MASTER'S THESIS | LUND UNIVERSITY 2016

Container-based Continuous Delivery for Clusters

Per-Gustaf Stenberg

Department of Computer Science
Faculty of Engineering LTH

ISSN 1650-2884
LU-CS-EX 2016-06

Container-based Continuous Delivery for Clusters

Per-Gustaf Stenberg
dt08ps5@student.lth.se

January 18, 2016

Master's thesis work carried out at Data Ductus Malmö AB.

Supervisors: Mario Toffia, mario.toffia@dataductus.se
Ulf Asklund, ulf.asklund@cs.lth.se

Examiner: Martin Höst, martin.host@cs.lth.se

Abstract

The focus of this master's thesis was aimed at E.ON's electricity saving project, 100koll, in collaboration with IT-consulting firm Data Ductus. The 100koll users demand high availability, which creates a complex underlaying system-infrastructure. The processes of deploying and preparing new releases to the system is presently done manually. Data Ductus is requesting an investigation on whether or not containers can facilitate the preparation processes to a point where delivery is done continuously. The solution which was introduced follows five steps of implementation which involves a programmable infrastructure, deployment strategies and a deployment pipeline with a feedback system. What the solution shows us is that containers were able to facilitate certain parts of the implementation process that previously prevented Data Ductus from achieving continuous delivery. However, the conducted research also proves with the help of the implementation processes that achieving continuous delivery is not all to do with having access to the correct tools; it also has to do with the mindset of the people involved.

Keywords: MSc, Continuous Delivery, Configuration Management, Docker, Continuous Integration, Containers, DevOps

Acknowledgements

Big thanks to Ulf Asklund for the guidance throughout this Master's thesis. I would also like to thank my examiner Lars Bendix, and in the later stage, Martin Höst.

Also, huge thanks for all the support at Data Ductus. Special thanks to Mario Toffia who made this project possible and Viktor Hansson for the great guidance and support during the implementation process.

Another big thanks goes out to Malin Wiborg for the proofreading, and of course to all other people involved in the project.

Contents

1 Background and Introduction **9**
1.1 100koll . . . 9
1.2 The Problem . . . 10
1.3 What is Continuous Integration, Delivery and Deployment? . . . 11
1.3.1 Continuous Integration . . . 11
1.3.2 Continuous Delivery and Deployment . . . 11
1.4 The Concept of Containerization . . . 12

2 Method **15**
2.1 Questions . . . 15
2.2 Approach . . . 15
2.2.1 Pains and Blockers . . . 16
2.2.2 Container-based Solution . . . 16
2.2.3 Solution Evaluation . . . 16
2.3 Related Work . . . 18

3 Containers **19**
3.1 Underlying Technology . . . 19
3.1.1 Namespaces . . . 19
3.1.2 Cgroups . . . 21
3.2 Docker . . . 21
3.3 Performance . . . 22
3.3.1 CPU and Memory . . . 22
3.3.2 Networking . . . 23
3.3.3 Disk I/O . . . 23
3.3.4 Summary . . . 23

4 Taking the Step **25**
4.1 The Anti-patterns . . . 25
4.1.1 Deploying Manually . . . 25

4.1.2 Manual Configuration Management . . . 27
4.2 Breaking the Anti-patterns . . . 28
4.2.1 Automatization is the Key . . . 28
4.2.2 Acceptance Testing . . . 29
4.2.3 Higher Deliver Frequency . . . 29
4.2.4 Generalize the Infastructure . . . 29
4.3 Containers role in breaking of the anti-patterns . . . 30
4.3.1 Automation . . . 30
4.3.2 Generalization . . . 31
4.3.3 Delivery Frequency . . . 32
4.3.4 Acceptance-testing . . . 32

5 The Implementation Process 33
5.1 The Dummy-project . . . 33
5.2 Step 1. Infrastructure as Code . . . 35
5.2.1 Best Practices . . . 35
5.3 Step 2. Generalizing the Infrastructure . . . 36
5.3.1 Service Discovery . . . 36
5.3.2 Standarized Configuration Management . . . 37
5.3.3 Monitoring . . . 39
5.3.4 Data Management . . . 39
5.4 Step 3. Deployment Strategy . . . 40
5.4.1 Rolling out Releases . . . 40
5.4.2 Blue-Green Deployment . . . 42
5.5 Step 4. Constructing a Pipeline . . . 43
5.5.1 Integration . . . 43
5.5.2 Quality Testing . . . 43
5.5.3 Implementation . . . 44
5.6 Step 5. Return Feedback . . . 44
5.6.1 Pipevis . . . 45

6 Evaluation & Results 47
6.1 Cycle Time . . . 47
6.2 Feedback . . . 48
6.3 Quality . . . 48
6.4 Implementation Effort . . . 48

7 Discussion 51
7.1 Method & Evaluation . . . 51
7.2 The Solution . . . 52
7.2.1 Stability . . . 52
7.2.2 Alternatives . . . 52
7.2.3 Hiccups . . . 52
7.2.4 Containerized Configuration Management . . . 52
7.2.5 Change Control . . . 53
7.2.6 DevOps . . . 53
7.3 Continuous Delivery . . . 53

7.3.1 Project Management . . . 54
7.3.2 Continuous Everything . . . 54
7.4 Docker . . . 55
7.4.1 Impact on the Process . . . 55
7.4.2 Possible Problems . . . 55
7.4.3 Resource Usage . . . 55
7.4.4 What Docker really brings to the table . . . 56

8 Conclusions **57**
8.1 Continuous Delivery . . . 57
8.2 Obstacles preventing CD . . . 57
8.3 Containers impact on CD . . . 58

Bibliography **59**

Appendix A Fibonacci Sequence **63**

Appendix B Golden Ratio **65**

Appendix C Service Register Role **67**
C.1 tasks/main.yml . . . 67
C.2 defaults/main.yml . . . 67

Appendix D Service Deregister Role **69**
D.1 tasks/main.yml . . . 69

Appendix E Deploy Playbook **71**
E.1 vars/devservice.json . . . 71
E.2 deploy.playbook.yml . . . 71
E.3 rollback.playbook.yml . . . 72

Appendix F Healthcheck Container - HTTP **73**
F.1 check.j2 . . . 73
F.2 Dockerfile . . . 73

Chapter 1

Background and Introduction

This chapter presents the background story for this Master's thesis, as well as an overview of Continuous Delivery/Deployment, the concept of containerization along with the problem, related work and the approach on getting the result and answers to these questions. The purpose of this chapter is to provide a better understanding of the project in general and hopefully the following text will spark an interest in the project as a whole.

The next chapter in this report will present a technical deep dive into the container technology and give the user a perspective on why containers are important in this context. The third chapter will analyze how the project can utilize the container technology in order to achieve its goals. The fourth chapter will present the solution implemented using the previous presented methodology. The last part of this report will introduce a discussion and a conclusion dependent on the result and the problems stated in the first chapter.

1.1 100koll

The residents of Sweden may have stumbled upon E.ON´s 100koll[1] project before, and if not you may still have heard about the electric power company E.ON. 100koll is an additional customer-service that allows you to monitor your power consumption in real-time. This will give the customers a better understanding about where they use the most electric power and where they can save the most energy. It seems very simple in the commercials, "just plugin the power-meter and download the app, and you are good to go". What the consumer and common person may not know is the extent of the underlying technology and the number of companies involved in the system. One of the contributing companies is Data Ductus, and this Master's thesis is carried out in collaboration with them.

Data Ductus is responsible for knitting the whole E.ON project together, making it possible for the front-end application to communicate with the service providers, which in turn

[1] More about the 100koll - http://www.eon.se/privatkund/Produkter-och-priser/Elavtal/100Koll/Om-100koll/

communicates with the hardware. They are achieving this by utilizing their *crossbreed-container*[2] which essentially is an API[3]-hub. The API-hub makes it possible to collect a numerous of API´s and make them work as one to a front-end application, which in this case is the smartphone app. It is crucial for Data Ductus to keep the system up and running at all time, in order to keep the whole 100koll service going.

1.2 The Problem

In order to be able to meet the zero-downtime[4] requirement there needs to be an infrastructure that can fulfill it, which involves corner stones like load balancers, clusters and firewalls; which can be seen in figure 1.1.

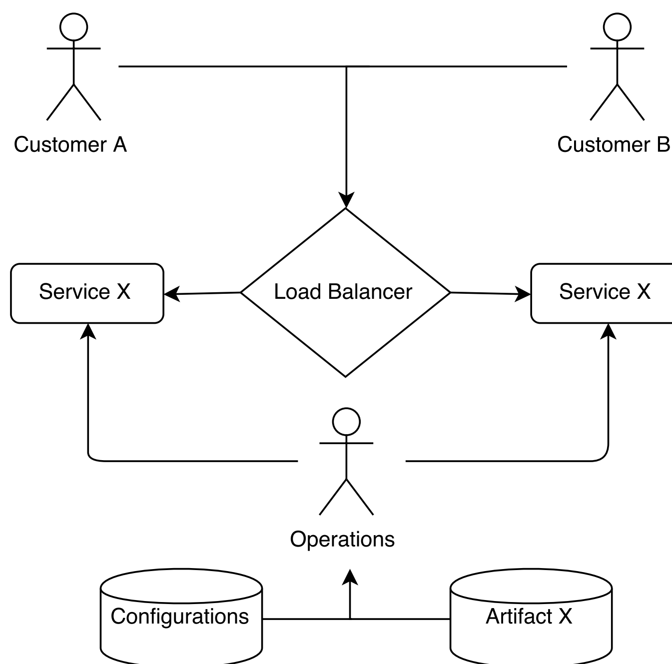


Figure 1.1: An overview of the infrastructure.

At this point all deployments and configurations of the infrastructure are done manually by the opererations-personnel. When new artifacts are ready for deployments the person in charge of operations receive these artifacts and the corresponding configurations from the build server. Then, by manually disconnecting one of the nodes from the load-balancer, a safe update procedure can be preformed without impacting the customer. This node is manually activated with the newly upgraded service, while the other nodes will be disconnected from the load-balancer, in order to perform the same update procedure. With the growing interest in containers, Data Ductus is requesting knowledge on whether or not containerization of the essential artifacts and configurations will help them

[2]More about Crossbreed - http://crossbreed.se/

[3]API stands for Application Programming Interface, which is a framework for communicating with a specific application.

[4]The service should never be down, or “under maintenance” during updates.

fulfill automatic deployment and staging to their infrastructure - that in the end will lead to continuous delivery. This thesis will hopefully provide sufficient knowledge on this matter, as well as propose a solid solution on how to achieve continuous delivery. Data Ductus are confident that a solid continuous delivery workflow will help them in their future work, and save them a lot of time.

1.3 What is Continuous Integration, Delivery and Deployment?

Continuous Delivery and Continuous Deployment are a relatively new terms for a very obvious and natural extension from Continuous Integration. So to be able to grasp the concept of continuous delivery you have to understand the basic concept of continuous integration.

1.3.1 Continuous Integration

Continuous Integration (CI) emerged shortly after the more agile and lean development processes were introduced into the software engineering market. Continuous Integration is a natural practice to utilize in the more lean and agile development teams. All the small integration steps are an essential part for keeping the process going. However to keep these integration steps continuous, some kind of automation needs to be done to compile and test new code. When you get these steps automated you can then integrate new code continuously, hence the name; Continuous Integration[5].

1.3.2 Continuous Delivery and Deployment

Although it seems very obvious to have some kind of deployment/delivery system extending your CI-system, the whole concept and principles were not realized on paper until David Farley and Jez Humble wrote the book “Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation”[8]. Continuous Delivery is the natural and final building stone in the integration workflow. By making not only the integration steps small but the deliveries as well, the product quality can be improved even further. However, to keep the delivery continuous there needs to be quality assurance steps before the initial deployment step. All these steps from new code to a deliverable product can be realized in a “deployment pipeline”[7]. “deployment pipeline” is a concept Farley and Humble discuss in their book as well. There have been a lot of confusion around the terms “delivery” and “deployment”, as they may be interpreted in various ways. In this master’s thesis they are defined as follows; Continuous Delivery is the same thing as Continuous Deployment except for the fact that Continuous Delivery makes a product delivery ready (deployable in the push of a button), but Continuous Deployment shoots the delivery out to production as well.

1.4 The Concept of Containerization

Workload and resource isolation have always been well-discussed subjects in computer science. By isolating dependencies and resources, the maintainability of a particular system can be dramatically improved, and in the end give a better overall quality of the system. In the age of “cloud” computing, systems are even more dependent on isolation. The required isolation and resource allocations make infrastructures heavily dependent on virtualization. The virtualization enables physical machines to act as multiple virtual machines[14]. These sub-machines can act as isolated instances, which includes dependencies and functionality that enables a service or services to operate. These isolated instances can then be transferred to different environments, independent of actual configurations surrounding this isolated instance. This isolated transfer principle will significantly simplify the processes towards continuous delivery, that are heavily dependent on the processes of getting artifacts from one environment to another.

When searching for a solid understanding as to why this topic is so important, Solomon Hykes talks about the “matrix of hell” in his introduction speech about Docker [9]. As can be seen in table 1.1, a set of different configurations can quickly escalate into an massive dependency matrix. Each question mark in table 1.1, is a combination of configurations specific for that environment and solution. Making sure the solution is persistent through the whole matrix requires a tremendous amount of tests, and even then the quality cannot be secured whenever a new set of configurations is about to arrive.

	Developers Laptop	QA environment	Single Production Server	Cluster	Public Cloud	Testers Laptop	Customers Servers
Static Website	?	?	?	?	?	?	?
Web Front-end	?	?	?	?	?	?	?
User DB	?	?	?	?	?	?	?
Analytic DB	?	?	?	?	?	?	?
Queue	?	?	?	?	?	?	?

Table 1.1: The Matrix of hell

The solution to this hazard is rather simple; containerization. By encapsulating the dependencies and configurations with the use containerization, it is possible to cover all combinations in the matrix described above with only one artifact; the container itself. As Hykes mentions, containers have been the obvious solution in the field of logistics, and makes up a unified solution which can be used by trains, trucks and boats. This unified container solution enables shipments from point A to B, without any specialized exceptions. The sender from point A is aware of the setup in point B, which enables B to use all the tools and systems to receive the goods without any hiccups. This concept can be

transferred into the binary world, by simply putting the artifacts into a container and then ship them to the system of your choice. If the container can be executed in your system it can be executed in the production servers and the stakeholders laptops alike, thanks to the unified container solution[1].

Virtual machines are another approach to solving the isolation problem, but there are some technical differences, which will be explored in Chapter 3.

Chapter 2
Method

This chapter will give the reader a better understanding of how the conclusions of this Master thesis were stated. By raising questions and providing them answers, the reader is given an insight on how containers will impact the processes towards continuous delivery, and in turn give valid conclusions.

2.1 Questions

Below is a list of questions that needs to be answered in order to be able to understand the containers impact on continuous delivery, and continuous delivery's overall impact on the project:

- What problems or blockers does Ductus have today that prevents them from achieving continuous delivery?
- How can we solve these issues with the use of containers?
- Will containers have the desired impact on the implementation process?
- How will continuous delivery impact the project?

The goal throughout this Master's thesis is to establish a discussion and conclusion based on the questions presented above. These questions will hopefully give the reader a better overview of this research, and acts as guidelines and goals during this study.

2.2 Approach

Reaching the answers to the questions raised in section 2.1, will call for a solid research approach. This section will present the way in which the research was conducted. The

hope is to provide the reader with a sufficient understanding of the approach to finding the answers to the questions raised. This research will use a qualitative approach. A qualitative research was chosen due to the fact that the problems at hand are somewhat specific for Data Ductus' case. A more quantitative approach, for instance with the use of polls, might provide a understanding of the need for continuous delivery, but this research is more of an investigation on how the use of containers will impact the process towards continuous delivery.

2.2.1 Pains and Blockers

The first step in this research process will pinpoint the hardships and blockades that stand in the way of Data Ductus achieving continuous delivery today. The bad practices that encourages these obstacles will from now on be referred to as the "anti-patterns". Observations and informal interviews have been performed with the developers and especially the personnel in charge of operations in order to find these anti-patterns. By physically being located among the people involved in the 100koll project provided a great opportunity to always get instantaneous feedback, as well as the possibility to observe how the developers run their day-to-day routines. Therefore, observations and informal interviews has been the main approach to this qualitative research.

2.2.2 Container-based Solution

When the bad practices or anti-patterns are identified a solution can be implemented, which will be purely based on containers. Implementing a solution based on containers will give the evaluation a better correlation with containers true impact on continuous delivery. The continuous delivery solution will be implemented with the use of a "dummy-project". This dummy-project is a trivial implementation of a fibonacci sequence solver. The architectural components that builds up this dummy-project will be depict the 100koll project as close as possible. Achieving continuous delivery with the use of the dummy-project will aid the adaptation to the 100koll project and other projects alike. The use of this dummy-project will provide fair conclusions on how the 100koll project will be impacted in reality, without having to learn the true workings of 100koll.

2.2.3 Solution Evaluation

To be able to get a solid evaluation of the container-based solution there needs to be some qualitative research before and after the solution is introduced into the agile workflow. These metrics are stated by interviewing operations personnel, sysadmins[1] and by a deeper analysis of the current workflow. By conducting a deeper evaluation of the solution and how it can eventually impact the 100koll project will hopefully give this research valid and solid metrics in order to find the answers to the questions raised in section 2.1.

[1] System administrators - The persons maintaining the systems in the infrastructure

Cycle Time

One of the most important metrics in a software delivery process is the cycle time. The cycle time refers to the time it takes for a new request to be realized and delivered to the customer. For instance, the point in time were a new feature has been requested to the point in time when the feature is actually available to the customer [8]. This metric can be hard to measure due to the fact that it spans over the whole delivery process and involves many steps, which makes it a very purposive metric, which in turn makes it a strong candidate for measuring the success rate of the solution. The value of this metric will be settled based on these two questions:

- How fast can a bug-fix or other “quick-fixes” be delivered to the customer?
- How much time is spent on staging and deploying new releases during an iteration?

Feedback

Feedback is the heart of the software delivery process [8]. It is essential for the developers to get integration feedback from their newly commit code. It is also important to archive all changes and reports accordingly to get a good overview of the project success rate. This is usually done by utilizing so called continuous integration servers like Jenkins CI.

To get even better feedback to the developer (or any involved stakeholders for that matter), the use of “information radiators” is not uncommon. These radiators are hard-to-avoid manners which usually consists of a big monitor or a poster displaying the status of the project.

This metric will be based on how fast developers can get feedback and on how well they are informed. The feedback loop will first be measured by observing how the developers get feedback today, and the overall mindset against it. This measurement will then be compared with the potential feedback improvements gained by the new feedback system. The new feedback system will be measured and evaluated by observing and recording first impressions and reactions from the developers when first introduced to the new system.

Quality

Unlocking the complete set of benefits from continuous delivery is crucial in maintaining the quality of the product. How the newly committed code retain the expected quality needs to be analyzed and reported back to the developers in the feedback cycle. The ease of discovering quality defects are therefore an important metric in order to be able to release highly quality releases.

This metric will be measured by implementing the same functionality in two different ways; one less efficient way (in form of execution time) and one in a more efficient way. By making two solutions that cover the same functionality, the less efficient way of solving the problem will hopefully be discovered with the use of quality testing, which in the end gives us a sufficient measurement of the metric. This will be done by implementing two ways of solving a trivial math problem, like a fibonacci sequence calculation. The less efficient way will then hopefully be discovered by introducing a threshold into the quality testing in the deployment pipeline. The threshold results in the less efficient way lagging behind its more efficient counterpart, due to the fact that the lesser alternative cannot make

it over the threshold. The result of this measurement will then be compared by the quality assurance measurements performed in today's delivery workflow.

2.3 Related Work

This Master's thesis will be heavily dependent on Jez Humble and David Farley's book about continuous delivery[8]. Both Humble and Farley have got masses of experience in this area and decided, in 2010, to publish a book about their knowledge on how to solve the problem of achieving continuous delivery. Introducing containers into the Farley and Humble guidelines and by observing the way in which they impact the processes toward continuous delivery, it is possible to get a better understanding on how they really impact the integration workflow, and give us the desired conclusions.

To get a better understanding on how containers work and how they can impact the infrastructure, Wes Felter, Alexandre Ferreira, Ram Rajamony and Juan Rubio have written an IBM report which will be used [16]. In the year 2014 they updated their research about containers to use Docker as the default container framework. This project also heavily revolves around Docker, and because of this the IBM-report is the perfect reference with which to touch base.

Other companies like Oracle have, by introducing the concept of continuous delivery, greatly improved quality and business value of their products[13]. However, it did come at a certain price, but in the end these costs can be exempt due to the opportunities gained. This, among other studies[4], display the potential value and gain of continuous delivery. However, continuous delivery may come with some challenges (which are described throughout the remainder of this thesis report). This Master's thesis study will hopefully be valuable addition to the research involving the processes towards continuous delivery and the true value gained from its introduction.

Chapter 3
Containers

This section takes a deeper dive into the technical aspects of containers, and give the reader a better understanding on how containers can be an alternative to virtual machines in order to provide isolation.

3.1 Underlying Technology

The basic principle of container usage is to get the same benefits as virtual machines but without unnecessary overheads. This is achieved by isolation on the operating system level layer instead of virtualizing the hardware and operating system kernel. This allows for the wanted level of isolation. As can be seen in figure 3.1 the isolation is closer to the operating system, which turns the hypervisor and the guest operating system into unnecessary footprints. The removal of the extra layering can also be reflected on the performance which will be discussed in section 3.3.

In order to achieve the isolation between the operating system and the top layer as shown in figure 3.1, functionality in the Linux kernel is used; namely namespaces and cgroups. These Linux kernel functionalities are then usually wrapped into a more user-friendly framework which will be discussed in section 3.2.

3.1.1 Namespaces

The duty of the namespaces is to give a process different view of the system than other processes. Using six different types of namespaces gives the system the possibility to wrap a particular global system resource in an abstraction, which makes the process within those namespaces act as if it has its own isolated instance of the systems global resources [10]. These six namespaces can be shown in the list below:

Mount namespaces Mounting namespaces was the first namespace that was implemented

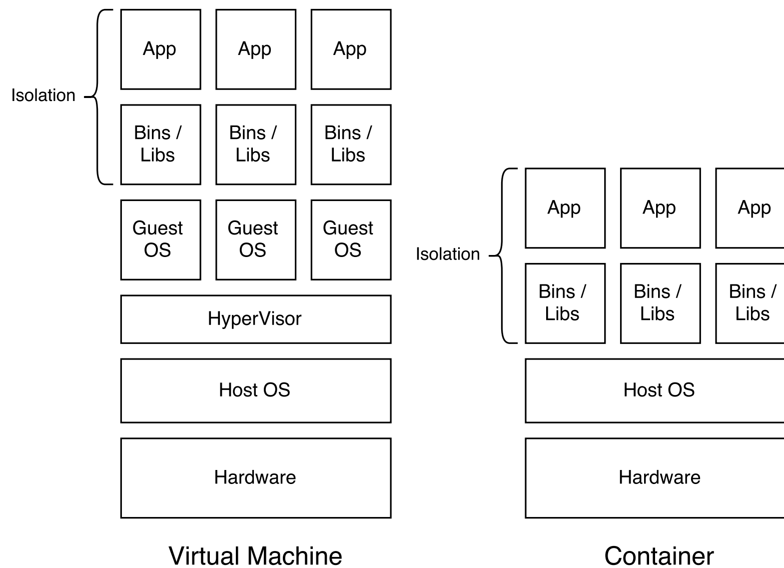


Figure 3.1: Architectural differences between virtual machines and containers.

and shipped with the Linux kernel 2.4.19[10]. This namespace allows a group of processes to see just a specific set of the filesystem mount points by isolating them from the rest, which makes it possible for a processes (or multiple processes) within that namespace to have a whole different view of the filesystem hierarchy.

UTS namespaces First appeared in Linux kernel 2.6.19[10], allows isolation of two system identifiers; the domain and node-name. This namespace can for instance make it possible for a container to have its own hostname.

IPC namespaces Along with UTS namespaces, IPC namespaces were implemented. IPC namespaces makes it possible to isolate certain interprocess communication resources. In the Linux kernel, System V IPC was used until 2.6.30, when it was later replaced by POSIX message queues. IPC (Inter-process communication) allows processes to exchange data between eachother.

PID namespaces PID namespaces arrived in Linux kernel 2.6.24[10], which makes it possible to isolate the process ID number space. This means, in practice, that the same PID can exist in different namespaces.

Network namespaces Network namespaces have existed since 2.6.24[10], but was not fully completed until about the time of the release of Linux 2.6.29. This namespace makes it possible to isolate all system resources associated with networking; devices, IP addresses, IP routing tables and ports etc..

User namespaces User namespaces is the last namespace that has been implemented in the Linux kernel. It started to occur as early as 2.6.23[10], but was not fully completed until 3.8. This namespace makes it possible to isolate User and group ids. This means that a process inside this namespace can have full root privileges for operations but is unprivileged for operations outside the namespace.

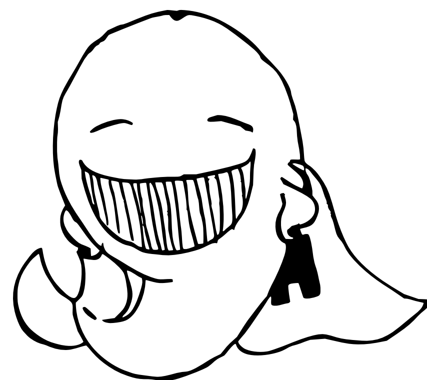
3.1.2 Cgroups

Cgroups (Control Groups) is another important feature in the Linux kernel. Cgroups are used for limiting, account for and isolate resource usage; CPU, Memory, Disk I/O and networking, among others. This gives the host the ability to constrain containers resource usage. Cgroups provides functionality for separating certain sets of tasks into hierarchical groups with specialized behavior from other hierarchical groups. For instance by limiting the resource usage for an apache server, apache and all apache's child processes will not exceed that resource limitation, apaches hierarchical group will not overrun another hierarchical group. Cgroups are therefore a fundamental corner-stone for isolating containers from each other.[11]

3.2 Docker

As mentioned in section 3.1 containers relies on a variety of Linux kernel functionalities which are usually wrapped into a user-friendly environment. One of those "frameworks" is Docker[6]. Docker utilizes a pre-built execution environment. Until version 0.9 Docker used LXC as the standard execution environment. LXC was later dropped and switched to libcontainer[15]. Libcontainer is considered to be a tighter integration with the Docker framework, hence the execution environment is developed by the community itself, as can be seen in figure 3.3.

Docker acts as an additional layer of abstraction for the container execution environment by utilizing a server-client architecture. The core building-stone of the Docker framework is the Docker Engine. The engine is basically a daemon running on your operating system that manages containers with the use of an execution environment. This engine can in turn be controlled by an API (Application Programming Interface), which are most commonly used by the Docker CLI (Command Line Interface). The Docker engine allows the operating system, for instance, to "link" containers or expose ports through the Docker Network interface[6], and the fundamental parts as starting, stopping and killing containers.



Our hero, the Docker Whale!

Docker utilizes a layering architecture in order to simplify the process of moving containers across different hosting environment. By creating images (read-only containers) and saving them using the AUFS (advanced multi layered unification filesystem), the footprint of these images can be significant minimized. This is due to the fact that AUFS utilizes the "copy-on-write" principle, which means that one set of information only exists once, and the top layers inherits from the same footprint. For instance if you have two images based on the same Linux distribution but two different web hosting systems, for example nginx and apache. These two images will point to the same base layer (which only exists once), and then put another layer on top of that.

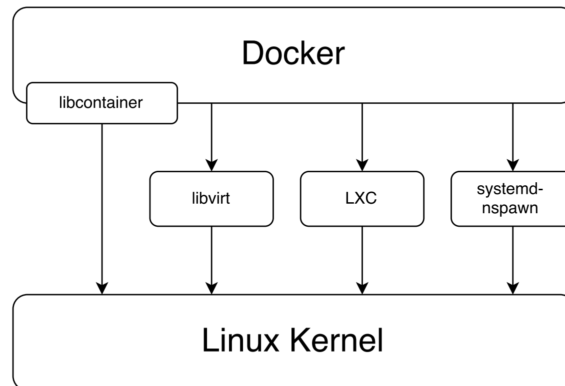


Figure 3.3: Architecture overview of Docker and its interfaces.

NAT (Network Address Translation) are also used by Docker to make it even more user friendly. By creating a separated network interface for Docker containers, the traffic can be bridged and give the user even more control.

3.3 Performance

As mentioned in the introduction to this chapter, containers are considered an alternative to virtual machines. With the increasing interest in “cloud” computing, virtualization has become a crucial factor in order to squeeze as much functionality as possible into todays powerful hardware. This is directly reflected on resource utilization and budgets, wanting companies to move their IT infrastructure into the “cloud”. As mentioned in the article by IBM [16], the trade-off for virtualization has always been a factor to consider, due to the fact that it directly affects the computing resource budget. By making the virtualization on a operating system level, in theory the trade-off will be less. In the article, IBM is stacking up containers (docker) against virtual machines (KVM) to see which technology got the best trade-off by comparing it to native performance.

3.3.1 CPU and Memory

The benchmarks during IBM’s research were conducted using numerous different techniques. CPU performance was measured by utilizing PXZ, which are a lossless data compression method using the LZMA algorithm. The result showed that Docker out performed KVM by 12%, and was very similar to the native performance[16]. HPC (High-performance computing) was measured using Linpack. Linpack performs LU factorization with partial pivoting in order to solve a dense system of linear equations. In this test Docker performed almost identical to the native with a 17% better performance from KVM[16]. However when KVM was fine tuned it performed only at 2% less[16]. The STREAM benchmark was utilized to measure the Memory bandwidth. STREAM makes simple operations on vectors in order to measures sustainable memory bandwidth. In this case Docker performed as the other tests, almost identical to the native. KVM had lost an average of 2,25% in performance, which is slightly less[16]. The benchmarking on the memory was executed by RandomAccess, which stresses the memory subsystem in a

regular manner, with a set of memory operations. This test showed almost no significant difference from the native results for both KVM and Docker. Docker performed 2% less and KVM 1%[16].

By calculating the average loss in performance relative to native performance, the result showed that Docker benchmarked 0,86% performance loss and KVM (tuned) 4,29%[16], a difference by 3,43%.

3.3.2 Networking

As mentioned in section 3.2, Docker uses a bridge which is connected to the network via NAT. IBM benchmarked Docker both with the NAT network setup and the host interface directly. To minimize virtualization overhead as much as possible while using KVM, virtio was used on the guest OS and vhost on the host. The first measurement was done by using nuttcp, which is a tool for measuring network bandwidth. The result from nuttcp showed that during transmission, Docker containers utilizing NAT gives an noticeable overhead, while containers using the host interface performs almost identical to the native. KVM performed significantly better than Docker using NAT during transmission. However while receiving data, Docker using NAT out-performed KVM[16].

The network latency was then analyzed using netperf. The result showed that Docker using NAT almost doubled the latency, while KVM adds a less overhead compared to the native[16].

3.3.3 Disk I/O

IBM utilized fio to analyze the Disk I/O performance drop for KVM and Docker. As described in section 3.2, Docker uses the AUFS filesystem to achieve the "copy-on-write" functionality. However this can be neglected by mounting the host volume into the container, which IBM did during the benchmarks. KVM however adds an extra layer, namely QEMU. The result shows that Docker (while mounted volume) performs as expected, identical to the native[16]. The IOPS (Input/Output Operations Per Second) result showed KVM could only withstand half of IOPS compared to Docker and the native[16]. All I/O operations done with the KVM setup had to go through QEMU. However, the KVM throughput performance was almost identical to native and Docker, but the extra layering of KVM needs to make more use of CPU cycles per I/O operation, which will be directly reflected on how much CPU that are available for the application.

3.3.4 Summary

The report from IBM showed that containers perform equally or better than VM's in almost all cases[16]. The overhead for CPU and memory performance was almost negligible, and networking and disk i/o should be used carefully. By letting Docker container utilize the host interface, the overhead introduced by bridging the traffic via NAT can be avoided. The AUFS filesystem introduces overheads as well[3], but can be avoided by mounting the host volume into the container. By introducing these two configurations a lot of overhead can be avoided while virtualizing with containers. However, with virtual machines utilizing hardware virtualization, it is impossible to avoid the extra layers that comes with them.

Chapter 4
Taking the Step

The title of this chapter "taking the step" refers to the leap from continuous integration to continuous delivery, by breaking the unproductive and bad practices around infrastructure and deployments, so called anti-patterns. This thesis assumes that there is already a working continuous integration workflow in place, and therefor the thesis is focusing on the step from integration to delivery. This last and somewhat fundamental part in the whole agile workflow is often left out because it is usually considered to be a hard and resource heavy component to achieve. This chapter will present the anti-patterns and the pitfalls for achieving continuous delivery and how to take the step from continuous integration to delivery by breaking these anti-patterns. The objective is to make the leap from continuous integration to continuous delivery a lot shorter by introducing containers. This will in turn make the breaking of the anti-patterns a lot more appealing.

The first section will present the anti-patterns Data Ductus are currently conducting. The second section is a way of breaking it and the last section will present the benefits with continuous delivery.

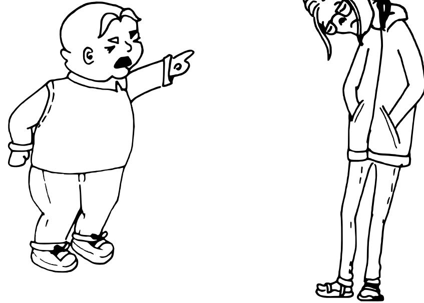
4.1 The Anti-patterns

This section will present the anti-patterns that Data Ductus is currently conducting with their 100koll project, which prevents them from achieving continuous delivery. However these patterns can be considered one of the most common patterns and can probably be related to a lot of other projects. This pattern does not take into account the integration-step as described in the previous section.

4.1.1 Deploying Manually

Deploying manually is one of the anti-patterns. Deploying new releases are usually considered painful, repetitive and really boring: boring to the point that developers takes

turns to deploy artifacts to the development environment or even use the "if you need to test your changes, deploy them yourself" practice. When the big release day is scheduled, it is usually associated with fear. The anxiety builds up inside when you realize that the last time a new release went into production you had this huge bug that did not reveal itself until the system was used by the customers. The development team needed to make a quick-fix while the ops-team in panic manually rolled back the system. When the release was finally ready for another rollout the clock was already 1am at night. This made a tired ops-person publishing the wrong artifact. The developers think they did not solve the problem and tries to figure out why their quick-fix did not work. A couple of intense hours later they realized that the ops-team did just deploy the wrong artifact to the production and everything escalates into CHAOS. This might be the worst case scenario, but a lot of people might be able to relate to it in some way.



Angry developer blaming operations personnel.

Deploying, and especially deploying to production, involves a lot of moving parts, and by performing these tasks manually the risk of failing increases considerably. The probability of a human error increases as the process gets more and more intense. This list will hopefully give a better understanding of what can go wrong and to give a better visualization of different failing scenarios.

- If the deployment-step is performed manually there is a probability that errors will occur. These errors might be so insignificant that they are hard to track down and will not appear until days later.

- Because the deployment is done manually it will probably not be repeatable which can give sporadic errors that leads to a lot of time being wasted on debugging deployment errors.

- Manual deployment-procedures need to be documented in some way. This means that every time a deployment routine changes the documentation needs to be updated as well. This might be time-consuming and will even come to the point where documentation is way to out-dated when it needs to be used.

- The deployments are usually (and hopefully) done by some deployment-expert, and when the expert is not available there cannot be any deployments done.

- Performing deployments manually is repetitive and boring, yet it requires, as mentioned, some degree of expertise to perform them. Asking people about doing boring yet technically demanding tasks is a certain way into human errors. Especially when the expert is having a bad day.

- You can only test a manual process by doing it, which is often time-consuming and expensive.

- The fact that deploying manually to a development environment gives the possibility that the environment is out of sync. Even though the artifacts are deployed based on the central repository and not the developers local source-code. This might occur when developers misunderstand each other or if a developer forgets to deploy his/her new changes to the development environment.

These are just a few examples on the many things that may go wrong.

4.1.2 Manual Configuration Management

There is a possibility that configurations of the infrastructure are done manually by the operations-personnel and sysadmins. This is the case for Data Ductus; for instance: if there needs to be some security reconfiguration in some third-pary library, the sysadmin needs to do these tasks on all these nodes individually, which can be very time consuming. To get a better understanding why manual configuration of the infrastructure is considered an anti-pattern, a small list were things can go wrong will be presented below:

- Even though having deployed new releases to staging successfully many times the service might not work in the production, and there is no way of knowing why.

- Nodes in a cluster might get a different behavior which might lead to an inconsistent and misbehaving system.

- The operation team needs to perform intense and techical-demanding tasks just to get a environment ready for deployment.

- With the lack of version controlled automated configuration management it is hard to step back to a previous configuration of the system or infrastructure, the sysadmin might not even know how he or she got the configuration that is in place today.

- Bugs might occur because there is an unintentional mismatch between nodes in some configuration.

- Configuration of the system might be performed by modifying the machines directly which give the possibility to mess something else up.

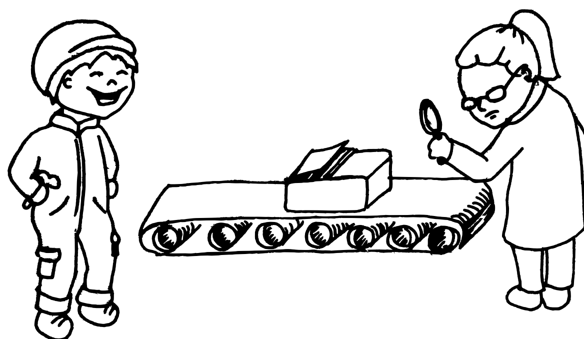
As you can imagine keeping a system running by manually configuring a cluster of machines is really demanding and time-consuming, and human errors are waiting to happen. Configuration mistakes can even be transferred to the development team, for instance; some load-balancer had a corrupt node in its pool and the developers just deployed some new artifact to the environment. They now think their code messed something up but the fact is that there is only some misconfigurations in a configuration-file on the load-balancing node.

4.2 Breaking the Anti-patterns

Breaking these anti-patterns are usually considered to be hard and resource heavy. Conducting these anti-patterns seems safer and a better solution for now. This section will try to point out the fundamental building-stones towards continuous delivery by breaking the anti-patterns with the use of containers. These container-based solution is however just a theoretical assumption and is to be tested and evaluated in a later stage by implementing a solution based on containers and Farley and Humbles principles[8].

4.2.1 Automatization is the Key

Automatization is the key element in achieving continuous delivery. Almost everything can be automated. It makes a lot of sense to automate demanding and repetitive tasks in a real manufacturing process, but when it comes to computer technology it is often omitted: Even a simple task such as transferring a file to the same folder every morning, or starting that backup every time you are about to go home from work. Two minutes a day becomes 10 minutes a week, 10 minutes a week becomes 40 minutes a month, 40 minutes a month becomes 480 minutes a year. Which means that if you had a little one-liner script that did this task for you everyday for a year you could save yourself a whole workday's worth of time.



Satisfied developer and tester

In order to achieve continuous delivery you need to make your software ready to be in a deliverable state by the use of some "deployment pipeline". This pipeline is dependent on chained automated tasks in order to construct a deliverable software into production. More about deployment pipelines and a implementation will be presented in chapter 5, where automated solutions will be presented along with it. However this section will try to point out which benefits you gain from automating your repetitive tasks:

- By automating a repetitive task, it is easier to find bugs and errors. Repeating a task exactly the same way multiple times will give, with a high probability, the same errors.
- Your scripts are your documentation. There is no need for keeping a document up-to date, because if your scripts are not up-to date the atomization will not work.
- Automated tasks are easier to test and these tests are cheaper to perform.
- If your scripts have successfully done a task 100 times it will probably work the 101:s time.

The scripts and automatization techniques do not always comes with just benefits. It is important to keep the scripts standardized in order to adopt to different infrastructures and

projects. Follow the principle of DRY (do not repeat yourself) during scripting towards automation is also important, to avoid the double maintenance problem. By making the deployments and configurations dependent on standarized scripts creates the need to keep these scripts updated during system changes.

4.2.2 Acceptance Testing

Another factor that keeps development-teams stuck in this anti-pattern might be that they feel a lot more comfortable by deploying manually, because then they feel like they have more control of the artifacts that goings into production. However, by utilizing good acceptance testing there is no need to worry about bad artifacts. Acceptance tests are there to make your team enough comfortable to take the step to deploy something new into production. Here are some key benefits by introducing good acceptance testing:

- Your release will always be ready for production. If the acceptance tests fail the release fails.
- It is better to discover that you have a quality problem before you release something into production then after.
- Computers are better to test the quality of a product by removing the human errors. Computers can follow strict testing routines that should not make any release slip through.

It is important however to make relevant acceptance tests, that confirm the quality of the product. If the acceptance tests wont be able to confirm the product quality, pure releases might slip through the pipeline.

4.2.3 Higher Deliver Frequency

Integrating new code into a project is usually a very painful process. The concept of agile and lean development is to make the processes that hurt more frequent to bring the pain forward, instead of make a huge mega integration at the end. So why do not take this principle a step further and practice it in the delivery state as well. By making frequent deliveries you are able to improve the quality of your product. Realizing that the quality requirements are not met as early as possible is an important factor, and a good way to find out quality flaws is to deploy your service into the QA-environment for testing. Instead of discover that the whole month of integration of new codes messes your quality up when it is finally ready for production. By developing good automated deployment strategies will make sure the deployment processes to the production will not mess up the production environment.

4.2.4 Generalize the Infastructure

The key benefit of keeping an infrastructure generalized is that we can keep our environments increasingly similar. The development environment can be similar to the quality

assurance environment and the quality assurance environment to the production environment. By keeping the differences at a minimum the production environment will behave some what like our development environment. This means that there is no need for specialized scripts for a specific environment purpose and we can keep and maintain just one script that works on all environments.

By keeping the scripts as general as possible gives a better proof and confidence that if it works for the development environment we know it will work with high probability in production as well.

If the configuration management is automated with the use of configuration management tools as well, the benefits can be even greater. By keeping configuration management automated makes it possible to construct dynamic load-balancers, scalability and a better way of maintaining the infrastructure. Basically you administrate your infrastructure by pushing configurations to a central repository and the system solves the rest. This makes it easier to debug the system and you always know how the latest revision of configurations looked like and makes it possible to rollback at any point in time.

4.3 Containers role in breaking of the anti-patterns

The goal of this Master's thesis is to, with the use of containers and Humbles and Farleys guidelines, break the current anti-patterns conducted at Data Ductus, in order to achieve continuos delivery. The problems determined and described in section 4.1, can be broken with the practices described in section 4.2.

An overview of how containers, and Docker especially, can be used with the current workflow to facilitate the breakage of these anti-patterns can be seen in Figure 4.3. By encapsulating the configurations, dependencies and compiled code into one artifact, a container (Docker) image, simplifies the deployment and staging processes significantly. The scripts and configurations used during deployments and staging can be encapsulated into a container (Docker) image as well. These images can then be utilized for staging and deployment in any environment, for instance on the operations personnels laptop, a continuous integration server or even a customer.

How to achieve such a solution will be described into further detail with the five steps of implementation in chapter 5. The solution described in chapter 5 is purely container based. This chapter will lay down the benefits gained by utilizing containers to break these anti-patterns. These benefits are then realized into a solution described in the later chapter 5.

4.3.1 Automation

By utilizing the concept of containerization, the automated tasks will be less painful. This list will present some examples where containers can help the automation process.

- You only need to compile once, due to the fact the binaries and configurations in a container is the same for all containers that are created from the same image. You

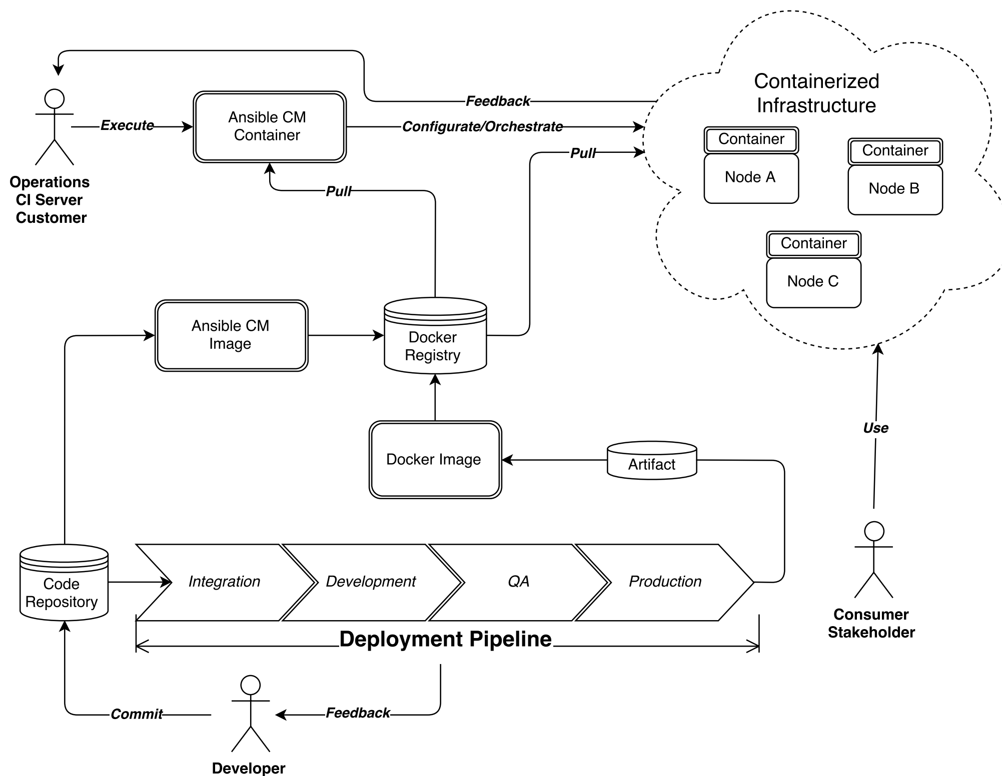


Figure 4.3: Continuous Delivery with the use of Containers

compile once, put the binaries and configurations into an image and then you are certain that this image will work the same on all machines.

- The container framework is the core functionality in the automation process. The framework lets you start, stop and kill containers on demand within seconds, which simplifies the automation process due to the fact that there are less moving parts involved.

Containers are a virtualization technique which are heavily isolated instance. It is important to keep this in mind due to the fact that this can make the automation processes even harder then without them, especially when containers needs to talk to each other. By strictly use automated configuration management this can however be avoided or be simplified.

4.3.2 Generalization

The fundamental building stone of a general infrastructure is the use of isolation. This gives the possibility to utilize the whole machine and give the machine different purposes at the same time. One machine can even be used for Development, QA, and Production at the same time. Isolation can also be achieved by the use of virtual machines, see chapter 3. However, the fact that containers are less resource-heavy makes it possible to squeeze in even more isolation into one machine.

As described in chapter 3, containers are faster as well, which can be beneficial when the configuration management should be automized. It is easy and fast to scale your cluster, dynamically load balance to different nodes and to restart and debug containers.

Another great benefit with containers is that you can isolate the configurations and containerize them specifically for the purpose of that container. For instance; you can have 10 different services with 10 different versions and 10 different configurations on one machine. This would also be possible with virtual machines, however, it would take a lot more resources due to the fact that you need the hypervisor, OS and everything else that sets the base for the virtual machine. By using containers you use the same footprint for all 10 containers, the only difference is the footprint from the different versions and configurations.

By utilizing a complete framework like Docker, the processes of configuring containers to work together can also be easily achieved. You can for instance "link" containers together in order to ease the communication between them.[6]

4.3.3 Delivery Frequency

As mentioned containers will in theory bring down the automation steps and make them a lot more simpler. This will have direct impact on the frequency that a new software are able to be delivered. By reducing the steps and make them simpler the whole process are less likely to fail, therefore the frequency can be increased.

Containers simple and lightweight framework makes them very easy to transfer, deploy and to start. Starting a container is a matter of seconds instead of minutes.

4.3.4 Acceptance-testing

By utilizing the fact that containers are prepared once and able to be executed anywhere, you are able to run your acceptance-test anywhere as well. You can even run your functional acceptance test directly on your CI-server. This makes the whole automation a lot more simpler. You do not need to worry about how your binaries are configured on other machines, because all the configurations are already wrapped into your containers. The fact that all configuration and dependencies are containerized, gives a higher probability that it will work on another environment with minor configurations.

Chapter 5

The Implementation Process

As mentioned in section 1.3.2, continuous delivery is about being able to deliver a quality product continuously to the customer. There needs to be various techniques established in order to achieve this goal. This masters thesis will be separating the implementation of the continuous delivery workflow into five fundamental steps. The first two steps is about the ability to get a infrastructure as code, the third to utilize the programmable infrastructure to perform deployments and the last two is about construct the deployment pipeline. As the solution may vary from project to project, these steps will hopefully help the reader in achieving a solution using these guidelines.

In order to get an adequate evaluation of the solution a "dummy-project" has been implemented. This project contains parts that are required in order to test the workflow, adopted to fit the attributes of the 100koll project. However, the solution and project is standardized and it will hopefully be possible to adapt it in accordance with other projects as well.

5.1 The Dummy-project

The Dummy-project is a RESTful web service implemented using Java. The project consists of a backend-part and a simple front-end part. In order to be able to utilize the whole spectrum of the delivery pipeline; functional (blackbox-testing) and stress-testing using Jmeter have been implemented, along with unit-tests and test-coverage. Maven is the standard building script for Java-based projects at Data Ductus, and because of this, Maven is used as the default building script in this instance as well. The Jersey-framework (based on JAX-RS) have been utilized in order to simplify the process of implementing a simple RESTful web service.

The "dummy-project" is a simple web service that helps a customer to calculate the golden ratio based on a fibonacci sequence. The sequence can be seen in figure 5.1.

$$f(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ f(n-1) + f(n-2) & \text{otherwise} \end{cases}$$

The fibonacci sequence have been chosen because the trivial implementation, and the possibility of implementing it in two different quality ways. The first and most obvious way of solving this function is by implementing a simple recursive call, as can be seen in Appendix A. However this function tends by very inefficient, due to the fact that the recursive calls gets exponentially worse with larger n. This can be avoided by "caching" the previous calculated sequences, which means a fibonacci number will only be calculated once. In this way we can avoid our big O notation to involve any exponent and make the complexity linear. The "cache" based solution can be seen in Appendix A, as well. To make it more convenient and to be able to easily switch between the two solutions, a simple interface has been created which the different implementations can inherit, as can be seen in figure 5.1.

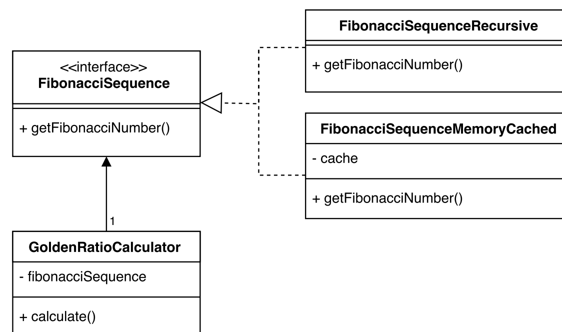


Figure 5.1: UML diagram over the logic implementation of the web service.

The ratio of consecutive Fibonacci numbers are called the golden ratio, which can be seen in figure 5.1. The source code can be seen in appendix B.

$$\lim_{x \to \infty} \frac{F_{n+1}}{F_n} = \varphi$$

Unit tests have been implemented using jUnit for the source code in appendix 5.1 and B. The golden ratio can then be calculated using these fibonacci numbers, by a requesting-client through the RESTful service. The functional testing or blackbox testing and the stress testing is performed by jMeter using the RESTful service as well.

Maven triggers all these types of tests along with the thresholds; one threshold for unit tests source coverage, one for functional and one for the stress-testing. These thresholds sets the requirements for newly integrated code, as the code needs to pass all these test through the deployment pipeline.

5.2 Step 1. Infrastructure as Code

Infrastructure as code or programmable infrastructure, is the art of writing code to manage configuration management and provisioning of an infrastructure. Not to be associated with infrastructure automation, which involves the technique of automating repetitive tasks in the infrastructure, infrastructure as code is the principle of maintaining a certain state of the infrastructure. So called configuration management tools are utilized in order to achieve a programmable infrastructure. These tools are all based on the some princible; describe the infrastructures state with the use of high-level or scripting languages, and let the machines configure them self to that given state.

Ansible[1] have been chosen to be the default configuration management tool in this Master's thesis, due to the fact that Ansible is completely agent-less. Agent-less architecture do not require a certain initial state of the infrastructure in order to make the automatization work. Most configuration management tools like Puppet[2] utilize an agent based architecture, which means that the tool requires a "master-node" to synchronize configurations from. Configuration management tools based on agent-less architecture like Ansible, is only dependent on ssh by "pushing" configurations out to the infrastructure instead of letting agents synchronize its configurations. The "pushing" functionality simplifies integration of deployment strategies especially with the use of Docker without any additional implementations.

5.2.1 Best Practices

As mentioned the task for configuration management tools is to maintain and upgrade a certain state of the infrastructure. In order to better grasp the concept of programmable configuration management, metaphors are usually introduced to describe the programming discipline and structure. In the case of Ansible, these metaphors are: roles, tasks, inventory and playbooks.

A certain machine in the infrastructure can have one or multiple roles. These roles contains tasks, which are state-based synchronized executions decribed using YAML[3]. For example; a machine can have the role as a "webserver". The "webserver" role can then contain tasks as following; "make sure you have apache version x" and "make sure your port 80 is open". In this way certain manifests for the infrastructure can be constructed with an mixture of roles and underlaying tasks. These roles can then be delegated to a certain machine or a group of machines by running a playbook, which contains a set of "plays". For instance; there can exist a playbook that describes your base state of the infrastructure. Running this base state playbook periodically, makes it possible for the infrastructure maintain its state, and a situation were the infrastructure is out of its state can be reduced. The machine groups in these plays can be described in inventories. These inventories can either be dynamically constructed using scripts or just described statically. For instance there can be an inventory group called "webservers" which is associated in the playbook with the role "webserver". Examples of roles can be seen in Appendix C and D.

[1] Ansible Configuration Management Tool - http://www.ansible.com/

[2] Puppet Configuration Management Tool - https://puppetlabs.com/

[3] YAML Ain't Markup Language - http://yaml.org/

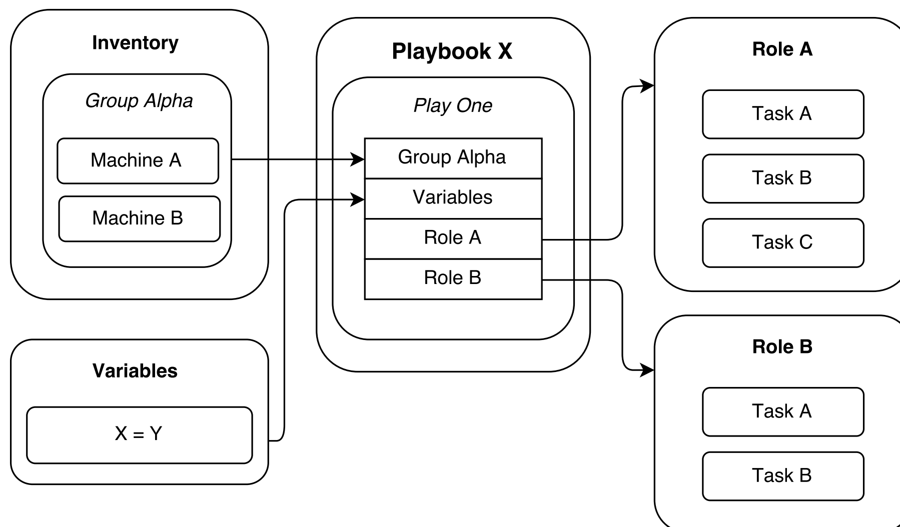


Figure 5.2: Graph of how Ansible can be structured.

As can be seen in figure 5.2, additional variables can be set in the context of plays, roles and inventory groups. By following the structure principle described in 5.2, gives the possibility to describe dynamic plays which can be composed into deployment routines. By keeping the roles and tasks general, describing the attributes with the use of variables and inventories, makes it even more flexible to create a generalized infrastructure as described in section 5.3.

In combination with the use of version control, configuration replications and rollbacks from different version our infrastructure can be constructed. As described in chapter 5.3.2, these Ansible scripts can act as an documentation for the infrastructure, which simplifies the maintainability considerably.

5.3 Step 2. Generalizing the Infrastructure

Keeping the infrastructure generalized will entails its benefits, as mentioned in section 4.2.4. This might be considered a hard task, but with the use of containers the problem will be significantly less palpable. However, as mentioned in section 4.2.4, the use of containers in the infrastructure will conceive problems. The fact that containers is such an isolated instance makes them not aware of its surroundings. This section will explain that with the use of service discovery and standardized roles solves this problem. As can imagine this step requires that the previous step is achieved, it is crucial to follow the best practices as described in section 5.2.1, in order to simplify this step as much as possible.

5.3.1 Service Discovery

By interpreting an instance of a process, which task is to solve a problem (for instance a customers need) through a specific port, as an service, the use of service discovery can be applied. Service discovery is a collection of protocols, which task is to always know were these services are located on which state they are in. By introducing service discovery in

a cluster of machines will give these machines the awareness of its surroundings in the infrastructure and the services within it.

In this Master's thesis Consul[4] was selected as the default service discovery tool. There are a a lot of different tools available that help the infrastructure achieve service awareness. However Consul was chosen because the key/value store functionality and the simplicity. The key/value store influence on the infrastructure will be explained in more detail in section 5.4.

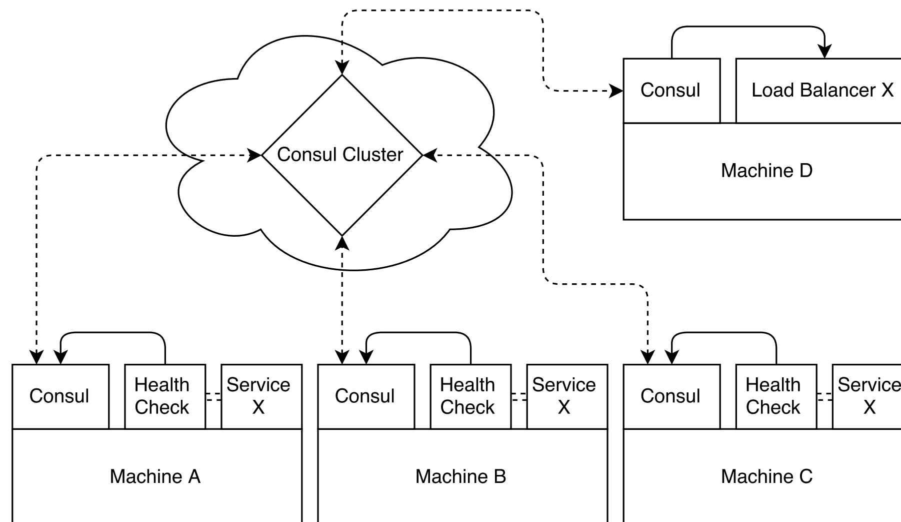


Figure 5.3: Infrastructure overview using Consul.

Figure 5.3 shows how Consul can be integrated into the existing infrastructure. By keeping one Consul server instance on each machine that hosts a service, all servers are aware of each other by utilizing the consensus protocol[5][12]. The consensus protocol takes advantage of the raft algorithm, which follows that the cluster required at least $(n/2) + 1$ members (a quorum). If one of the $(n/2) + 1$ members goes down, will make the other members be aware of it.

With the use of health checks or hearth beats Consul will also be aware of the service states hosted on these machines. Load-balancers can then dynamically configure it self dependent on the information given by the Consul cluster, a very important functionality used during rollouts, described later in section 5.4. Consul's services are defined by one unique name and which port the service is reachable through. This makes it optimal to use for containers, were the ports on the host are bound to certain ports within a container.

5.3.2 Standarized Configuration Management

Keeping the configuration management as standarized as possible can gain many benefits, as described in chapter . Following the best practices described in section 5.2.1, standardized playbooks can be constructed with the use of different set of variables associated with a specific set of hosts, as can be seen in figure 5.4.

[4]Consul Service Discovery Tool - https://www.consul.io/

[5]Consensus Protocol - https://www.consul.io/docs/internals/consensus.html

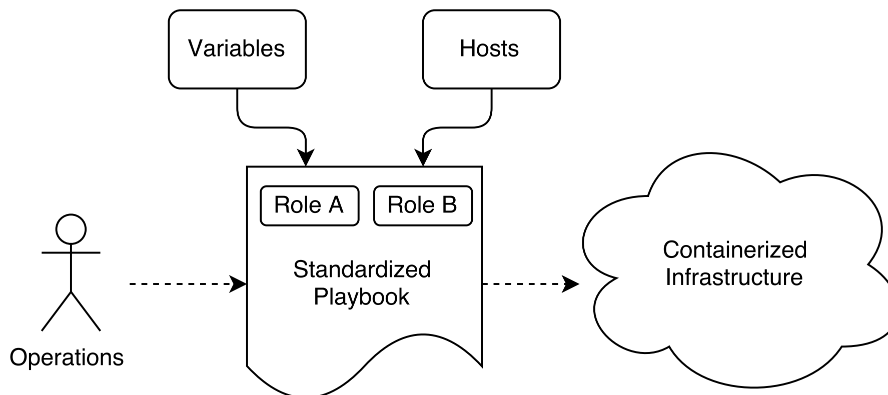


Figure 5.4: Standarized configuration management using Ansible.

These variables can even dynamically, with the use of logic operations, configure a standardized role. The list below will present the fundamental standardized roles that makes it possible to generalize the infrastructure and adapt it to serve different set of services. In this master's thesis those services are the QA, development and production environment hosting the "dummy-project".

Docker Node This role configures a Linux based operating system to serve Docker containers. The goal of this role is to make the machine a Docker host, independent on the Linux distribution, for instance CentOS or Ubuntu. All other roles are dependent on this role, due to the fact that the whole service infrastructure should be containerized.

Docker Image Another very fundamental part of the generalized infrastructure is the Docker Image role. This role makes it possible to dynamically build docker images based on a variety of variables and attributes. The images are built dynamically by templating configuration files, send them to the nodes and let them build the docker images. For instance; one docker image can be created specific for a host and its underlaying service which produce hearthbeats in order to check the state of that specific service. Docker containers can then be started based on these images, without the need to specify any configurations during staring phase, due to the fact that the configurations have already been settled within that image.

Service Node Makes it possible for a node to act as an Consul service host. This is achieved by constructing consul agent containers based on machine variables or facts. This Consul agent container can then be used to bootstrap a Cluster of service nodes. This is done automatically based on a group of hosts and the groups associated variables.

Load Balancer This role makes it possible to initiate a dynamic load-balancer (HAproxy) based on service attributes. With the use of Consul this container can then adapt the configuration and balance based on the service data and the key/value store.

Registrate Service This role registrates a service with the use of the consul agent container and set ups the healthchecks.

Deregistrate Service This role deregistrates a service with the use of the consul agent container.

Docker Deploy Deploys a container based on a docker-image associated service information.

Docker Rollback Makes it possible to rollback to a previous container state.

Docker Data Bind a data-volume to a specific container and service, which will be described on more detail in a section 5.3.4.

This is a just a few the roles that were implemented during this project. However these roles set the base of the generalized and containerized infrastructure.

5.3.3 Monitoring

Being able to monitor the infrastructure at all times is an important requirement. A sysadmin can be forever all-knowing regarding the state of the infrastructure, by combining health checks with the use of watches. In this project the monitoring was solved by containerizing the healthchecks and performing general scripts for testing a service and the machine health. If the containers incorporate scripts for testing services, they will be able to take advantage of the Docker framework in order to link itself together with the service-container. This eases the networking. A simple example of a HTTP-check can be seen in appendix F.

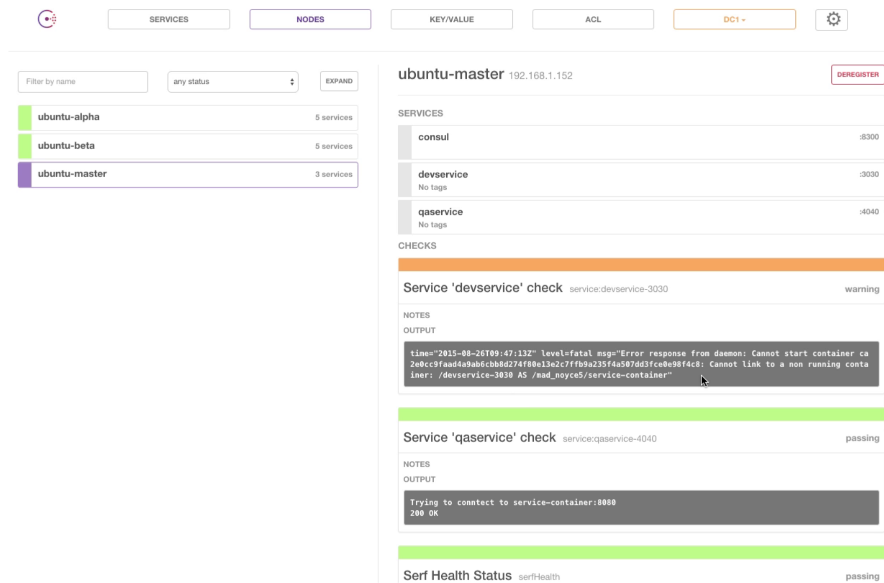
In order to notify a sysadmin if a script fails, watches can be used. A watch is, basically, a process that waits for state changes in order to pipe data into another process. At Data Ductus Slack is heavily used as the default communication tool, therefore an integration[6] using watches, Consul and python-scripting with Slack was a obvious solution. Figure 5.5 shows a failing Consul service notifying the user with the use of Slack.

5.3.4 Data Management

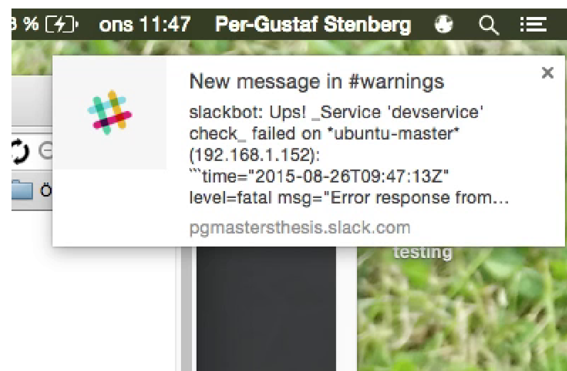
As described in section 3.3, Docker utilized the AUFS filesystem to get the layer functionality as needed. The AUFS filesystem is slow and should by avoided by mounting host volumes into the containers during heavy disk I/O operations. However the data mounting can be generalized due to the fact that Docker gives the possibility to inherit data used from a container from another container, as shown in figure 5.6. This technique can be used when containers needs to share data between them by creating a "data-container". The only purpose of this data-container is to host data. This data-container can then be used by "data-manager" containers, which makes different operations on the mounted data. This technique can for instance be utilized for data backups and data restoration, as shown in figure 5.6.

Data-management are outside the frames of this Master's thesis, hence the data-management implementation is not a priority, and should be seen as a prototype or a proof-of-concept.

[6]Consul Slack Integration - https://github.com/pgstenberg/docker-consul-slackbot



(a)



(b)

Figure 5.5: Screenshot of a failing Consul service, notifying the user with the use of Slack.

5.4 Step 3. Deployment Strategy

The next step in the implementation phase is to create deployment strategies. This section will describe two methods, one for general deployments, and another specialized for deployments into production. The goal of these methods is to achieve a controllable zero-downtime deployment, hence the customer will not notice any downtime during update. This is a crucial requirement for the 100koll project and is often crucial requirement for other projects as well.

5.4.1 Rolling out Releases

“Rolling” out releases will be the general method for deploying new artifacts to the cluster. This method will in this Master’s thesis be executed during deployments to development

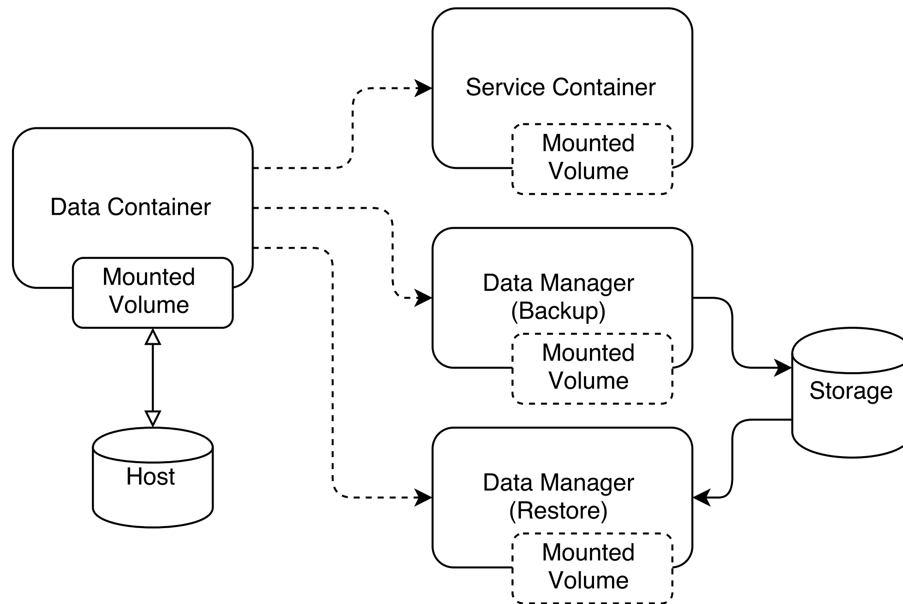


Figure 5.6: Data management with Docker containers.

and quality assurance environment. The deployments to these environments will usually not require the same degree of deployment control as for the production environment.

Rolling out deployment are built up with sequential, performed by Ansible. In the case of an failure, a rollback routine will be executed. The deployment steps involved in a rollout can be described as following:

1.) Prepare Rollout (all nodes) This first step, and fundamental step, is required to be able to rollback when so is desired. By committing the state of the service container and utilize the key/value store hosted by Consul to point to this committed image, a rollback will be possible in case of a occurring failure.

2.) Pull Docker Image (all nodes) To speed up the deployment time even further this step will be downloading and prepare the upgraded artifact which will be replacing the current service container in a later stage.

3.) Deregistrate Service (50% of the nodes) When the two preparation steps are done, the rollout process can be initiated. The first step is to deregistrate the service from the Consul cluster described in section 5.3.1. This will notify the load-balancer that the service will not be available. The load-balancer will then be reloading the traffic pool without the node with the deregistrated service.

4.) Deploy (50% of the nodes) This step will first stop and remove the current service container, which will then be replaced by the newly pulled image. It is basically the step were the current service container will be replaced by a upgraded one.

5.) Smoke Test (50% of the nodes) After a certain given delay a smoke test will be executed on the newly started container. In case this fails, the rollback procedure will be executed, and the “smoked” container will be committed for further investigation.

6.) Registrate Service (50% of the nodes) If the smoke-test passes, the service on the node will be registrated again. This will notify the load-balancer that the service is ready for traffic. As soon as this procedure is done, the rest of the nodes will execute the four last steps as well.

If this routine is executed as expected the deployment will be considered as a success. However in case something fails, for instance the smoke-test, a rollback plan need to be conducted. The rollback steps can be described as following:

1.) Pull rollback image Pull down the Docker image based on the previous committed container pointed by the key/value store hosted by the Consul cluster.

2.) Deploy rollback image Stop and remove the none-working container and replace it with the last functional one (the rollback image). The rollback image will always be a working candidate, hence it inherits from the last working state.

3.) Registrate Service Registrate the service again to let the load-balancer know that it is in a healthy state again.

These two plans will cover the zero-downtime deployment requirement, even if a failure occurs. The customer will never notice (in theory) any down-time during deployments.

5.4.2 Blue-Green Deployment

Deploying to production requires a higher degree of control, which can be achieved by following a strict deployment plan or routine. Farley and Humble presents a method in their book they call blue-green deployment[8]. The concept is about having two identical environments, one blue and one green, there will always be a possibility to switch back and forth between them without impacting the customers, with the use of load-balancing. One huge downside with this method is that there will always be one idle environment, this solution is there for not resource optimal. By introducing the use of service discovery, dynamic load-balancing and containers, it is possible to exploit both of these environment at the same time. In this master's thesis one of such implementation have been constructed, as can be seen in figure 5.7.

Consul makes it possible to tag certain services. Those tags can be direct taken advantages of by the load-balancer, by directing the traffic to a specific tag. In the deployment plan shown in figure 5.7, we can see that in figure (a), that both blue and green are tagged *PROD*. In the next step (b), one of these environments have switched tag to *STAGE*. In this way a new temporary load-balancer can be initialized which only directs traffic to a the newly updated and tagged environment. The auditor (usually a operations personell), can then approve or disapprove the new version. If the auditor disapproves, the rollback plan can be executed and the environment can be switched back to *PROD* again. If the auditor approves the change the updated environment will then be tagged *PROD*, and the other will be untagged to perform the same rollout as the environment tagged *STAGE* earlier. In this way the deployment procedure can be controlled, and in case of failures, switching between these environments will always be possible.

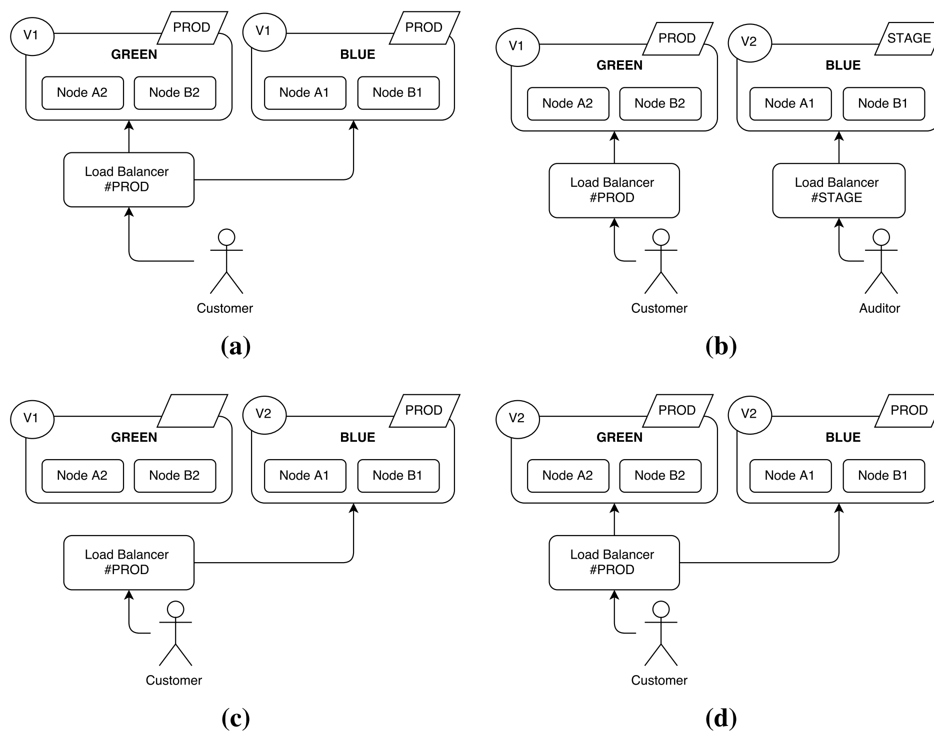


Figure 5.7: Blue-green deployment sequence with the use of service discovery and containers.

5.5 Step 4. Constructing a Pipeline

When the configuration management have been automated and good deployment strategies have been established, an deployment pipeline can start to take form. The deployment pipeline is essentially a number of sequential steps or tests which a newly commit code needs to pass in order to state that a change is releasable. If any of these steps fails, the whole pipeline will fail and the new changes till be considered unreleasable. However it is important to make these steps relevant and make sure the quality will be tested throughout the whole pipeline.

5.5.1 Integration

The first step in the pipeline is to integrate new code into the existing project. This practice are usually already implemented, due to the fact that it is part of the continuous integration concept. By utilizing unit testing, will make sure the new code will not negatively impact other part of the source-code. If the unit-testing passed, the code will be compiled and be considered as a new artifact, with a version associated along with it.

5.5.2 Quality Testing

As described in chapter 5.3.2, it is very important to setup quality testing to make sure no quality deficient artifacts will go out into production. In this Master's thesis the first

quality test is test-coverage. These tests executed even before the artifact are deployed into the development environment. When the deployment to the development environment are successful (hence the smoke-tests passed etc.), the deployment to QA will be initialized. The QA environment will be used to quality assure the newly created artifact, in this case by two types of tests. One functional-testing or blackbox-testing and one stress-testing. The stress-testing is important in this project due to the fact that the 100koll project is a hight currency system that relies on stability and speed.

When all quality-tests have passed the artifact can start rollout to production using the blue-green deployment plan described in the previous section 5.4.2.

5.5.3 Implementation

Deployment pipelines are usually implemented with the use of continuous integration tools, in this case Jenkins[7] were used[2]. By declaring different types of "jobs" in Jenkins and by linking them together, gives the possibility to construct chaining events, hence your pipeline. However chaining jobs together in Jenkins lacks maintainability. It is possible to version control your Jenkins configurations, but the there will always be a need for administrating the jobs individually inside the graphical interface.

CloudBees solved this problem with their workflow Jenkins plugin[8]. With the use of domain specific Groovy-scripting it is possible to construct flows with tasks tightly integrated with Jenkins. This have numerous of benifits; for instance, the pipeline can be version controlled and maintained as a script. Instead of creating multiple linked jobs, the pipeline will only require one, which triggers the groovy script that executes the workflow. Another great benefit by using scripted pipelines is that the script can make user interaction. In this project, under interaction were utilized during blue-green deployment auditing and during specifying the version of the artifact to deploy into production.

In this project two workflow were created, one for the pipeline and one for release into production, as can be seen in figure 5.8.

5.6 Step 5. Return Feedback

When the first four steps are established and a deployment pipeline have been taking shape, some kind of feedback system can be implemented to the project members. The feedback is essential in order to be able to notify failures during changes, or successfully code integrations. As mentioned in the book about continuous delivery, it is not uncommon to have some sort of information radiator[8]. As mentioned before, Jenkins have been used as the default continuous integration and delivery tool, with the use of groovy scripting based workflows. There exists Jenkins plugins to be able to visualize chained jobs as a pipeline, however they are very inflexible and lacks functionality, therefore an stand alone alternative was implemented during this project.

[7] Jenkins Continuous Integration Server - https://jenkins-ci.org/

[8] Jenkins Workflow Plugin by CloudBees - https://github.com/jenkinsci/workflow-plugin

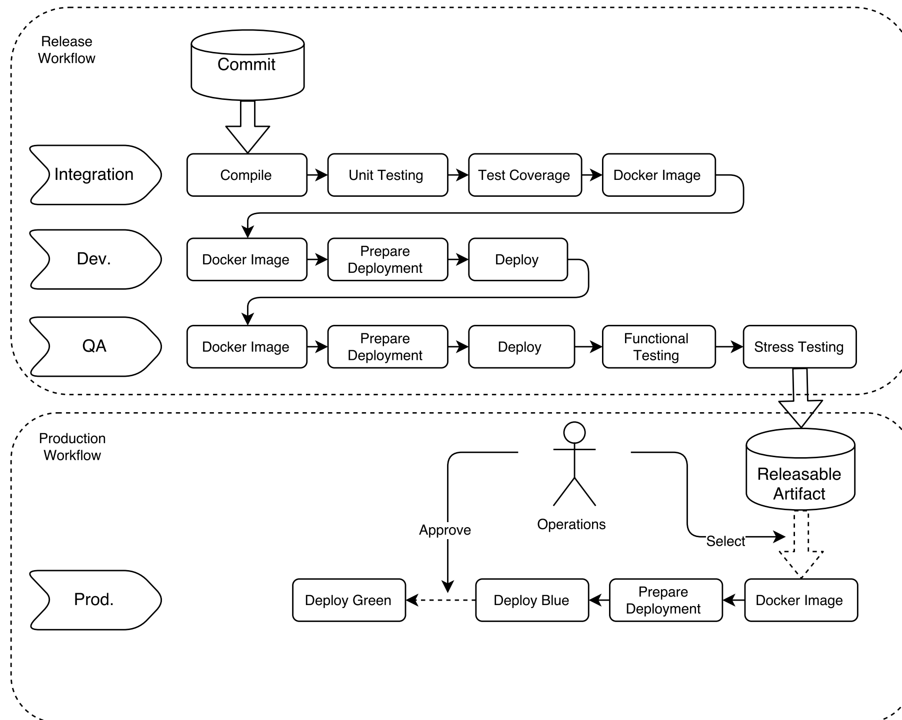
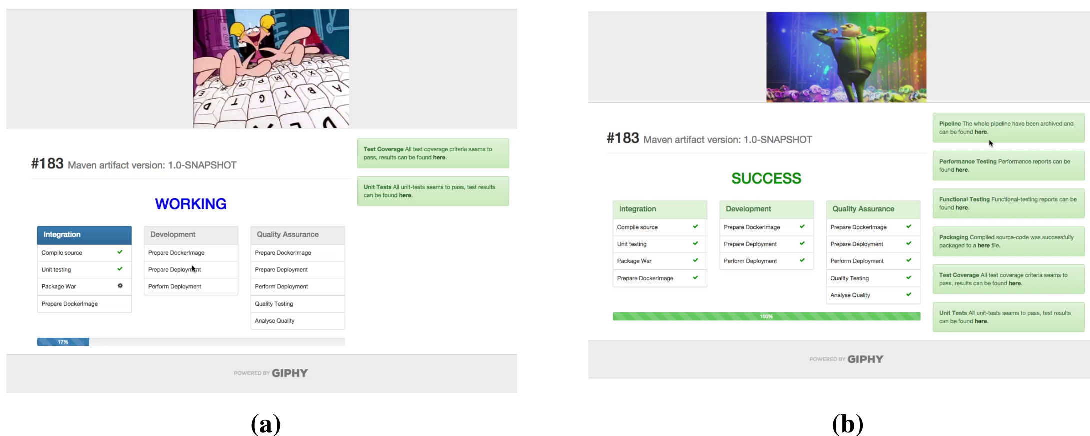


Figure 5.8: Flowchart over the deployment pipeline.

5.6.1 Pipevis

Pipevis[9] is a single page application that can act as an information radiator; with artifact links, reports and pipeline-progress. Pipevis have been developed with the use of numerous open-source frameworks. Pipevis main goal is to maintain the importance of making information radiator pleasing for the project members to be not considered as just another mandatory thing to keep track of. It will help the developers keep the importance of checking reports through simple links, and by always informing which state the project is in.



(a) (b)

Figure 5.9: Pipevis screenshots

[9]Pipevis Information Radiator - https://github.com/pgstenberg/pipevis

Chapter 6
Evaluation & Results

This section presents the results from the evaluation based on the metrics presented in section 2.2.3. These results will support the discussion in the later chapter and answer the questions raised in section 1.2. The solution which was presented earlier in the chapter 5 is purely based on containers. This will provide us with the most precise evaluation possible and, in turn, allow for a qualitative research with stellar properties. During the point in time in which this report was originally written, the solution had not yet been used in actual live production, therefore these results are based on the first impressions of the project members as well as a deeper analysis of the impact of the solution on the workflow. Close collaboration with the operations personnel have also been a huge factor during the qualitative research.

6.1 Cycle Time

As mentioned in chapter 5.3.2, manual deployment and operations are today considered a huge anti-pattern. This anti-pattern makes it hard to measure an accurate cycle time, due to the fact that most operations are done manually. However, when analysing the current workflow further, a rough estimation have been done; request for a bug fix from a customer may take several days to be put into production. By introducing automated processes for quality assurance, testing and deployments, the time to get a new release into production can be reduced to under an hour. This is based on how long the automation will executed, and not based on the manual labour behind a new deployment. However as described in previous section, some manual auditing will be required during operations towards production environments.

With continuous delivery, the cycle time can be seen as the span from a customer request to the state were a releasable artifact produced through the pipeline. For a single line code-fix this can be a huge benefit; the operations and deployment can, in these cases, be considered a bottleneck. The deployments will not be dependent on a deployment

expert either; they can be executed by any of the project members whom have got the approval-privileges for artifacts headed for production.

The evaluation shows that introducing continuous delivery into the project workflow will facilitate small iterations. This is due to the fact that deployments will not be regarded as time consuming factors. During larger iterations the deployment time will have a relatively lesser impact on the overall cycle-time.

6.2 Feedback

During the analysis of the continuous integration workflow for projects at Data Ductus, the need for a standardized feedback system was discovered. The most common continuous integration server used is Jenkins, where reports from different projects are presented. However, the usage of these reports were not consistent throughout the whole development team. With the use of an information radiator, developers can be more aware of the state of the project. With direct access to the reports, the developers can receive feedback without even knowing the underlaying technology for generating them. By keeping the information esthetically pleasing, developers and the stakeholders will be more motivated to always keep an eye on the information radiator, which in the end will hopefully motivate project members to keep the project at a healthy state at all time. Pipevis, the pipeline visualization tool showed promising result among the developers.

6.3 Quality

As described in the previous section there is no standardized system for feedback. Maven is mostly used as the default building script for Java based projects, where the scripts may vary from project to project. By introducing thresholds during tests executed by Maven, it is possible to reflect on these tests, wether they pass or not directly into the feedback cycle. Quality testing have been done before using jMeter, and the concept is, therefore, not unfamiliar. However, by combining thresholds with the deployment pipeline, the quality can be approved even further, by failing artifacts that do not fulfill the quality requirements.

In this Master's thesis two different implementations were constructed in order to solve a fibonacci sequence, as described in section 5.1. These implementations were easily switched with the use of different configuration files for the QA, development and production environments. Switching to the less efficient solution did not show any differences in the development environment; although, during the stress-testing against the QA environment the thresholds set in the Maven script were not fulfilled. Thus, the pipeline failed and a notification was shown. With that, the developers can come to the immediate realization that the last change in the source-code was not release-friendly, due to the lack of quality.

6.4 Implementation Effort

Containers had a huge impact on the implementation aspects of this project. Containers made it possible to reduce almost all configuration dependencies. The five implementation

steps presented in the chapter 5, was conducted within 2 months, with minor knowledge of the technologies on beforehand.

The solution could, within three hours, be maintained and used by the operations personnel in a test environment, based on the demonstrations that were conducted. The service discovery technique when introduced seemed trivial to most of the developers and operations personnel.

Chapter 7
Discussion

This chapter will hopefully give the reader solid answers to the questions raised throughout this report. The approach, method and evaluation will be discussed along with the solution. This section also features a more thorough discussion on continuous delivery and the impact on the 100koll project, as well as the role of containers in the implementation process. It is worth to mention again that this system has not been introduced into production, therefore a lot of these conclusions are assumptions based on facts rather than finalized results.

7.1 Method & Evaluation

A qualitative research was chosen for this study. It took some consideration, but since the better perspective would come from a deeper analysis of the current workflow as well as an understanding on the work-methods of the small 100koll-team at Data Ductus, quality trumped quantity. A quantitative research was considered but later dropped, as this would not contribute to sufficient answers. Larger amounts of data, from example questionnaires, would not provide enough relevant data on how containers facilitated the process towards continuous delivery. By close collaboration with the operations personnel, and the team involved in the project, understanding which anti-patterns that could be solved at Data Ductus gave the research solid conclusions. Without the possibility to be as tightly integrated with the development team at Data Ductus, a quantitative would probably be an alternative.

The metrics chosen for the evaluation could have been established and considered earlier on in the process, as this would have made the path to the conclusion somewhat less crooked.

The collaboration with Data Ductus was very close, and most of the work was done in their offices. This tight knit setting provided the required insight that was needed to, in combination with the metrics, reach the conclusions.

7.2 The Solution

7.2.1 Stability

The system was developed using the well defined five steps of implementation described in chapter 5. The system seemed solid and stable, due to the fact that the pipeline was successfully triggered about 200 times, even though the infrastructure behind the test and development environment was very poor. The system was also stressed constantly during three days without any noticeable hiccups. The system did preform as expected when it was migrated into a another, better suited, infrastructure set-up by the operations personnel. This shows that the system is very adaptable, general and standardized, which was a goal during the implementation process. This makes it very easy to setup a new infrastructures if so is desired, or even scale the infrastructure and resources at will. These properties were made possible by making the infrastructure programmable, with minimal manual processes.

7.2.2 Alternatives

The orchestration of containers in the infrastructure was implemented via the use of service discovery, Docker and a configuration management tool. Finished solutions already exist for these functionalities, and a lot of them were analyzed and tested before the first prototype was constructed. However, it was soon discovered that a lot of these tools seemed very immature, and most of them were still in the early alpha stage. The fact that this system was aimed at helping Data Ductus in production, the use of alpha staged tools was not an option. Therefore it seemed to be a better choice to use already tested and verified techniques and tools to implement the solution. As these finished solutions mature they might be better candidates in the future, but for now the combination of service discovery and configuration management tools is the best option.

7.2.3 Hiccups

When the system was introduced into Data Ductus own servers by the operations personnel, it was soon discovered that one important dependency was not covered. The fact that Ansible is heavily based on Python, gave a version conflict with one of the Python modules. This was however easily fixed with the use of virtual environments, by initializing one isolated Python environment to be used by Ansible. As this solution will go into production more hiccups might occur, but will be solved along the way.

7.2.4 Containerized Configuration Management

The Ansible runtime and its configurations are containerized as well. The containerization of Ansible and with the use of virtual environments makes it possible to avoid all configuration and runtime dependencies. The containerization of Ansible, however, expects that the Docker framework is already installed on the machines executing the Ansible container. This container, or configuration management container, is usually executed by

the continuous integration server. This is a great practice, since the execution will always be logged and archived, in turn gives better change control.

The fact that the configuration management is container-based gives the possibility, with no substantial extra efforts, to execute it on any machine with an OS based on Linux, for instance on any of the project members laptops. If the user does not have a machine running Linux, the Docker community has constructed great tools for managing a virtual machine running Docker. The containerized configuration management concept can even make it possible to deliver entire ready-configured infrastructure setups directly to the customer, with the intention of hosting the customer's requested service. Data Ductus works as consultants and this serves their line of work very well.

7.2.5 Change Control

Another huge benefit optained from introducing infrastructure as code, is the possibility to use ordinary source control tools. The state of the infrastructure can be somewhat more controllable which will facilitate the whole configuration management aspects of a project. A lot of focus has been placed on this during this project which resulted in version controllable pipelines, feedback systems and infrastructure, in addition to the traditional configuration items like source-code, tests and documents. These additional configuration items can then be included into the controlled change process and baselines which can facilitate the configuration management.

7.2.6 DevOps

The containerization of the infrastructure and the introduction of infrastructure as code gave circumstance to a better collaboration between the operations and the developers, which in turn will give the team a more DevOps approach during development. By introducing the infrastructure as a configuration item along with the source-code, provides the possibility for developers to better grasp the underlaying infrastructure and operations to better understand how the infrastructure collerate with the source-code. Using the programmable infrastructure, developers even have the option to initialize the same infrastructure setup as the production environment with the use of virtual machines on their local development machine.

7.3 Continuous Delivery

The goal of this Master's thesis was to introduce continuous delivery to a cluster with the use of containers and to see if containers impacted the road towards continuous delivery. During the project progression, it was soon realized that this was not a simple task. Constructing the deployment pipeline to achieve the finishing results of continuous delivery was a small and trivial part of the whole project. The main part of the focus was faced towards developing a programmable infrastructure using containers. However, the work served its purpose, by introducing infrastructure as code, made it possible to easily achieve continuous delivery with a lot of added bonuses.

7.3.1 Project Management

This Master's thesis provided the possibility to see the true value of continuous delivery in the context of project management. The result shows that; small iterations will benefit the most from introducing continuous delivery. Small iterations correlate with improved product quality; smaller changes will go trough the pipeline, in order to verify its quality, which can in turn be improved even further with the closed feedback loop. The deployment pipeline provides the possibility to establish the definition of "done". By defining that a story is not done until it passes through the whole pipeline, instead of having a vaguer definition of when the story is "done", gives for instance the SCRUM master during sprints better backlog control.

A happy project manager

As the result also shows, by introducing continuous delivery, will make the release of new versions not a [as] time consuming factor. This makes it possible for instance during a sprint, implement a bunch of stories to the state of done (with the new definition), and by the end of the sprint always have a releasable artifact that can be automatically sent out into production (if so desired). This will decrease the cycle time dramatically and increase the time to market. This is especially shown during "single-line of code fixes" or "quick fixes", hence these type of customer requests will consider operations and deployments as a bigger bottleneck. The consequence of better time to market is happier and more satisfied customers, which in turn gives better revenue.

7.3.2 Continuous Everything

In order to gain access to the full benefits from continuous delivery, a cultural change needs to be in place. Continuous delivery extends the use of continuous integration, which is a combination of two words *continuous* and *integration*. Therefore the integrations need to be conducted continuously, or the whole idea of continuous integration will fail, and in turn the delivery process as well. Most developers fear the integration process and are usually more comfortable working on their own branch in silence. By introducing tools and features which will make developers more aware of their surroundings are therefore very important. These tools should not feel like another burden and should be seen as a type of motivation and utility. By introducing esthetically pleasing information radiators of feedback systems like pipevis, developers might be inspired by the urge to achieve something good, or even grow an addiction to seeing how their new code integrates with the rest. These types of information radiators or feedback systems can even be taken a step further by integrating real physical things. For instance by integrating a microprocessor into the feedback system which flashes lights depending on the project status. To get that extraordinary and pleasing feeling during this project an integration with Giphy[1] was implemented. Giphy was used by randomizing different gif:s depending on the outcome of the new code integration and project status. These types of continuous feedback systems

[1] Giphy Animated Gifs - http://giphy.com/

can hopefully improve the communication between the projects members and force the developers out of the "branching bubble" and force them to start integrating their code more continuously. If this mentality is set in place, it is possible to receive even more feedback from the customer. The feedback loop will be more consistent due to the increasingly continuous releases. In turn,the product quality will be increased continuously, thanks to the continuous delivery system.

7.4 Docker

7.4.1 Impact on the Process

Containers are not a new concept and have been around for a while. However, when Docker hit the market they increased the availability of containers. By constructing easy to use tools around the container runtimes, made it possible to, without any considerable time investments, utilize the container functionality. This research shows that it is possible to construct a solid continuous delivery workflow with the use of Docker along with infrastructure as code and basic monitoring within a two-month time frame. Without the use of containers, these result would have been a lot harder to achieve.

7.4.2 Possible Problems

During the progression of this research, it was soon discovered that Docker is still an immature technology. The framework has not been around for very long and during the implementation a couple of new releases were introduced, along with new alpha staged tools. With this, the conclusion that Docker works better alongside virtual machines, instead of using Docker as a substitute, can be drawn. Virtual machines provides the infrastructure with the mandatory trusted and secure environments. In this way Docker can be used as another layer of abstraction, which makes the infrastructure a lot more flexible. In Data Ductus' case, the immaturity was directly reflected on their default Linux distribution for the 100koll project, CentOS 6.6 from RedHat. CentOS 6.6 ships with an older Linux kernel that was too unstable with the Docker framework. When switching to a newer version CentOS 7, which is based on a newer Linux kernel, Docker worked as expected, even though the Docker community told the users that CentOS 6.6 was supported. They later dropped the support for CentOS 6.6 when a multitude of issues arose.

Another issue to consider is to view Docker as a problem solving tool and not a way of disguising the issues. For instance the comfort of the ability to just restart a container if something fails, might translate into bad qualitative code running in production. Instead of eliminating the issues in the source-code from start, it might seem more comfortable to just restart the container during crashes. The problem might not even be detected if the container restarts itself during crashes.

7.4.3 Resource Usage

As mentioned these kinds of end results would not be possible without the use of Docker. With the right trusted and tested tools, the framework can be very powerful and gain huge

benefits, and the extra level of isolation is perfect for today's "cloud" computing. The fact that containers are so lightweight made it possible for containers to start and stop within seconds. This in turn gave the infrastructure fast scaling and adapting possibilities, and finally provided better resource usage. Introducing containers to the infrastructure provides the possibility to squeeze more isolated functionality into one single machine. For instance one machine can be a cluster node for both the QA and development environment, with 5 different versions of the same service. Optimized resource usage can be directly reflected in tighter budgets, which can be a deal breaker for most businesses.

7.4.4 What Docker really brings to the table

It might sound like Docker is the "magic" solution to everything, but in fact Docker will only facilitate minor processes which can be seen as bottlenecks. As have been described and discussed before the functionality of containers can be achieved with the use of virtual machines. However, the big difference is the flexibility that comes with the lightweight framework. The fact that containers can start, stop and transfer isolated instances within seconds is not possible with the ordinary use of virtual machines. It can be seen as a unified solution for transferring and executing isolated artifacts from point A to point B. This makes Docker perfect for fast-adapting infrastructures and with the possibility of quick deployments. Docker should be therefore seen as an additional infrastructure feature in combination with virtual machines for increasing scaling, deployment and resource usage. Virtual machines gives the security and trustability that cannot be provided with Docker. Docker should not be a dependent technique but should be seen as a possibility to get the desired extra layer of abstraction. This attribute makes it, in turn, a lot easier to construct a infrastructure system that supports continuous delivery.

Chapter 8
Conclusions

This section presents conclusions and answers to the questions raised in section 2.1. The first section presents the conclusions involving continuous delivery. The second section presents the anti-patterns that preventing Data Ductus from achieving continuous delivery today. The conclusions on how containers impacted the process towards continuous delivery will be prestented in the last section.

8.1 Continuous Delivery

It is important to note that continuous delivery is not just about implementing a new workflow and system for delivery. It has more to do with a cultural change. The introduction of continuous delivery will not magically make a development team more agile. The agile mindset needs to be in place beforehand in order to get the full potential of continuous delivery. However, continuous delivery and especially the feedback potential gained from the deployment pipeline can be seen as a motivation to keep the thinking more lean and agile. This can, in turn, reflect directly on the product quality, time to market and in the end happier customers and better revenue.

8.2 Obstacles preventing CD

The most obvious anti-pattern at Data Ductus is manual deployments. The conservative mindset around the manual deployment process is usually protected by the feeling of having control of which artifacts are sent out to production. By introducing a good automated quality assurance testing option, the operations and developers are provided with the confidence to rely on the artifacts to fulfill the quality requirements and to then be sent out, automatically, to the different environments. Another big anti-pattern is the use of manual configuration of the infrastructure. Manually configurating the infrastructures can lead to

failures which are hard to track down and locate. It also leads to node inconsistency. With the introduction of programmable infrastructure or infrastructure as code, it is possible to utilize version control, which leads to better version control. This will in turn simplify the orchestration processes during deployments, better failure handling as well as the possibility to rollback to previous configurations of the infrastructure. During this project it was discovered that the biggest amount of time was focused on the process of constructing infrastructure as code. The construction of the concept of infrastructure as code was not in vain, as mentioned, it did create a lot of benefits.

8.3 Containers impact on CD

Achieving continuous delivery is not an easy task. A lot of techniques, patterns and the basic setting needs to be in place on beforehand. The use of containers made this struggle less painful. However, Docker should not be seen as the magic tool which saves people from the outdated workflows, but as more of a tool which can facilitate the processes towards new agile thinking. By identifying the anti-patterns which prevents the team from achieving continuous delivery, the bottle-necks that emerges can be dissolved more easily with the use of containers. This, in turn, can break these anti-patterns. Even though containers helped the process of converting the infrastructure to code, it was not a necessary utility. The containers worked best in combination with deployment and infrastructure configuration, due to the fact the containerization of essential artifacts associated configurations without unnecessary resource usage, provided the possibility of moving artifacts from point A to B in a less straining way. These container attributes in turn facilitated the possibility to move from continuous integration to delivery.

Bibliography

[1] Charles Anderson. Docker. *IEEE Software*, 32(3):102 – 105, 2015.

[2] V. Armenise. Continuous delivery with jenkins: Jenkins solutions to implement continuous delivery. *2015 IEEE ACM 3rd International Workshop on Release Engineering (RELENG). Proceedings*, pages 24 – 7, 2015.

[3] Jie Chen, Jun Wang, Zhihu Tan, and Changsheng Xie. Effects of recursive update in copy-on-write file systems: A btrfs case study. *Canadian Journal of Electrical and Computer Engineering*, 37(2):113 – 22, Spring 2014.

[4] Lianping Chen. Continuous delivery: Huge benefits, but challenges too. *IEEE Software*, 32(2):50 – 54, 2015.

[5] Martin Fowler and Matthew Foemmel. Continuous integration. *Thought-Works http://www.thoughtworks.com/ContinuousIntegration.pdf*, 2006.

[6] Sébastien Goasguen. *Docker Cookbook*. O'Reilly Media, 2015.

[7] J. Humble, C. Read, and D. North. The deployment production line. *AGILE 2006*, 2006.

[8] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley, 2010.

[9] Solomon Hykes. Introduction to docker engine. `https://www.youtube.com/watch?v=jB-Ddfph7EI`, 2015.

[10] Michael Kerrisk. Namespaces in operation. `http://lwn.net/Articles/531114/`, 2013.

[11] Paul Menage. Control groups (cgroups). `https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt`, 2004.

[12] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, Philadelphia, PA, June 2014. USENIX Association.

[13] Oracle. Adapting to peoplesoft continuous delivery. *Oracle White Paper*, sep 2014.

[14] J.E. Smith and Ravi Nair. The architecture of virtual machines. *Computer*, 38(5):32 – 8, 2005.

[15] Chris Swan. Docker drops lxc as default execution environment. `http://www.infoq.com/news/2014/03/docker_0_9`.

[16] Ram Rajamony Wes Felter, Alexandre Ferreira and Juan Rubio. An updated performance comparison of virtual machines and linux containers. *IBM Research Report*, jul 2014.

Appendices

Appendix A
Fibonacci Sequence

```java
public interface FibonacciSequence {
  public abstract BigInteger getFibonacciNumber(int n);
}

public class FibonacciSequenceRecursive implements FibonacciSequence{
  public BigInteger getFibonacciNumber(int n) {
    if (n <= 1) return BigInteger.valueOf(n);
    else return getFibonacciNumber(n -1).add(getFibonacciNumber(n-2));
  }
}

public class FibonacciSequenceMemoryCached implements FibonacciSequence{
  private static ArrayList<BigInteger> cache = new ArrayList<BigInteger>(){
  {
    add(BigInteger.ZERO);
    add(BigInteger.ONE);
  }};
  public BigInteger getFibonacciNumber(int n) {
    if (n >= cache.size()) cache.add(n,
               getFibonacciNumber(n-1).add(getFibonacciNumber(n-2)));
      return cache.get(n);
  }
}
```

Appendix B
Golden Ratio

```
public class GoldenRatioCalculator {
  private FibonacciSequence fibonacciSequence;
  private int goldenRatioDecimals;
  public GoldenRatioCalculator(FibonacciSequence fibonacciSequence,
                               int goldenRatioDecimals){
    this.fibonacciSequence = fibonacciSequence;
    this.goldenRatioDecimals = goldenRatioDecimals;
  }
  public BigDecimal calculate(int n){
    BigDecimal fibX = new BigDecimal(fibonacciSequence.getFibonacciNumber(n));
    BigDecimal fibY = new BigDecimal(fibonacciSequence.getFibonacciNumber(n+1));
    return fibX.divide(fibY,goldenRatioDecimals, BigDecimal.ROUND_HALF_UP);
  }
}
```

Appendix C
Service Register Role

C.1 tasks/main.yml

```
---

#Construct hearthbeat_script based on service
- name: Construct hearthbeat script based on service
  set_fact:
    hearthbeat_script: "docker run --rm -t --link {{ service_name }}-{{
        service_port }}:service-container {{
        docker_healthcheck_base_image }}:{{ service_name }}-{{
        service_port }}"
  when: service_name is defined and service_port is defined

- name: Registrate consul service with healthcheck
  consul:
    state: present
    service_name: "{{ service_name }}"
    service_id: "{{ service_name }}-{{ service_port }}"
    service_port: "{{ service_port }}"
    tags: "{{ service_tags }}"
    script: "{{ hearthbeat_script }}"
    interval: "{{ hearthbeat_interval }}"
```

C.2 defaults/main.yml

```
  hearthbeat_interval: 30s
  service_tags: []
  docker_healthcheck_base_image: "ductus/healthcheck"
```

Appendix D
Service Deregister Role

D.1 tasks/main.yml

```
---

- name: Deregistrate consul service
  consul:
    state: absent
    service_id: "{{ service_name }}-{{ service_port }}"
```

Appendix E
Deploy Playbook

E.1 vars/devservice.json

```
{
  "service_name": "devservice",
  "service_port": 3030,
  "container_port": 8080,
  "insecure_registry": "True",
  "rollbackable": "True",
  "docker_push_registry": "False",
  "docker_state": "reloaded"
}
```

E.2 deploy.playbook.yml

```
---

- hosts: service-nodes
  sudo: yes
  gather_facts: yes
  roles:
    - { role: ductus/docker/rollback/prepare }
    - { role: ductus/docker/pull }
  tags:
    - prepare

- hosts: service-nodes
  sudo: yes
  serial: "49%"
```

```
  roles:
    - { role: ductus/service/deregister }
    - { role: ductus/docker/rollout/perform }
    - { role: ductus/service/register }
  tags:
    - deploy
```

E.3 rollback.playbook.yml

```
---

- hosts: service-nodes
  sudo: yes
  roles:
    - { role: ductus/docker/rollback/perform }
    - { role: ductus/service/register }
```

Appendix F
Healthcheck Container - HTTP

F.1 check.j2

```
#!/usr/bin/env python

import httplib
conn = httplib.HTTPConnection("service-container:{{ container_port }}")
conn.request("HEAD", "/")
print ("Trying to conntect to service-container:{{ container_port }}")
r1 = conn.getresponse()

if r1.status != 200:
    print "Ups! service returned code %i" % r1.status
    exit(1)

print r1.status, r1.reason
```

F.2 Dockerfile

```
FROM python:2-wheezy

COPY check /bin/check
RUN chmod +x /bin/check

ENTRYPOINT ["/bin/check"]
```

EXAMENSARBETE Container-based continuous delivery for clusters
STUDENT Per-Gustaf Stenberg
HANDLEDARE Ulf Asklund (LTH), Mario Toffia (Data Ductus)
EXAMINATOR Martin Höst

Containers - en snabbare väg ut i drift?

POPULÄRVETENSKAPLIG SAMMANFATTNING **Per-Gustaf Stenberg**

Dagens krav på ständig uppkoppling ställer allt högre förväntningar på våra IT-system. Nya lösningar och funktioner måste leveras i en högre takt och kvalitet. Är containerramverket Docker nyckeln till snabbare leverans?

Dagens samhälle är ett uppkopplat samhälle. Den ökade internetanvändningen ställer större krav på IT-system och tiden det tar att utveckla, underhålla och uppdatera dessa. Målet för de flesta IT-företag är att kunna uppdatera sina system utan att man påverkar kundens tillgång till tjänsten, detta genom så kallad "zero-downtime deployment".

Dessa krav bidrar gemensamt till att infrastrukturen bakom IT-systemen blir allt större och mer komplicerade. Ny programkod måste levereras kontinuerligt. Detta bidrar till att problem upptäcks och kan åtgärdas snabbare. För att uppnå den kvalitetshöjande så kallade "continuous delivery"-principen, är det möjligt att använda sig av Docker-containers som kapslar in konfigurationer för att underlätta leverans och körbarhet till IT-systemet.

E-ons hundrakollprojekt är ett bra exempel på en tjänst som ställer höga krav på att konfigurationer samt mjukvara levereras felfritt och kontinuerligt. Hundrakoll hjälper E-ons kunder att hålla koll på sin elförbrukning; i sitt nuvarande tillstånd tillhandahåller tjänsten ett perfekt tillfälle att undersöka processen mot continuous delivery. I samarbete med IT-konsult-bolaget Data Ductus skapades, med hjälp utav Docker, en lösning för att kunna upprätthålla en kontinuerlig leverans av programvaruändringar.

Under arbetets gång skulle det snart visa sig att målet, continuous delivery, inte uppnås genom att bara ha tillgång till rätt verktyg (d.v.s. Docker). Docker kan dock underlätta hur konfigurationer och tillhörande körbar kod kan tas från en miljö till en annan. Dock kvarstår frågan om hur miljöerna i sig ska kunna konfigureras per automatik, samt att leda in utvecklarna i rätt tankesätt. Det krävs att ny kod testas och integreras med den befintliga koden kontinuerligt.

Projektet inleddes med att de arbetsprocesser som förhindrade Data Ductus från att uppnå continuous delivery identifierades. Docker, i kombination med andra konfigurationshanteringsverktyg, användes sedan för att bryta dessa så kallade "anti-patterns". Tillvägagångssättet för att kunna göra detta bestod av en implementeringsmetod i fem steg, i vilka containers stod i fokus.

Det första problemet att lösa var att beskriva infrastrukturen med hjälp av programmeringskod. Genom att sedan generalisera och standardisera denna programmeringskod kunde nya uppdateringsstrategier skapas för att uppnå en zero-downtime deployment. Detta gav tillräckliga förutsättningar för att gå vidare mot målet att uppnå continuous delivery.

Nästa steg var att skapa sekventiella steg eller trösklar som den nyintegrerade koden måste genomgå. Dessa steg bildar en så kallad "deployment-pipeline". Denna deployment-pipeline ger sedan möjligheten att återkoppla information till utvecklarna rörande hur deras kod klarade av integrationsprocessen. Detta ger i sin tur goda förutsättningar att uppnå ett kontinuerligt flöde av nya leveranser av kod med hög kvalitet.