the end result is a success or not it is important to know what defines a successful project. In the problem statement there was a few properties that were used to describe the end result:

Automated Going through the pipeline without any user intervention.

Efficient Functioning in the best possible manner without wasting time and effort.

Continuously improving Has a system in place which will ensure that it will work over time and increase the quality.

When analyzing the end result these two properties will have to be assessed. Automated and efficient are what can be called the explicit properties that help define the project. Explicit properties in this case are the properties that are the primary goal of the thesis. They are not given properties that can be taken for granted, but instead try to capture something new and unique.

In addition to these explicit factors there are a number of implicit factors that are expected from the project. Implicit factors are the factors that are expected to be there, but when analyzing if the project has been a success it helps to explain exactly what is expected. With this project the implicit properties that the model has to live up to are:

Usable If the model does not make any sense, or does not actually work in any other scenario the project has not been a success.

Length A model that is too excessive and too detailed will be difficult to adapt to another scenario. As every software project has differences it needs to be generalized enough to fit a number of use cases. A model that is too large might also deter new adopters due to a high barrier of entry.
Action items When using a model and following its workflow having a number of activities that are easy to follow is important. If the model is all text and it is up to the reader to decipher and generate work tasks the model becomes harder to utilize. In order to get users to think about implementation and specific work tasks as easily as possible having action items is a good start.

Goals In the same way as having action items will help users get started with the implementation having goals will help users know when their implementation is completed. It is a motivating process to complete goals and see that working with the model is providing feedback in terms of goal completion.

Valuable The model aims to ease the transition into implementing an automated testing pipeline. This means that using the model should give new insight or give the user real ideas for implementing the pipeline into their workflow. If it does not accomplish this goal then it is not a very useful model.

Accessible In order for the model to be used it has to be easy to find and use. If it is buried deep in the thesis paper it will not be easily accessible for future projects. That means that the model part of the paper has to be accessible by it self. It becomes a deliverable and needs to be planned.

3.3.2 Assessing properties

When the project is completed and the analysis needs to be performed there needs to be a method of uncovering if the targeted properties of the model are present or not. When completing an assessment of the completed model it will become apparent if the goals were reached. This means that the assessment needs to focus on the properties defined above and measure if they were reached or not. There are a number of ways of making this assessment, some of which are more feasible given the time restraints and nature of the task. The overall goal of the assessment is to figure out whether or not the model will help someone starting the work of designing and developing an efficient and automated testing pipeline.

New literature study After completing the model it could be compared to other similar models out there. Part of the issue here is the lack of literature on this specific subject, and at the end of this thesis the number of new papers on the subject can not be expected to have increased significantly.

Self assessment When the model has been completed a self assessment step would attempt to answer if the model in question would have benefited the job of working on the case study. This way of assessing the model form a sort of looping dependency, where the model is based on the case study, so the usefulness of it will be hard to deny.
By self-assessing the usefulness of the model it requires you to step back to see if it will be useful in another scenario than the one it was created from. This requires some theory crafting where other proposed scenarios are theorized and the usefulness of the model can then be assessed.

**Apply model to project** An ideal test of the model would be for an unrelated new project to attempt to use the model. This is the exact use that the model is created for and would give highly realistic feedback if the targeted properties are met or not. The downside to this approach is the time it would take to complete. It would first of all mean that a separate project would have to be available, and there would have to be a need for an automated testing pipeline. Given the time restraints of this thesis this is not deemed a feasible way of assessing the goals.

**Comparative study of projects** This way seeing the effectiveness of a model has a number of prerequisites. This assessment bases itself upon comparing projects that have used the model to projects that have not. This means that there must exist data on a number of projects that have used the model. When this data is available a whole study will have to be made where the projects are compared. Again this way of assessing the goals is not possible due to time restraints.

**Expert opinion** Completing an entire new project to assess the model takes too much time, which means that an alternative needs to be considered. Showing the completed model to a group of experts and asking for their opinion is a valid approach. This could either be targeted in the way that a set number of individuals are asked to participate and give their feedback, or a more open approach where the model is posted to an open forum and anyone could give their feedback. The first option guarantees a feedback and there is a clear opportunity of directing the conversation by providing clear questions. The second option could generate a larger number of responses, but the quality of the responses might be harder to work with.

From these options it is clear that seeking the feedback from a set of experts is the best approach. In order to be prepared for this session there are a couple of prerequisites:

A. The model has to be available in an easily accessible location. This reflects the implicit property that specifies that the model has to be accessible. When interviewing an expert it cannot be expected for them to read the entire thesis to comment on the model. A concise and stand-alone model will be easier for them to read, process and comment on.

B. The meeting has to be planned. The expert has to be contacted and meetings scheduled. In addition the meetings need to be planned in
order to maximize the usefulness in the analysis chapter. Creating a number of questions will have to be done prior to these meetings.
Chapter 4

Results 1: Case study: A solution for IncludeOS

4.1 State of the code

Before starting to improve the code it is important to know exactly what the origin looked like. There was an automated testing system in place when I got involved with the project, but the efficiency in the testing was not the best. There was automation in the system, but it was on a timer and not triggered by developer events. Interfacing the testing results with the developers was also present, but not in the most accessible way.

At this point IncludeOS was at an early stage and the testing was starting to get more focus. The stages of the test process are shown in figure 4.1 on the next page. This figure shows the parts of the process that are present at the beginning of the project. Visualizing and discussing the start of the project is important so that the incremental improvements made are put in context.

The overall system for testing at this stage is based on a set of scripts that perform the testing. These are split into testing the compilation of the operating system and other functionality. In order to complete these tests an infrastructure had to be in place. At this stage the infrastructure consists of a physical server that runs the test scripts. Due to the scripts being run at night there was not a very high demand for large infrastructure to handle several tests at once. When running nightly the single server infrastructure was enough and served the project well.

The second piece of the process was the testing stage. When creating an operating system it was important to focus on the build stage. Having an automated test of the build process for IncludeOS has always been crucial. Another early requirement in the testing was to evaluate the networking component of IncludeOS. As networking is one of the key functionalities of any operating system today having tests for it was necessary from the
beginning. These tests were performed by utilizing the kvm hypervisor present on the physical server performing the tests. The two test processes performed were:

**OS build** Building and installing the operating system consists of:

A. Build library dependencies used by IncludeOS.

B. Build IncludeOS

C. Install IncludeOS

D. Install requirements for actually running IncludeOS. Such as a hypervisor and a networking bridge.

All of these steps can produce an error. The library dependencies can be updated and cause incompatibilities, IncludeOS by itself could cause compile errors and changes to the other requirements could cause functionality to break. With such a large number of possible causes of failure the importance of testing becomes apparent.

**Networking** A series of simple networking tests towards the running IncludeOS instance. The networking tests run at this point were very barebones and primarily ensure that the instance is able to respond to ping and arping. The commands used at this point are shown in listing 1.

```bash
1 /usr/sbin/arping -c 1 $IP
2 /bin/ping -W $timeOut -c 1 $IP
```

Listing 1: Commands used for networking test
The final step is to have a system for visualizing the results of the tests to the developer. At this point the feedback is done by posting the results of the test to the wiki that is part of the github repository. This means that the test results are always available, and no there are no requirements of self hosting the results.

Finally there is an emphasis on automation. At this early stage the automation consisted of running the tests specified at a set schedule every night. That meant that the results from the test if they were to fail would be ready the next morning. If there were not made any changes to the code during the day then the nightly builds would not be run as that would be a waste of resources.

### 4.2 Start of automation

The initial motivation for this project was to improve the visualization aspect of the pipeline. Having the results of the tests be posted to the wiki was functional, but not very elegant. There was a desire to start using a continuous integration tool that would interface the test results to the developers in a better more elegant way. The plan was to start using the continuous integration tool Jenkins (“Jenkins.io,” n.d.) that would act as the platform for both running tests and showing the results.

In addition to a new automation front end a change in the infrastructure was needed. This was motivated by a need for having the possibility to install the operating system on a fresh instance for every test run.

---

**Key lesson**

Ensure that the testing infrastructure/environment is not “spoiled” by previous builds. This means making sure that:

- Dependencies previously installed do not prevent future builds from running.
- Previous build and compile stages don’t prevent a full new build from getting run.
- Software left from previous builds or tests should be fully removed before running again.

This was important because:

- The installation script should properly install and configure dependencies on a clean system. Making sure this installation procedure works should be completed regularly and automatically.
- Previous failed builds and tests could leave behind artifacts that could potentially affect the next build and test. Ideally all files and resources used should be known so that they can be removed when a build has been completed, but in a real world scenario this could get difficult. Dependencies can make changes that are difficult to track.
The worst case scenario that this is trying to prevent is the case shown in figure 4.2. In this instance there is a test that is run on three machines:

A. A developer laptop who has written the code
B. The testing server
C. On a freshly installed independent machine

On the first run the test passes the first run of the tests. After running the tests the environment is kept as it is on the developer laptop, but removed on both the test server and on the freshly installed machine. When running another test later it passes on the developer laptop, while failing on the two other platforms. In this case there is a part of the environment that is passing the test on the laptop, while failing in the other instances. The most important place for this build to pass is on the freshly installed machine as it is the only place that can be guaranteed to be the same every time.

The opposite case shown in figure 4.3 on the facing page is also not ideal. In this instance the second run fails on the developer laptop with the contaminated environment. It is unreasonable to expect developers to wipe their own local environments between tests, so this is where the testing infrastructure shows its importance. The developers should have enough confidence in the results shown by the testing system to trust the results.

These lessons learnt go into designing the first version of the testing pipeline. This updated pipeline is shown in figure 4.4 on page 32. This shows how the infrastructure stage now uses openstack. The test stage has not been altered very much as the focus in this first stage was to get the infrastructure and visualization aspects under control. The visualization
has been improved to utilize jenkins as a frontend. Using jenkins has also meant that improvements could be made to the automation. Instead of running builds on a set nightly schedule this stage introduces builds triggered by pull requests to the github repository.

4.2.1 Openstack as infrastructure

When deciding that there was a need for a cloud infrastructure this was motivated by:

- A need for easily reproducible fresh environments.
- An easily expandable infrastructure.
- Easy access to a variety of operating systems.

The first goal to reach was to utilize the fresh environments to install and test the build process of IncludeOS. At this point in time there are two ways of installing the operating system. When compiling an IncludeOS image it only packs in the libraries and functionality it needs to run. In order to run it has a lot of dependencies on external standard libraries. These need to be available upon compilation, and there are two ways of acquiring these dependencies.

- They can be compiled from their sources and installed.
- Or the precompiled libraries from a previous compilation can be downloaded and used.

Both of these ways of installing are initially tested. First of all the precompiled bundle is tested, the dependencies are downloaded, extracted
and used when compiling IncludeOS. The second way of installing by compiling the whole dependency stack is also tested.

Using Openstack at this stage means that we are sure that the installation procedure works on a clean and fresh version of the operating system. In order to facilitate the spawning and configuration of these virtual machines used for the testing the configuration tool MLN (“MLN - Manage Large Networks,” n.d.) was used. This allowed for quick spawning of multiple virtual machines with predetermined configurations. The workflow for the infrastructure is shown in figure 4.5 on the facing page. The steps gone through were the following:

1. The Jenkins build job is triggered and requests a new virtual machine from the Openstack master. The Openstack master is the machine that has the credentials and interfaces with Openstack.

2. To spawn the machine MLN is used. The configuration provided by MLN is used to spawn a machine with the needed properties and ssh keys.

3. The Openstack server spawns a new VM.

4. The new VM is now ready to be used by the Jenkins build job.

### 4.2.2 Implementing Jenkins

The primary motivation for this first step is to improve the visualization of the test results to the developers. Jenkins was chosen due to its open source nature, active plugin community and free price. When installing and configuring Jenkins it was important to have the running of the tests
in mind. As the tests would need to have full access to the openstack resources the Jenkins master was installed in the same private cloud. Figure 4.6 shows how the different virtual machines fit together in the private cloud.

When figuring out how to create these tests an important fact was discovered. The goal of the testing pipeline is to verify that the code recently checked into the Github repository is functional. The test that is getting created to verify this functionality utilizes Openstack, MLN and Jenkins. As it stands at this point it feels wrong to include a test in the IncludeOS repository that has dependencies like these. The solution at this point is to have a separate repository where these tests are stored. This does not feel like a complete solution as it means two repositories have to be maintained, but at the moment it is the solution that is opted for. To make sure that commits from both repositories are pulled when executing a test both need to be considered. When using Jenkins it will detect changes made to one repository and trigger build jobs based on that. This means a
manual step of pulling the test code needs to be implemented.

**Key lesson**

When creating tests that have dependencies and functionality that differ widely from the core functionality of the repository these tests could be kept in a separate repository. This is to avoid clutter in the main repository.

The technical build processes that Jenkins was tasked with doing are triggering based on a new pull request and building the code.

**Pull request triggering**

One of the key goals was to improve the duration between code commits and tests getting run. It was decided on using pull requests to the main IncludeOS repository as the trigger for tests to begin. The alternatives considered were:

- **Nightly builds** This was the previous solution. Very long feedback cycle. The positives sides are that the time to complete a test is not that important as there are hours before the results are seen.

- **Timed builds** This is a viable solution, the timer could be set to every hour or even every minute. Would generate some extra traffic and there would still be a delay depending on if the commit hits the refresh cycle of the polling system.

- **Commits to main repository** Builds could be triggered when commits were pushed to the main repository. This means that possibly faulty code could be present in the main repository. This could be solved by utilizing branches, but as the development team grows this does not feel like a solution that scales well.

- **Commits to developer repository** To solve the problem of scale the individual developers repositories could be tested. The problem with this approach is that it is difficult to track all the places where testing is happening. All the branches will get tracked through Jenkins, but there might be a lot of noise. In addition all new developers would need a new build job set up for them.

- **Pull request trigger** This enables tests to be run when someone deems their code ready for merging into the main repository. The trigger will also ensure an immediate start to the testing process.

To enable Jenkins to trigger from a pull request the plugin called GitHub Pull Request Builder (“GitHub Pull Request Builder,” n.d.) was used. This allowed for all pull requests submitted to cause a test to get triggered immediately. Figure 4.7 on the facing page shows the processes that happen when a pull request triggers the testing.
Build step

The final part of implementing Jenkins was to make sure it built IncludeOS properly. What was important with this process was to rely as much as possible on the build scripts that are already part of the IncludeOS project. These build scripts are what the users will be using, so that is the ones that should be tested. The work that Jenkins needs to do to facilitate the build job consists of:

A. Download repository.
B. Generate a workspace and environment for running the code.
C. Run the install and build scripts.
D. Run the tests that are required.

4.2.3 What more was required?

After having implemented a new infrastructure, a new visualization system and an improved trigger for automation it is time to look towards the next step. The last piece of the puzzle that has not been improved is the testing stage. This step is at the moment quite shallow, the build step is only on one specific OS architecture, and the functional testing is only considering the networking. There is a clear room for improvement both in the architectures that are considered, and the scope of the tests getting run.

4.3 Expanding the testing stage

When expanding the testing stage there is a clear split in what is to be tested. On one part there is the operating system used to compile and
Figure 4.8: Expanding the testing stage with multiple operating systems and multiple test categories

run IncludeOS. Developers today use a wide variety of Linux distributions and having support for the most popular ones will only help with user adoption. It is therefore important to test the build stage on a wider variety of operating systems, and expose breaking compatibility as early as possible.

The second area to be expanded is in the types of test to be run. Earlier only superficial networking tests were performed. These only tested if it was possible to contact the IncludeOS image through ping and arping. The networking stack by itself has a lot more advanced functionality that should get tested. In addition there are other areas of functionality in an operating system that needs testing.

- Filesystem
- Hardware
- Standard libraries
- Kernel

So the focus of this part for this part of the project was in expanding the scope of the testing. The state of the pipeline can be seen in figure 4.8.

4.3.1 Testing on different operating systems

The motivation behind testing the compilation step on different operating systems was to support the variety of operating systems that developers all over the world use. Showing that a number of different operating systems...
are supported will certainly help with the adoption of the product. When listing that a certain operating system is supported it is also necessary to actually have an automated test that verifies that this support is not broken by a future update. Listing that a feature or operating system is supported and then later figuring out that it was indeed not functioning as expected will potentially harm any user who tries the product.

**Key lesson**

*All listed functionality should be tested before every release. Features could also be listed as experimental until there is full confidence in its state.*

For this project there was a goal of adapting the build stage to be compatible with a number of different operating systems. To verify this compatibility the two tests that have been previously run were also ready to run on the different operating systems. In Jenkins this sort of test setup would allow the results to be shown in a build matrix as shown in figure 4.9.

![Figure 4.9: An example build matrix for different operating systems](image)

When expanding to multiple operating systems there were a few challenges that became apparent:

**Installing dependencies** Handling the installation of dependencies needs to be done in different ways. When writing a script which installs the needed dependencies for the specific platform the commands are going to be different for another operating system. There are platform independent tools for installing packages and maintaining environments like puppet and chef, but for a simple installation script these tools feel like they are just going to add bloat to the installation process.

**Different environments** When the environments that are supposed to be supported are so different that they require active adaptation to function the work of supporting different operating systems becomes...
more strenuous. For instance there is no implemented support for creating images with grub on mac. Grub must be compiled from source in order to be used on a mac.

**Cloud image support** In order to ease the testing process having cloud images that assist in easily creating fresh images for testing is important. There is a high availability for these images for Unix operating systems. When attempting to test on the Mac OS X operating system there are no cloud images available. This is due to Apple having a strict policy on the hardware that their operating system runs on. In order to test on OS X there is a demand for Apple hardware that can host these tests.

---

### Key lesson

*When testing in different operating systems there needs to be a system in place for handling the operating system specific commands. Especially for handling dependencies used in the testing process.*

---

In order to minimize the footprint in the code for the operating system support there is only varying code in the scripts used for installing dependencies. Listing 2 shows that it is necessary to identify the system and the release in order to verify support. This system can be expanded to work across multiple versions of certain releases as well. When the system and release have been identified there is a separate dependency installation script that handles the actual installation.

```bash
SYSTEM='uname -s'  # Linux/Darwin (Mac OS)
RELEASE='grep -m 1 "ID=" /etc/os-release'  # Ubuntu/Debian/Linuxmint
```

Listing 2: Commands for finding system and release

Listing 3 on the facing page shows the only place where there are different commands used for installing dependencies. By limiting these types of code where there are different commands used the maintainability and ease of use will increase.

### 4.3.2 Increasing the amount of tests

The networking tests was the only test used in the beginning. It was becoming clear that a lot more tests were needed. Other functionality in the operating system also needed to get tested and at this point this work was getting done. The process of increasing both the number of tests and the scope was showing itself as a process that the entire development team needed to be involved in.

At this point it was becoming apparent that development process had to take a step in the direction of test driven development. The developer in
charge of a new functionality has by far the best understanding of its usage and should be in charge of developing a test procedure for it. For tests that require certain infrastructure or run environments this needs to happen in cooperation with the operations staff.

Key lesson

*The tests benefited greatly from being created by the developer behind the functionality it is testing.*

The first tests created at this time mainly fit the same structure as the networking test. Services were created that included the desired functionality, and custom scripts were written alongside them that tested that specific scenario.

### 4.3.3 A mess of tests

With this increase in the testing scope and complexity it was becoming clear that the tools required for testing had to evolve.

The first type of test that had been used for a while was the networking test. In order to run this test the procedure was:

A. Create an IncludeOS service with networking functionality.

B. Compile service and generate a bootable image.

C. Launch this service either on a local hypervisor or on a cloud platform.

D. Launch script which performs the networking tests.

E. If the tests receive the expected responses the test is a success.
This is a pretty simple setup, and does not require a lot of interaction between the service and the test on the outside.

Where this does start to get complex in when you want tests and services that:

- Interact more than at the start and end of the test.
- Branch depending on certain situations throughout the test.
- Require specific tools that interface with the test, such as dhcp or dns services.

Through the custom scripts that were created a some of these functions were started, but it was becoming apparent that this was spinning out of control. As more and more tests were created two very obvious issues were appearing:

A. The more complex functions that were needed needed to be centralized as they were now becoming fragmented and code was getting repeated in different scripts.

B. The number of tests was increasing in such a way that keeping control of all the tests was becoming difficult. As shown in figure 4.10 on the next page at one point you will suffer from a lack of overview. In this case with Jenkins the number of tests to show in the pull requests was getting out of control.

In order to remedy this situation it was becoming clear that a tool that could organize the tests and provide more advanced options was necessary.

Key lesson

Plan for an ever increasing number of tests and make sure the visualization of said tests can handle the larger number of tests.
4.4 Creating a dedicated test tool

When expanding into a larger amount of tests the apparent problem is to keep everything organized. Using pull request triggering allows one pull request to trigger a number of tests to start at the same time. If every single test was a separate process then the pull request would quickly be bogged down with a very large number of tests. This led to the need for a tool that could categorize the tests so that for instance all tests that were networking related could be run at the same time.

The second need that the testing tool would fulfill was to handle more complex test demands. The testing tool would provide a common framework for how testing was to be done.

A few other changes were made to the testing pipeline at this stage. These can also be seen in figure 4.11 on the following page.

A. There will not be spawned new virtual machines for every single test run. This is due to the way Jenkins handles ssh hosts and the time it took to launch new virtual machine. One of the main time wasters was installing all dependencies for every test run. With persistent test machines the tests run through jenkins are confined to separate folders which are wiped after each test run. If the operating system itself can be kept unchanged this will be a satisfactory solution.

B. Instead of showing the individual test results through pull requests it is now changed to show the status of the various categories introduced. If there is a failure within a category there is a link provided which will show in detail which test that failed.

When testing an operating system it was known at an early stage that the tests were potentially going to be quite complex. In this project the better
solution which was to create a test tool that interfaces with the tests was not apparent from the beginning. Getting to this revelation was an incremental process, and only due to the implementations that came before it did this sort of solution become apparent for this specific project.

**Key lesson**

Try to figure out as early as possible what features a potential testing tool will require. Building the tool with expandability in mind will ease any future ideas and considerations that are made.

### 4.4.1 Categorizing tests

The first issue was with the number of tests steadily increasing. The process for running tests at an earlier stage is shown in figure 4.12 on the next page. This shows that the tests were placed in separate folders and the test system called on the tests that were part of that certain category of test. This type of system demanded a lot of manual intervention in order to update the running tests in Jenkins. For every test that was added it needed to be added to the test job in Jenkins. It was becoming clear that some sort of system that could help categorize the tests was needed.

The solution was to create a tool that would help with the categorization. The general layout of the solution is shown in figure 4.13 on page 44. The solution was to create a tool named testrunner which is called by the triggered test and specifies which type of tests it wants to run. The tests in the folders are now given attributes which helps with the categorization.
The system started out with simply being able to group together tests that fall under the same general field. In this example there are two categories shown, networking and filesystem tests.

### 4.4.2 Complex test demands

In addition to the demand for sorting through the tests there was a demand for a centralized structure which would run the tests. Some of the new demands were:

- Requirement for a tool which could build any service and launch it on the local qemu hypervisor. This would then take into account specific demands such as memory requirements, network interfaces, mac addresses or cpu configurations.
- Launch multiple virtual machines simultaneously to test communication between virtual machines.
- Install extra support tools and dependencies.
- Interact with the running service and perform external commands.
Figure 4.13: By introducing attributes to the tests they were much easier to control

4.5 Expanding and optimizing

The final chapter of the case study describes some final optimizations that were done to smaller parts of the pipeline. These optimizations were made in what could be described as a quality assurance loop. The main product was created and completed and after reviewing the operation these improvements were implemented. One improvement was to create a new test flag which marked a test as intrusive. Another time saver was to centralize the build process to a single instance and distribute the compiled code to test executors.

Finally there is a short summary of some future improvements that could potentially be implemented over time.

4.5.1 Intrusive tests

One of the tests that were created by the IncludeOS team required that an IncludeOS instance functioned as a router. The scenario is shown in figure 4.14 on the next page. The IncludeOS router is connected to the two networks, and the issue in this scenario is that the host machine also needs to be connected to two different networks. The host machine must also be able to transmit a packet through network 1 and make sure it receives a reply through network 2.

The solution in this case was to create what are called network namespaces. These are basically logical networks that can be connected through with
the shell on a linux machine. This allowed the router to be tested all while running on a single ubuntu machine.

This issue with this test was the setup required for the network namespaces to work. To enable this feature intrusive networking procedures were performed that could possibly impact any test that were to be run after it. To prevent any failed tests these tests were deemed intrusive and were handled in a more delicate manner than other tests. Any test that requires to change the default environment will be run in a temporary environment that is deconstructed after test execution.

4.5.2 Central build

One of the most time consuming parts of the IncludeOS installation process is building the operating system. With every run of the testing pipeline there is a need for a new compilation of the source code. Actually running the test routine has a variable time duration. The time it could take to run a number of tests is shown in figure 4.15 on the following page. The individual tests are run on separate machines so the compilation step is getting run in parallel.

This is not a very efficient way of running the tests as the same process is repeated multiple times. This is why some work was put into creating a centralized build step that then distributed the compiled code to the test executors. The immediate benefit to this change is that it reduces the resource usage, but there is another benefit. By dedicating more resources to the build server time is saved from the execution of the complete pipeline. With the old way where there was distributed compilation every single node would need an upgrade to see considerable time savings.
Figure 4.15: There is a fixed compilation time regardless of the test. The test duration differs with the type of test.

Figure 4.16: Centralized build stage and distributed test execution.
Figure 4.17: Modular tests allows for dynamic reordering of the tests in execution

Key lesson

Look for any task that is repeated multiple times and try to centralize the execution to one location. If it is a demanding task then any extra resources put towards it will yield great benefits.

4.5.3 Modular test design

By creating standalone test routines as described in the central build chapter there are a lot of new opportunities that arise when executing the tests. The goal when running the tests is to execute them as quickly as possible. When there are limited resources available having the opportunity to make sure the resources are fully utilized is important.

One possibility is to create a dynamic system where the tests can be executed on the resource that is free. This concept is shown in figure 4.17.

4.5.4 Future plans

The optimizations done in this final chapter were possible and apparent because of the incremental nature of the development of the test pipeline. At times the choices made did not branch into the state that the pipeline is in today, but it still led to a key lesson and forced a lot of discussions. One of the advantages of constant experimentation is that even though the experiment fails there is a lot to be learned, and the experiences can be used in future features.
The basics of this testing pipeline consisted of utilizing the infrastructure and properly interfacing with the tests by creating a testing tool. Mastering these basics has led to more complex actions being possible. At this point in time there are more optimizations available that are planned as future work.

One of the developments that will allow the most flexibility in the future seems to be that the individual tests are now treated more like objects. This is in the sense that each test can have various options and properties. Having this expandable set of properties for each test means that highly dynamic test pipelines can be created. The ordering of the test execution can be altered based on properties. This has inspired two possible optimizations:

- Look at what tests failed on either the previous run or which tests have the historical greatest chance of failing. These tests will then be run first.

- Analyze which parts of the repository the committed code touches and run the respective tests first.

By performing this dynamic ordering it will help ensure that any failures that happen will come as early as possible and shorten the amount of time in waiting for any developer.

To enable some of these options it requires to save information about past test runs. By collecting this data there is another positive outcome. The work with the testing pipeline is an iterative process where improvements and time savings are a constant requirement. When collecting historical data it helps highlight where time savings could be prioritized. It will also show the amount of time the time saving efforts have actually saved.
Key lesson

If a goal of the testing pipeline is to shorten time to release there are a lot of small operations that can save time. Performing regular reviews of the time usage in the test pipeline will highlight areas that need extra focus.
Chapter 5

Results 2: Model development

The case study was completed as a practical exercise in order to give experiences upon which to build a model. Creating the product described in the case study was an iterative process which resulted in a testing pipeline that is in use by IncludeOS today. Going through the process of creating the complete testing pipeline led to a lot of key lessons. These lessons are highlighted in the chapter describing the case study, and are the moments where a breakthrough in the work was made.

There are many directions to take when creating a model, due to many scenarios and use cases. Examples can be:

- Create a step by step guide where there are only a few variables that can be modified.
- Create a best practices model which tries to find a middle ground upon which all pipelines can build a foundation.
- Create a loose framework which puts the user in a state of mind where a lot of the blanks can be filled in later.

It is therefore important to figure out which type of model that is getting created. The type of model that will be created for this thesis will attempt to create a loose framework which will attempt to change the thought process of the users. By having this general approach the thought is that the model will fit a larger number of potential projects and will not be outdated by shifts in technology. This is achieved through giving the user a process where most of the parameters are provided by the user himself. The benefits and processes that this model provide are:

- Introduce terminology that will help in the communication when planning and discussing testing pipelines.
- Help put the goals of the product in line with the design of the testing pipeline.
• Transform the goals into numeric variables that act as a tool for tracking continuous improvements.

• Inform of best practices and processes that will be useful when implementing the actual pipeline.

The final requirement is that the model should be a standalone product. This means it should not be a requirement to read the entire thesis in order to get something worthwhile from the model. The actual model should be easy to extract from the rest of the thesis.

5.1 Designing the model

The model was designed after having worked with the case study for an extended period of time. The discussion that led to the model focused on the following areas:

• What could be done differently knowing what you know now?
• Are you happy with the state of the project at this current time?
• What factors stand in the way for future work?

It was clear that the case study created a solid continuous integration pipeline that was fully functional. What was turning out to be a larger issue was continuing to work on it. There was a feeling that the work had stalled, and knowing what to prioritize was getting more difficult. Through discussion with my supervisor Kyrre Begnum we uncovered the need of a framework that would assist future work. Designing the model was an iterative process that was possible due to having a very specific end goal in mind. This end goal was to help the pipeline improve continuously.

The discussions led to prototypes of the model getting created. Reading through these prototypes uncovered that they could be too technical and abstract. To counter the technical approach real life examples were added that explained the process step by step. The final piece that was needed was a work sheet that could be filled out when going through the process. This work sheet made the entire process more tangible as it now resembled a finished product that can get implemented.

5.2 Desired impact for model

Before going to the model it is crucial to make sure that the context of the model is fully explained. A proper explanation of the problem that this model will attempt to improve will put the reader in the right frame of mind.

After working within a software project and maintaining and adapting the continuous integration pipeline along the way the need for a framework
arose. There are multiple stages to go through when implementing a new project. These could be:

A. Get a basic proof of concept to work.
B. Implement the bare bones functionality to build upon.
C. Extend the functionality to cover what was planned.

This means that at one point the product that has been implemented reaches a point of maturity where all the basic functions are getting fulfilled. There are no glaring faults or missing parts, but that does not mean that the pipeline is completed. As the product it is supporting evolves there arises a need to also evolve the pipeline.

It is at this stage that I believe there is room for improvement. When there is a pipeline in place that is in need of a system that facilitates continuous improvement over time. At that point in time it is not always apparent what should get prioritized, and how it aligns with the project goals.

To further illustrate this issue figure 5.1 shows how the management and the pipeline can be disconnected. This situation can arise due to the way the pipeline was first developed. It is natural for the team that created the pipeline to continue operating and evolving it, but that means there is a highly localized expertise that is most likely not very transparent. Coming out of a very focused development loop often means there should be a framework in place that puts the goals up front to reflect upon.

![Diagram of management and pipeline disconnect]

Figure 5.1: Disconnect between management and goals and specifics of the pipeline.

By implementing a system that continuously reevaluates the project goals in regards to the CI pipeline the whole process becomes a lot more transparent. Goals are relatable both to the management and to the technical staff. The goals that are set by management need a more visible link to features and properties in the pipeline.

With clear goals that are linked to the pipeline it is easier to create
Figure 5.2: By introducing goal based management to the CI pipeline the process becomes much more transparent. Actionable tasks that will directly improve the pipeline.
5.3 POP - Pipeline Optimization Process

5.3.1 Intended use and audience

In order to compete in today’s agile software development cycles, software needs to be adaptable in order to quickly respond to any changes in demand. To meet these needs a dynamic workflow is needed. A possible solution is to implement a continuous integration pipeline which helps with automating testing and deployment. The pipeline itself is not a static product which is set up once and left alone. It will also have to adapt to any changes to the software it is supporting. In order to create the best pipeline possible there needs to be goals specified that govern the functionality of the pipeline. These goals are not often specified explicitly and should be based upon the goals of the product it is supporting.

POP is a framework created to help highlight the link between a software products development goals and the implementation of a continuous integration pipeline. Product goals such as rapid deployment or high software stability demand that the continuous integration pipeline completes quickly and performs thorough testing. POP will help translate specific product goals into actionable tasks that will directly improve the continuous integration pipeline to increase goal achievement. This is done through establishing a continuous feedback loop which ensures a dynamic pipeline that can quickly adapt to any new demands.

The framework is intended for any organization which utilizes any soft of continuous integration pipeline. It can be implemented by an organization just beginning to use a pipeline, or it can be adapted to work in an organization that has a fully functional process. It does not only relate to the engineer working on the pipeline. By focusing on product goals it engages management to participate in the discussion and by creating actionable tasks developers can get involved by implementing new functionality that benefits the pipeline.

5.3.2 Overview and goals

The end goal of implementing POP is to create a better testing pipeline. This is done through knowing and identifying the product goals, creating quantifiers which show properties of the pipeline and finally evaluating progress continuously.

The model is therefore split into three separate sections all focusing on a different activity. The workflow when using POP is shown in figure 5.3 on the next page, this figure also shows what activity each section focuses on. Outside of POP is the box labeled "perform pipeline optimization tasks", this is where the actual work of improving the pipeline is performed. POP is the management overhead that helps ensure that the work performed takes the pipeline one step closer to goal completion.
Figure 5.3: Workflow when using POP showing the three main activities

The first activity is **goal alignment**. The goal there is to translate the software product goals into goals that fit the pipeline. Many projects might not be fully in touch with what their product goals are and this section will help uncover what is most important for their project.

The second activity is **tracking evaluation**. When pipeline goals are uncovered there is a focus on defining numeric variables that can be extracted from the pipeline. With these variables it is later possible to show if changes are taking us towards our goals. This activity also assesses if the variables are available in the current pipeline infrastructure. Any missing variables are registered and the amount of work to implement them is noted.

The third activity is **plan goal achievement**. This is the final activity where any changes made can be evaluated. If there was put in work to improve the pipeline there is now a chance to see if any of the variables show a visible improvement. Based on the current state of the pipeline this is also where a prioritized list of new tasks can be created. This task list is then based upon goals and the impact of the change can be evaluated after it has been implemented.

In order to help reach these goals POP suggests processes and activities. Through these activities there will be an increased focus on the factors that help create a process that improves over time. When using the model there is a focus on filling out the variables that are uncovered in a work sheet. This work sheet is available at goo.gl/QrwyxO.
Using the model for the first time could require a little assistance. In order to provide an additional resource all the major events of the model are assisted by an example which shows how the activity could have been completed. The work sheet will also be filled gradually when going through the example.

Example small organization

This is a small organization consisting of 5-10 people. They are working on an open source application which they are hoping to monetize in the future. At the moment the primary focus is on creating a solid foundation to show potential users that they are an active team who are frequently updating the software. The team has a very flat structure, though one team member has been tasked with setting up and maintaining the continuous integration pipeline.

5.3.3 Terminology

When embarking on the design and creation of a testing pipeline a common understanding and language is important. This is why the terms are defined at an early stage.

Infrastructure The environment that is used to run the software and the tests. This can be anything from physical computers to a cloud based solution.

Pipeline A set of processes that follow one another in specific order. Is often used to describe a test and release scenario where code is expected to first go through a series of tests and if the tests pass the code moves on through the pipeline and get a release. A pipeline is often automated, but can require manual intervention.

Test tools Tools are software that help facilitate the operation of other software. Test tools are tools that are only needed when testing the software. The actual release and deployment of the software might not require the same tools. This means that the tests tools are completely separately maintained from the rest of the project.

User feedback This refers to the activity where the user is given feedback from the testing pipeline. This can be activities such as test results, reports and log files.

Key Process Indicator A key process indicator (KPI) is a quantifiable value that helps describe parts of a process. This is so that any changes to this value can be tracked. An example of a KPI is the time it takes to run through all the tests in a project, or the total number of operating systems supported by the product.
5.3.4 Goal alignment - Translate product goals to quantifiable goals that fit the pipeline

The first activity is focused on figuring out what is important for the continuous integration pipeline. The workflow in this first activity is shown in figure 5.4. By using project goals and pipeline metrics a discussion is conducted which results in creating a list of goal aligned pipeline metrics. These are the metrics that describe the pipeline that we are trying to create. In order to know if our pipeline is getting better there is a need to track it’s evolution.

![Diagram of goal alignment process]

Figure 5.4: Flow for the goal alignment activity

**Project goals**

The first step is knowing what the product goals are. This could be something as high level as selling tickets, or connecting people. What we are interested in might relate closer to what are called project goals. A few examples of goals that can influence a continuous integration process are:

- **Adoption** The potential reach of the product is dictated by the number of platforms it supports. In order to achieve high adoption the software might need to run on multiple platforms and operating systems.

- **Release frequency** It is important to have an intentional relationship with the release frequency. A high frequency lets the team quickly respond to market shifts, product demands and security updates. In order to have a short release frequency a lot of automation and confidence is needed. So by deciding that a short release frequency is not crucial a lot of work can be avoided.

- **Software stability** Certain applications have a very high demand for stable software. Aircraft applications and cooling systems highly
prioritize stability. Other types of products might have a higher
tolerance for instability and knowing that any bugs can be fixed in
the next release is good enough.

**Budget** The budget often correlates to the scale of the project. It impacts
the number of people in the team and the hardware for testing. With
a larger budget there might also be a higher demand for both the
quality and size of the finished project.

The goals might not be clearly defined, so in order to figure out what
these goals are means involving the right people. Depending on your
organization this could mean involving people like:

- Product owner
- System architect
- Service manager
- Project manager

**Pipeline metrics**

When going into a meeting where these goals will be discussed there is a
second factor that needs to be part of the discussion. That is the metrics that
help show the properties of the pipeline. Where a manager can supply a
project goal a continuous integration engineer needs to know the different
properties of the pipeline. Examples of metrics that describe the pipeline are:

**Time delta** Time it takes from triggering until the entire pipeline is
completed. Low time delta gives a fast pipeline that provides quick
feedback. A longer time delta could imply a more thorough test suite.

**Test scope** The test scope describes the test coverage in the project. A high
test scope aims to uncover more errors when run, but will usually
take longer to complete. The advantages of a small test scope could
be a faster pipeline or less maintenance work.

**Test linearity** When running a large number of tests the order in which
they are run could be important. A dynamic ordering could enable
frequently failed tests or the most relevant tests to the changes
committed to be run early. This sort of behaviour requires the test
tools to be compatible. A linear test system will often be easier to
implement.

**Infrastructure dependency** Certain testing scenarios have demands for a
certain type of infrastructure. This could mean that the infrastructure
itself must be actively managed. An automatic way of setting up
environments for the test runs will also need to exist. A system with
a low infrastructure dependency is highly portable and can run in a
large number of operating systems.
**Support scope** Indicates how many operating systems or hardware configurations that should be supported. Increasing the support scope increases the reach of the software as more people are given the opportunity to use the software. It also increases the demands of the testing as multiple platforms and environments will need to be maintained. Having a low support scope focuses the testing to a smaller number of platforms and reduces the complexity.

**Tool complexity** The complexity of the tools utilized to create and maintain the testing pipeline. A low tool complexity means it will be easier to maintain and adapt to new scenarios. A high tool complexity could enable more options.

**Feedback scope** The feedback scope gives an idea of how much information should be provided about the results of the tests. The smallest possible feedback scope could limit the information to a simple green light if the build and tests passed. Detailed test reports and test console outputs increase the feedback scope but might require a more complex setup in order to support.

**Discussion - Goal aligned pipeline metrics**

With both the **project goals** and the **pipeline metrics** in mind the team will have to figure out how they wish to prioritize between these.

The goal of this discussion is to figure out what project goals are most important, and then which pipeline metrics help highlight that specific project goal. If for instance frequent releases is a project goal then having a low time to test is a pipeline metric which corresponds nicely and helps achieve this project goal. By the end all the important project goals should have pipeline metrics which are weighted and have a link to a project goal.

All of the project goals and pipeline metrics should then be written down in a table that is called **goal aligned pipeline metrics**. An example of such a table is shown in the example below.
Example small organization

The small organization gathered the entire team for the discussion. Their product is a small open source project so their primary goal is to show a lot of activity by pushing out frequent releases in hopes of increasing adoption. This means that they strive for a low time delta, one possibility to achieve this is to increase the test linearity. They decided to only support one major platform in the beginning to hold the support scope low.

Due to being at an early stage in development software stability is not the most important factor. This leads to them choosing a smaller test scope and the feedback provided by the support scope is not prioritized yet.

The final limiting factor is a low budget. The team is small so the task of maintaining the continuous integration pipeline is limited to one person. This means they want to keep both the infrastructure dependency and tool complexity low.

The output of the discussion is noted down as in the table below.

<table>
<thead>
<tr>
<th>Project goal</th>
<th>Pipeline metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequent releases, high adoption</td>
<td>Low time to test</td>
</tr>
<tr>
<td>Low software stability</td>
<td></td>
</tr>
<tr>
<td>Small test scope</td>
<td>Small feedback scope</td>
</tr>
<tr>
<td>Low budget</td>
<td></td>
</tr>
<tr>
<td>Low infrastructure dependency</td>
<td>Low tool complexity</td>
</tr>
</tbody>
</table>

61
5.3.5 Tracking evaluation - Tracking pipeline properties in your project

At this point all the pipeline metrics are discussed and prioritizations are made for your specific project. The next step is to go from a theoretical metric to a numeric variable that fits a specific project and their technical implementation. The numeric variable are what we call a "key process indicator" (KPI), which summarizes a metric into a single variable. Integrating these variables into a project is a different process for every implementation due to widely different project architectures. The solution is to create a report which documents the readiness of the project in relation to all the required KPI variables.

The process for generating the KPI readiness review is shown in figure 5.5. You take the goal aligned pipeline metrics from the previous activity, turn them into KPI variables and then you consider your specific implementation and create a KPI readiness review.

![Diagram of the tracking evaluation process](image)

**Figure 5.5: Process for setting up the tracking of the KPI variables**

**From theory to real world variables - KPI variables**

The first step is to create the numeric variables that are referred to as key process indicators (KPI). Setting up these variables enables the continuous evaluation of the CI pipeline.

Each of the pipeline metrics that have been adapted to fit a specific project will have to find a variable which has a real world meaning. The variable
that is chosen needs to be a practical number which is possible to extract from a pipeline.

Example small organization

In order to create KPI variables for their project the team assess the pipeline metrics they have chosen to focus on. In order to verify a low time to test they will need to measure the \textit{delta_time}. A goal was to keep the number of supported platforms low so the \textit{support scope} was to be kept low. In order to track this they created the variable \textit{num\_platforms} which keeps a track of the number of platforms they are testing on. The final metric which helps the team reach their goal of frequent releases was to have a high \textit{test linearity}, to measure this they want to know how many tests they are capable of running in parallel \textit{num\_linear\_tests}. The team perform the same considerations for the remaining metrics and fill in their worksheet with the variables. The result can be seen below.

<table>
<thead>
<tr>
<th>KPI variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project goal</td>
</tr>
<tr>
<td>Pipeline metrics</td>
</tr>
<tr>
<td>KPI variable</td>
</tr>
<tr>
<td>Project goal</td>
</tr>
<tr>
<td>Pipeline metrics</td>
</tr>
<tr>
<td>KPI variable</td>
</tr>
<tr>
<td>Project goal</td>
</tr>
<tr>
<td>Pipeline metrics</td>
</tr>
<tr>
<td>KPI variable</td>
</tr>
</tbody>
</table>

Plan and consider implementation - KPI readiness review

This is where the actual implementation is considered. For each of the KPIs in the project the following must be considered:

**Availability** Is the information already available? If it is and it can be easily extracted then it should be easy to use going forward.

**Format** The format of the information is important. The information can be accessible from an API, a dashboard with visual graphs or could require a manual count. The format decides how accessible the information is and how easy it would be to automate the collection of it.

**Completion** An estimate of how accessible and complete this KPI variable
is. If it is easy to automatically extract this variable the completion can be set to 100%, if it does not currently exist give it 0%. If it requires manual work then give it an estimate in between depending on how much work it would need to automate it.

After these considerations the actual KPI readiness review can be created. This acts as a summary of the status of all the KPIs. The factors that can be put into the review can be:

**Total KPI variables**  A count of the total variables that should get tracked.

**Total completion**  An average completion for all the variables.

**Work to do**  For all the variables that do not have 100% completion there should be a description of the work needed to complete the implementation.
Example small organization

The team considered the KPI variables they needed and assessed their continuous integration pipeline. For `delta_time` they knew this was available through their Jenkins API. They supplied the link to the variable directly and set this to 100% completion. For `num_platforms` there was no API access, but they could access the information manually by looking in their config files. They considered this as a 50% finished implementation. Finally for `num_linear_tests` they knew that they ran tests in parallel, but that it depended on their resources. To figure this out it would require manual checks of the log files. They deemed this implementation as 25% done. They filled in the form for the rest of their KPI variables, they are not shown in this example.

After these considerations the first KPI readiness review was conducted. They plugged in 3 as total number of KPI variables. An average of 58% completion was filled in. To increase their completion a number of work tasks were detailed.

---

### Goals, Metrics and KPIs

<table>
<thead>
<tr>
<th>Project goal</th>
<th>Frequent releases, high adoption</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pipeline metrics</td>
<td>Low time to test</td>
</tr>
<tr>
<td>KPI variable</td>
<td>delta_time</td>
</tr>
<tr>
<td>Link</td>
<td>10.0.0.1/delta_time</td>
</tr>
<tr>
<td>Form</td>
<td>Jenkins API</td>
</tr>
<tr>
<td>Completion</td>
<td>100 %</td>
</tr>
</tbody>
</table>

### KPI readiness review

| Total KPI variables | 3 |
| Total completion | 58 % |
| Work to do | num_platforms | Create an automated way of capturing number of platforms |
| | num_linear_tests | Extract number in test tool so it is easily available |
5.3.6 Optimize goal achievement - Create a continuous improvement cycle

The final section is where the goals and KPIs are turned into actionable tasks that can be completed and implemented. The process that is completed to produce this list of tasks is shown in figure 5.6. The input used in this activity are the previous tasks completed and the KPI readiness review. Looking at these factors and evaluating compared to the state of the continuous integration pipeline gives an updated view of the current situation. Based on this situation a list of new tasks can be planned.

![Diagram](image)

Figure 5.6: Process for going from goals and KPI variables into actionable steps

By creating this continuous cycle there arises a direct connection between the project goals and the tasks performed to help reach them. If the KPIs generated properly convey the needs of the product there is now quantifiable data which will show that the specified goals are getting accomplished. If there are signs that the goals are not met there is the option to either reevaluate the goal or change the approach.

If the progress is evaluated at a regular interval there will be a visible link between changes to the product and improvements seen in the KPIs. After an extended session where considerable changes were made it is possible to see how they impacted the goals of the pipeline.

One of the largest benefits is that you are close to the project goals on a regular basis. The criteria of success are constantly evaluated.
In order to implement this continuous improvement process there is a need for time and resources. By creating a system that is used to show progress towards project goals this will help justify the time usage.

**Evaluating the current state - Change review**

The first activity to perform is to use the previous tasks that were completed and the KPI readiness review to assess where the continuous integration pipeline is at this time. The output from this process should tell you how recent changes to the CI pipeline moved you closer or further apart from your goals. The output is what is called a change review, which summarizes the changes made and shows how this impacted the KPI variables.

The change review consists of two tasks:

**Review tasks completed** Since the previous review was completed the work done in between should be reviewed. This can be one of two types of work:

- **KPI implementation** Tracking of a KPI variable has been implemented or improved. This will mean that future evaluations will benefit from an additional KPI variable tracking.

- **CI pipeline change** These are tasks that directly affect the CI pipeline. This can be the implementation of a new tool, the improvement to an existing process. These changes should change the values of your KPIs in some way as they will affect either the complexity of the pipeline or make changes to the efficiency.

**KPI progress evaluation** This is where all our actively tracked KPI variables are listed and the value from the previous run is compared to the current value. Then the change can be calculated. With a number set for the change it is easy to see if there has been a positive or negative impact to the overall CI pipeline.
The organization is going to perform a change review after a sprint consisting of some work related to the CI pipeline. There were two main tasks that were completed since the last change review. The first was a **direct CI pipeline change** which added hardware with the hope of improving \textit{num\_linear\_tests}. The second was a **KPI implementation** where the generation of the value for \textit{num\_linear\_tests} was simplified.

After listing out the tasks that were completed the KPI variables were extracted from the pipeline and a comparison to the previous run was done. This showed that the \textit{delta\_time} was improved by 16%, and that the \textit{num\_linear\_tests} was improved by 25%.

### Change review

<table>
<thead>
<tr>
<th>KPI variable targeted</th>
<th>Change made</th>
<th>Type of change</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{num_linear_tests}</td>
<td>Added hardware to increase number of parallel tests</td>
<td>CI pipeline change</td>
</tr>
<tr>
<td>\textit{num_linear_tests}</td>
<td>Test tool now reports number of linear tests automatically</td>
<td>KPI implementation</td>
</tr>
</tbody>
</table>

### KPI progress evaluation

<table>
<thead>
<tr>
<th>KPI variable</th>
<th>Previous</th>
<th>Current</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{delta_time}</td>
<td>60 sec.</td>
<td>50 sec.</td>
<td>16% improvement</td>
</tr>
<tr>
<td>\textit{num_platforms}</td>
<td>3</td>
<td>3</td>
<td>0 %</td>
</tr>
<tr>
<td>\textit{num_linear_tests}</td>
<td>4</td>
<td>5</td>
<td>25% improvement</td>
</tr>
</tbody>
</table>

### Plan tasks - Task list

After creating a change review where the previous work was evaluated the final task is to create a plan for new tasks to complete. There are a few ways to go about this, depending on what you wish to focus on. To make a decision on what tasks to create the project goals should be considered, and the KPI readiness review. If there are still KPI variables that have no data this could get prioritized. If all variables have data then the time can be spent improving the data gathering or the time can be spent improving the CI pipeline itself.

One possibility is to do a **hardening sprint**. If the goal is to improve the pipeline considerably a whole work sprint could be dedicated to improving parts of the pipeline. If there is a need to improve the time usage of the pipeline the entire team could get involved and it could turn into a team project. After the hardening sprint is complete another change review is
conducted to evaluate the impact of the changes made and form the basis of another task list.

**Example small organization**

The final task for the team was to create a task list for the next work sprint. There was a need to improve the `delta_time` even more, so they decided to drop the number of supported platforms. This they hope would drop the pipeline execution time. In order to evaluate this change in a better way they also wanted to improve the KPI implementation of the `num_platforms` variable by creating a script which automatically extracts the number of platforms it is tested on.

<table>
<thead>
<tr>
<th>KPI variable targeted</th>
<th>Task</th>
<th>Type of change</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>delta_time</code></td>
<td>Decrease number of platforms targeted</td>
<td>CI pipeline change</td>
</tr>
<tr>
<td><code>num_platforms</code></td>
<td>Create script for capturing number of platforms from configuration</td>
<td>KPI implementation</td>
</tr>
</tbody>
</table>
5.3.7 Summary

At the end of this process there should now be a relationship between the project goals and the actionable tasks that are to be completed in the next work session. In addition there are also 4 documents that help show this workflow. These are:

**Goals, Metrics and KPIs** A table summarizing the link from a project goal to pipeline metrics to KPIs and their implementation.

**KPI readiness review** Summary table of the total KPI readiness of the entire pipeline.

**Change review** Evaluation of a previous work session. Changes made are documented and the effect on specific KPIs are shown.

**Task list** List of new tasks to perform and a desired effect from the changes.

The deliverables are also shown in the complete workflow summary in figure 5.7. A complete work sheet showing all these tables is also provided.

![Figure 5.7: Process summary with included deliverables](image)

70
## Goals, Metrics and KPIs

<table>
<thead>
<tr>
<th>Project goal</th>
<th>Frequent releases, high adoption</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pipeline metrics</td>
<td>Low time to test</td>
</tr>
<tr>
<td>KPI variable</td>
<td>delta_time</td>
</tr>
<tr>
<td>Link</td>
<td>10.0.0.1/delta_time</td>
</tr>
<tr>
<td>Form</td>
<td>Jenkins API</td>
</tr>
<tr>
<td>Completion</td>
<td>100 %</td>
</tr>
</tbody>
</table>

### KPI readiness review

<table>
<thead>
<tr>
<th>Total KPI variables</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total completion</td>
<td>58 %</td>
</tr>
<tr>
<td>Work to do</td>
<td></td>
</tr>
<tr>
<td>num_platforms</td>
<td>Create an automated way of capturing number of platforms</td>
</tr>
<tr>
<td>num_linear_tests</td>
<td>Extract number in test tool so it is easily available</td>
</tr>
</tbody>
</table>

### Change review

#### Tasks completed

<table>
<thead>
<tr>
<th>KPI variable targeted</th>
<th>Change made</th>
<th>Type of change</th>
</tr>
</thead>
<tbody>
<tr>
<td>num_linear_tests</td>
<td>Added hardware to increase number of parallel tests</td>
<td>CI pipeline change</td>
</tr>
<tr>
<td>num_linear_tests</td>
<td>Test tool now reports number of linear tests automatically</td>
<td>KPI implementation</td>
</tr>
</tbody>
</table>

#### KPI progress evaluation

<table>
<thead>
<tr>
<th>KPI variable</th>
<th>Previous</th>
<th>Current</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>delta_time</td>
<td>60 seconds</td>
<td>50 seconds</td>
<td>16% improvement</td>
</tr>
<tr>
<td>num_platforms</td>
<td>3</td>
<td>3</td>
<td>0 %</td>
</tr>
<tr>
<td>num_linear_tests</td>
<td>4</td>
<td>5</td>
<td>25% improvement</td>
</tr>
</tbody>
</table>

### Task List

<table>
<thead>
<tr>
<th>KPI variable targeted</th>
<th>Task</th>
<th>Type of change</th>
</tr>
</thead>
<tbody>
<tr>
<td>delta_time</td>
<td>Decrease number of platforms targeted</td>
<td>CI pipeline change</td>
</tr>
<tr>
<td>num_platforms</td>
<td>Create script for capturing number of platforms from configuration</td>
<td>KPI implementation</td>
</tr>
</tbody>
</table>
5.4 Expert opinion

In order to assess the model it was desirable to get an external opinion. This was done by conducting interviews with people who have relevant experience in the field of continuous integration.

The people were asked to read and assess the model before we sat down and talked about their take on it. During this interview the main points that were in focus were:

- Overall structure and readability
- Applicability in a real life scenario
- Usefulness in a real life scenario

The interviews were conducted with:

- Michael Long. Works at Praqma and consults organizations wanting to make the necessary changes to improve their DevOps work flow.
- Per Buer. Current CEO of IncludeOS. Experience from a wide range of technical positions.

The feedback can be summarized in the following points:

**Context** When first reviewing the model it was not fully apparent exactly what problem the model was attempting to solve. It was not before one of the examples given that the context became more clear.

**Details** When the context is not clear the model would seem overly focused on details. The key process indicators that were proposed were very detailed, and their usefulness was not apparent.

**Audience** Exactly what person or which team should use the model was also not properly conveyed. A better instruction as to which team members would benefit from the model was needed.

Overall the feedback provided showed that the introduction to the model needs to be improved. To remedy this the chapter named desired impact for model was added prior to the model in the thesis.
Chapter 6

Analysis

After having laid out the results in the previous chapters it is time to assess. The problem statement accurately describes what we are trying to answer. In the analysis chapter it is time to compare the initial problem statement to the finished result. To perform this analysis the focus will be on the adjectives of the problem statement. The adjectives describe the desired properties of the finished product. The analysis will weigh these properties and decide if they fulfill the problem statement.

6.1 Feature analysis

In order to assess if the goals have been achieved a feature analysis will be performed. The feature analysis looks at the individual parts of the model and tries to compare them to the desired properties of the problem statement.

6.1.1 Terminology

The model contains a terminology list that aim to clarify some key terms that are used within the model. It is important not to overload the reader with too many terms, and to make sure the ones specified are properly described. To evaluate whether or not these goals were achieved table 6.1 on the following page has been created which evaluates every term from the terminology.

6.1.2 Model/framework

The problem statement sets the goal of developing a model. The model should help develop a pipeline that is efficient, automated and continuously improving. These properties of the pipeline are what should get evaluated.
Table 6.1: Table evaluating the terms described in the terminology

<table>
<thead>
<tr>
<th>Term</th>
<th>Properly described</th>
<th>Used in the model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Infrastructure</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Pipeline</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Test tools</td>
<td>Yes</td>
<td>Barely</td>
</tr>
<tr>
<td>User feedback</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Key process indicator</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Efficient** In the introduction efficiency was defined as performing or functioning in the best possible manner with the least waste of time and effort. The way the model ensures efficiency is by constantly making the user highlight in what areas the pipeline is lacking. Implementing the key process indicators as suggested in the model links the performance of the pipeline to a numeric metric which allows it to be monitored. Through this monitoring it will be visible which parts of the pipeline needs to be improved to increase efficiency.

The model does not by itself ensure efficiency, it is up to the user to prioritize efficiency if that is what is most important. With that in mind it can be concluded that the model does not guarantee efficiency, but it facilitates it if the user makes it a priority.

**Automated** The second property is automation. This property is not as easy to spot in the model. Surrounding the gathering of data for the KPI variables there is a clear advantage to implementing automation, but as for the pipeline by itself there is no real incentive which ensures automation. In the same way that efficiency is encouraged through highlighting what areas of the pipeline to improve it can be argued that automation is also encouraged. It all comes down to the end user and what they choose to prioritize in their individual pipeline.

The one part that is encouraged to automate is the acquisition of the KPI variables. This ensures the process of evaluating changes and improvements is much easier.

**Continuously improving** The one area where the model really shines is in the way that continuous improvement is encouraged. The whole system of identifying, acquiring and tracking KPI variables is so that continuous improvement can be tracked and encouraged. By performing this tracking there is data available which can be used to identify which part of the pipeline needs extra work. Utilizing this data to generate concrete work tasks gives it a very concrete application.

Based on the feedback from the experts the one area where the model seems to be lacking is providing a clear enough context. When they studied the model it was not clear what specific issue the model was trying to solve. This means that the details of the model do not make as much
sense. The introduction to the model should provide the reader with a clear sense of the specific problem that the model can help them with. The examples were highlighted as a positive addition that helped contextualize the model.

In total the model seems to accomplish what it has set out to do, but with a clear need for a better introduction to contextualize the need for the model and framework.

6.1.3 Framework as a tool for improved devops

The problem statement was not proposed in a vacuum. It fits into a reality where there is a demand for fast and highly agile software projects. The framework proposed has to fit into this reality.

By focusing on continuous improvement the model ensures that it will help with constant changes. There is some overhead to utilizing a model such as this so the benefit needs to outweigh the time spent.

One of the most important properties of the model is the promise of continuous improvement. The thought is that the pipeline will increase in quality over time and it will adapt to changes in the project it is supporting. Due to not having implemented the framework in an actual project it is not verified that the promises of the model can be realized.
Chapter 7

Discussion

The discussion is where a proper reflection of the whole thesis will take place. The results are in and the analysis provides an assessment of goal achievement compared to the problem statement. The discussion gives room to put all the parts of the thesis in context. The choices made in the approach can be analyzed and their impact on the results can be laid out. Choices that were made can be discussed and other options can be explored.

The discussion will be structured in a reverse fashion. The most recent chapters will be discussed first, and the end goal is the problem statement and the impact of the thesis. It will be split up into sections that capture a novel idea or observation. These sections discuss whether a specific issue went as planned or if it did not. The cause of this mismatch or success will be explored and finally a summary of how to avoid or repeat this issue in the future.

7.1 Analyzing a framework

The main deliverable of this thesis was the POP framework. This is a very soft release, which means that there is no definitive way to know if it fulfills the goals of the problem statement or not.

To combat this uncertain way to perform the analysis the main structure of the analysis tries to relate to the overarching goals of the thesis. The problem statement specifies to develop a model and terminology so the first part of the analysis analyses these two parts of the model. The terminology was analyzed by considering the descriptions and looking to see if they were actually used in the model. If the terms were never used in the model there is no value in including them in the terminology. The second analysis that could be performed would be to look for terms in the terminology that are missing. It might also be smart to highlight this during the interview with the experts as they could easier know what was
The second part of the analysis looks at the model. Again the problem statement is used to measure how well the model achieved its goals. The model should help develop a pipeline that is efficient, automated and continuously improving. These three properties of the pipeline are discussed in relation to the content and outcome of the model. This approach works well and ensures that the analysis links back to the goals of the problem statement.

The final part of the analysis tries to relate the model to the context it is created from. This section proved more challenging to analyze properly. This is mostly due to a lack of data. Implementing the model was out of scope so there are no practical experiences to analyze. Letting this final part of the analysis be a little short does not undermine the actual model, but it leaves the proof up in the air. Following up the thesis with an actual implementation would help prove the validity of the model.

When performing an analysis of such a soft result the main takeaway is to link it as much as possible with the problem statement.

### 7.2 Using an interview to assess the model

The interview of experts was chosen as a means of data gathering. The actual data that came of this process proved to not be very useful.

This was mostly due to a lack of proper preparation. Before going into an interview like this there should have been a clear objective of what data to gather. The interviews ended up being much more loosely structured. This loose structure meant that the interview had to be turned into data after the interview was finished. Again this was an issue as there was still no clear format or actual data to turn the interview into.

It is clear that the interview process should have been better prepared in the approach part of the thesis. This comes down to two factors that would have made this better:

- Identifying what data would help properly analyze the model and impact of the model.
- Planning how the interview would ensure that the data required would get answered.

The second factor that contributes to the lacking outcome from the interview process is the fact that it is not a normal skill for this area of study. Conducting data gathering through interview is an area of expertise that should have been delved into. Expecting results without proper background knowledge is not realistic. Given the field of study which was network and system administration it was natural that conducting
7.3 Importance of considering the problem statement in every part of the thesis

The problem statement is preceded by the introduction. The introduction does a good job of gradually building up to the problem statement. This narrows the scope of the thesis, but it also puts the reader in the same frame of mind as the author. This gradual narrowing should have also had a more prominent part in the model. There was a clear need of more contextualizing in the model as it took too long to convey the problem the model was solving.

Basing the model on a case study means that the author has had a long time getting properly acquainted with the subject matter. To then take a step back and write a model intended for people who are not as familiar with the intricate details proved to be a challenge. One solution would be to view the model as a miniature thesis which needs to have an introduction, a clearly defined problem statement backed by some background information, a clear approach and then the actual result. Including all of these aspects in the model would have improved the flow and better conveyed the problem that the model is attempting to solve.

7.4 Generalizing the usage of the model might hurt its usability

As the model was getting developed there was always an idea that the model could be used for multiple different scenarios. It could work for new projects that were starting out, or it could be implemented in an already existing project. Trying to cover both of these situations turned out to cause a few problems when considering the usability of the model for a first time user.

At one point during the development of the model there was a moment where the endless possibilities of the framework were uncovered. There grew a desire to create a more general framework which could be used for more diverse situations than only when creating continuous integration pipelines. Luckily the supervisor quickly ended this diversion and made it a point to focus on the specifics of pipeline creations.

There was one aspect which lived on and that was that the model should work with both new projects and existing ones. The examples created to illustrate the usage of the model were based on a new project.
beginning a new project and using the model there is a certain way to do this. The other option is to use the model on an existing project and the implementation could turn out to be slightly different. There is no example provided of implementing the model on an existing project so this might turn out to be more difficult.

The solution could have been to declare the model only to be used for new projects, but that might limit the adoption. Ideally the model is clear enough so that the reader can adapt it to the usage that it is needed for. What is clear is that if there are major differences in the usage of the model based on preexisting conditions these should be explicitly factored into the model. That will ensure a clear usage pattern for any new users.

### 7.5 Linking the case study with the model

The results section of the thesis consists of two separate parts. First is the case study which is a summary of the work done to create a testing pipeline. The second part is the model whose aim is to create a framework for helping others implement a pipeline of their own. Having these two separate parts proved to be a bit of a challenge as linking them together did not have an obvious solution.

The goal was to perform the case study and use the experiences to create a model for helping others. That aspect of the plan worked out well as the model is now a stand alone product which works well by itself. The structure of the case study followed the chronological progression of the pipeline created. To extract the most important lessons from the case study these were highlighted as they were uncovered throughout the results chapter. When reading the model it is difficult to spot exact pieces from the case study. This is mostly because the model was created by having conversations with the supervisor. These conversations would not have been possible had it not been for having gone through the case study first. The approach chapter sketched out an idea for how the model could look, and this early sketch impacted the design of the resulting model.

The link between the two results parts is still not very visible. One solution would be to flesh out the subchapter discussing how the model was designed. Ideally there would be specific incidents from the case study that influenced certain design decisions. However this turned out to be difficult to actually describe as the design of the model was created through creative discussion. The main benefit of adding this extra description would be to improve the flow when reading through the entire thesis. It would not change the actual model, as that had been created already.
7.6 Creating a model which is easy to use

The first iteration of the model was a document created in a scientific vacuum. There was a wall of text, a few diagrams and a lot of abbreviations. There was a clear need for an improvement in the usability of the model.

In order to accomplish this there were mainly two things that were done:

A. Create an example scenario which the reader can follow when going through the model.

B. Create a work sheet which can be filled out to include the proposed variables and properties.

These two improvements created a much more tangible product which now actually feels like it can be used in a real life scenario. The experience of creating these two products proved their importance. It proved to be much more difficult to create a work sheet which was simple enough yet providing the relevant information. Having these two concrete examples proved to be very important as the expert interviews highlighted that they had helped contextualize what the model was doing.

When the creator of a model finds it difficult to turn the model into something practical and concrete there is a problem. So for the model to include these practical additions helps ease the barrier of entry for new users.

7.7 No detailed plan for the data gathering

The approach chapter had a section dedicated to figuring out how to analyze the end result. The properties needed are explained in a good manner, but the plan is very lacking in how the data gathering should happen.

The approach chapter starts with a detailed description of how the properties of the problem statement should be used in the final analysis. As discussed earlier this approach turned out to be very beneficial in the analysis chapter. These properties are followed by implicit properties that also assist in describing the model. These implicit properties are all valuable factors which will help in figuring out if the model is a useful final product.

After a description of the properties there is a comparison of the various ways that data gathering could happen. Applying the model to an entirely new project, performing a comparative study or gathering expert opinions are all valid ways of gathering the data. In the end the expert opinion is chosen as the most appropriate approach. To make sure the data regarding
the various properties would get populated some sort of form should have been created which forced the experts to consider these specific aspects. In addition there should have been a clear formula for the interview and what specific questions should get answered.

By planning the data gathering more in detail the output from the interviews would have been much more useful.

7.8 Changing the problem statement during the thesis

One of the most challenging aspects of this thesis was dealing with changes to the problem statement. The case study went on for an extended period of time and due to discoveries made throughout the case study there was a need to actually change the problem statement.

The thesis was originally geared towards creating a framework for setting up a continuous release pipeline. In order to reach the point where a framework could be created a lot of prerequisites were necessary. The rest of the pipeline had to be created, and a clear testing strategy had to be developed. As the work with the case study went on it became clear that there was simply not enough material and ability to create a meaningful model.

The choice to modify the original problem statement was made in cooperation with the supervisor. It was clear that the experience gathered from the case study could be used if the problem statement was modified slightly. This need for a change should have been identified at an earlier time as a lot of the preliminary work had to be modified to fit with the new direction.

On the other hand the final problem statement was born from a real life situation where it was identified as a necessity. This kind of natural foundation to a problem statement was only possible due to the long time frame of the master thesis. The problem of adapting a continuous integration pipeline over time and aligning it with the project goals was not an apparent problem when the work on the thesis began. The discussions and initial work to create the model came very natural and the examples used in the model were all based on real life experiences.

The end result from this change of direction turned out better than what could be expected. This is mostly due to the final problem statement coming from the work with the case study. It is an ideal solution to end up with such a natural source, but in this case it happened mostly due to the original problem statement turning out to be difficult to answer within the time frame of the thesis work. The earlier parts of the thesis such as the background chapter and the approach suffered slightly from this late change in the direction of the thesis.
7.9 Difficulty of writing a background chapter while working a case study

The role of the background chapter is to survey the existing material that the thesis will build upon. A case study is meant as a hands on approach towards gathering data and experiences. Doing both of these activities at the same time means that there needs to be a way of differentiating between what knowledge should be acquired from the background chapter and what should come from the case study.

The potential downfall of getting the balance of these two wrong is that the case study is performed with the wrong starting point. When performing the background research at the same time it is therefore important to continuously evaluate the approach of the case study.

In this thesis the focus of the problem statement was modified at a later stage. This was after the background chapter was completed. Ideally there should have been done more specific research into the new aspects of the new problem statement as that would have benefited the model. Specifically there has been done a lot of research concerning agile software development. This research focuses heavily on linking project goals and work tasks.

If a change of problem statement were to happen in another project there should be made some extra research into the specifics of that change. This will help ensure that there is not something obvious that is missing with the change of direction.

7.10 Was creating a model the right solution?

The introduction chapter leads up to the problem statement. This is where it is decided that developing a model would be the appropriate approach for this thesis. After having created the completed model there were a lot of challenges that were uncovered.

Creating a model is much more than simply writing a step by step instruction for someone to follow. It is all about creating a process which can be adapted to various circumstances. The level of detail can’t be too high as it could potentially contradict certain users experiences. The most difficult part of creating the model was the pedagogical aspect. There is an aspect of psychology involved in creating the best method for conveying information. This aspect of the thesis was not dedicated the appropriate amount of time. There should have been done more research into the actual creation of models as best practices would have helped create an ever better model.

The system administration field of work is not very known for creating models and having a very scientific approach towards standardizing
processes. This is mostly due to a long history of having a hands on approach. There is also a need to be very adaptable as the technology is changing very quickly. A lack of established practices leads to many unique solutions getting developed which prevent unity and communication. In order to grow as a discipline having well thought out models and best practices will certainly help with that.

Creating the model seems to have been the correct approach when considering the goal of wanting others to learn from the experiences gathered. Writing a thesis about a specific implementation is certainly useful, but then it is up to the reader to extract what is relevant to them. The advantage to delivering a model is that the author has hopefully done a lot of the groundwork in creating a useful product for others.
Chapter 8

Conclusion

In order to keep up with the need for rapidly changing and adapting software a popular approach is to implement a continuous integration pipeline. This can help automate building, testing and releasing. The pipeline itself is a dynamic product which needs to adapt to changes in technology and changes to the project it is supporting. To ensure the pipeline can be a continuously evolving product this thesis creates a model which helps put the focus on improvements over time.

The model is based on the practical work done through implementing a testing pipeline for a company developing a unikernel operating system. This case study went on for over a year and by the end the pipeline had gone through a lot of large changes. As the pipeline grew it was becoming increasingly difficult to plan for changes and track improvements. This is where the need for a continuously improving pipeline came from.

The model proposes a workflow which first identifies the project goals and then helps translate them into goals that make sense for pipeline development. In order to track these goals there needs to be concrete variables that can be extracted from the pipeline. These are identified and knowing how to extract them is planned for. The final step assesses the project goals and creates work tasks on how to achieve them. The work tasks should then be completed and a the pipeline is then ready to be continuously improved. After a successful implementation the impact of the changes can be evaluated and used as input for the planning of the next work tasks.

To help implement this workflow the model comes with a work sheet used for filling out all of the information needed. This eases the barrier of entry as there is a logical place to put all the variables and keep track of them. It will also create a predictable process which will be the same for independent users of the model. The structured approach provided by using the work sheet removes any personal bias of the user.

By using the model it will enable projects that use a pipeline to continuously improve as the software project evolves. It will highlight how goals
should affect design decisions through a proper discussion of project goals and pipeline metrics. In the end it should create a workflow which is more dynamic and transparent.

In the future the model should be tested in a real life scenario. Using the model will help highlight if there are any design issues that prevent the implementation of the model.
Bibliography


Olsson, H. H., Alahyari, H., & Bosch, J. (2012). Climbing the” stairway to heaven”–a multilple-case study exploring barriers in the transition from agile development towards continuous deployment of software. In *2012 38th euromicro conference on software engineering and advanced applications* (pp. 392–399). IEEE.


