UiO : Department of Informatics
University of Oslo

Simplifying Release Engineering for Multi-Stacked Container-Based Services

A constraint-based approach

Lars Haugan

Master's Thesis Spring 2015

Simplifying Release Engineering for Multi-Stacked Container-Based Services

Lars Haugan

18th May 2015

Abstract

Today, large and complex solutions are needed to provide the services that users require, where cloud-based solutions have been the panacea to help solve this problem. However, these solutions, are complex in the nature of their implementation, and the need for a standardized way of handling the services are in order.

This thesis aims to explore the possibilities of simplifying release engineering processes, with the usage of multi-stacked container-based services.

A model is designed with the goal of reducing the complexity in release engineering processes. It enables restriction of possible outcomes by enabling constraints to specify the wanted state of an environment, and enforces a single method approach towards achieving a more uniform environment.

The model has been implemented in a prototype, which enables the documentation, configuration and orchestration of the deployed services, that are deployed with the usage of Docker containers.

Through the implementation, the validity of the designed model is verified, and the complexity of the release engineering processes are reduced.

Contents

List of Figures

List of Tables

Acknowledgements

First and foremost I would like express my sincere gratitude towards my supervisor Kyrre Begnum, for his expertise and inspiration, that has helped me during this project. His encouragement, enthusiasm, creativity and constructive suggestions, is greatly appreciated.

Next I would like to thank Morten Iversen, Fredrik Ung, and Ratish Mohan for sticking around with me for the duration of this thesis, and their endless inspiration and input.

I would also like to acknowledge my coworkers, for their input and inspiration throughout the masters, for which I am far better off than without.

In addition I would like to extend my gratitude to my family, for the support and understanding during this thesis.

Last but not least, to the person who has helped me endlessly through the entire course of this masters program, Katrine Dunseth, thank you for the patience and love you have shown.

Sincerely,
Lars Haugan

Chapter 1

Introduction

Our society is today being rapidly digitized. With a rapid growth of Internet users over the last decade, the number of users of the Internet, has increased by almost 60% [1]. Today, there are almost 3 billion people that have access to the Internet [1, 2]. But the users that are already connected, have increasing requirements to the services already available, and the once that are not yet digitalized.

Large services are today used and are depended upon by millions. Online banking services, newspapers, and social media sites are just a few. As worlds larges social media site, Facebook has on average 890 million daily active users each day [3], which is an astonishing 30% of the active Internet users. With this volume, and huge expectancy from the users, new challenges appear in the process of providing architecture and services.

One of the trends over the last year has been the digitalization of currency and mobile payment solutions. This show that there is an expectancy of uptime and reliability of the services provided to the users. Furthermore, new features are expected at a more rapid rate than what were previously expected.

Large and complex solutions are needed to provide the services the users require. Cloud-based solutions have over the last years appeared as a *panacea* to this problem. These solutions are only part of the solution, as they are complex in the nature of their implementation. Today's applications are more complex with loose coupling and complex relations, and yet they must be developed more quickly than ever. This is a challenge for the IT teams, where the processes that has been in use, no longer is suitable.

As the number of components grow and the need for a larger underlying architecture changes, so does the complexity and a need for more rapid change. The more rapid change rate, is a result of the requirements customers have to new features and resolution of errors. Strong business incentive also apply, where return of investments of time to market are

important key factors of todays IT services. The rate of change we see today, is more common than what were common just a few years ago. As a direct result of this the System Administrators, with the task of the management of these huge and complex architectures, are depended upon to manage a much larger and rapidly changing service portfolio than what were previously needed. With this change, the element of human error is ever present.

To be able to handle this change in pace, a new paradigm is needed to handle the process of service and architecture deployment. This is a change from a waterfall methodology, to a more lean process. To accomplish this, automation is needed. While automation processes are deployed, repetition and local customization with tailoring in each business is implemented. This is a result of a lacking standardized way of handling automated deployments, due to the huge amount of different technologies.

As the technology has been made available, the methodology and use of the technology has been implemented at a case-by-case basis. New technology is now available that may help in the way release engineering processes are being handled, making it more safe and easier to handle continuous delivery of applications. One of the promising ways of working with software releases are with the usage of containers. Like a small version of a Virtual Machine, a container separates an application into a separate domain, ensuring that the processes are not interfering. A software called Docker uses the new features available in the Linux kernel that enables the separation of processes into containers. Based on a Docker container, it is possible to easy install new applications, that always stay the same.

We need a model, and methodology that can verify the correctness of a state beyond best practice, as best practice focuses on the technical aspect. For complexity truly to be reduced, we need to support automated verification in addition to simple reduction of manual tasks.

Problem statements:

1. Design and develop a model in order to reduce complexity in release engineering processes, using multi-stacked container environment.
2. Develop a prototype that implements the model designed based on the first problem statement.

A *model* is needed to create a clear way of standardizing the way of handling software releases with containers. The design of the model is important to ensure that it scales to large systems, but can also be used in smaller environments.

The *model* is needed to combat the *complexity* of the process of handling release engineering. It should be easy to create a new version of a software, and motivate towards faster releases.

Release engineering is the process of handling software when it is delivered

from the developers, and is ready to be released to production. In release engineering, there are multiple different stages, which are meant to ensure the stability and reliability of a software release. This can often be the stages of development, test, QA and production. These are respectively the environment for developing the software release, to testing the release, and thereafter to Quality Assure the release before it is released to the production stage.

Though container technology has been available many years, it is first now that it is mature enough to be used to handle important infrastructure and software. Through the usage of Docker containers it is now possible to run multiple versions of a software on a single compute node. Though there are normally many versions of a software in use. The model will base the release engineering process on a *multi-stacked environment*, which is using virtual machines to handle many versions of a applications, basically stack's the different software in a virtual environment.

To be able to handle the different versions of the software and combat the complexity, a tool is needed. A prototype of a tool needs to be *developed*, that enables the release of new versions of software through the usage of containers, and also the ability to specify the stages.

The *implementation* of this tool should be based on a way of completing the model designed, so that it may be possible to simplify the process of releasing new software versions, and keeping control of them.

From the implementation, it is possible to see the **benefits** of the process and the designed model, in the way that the complexity of the process of managing software versions and releases are mitigated. This is done through the abstraction of the server implementation, giving a broader overview of the site structure. Less work is then needed to be done for the complex operations of releasing new software, and keeping track of the different versions installed.

With this thesis, an abstraction layer is designed to help mitigate the complexity introduced in the myriad of new services and combination of services. This to combat the problems introduced with the now rapid pace of needed deployments, but still keep the existing processes of the businesses at a expected way that correlates to the implemented processes like ITIL.

Chapter 2

Background

This chapter introduces several technologies, concepts and applications which will be used in later chapters, and are of importance to this project.

In businesses today, there are a lot of processes involved to be able to produce results and handle the usage of IT services correctly. These processes are needed to be able to deliver software, handle existing applications running like the infrastructure, or when maintaining the existing software that are in use. The processes needed for this is a set of rules that are followed to ensure that the stability of the environment, and the expected reliability of the services are met.

Every business that handles IT today is in need of handling IT services in a professional way, to ensure that the specified needs are met, and the overall goals of the business are always present. The *model* of this thesis will design a such method for handling the release process of software, that ensures that the defined process parameters are met.

2.1 Services on the Web

Today, most of the services that are used are presented through the web. Spanning from in-house applications to information services on the Internet, almost every service that is available nowadays is a service that is accessible through a web browser. This is a transition from the period where most of information systems were presented through a desktop application, that needed to be installed beforehand.

The desktop applications, either if they were am application with a connection to a database, or a mainframe terminal, had a need for backend services that stored and presented the data to the applications. Today, the services on the web work in much the same way, with the exception that the client is moved into the architectural part of the service. The client's

web browser is now the new desktop application, and there is no longer a need for special software on the machines.

As this is a very good implementation method, most of the services today, that do not require any heavier integration with the computers operating system, use the web as a way of presenting the data to customers, users and everybody that needs access to it.

2.1.1 Technology of the web

There are today a lot of different technologies that make possible to serve services over the Internet. As designed by Sir Tim Berners-Lee [4–6], the HTTP protocol is the foundation of the implementation that enables services on the web. This protocol had its start due to the insufficiently amount of feature and slot FTP protocol (that at the time were used for file transfer), and HTTP were meant to replace it to be able to operate at the speed necessary to traverse hypertext links [6].

Later, this protocol has been extended with security features like HTTPS, which implements termination of certificate based authentication. All web browsers uses either HTTP or HTTPS to communicate with the web servers that presents the services.

Though the basics of HTTP are still the same, there has been a remarkable change in how these services are built up. Today, there is a need of being able to deliver reliable services, that also need to have a lot more functionality, but at the same time able to handle the load of many users.

But behind a service that are available on web, there are a lot more services that are doing some smaller operation to complete the end-user experience. This could be other applications, that are doing some specific work on some data, or a database server that only stores the data.

But these are only the services that have to be able to present the data to the end user. Most companies have more than a single service, and there is a need for many servers that can host them all. Nowadays, this is normally virtual machines, but these are some part of a stack that builds an entire environment of infrastructure needed to be able to run the services the end users want in the first place.

2.1.2 Changes in methodology and technology

During the last 10 years, there has been a paradigm shift in how services and applications are run. New technology, like cloud, has changed how services are being developed, and the processes around applications are changing.

Before virtual machines were a part of large businesses, the standard way of deploying IT services was with a *waterfall* methodology. Software were deployed and upgraded slowly, and new servers took long time to get ready, as they needed to be configured manually. Large multi-purpose servers that hosted many different applications were common, which were a challenge for dependency handling and management. Figure 2.1 shows the changes from the waterfall model, to *agile* and to where we are now, at a *DevOps* methodology driven market.

With the usage of agile and the technology of virtual machines, it were possible to provide new virtual machines for each role. This in combination with faster delivery time for virtual machines, and software, made it possible to get code faster in production.

With DevOps (Section 2.4.1), deliveries are now moving towards continuous release cycles, which provides faster *time to market* (TTM). Faster TTM ensures that customers get benefits of the changes made, faster.

The challenges today are that there are now Today there are many more applications than ever before, and many of them are loosely coupled. This creates complex environments and challenges for the operations teams, that now will need to handle both virtual machines and Containers (Section 2.6).

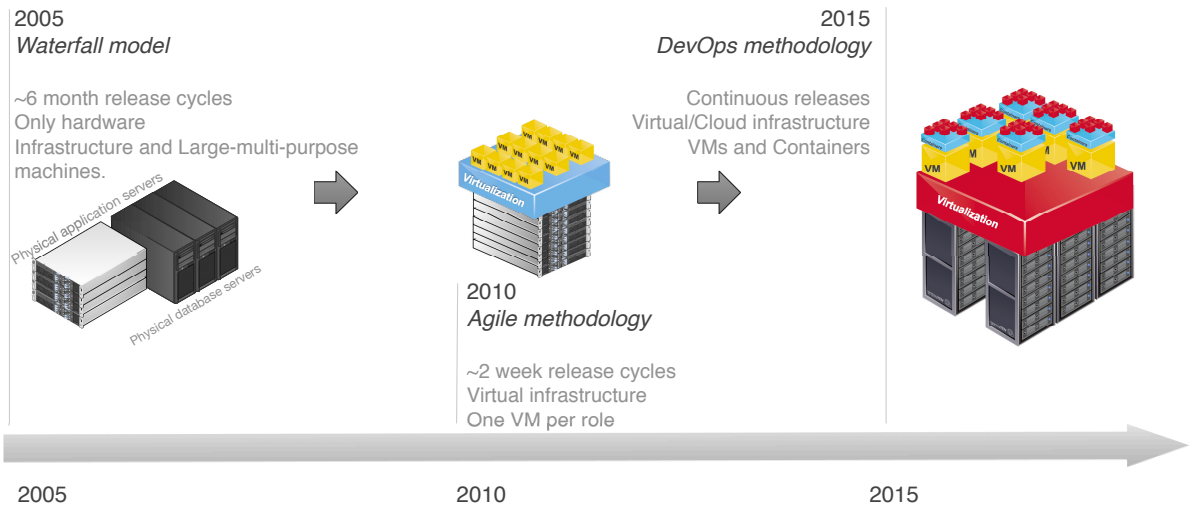

Figure 2.1: The development of infrastructure and methodology

2.2 Service architectures

Today IT departments are handling evolving systems and a constant stream of new applications and implementations. One of the things that make the work of IT departments even more complex is the different service architectures that are implemented as a result of the evolving systems.

The need to integrate these different applications is often a result from satisfying business requirements and needs [7].

Service architectures is a design of the communication of different applications. Combined the applications can provide information to build new applications. Most notable is a strategy called Service-oriented architecture (SOA), which is an architectural style for building loosely coupled distributed systems, that can deliver application functionality as services [8].

One of the more used features of SOA in larger enterprises are the service buses. These service buses are a service that can provide Web services that integrates the features of the underlying backend services. Figure 2.2 is an illustration of a conceptual service architecture of a business. This is a high-level view, where the individual components of the services are not shown. This shows the nature of the concept of an enterprise service bus, that connects the

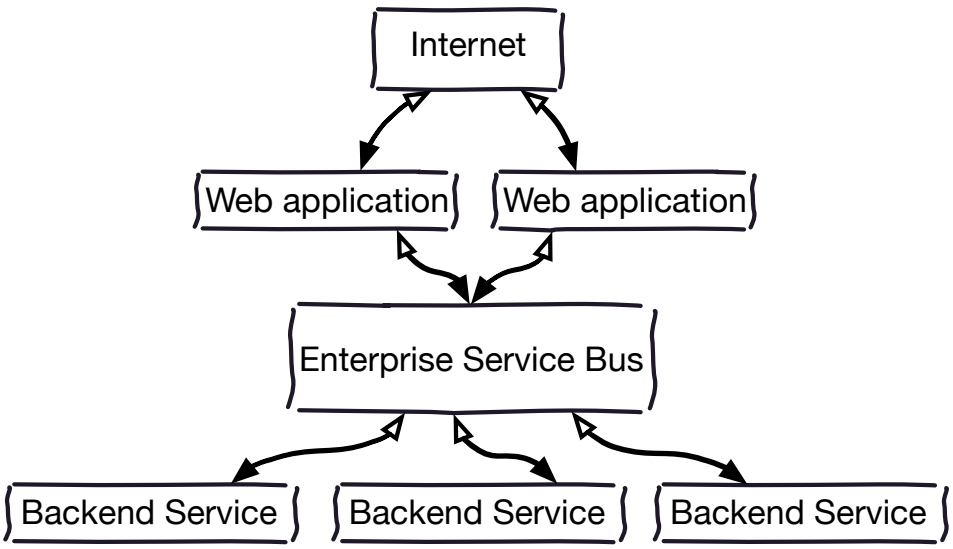

Figure 2.2: Service Oriented Architecture with ESB

In SOA there are three different kinds of architectural perspectives. The Application architecture, service architecture and component architecture [9]. The application architecture is the architecture of the business facing solutions which uses the services and integrates them into the business processes [9]. In figure 2.2, this is the web applications, which uses HTTP to communicate with the end user and the services provided by the ESB. The ESB on the other hand can connect the different backend services and provide a single endpoint for the different applications.

A similar approach to the problem of providing services are the Application Programming Interfaces (APIs). In relation to the SOA implementation, which normally is based on eXtensive Markup Language (XML) and Simple Object Access Protocol (SOAP) [10] while the API implementation

is normally through a Representation State Transfer (REST) interface or JavaScript Object Notation (JSON) serialization. Though there are different usages for the SOA and API approach, as SOA mainly focuses on presenting many services, the APIs can also be a subset of SOA.

There are more implementations than these two models, but they present a way of thinking in relating the different services of the business. With the usage of basic internet web technique's, discussed in 2.1.1, different services with business logic and data are connected together making it possible to create more advanced services to the end users. This is the trend that is needed to make businesses win. However, with the need for existing services to communicate, there is a complexity in the relationship between the different services. This is somewhat mitigated through the usage of ESBs, which abstracts the different underlying services. This in succession increases the complexity of the environment, where you can have several services that are interconnected and dependent on each other, that also are available through the exposure of a ESB.

2.2.1 Components of complex service architectures

Complex architectures are in play to be able to combine the different services in the different businesses, and to be able to deliver products according to business needs. Each service has its own different components that makes it able to handle the work that needs to be done. Following is a introduction to different components that are important to make the individual services able to operate in a service architecture.

Load Balancers
Load balancers are components in the network that help with the services achieve High Availability (HA). To achieve this, different functions are used in the load balancer ensuring that traffic is only sent to service endpoints that are actually working and splitting traffic between multiple nodes. Most load balancers also provide more advanced and useful features, like packet inspection, SSL/TLS termination and different load balancing methods.

The terminology of load balancers differs between the different vendors, but the usage of *frontends*, *backends* and servers are often used. Frontends are the part of the load balancers that are exposed to external consumers, and backends are the parts that handles the servers related to the service. Additionally, *healthchecks* are used to verify the status of the service at the specific nodes/servers, by issuing requests to a given address , and testing for expected output.

Web servers
There are many different web servers that are in use today, and this can

differ from application to application. While Apache has been the clear leader overall usage, Microsoft and the relatively new web server nginx is taking market shares [11].

Web Servers are normally used in conjunction with web applications to provide a unified abstraction of the applications running below. Though normally each language has a web browser that can be used, the normal consensus is that a web server should be used in front, Some of the benefits are security, add-ons, speed and basic logging.

Databases
Today, there are many different database systems that specify in different areas of data storage. Normal relation database like MySQL, PostgreSQL, Microsoft SQL and Oracle have been around for a long time, and are today used a lot. These databases uses the much known language Structured Query Language (SQL) to communicate with the databases. Even though the different database engines use the same language, the configuration and queries are not interoperable. Some different technologies like Object-Relation Mappers have tried to combat this by allowing the same query method to be mapped to the correct on different systems.

However, new technologies have emerged during the last decade, and NoSQL databases are now common. These databases can be used for big data handling like log storage, or data that would normally fit in relational databases. Apache CouchDB, MongoDB and Redis are among the popular NoSQL products. These new database technologies introduce new problems, where complexity is the key. Different software, version handling and clustering are just the tip of the iceberg. The NoSQL databases are not a replacement for relational databases for now, but introduce new components into the already complex environment.

All of these solutions complement the services which are known on the web today, and is part of a complex environment that is constructed to be able to provide the infrastructure and services that ensure that the business needs are met.

2.3 Release engineering

Release engineering is a sub-discipline in software engineering. According to '1st international workshop on release engineering (releng 2013)', by Adams et al. [12], release engineering is defined by *"all activities in between regular development and actual usage of a software product by the end user"*. The activities normally includes integration [13], testing [14], building [15], packaging [16] and delivering [17] of the software. While software engineering normally is related to the processes done by developers, release engineering is most important for System Administrators.

The release engineering that is implemented today has been helping enterprises to streamline the processes of delivering software. Through the streamlining and standardization of software delivery, it has been possible to support the increasing number of applications [18].

Release engineering has been around for a long time, but the defining of the processes has had a tight connection to software engineering. In 2013 the first releng *(release engineering)* conference were hold [12], and in 2014 Usenix organized the first Release Engineering Summit (URES). Up until now, this has been an area off business driven development, where the needs of businesses come up with the relevant tools and methodologies of best practices. This shows that the importance of the handling of new releases are ever more important, especially with the focus on DevOps (2.4.1).

There has been different ways of delivering software and releasing it in production over the years. Manually extracting archives and making the binaries were not thought of as bad practice, as the goal were to get the service up and running as fast as possible. Without any further automation, this process has been unacceptable for a long time. This is due to the need for reproducibility and stability.

This has been resolved with the usage of package managers. These package managers enables software to be packaged as rpm- or deb packages, which is numbered by version and can handle other package dependencies. These packages are managed with specification files, which defines what the packages should do during the different phases of installation and uninstallation. However, the problem with the package manager approach is that its been around for a long time, and technical debt, local adaption and customization has made the process complex and time consuming. Technical debt in this sense is a result of local packages that has been built that does not follow a strict standard. This can result in errors, and extra work to fix problems.

Release engineering includes the testing and integration of software, but businesses adopt this in different ways or just partially. A part of this is the environmental stages, that provide testing through multiple implementation of the software.

Stages: Normally implemented as three technical separated environments, that are designed in the same manner as the production environment. *Development/test*, *Quality Assurance* (QA), and *Production* (prod) are the normal implementation, but not every business uses all three. The test stage is where development and/or testing on the configuration the business uses are done. The QA stage is the stage that is used to ensure that the software is working, and if something is wrong in production, the software can be verified, and solutions can be tested. The production environment is where all the services available to the end users are running, and the usage of Service License Agreements (SLAs) are in use to provide goals for the sta-

bility of the services. Stages does also have a relation to security, where the production stage has stricter security measures than what of QA and test. This is needed as customer data is stored in the production systems.

Overall, the technical release engineers are supporting the services that are running in production. The phases, before software are deployed to production, are the parts that ensures that the software is working. This is done through both automated and manual testing, but and verified through the different stages. Now that the complexity of emerging technology and increasing amount of services that are supported, automation and standardization is needed to be able to follow in regards to release engineering.

2.3.1 Common challenges

Release engineering is hard to get right. Increasing amount of supported applications, and the need for faster delivery times through the phases of integration, testing, building, packaging and delivering create hard and difficult challenges for System Administrators. In relation to complex enterprise architectures and SOA, the complexity and loosely coupled services could be an operations nightmare [19, p. 74]. Each service needs to be updated when new software is ready, but the Service-level agreements *(SLA)* of the higher level services which depends on the loosely coupled services will fail when the service is not delivered.

At the same time, software needs to be updated faster, which has been one of the game changers since the beginning of agile. Developers now follow a more agile method, than compared to the old waterfall-model, but the delivery process and release engineering has still relied on the same methods. This is about to change with the introduction of DevOps 2.4.1.

2.3.2 Continuous delivery and deployment

Think of yourself as a system administrator in a large business. This business have a lot of different applications, some proprietary software, but also a large amount of different in-house software. Behind these applications are different software development teams that are working on new features and improvements for their application. The team of system administrators you are a part of is tasked with the job of managing the software, and ensure that it keeps on running. This also includes the deployment of new versions of applications. At this point the management and deployment of the software is done in manual steps, but with some automation at each different stage, through testing, building and packaging and installation of the applications. This takes a lot of time, and with dozens or scores of different applications, and many other tasks the backlogs will keep growing until something is done.

The challenges that appear in release engineering is mitigated through automation, and a part of this enables the process of continuous delivery. Continuous delivery is only a part of the release engineering process specifically tasked with the deployment of the new versions of the application. This enables the automated release of new versions that can be build and tested through continuous integration (CI).

As shown in figure 2.3 there is another form of continuous application releases, called continuous deployment. The difference being the manual step to initiate the deployment to production. None are these techniques are "right", as they are dependent on the business needs.

The automated deployment processes ensures that the problems with the error prone processes of deployment can be mitigated through faster but smaller releases that are deployed automatically [20]. Each of these deployments are referred to as a release, where the release is a fixed version of the code presented in a version control system. In addition the release should contain configuration for the different stages (development/QA and production). This is tightly connected to the common IT operations methodologies, and a part of the important workflow in IT operations.

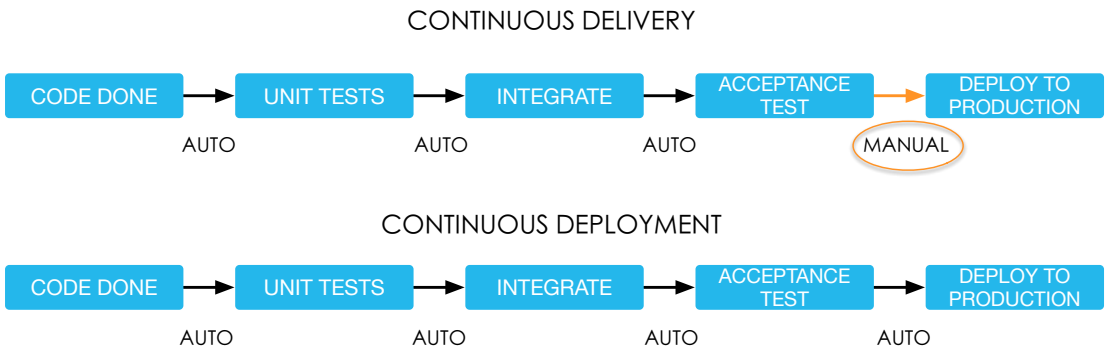

Figure 2.3: The difference between continuous deployment and continuous delivery

2.4 IT operations methodologies

IT is a fast evolving field, with many different ideologies and methods to deliver products and service. IT permeates through every part of todays businesses, and almost all departments are today depended upon it. The need for methods that enables fast releases, stable and compliant services is therefor extremely important.

ITIL (*IT Infrastructure Library*) has been the de facto standard for implementing IT service management for many years [21]. It is a set of best practices derived from common practices, that enables businesses to ensure consistent handling of services. ITIL comes from a time when there

were a lot fewer services, and they moved a lot slower than what is seen today. Today it is normal with many releases daily [22, 23].

As a result of the increase in amount of services, and the needed work to get through the ITIL process, new methods describing service management have emerged.

2.4.1 DevOps

According to John Willis [24], DevOps is coming from three main threads. This is the Agile Infrastructure Thread, the Velocity Thread and the Lean Startup Thread. Agile has been around from 2001 [22], and has the merits in continuously improving the software through iterative steps. DevOps is a logical continuation of agile, since the code is not done when development is done coding, but rather when it is in production [22]. Agile development does this by breaking up larger tasks, to tasks that require little planing, and are fast to implement.

At the Velocity conference in 2009, two guys from Flickr had a talk on how they managed to handle 10+ deploys per pay, through the collaboration between the developers and operations people [25]. Both the achievement of managing 10+ deploys and doing it through collaboration were a massive boost for the DevOps movement.

The success of DevOps is partly related to the understanding that code does not have value, until it is in production [22], and the need for communication and collaboration to be able to deliver the code to production faster. This of course increases the amount of changes that is needed to deliver the code into production. A study done in 2003 shows that the most cases of failures are due to operator errors, whether it is due to configuration or lack of testing [26].

The high performing organizations two important tools that have helped in decreasing the failure rate during changes. These are first and foremost Version Control and automation [23, 27]. This has also an effect on the amount of changes that are possible, which also is shown through the reduced lead time where DevOps has been implemented [23, 27].

With the automation that is being implemented, some of the complexity of handling all the different services running are being reduced, and System Administrators are able to focus on improvements, rather than keeping up with the needed work. It is however important to note that DevOps is not a replacement for ITIL, but are compatible and describe many of the capabilities needed in order for IT Operations to support a DevOps-style work stream [22].

2.5 Cloud computing solutions

Cloud computing has been around for some years already, but it is first now that businesses are beginning to utilize the new solutions. Software providers are now supporting open source solutions like OpenStack to provide support and reliable releases of the normally quite complex software.

The result of cloud computing can be reduced down into three main categories, namely infrastructure-, platform-, and software as a service. These three categories include a set of software that provides solutions on a specific level of the virtualized stack.

Infrastructure as a Service
Infrastructure as a Service (IaaS) provides the main platform and utilities that are needed to host machines and software. This is usually where you get your virtual machine, but need to do all the configuration and installation of services afterwords. This is the lowest level in the cloud stack, where there are only the basic infrastructure that is implemented.

Platform as a Service
With Platform as a Service (PaaS) a layer of complexity is removed from the user. The service provided could be only a web server, with the capability of running your code, databases or load balancers that only need your basic site specific configuration to be working. Here complex services are offered to the user that no longer need complex knowledge on how every solution works. An example of a PaaS solution is Amazon Web Services or the open source solution OpenShift provided by Red Hat.

Software as a Service
Software as a Service is the highest level, and is a way to deliver software that is hosted in a cloud. This could be mail and collaboration tools, or customer relationship management (CRM) systems, that you pay for the number of users or for your business. What you are paying for is the possibility to use the service, and not for the configuration or installation of it. These solutions help businesses lower IT costs by buying cheap solutions in the cloud rather than hosting it them self.

2.6 Containers and container technologies

Containers has been around for a while, and was first introduced in 2004. Now however, they are now more popular than ever, and some of the reasons for this is the new implementations and support [28].

Containers are another form of virtualization. In relation to the now normal virtual machines, which uses a hypervisor like KVM, Xen or VMware ESXi, containers use functionality directly in the Linux kernel [28].

In conventional virtualization, the hypervisor is the software that handles all the abstraction of the hardware that are presented to virtual machines. As shown in figure 2.4 a virtual machine is running on top of the hypervisor. The hypervisor can either be running directly on top of the hardware, or on top of the host operating system. Either way, the virtual machines are in itself complete operating systems with all the necessary packages and applications on top.

Containers on the other hand, are based on functionalities that separates the processes at the operating system level. This means that (as shown on the figure 2.4) only the necessary dependencies of the applications that should run are loaded in the container. The figure 2.4 shows the differences between virtual machines and containers. On the left side, are the virtual machines. This is a case of a type 2 hypervisor, but the layers of "host os" and "hypervisor" could be combined with other hypervisors. Type 1 which runs directly on top of the hardware is better compared to containers.

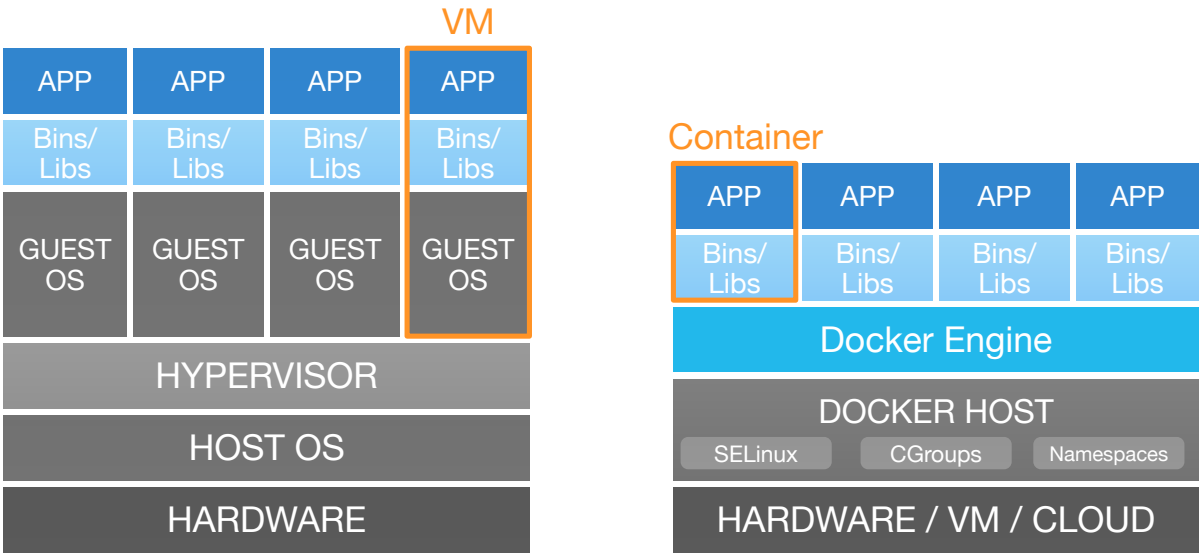

Figure 2.4: Virtual Machines in relation to Containers. In this case using Docker.

2.6.1 Containers compared to VMs

Due to the lack of a hypervisor, containers are of a more lightweight than virtual machines, and does not contain the same amount of overhead as with some hypervisors. Although some overhead exists, the benefits of containers extends to convenience, faster deployment, elasticity and better performance [29]. Compared to VMs, an example of this is the achieved boot time of a container in relation to VMs. As the VM needs to go through the boot process, a container only needs to start the processes, which can take around a second. Fast boot times of virtual machines span from ten

seconds and up. This is a powerful feature for the containers when in use in relation to continuous delivery.

Containers are more facilitated for the needs of todays DevOps (2.4.1) work style, and have many benefits over VMs. The following list present some of the benefits of containers [28–30]:

- Isolation for performance and security
- Lowering costs by optimalization of resources
- Provide tools for faster application development and delivery
- Dependencies are handled, and always consistent
- Issues with platform dependencies are mitigated

2.6.2 How does Containers work

The basis of todays containers are three main kernel features. Namespaces, cgroups and SELinux. Namespaces and cgroups have been the basis from the start, while SELinux has been added later. Namespaces were implemented in the Linux kernel in 2007 [31], and have matured since its release. With containers today, namespaces are used to provide the isolated workspace that is called a container, and is the basis for containerized virtualizaton.

There are currently six different namespaces. Five of these are in use to create a Docker container [32]:

- ***pid* namespace:** Process isolation
- ***net* namespace:** Network isolation
- ***ipc* namespace:** Inter-process communication management
- ***mnt* namespace:** Mount point management
- ***uts* namespace:** Isolation of kernel and version identifiers

Control groups, called *cgroups*, are another technology that is in use, to restrict the resources that a container has available. This could be to limit the available memory of a spesific container.

The latest addition, that are not a part of the containerization in itself is SELinux. SELinux is important when running container hosts, where it is used to protect the host and containers from one another. SELinux enables another layer of security to ensure that attacks are contained if the container is broken out of [33]. Another alternative to SELinux is AppArmor, which has the same effect as SELinux.

Newer advances has made container based virtualization more popular and easier to use. This has resulted in many new implementations in ways you can run containers, such as Docker and Rocket. First we look at the

technology that were available before containers existed, and the evolution of Linux containers.

2.6.3 chroot jails

Most of the container applications today build on the basis of chroot. Chroot or *change root* is an old feature in Linux which changes the apparent root directory for the running process [34]. This ensures that the process does not have access to the resources outside the environmental directory. At the time this feature was used to trap and lure crackers and study them as shown in 'An Evening with Berferd: In Which a Cracker is Lured, Endured, and Studied' [35]. Chroot is not totally secure, and it is possible to break out of the jail, but it enables a extra level of security. Today, new technologies are available that can handle more aspects of a jail than chroot.

2.6.4 Linux Containers - LXC

LinuX Containers (LXC) were one of the first implementation on containers on Linux that made strides towards integration container support into mainstream Linux kernel [36]. It is both userspace tools which enables the creation and management of containers, and a kernel patch that handles the low level container management operations. The focus of the group behind LXC is as they state *"containers which offer an environment as close to possible as the one you'd get from a VM but without the overhead that comes with running a separate kernel and simulating all the hardware" [37].* However, the configuration and setup needed to get started with LXC is complex [36], and the creation and maintenance of containers are non trivial compared to other solutions, like Docker.

2.6.5 Docker

Docker were released in 2013 by the company dotCloud (now Docker Inc.), and is one of the most popular Container softwares available. It is an open source platform for developers and sysadmins that enables the building, shipping and running of applications in containers.

Docker began with a userspace program for running Linux containers (LXC). The difference is that it enables the packaging of applications, and normally just one application in a container. From Docker version 0.9 the LXC backend were changed out with their own project libcontainers, that connects to the different functionalities in the Linux kernel. As mentioned earlier, these are cgroups, namespaces an SELinux, but there are also other functions like netlink, netfilter and capabilities that are important in creating a secure container, that are used with Docker.

The idea of Docker is about standardizing containers. The idea comes from the shipping problem, where the solution has been physical containers as a fundamental unit of physical delivery [38]. This has given Docker some key features: standard operations, content-agnostic, infrastructure-agnostic, designed for automation and industrial-grade delivery [38]. This means that the container looks the same, and it should not matter where your run it, but the inside is different.

Docker is an open source platform with many contributors and other vendors creating software that helps in the regards of deploying and running containers. This makes Docker an extremely fast moving technology, and it is constantly updated with new features and extending tools.

One on the reasons Docker has become so popular, is its ease of use. The following shows how a new container is downloaded from the Docker Hub (which is an online registry with different container images) and then a basic shell is being executed in it. With shell access, new software can be installed as it were an Ubuntu machine.

```
# First the image is downloaded from Docker Hub
$ docker pull ubuntu

# Run the container and execute a interactive shell
$ docker run -i -t ubuntu /bin/bash
root@06d1e5ae400e:/# ps
  PID TTY          TIME CMD
    1 ?        00:00:00 bash
   19 ?        00:00:00 ps
```

2.6.6 Rocket

The CoreOS project has announced that they are working on a container runtime called Rocket. It is an alternative to Docker, created as the developers of CoreOS wanted a simple composable building block that "we can all agree on" [39]. Docker is developing into a platform, with a monolithic binary that runs primarily at root [40]. Rocket is therefore created as a lightweight alternative to Docker.

2.6.7 New cloud solutions for containers

There are a few fairly new projects that are designed with containers in mind. They utilize the possibility of small lightweight nodes for distributed computing, and can provide small-footprint operating systems that are optimized for running containers.

CoreOS, Atomic and Ubuntu core The three different projects provides an operating system platform that are created to host containers [39].

They are intended as lightweight hosts that are perfect for large-scale cloud container deployments. With a lower overhead of default installed packages, they are consuming less resources in the cloud environment, and are specifically tuned for better security for containers. CoreOS is a new operating system, whereas Atomic builds on the Fedora/RHEL/CentOS stack, and Ubuntu core on Ubuntu.

OpenShift, OpenStack and Apache Mesos OpenShift is a cloud application platform that provides Platform as a Service functionality. This open-source solution provided by Red Hat enables the hosting of code either in a public or private PaaS cloud. By launching a cartridge as it is called, you get access to a git repository that you can push the code into. This will launch the application from your code. Currently OpenShift has implemented these cartridges as native containers, but the intention is to implement Docker as the provider for containers in OpenShift.

OpenShift as the IaaS cloud platform that it is, has currently a project that enables the implementation of Docker in the compute service called nova. This enables native docker containers to be launched, and take advantages of the powerful features that OpenStack provides.

Both OpenShift and OpenStack are solutions that has been widely implemented, and their focus on containerization shows that there is still unexplored possibilities with these solutions, that we are likely to see over the coming years.

Apache Mesos is a new way of scaling out your data center, and enables the abstraction of CPU, memory and storage. It can run containers like Docker, enabling scalable application in a cloud fashion. This is a relatively new technology that are adopted by large companies. Through their continued work to enable the usage of containers, there is a future for the large-scale container usage with Apache Mesos as the underlying platform.

2.7 Relevant research

This thesis involves several different areas of computer science. The different areas are connected to release engineering, software management and system administration in general.

Different issues have appeared with the migration to cloud computing, and at the same time the businesses are being drawn towards a DevOps oriented approach for release management. Large service architectures are hard to administrate, and it takes a lot of resources.

Most of the relevant research for this thesis is related to work done in the areas of interest. No relevant scientific work is done on the release engineering processes of handling software versions and releases through

containerization. The processes and methods used today, are mostly a result of the business needs and their implementations.

At the same time there is a change in how release engineering is viewed, and during the latests years, new conferences have appeared with this specific focus. This includes USENIX Release Engineering Summit (URES) which first appeared during the USENIX Federated conference Week in 2014, and the RELENG conference [12] in 2013.

2.7.1 TOSCA

Topology and Orchestration Specification for Cloud Applications (TOSCA) is an emerging orchestration standard for Cloud Applications [41]. The goal of the standard is to enhance the portability and management of Cloud applications, and provides ways to enable portable and automated deployment and management of applications [42]. 'Standards-Based DevOps Automation and Integration Using TOSCA' [43] uses the TOSCA model to design an approach of a orchestrated environment (Infrastructure as Code) based on existing DevOps tools like Chef and Juju. The TOSCA standard does not provide any software, but an implementation of a graphical interface for modeling of applications has been prototyped [44]. An integration of configuration management and cloud management has been done [45] to further the unification of TOSCA.

The work done in the field is done towards defining architecture (or infrastructure as Code) which enables automation of the whole management of services, to reduce costs, make management tasks less error-prone [45] and reduce complexity [42]. However this project does not work on the implementation or process of release engineering, and lacks process in regards to needed day-to-day operations.

2.7.2 Towards the holy grail of continuous delivery

Through a study of three different project teams that were striving towards practicing continuous delivery. It suggests that architectural design decisions are equally important for projects to achieve goals such as continuous integration (CI), automated testing and reduced deployment-cycle time [46].

2.7.3 The Deployment Production Line

Todays release engineering processes needs to strive towards automation. If left unchecked the environment can, over time, become a long and error prone process [20]. Humble, Read and North defines the process of delivering application to production as a deployment production line, where the different parts are assembled along the way. This means that

the important of testing throughout the different stages and environments become utmost important, where the automation of these processes are the key.

Humble, Read and North provides two principle for a successful build line. First the organization that defines that each build should be completed as quickly as possible and should be linked to the version of source code. The second principle is to separate binaries from configuration. These principles comprise the deployment production line, which facilitates a multi-stage and automated workflow that enables multiple teams to build and deploy into complex environments.

2.7.4 MLN

There are many different tools that are working to combat the complexity that is related to the emerging cloud solutions. One of this project has been Manage Large Networks (MLN), which has enabled the management of large number of virtual machines through its configuration language [47].

The development of both the methodologies and technologies for release engineering and the usage of containers, is progressed by the improvements and open-sourcing of the industry-driven processes. Research has until recently provided few improvements on the delivery aspects of release engineering. Docker is an example of one of the tools that has sprung forth through the need of easier solutions.

2.8 Usage Scenarios

This thesis will work towards mitigating some of the manual processes that are being used today. To understand, and be able to see the result of the current work, different scenario's needs to be defined beforehand.

Imagine yourself in a important business, that handles bleeding edge products in a competing market, with demanding customers. The business is able to develop new features quickly and improve their application continuously. You are a part of the IT operations department, in a team of System Administrators, tasked with keeping the software running, and provide the needed technology that help the business win.

The state of IT shows that IT is keeping to the SLA in regards to uptime and stability, but each release takes weeks and months. And when the new releases are finally ready, they often contain large and complex changes that has a chance of breaking the production systems, so called high-risk-changes.

2.8.1 Service state management

With many applications that are distributed throughout your datacenter, it is important to be able to keep track on the different versions and their relation to other application. A team of software developers will not have the necessary knowledge of the relations in of the different application dependencies beyond their own applications. When moving to a more lean and DevOps oriented organizational structure, it is important to have the tools necessary to be able to take the right choices when deploying new versions of software.

IT operations needs to be able to understand the relation between application, the different versions, and where they are running, to be able to support the environment during destructive issues. Today, the most optimal solution for this problem is with the usage of configuration management systems, but with many different applications spread over many nodes, this does not give an apt description of the current service state.

With an understanding and computational relation in a service state management tool, many different benefits occur. Most notably the possibility for visual representation of sites and services, and exportation/importation of site-vide configuration, enabling recovery solutions for containerized environments.

2.8.2 The release of a new service

A new team of software developers have created a new bleeding edge system. The PR department have gone wild and brought advertisements for a heck of a lot of money. It is supposed to be in production by tomorrow.

This would normally take weeks, but with the IT operations new system for deployment with containers, this should now just take a couple of minutes. The software is stored in a version control system, and the master branch contains a tagged commit with the current version that is supposed to be deployed.

Viewed from the developers perspective, what do they need to do to get their application into production, and how will this affect their process.

2.8.3 Deployment of a new version

The software has been in production for some time, but a bug has been discovered in the code, that due to external causes have caused a malfunction with a registration form, as the data is stored in an external service. Both services needs to be upgraded in order for the system to work

properly again. The fix has been written, but the IT operations do not want to do a manual fix, so the software developers are forced to do it the new way, and commit and tag the specified bugfix that will enable the software to be upgraded.

2.8.4 Management of the service

New versions of software are not directly deployed into production. It first needs to go through thoroughly automated testing, and the different stages of the deployment production line [20].

Businesses have adopted many functions to enable high quality releases, where different stage installations is one of them. The application is installed in development before it is installed in Quality Assurance. After the application has been accepted through QA, it can be installed in production. This means that there are always many different versions of the applications running. Production will have the oldest version, QA will have a more recent version and development will have a more recent version. This needs management, and without automatic features, needs to be done manually.

To actively manage a single service, the versions in their respective stage will need management, but also the numbers of instances of the different stages. In dev you may only need one instance, but in QA you may need two, and so on.

Chapter 3

Approach

This chapter provides an overview of the approach and its different phases, where the design and implementation of the desired goal derived from the problem statements appears.

Each business has today its own approach towards the implementation of software deployment pipelines. Depending on how long-lived their applications are, many businesses are experiencing slow release cycles, due to the old fashioned release engineering processes being used, that are not in all compliant with the new cloud solutions movement for scalable environments.

This results in the need for reduced complexity, faster release cycles and a common process that enables better standardized solutions. This chapter will describe the process of answering the problem statement: **Design and develop a model in order to reduce complexity in release engineering processes, using multi-stacked container environment.**

This problem statement explores the technology of managing software based on container technology (as opposed to virtual machines and software packages), and the combination of stacked environments.

To ensure the possibility and feasibility of the proposed model, the second problem statement states: **Develop a prototype that implements the model designed based on the first problem statement**.

By these two different problem statement, the approach and the coming results chapters are defined by each different problem statement. As shown in figure 3.1, the outline of the approach begins with the *problems* at hand, where we first need to define the properties, and with them, the terminology to be able to proceed to the defining of the model and design of the release engineering process with containers.

This results in a prototype or proof of concept, that enables the simulation of different scenarios (defined in section 2.8), which are important day-to-day operations for IT departments.

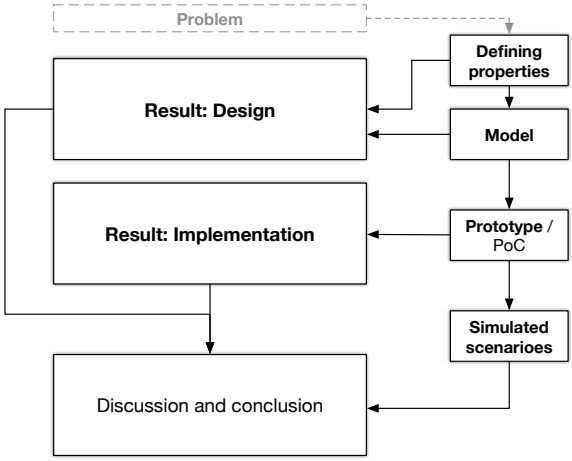

Figure 3.1: Outline of the approach

3.1 Design phase

The design phase will focus on the development of a model that defines the different aspects of release engineering and management through the usage of the emerging container technologies. This is based on the first problem statement (defined on page 2), where the focus is to enable the reduction of complexity in the processes of handling release engineering processes.

This phase is the most important aspect of this thesis, that will result in a model that enables a strict set of rules and constraints. This ensures that automation and reliability can be enforced, and that many manual processes can be eliminated through the automation. The model will, based on this, enable the reduction of complexity for the end user.

The model should be the formal description of how a complete environment of running services should be set up and maintained with the usage of containers. This means that all the applications that are needed to create a services and user endpoints, should be handled and administrated based on the model. Figure 3.2 shows a conceptual service architecture, that is closely related to the underlying architectural design that the model should work with. This includes different stages (dev, QA and prod), multiple instances of a service for high availability and their relation. Though this could be done with virtual machines, this thesis will focus on the design of the model with the usage of containers.

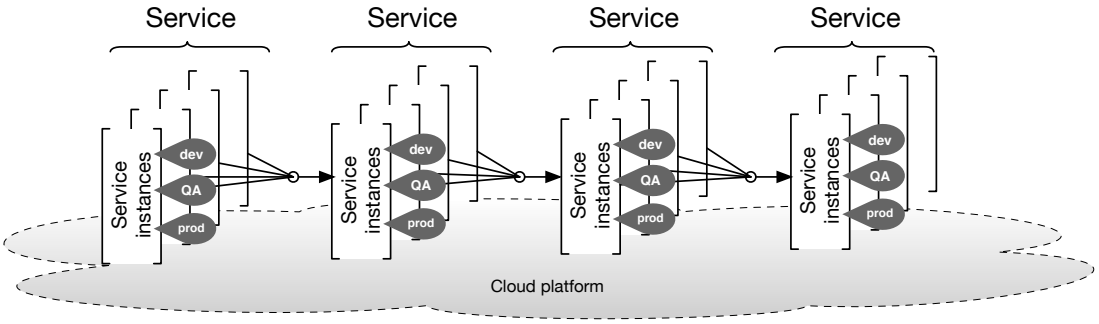

Figure 3.2: Basic representation of different services, running multiple instances with some relation and different stages.

The designed model will be able to handle the different aspects of the service management, and the processes related to releases of new versions. However, different properties needs defining, that will have a relation to the actual environment.

3.1.1 Model scope and properties

To enable the creation of the model, there are different properties that needs to be defined. These are part of the model itself, but does also outline the needed functionality. The property is related to the model in the way that it is a set of what is needed in the model to define the different parts the model needs to handle.

The following properties define the scope of the model.

- Terminology and component definition
- Rules and constraints
- Operations or process, based on scenarios

The **terminology and component definition** defines each of the different parts and components of the model. This contains both the naming and formal notation. This enables the definition of **rules**, that makes it possible to define different restrictions and requirement for the different **operations** done on the environment.

3.1.2 Model

The defined properties are the basis of the model. These are part of the features, functionalities and operations of the model, which needs to be properly described in the model.

The model will be described in text, formal descriptions and diagrams to help the reader more easily understand the concepts. The formal

description in the model enables precise definition of the properties and underlying actions.

The given scenarios (defined in section 2.8) define the different operations that is the main focus of the model. The model is not limited to these scenarios, and enables through the definition of state, the possibility of adding new scenarios and actions. The following processes defines the given scenarios in the model, as a set of standard processes.

Process 1: State of the environment and services

Process 2: New service to be deployed

Process 3: New version of a service needs to be deployed

Process 4: Management of a running service

Each set of these scenarios need to make use of their related **terminology**, **rules** and **constraints** to enable the verification of the compliance of an environment in regards to the model. They also contain different actions, which defines something that happens, and these actions needs to be taken into the model. Each action should be described, and modeled as a function, with the needed input parameters and output. Graphical representation of the different processes are also needed.

The description of state Will enable some of the basic features that are needed to build the basis of the model. The state of the environment and its services are important in every aspect of the model, and enables the verification and insurance of compliance. State will describe how services are viewed and give an overview of how the environment is right now. If new actions would be performed, it is possible to verify that the state is compliant with the model after the action has been performed, without it actually being done. A so-called dry-run. The correctness of state will therefor in this case be the result of the current state (what the environment actually look like) plus limitations.

Definition of constraints The verification of state will depend upon the constraints and rules that are presented in the model. These constraints are information or algorithms that describes the expected or wanted state of state. They are important in the way that they enable different features, such as high availability, where the constraint or rule could be to state that a service should run on at least two servers.

Actions The model by itself will focus on the defined processes, but in each of these processes there are different actions that can be taken. These should be defined in the model, but the model should also support new actions being implemented later on. This is important, as the different actions may be based on the design choices of the implementation. The

different actions should in the model be described and modeled in a way that enables its programmatic implementation. This also includes possible constraints that are needed, where the different constraints also may be altered.

3.1.3 Expected outcomes

The expected outcomes of the design phase is a model. The model will give a *formal definition* of the terminology and needed forms to enable a functioning model. The formal definitions will define the building blocks of the model, along with the important terminology and definition of the core items. Most important is to define the needed information, the state, of an environment and how this will affect the models *actions*, and *processes*.

Different processes will outline expected use cases, but these should not limit the functionality of the model that should be extendable.

The model that is to be developed is expected to be agnostic to the technologies used, but be enabled for the common go-to-technologies available. This also ensures that the model should be usable by everyone with the building blocks that are provided.

This will help with the challenges that release engineers are experiencing, with the increasing amount of releases and new applications.

3.2 Implementation phase

The implementation phase will focus on the second problem statement. This states that a prototype implementation of the model is to be developed. This will enable the verification of the validity of the model. The implementation will however only cover a subset of the overall needed functionality. This is due to the time constraints on the implementation phase.

3.2.1 Environment

To enable an implementation of a prototype, that is applicable to any given environment, a basic sample environment should be configured, to enable the testing of the prototype.

Though the model should be agnostic to the underlying technology, the implementation need to be implemented through specific technology. Docker is currently the largest platform that provides solutions for distributing applications through containers. It provides a lot of functionality, and is supported and implemented at large scale. This makes Docker a good choice for the providing of services in this model.

This means that an environment should be configured where the Docker software is running and able to run containers, but at the same time is available from a central node, in which the prototype can run. Though there are many different solutions for orchestrating where and how the containers should run (software like Google Kubernetes), this is not part of the scope of the prototype, and therefore not more relevant than its possible support in the prototype.

The environment should with this consist of at least three nodes. One is the central machine, which the prototype should run on, and two other nodes in which the Docker engine is running, and providing the platform for running containers. For simplicity, the central node would also consist of a load balancer, so that the implementation could dynamically update the load balancer.

3.2.2 Prototype

The prototype should be designed to enable the verification of the model. It should also enable the basis for a tool that enables the management of services that is based on containers, and through it both be able to view and manage the environment in which it runs.

This means that the prototype should enable the different items, processes and terminology as the model implements. The processes with the underlying actions, enables the functionality of the prototype, which will ensure that it is possible to deploy a new service, but also add new versions of the service, as defined in the approach section for the model.

Through the implementation of the processes outlined, it becomes possible to perform artificial testing on the prototype. This means that no real services are deployed, but the containers that are deployed only contains testing services, which can resemble real services. This results in the use of real technology and methods, but not real applications. This ensures that time is spent on the prototype, rather than the services.

The prototype should enable the implementation of the underlying services, which means that it needs an integration to the Docker engines, which enables the deployment of new containers. It should also be possible to integrate the prototype with a load balancer, but this should be implemented only as an extended feature, if time permits.

Python will be the programming language, in which the prototype will be programmed. This allows for the use of the docker API implementation, and the use of sockets to the HAProxy load balancer. Python also enables fast implementation of the large prototype, this will result in.

As python is a universal language, the platform on which it is built, is not especially important. However, to better illustrate a business like environment, the Linux distribution on which it is run, is CentOS.

State and actions

As one of the most important parts of the model, the state, needs to be a central part of the implementation. A good implementation of state, will enable the verification and enforcing of constraints, but also flexibility in the different processes which is enabled. The constraints also needs to be implemented in the actions, to ensure that the changed made to the environment is in compliance with the model.

The different actions should be implemented through functions that enables the necessary functionality. How many of the actions that are implemented is up to the time constraint, but they should enable the verification of one process, as to ensure that the compliance with the model will be real.

3.2.3 Apprising properties

To ensure that it is possible to conclude if the model is enabling the reduction of complexity in release engineering processes, there is the need for the prototype implementation of the model. This will enable automation of deployment of services, but also the abstraction of the deployment process.

But to enable the verification of this, apprising properties are needed, as to enable evidence for the reduced complexity. The following are surrogate variables that illustrates the level of complexity.

- Time to completion
- Needed steps
- Expertise, prerequisites and needed understanding of the environment
- Reproducibility

This is comparable to manual processes, where each of the steps that the model implements would need to be done separately.

3.2.4 Expected results

The outcome of the implementation of the model is expected to result in the reduction of complexity of the release engineering processes. The different processes which is outlined in the design, should be possible to implement in the model, but due to the time constraints, not every process can be implemented to its full potential.

The implementation should provide a working framework, that enables the implementation of additional processes and actions. The framework

should consist of the state, where the terminology and items from the model is implemented.

Through the prototype, it should be possible to test out the processes, and enable the deployment of services through integration to Docker. This enables the verification of at least one process, where it is possible to verify the different actions done to the state. This should be done where at least one action succeeds, but there should also be cases where the constraints of the model limits the actions, and restricts the changes done.

Chapter 4

Design

In this chapter the design of the model will be addressed and presented. The model is designed, and specific scenarios are taken into account to better understand the underlying needs of this model. An overview of the model, and a look at its different definitions and terminology is needed to be able to understand the different aspects of the model.

4.1 Model overview

There is a need for todays environments to be controlled in a strict manner, to be able to handle a multitude of running instances of services. This includes the ability and possibility of automation and standardization. Through this arises the need for a model. A way of controlling and ensuring that the services that are being run, are in compliance with the required needs of the system environment, and to be able to perform tasks on the environment.

This model is designed to be agnostic with respect to the underlying technology, but is designed after a microservice and container based environment. Microservices in contrast to large monolithic services, are hard to maintain in a cloud environment. They are normally small applications with a specific subset of operations needed to enable the overall functionality of a larger service. Containers are a way of deploying these microservices, as some of the features it implements are fast deployments, and dependency handling. This enables a lot of different features, but also creates new challenges, where management, orchestration, life cycle management, and complex networking with containers not yet have any solid solutions.

The model is based on, but not limited to, the different scenarios that are outlined, but before these can be defined in the model, different key aspects of an environment needs to be defined. Figure 4.1 shows the composition and relation of the model. The state of the environment, with its constraints

is a key aspect, along with the different processes, and actions. The terminology is important for understanding of the different aspect of the model, as it defines the different terms and items.

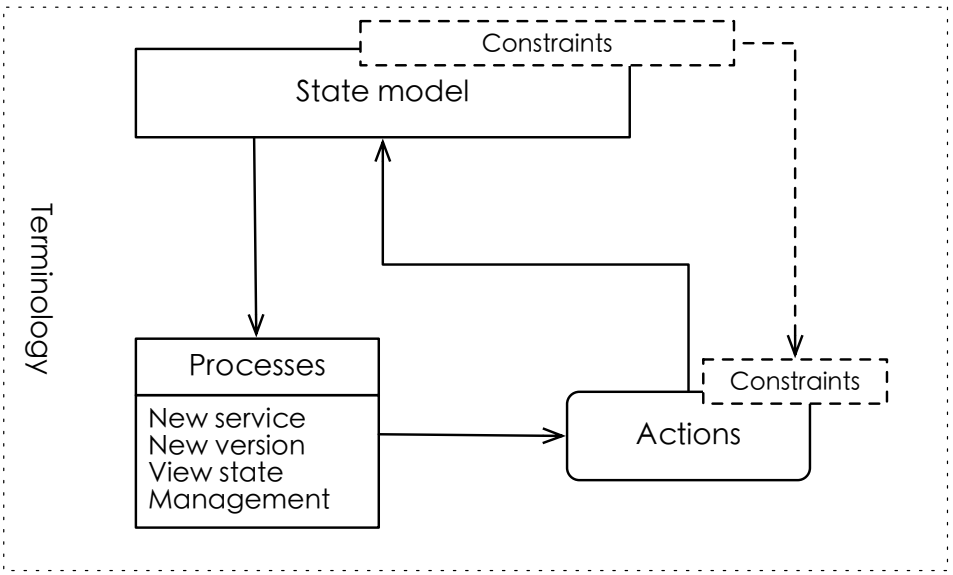

Figure 4.1: Model overview

4.2 Terminology and definitions

To provide a model that is both understandable and is descriptive, different terminology and definitions are needed. The use of containers in service deployment is beginning to mature, but there is still no complete terminology for service environments with containers. This is needed to be able to provide a model that is describes the state and order of an environment.

Firstly some key definitions are needed, as *processes*, *actions* and *constraints* are key features of the model.

4.2.1 Processes

A process is previously in this thesis referred to as a scenario. The scenario in relation to a process is more related to a intention with a setting. A process on the other hand is an operation that contains different actions.

The process can be defined by the different actions that are related to it, to enable the process to achieve its goal. A specific action contains different constraints, but the actions with constraints could be performed to do some action on services and its containers. This is loosely outlined with the following formula.

$$Process = [Actions, \dots] \; with \; [constraint, \dots] \; on \; STATE$$

A process can be seen as a overall operation, that works to promote a desired goal. A process in our daily life, would be to go to the store and buy a meal. The desired outcome is known, but the specifics of how this is done are not defined. These specifics are defined as actions. The process can have multiple actions, which implements a small part of the process. An action of buying a meal, would be to pay the cashier for the items that is bought.

4.2.2 Actions

An action is an operation that does a job. It can be seen as an algorithm or the act of doing a specific task. This is related to the process where, the process defined the overall outcome for the action. An action could achieve the overall goal of the process, or just a small part of it.

In a programmatic way, and as this model is referring to it, the action is mutable with a function. It is assumed that an action can have specific input data, and possible output data. It can also enable other actions or be directly responsible for them. Yet an action does not necessarily do a task that alters the environment, or any of the stored information. The different actions can operate on different levels, where an action can show how the environment looks, try to alter the information and check if that meets the constraints defined, or actually altering the environment and the information.

If actions should be related to the creation of a new services, there would be need for multiple actions that perform these different tasks. One action would give out information about the information that is currently stored, while another action would check if the new service would meet the configured constraints. The last action would then create the actual service, and do the changes to the information and environment.

4.2.3 Constraints

Each action has the possibility of having constraints. A constraint is a limitation or restriction to actions, items or the state, that should be defined to ensure that an environment is compliant with the needed specifications. These constraints are intended to reduce the risk of actions. The different constraints are also important as they ensure that the state is always correct, and unexpected situations are limited.

Identifying the different constraints is not an easy task, and as such the defined constraints can be extended to a much finer grain. The constraints do however contribute with the ability to closer define the actions as to limit the negative effects. An example of a constraint, would be in the case

of having a service that is important to the business, and always needs to be available. This would result in a constraint that identifies this need, and specifies counter measures that ensures that the risk is lowered. This could specifically be to define a constraint that says that the service need to run on two servers at all times.

4.2.4 Terminology

The terminology in this model includes abstract service terminology, and more technical informational definitions of components related to containers, and cloud solutions. It is also important to understand the terminology, and each term does also include different factors that are important to mention. The following are the most notable that need to be mentioned and described.

Endpoint

The name endpoint comes from the WDSL definition [48] where it is a abstract or concrete service. In this model, the endpoint is an explicit service that is **usable** by an end user. It is the culmination of other services that enable the possibility to provide the data needed. The endpoint could be the front page with the URL of *www.example.com* or a development version of the same page with another URL *www.dev.example.com*. The endpoint will point to a specific service, but the technical aspect allows for load balancing between multiple instances of the same service, enabling horizontal scaling.

The endpoints are formally described as:

$$E = tuple()$$

The different endpoints are the gateway to the services, and contains the information relevant for its configuration. This can include the IP address that is available to the customers, or fully qualified domain name (FQDN). The port number that the service should be exposed on, does also needs to be defined. And last but not least the specific service that it should connect to. From this relation, it is possible to explore a tree of services, that spans out from the endpoint and main service.

Service

In this model a service is a provider for an endpoint. It does not need to be directly connected, but is a part of the needed architecture to enable the serving of a endpoint. This can be split up into **services** and **sub-services**, where the service is the highest level, but depends upon data or logic from

underlying sub-services. Each of these services can be part multiple service trees, that each spans out from an endpoint.

One service could for example be a web server that is connected to a database server. The web server is where the endpoint is pointing, but it requires the data from the sub-service which is the database. In microservice architectures, the database could be another web service, which does some operations on the data, before it is given to the requester service.

The service itself is normally a single or multiple applications, that are essential to provide the desirable functionality. In addition to the applications, there are needs like software dependencies and outer parameters that are needed.

$$S = tuple()$$

The service describes the application needed to provide the service. With this it needs a unique name, and the application that it should run. In an implementation, this could refer to a container image or software package that houses the services application. A service is in the middle of the relations between the different definitions of the terminology in a service environment based on containers. It has a relation to different trees, stacks and containers, and is connected to other services that are providing extended functionality.

Stages

Services will always continuously be improved. This results in constant change of the services. The risk of problems occurring increases with changes done on the environment. However, by introducing elements to test and verify the changes, the risk can be minimized to ensure that systems always are stable and available.

By splitting up the service environment into stages, or parts of the environment, new versions of the services can be tested and verified both before deployment to production and during development. These different stages have names related to their function. Two or three stages are normal. The first stage *production*, contains the services that are in use. *QA* (Quality Assurance) are normally introduced as an implementation of the ITIL process for *Service Testing and Validation* [49, p. 121], where the software can be tested before it is introduced into the production stage. The third stage is *dev or test*. These are used for testing and the development of new and current services. Updates to existing services or new services are first introduced to the *dev* stage, before it is verified in the *QA* stage. After this it can be used in production.

As stages is a normal part of service operations, it has been important to enable the same functionality in this model. However, a stage can here be

seen as a service tree, with its own endpoint. This enables a far greater fine tuning of the different services, but at the same time employing the means of testing the solutions before it is used in a production setting.

Enabled through different trees, and endpoints, the different stages are applied in the form of stacks of containers. The different endpoints and trees can be defined to which container on the stack that it should refer to, and when different versions of containers are applied to the stack, the dev and QA method is ensured.

An environment may have as little as one stage, but also as many as needed. The normal naming are *dev*, *QA* and *production*.

Container

A container is the technical operating-system-level virtualization environment for running isolated services. Each of these containers holds a specific version of a service, and all of the software dependencies that are needed.

This virtualization technique enables rapid spawning of new instances of a service application that has negligible performance impact compared to virtual machines, which enables faster deployments and different distribution strategies. The possibility for fast horizontal scaling, means that the distribution of a service over multiple sites and clouds is a lot easier than with platform aware deployment solutions.

$$c_v = container$$

The attributes of containers are mainly defined as the file system image and the version of an image. Each version of the image would contain a different version of the software. Along with this the container gets different attributes after it is started, which includes its network port mappings.

One of the benefits of containers are the packaging of applications. This is the function that is important for a service environment which is based on containers, as it enables the creation of a container, which always is consistent. Docker is one of the technologies that implement the functionality of containerization.

Service Stack

A service stack is a set of the same type of containers, but are the same or different versions of a software. A service can have multiple service stacks, but the stacks should contain the same amount of containers, and that they are the same version order. This enables a way of duplicating a service, enabling functions like high availability, but at the same time

retaining the version state. Two service stacks of one service, are identical in both number of containers, and their versions, and order.

The use of service stacks are important when defining different service trees. One service tree would refer to a specific container on the stack, effectively making that version of the service the part of that service tree. When defining which version of container is in which tree, different deployment strategies is made possible, where other departments of a business, and not just the operations team would be able to define which versions to run, and also deploy new version when they want to.

An example of a stack that has three different service trees, where each tree relates to a stage of production, quality assurance or development. This would look like the following, where container 0, would be connected to the production, as this is the most versatile version of the service. If however production were to be upgraded, the container in place 1 (QA) would be taken in use.

Version 0: PROD

Version 1: QA

Version 2: DEV

Version 3: next container

The service stack is represented as list, where each place of the list houses one container. These places of the containers are enumerated from 0 to N, and multiple stacks can exist per service.

$$sst_n = [c_0, c_1, c_2, \ldots, c_n]$$

Stack Height

Stack height is a term that is directly related to the service stack. The stack height describes the number of containers in the service stack. Each copy of a service stack, should at all times have the same stack height.

$$SH = |sst_n|$$

Service tree

Throughout the service environment, there are different services that are communicating to enable the usage of data from different systems. This is a type of service architecture (SOA) or microservices.

Figure 4.2 shows a basic service tree, which has three different services, and one endpoint. The endpoint is where the clients or user connects to get the

first service, S. As S needs two sub-services to be able to serve its purpose, it needs to be related to s' and s''.

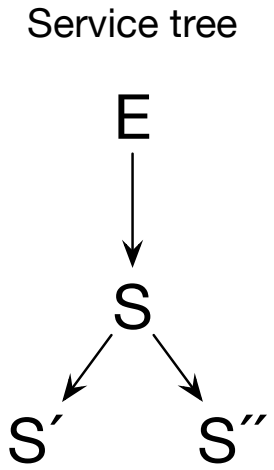

Figure 4.2: A service tree with three services

A service tree will span out from an endpoint. Its nodes will be different services that each are responsible for a part of the main service that the endpoint is related to. What this means is that the service tree shows the logical relation path for each service.

The different endpoints should be defined after the purpose they will serve, and have containers with the versions that are suitable for that purpose. This means that an endpoint that is delivering an important service to customers, all the containers which provides the different sub-services, should be fit for that use. This means that they should first be throughly tested. This can be done with another endpoint and tree, which is meant for test purposes.

In this model, the way the trees are choosing which container to communicate with through the service tree, is by pointing the tree at a specific point in the service stack.

Figure 4.2 showed a service tree with three different services that are providing service for a endpoint. In a real use case, this could be a public web page for a company. Figure 4.3 shows the exact same tree, but with the underlying technologies exposed. The service tree itself only exposes the service, but the service could be multiple web servers, a database or event content servers. Load balancers are an important part of the infrastructure, but is not a service in itself.

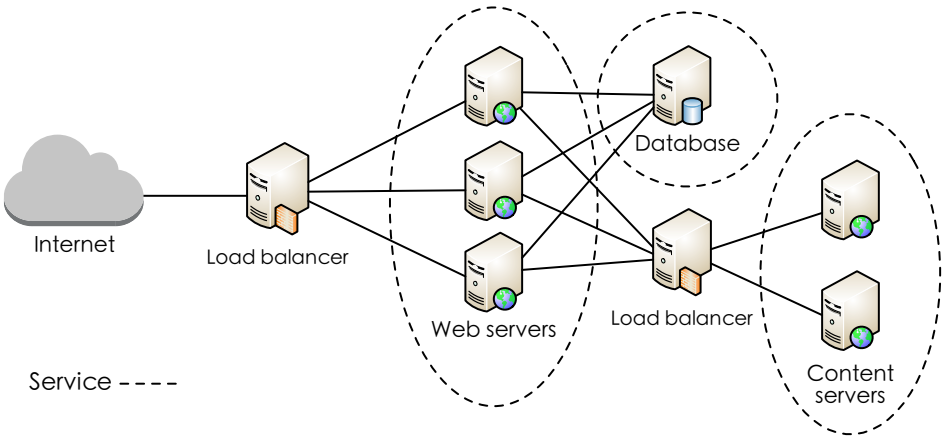

Figure 4.3: Example of a service tree implemented

If the example is explored further, multiple service trees can be shown as the implementation in figure 4.4. This shows the exact same service, but with them running in different containers. This corresponds to three different endpoints, which refers to different containers. Each of these endpoints then spans out into the trees that includes every of the different components. This greatly increases the complexity of the environment, as the number of items and components increases.

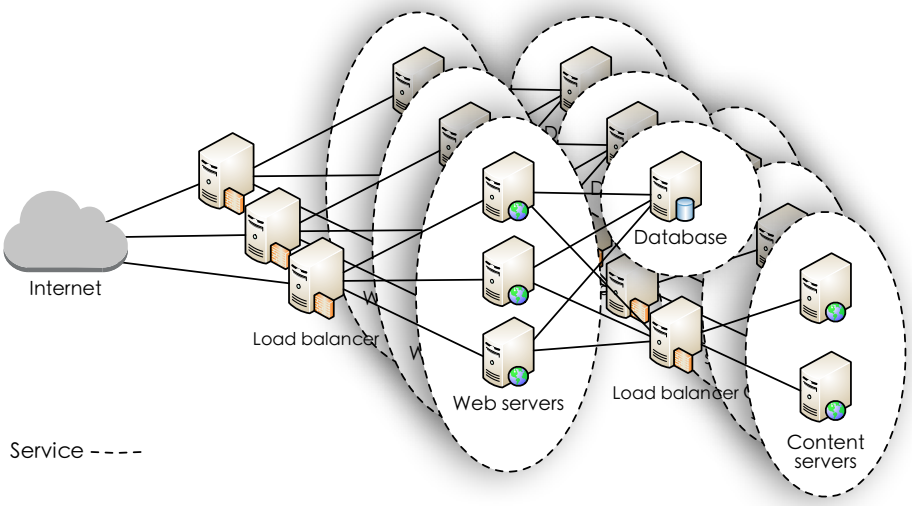

Figure 4.4: Example of multiple service trees implemented

A service tree can be defined by an adjacency matrix, that defines which services have a logical connection to each other. Figure 4.2 would relate to the following matrix, that shows the relation between the endpoint, the main service, and the two sub services s' and s". 1 means that there is a connection and 0 means that there is no connection, and relates

to true/false. An implementation of this would however use a more human readable relation, but a matrix is a good and specific design definition.

$$T = \begin{array}{c} \\ E \\ s \\ s' \\ s'' \end{array} \begin{array}{c} \begin{array}{cccc} E & s & s' & s'' \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \end{array}$$

Service Tree Count

The tree count is directly related to the different service trees that are in an environment. As a service tree consists of the path from one endpoint, there is a need for multiple service trees for each of the different uses, like development and production. This means that a single service can have multiple service trees that are connected to individual containers of the service. The tree count describes the number of trees that are connected to the service.

With the number of trees that are connected to the service, the importance of the service can be derived, and functionality like capacity planing and high availability can be implemented. This could be done through a constraint that defines that the needed amount of containers should equal the number of service trees, or different stages.

The relation between the service trees for a service and the tree count can be shown as the following.

$$TC = |\{T_s \cdots\}|$$

4.3 State of the service environment

The core of this model is the state. The state describes the current condition of the environment at all times. This is an important aspect to be able to handle continuous changes, and ensure that every action that is taken is in compliance with the constraints that are configured.

The state looks at different items with attributes to ensure that the condition and correlation of the different items are correct in accordance with the definitions and constraints.

The following items are needed in the state of the model, as they are central components of the model.

- Endpoint
- Service
- Service Stack

- Service tree
- Containers

4.3.1 Defining state

A service environment contains a lot of different items that are important for the daily operation, from the infrastructure services to the user services. As this model is working towards defining the given operational aspect on the running services that provides value to the business, it is important to have a definition of the state of the environment.

The state can be defined as ***the environments items*** (as listed in 4.3), ***its relations and attributes***.

This can also be formulated in a formal way where the items *service tree (ST)*, *endpoint(E)*, *services (S)*, and *service stack* (SS) are represented as different sets. Each of the items contains the relations and attributes, which represents the state of the environment.

$$state = (STS, ES, SS, SSS)$$

This relationship between the different items can be shown as an ER model as shown in figure 4.5. A short explanation of the figure is in order. There should exist one *endpoint* for every stage, and that is connected to a service. The relation between the services is defined by one *service tree* for every stage. For every service there can be one or more *service stacks* which contains one or more *containers*.

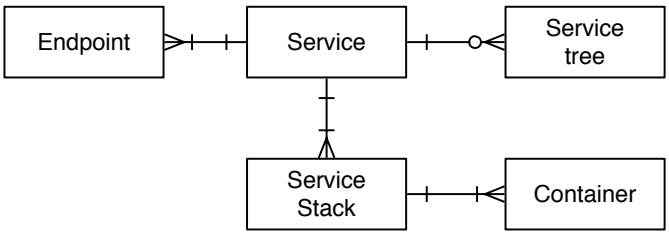

Figure 4.5: Entity relation of items

It is important to ensure that the state can be tracked and ensured to be correct, as to implement the model in a way that both enables proactive, and reactive compliance measures. This enables detect of situations that are not in compliance with the constraints of the model.

As previously outlined, the different processes have accompanying actions that are used to do changes to the state and the environment. The following formulas describe how different actions are related to the state. To ensure that the actions taken will be in compliance with the constraints that are

defined, the state of the environment needs to be taken into account. This results in two different possibilities.

$$ST = \begin{cases} items \\ relations \\ attributes \end{cases}$$

The first possibility is that when a action is performed, the state is passed as an argument to the function. The function can therefore verify if the state still will be in compliance after the action should be run. This will include the checking of constraints of state, and the action. If the option to only verify the state, it will not do any actions on the state, but would check if the action would cause compliance issues. This can be defined as a *immutable* action.

$$action(ST) \rightarrow ST$$

However, when a action is performed on the environment, it still takes the state as the parameter, but it performs the operation, and the state is updated. The following is a *mutable* action that performs the action on the state.

$$action(ST) = ST'$$

This structure enables verification of state, as well as continuously updated state. With the first action, that only tries to verify the state after an operation, would create the possibility for a dry-run, which means that the action could be tested before it is run. It is important to remember, that in both cases, ST and ST' will represent a correct state.

4.3.2 Example environment

To better understand what an environment and what state actually covers, a explanation of a sample environment is in order.

Figure 4.6 shows a large environment. It contains four different main services (which are marked as S_w services), that each got two connected endpoints (E_{QA} and E_{PROD}). The endpoints are in different stages, meaning that a service has an endpoint that is in the quality assurance stage, and the other is the production stage. Each of the main services (which in real use cases would be a presentation layer or the first part of a presentation layer) are connected to sub-services, which can provide more functionality. If this is also related to a real use case, the sub-services could be part of a business logic layer. In the case with microservices, each of the sub-services would each provide a part of the complete main-service.

In this environment, the leftmost service (S_m), could be a SOA implementation of a Service Bus, which has the key feature of translating different API/soap calls to different *backend* services. These are key services, that if affected by outages, would result in loss of service to some extent in all of the above services. This is important to be aware of, as it enables the system administrators to take proactive measures to ensure the stability of the service.

The figure (4.6) can be seen as multiple trees (as shown in figure 4.2), where each tree spans out from an endpoint. Each of the elements (or subservices) of the different trees corresponds to a different service stacks, with corresponding containers. The dotted line on the left side of the illustration, shows *one* of the services trees.

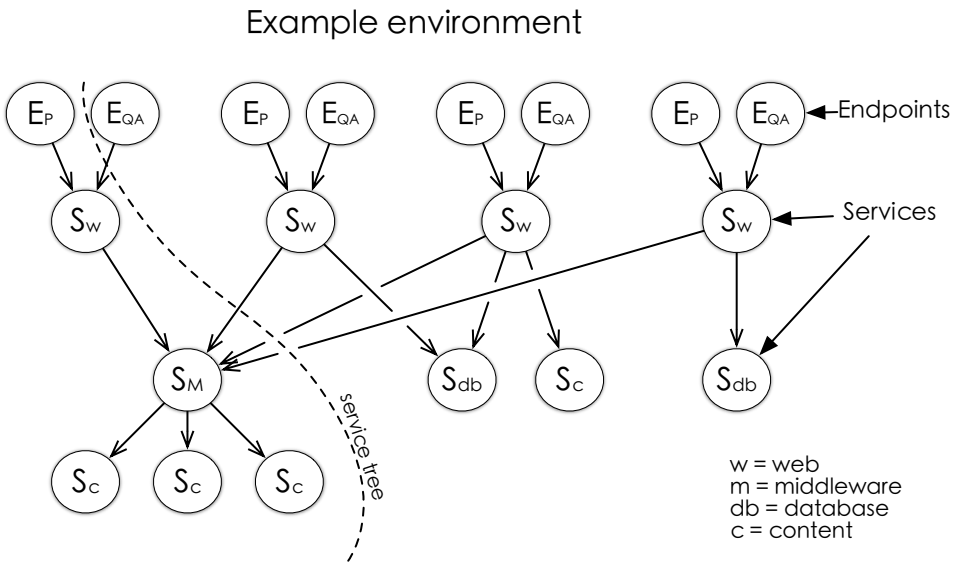

Figure 4.6: A large environment with a lot of different services which has endpoints of different stages.

However, there is need for more than just the services that provides the end users with the needed functionality. As shown in figure 4.7 there are many different infrastructure services and processes that needs to be available to be able to handle different services in the service environment. Many of these are out of scope for this thesis, but different aspects of these services do need to be taken into consideration, to ensure that they are implementable in the model.

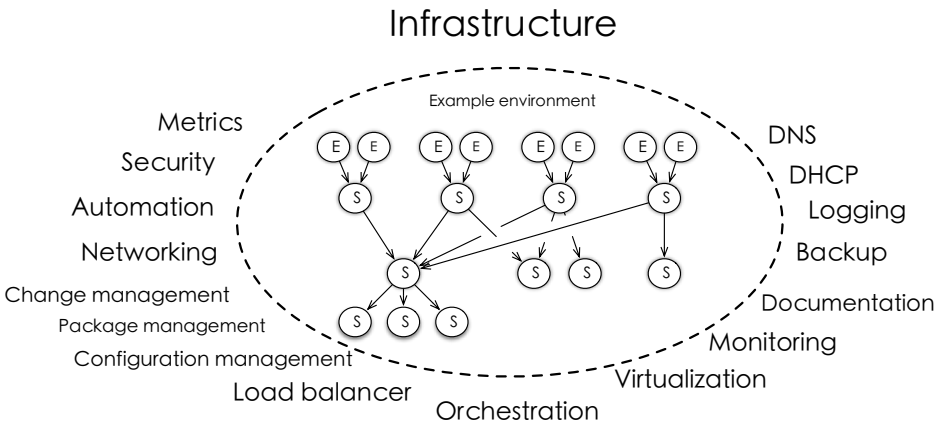

Figure 4.7: Environment shown with items that are related to the infrastructure

As shown in the large service environment illustration (4.6), there could be many different services and many different endpoints. These are however, just a higher level of abstraction of the reality. Figure 4.8 shows the relation between the endpoint and the main service, and how this corresponds to the underlying containers. In this example, the endpoint corresponds to one service. This service, is built up with different containers, that are different versions of the software of the service. The number of running containers results in the current *stack height* of the service. The current endpoint shown in the figure, only corresponds to one container in the single stack. If the endpoints stage were production, the container corresponding to the production container in the service, would be the one handling the user requests. If the service needs more resources or high availability, another stack would be created.

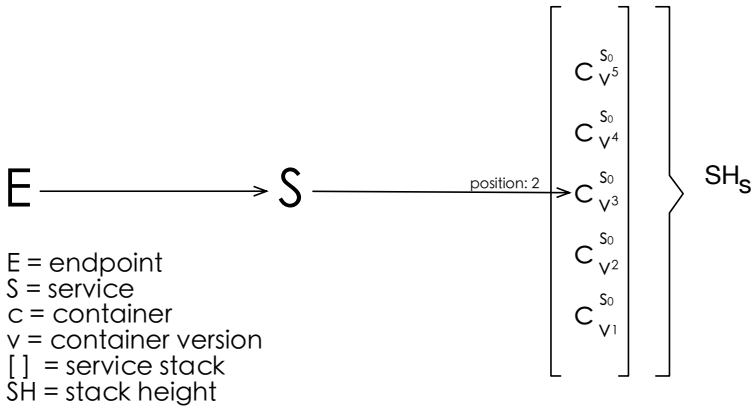

Figure 4.8: The relationship between the endpoint, service and the services stack of containers

These figures are good for understanding the underlying technology and terminology, and enables the definition of the state of the environment. As

the state is dependent upon the running items, it is important to know their relation to further understand their importance.

The service shown in figure 4.8 is also dependent upon sub-services, if seen in combination with the complete illustration of the service environment (figure 4.6). As there are many layers of abstraction down towards the real setup, what is seen in the overall view, is hiding the real structure of the services. Figure 4.9 shows how the service tree spans from the endpoint to the main service, and are connected with the two sub-services. The service s_1 has one service stack, with its corresponding value SH_{s1}, while service s_2 has two service stacks. For s_2 the two service stacks are the same, and SH_{s2} is therefore equal for both of the stacks. Each of these stacks, contains the containers for the specific service, the same way the main service has different containers in the stack.

This figure does show the relation between the different items, and this is important to enable a correct relationship-mapping in the state of the environment.

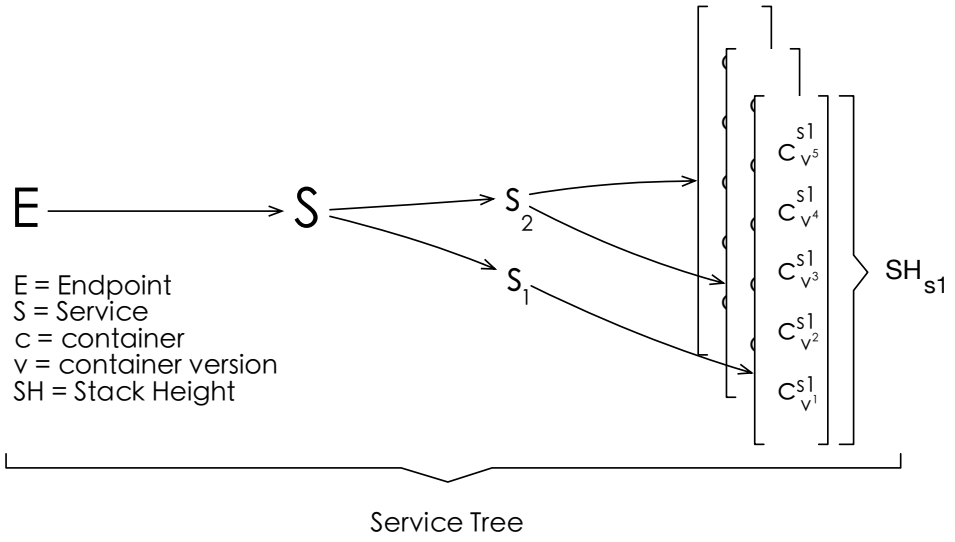

Figure 4.9: A complete service tree from endpoint to sub-services, with multiple container stacks, and their relation.

Figure 4.10 shows the same tree as figure 4.9, with the containers of S defined. When a endpoint refers to a service, what it really is pointing at, is either a stack position, or a specific container. If E is a production endpoint, the container of service S, would be the container that is ready for production. This should be the container at the lowest position of the stack, and in this example point to container C_{v^1}. The tree can be defined as to use the services and their stack 0 position, and this would make the containers of the sub-services ($c^{s1}_{v^1}$ and $c^{s2}_{v^1}$) relate to the production endpoint E.

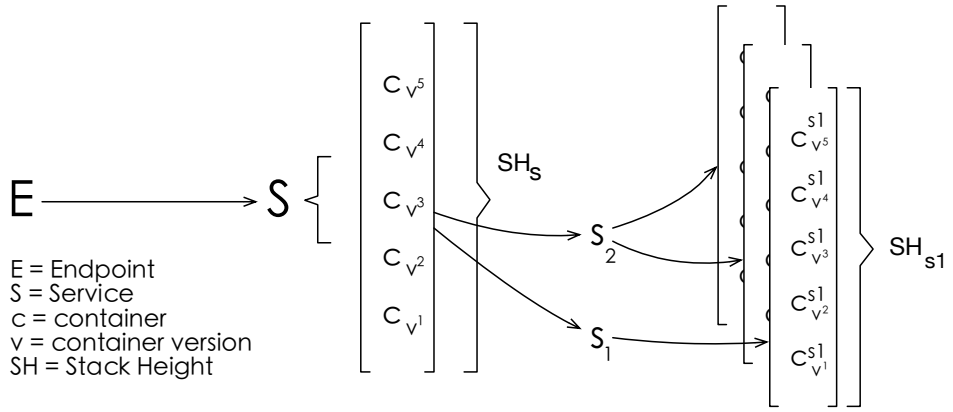

Figure 4.10: Detailed look on the service tree

Combined, the state of an environment can be shown in another manner. Figure 4.11 is a cut-out of a larger service environment, but where the stacks have been labeled after their relation to a given stage. These labels are what relates to the different service trees, and ensures that the different services are connecting to the right places.

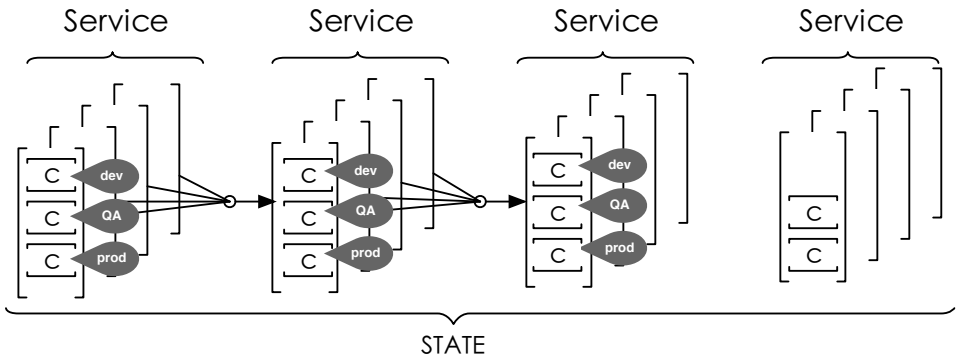

Figure 4.11: State of an small service environment

4.3.3 Constraints of the state

To ensure that the state of the environment is continuously correct in regards to what would be a good configuration of the environment, there is need for constraints that ensure that actions taken, do not create situations where loss of service, or unfortunate situations could happen.

This section outlines the basic constraints that are needed in conjunction with the definition of state. This creates a baseline for the standard environment of services, by defining the necessary parts that needs to be configured in order to provide a service.

Endpoints needs to be connected to a service. This is an item that is used for describing the services which are available to the end users. All services without an endpoint is a sub-service, and only available through other services. An endpoint should also refer to a specific stage of the service, which means that an endpoint is specific for the stage, and there is a possibility for multiple endpoints per service. The representation connects the endpoint to the service, but in reality the endpoint has a direct link to the containers, which runs the service and stage in question.

$$E^* \rightarrow S_{stage}$$

Services are the abstraction and relation to the running stacks, containers and service trees. Most of the constraints on the services are employed by the underlying items. However, it relates to the services stacks, where a service can have one or more service stacks, but must at least have one service stack. If there are multiple stacks, they should contain the same number of containers with the same versioning.

$$S = \{sst_n, \dots\}$$

Service stacks need to be connected to a service. A service stack can only be connected to a single service, as it contains containers for only one service.

$$sst_n \rightarrow S$$

There are two other main constraints that apply to the service stacks, which are important for the implementation of the model, namely container versioning and the number of containers in a stack (called stack height (SH)).

The following equation shows a stack with an arbitrarily amount of containers. Every container has two attributes which are the stack counter and the version number of the container. The container which was first added, and has the lowest version number gets the stack counter of 0, which is a reference to the place on the stack. The version number represent which version of a container is being used.

$$sst_n = [c_0^{v01}, c_1^{v02}, c_2^{v03}, \dots, c_n^{v_n}]$$

The number of containers in the stack (named stack counter) should also be more than 0. This is not always possible when establishing new services, which means that the stack should be allowed to contain 0 or more containers.

$$SC_s >= 0$$

Consistency in versioning of container in the stack
This results in a constraint, that is important for ensuring consistent deployment of new services. A stack should only contain containers with same or increasing versioning upwards in the stack. The lowest version of the service should be in position 0 on the stack, whilst the next container on the stack has either the same or a higher version number. In the following formula, *sst* is the Service Stack, v is the version of container c_i, and v' is the version of the container above in the stack (c_{i+1}). n is the number of containers, or stack height.

$$For\ all\ c_i^v\ and\ c_{i+1}^{v'}\ in\ sst:\ v \leq v'\ where\ i+1 \leq n$$

Number of containers
The number of containers in a stack has a minimum amount based on the number of stages and trees that the service should support. Standard configuration should result in three different stages; the *dev*, *QA* and *prod* stages.

$$SC_s >= stages \wedge ST$$

Service Trees An important feature of the model and the importance of state is to contain the information and relation of the services. This is the feature that belongs to the service trees. As the service trees contains the information about the relation from endpoint, to service and to sub-services, they are important to ensure that the services are correctly defined.

It is therefor important to ensure that each service has a tree count of more than 0, where the tree count is the amount of trees for a single service.

$$TC > 0$$

The number of trees related to a service is also dependent upon the number of stages. The number of stages is here important as there should exist a tree for each different stage, which relates to a specific placement in the service stack. This gives us the following:

$$TC = Number\ of\ trees >= Number\ of\ needed\ stages$$

Containers Containers have few constraints connected by default. The constraints that apply is that of the containers relation to the other item. The containers itself is more related to the practical implementation of the deployment of the service application. The key value that all containers

must have, is a version tag, which identifies the specific version of software and container.

$$c \Leftarrow version$$

These constraints are important to ensure that the overall state is consistent, when working with the different items. It is also equally important to combine the constraints so that the state is consistent at all levels. This gives us the following equation which is a representation of the relation between service stacks, services, trees and their count.

$$For\ all\ SH_S\ in\ S\ where\ S\ \in\ T:\ SH_S\ \geq\ TH$$

Summary of constraints

The following table contains a summary of the constraints in a short form. This enables easier reference.

Nr	Short name	Formula	Description
1	Endpoint	$E* \rightarrow S_{stage}$	Connected to a service at a stage
2	Service	$S = \{sst_n, \dots\}$	A service has one or more stacks
3	Service stacks	$sst_n \rightarrow S$	Needs to be connected to a service
4	Stack versioning	$sst_n = [c_0^{v01}, \dots, c_n^{v_n}]$	Increasing or equal versioning (See 4.3.3)
5	Stack height	$SC_s >= 0$	Number of containers on stack (See 4.3.3)
6	Service tree	$TC = \|trees\| >= \|stages\|$	Number of trees (See4.3.3)
7	Container	$c \leftarrow version$	Needs to have a version

Table 4.1: Table with constraints of state

4.4 Processes

The model employs different processes to enable different operations on the environment. These processes work to change the state in different ways, where the goal is to provide new value to customers. How they affect the state depends on the operations that has been initiated, and through the actions that are needed to complete the operation.

A single process can achieve different results, which will affect the state of the environment in different ways. However, when changing the environment, there are three different algorithms that can be used.

0. Best effort
1. Continuous change with action rollback
2. Commit based at end

The best possible outcome is to have a complete state at all times, and have the proactive measures of ensuring the consistency. Algorithm one and two will achieve these proactive measures, while algorithm zero may result in a half completed state, where repercussions of inconsistent state may apply.

Each of the following processes contains a set of actions, which are performed in order to achieve a result which completes the assigned task of the process. Different actions apply to the different processes, which will result in some part of the state being changed. The following processes will be described in more detail.

Process 1: State of the environment and services

Process 2: New service to be deployed

Process 3: New version of a service to be deployed

Process 4: Management of a running service

4.4.1 Process 1: State of the environment and services

As an environment increases in size, where new services are implemented and existing services are expanded, it is easy to loose track of what is going on. Different parts of an organization is also in need of an understanding of how an environment is structured, from decision makers, application developers to IT operations. By enabling a way to give understanding of the current state to all the stakeholders, new operations and choices can be based on fact, and not only a lesser perceived state.

This process employs different actions that should enable viewing of the state in different ways. The intention of this is the need to be able to get the current and accurate information. This is normally performed with a configuration management data base (CMDB) which keeps track of the business key items. This process does the same, but with a state, that should always be a correct state.

The following illustration (4.12) shows how the process of showing state. This relies on the gathering of key data as items, its attributes and the relations between the items. The input is illustrated as a construct of a data structure, inspired by python. When state is known, the only needed input is *what* to view, and possible constraints of what is shown. This enables easy access to the different elements that the state is built upon.

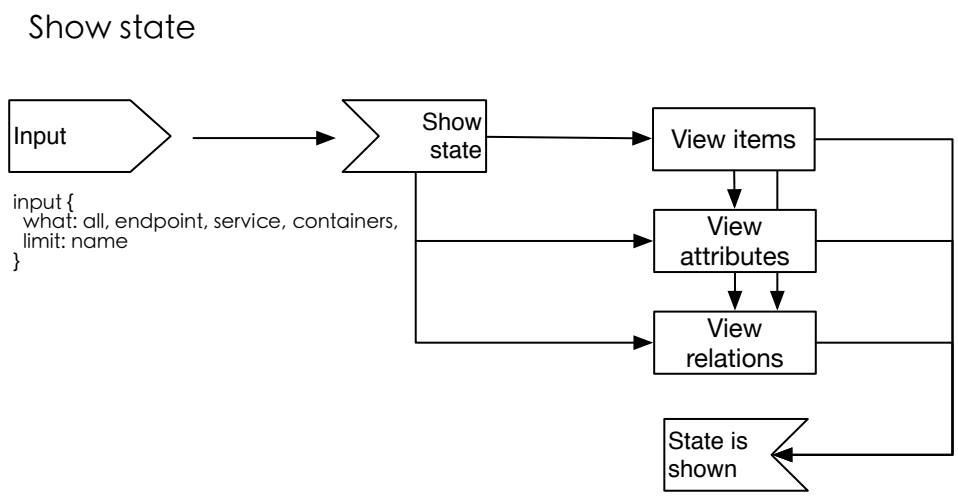

Figure 4.12: Process view of state description

The different actions in the case of this model is based on the expected outcomes. This can be illustrated through pseudo code, as functions.

The most verbose action is to show all of the items with attributes and relations. This will give an output where all the content is presented. No constraints apply, as nothing is changed but only displayed.

```
view_state(state)
>> {(services, endpoints, stacks)}
```

Each of the other items can likewise be shown with actions, that with the state, enables the presentation of the different items based on their attributes, like their name. The actions return the state, that is here represented as a data structure inspired by python dict and json. But these actions would likely been in use to print out information to a terminal, or to a web interface.

```
view_service(state, service)
>> {name, parents, childs, stacks, endpoints}

view_stack(state, stack)
>> {name, host, image, service, containers}

view_endpoint(state, endpoint)
>> {name, url, ip, port, stackpointer, stage, service}
```

The input of the different actions shows what is needed by the user to enable the process and actions to gather the necessary data, and present them. This process is the most basic, which uses the information which can be gathered through the automation done in the following processes.

4.4.2 Process 2: New service to be deployed

With new business objectives, changed market values, or need for more effective business processes, new applications are often born to enable the solution of these objectives. When time-to-market and faster release cycles becomes important, the need to get the application into production and as soon as possible, can result in bad products that are released to early.

This model enables faster releases of services, at a standardized way, which reduces the risk that often are accompanied with large and manual processes.

The following figure (4.13) shows the different actions needed for a process that enables the release of a new service. This is based on the minimal input, where the different changes to the environment is built into the actions of the process. The figure also shows the complete process from the input of a interface, through each of the different actions, that enables the different operations, to the service is fully created.

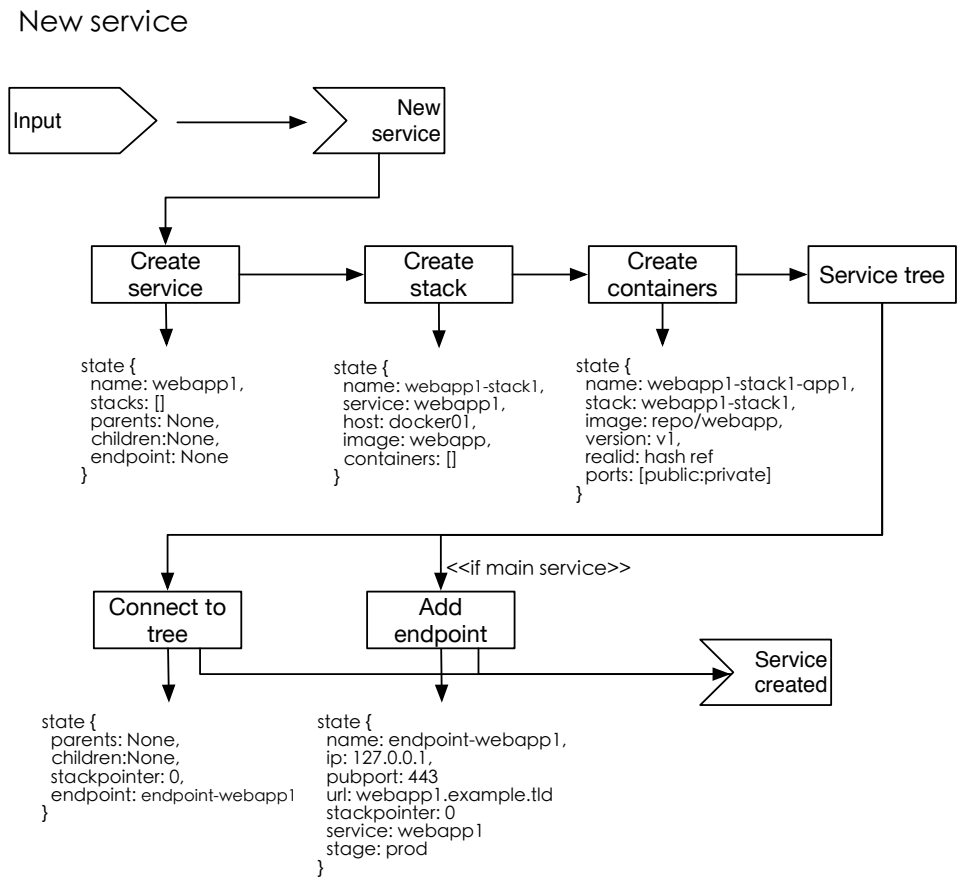

Figure 4.13: Process view of the deployment of a new service

When a new service is to be deployed, there is a lot of different actions that are needed to be able to get the application running in production. In a

manual environment, this would require the installation of a new virtual machine, configuring of DNS, load balancers and maybe registration of IP addresses, and of course the installation of the application. However, with containers, this process is simplified as the operating system and the installation of application is combined.

Each of the actions illustrated, handles different parts of the information, and based on this and the state, ensures that the services that is created is in compliance with the constraints on the environment. An action can in each case fail, but the state will always be correct, as the process enables the structure for the calling of actions. The state is not necessarily *complete*, but correct. For this reason, the actions that are connecting the services to other services and endpoints, are the last to be initiated.

Create service is the overlying action that provides the creation of the service (one of the illustrated dots in figure 4.6). The action in itself only requires one input parameter, the name, to enable the creation of the service in the state. This enables the setup of other needed parameters, like the stacks connected. When the service is first created, the amount of stacks is none, which is an allowed state.

```
create_service(state, name)
>> {name, stacks, parents, children, endpoints}
```

After the service has been created, the **create stack** action can be initiated. This action can be run multiple time for each service, to create multiple service stacks. At the time of their creation, they are empty of containers, but the next action enables the pushing of containers onto the stack. The create_stack function also needs some information, where the service that it is related to is defined, along with the name of the image that the containers consists of, and the host that the container should run on. The host is defined in the stack, so that the stack of containers consistently are present on the same host. This is a design choice, that ensures that the containers of the stack is placed on the same host. This makes it easier to ensure that two stacks, that contains the same versions and number of containers are not placed on the same host.

```
create_stack(state, service, image, host)
>> {name, service, host, image, containers}
```

When the service stacks are created, it is possible to **create containers** that are *pushed on the stacks*. The action gathers the needed information about the container that are not stored in the stack.

```
push_on_stack(state, stack, version)
>> {name, stack, image, version, realid, ports}
```

If the service that is deployed is a main service, meaning one which consumers would connect to, an action to **add an endpoint** is needed. This creates a new endpoint, which exemplified could be used in the configuration of load balancers. This is connected to the service that was

created. The name could be used as a reference to the stage the endpoint is the provider for.

```
make_endpoint(state, name, service, publicport)
>> {name, ip: None, pubport, url: None, stackpointer: 0, service}
```

If the service is not a main service, but a sub-service, the service needs to be **connect to a service tree**. This is done through the action create_tree, which connects the different services in a relations mapping. This connects the services together based on the information from the endpoint and the relations provided. Optionally the service tree can contain information about which stack position the tree points to, that extends the stackpointer contained in the endpoint.

```
create_tree(state, endpoint, relations)
>> updates the service tree related to endpoint
```

These are low level actions that enables the process to be completed. The implementation of the process itself, handles that the different constraints of the state are handled.

4.4.3 Process 3: New version of a service to be deployed

New versions of services are continuously developed, as to improve and maintain the current functionality of the service. This means that new versions of the services needs to be deployed. There has been many different ways of handling the process of delivering services, where many of them includes manual elements.

This process handles the actions of deploying new versions of services. This is technically done through the building of a new container, which is tagged with a specific version. This can then be referenced to, which enables the deployment of the new version.

Though the creation of the container and the reference tagging is easy, how to change the version of a running software is a bit more complex. There are multiple strategies that can be used to release the new version, but all of them have in common that the new version should go through the normal release cycles that employs a stage based testing regime.

Figure 4.14 shows the process of releasing a new version of a service. Through the actions, a new container is created and pushed on each of the stacks for the current service. Though pushed on the stack, the new container is not yet part of a service tree. To enable the new version, this can be done in two different ways.

- Branching the service tree
- Pop existing containers

Both of these are fully supported of this model. Branching of the service tree, means that the pointer of the tree that points to a specific place on

the stack, is switched to another place on the stack. This is a tricky to implement, and could be more error prone than the preferred method of pop'ing containers. This ensures that the service versioning is consistent, and that each service is tested before it is in production.

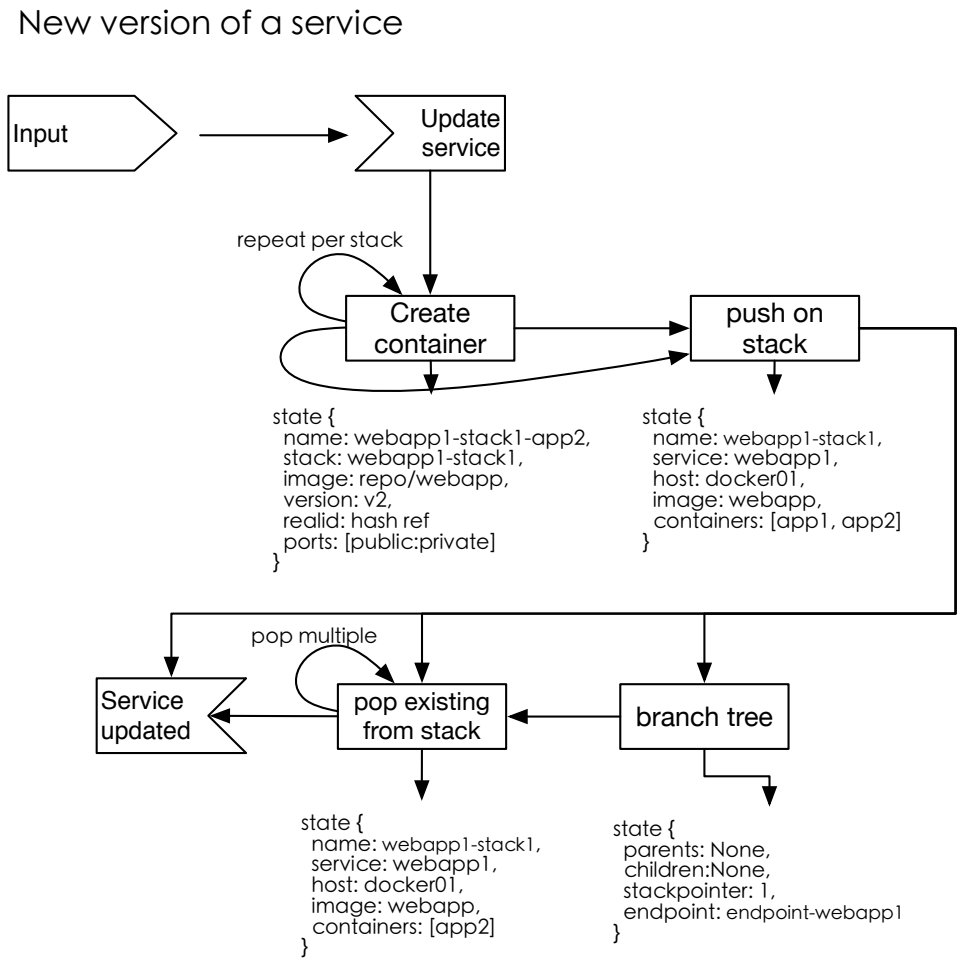

Figure 4.14: Process illustration of a new version of a single service

The action that enables new containers on the stack, is the action *push_on_stack*. This ensures that a new container is added to the stack, with a new version that is specified. If multiple stacks exists for a service, the action need to be performed multiple times.

```
push_on_stack(state, stackname, version)
```

The other action which actually ensures that a new version takes over, is the pop action. What this does is to remove a container from a specific stack, enabling the container in the next position to take its place. The input of this action is which stack, and the position, which is based on the stack counter.

```
pop(state, stack, pos=0)
```

4.4.4 Process 4: Management of a running service

The last process is intended as a management process that enables operations on the services after they are deployed, but are management operations that are not about the release of new versions. The process does not include all the actions that are needed in a normal environment, as this depends upon the needed functionality for the business.

Replace is a function that enables the replacement of containers on a stack, that are not in compliance with the intended quality of a release. When such a container is on the stack, an action is needed to enable the removal of the version, but at the same time be in compliance with the constraints of the state. The constraint of the environment regarding versioning states that a stack can only contain versions that are higher or equal to the container below. This enables the replace action to either replace the current container with the version of the container above or below, and still be in compliance. This is shown in figure 4.15, which illustrates the replacement of the middle container, which then either can get version 1 or 3.

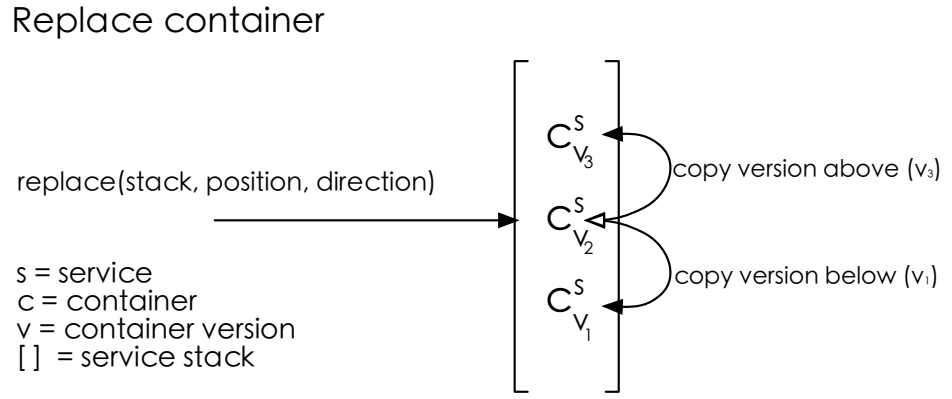

Figure 4.15: Replacement of container on stack

A programmatic representation of the action would be the following, where the position is spanning from 0 an up, where v_1 in the figure is at position 0.

```
replace(state, stack, position, direction)
```

The process also allows for other actions, where management of the existing solutions is needed, but also the management of the entire state. This could be the scaling of a service, where new stacks should be added or removed, or the changing of a service tree. Exportation and importation of state is also a possibility, as the properties of the items are known and recorded.

An action for changing the service tree, would allow for changing the version that the tree is using, without changing the stack. This would

be an action that changes the state by editing the conditions of a load balancer.

```
changetree(endpoint, fromservice, toservice, position)
```

4.5 Summary

In this chapter, the design of the model has been presented. The model has described the needed terminology, which are present throughout the model and helps create a consistent model. The terminology have defined key terms, definitions, but also items which has helped to define the other parts of the model.

State of the environment have been exemplified and defined. Different constraints have been found which are related to the continuous state of the environment, and helps with the sanity and consistency of a service environment which is based on this model.

Four different processes are defined, but the model is not limited to the these defined processes. They provide basic functionality through defined actions, that enable the consistency in state, and also the ability to perform actions on the state and environment.

The model has been designed, but as the results are near impossible to correctly verify, there is a need for a way to test the model by an implementation. The model is closely connected to the implementation, due to the generality of the model. To look closer at the benefits of reduced complexity in environment operations, the model needs to be illustrated through an implementation. The next chapter will focus on the implementation of some of the processes defined in this chapter, but with the basis of providing a state that is current and correct.

Chapter 5

Implementation

This chapter describes an implementation of the model designed in the previous chapter. Through the different life cycle events, state and processes that the model enables, an implementation is in order to enable the measuring of surrogate variables, that may indicate if the model is contributing to the desired goal of reducing complexity for the administrators and other stakeholders.

The structure of this chapter is built up around the deployment of one new service, where the environment is first configured for the prototype implementation that has been done, and the different decisions that have been made in order to implement the model. First, an introduction to the implementation of the model is in order.

5.1 The implementation and the decisions made

To be able to implement the model defined in the previous chapter, there are several aspects which needs considering. How can the state be implemented in a way that enables all the features of the model, but at the same time be of a good design? How can the different actions be implemented as to achieve the desired processes of the model?

5.1.1 Handling the state

As the state has been defined to a fine level in the design of the model, it is possible to use the same modelling to ensure that the state in the prototype will be a good implementation. There are two different ways that have been considered. *Structured* or *unstructured* storing of state?

Structured storing of state relates in this case to a relational database. Each of the different items defined in the model can be structured as a table, and the relations are a good representation of their correlation.

Unstructured data on the other hand is easier to implement than that of a relational database, which needs to be modelled and created. Unstructured data can essentially be implemented as a dictionary or list of dictionary or json. This can easily become complicated to handle and is error prone.

In the prototype this has been considered, and the choice to use a relational database was made. With the use of a correct relational database, many of the constraints that are presented in relation to the state can be enforced through the possibilities of the database. More importantly, the basic framework of the prototype enables the data to be used at multiple places at once, enabling both a command line too, and a web interface. 4.5

5.1.2 Docker

In this implementation, the choice of provider for container solutions is Docker. The Docker project, is the containerization project that has come the furthest. It has huge backing, and is probably one of the larger open source projects at the moment. Along with the huge backing from huge companies, like Red Hat, it is acquiring new firms with new technology. However, the technology and functionality of Docker is shifting fast. This implementation, will therefor focus as little on the functionality of Docker as to ensure its compatibility with other solutions, but use the capabilities of containerization and management of dependencies.

5.1.3 Python and libraries

The prototype is written in Python, which allows for fast coding with rich features. There is also a lot of additional libraries that are freely available, such as a implementation of the Docker API [50], which has been implemented in the prototype (A.13, on page 119).

This library enables the communication with the different Docker engines, and the features which are available through the Docker API. Included in this is the needed functionality of creation, listing, starting and deleting containers, enabling the basic management features needed.

As the state is handled as structured data, this needs to be handled specifically in Python. This is done through a python library called *SQLAlchemy* which is a Object Relational Mapper (ORM), that can map Python objects to tables in a relational database. With objectification of the items in Python, additional attributes are easily added, with the database abstracted behind code. One of the powerful features with the use of the ORM, is the portability that it enables. The same code allows the use of different database engines, such as SQLite, PostgreSQL, MySQL and Oracle, without any modification.

One important feature that the ORM adds, is the session handling. This enables the possibility for handling rollbacks. This is done through the usage of a *commits* and *rollbacks*. If an error or unwanted state is detected, the session can be rolled back before it is committed to the database, ensuring that the data in the database always is correct.

5.1.4 Structure of the prototype

The structure of the prototype is designed to be both scalable and maintainable. This is done through its separation into different files, practically making each file a model in Python. This ensures that vital functionality can be used in the different parts, without the need for replicated code.

The different items defined in the model are implemented as Python objects, enabling the usage of the ORM, and relates the different objects to a table in the database. Each of the instantiated objects then relates to a row in the database.

In the model, a diagram of the relation between the different items of the state is defined (figure 4.5 on page 43). This has been implemented in the prototype directly, without any modifications. At this stage, it is possible to see that the model can be directly implemented.

The implementation has a total of 5 different Python classes for the items in the model:

- Service
- Service_tree
- Stack
- Endpoint
- Container

In the following code, the class declaration for the Service object is shown. Here the different attributes and relations to other objects are defined in a way that the ORM is able to understand. This is done with extending the python class (on line 1) with the SQLAlchemy object Base, and by defining each of the different attributes with either the keywords *Column* or *relationship*. *Column* refers to a column in the database table, while *relationship* will add a reference column with the id of the foreign table row.

Listing 5.1: Service object: How a database object is made

```
1
2  class Service(Base):
3      __tablename__ = 'service'
4
5      id = Column(Integer, primary_key=True)
6      name = Column(String, nullable=False, unique=True)
7      parents = relationship('Service_tree', backref='child',
8                             primaryjoin=id == Service_tree.child_id)
9      childs = relationship('Service_tree', backref='parent',
10                           primaryjoin=id == Service_tree.parent_id)
11     stacks = relationship('Stack', cascade="delete",
12                           backref=backref('service', order_by=id))
```

As the different items of the model is defined in this way, the ORM is able to create the database from scratch, without any SQL actually being written.

Program structure

The different objects that relates to the items of the model is split into different files. This makes it easier to handle the different objects, but also increases the complexity of the implementation in itself. The way the prototype has been written has resulted in a model-view-controller implementation, which separates the data from the logic and the presentation. This is a commonly used way to implement larger web pages, and can be seen as a best practice approach.

The different part of the prototype can be related to this. The directory listing below, shows the different files the program is built up of. The main program **manage.py**, a command line tool, is the base of it all, and relates to the view part of the MVC architectural pattern. This uses the functionality of basefunc.py. It contains the basic functionality that is outlined through the actions in the model, and the combination of the actions.

The lib folder contains all of the other code, that is either related to the objects and database implementation, or the integration to external sources. The lib_docker.py module is one of these, and integrates the functionality of the Docker written Python library docker-py.

```
cmanage
├── manage.py ..... (Listing A.4 on page 98)
├── basefunc.py ..... (Listing A.5 on page 102)
├── lib
│   ├── __init__.py
│   ├── init.py ..... (Listing A.6 on page 111)
│   ├── base.py ..... (Listing A.7 on page 112)
│   ├── service.py ..... (Listing A.8 on page 112)
│   ├── stack.py ..... (Listing A.9 on page 114)
│   ├── container.py ..... (Listing A.10 on page 115)
│   ├── endpoint.py ..... (Listing A.11 on page 117)
│   ├── config.py ..... (Listing A.12 on page 118)
│   ├── lib_docker.py ..... (Listing A.13 on page 119)
│   └── haproxy/ ..... Se section A.3 on page 124
├── hap.py ..... (Listing A.14 on page 121)
└── etc
    ├── config.conf
    ├── docker.conf
    ├── logging.conf
    └── rules.conf
```

5.1.5 Getting started with the prototype

Before the prototype can be used, some first time configuration and initiation is needed. Under the directory *etc*, different configuration files for different parts of the prototype is included.

The main configuration file *config.conf* contains the essential parts such as database connection information. The database connection information is used to connect and initiate the database on which the system depends.

There are two other important configuration files, the *docker.conf* and *rules.conf*, where they respectively handles information about Docker that is normally changing, and the rules which should be easily configurable.

The docker.conf file contains the connection information about the hosts that it can deploy new containers on, while the rules.conf contains default values for the different constraints.

When the configuration is performed, the new system can be initiated, and is ready for use with the following command, which creates the database (if SQLite) and sets up the different database tables and relations.

```
$ python manage.py init
```

5.2 Getting ready for deployment

A new application has been developed and is almost ready to be deployed as a new service, but there is a few steps that needs to be made to be able to deploy the new service with Docker.

First the code needs to be marked with the current version that the code represents. This can be done by tagging the committed version of the code as a specific version. Tagging the latest commit is done with the following commands, making the latest commit version 0.1.

```
$ git tag -a v0.1 -m "This is version 0.1"
$ git push origin v0.1
```

This enables the building of a Docker container that has a specific version of the code. The building of the container can be done in two ways. Either it can be done locally, through the command line on a machine that has Docker installed, or through the service called Docker Hub. The Hub supports direct connection to GitHub and Bitbucket, where you can pull the different tagged versions of the code, and have the site build the containers for you.

Figure 5.1 shows the process from the tagging of the new version, to the new version is added on the Docker Hub, before it can be pulled and run on a Docker host as a new container.

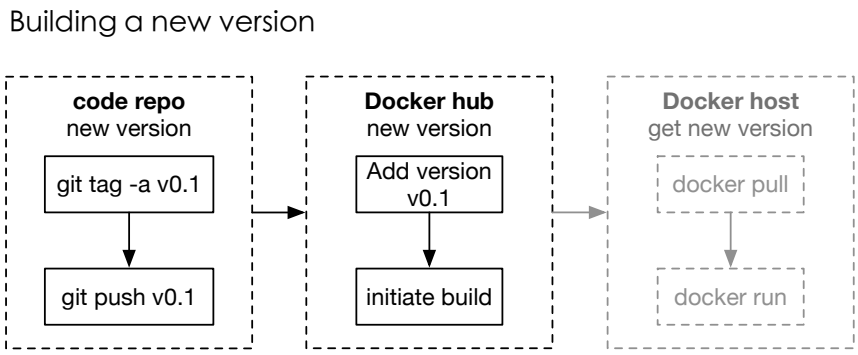

Figure 5.1: Building a new version

But there is a lot of stuff that is happening in the background, and there is a need for a specification that instructs Docker of how it should do the building. This is done through something called a Dockerfile. The file contains information about what image it should be based on, what software requirements the application needs, and what other commands it should run.

A test application has been built which is a simple python program which shows a web page with its version, and the possibility of connecting it to other services (code can be found in appendix A.15 on page 123). The

following Dockerfile is used to download and install the all of the needed packages to be able to run the application with Docker.

Dockerfile

```
FROM ubuntu:14.04
MAINTAINER Lars Haugan
RUN apt-get install -y -q curl python-all python-pip
ADD ./webapp /opt/webapp/
WORKDIR /opt/webapp
ADD requirements.txt /opt/webapp/
RUN pip install -r requirements.txt
EXPOSE 80
CMD ["python", "webapp.py"]
```

This file is used in both the case of the service being built on Docker Hub or locally. It is also required that this file is in the code repository, to ensure that different versions of required packages are handled with the correct application version. The following command is how the container would be built on a local machine. The build is tagged with the name webapp, and the version v0.1. By specifying *dot* (.) at the end, the Dockerfile in the current directory is used.

```
$ docker build -t webapp:v0.1 .
```

When building a complete environment, there would also be need for additional verification of the container, with the usage of continuous integration tools that are able to test that the container has been built correctly. This is however not in the scope of this thesis, but is fully implementable in the process.

The new version of the software is now built, and the container is available on the different Docker hosts, and ready to be taken into use. It contains all of the required packages and commands needed to start the application it contains. The new container is available on any host and can be downloaded and run with the following commands.

```
$ docker pull larhauga/webapp:v0.1
Trying to pull repository docker.io/larhauga/webapp ...
d0af202cf5ea: Download complete
...
Status: Downloaded newer image for docker.io/larhauga/webapp:v0.1
$ docker run -d -P larhauga/webapp:v0.1
```

The new version and service is now ready for deployment, and here the prototype and the processes of model, comes into the picture.

5.3 Process 2: New service to be deployed

A new application, the webapp, is ready to be deployed into the environment. The process of getting ready for deployment is done, and the newest code tagged with a version, and a Docker container has been built to the specifications of the Dockerfile. Now the container needs to be deployed, and the meta data added into the system.

This is defined by the second process in the model (4.4.2 on page 54). The process defines the needed actions, but also what changes are needed and in which order they need to be performed. The different actions have been implemented into the prototype as to enable the deployment of new services with the usage of containers.

The goal has been to create a minimal input operation, that only takes the least needed parameters in an easy way. To enable this in a good way on the command line, the process has been split up into two different commands. This is shown in figure 5.2 where the service is first created before it is connected to the service tree, by either creating a new endpoint or connect to an existing endpoint. If a more user friendly interface, like a web GUI is used, this process would be the same as outlined in the model.

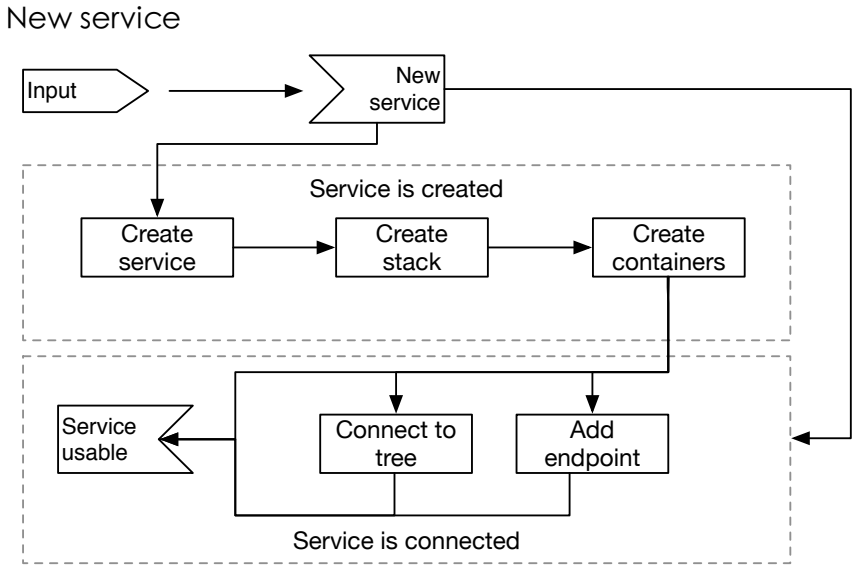

Figure 5.2: New service process diagram

In the prototype command line tool, this relates to the two command options to manage.py, *addservice* and *addendpoint*. All of the available options are shown in the appendix (addservice (A.2) on page 97 and addendpoint (A.3) on page 97).

5.3.1 Adding the service

When adding a service, the minimum needed parameters, is the name *(webapp)*, Docker image (*-i 'larahuga/webapp'*), number of stacks (*-s 2*) and the versions of the image to add. If only one version is listed, this version will be added as to comply with the specified minimum Stack Height.

```
$ python manage.py addservice webapp -i 'larhauga/webapp' -s 2 -v v0.1 v0.2 v0.3
```

When the service is added, the containers are launched on their designated hosts. In the case of the command above, this would launch three containers in two stacks.

A look at the code (in listing 5.2), shows that the actions defined in the model is present, where the service is first created on line 6, the new stack(s) are created at line 14, and the new containers are created at line 17 after the versioning is checked to be in compliance with the model. The function for creating containers is however an abstraction, that implements constraint checking and looping, and the deployment of the containers.

Listing 5.2: Initiation of a new service

```
1  def new_service(args):
2      """Process function for deploying service
3      Arguments:
4          args: name, image, versions, stacks
5      """
6      service = basefunc.create_service(args.name)
7      if not service:
8          print "Service not registered"
9          return
10     # Stacks = 1 if not present as argument
11     stacks = args.stacks if args.stacks else 1
12     for i in range(0, stacks):
13         host = service.choose_host()
14         stack = basefunc.create_stack(service, args.image, host)
15         try:
16             if basefunc.check_versions(None, args.versions):
17                 basefunc.create_containers(stack, args.versions)
18             else:
19                 print "Versions not in compliance with constraints"
20         except StandardError as e:
21             print e
```

5.3.2 New endpoint for the new service

When the service has been added, and the containers launched, an endpoint is needed to enable the service for use. This is the functionality of the option *addendpoint*. It takes the name of the service, the public port and the name of the stage. Other options also apply, but these are the most basic that enables the endpoint, such as the stack pointer, which defines which of the containers are connected to the newly created endpoint.

```
$ python manage.py addendpoint -s webapp -p 80 prod
```

The following code shows how the endpoint is added, which is done at line 6 (5.3), as long as the service exists. If it does not work, the session is rolled back.

Listing 5.3: Initiation of a new endpoint

```
1  def new_endpoint(args):
2    """Creates a new endpoint connected to a service"""
3    service = basefunc.get_service(args.service)
4    if service:
5      try:
6        endpoint = basefunc.make_endpoint(service, args.name, args.port, stackpointer=
       args.stackpointer)
7        basefunc.session.commit()
8        basefunc.view_endpoint(None, obj=endpoint)
9      except IntegrityError as e:
10       print "Endpoint not added: %s" % (e.message)
11       basefunc.session.rollback()
```

The first service that were created in the last section is now added as a new service, and no less than 6 containers with the specified versions are launched and running on the hosts specified in the config file.

5.3.3 Creating a service tree

Before a service tree can be created, there is the need for at least two services. At this point, only one service exist, so another service needs to be added. The following command adds one of the training apps that is provided by Docker (It is downloaded from the Docker Hub), and two stacks are created with containers.

```
$ python manage.py addservice training -i 'training/webapp' -s 2 -v 'latest'
```

The option *connect* enables a service to be connected to another service. This is done in a parent-child relation. The new service that is called *training* should be a child of the webapp that were created earlier. This is done by specifying that the webapp service should be connected to the child (-c) training, and that they are connected to the endpoint webapp-endpoint-prod.

```
$ python manage.py connect webapp -c training -e webapp-endpoint-prod
```

5.4 Process 3: New version of a service

After a while, a new version of the application webapp is ready to be deployed. It does not mean that it is ready for production, but it is ready to be deployed into the environment, and combined with other applications.

The implementation is defined by the third process in the model (4.4.3 on page 56), where the actions defined has been implemented in the prototype.

This process fits, as it is now implemented, into the continuous delivery strategy of release engineering. The model does also support fully integration with continuous deployment, making the system ready for high deployment scenario's.

As with the previous process, this process has been split into two separate commands. As shown in figure 5.3, the deployment process of the new version, and the change of getting it into the service trees are separated. The first block illustrates the new containers being deployed into the environment, and pushed onto the stacks of the service. When this is done, they can be taken into use, but this needs another command, to change the tree. Branch tree is grayed out, as this is not implemented in the prototype due to the time constraints of the project.

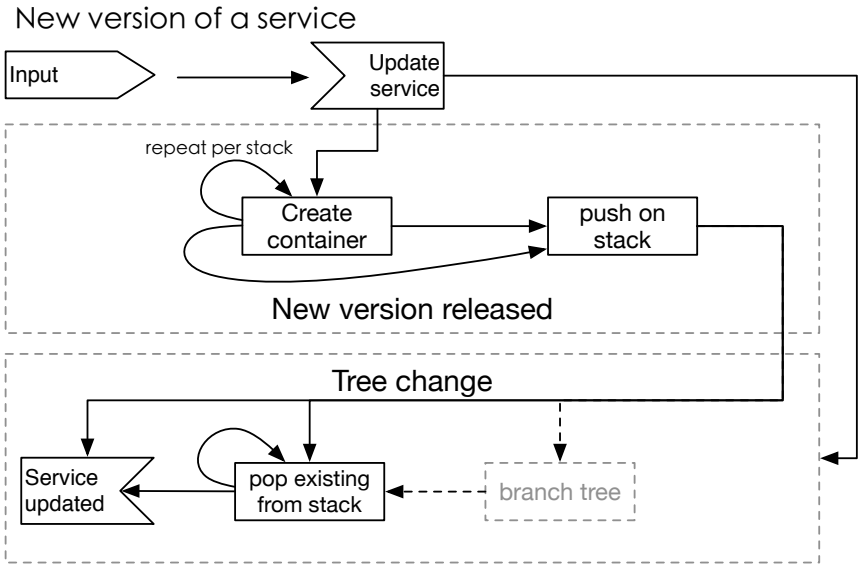

Figure 5.3: New version process diagram

5.4.1 Deploying the new version

The first operation that needs to be taken when releasing a new version of the service is to deploy the new container with the newer version. The service also runs multiple stacks, which means that the new container should be deployed to all of the stacks.

The new version is deployed with the following command *release*. The options this needs is the name of the service and the version number of the new version. The command pushes the new version, version *v0.2*, onto all of the stacks of the service *webapp*. As the defined constraint stack height is

configured to be three, the new version is the fourth container on the stack. The service now has a total of 8 containers running.

```
$ python manage.py release webapp -v v0.2
Container webapp-stack1-app4_0x985367194 running on docker01, port 49270
Container webapp-stack2-app4_0x357324409 running on docker02, port 49233
```

This command resolves to the function new_version as shown in the code below (listing 5.4), which uses the function *push_on_stack* from basefunc on line 12, as defined as the action to push new versions in the model. This is done for every stack that the service has. As seen on line 9, also here is the versioning checked, so no action is performed on the state, if the action that is going to be taken is not compliant with the current state.

Listing 5.4: Adding a new version with deploy

```
1  def new_version(args):
2      """New version of container
3      Finds service, pushes new containers
4      """
5      service = basefunc.get_service(args.service)
6      containers = []
7      for stack in service.stacks:
8          # Checking constraint of versioning
9          if basefunc.check_versions(stack, args.version):
10             print "Pushing new container with version %s on stack %s" % \
11                 (args.version, stack.name)
12             containers.append(basefunc.push_on_stack(stack, args.version))
13         else:
14             print "Versioning not in compliance with constraints"
15             return
16
17     for container in containers:
18         container.deploy_container()
19
20     basefunc.session.add_all(containers)
21     basefunc.session.commit()
```

An example of an action that would have resulted in a state that is not in compliance with the constraint would be when a new version is released, with a version number that is lower than the latest that was released. This would result in an error message, and the state is not changed.

```
$ python manage.py release webapp -v v0.1
Versioning not in compliance with constraints
```

The new version has been deployed, but is not yet enabled in a service tree. This needs another command as defined in figure 5.3.

5.4.2 Changing the stack

There are two different approaches for changing the stack in accordance with the model. This can be done by popping containers, or changing the

service trees connected to the service.

Pop is an action that is defined in the third process of the model (4.4.3 on page 56). The action enables the removal of a container of the stack. When the container is removed, the next container on the stack replaces the removed container. This is cascaded upwards in the stack, ensuring that new versions are deployed into the different service trees.

The following command uses the pop option, which in this case pops the container from position 0 (stackpointer) on all of the stacks of service webapp.

```
$ python manage.py pop -s webapp --stackpointer 0
```

The following code is a more simple overlay over the underlying function pop of the file basefunc (A.5 on page 102 line 290). Here, the constraints are checked, as to ensure that the minimum number of containers are running. The containers on the pointer is then removed from all of the stacks.

Listing 5.5: Popping an existing container

```
1 def pop(args):
2     service = basefunc.get_service(args.service)
3     if service:
4         if args.single:
5             stack = service.stack[0]
6             print "Only poping on stack %s" % stack.name
7             basefunc.pop(service, stack, position=args.stackpointer)
8         else:
9             basefunc.pop(service, None, position=args.stackpointer)
```

If the stack consisted of three containers, and the constraint on the state is configured with the least stack height of three, it is not allowed to pop a container before a new container is released on the stack.

If a pop is tried, and the pop would result in a non compliant state, the pop is denied with an error message:

```
$ python manage.py pop -s webapp1 -stp 0
StandardError: Not enough containers on stack. Poping not compliant with rules
```

Service tree branching is not implemented in the prototype, but is fully supported by the implemented state. The only requirement for its implementation is a new function that handles user input in the manage.py program, and a function to handle the business logic part.

5.5 Process 1: Showing the state

Now that the implementation of the processes of deploying a brand new service, and releasing new versions of a services has been showcased, a way to show what information is stored is needed. Through the commands

that has been executed, the data about the environment, the state, has been stored and kept up to date.

As every process and action uses the implementation when something changes in the environment, the state is also correct, and can be displayed.

There are different ways of displaying data, and the importance of illustrating data is to enhance understanding of the environment. This is a hard task, especially through a command-line-tool.

The result is sub-command of manage.py that prints out a table with the information about each of the different items stored.

5.5.1 Presentation of services

As the services are the most important item, these can be related to all of the other items in turn. This is a perfect opportunity to present key attributes from all of the items in the state.

The following command presents all of the services, the different stacks connected to the service and their containers. The service tree item is also present with the relation between the parent and child. Here both *webapp2* and *webapp3* are child services of *webapp1*. Webapp1 on the other hand is a main service, that are connected with a endpoint called prod. Some of the lesser important information is removed for better presentation here.

```
$ python manage.py show services
```

```
+---------+----------------+--------------------+----------+---------+--------------------+
| Service | Stacks         | Containers         | Parent   | Child   | Endpoints          |
+---------+----------------+--------------------+----------+---------+--------------------+
| webapp1 | webapp1-stack1 | webapp1-stack1-app2|          | webapp2 | webapp1-endpoint-  |
|         |                | webapp1-stack1-app3|          | webapp3 | prod               |
|         |                | webapp1-stack1-app4|          |         |                    |
+---------+----------------+--------------------+----------+---------+--------------------+
| webapp2 | webapp2-stack1 | webapp2-stack1-app1| webapp1  |         |                    |
|         |                | webapp2-stack1-app2|          |         |                    |
|         |                | webapp2-stack1-app3|          |         |                    |
+---------+----------------+--------------------+----------+---------+--------------------+
| webapp3 | webapp3-stack1 | webapp3-stack1-app1| webapp1  |         |                    |
|         |                | webapp3-stack1-app2|          |         |                    |
|         |                | webapp3-stack1-app3|          |         |                    |
+---------+----------------+--------------------+----------+---------+--------------------+
```

5.5.2 Presenting the other items

The same way the services can be presented, are also available for the other items. Another example of this is the endpoint. Here the service tree is illustrated as directional table, with also the stackposition of that special connection present.

```
$ python manage.py show endpoints

+-----------------------+---------+-------------+--------------------------------------+
| Endpoint name         | pubport | mainservice | tree                                 |
+-----------------------+---------+-------------+--------------------------------------+
| webapp1-endpoint-prod | 80      | webapp1     | +---------+---------+---------------+ |
|                       |         |             | | parent  | child   | stackposition | |
|                       |         |             | +---------+---------+---------------+ |
|                       |         |             | | webapp1 | webapp2 | 0             | |
|                       |         |             | | webapp1 | webapp3 | 0             | |
|                       |         |             | +---------+---------+---------------+ |
+-----------------------+---------+-------------+--------------------------------------+
```

Each of these items can also be limited through search, where a specific service, or multiple services can be found. The following command illustrates the searching, where the first uses a wildcard (%) which matches all. The next matches on the name webapp, which does not exist, while the last command finds a hit on webapp1.

```
$ python manage.py show service webapp%
>> webapp1,2,3
$ python manage.py show service webapp
>> None
$ python manage.py show service webapp1
>> webapp1
```

The most important processes of the model has been shown in the implementation of the prototype. This has enabled the verification of the model, and also an illustration of how the model works, as to enable better understanding of the use cases.

5.6 Apprising properties

So how is this different to a normal procedure? And how is the implementation and design of the process helping in reducing complexity, in relation to the defined apprising properties defined in the approach 6.1.1?

5.6.1 Time to completion and needed steps

A new deployment of a service would result in two stacks (that enables high availability) and the minimum of three containers (based on the constraint). Without any tools, the deployment of the corresponding 6 containers would relate to the usage of 12 commands on two different servers. The one command would be to download the new Docker image (*docker pull*) and the other to start the container (*docker run*).

With the usage of this implementation, this relates to **one** single command.

But the system is not yet available to the public. This is done through another command, which adds the endpoint. This command abstracts the information gathering of every container related to the service, enabling the configuration of load balancers, which otherwise would be configured manually. Not only that, it also ensures that the information is available and is correct.

Containers are much faster to start, than what a virtual machine manages. If we look at the time it takes to re-deploy the two services that are defined previously in this chapter, it takes only 12 seconds:

```
$ time python manage.py deploy containers
Container webapp-stack1-app1_0x175799796 running on docker01, port 49261
Container webapp-stack1-app2_0x402572336 running on docker01, port 49262
Container webapp-stack1-app3_0x670609799 running on docker01, port 49263
Container webapp-stack2-app1_0x319056192 running on docker02, port 49225
Container webapp-stack2-app2_0x912180658 running on docker02, port 49226
Container webapp-stack2-app3_0x799977594 running on docker02, port 49227
Container training-stack1-app1_0x638443664 running on docker01, port 49264
Container training-stack1-app2_0x996088675 running on docker01, port 49265
Container training-stack1-app3_0x563675956 running on docker01, port 49266
Container training-stack2-app1_0x121761286 running on docker02, port 49228
Container training-stack2-app2_0x614960769 running on docker02, port 49229
Container training-stack2-app3_0x161790286 running on docker02, port 49230

real    0m11.987s
user    0m1.883s
sys     0m0.146s
```

The time of running the other commands are negligible, and the one thing that takes time, is to download new versions of the containers. This is easily solved in real environments, by running an internal Docker Hub (as this is an open-source solution).

5.6.2 Experience and understanding

There are many different options available to the command line tool, but they are not all required. The reason for this is the intention of reducing complexity in the model. Many of the requirements that are defined in the model enables the elimination of user input, but also better storing of the right information.

One of the things that the user needs to know, is the name of the services. The name of the service is then reproduced into the naming of the endpoints, stacks and containers so that the relation is also present in the naming. This ensures that local inspection of a Docker host, reveals how the containers are related. This both reduces the information that is needed, and extends the understanding when investigating issues. It also ensures that the user does not need to know anything about the environment beforehand.

As the command line tool also employs the usage of argument parsing,

documentation of each parameter is available, with the usage of the -h parameter.

The process of deploying new services with this implementation is far more easy than what is needed when doing it the old fashioned way, with package managers. That process needed at least the following operations to get started:

- New virtual machines needs to be deployed
- The machine needs to be configured
- Packages installed on all of the new virtual machines
- Starting the new application
- Configuring the load balancer to point to the new virtual machines

In comparison, this only takes two commands. Where the options to the prototype manage.py are addservice and addendpoint.

5.6.3 Reproducibility

Since the state contains all of the different items and their attributes and relations, it is possible to reproduce the complete deployed environment with one command.

In summary the benefits of the implementation of the process can be defined by these points:

- Launching of any number of containers distributed
- Possibility of configuring load balancers
- Continuously updated information

Chapter 6

Discussion

This thesis has been aimed towards reducing complexity in release engineering, with the use of multi-stacked container environments. This is done through the design and implementation of the model.

The choices and findings made in this thesis will be discussed in this chapter, which includes the implementation, design of the model and the approach.

6.1 Implementation

The implementation of the model has enabled the verification of the validity of the created model, and enabled the understanding of how this implementation affects the process of release engineering.

This is done as the implementation creates an extra level of abstraction, that enables multiple operations at the same time. Yet, the operations does not only consist of orchestration operations, that are distributing the containers, but also collection of all the meta data that are necessary to create a replicable, scalable and reliable solution. One single step provides documentation, configuration and orchestration, which would otherwise need many.

6.1.1 Apprising properties

With the goal of reducing complexity, different surrogate variables were found that enables description of complexity from a user perspective. The implementation has helped with finding answers to the given properties.

Time to completion is reduced

The time to complete the process is difficult to define accurately. However, with the information that is available, it is quite evident, that the implementation will reduce the time needed to get a service up and running. This can be said as the implementation provides automation to the processes, and at the same time providing collection of data.

A business would require that the new, and changed deployments would be adequately documented. This is done automatically when using the tool, and with the defined constraints, the system is as desired.

Reduction in needed steps

The implementation works as an abstraction of the underlying needed actions needed to implement the model. If the model were to be implemented with every action needed, this would still result in a solution with fewer steps than what of a manual environment.

As presented in the implementation chapter, the needed steps to add a new service, and new versions of the service is drastically reduced as to the underlying commands needed. In addition the gathering of data, and its storage is included in the state. The state can therefore be seen as a configuration management database, that keeps track of the required data.

Less expertise needed. Higher degree of distribution

With fewer commands, less input and with less to no knowledge of the underlying environment, more people in a business is able to do release engineering operations on the system.

The information needed is always available in the state, which allows for decision making and distribution of tasks. The implementation envisages the implementation of web interfaces, which furthers the simplicity of the model, and reduces the already reduced needed input options.

However with the extended range the tool provides, there are still need for further development that may enable role based restrictions. This would enable the different departments and teams, that actually develops the application, to be invested in the release engineering processes. This could result in more affiliation with the processes of releasing new versions, where the teams are more personally responsible for the stability of the application in the different stages. The question then becomes wheter or not this is desirable?

However, it is possible to distribute roles with the implementation as the

tool for this job. This especially since the underlying architecture is abstracted, and the operations are limited by the state and its constraints.

Complete environment reproduction

When the complete environment is deployed, all the necessary information needed to deploy the services are stored persistently. This means that features like disaster recovery and backup of services is possible through the storage of the meta data that the state contains.

6.1.2 Reduced complexity through reduced variation

With the limitation of possible outcomes, and through standardization of the deployed environment, the outcome has resulted in reduced complexity. The number of operations needed, has been reduced along with the needed options for each of the operations.

The reduced complexity is achieved by limiting the amount of variation, as it is only possible to create an environment in the same way, based on the constraints that are defined. It should only be one way to do it.

As the needed commands and their options are limited to the least needed information, it means that it is possible to involve more parts of the organization, than what is normally possible.

The implementation shows that the model is working, and that it achieves the goals outlined in the problem statement, and in the approach.

6.1.3 Scalable solution

The solution that has been implemented is scalable and enables the usage of other tools. Through minimal change of the prototype, it is possible to employ different orchestration tools, that enables other means of deploying containers. With this solution on top of other orchestration solutions, it is possible to ensure that an environment is consistent, reproducible and susceptible for fast changing releases. This enables horizontal scaling for every part of the solution, either it is hosted on hardware on different sites or in clouds across the globe.

This solution enables scaling of the services added, without the need for more operations from the users. The number of operations needed to deploy a service with a broader horizontal scale, is independent from the number of commands needed. The services can scale horizontally by replicating the service stacks. By enabling this scaling solution, it also helps to reduce the complexity, as the same operation enables larger solutions, and easier management.

6.1.4 Structured state, not unstructured state

A big part of the design of the prototype in the implementation was the decisions of the usage of structured of unstructured storage of state. Structured data is data in a relational database, while unstructured data, (or actually semi-structured data) would be in the form of a python dict or json.

The json dict is easier and faster to implement, but not scalable. Databases on the other hand, will be able to handle scaling as well as containing the relations of the items in the model.

The choice was made to use a database, through the usage of an object relation mapper, which abstracted away the database, in favor of Python objects. This has also helped with the implementation of constraints, where the restrictions of the data model ensures that the data needs to be present. The integrity of the data is also ensured by the usage of database transactions.

The time it took to get familiar and solve the problems faced with the usage of the ORM SQLAlchemy, was worth the time it took, as it ensured the implementation of constraints on the state and consistency of data.

6.1.5 An expandable and Open Source solution

The solution is easily expandable. The most important part, the state, is implemented in a flexible way through the ORM, which enables easy alterations to the database, without the knowledge of SQL. Other parts are configurable, while new operations and libraries can be added.

As the solution is easily expandable, it also facilitates a high impact. The solution is made available publicly, and developed in a way that makes it possible to open source, and contribute to the community of interested parties of release engineering.

6.1.6 Ease of implementation

The prototype is not a complete implementation of the model, but it represent the most important features that the model envisions. It establishes the verification of the model by it being possible to implement, without modification. The different items in the model with their tuples of information, were easily translatable to the data structure. The different processes become the goals of the management tool, while the different actions became functions in the prototype.

This can be explained with the fact that the implementation was in mind when the model was devised. Is it possible that this means that the model

become limited as a result? It is possible, but it is believed that this is not the case, as the model was implemented after it being devised.

6.2 Model

A constraint-based approach has enabled an extended model, that is not a fixed set of rules, but a configurable and maintainable set of descriptions, that enables a common understanding of a multi-stacked container environment.

6.2.1 How its defined

A big part of the model were defined with formal descriptions and constraint. They are the fundamental definitions that enables the verification. These are defined by their formal description as well as illustrative models. The importance has been to enable the understanding and a basis for a model, which is extendible and general. This ensures that the results are applicable to a major part of solutions.

There are two distinct ways that the state can be defined. It can either be specified by a declarative or procedural process, where this thesis implements it in a procedural way. This means that the state is defined when the processes are followed, and not beforehand in a declarative way. This is seen in relation to the possibility of declaring an infrastructure as code. This thesis implements it as a database, but it can in practise be exported and used in an infrastructure as code way.

6.2.2 Simplicity, best practices or both?

The model were defined by the intention of reducing complexity. This results in cases where simplicity overrules the best practices of today. This does not mean that best practice is not implemented, but it plays a second role to the simplicity. The model is created by a good understanding of the domain and best practices, but choices were made that resulted in solutions that are not in correlation with the best practices of today.

An example of a case where the simplicity has overrules the best practice and domain knowledge, is how one service stack should be deployed. While best and common practice would have the different stages (of eg. dev, QA and prod) separated on machine level, connected to separated networks, and have other security implementations, this model does not take this into account. This is a result of the need for complexity, and an approach to flat cloud environments, rather than traditional architectures.

6.2.3 Enabling extensions

The focus area of the model has been to provide the fundamental model to ensure that complexity of a multi-stacked container environment would be reduced. The domain of release engineering with containers is large, and there are therefore many parts that are not explicitly implemented in the model.

A/B testing is one such tool that helps release engineering provide viable services. It is in short the basis of running two version of the service as a controlled experiment, to identify changes that affects the quality. It is not widely used, but provides effective ways of finding errors during releases. This is very attractive, and is fully supported by the model. This could be solved by letting one endpoint point to multiple trees, and give the different trees different weight which distribute the traffic based on the weight. This is implementable in a load balancer without breaking any of the constraints.

Another solution to solve A/B testing, is on the stack level, and not on the tree level. This could be done when using multiple stacks, where the operation is done first on one stack, and then on then other. However, this introduces more complexity in both the model and implementation, but would at the same time be more like a canary test environment.

In the model, the best practices has a right to be heard, and the different needs need to be taken into account. However, the model is kept as complex free as possible.

6.3 Approach

The approach of this thesis has been based on first designing the model, and then implementing the model, with a clear distinction between the two phases. From a research point of view, this has been the right approach, where the thinking is done first, and then it is implemented and tested.

This approach resulted in most of the time being focused on the design of the model, and then the focus shifted to the implementation. In the beginning of the project, this seemed like a good approach.

A classical, theoretical, approach the design of the model would be the beginning. However, programming is a creative process for us that are practically experienced with programming, which may have helped with the design of the model.

The question is if the project would have advanced further with a mix of the two phases from the start, as the time frame of the project is limited?

It is possible, but what would the result have been? It is possible that the focus on the importance of constraints would be limited, along with a process that were to focused on the best practice rather than a combination. A new tool would be created, that has the same attributes as what is already possible.

As described in the introduction, the is a need for a methodology that can verify the correctness of a state beyond best practice, for the complexity to truly be reduced. This gets at the need for strict handling, but in a right way. The mindset needs to be changed, and that is what the model offers.

So, if this project were to be done again, would a better result be achieved, if the design and implementation were combined?

6.4 Related work

Release engineering is a broad and large topic, with limited academic research. The field of study for this thesis is even more limited. This is mostly due to the fact that the field is mostly advanced by products and solutions made by the business markets, with large companies leading the way. In addition, a large community exists, that creates advancing open source solutions, such as Docker and OpenStack.

One has understood that there is a need for more research in the field, and as such new conferences has emerged, that combine the efforts of practitioners and researchers [12].

6.4.1 Infrastructure as code

This thesis has worked towards the reduction of complexity in the large and complex environments, where new solutions pollutes the existing environments with ever more complexity. Work has been done to combat this complexity, and provide new solutions to describe the complex environments that exists today.

TOSCA is an emerging standard which tries to enhance management and portability of cloud applications [41]. Wettinger et al. provides a way of using common management tools to define the infrastructure as code [45]. The project also states that one of the major concerns of today's enterprise IT is the (automated) management of composite applications and their portability [42].

With the usage of containers in this thesis, the major concerns is solved, and the state is the equivalent of infrastructure as code. However, the usage of configuration management systems as the provider of information of infrastructure is not good enough with the rapid change that the businesses of today are in need of. The TOSCA standard is recognized as a good standard, but the complexity is still present, as the same mindset is present

as with all other solutions. However, the solution shows that there is need for such a solution, and as TOSCA work towards solving the problems with todays tools, this thesis implements restrictions to what should be possible, and is related to container solutions, as the emerging solution for handling of software.

6.4.2 Continuous integration and the role of orchestration

One of the more talked about aspect of todays release engineering processes is continuous integration and delivery. This thesis help towards implementing a continuous integration, but does not include it. However, as the study done by Bellomo et al. it suggests that the architectural design decissions done are important to succeed [46]. Achieved in this thesis is the reduced deployment-cycle time, which means that new versions can be deployed faster into production. However, the achievement of the holy grail, as defined [46], this needs the usage of automated testing, which is not part of this thesis.

However with the usage of containers in this thesis some of the challenges outlined by the principles in the 'The deployment production line' by Humble, Read and North, is addressed. This has influenced the design of the model in multiple ways, but especially with the different stages that the application should go through before it reaches production. With the usage of containers, the first principle of [20] is achieved, where the container remains unchanged through the production line. However, the second principle is not achieved as part of the implementation of the model, as there is currently no support for configuration that are not inside the containers.

6.5 Impact

This thesis started with the desire to change and improve the situation in todays implementation of release engineering. It is today based on broadly implemented practice and tools, but there is change in business needs which affects how these processes are being handled. This results in the need to change, which forces processes to be implemented, and not just the engineering part.

If related to the best practice framework ITIL, this thesis has landed on a business related approximation to release engineering, where the focus on release engineering is seen as *service management*.

The model with its state and constraints is just a framework for the release engineering processes. The focus of the model is more related to the architecture and infrastructure than all of the different aspects of release engineering, but the processes of the model brings the aspects of release

engineering into the model. This was a way of simplifying how the world works, and therefore the processes also got simpler.

6.5.1 How does it fit?

So is there a place for the model and the implementation? Yes!

Even though Docker, and equivalent projects keeps on growing, and are acquiring new projects and firm all the time, there is still a place for this project. Most of these are solving important aspects of container solutions that are needed to meet the coming need for containers in release engineering.

There is a lot that happens in this problem domain, but this model and implementation stands out with its approximation to reduction of complexity as its main objective. Other solutions are lacking in the functionality of verifying the state.

There are no contradictions with the solutions being developed by the business and this implementation. On the contrary, they are mutually beneficial. This solution can be placed on top of other solutions. This implementation can verify the validity in regards to the constraints, while the underlying technology works with the orchestration and life cycle of the containers.

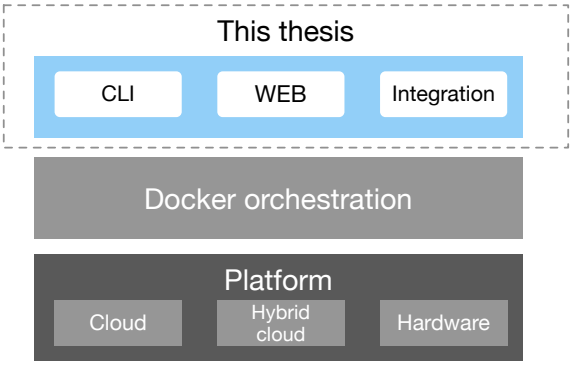

Figure 6.1: Thesis on the stack

This provides a bric in the future work needed to improve release engineering and service management processes with containers. With this implementation and its feature of verifying the state, a basis for future work of self healing systems is possible.

6.5.2 Road ahead

So what is the future work and how is the road ahead from this point? What possible expansions are there?

State of the implementation

The implementation of the model is not finished. There is more work to be done before it can be used in production environments, and it also needs to be tested in production like environments, where it is possible to see how businesses relates to the changes. A lot of work needs to be done before the tool can meet production standards. It would work, but a lot of features that are both nice-to-have, and mission critical are missing, due to the time constraints.

The prototype has code to handle management of the load balancer HAProxy, but there is still need for its integration into each of the different actions. This is not part of the prototype, as this a complex case of implementation. The needed functionality is in place, but the specific handling needs to be implemented.

Like a configuration management tool for environments

The key to this implementation, is the state and constraints. It can be related to configuration management, where the different properties are configured, and the system is defined after the defined patterns. Constraints can be seen as policies, and the state can be seen as the state of the system.

The model however is at this time not good at handling deviations from the desired state, as the configuration management solutions are doing. If this were to be implemented in the model (and implementation) this would give us self healing systems with a proactive behaviour.

With the addition of constraints, that are providing definitions and restrictions about the quality of service, the result would be an approach that is built for a proactive behaviour. This could result in automatic horizontal scaling. The horizontal scaling is already a part of both the model and the implementation, and are ready to be used.

Large sets of the implementation is something that is possible to build further on. The state as the central component it is, is done in a way that enables further development.

Further down the road

Further down the road, there are possibilities of numerous other features and outcomes. There is however need for further research on the topic, as to overcome new challenges that arises with containers and their distribution.

Monitoring and analysis is one part that needs further exploration, as to enable more proactive and reactive systems, that are better featured for

a large production environment. Monitoring would help resolve issues, while analysis of the environment would help solve future errors, where the result would be a proactive and autonomic solution.

Migration between clouds is easy with the model and container solutions. There is however issues that needs solving, that are of a technical nature. How one can migrate data and container specific configuration is not solved, and there is need for more research on this topic to enable the migration of with host specific dependencies, such as data.

Service discovery is one of the things that enables a dynamic environment. Service discovery is in this model solved as the state contains the information about the services. But, this also means that the services does not have a perspective of where they are in the environment. To enable better configuration management, an implementation of configuration discovery is needed for further improve the implementation. There are already different solutions that enables this functionality [51], which may help further the impact of the implementation.

Continuous delivery and integration is not included as features of the implementation, but is fully supported of the model. However, there is need for further work to improve the processes that can implement the testing such as A/B testing and canary on these kinds of processes. There are many possibilities that needs to be explored.

Chapter 7

Conclusion

The goal of this thesis has been to design and implement a model in order to reduce the complexity in todays release engineering processes with container based services. This thesis provides a model that has been designed with the reduction of complexity in mind, and each key aspect of the model adheres to this intention.

The result has been a model that is created, that defines the different parts of an environment that is needed to both define, and build a system that can handle the complexity of todays composite systems, with the new turn of them running in containers.

This thesis provides answers to service management with containers where the reduction of complexity is the intention. This enables both system administrators and businesses to meet the coming challenges with increasing amount of systems, and facilitates the further scaling with limited effort. Complex release engineering processes has been reduced to single restrictive steps, enforcing simplicity.

The problem statements are satisfied with the designed and developed model, that has enabled reduced complexity when releasing and managing new services. A prototype is implemented, that validates the validity of the model, and also defines a fundamental implementation, that enables future work.

Bibliography

[1] Statista. *Number of worldwide internet users from 200 to 2014*. 2015. URL: http://www.statista.com/statistics/273018/number-of-internet-users-worldwide/ (visited on 05/02/2015).

[2] Internet Live Stats: Internet Users. 2015. URL: http://www.internetlivestats.com/internet-users/#trend (visited on 16/02/2015).

[3] Facebook. *Facebook official company info*. 2015. URL: http://newsroom.fb.com/company-info/ (visited on 16/02/2015).

[4] Tim Berners-Lee. 'Information management: A proposal'. In: (1989).

[5] Tim Berners-Lee, Roy Fielding and Henrik Frystyk. *Hypertext transfer protocol–HTTP/1.0*. 1996.

[6] Berners-Lee Tim. 'www: past, present, and future'. In: *computer* 29.10 (1996), pp. 69–77.

[7] Wu He and Li Da Xu. 'Integration of Distributed Enterprise Applications: A Survey'. In: *Industrial Informatics, IEEE Transactions on* 10.1 (Feb. 2014), pp. 35–42. ISSN: 1551-3203. DOI: 10.1109/TII.2012.2189221.

[8] Tom Yoon and Pamela Carter. 'Investigating the antecedents and benefits of SOA implementation: a multi-case study approach'. In: *AMCIS 2007 Proceedings* (2007), p. 195.

[9] David Sprott and Lawrence Wilkes. 'Understanding service-oriented architecture'. In: *The Architecture Journal* 1.1 (2004), pp. 10–17.

[10] Sachin Agarwal. *API vs. SOA? Are they different?* Dec. 2013. URL: https://blog.soa.com/api-vs-soa-different/ (visited on 09/03/2015).

[11] Netcraft. *February 2015 Web Server Survey*. Feb. 2014. URL: http://news.netcraft.com/archives/2015/02/24/february-2015-web-server-survey.html (visited on 10/03/2015).

[12] Bram Adams et al. '1st international workshop on release engineering (releng 2013)'. In: *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press. 2013, pp. 1545–1546.

[13] William B Frakes and Kyo Kang. 'Software reuse research: Status and future'. In: *IEEE transactions on Software Engineering* 31.7 (2005), pp. 529–536.

[14] James A Whittaker, Jason Arbon and Jeff Carollo. *How Google tests software*. Addison-Wesley, 2012.

[15] Shane McIntosh et al. 'An empirical study of build maintenance effort'. In: *Proceedings of the 33rd international conference on software engineering*. ACM. 2011, pp. 141–150.

[16] Robert DeLine. 'Avoiding packaging mismatch with flexible packaging'. In: *Software Engineering, IEEE Transactions on* 27.2 (2001), pp. 124–143.

[17] André Van der Hoek and Alexander L Wolf. 'Software release management for component-based software'. In: *Software: Practice and Experience* 33.1 (2003), pp. 77–98.

[18] Dalibor Siroky. *Release Management vs Release Engineering*. 2014. URL: http://www.plutora.com/insights/2014/release-management-vs-release-engineering/ (visited on 11/03/2015).

[19] Dan Woods. *Enterprise services architecture*. " O'Reilly Media, Inc.", 2003.

[20] Jez Humble, Chris Read and Dan North. 'The deployment production line'. In: *Agile Conference, 2006*. IEEE. 2006, 6–pp.

[21] A. Hochstein, R. Zarnekow and W. Brenner. 'ITIL as common practice reference model for IT service management: formal assessment and implications for practice'. In: *e-Technology, e-Commerce and e-Service, 2005. EEE '05. Proceedings. The 2005 IEEE International Conference on*. Mar. 2005, pp. 704–710. DOI: 10.1109/EEE.2005.86.

[22] Gene Kim, Kevin Behr and George Spafford. *The phoenix project: A novel about IT, DevOps, and helping your business win*. IT Revolution Press, 2013.

[23] *2013 State of DevOps Report*. 2013. URL: https://puppetlabs.com/wp-content/uploads/2013/03/2013-state-of-devops-report.pdf (visited on 05/03/2015).

[24] IT Revolution Press (blog). *The Convergence of DevOps*. 2012. URL: http://itrevolution.com/the-convergence-of-devops/ (visited on 04/03/2015).

[25] Paul Hammond John Allspaw. *Velocity 09: 10+ Deploys Per Day: Dev and Ops Cooperation*. 2013. URL: https://www.youtube.com/watch?v=LdOe18KhtT4 (visited on 05/03/2015).

[26] David Oppenheimer, Archana Ganapathi and David A Patterson. 'Why do Internet services fail, and what can be done about it?' In: *USENIX Symposium on Internet Technologies and Systems*. Vol. 67. Seattle, WA. 2003.

[27] Nigel Kersten "Nicole F. Velasquez Gene Kim and Jez Humle". *2014 State of DevOps report*. 2014. URL: https://puppetlabs.com/sites/default/files/2014-state-of-devops-report.pdf (visited on 06/03/2015).

[28] Dirk Merkel. 'Docker: lightweight linux containers for consistent development and deployment'. In: *Linux Journal* 2014.239 (2014), p. 2.

[29] Wes Felter et al. 'An Updated Performance Comparison of Virtual Machines and Linux Containers'. In: *technology* 28 (2014), p. 32.

[30] Ivan Melia et al. *Linux Containers: Why They're in Your Future and What Has to Happen First*. White paper. Cisco, RedHat, 2014.

[31] Michael Kerrisk. *Namespaces in operation, part 1: namespaces overview*. 2013. URL: http://lwn.net/Articles/531114/ (visited on 18/03/2015).

[32] Docker Inc. *Understanding Docker*. 2014. URL: https://docs.docker.com/introduction/understanding-docker/ (visited on 18/03/2015).

[33] Daniel Walsh. *Docker and SELinux*. 2014. URL: https://www.youtube.com/watch?v=zWGFqMuEHdw (visited on 10/03/2015).

[34] Berkley Distribution. *chroot man page*. Aug. 1985. URL: http://www.freebsd.org/cgi/man.cgi?query=chroot&apropos=0&sektion=2&manpath=2.10+BSD&arch=default&format=html (visited on 10/03/2015).

[35] Bill Cheswich. 'An Evening with Berferd: In Which a Cracker is Lured, Endured, and Studied'. In: *USENIX Summer Conference Proceedings* 29.10 (1996), pp. 69–77.

[36] Matt Helsley. 'LXC: Linux container tools'. In: *IBM devloperWorks Technical Library* (2009).

[37] Stéphane Graber. *Linux Containers*. 2014. URL: https://linuxcontainers.org (visited on 19/03/2015).

[38] Solomon Hykes. *Docker README*. 2013. URL: https://github.com/docker/docker/commit/0db56e6c519b19ec16c6fbd12e3cee7dfa6018c5.

[39] David A Wheeler. 'Cloud Security: Virtualization, Containers, and Related Issues'. In: ().

[40] Alex Polvi. *CoreOS is building a container runtime, Rocket*. 2014. URL: https://coreos.com/blog/rocket/.

[41] Topology OASIS. 'Orchestration Specification for Cloud Applications Version 1.0, May 2013'. In: *URL http://docs. oasis-open. org/tosca/TOSCA/v1. 0/cs01/TOSCA-v1. 0-cs01. html* (2013).

[42] Tobias Binz et al. 'TOSCA: Portable automated deployment and management of cloud applications'. In: *Advanced Web Services*. Springer, 2014, pp. 527–549.

[43] J. Wettinger, U. Breitenbucher and F. Leymann. 'Standards-Based DevOps Automation and Integration Using TOSCA'. In: *Utility and Cloud Computing (UCC), 2014 IEEE/ACM 7th International Conference on*. Dec. 2014, pp. 59–68. DOI: 10.1109/UCC.2014.14.

[44] Oliver Kopp et al. 'Winery–A modeling tool for TOSCA-based cloud applications'. In: *Service-Oriented Computing*. Springer, 2013, pp. 700–704.

[45] Johannes Wettinger et al. 'Integrating Configuration Management with Model-driven Cloud Management based on TOSCA.' In: *CLOSER*. 2013, pp. 437–446.

[46] S. Bellomo et al. 'Toward Design Decisions to Enable Deployability: Empirical Study of Three Projects Reaching for the Continuous Delivery Holy Grail'. In: *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*. June 2014, pp. 702–707. DOI: 10.1109/DSN.2014.104.

[47] Kyrre Begnum, Nii Apleh Lartey and Lu Xing. 'Cloud-oriented virtual machine management with mln'. In: *Cloud Computing*. Springer, 2009, pp. 266–277.

[48] Erik Christensen et al. *Web services description language (WSDL) 1.1 http://www. w3. org*. Tech. rep. TR/wsdl, 2001.

[49] Jan van Bon. *ITIL Pocket Guide*. "Van Haren Publishing, Zaltbommel", 2011.

[50] *Docker-py*. 2015. URL: https://github.com/docker/docker-py.

[51] *The Docker Ecosystem: Service Discovery and Distributed Configuration Stores*. Feb. 2015. URL: https://www.digitalocean.com/community/tutorials/the-docker-ecosystem-service-discovery-and-distributed-configuration-stores.

Chapter A

Appendix

A.1 Prototype

The prototype contains a lot of different files, which are listed below. The code is also available on Github: https://github.com/larhauga/cmanage

Listing A.1: manage.py: User input options

```
usage: manage.py [-h]
                 {init,show,addservice,addendpoint,connect,release,pop,delete,deploy,upstream}
Service management tool
```

Listing A.2: manage.py: Add service options

```
usage: manage.py addservice [-h] [-i IMAGE] [-s STACKS]
                            [-v VERSIONS [VERSIONS ...]]
                            name

Add new service

positional arguments:
  name                  Name of service

optional arguments:
  -h, --help            show this help message and exit
  -i IMAGE, --image IMAGE
                        Docker image path for pull
  -s STACKS, --stacks STACKS
                        Number of stacks to add
  -v VERSIONS [VERSIONS ...], --versions VERSIONS [VERSIONS ...]
                        Version(s) to add
```

Listing A.3: manage.py: Add endpoint options

```
usage: manage.py addendpoint [-h] -s SERVICE [-p PORT] [-stack STACKPOINTER]
                             name

Add a new endpoint to service

```

```
positional arguments:
 name                Name of endpoint

optional arguments:
 -h, --help          show this help message and exit
 -s SERVICE, --service SERVICE
                     Name of the main service
 -p PORT, --port PORT  Public port
 -stack STACKPOINTER, --stackpointer STACKPOINTER
                     Default stack pointer for tree
```

A.1.1 Python files

The code can also be found on GitHub
https://github.com/larhauga/cmanage

Listing A.4: manage.py: Main program

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

# Global imports
import argparse
import requests
from sqlalchemy.exc import IntegrityError

# Local imports
import basefunc
from lib import init
from lib import config as cfg
config = cfg.get_config()
rules = cfg.get_rules_config()
cconfig = cfg.get_container_config()


def create_db(args):
    """Initiates based on config"""
    init.init(create=True)


def new_service(args):
    """Process function for deploying service
    Arguments:
        args: name, image, versions, stacks
    """
    service = basefunc.create_service(args.name)
    if not service:
        print "Service not registered"
        return
    # Stacks = 1 if not present as argument
    stacks = args.stacks if args.stacks else 1
    for i in range(0, stacks):
        host = service.choose_host()
        stack = basefunc.create_stack(service, args.image, host)
        try:
            if basefunc.check_versions(None, args.versions):
```

```
39              basefunc.create_containers(stack, args.versions)
40          else:
41              print "Versions not in compliance with constraints"
42      except StandardError as e:
43          print e
44
45
46  def new_endpoint(args):
47      """Creates a new endpoint connected to a service"""
48      service = basefunc.get_service(args.service)
49      if service:
50          try:
51              endpoint = basefunc.make_endpoint(service, args.name, args.port, stackpointer=
            args.stackpointer)
52              basefunc.session.commit()
53              basefunc.view_endpoint(None, obj=endpoint)
54          except IntegrityError as e:
55              print "Endpoint not added: %s" % (e.message)
56              basefunc.session.rollback()
57
58
59  def connect(args):
60      """Connect two or more services together"""
61      pobj = []
62      cobj = []
63      endpoint = basefunc.get_endpoint(args.endpoint)
64      service = basefunc.get_service(args.service)
65
66      if endpoint and service:
67          # Find all parents and childs
68          if args.parent:
69              for parent in args.parent:
70                  pobj.append(basefunc.get_service(parent))
71          if args.child:
72              for child in args.child:
73                  cobj.append(basefunc.get_service(child))
74
75          try:
76              basefunc.add_relation(endpoint, service, pobj, cobj, args.stackpointer)
77          except IntegrityError as e:
78              print "Duplicate detected: %s" % e.message
79      else:
80          print "Endpoint (%s) or service (%s) not found" % (args.endpoint,
81                                                  args.service)
82
83
84  def new_version(args):
85      """New version of container
86      Finds service, pushes new containers
87      """
88      service = basefunc.get_service(args.service)
89      containers = []
90      for stack in service.stacks:
91          # Checking constraint of versioning
92          if basefunc.check_versions(stack, args.version):
93              print "Pushing new container with version %s on stack %s" % \
94                  (args.version, stack.name)
95              containers.append(basefunc.push_on_stack(stack, args.version))
96          else:
```

```
97          print "Versioning not in compliance with constraints"
98          return
99
100     for container in containers:
101         container.deploy_container()
102
103     basefunc.session.add_all(containers)
104     basefunc.session.commit()
105
106
107 def pop(args):
108     service = basefunc.get_service(args.service)
109     if service:
110         if args.single:
111             stack = service.stack[0]
112             print "Only poping on stack %s" % stack.name
113             basefunc.pop(service, stack, position=args.stackpointer)
114         else:
115             basefunc.pop(service, None, position=args.stackpointer)
116
117
118 def view(args):
119     # show all services
120     if 'services' in args.type:
121         basefunc.view_services()
122     # Show one service like name
123     elif 'service' in args.type and args.name:
124         basefunc.view_service(args.name)
125     # Show all endpoints
126     elif 'endpoints' in args.type:
127         basefunc.view_endpoints()
128     # Show one endpoint
129     elif 'endpoint' in args.type and args.name:
130         basefunc.view_endpoint(args.name)
131     elif 'containers' in args.type:
132         basefunc.view_containers()
133     elif 'stack' in args.type and args.name:
134         service = basefunc.get_service(args.name)
135         basefunc.view_stack(service)
136
137
138 def delete(args):
139     if 'containers' in args.type:
140         basefunc.remove_all_containers()
141     else:
142         print "Not running. Argument %s not supported" % (args.type)
143
144
145 def deploy(args):
146     if 'containers' in args.type:
147         basefunc.deploy_all_containers()
148
149
150 def upstream(args):
151     """Get the upstream versions of a container"""
152     url = "https://registry.hub.docker.com/v1/repositories/%s/tags"
153     service = basefunc.get_service(args.service)
154     if service:
155         if service.stacks:
```

```
156         r = requests.get(url % service.stacks[0].image)
157         # THIS IS REALLY INSECURE! Need a fast implementation :)
158         tags = eval(r.text)
159         for tag in tags:
160           print "Tag: %s, layer: %s" % (tag['name'], tag['layer'])
161       else:
162         print "Docker image not defined"
163     else:
164       print "Service not found"
165
166
167 if __name__ == '__main__':
168     parser = argparse.ArgumentParser(
169       description='Service management tool',
170       formatter_class=argparse.RawDescriptionHelpFormatter
171     )
172
173     subparser = parser.add_subparsers()
174     # When init is sendt
175     parser_init = subparser.add_parser('init', description='Initiates database')
176     parser_init.set_defaults(func=create_db)
177
178     # Parsing for show command
179     parser_view = subparser.add_parser('show', description='Show the different items')
180     parser_view.add_argument('type', type=str,
181                              help='Different types of items to show',
182                              choices=['services', 'service', 'endpoints',
183                             'endpoint', 'stacks', 'stack', 'containers'])
184     parser_view.add_argument('name', type=str, nargs='?',
185                              help='Name of service. Use percent to search')
186     parser_view.set_defaults(func=view)
187
188     # Parser for the addservice command
189     parser_add_service = subparser.add_parser('addservice',
190                                               description="Add new service")
191     parser_add_service.add_argument('name', type=str, help='Name of service')
192     parser_add_service.add_argument('-i', '--image', type=str,
193                                     help='Docker image path for pull')
194     parser_add_service.add_argument('-s', '--stacks', type=int, default=1,
195                                     help='Number of stacks to add')
196     parser_add_service.add_argument('-v', '--versions', nargs='+', type=str,
197                                     help='Version(s) to add')
198     parser_add_service.set_defaults(func=new_service)
199
200     parser_add_endpoint = subparser.add_parser('addendpoint',
201                                 description='Add a new endpoint to service')
202     parser_add_endpoint.add_argument('-s', '--service', type=str, required=True,
203                                      help='Name of the main service', )
204     parser_add_endpoint.add_argument('name', type=str, help='Name of endpoint')
205     parser_add_endpoint.add_argument('-p', '--port', help='Public port')
206     parser_add_endpoint.add_argument('-stack', '--stackpointer', type=int, default=0,
207                                      help='Default stack pointer for tree')
208     parser_add_endpoint.set_defaults(func=new_endpoint)
209
210     parser_connect = subparser.add_parser('connect',
211                                           description='Connect services')
212     parser_connect.add_argument('service', type=str, help='Service to connect')
213     parser_connect.add_argument('-p', '--parent', nargs='+', type=str,
214                                 help='Parent service')
```

```
215 parser_connect.add_argument('-c', '--child', nargs='+', type=str,
216                     help='Childe service')
217 parser_connect.add_argument('-e', '--endpoint', type=str,
218                     help='Endpoint')
219 parser_connect.add_argument('-stp', '--stackpointer', type=int,
220                     help='Optional stack pointer')
221 parser_connect.set_defaults(func=connect)
222
223
224 parser_do_release = subparser.add_parser('release',
225                                 description='Do a service release')
226 parser_do_release.add_argument('service', type=str,
227                       help='Service to do release on')
228 parser_do_release.add_argument('-v', '--version', type=str, required=True,
229                       help='Version of the new container')
230 parser_do_release.set_defaults(func=new_version)
231
232 parser_pop = subparser.add_parser('pop',
233                         description='Pop container from service')
234 parser_pop.add_argument('service', type=str,
235                     help='Service to pop container from')
236 parser_pop.add_argument('-stp', '--stackpointer', type=int, default=0,
237                     help='Which container to pop')
238 parser_pop.add_argument('-s', '--single', default=False, action='store_false',
239                     help='Remove from one stacks')
240 parser_pop.set_defaults(func=pop)
241
242 parser_delete = subparser.add_parser('delete', description='Delete')
243 parser_delete.add_argument('type', type=str, choices=['containers',
244                                       'services', 'stacks'])
245 parser_delete.add_argument('-n', '--name', type=str,
246                       help='Name of item to remove')
247 parser_delete.set_defaults(func=delete)
248
249 parser_deploy = subparser.add_parser('deploy', description='Deploy')
250 parser_deploy.add_argument('type', type=str, choices=['containers'])
251 parser_deploy.add_argument('-n', '--name', type=str,
252                       help='Name of item to deploy')
253 parser_deploy.set_defaults(func=deploy)
254
255 parser_upversions = subparser.add_parser('upstream',
256                                 description='Find upstream versions')
257 parser_upversions.add_argument('service', type=str,
258                       help='The version to check versions on')
259 parser_upversions.set_defaults(func=upstream)
260
261 args = parser.parse_args()
262 args.func(args)
```

Listing A.5: basefunc.py: Basic model functions

```
1 #!/usr/bin/env python
2 # -*- coding: utf-8 -*-
3
4 # Script for defining base functions
5 from sqlalchemy.exc import IntegrityError
6 from sqlalchemy import func
7 from terminaltables import AsciiTable
```

```
8  from exceptions import NotImplementedError, StandardError
9
10 from lib import config as cfg
11 from lib import init
12
13 from lib.service import Service, Service_tree
14 from lib.stack import Stack
15 #from lib.tree import Tree
16 from lib.container import Container
17 from lib.endpoint import Endpoint
18 #from lib.stage import Stage
19 from docker.errors import APIError
20 import lib.lib_docker as docker
21
22 config = cfg.get_config()
23 logging = cfg.get_logger()
24 rules = cfg.get_rules_config()
25
26 session = init.init()
27
28
29 # Service functions
30 def create_service(name):
31     """Creates a new service
32     Arguments:
33         name: Name of the new service
34     Constraints:
35         name: unique
36     """
37     # HERE NEEDS CHECK OF STATE
38     # check if service exists (name)
39
40     if not session.query(Service).filter(Service.name == name).first():
41         try:
42             s = Service(name)
43             session.add(s)
44             session.commit()
45         except IntegrityError as e:
46             print('Port allready in use: %s' % str(e.orig))
47             return None
48     else:
49         print('Service already exists')
50         return None
51     return s
52
53 def get_service(name):
54     """Searches after the service name"""
55     return session.query(Service).filter(Service.name == name).first()
56
57 def view_service(name):
58     """Prints out information about one service based on its name"""
59     view_services(filterquery=session.query(Service).filter(Service.name.like(name)))
60
61 def view_services(filterquery=None):
62     """Prints out list of services and its relevant information"""
63     table = []
64     table.append(["Service Name", "Stacks", "Containers", "Parent S", "Child S", "Endpoints"
           ])
65     if filterquery:
```

```
66      services = filterquery.all()
67      #services = session.query(filterquery).all()
68    else:
69      services = session.query(Service).all()
70    if not services:
71      print "No services met the search"
72      return
73
74    for service in services:
75      state = service.get_state()
76      parents = [p['parent'] for p in state['parent']]
77      children = [c['child'] for c in state['childs']]
78      cs = []
79      for stack in state['stacks']:
80        for i, container in enumerate(stack['container']):
81          endpoint = service.tree_on_stack_pointer(i)
82          if endpoint:
83            cs.append("%s:%s:%s" % (container['name'], container['version'], endpoint.
       name))
84          else:
85            cs.append("%s:%s" % (container['name'], container['version']))
86        #cs.extend(["%s:%s" % (c['name'],c['version']) for c in stack['container']])
87      table.append([str(state['name']),
88                    "\n".join([ s['name'] for s in state['stacks'] if s]),
89                    str("\n".join(cs)),
90                    "\n".join(parents),
91                    "\n".join(children),
92                    "\n".join(state['endpoints'])])
93    t = AsciiTable(table)
94    t.inner_row_border = True
95    print t.table
96
97
98  # Stack functions
99  def create_stack(service, image, host):
100   """Creates an empty stack for containers
101   Arguments:
102     service: service object
103     image: base image for container
104     host: Where to run stack
105   Constraints:
106   """
107   # HERE NEEDS CONSTRAINTS CHECK
108   stack = Stack(service, image, host)
109   session.add(stack)
110   session.commit()
111   return stack
112
113 def update_stack(service):
114   """Update stack for service
115   Dont know what this will do yet.
116   Intended to add one container version on a stack...
117   """
118   raise NotImplementedError()
119
120 def view_stack(service, stackname=None):
121   """View the stack container version and position and tree points
122   Arguments:
123     service: service object
```

```
124         stackname: Name of stack if only one stack should be viewed
125     """
126     table_data = [['Stackname', 'host', 'image', 'conatiners']]
127     for stack in service.stacks:
128         table_data.append([str(stack.name), str(stack.host), str(stack.image), "\n".join([c.
            name for c in stack.container])])
129 
130     print table_data
131     table = AsciiTable(table_data)
132     table.inner_row_border = True
133     print table.table
134 
135 
136 # Endpoint functions
137 def make_endpoint(service, name, publicport, stackpointer=None):
138     """Create and connect a endpoint to a service
139     Arguments:
140         name: additional name: "service-endpoint-%s"
141         service: service object
142         publicport: public port
143         stackpointer: default place on stack
144     Constraints:
145     """
146     e = Endpoint(name, service, publicport, stackpointer)
147     return e
148 
149 def get_endpoint(name):
150     """Finds an endpoint based on name"""
151     return session.query(Endpoint).filter(Endpoint.name.like(name)).first()
152 
153 def view_endpoint(endpointname, obj=None):
154     """Prints out a single endpoint"""
155     table_data = [['Endpoint name', 'ip', 'pubport', 'url', 'mainservice', 'stackpointer', 'tree'
            ]]
156     if not obj:
157         endpoint = session.query(Endpoint).filter(Endpoint.name.like(endpointname)).first()
158     else:
159         endpoint = obj
160     if endpoint:
161         print endpoint.get_state()
162         subtree = view_endpoint_tree(endpoint)
163         table_data.append([str(endpoint.name), str(endpoint.ip), str(endpoint.pubport), str(
            endpoint.url), str(endpoint.service.name), str(endpoint.stackpointer), subtree])
164         tree = AsciiTable(table_data)
165         tree.inner_row_border = True
166         print tree.table
167     else:
168         print "Endpoint not found"
169 
170 def view_endpoint_tree(endpoint):
171     subtree_data = [['parent', 'child', 'stackposition']]
172     for tree in endpoint.service_tree:
173         state = tree.get_state()
174         subtree_data.append([str(state['parent']), str(state['child']), str(state['stackpos'])])
175     subtree = AsciiTable(subtree_data)
176     return subtree.table
177 
178 
179 def view_endpoints():
```

```
180     """Lists the endpoints defined
181     Arguments:
182       *sort by service
183       *sort by stage
184     """
185     table_data = [['Endpoint name', 'ip', 'pubport', 'url', 'mainservice', 'stackpointer', 'tree'
          ]]
186     for endpoint in session.query(Endpoint).all():
187       subtree = view_endpoint_tree(endpoint)
188       table_data.append([str(endpoint.name), str(endpoint.ip), str(endpoint.pubport), str(
          endpoint.url), str(endpoint.service.name), str(endpoint.stackpointer), subtree])
189     table = AsciiTable(table_data)
190     table.inner_row_border = True
191     print table.table
192
193
194 def switch_stackpointer(endpoint, service, newpointer):
195     """Switch the pointer of an endpoint"""
196     tree = session.query(Service_tree).filter(Service_tree.child == service,
197                               Service_tree.endpoint == endpoint).first()
198     tree.stackpos = newpointer
199     session.commit()
200
201
202 # Container functions
203 def create_containers(stack, versions):
204     """Initial function that creates containers
205     Enforces the state rules"""
206     containers = []
207     if not check_versions(stack, versions):
208       pass
209     if type(versions) == list:
210       if len(versions) < rules.getint('stack', 'min_containers'):
211         diff = rules.getint('stack', 'min_containers') - len(versions)
212         # Pushes the different versions on the stack
213         for version in versions:
214           containers.append(push_on_stack(stack, version))
215         # Padds with N versions to fill up stack according to constraint
216         for i in range(0, diff):
217           containers.append(push_on_stack(stack, versions[-1]))
218       else:
219         for version in versions:
220           containers.append(push_on_stack(stack, version))
221     else:
222       for i in range(0, rules.getint('stack', 'min_containers')):
223         containers.append(push_on_stack(stack, versions))
224
225     # Check if image exists on host
226     for container in containers:
227       if not docker.image_exists(container.stack.host, container.get_version()):
228         response = docker.pull_image(stack.host, stack.image, container.version)
229         if 'not found' in response:
230           print "Image not found. Rolling back containers"
231           session.rollback()
232           raise StandardError('Image not found. Reverting')
233         else:
234           print "Image %s:%s downloaded on %s" % \
235             (stack.image, container.version, stack.host)
236
```

```
    for container in containers:
        session.add(container)

    session.commit()

    for container in containers:
        container.deploy_container()
    session.commit()


def check_versions(stack, versions):
    vconf = rules.get('stack', 'versions').split(',')
    if not stack:
        v = versions
    elif type(versions) == list:
        existing = stack.get_versions()
        v = existing + versions
    else:
        existing = stack.get_versions()
        v = existing + [versions]

    if len(v) == 1:
        return True
    ok = False
    for i, ver in enumerate(v):
        if i == 0:
            continue
        if 'incremental' in vconf and 'equal' in vconf:
            if 'latest' in ver or ver >= v[i-1]:
                ok = True
            else:
                ok = False
        elif 'incremental' in vconf and not ver > v[i-1]:
            return False
        elif 'equal' in vconf and not ver == v[i-1]:
            return False
        elif 'all':
            return True
    return ok


def push_on_stack(stack, version):
    """Push new version of a container on stack
    Arguments:
        stack: stack object
        version: new version of image
    Returns:
        the new container
    """
    c = Container(stack, version)
    return c


def pop(service, stack, position=0):
    """Removes a container in the specific position
    Arguments:
        service: service object
        stack: specific stack or None for all stacks
        position: position on stack to pop
```

```
296     """
297     if not stack:
298         for stack in service.stacks:
299             if len(stack.container) <= rules.getint('stack', 'min_containers'):
300                 raise StandardError('Not enough containers on stack. '
301                             'Poping not compliant with rules')
302     else:
303         if len(stack.container) <= rules.getint('stack', 'min_containers'):
304             raise StandardError('Not enough containers on specified stack. '
305                         'Not compliant to pop')
306     # HERE NEEDS CONSTRAINTS CHECKING
307     # HERE NEEDS HAPROXY INTEGRATION
308 
309     if not stack:
310         for s in service.stacks:
311             c = s.containers.pop(position)
312             # HERE NEEDS DOCKER/HAPROXY
313             c.remove_container()
314             session.delete(c)
315     else:
316         c = stack.container[position]
317         c.remove_container()
318         session.delete(c)
319 
320     session.commit()
321 
322 def replace(service, position, direction, stack=None):
323     """Replaces a container on a stack, with neighbour
324     Arguments:
325         service: service object
326         position: position of stack to replace (arraypos 0-n)
327         direction: which of the surrounding containers to copy
328     kwargs:
329         stack: name of stack, if only one stack
330     """
331     raise NotImplementedError()
332     # find the new version
333     containers = service.stacks[0].containers
334     version = containers[position+direction].version
335     # Create a new container that is not related to a stack yet
336     c = Container(None, version, image=service.stacks[0].image)
337     #for stack in service.stacks:
338         ## find the version
339     #if stack:
340         ##pass
341 
342 
343 def view_containers():
344     services = session.query(Service).all()
345     table = []
346     table.append(['Name', 'image', 'version', 'port', 'containerid'])
347     for service in services:
348         for stack in service.stacks:
349             for container in stack.container:
350                 st = container.get_state()
351                 table.append([str(st['name']), str(st['image']),
352                             str(st['version']), str(st['port']),
353                             str(st['containerid'])[0:15]])
354 
```

```
355     t = AsciiTable(table)
356     t.inner_row_border = True
357     print t.table
358
359
360 def remove_all_containers():
361     for container in session.query(Container).all():
362         try:
363             print "Removing container %s from %s" % \
364                 (container.name, container.stack.host)
365             container.remove_container()
366         except APIError as e:
367             print e.message
368             if 'Not Found' in e.message:
369                 print "Container %s not found on host %s" % \
370                     (container.name, container.stack.host)
371     session.commit()
372
373
374 def deploy_all_containers():
375     for container in session.query(Container).all():
376         try:
377             container.deploy_container()
378         except APIError as e:
379             if '409 Client Error: Conflict' in e.message:
380                 print "Container %s allready running on %s" % \
381                     (container.name, container.stack.host)
382             else:
383                 print "Error on container %s: %s" % (container.name, str(e))
384     session.commit()
385
386
387 # Tree operations
388 def create_tree(endpoint, relations):
389     """Create tree from list of named dicts
390     Arguments:
391         endpoint: object of the endpoint or name
392         relations: string or list of dict: {parent, child, stackref}
393     """
394     e = None
395     if endpoint == Endpoint:
396         e = endpoint
397     else:
398         e = session.query(Endpoint).filter(Endpoint.name == endpoint).first()
399
400     if type(relations) == str:
401         relations = eval(relations)
402     for item in relations:
403         tree_node = Service_tree(item['parent'], item['child'], e,
404                                  next_pointerport())
405         session.add(tree_node)
406     session.commit()
407
408
409 def add_relation(endpoint, service, parents, childs, stackpointer):
410     """Create a new relation in a service tree based on service
411     Arguments:
412         endpoint: Tree to connect to
413         service: The service to connect
```

```
414         parents: the parent services of the service
415         childs: the child of the service
416         stackpointer: optional stack pointer"""
417     # Handle parents
418     trees = []
419     for parent in parents:
420         trees.append(Service_tree(parent, service, endpoint,
421                             next_pointerport(), stackpointer))
422     for child in childs:
423         trees.append(Service_tree(service, child, endpoint,
424                             next_pointerport(), stackpointer))
425 
426     session.add_all(trees)
427     session.commit()
428     # HAndle childs
429 
430 
431 def next_pointerport():
432     curpointerport = session.query(func.max(Service_tree.port)).scalar()
433     if curpointerport:
434         return curpointerport + 1
435     else:
436         return config.getint('service', 'portstart')
437 
438 def view_tree(service):
439     """Shows a tree"""
440     raise NotImplementedError()
441 
442 
443 # State
444 def get_state():
445     """Gets the state of the complete environment"""
446     state = {}
447     for service in session.query(Service).all():
448         state[service.name] = service.get_state()
449     return state
450 
451 def get_service_state(service=None, servicename=None):
452     """Gets the state of service and all of its relations"""
453     if service:
454         return service.get_state()
455     elif servicename:
456         s = session.query(Service).filter(Service.name == servicename).first()
457         return s.get_state()
458     else:
459         return None
460 
461 def get_endpoint_state(service=None, endpointname=None):
462     """Gets the state of an endpoint"""
463     if service:
464         return service.endpoint.get_state()
465     elif endpointname:
466         e = session.query(Endpoint).filter(Endpoint.name == endpointname).first()
467         return e.get_state()
468     else:
469         return None
470 
471 def get_stack_state(stack=None, stackname=None):
472     """Gets the state of a stack"""
```

```
473     if stack:
474         return stack.get_state()
475     elif stackname:
476         stack = session.query(Stack).filter(Stack.name == stackname).first()
477         return stack.get_state()
478     else:
479         return None
480 
481 def get_container_state(container=None, containername=None):
482     """Gets the state of a container"""
483     if container:
484         return container.get_state()
485     elif containername:
486         c = session.query(Container).filter(Container.name == containername).first()
487         return c.get_state()
488     else:
489         return None
490 
491 if __name__ == '__main__':
492     view_service('webapp1')
493     #view_services()
494     #s = get_service('WebFront')
495     #view_stack(s)
496 
497     #view_endpoint('alabi')
498     #view_endpoints()
499     #s = create_service('WebFront', 20202)
500     #print s
501     #print s.name
```

Listing A.6: lib/init.py: Initializer of database connection

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3  
4  from sqlalchemy.orm import sessionmaker
5  from sqlalchemy import create_engine
6  from os import path
7  
8  import base
9  import config as cfg
10 from service import Service
11 from stack import Stack
12 from container import Container
13 from endpoint import Endpoint
14 from stage import Stage
15 
16 logging = cfg.get_logger()
17 config = cfg.get_config()
18 
19 def init(create=False):
20     Base = base.Base
21 
22     engine = create_engine(config.get('database','engine'),
23                            echo=config.getboolean('database','echo'))
24 
25     Session = sessionmaker(bind=engine)
26     session = Session()
```

```
27    if create:
28        Base.metadata.create_all(engine)
29    return session
30
31
32  def base_test_data(session, servicename,pubport):
33    s = Service(servicename)
34    for i in range(1,3):
35        webstack = Stack(s, 'image')
36        session.add(s)
37        session.add(webstack)
38        for i in range(1, 10):
39            c = Container(webstack, 'v0.%s' % i)
40            session.add(c)
41        session.commit()
42
43    session.commit()
44
45    s = session.query(Service).filter(Service.name == servicename).first()
46    e = Endpoint('web', s, pubport)
47    session.add(e)
48    session.commit()
49
50
51  def get_state(session, service):
52    s = session.query(Service).filter(Service.name == service).first()
53    print s
54    if s:
55        endpoint = s.endpoints
56        print endpoint
57        print s.name
58        for stack in s.stacks:
59            for container in stack.container:
60                print container.name
61
62
63  def main():
64    session = init(create=True)
65    base_test_data(session, 'webapp1', 1010)
66    base_test_data(session, 'webapp2', 1020)
67
68    get_state(session, 'webapp1')
69    get_state(session, 'webapp2')
70
71  if __name__ == '__main__':
72    main()
```

Listing A.7: lib/base.py: Singleton for database connection

```
1  #!/usr/bin/env python
2  # −∗− coding: utf−8 −∗−
3
4  from sqlalchemy.ext.declarative import declarative_base
5  Base = declarative_base()
```

Listing A.8: lib/service.py: Object class of Services and Service trees

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

```
2  # −*− coding: utf−8 −*−
3  from sqlalchemy import Column, Integer, String, ForeignKey
4  from sqlalchemy.orm import relationship, backref
5  from exceptions import TypeError
6
7  import config as cfg
8  containerconfig = cfg.get_container_config()
9
10 import base
11 Base = base.Base
12
13
14 # http://docs.sqlalchemy.org/en/latest/orm/basic_relationships.html#association−
       pattern
15 # http://stackoverflow.com/questions/25958963/self−referential−association−
       relationship−sqlalchemy
16 class Service_tree(Base):
17     __tablename__ = 'service_tree'
18     parent_id = Column(Integer, ForeignKey('service.id'), primary_key=True)
19     child_id = Column(Integer, ForeignKey('service.id'), primary_key=True)
20     endpoint_id = Column(Integer, ForeignKey('endpoint.id'), primary_key=True)
21     stackpos = Column(Integer)
22     endpoint = relationship('Endpoint', backref=backref('service_tree'))
23     port = Column(Integer, unique=True)
24
25     def __init__(self, parent, child, endpoint, port, stackpos=None):
26         self.parent = parent
27         self.child = child
28         self.endpoint = endpoint
29         if stackpos >= 0:
30             self.stackpos = stackpos
31         elif endpoint.stackpointer >= 0:
32             self.stackpos = endpoint.stackpointer
33         elif not stackpos and not endpoint.stackpointer:
34             raise TypeError('stackpos not set')
35         self.port = port
36
37     def get_state(self):
38         state = {}
39         state['parent'] = self.parent.name
40         state['child'] = self.child.name
41         state['endpoint'] = self.endpoint.name
42         state['stackpos'] = self.stackpos
43         return state
44
45
46 class Service(Base):
47     __tablename__ = 'service'
48
49     id = Column(Integer, primary_key=True)
50     name = Column(String, nullable=False, unique=True)
51     parents = relationship('Service_tree', backref='child',
52                            primaryjoin=id == Service_tree.child_id)
53     childs = relationship('Service_tree', backref='parent',
54                           primaryjoin=id == Service_tree.parent_id)
55     stacks = relationship('Stack', cascade="delete",
56                           backref=backref('service', order_by=id))
57     endpoints = relationship('Endpoint', backref=backref('service'))
58
```

```
59    def __init__(self, name):
60        self.name = name
61
62    def get_state(self):
63        state = {}
64        state['name'] = self.name
65        state['stacks'] = []
66        for stack in self.stacks:
67            state['stacks'].append(stack.get_state())
68        state['parent'] = [parent.get_state() for parent in self.parents]
69        state['childs'] = [child.get_state() for child in self.childs]
70        state['endpoints'] = [e.name for e in self.endpoints]
71        return state
72
73    def choose_host(self):
74        """Finds out which server a stack should run on based on current stacks
75        Simple implementation. Implement on first host first
76        """
77        hosts = containerconfig.get('main', 'hostnames').split(',')
78        if self.stacks:
79            # if even
80            if len(self.stacks) % 2 == 0:
81                return hosts[0]
82            # if odd
83            elif len(self.stacks) % 2 == 1:
84                return hosts[1]
85        else:
86            # No existing stacks
87            return hosts[0]
88
89    def tree_on_stack_pointer(self, stackpointer):
90        for parent in self.parents:
91            if parent.stackpos == stackpointer:
92                return parent.endpoint
93        return None
94
95 if __name__ == '__main__':
96    s = Service("hest")
97    print type(s)
98    s.name = "Hest"
99    print s.name
```

Listing A.9: lib/stack.py: Object of the Stack

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3  from sqlalchemy import Column, Integer, String, ForeignKey
4  from sqlalchemy.orm import relationship, backref
5
6  import base
7  Base = base.Base
8
9
10 class Stack(Base):
11    __tablename__ = 'stack'
12    id = Column(Integer, primary_key=True)
13    name = Column(String)
14    host = Column(String)
```

```
    image = Column(String)
    service_id = Column(Integer, ForeignKey('service.id'))
    container = relationship('Container', backref=backref('stack'),
                             order_by='Container.id')

    def __init__(self, service, image, host=None):
        """Creates a stack from a service
        Arguments:
            service: Parent service object
            host: Machine the stack exists on
        Sets:
            name: service name and len of stack
        """
        self.service = service
        self.name = '%s-stack%s' % (self.service.name, len(self.service.stacks))
        self.host = host
        self.image = image

    def get_state(self):
        state = {}
        state['name'] = self.name
        state['container'] = []
        for container in self.container:
            state['container'].append(container.get_state())
        return state

    def get_versions(self):
        versions = []
        for container in self.container:
            versions.append(container.version)
        return versions
```

Listing A.10: lib/container.py: Container object

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
from sqlalchemy import Column, Integer, String, ForeignKey
from sqlalchemy.orm import relationship, backref
from exceptions import LookupError
from random import getrandbits
from time import sleep

import base
Base = base.Base

import lib_docker as docker


class Container(Base):
    """ Class for containers """
    __tablename__ = 'container'

    id = Column(Integer, primary_key=True)
    name = Column(String)    # The name the container will have
    version = Column(String)  # 'version'
    image = Column(String)    # 'image' without version
    # source_image = Column(String)
    # ip = Column(String)  # The local ip
```

```
25    port = Column(String)  # The local port
26    containerid = Column(String)
27    stack_id = Column(Integer, ForeignKey('stack.id'))
28    #stage_id = Column(Integer, ForeignKey('stage.id'))
29    #stage = relationship('Stage', backref=backref('container'))
30
31    def __init__(self, stack, version, image=None, name='app'):
32       """Creates a new container object
33       Arguments:
34          stack: stack object of parent stack
35          version: version of image
36
37       Keyword arguments:
38          image: name of container image (when None: from stack): LookupError
39       Sets:
40          name: based on stackname + app + nr, if set (chagnes app)
41       """
42       if stack:
43          self.stack = stack
44          self.name = '%s-%s%s_0x%s' % (self.stack.name,
45                                name,
46                                len(self.stack.container), getrandbits(30))
47       else:
48          self.name = '%s_0x%s' % (name, getrandbits(30))
49       if not image:
50          if stack and stack.image:
51             self.image = stack.image
52          else:
53             raise LookupError('No image defined in stack or init')
54       else:
55          self.image = image
56       self.version = version
57
58
59    def __repr__(self):
60       return self.name
61
62    def get_version(self):
63       return "{}:{}".format(self.image, self.version)
64
65    def get_state(self):
66       state = {}
67       state['name'] = self.name
68       state['version'] = self.version
69       state['image'] = self.image
70       # state['source_image'] = self.source_image
71       # state['ip'] = self.ip
72       state['port'] = self.port
73       state['containerid'] = self.containerid
74       #state['stage'] = self.stage
75       return state
76
77    def deploy_container(self, output=True):
78       check = docker.image_exists(self.stack.host, self.get_version())
79       if not check:
80          docker.pull_image(self.stack.host, self.image, self.version)
81
82       c = docker.create_container(self.stack.host, self)
83       self.containerid = c['Id']
```

```
84      # Handle ports here
85      docker.start_container(self.stack.host, c['Id'])
86      info = None
87      counter = 5
88      while not info and counter > 0:
89          info = docker.get_container(self.stack.host, self.containerid)
90          if output:
91              if info:
92                  print "Container %s running on %s, port %s" % (self.name,
93                                                  self.stack.host,
94                                                  info['Ports'][0]['PublicPort'])
95              else:
96                  print "Waiting for container to be listed %s" % (str(counter))
97                  sleep(1)
98
99          counter -= 1
100
101     self.port = str(info['Ports'][0]['PublicPort'])
102
103 def remove_container(self):
104     if self.containerid:
105         id = self.containerid
106         docker.stop_container(self.stack.host, id)
107     else:
108         id = self.name
109     docker.stop_container(self.stack.host, id)
110     docker.remove_container_byid(self.stack.host, id)
111     self.containerid = None
112     self.port = None
113
114 if __name__ == '__main__':
115     c = container()
```

Listing A.11: lib/endpoint.py: Endpoint object

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  from sqlalchemy import Table, Column, Integer, String, MetaData,ForeignKey, text
5  from sqlalchemy.orm import relationship, backref
6
7  import base
8  Base = base.Base
9
10
11 class Endpoint(Base):
12     __tablename__ = 'endpoint'
13
14     id = Column(Integer, primary_key = True)
15     name = Column(String, unique=True)
16     ip = Column(String)
17     pubport = Column(Integer, unique=True, nullable=False)
18     url = Column(String)
19     stackpointer = Column(Integer)#, nullable=False)  # Inteneded as default placement on
           stack
20     service_id = Column(Integer, ForeignKey('service.id'), nullable=False)
21     stage = Column(String)
22     #stage_id = Column(Integer, ForeignKey('stage.id'))
```

```
23    #stage = relationship('Stage', backref=backref('endpoint'))
24
25    def __init__(self, name, service, pubport, stackpointer=None, stage=None):
26        self.name = "%s-endpoint-%s" % (service.name, name)
27        self.service = service
28        self.pubport = pubport
29        self.stackpointer = stackpointer
30        self.stage = stage
31
32    def get_state(self):
33        state = {}
34        state['name'] = self.name
35        state['ip'] = self.ip
36        state['pubport'] = self.pubport
37        state['url'] = self.url
38        state['stage'] = self.stage
39        state['service'] = self.service.name
40        state['stackpointer'] = self.stackpointer
41        return state
42
43
44 if __name__ == '__main__':
45    e = endpoint("hest")
46    print type(e)
47    e.name = "Hest"
48    print e.name
```

Listing A.12: lib/config.py: Config parsing

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  import ConfigParser
5  import logging, logging.config
6  from os import path
7  from sys import exit
8
9  basepath = path.split(path.dirname(path.abspath(__file__)))[0]
10 configpath = path.join(basepath, 'etc/config.conf')
11 logconfigpath = path.join(basepath, 'etc/logging.conf')
12 constraintpath = path.join(basepath, 'etc/rules.conf')
13
14 if path.isfile(configpath):
15    config = ConfigParser.ConfigParser()
16    config.read(configpath)
17 else:
18    logging.error("Missing configuration file %s" % configpath)
19    exit(1)
20
21 if path.isfile(logconfigpath):
22    logging.config.fileConfig(logconfigpath)
23    logger = logging.getLogger('main')
24 else:
25    logging.error("Missing logger configuration file %s" % logconfigpath)
26    exit(1)
27
28 containerpath = path.join(basepath, 'etc/%s.conf' % config.get('main', 'containerbackend')
       )
```

```
29
30 if path.isfile(containerpath):
31     containerconfig = ConfigParser.ConfigParser()
32     containerconfig.read(containerpath)
33
34
35 if path.isfile(constraintpath):
36     rules = ConfigParser.ConfigParser()
37     rules.read(constraintpath)
38 else:
39     logging.error("Missing rules configuration file")
40     exit(1)
41
42 def get_config():
43     return config
44
45 def get_logger():
46     return logging
47
48 def get_container_config():
49     return containerconfig
50
51 def get_rules_config():
52     return rules
```

Listing A.13: lib/lib_docker.py: Library for Docker functionality

```
1  #!/usr/bin/env python
2  # −*− coding: utf−8 −*−
3
4  from docker import Client
5  from requests import ConnectionError
6  import config as cfg
7
8  config = cfg.get_config()
9  cconfig = cfg.get_container_config()
10
11 connections = {}
12
13 hostnames = cconfig.get('main', 'hostnames').split(',')
14 hosts = cconfig.get('main', 'hosts').split(',')
15 port = cconfig.get('main', 'port')
16
17
18 def connect(host='localhost', base_url='unix://var/run/docker.sock', tls=False):
19     if host not in connections.keys():
20         try:
21             c = Client(base_url=base_url, tls=tls)
22             connections[host] = c
23         except ConnectionError:
24             return None
25
26
27 def get_containers(host):
28     """Returns the docker containers
29     Arguments:
30         host: hostname of host
31     """
```

```
32    return connections[host].containers()
33
34
35  def get_container(host, containerid):
36    """Get info about a specifc container
37    Arguments:
38      hostname
39      containerid: hash identifier
40    """
41    containers = get_containers(host)
42    for container in containers:
43      if containerid in container['Id']:
44        return container
45
46
47  def pull_image(host, image, version):
48    """Pulls the image and version to the host
49    Arguments:
50      host: hostname of host
51      image: image name
52      version: image version
53    returns the status {'status','progressDetail', 'id'}
54    """
55    status = connections[host].pull('%s:%s' % (image, version))
56    return status
57
58
59  def create_container(host, container):
60    """Create a container (needs to be started)
61    Arguments:
62      host: hostname of host to start on
63      container: container object
64    Returns:
65      dict: {Id: hash, Warnings: None}
66    """
67    # hostname, volumes, detached, ports=[1234,134]
68    # image: 'name:tag'
69    c = connections[host].create_container(image=container.get_version(),
70                                           #command=container.cmd,
71                                           name=container.name)
72    return c
73
74
75  def start_container(host, idorname):
76    """Start existing container
77    Arguments:
78      host: hostname the container is on
79      idorname: container identifyer
80    """
81    # port_bindings={1111: ('127.0.0.1', 4567)},
82    response = connections[host].start(container=idorname,
83                                       publish_all_ports=True,
84                                       restart_policy={'Name': 'always'}
85                                       )
86    return response
87
88
89  def remove_container(host, name):
90    """Removes a container from a host
```

```
91      Arguments:
92          host: host to remove container from
93          container: container object to remove
94      """
95      return connections[host].remove_container(container=name,
96                                                force=True)
97
98
99  def image_exists(host, imagetag):
100     """Checks if image is on host
101     Arguments:
102         host: hostname
103         imagetag: name and optional tag of image
104     Returns:
105         true: image exists
106         false: does not exist
107     """
108     images = connections[host].images()
109     upstreamtag = "docker.io/%s" % imagetag
110     for image in images:
111         if imagetag in image['RepoTags']:
112             return True
113         elif upstreamtag in image['RepoTags']:
114             return True
115     return False
116
117
118 def container_exists(host, name, all=False):
119     """Check if a container with the same name is running on the host
120     """
121     for container in connections[host].containers(all=all):
122         if name in container['Names']:
123             return True
124     return False
125
126
127 def stop_container(host, id):
128     """Stop container"""
129     connections[host].stop(resource_id=id, timeout=1)
130
131
132 def remove_container_byid(host, id):
133     """Remove container"""
134     connections[host].remove_container(id)
135
136
137 def init():
138     for i in range(0,len(hostnames)):
139         connect(host=hostnames[i], base_url='http://%s:%s' % (hosts[i], str(port)))
140
141 init()
```

Listing A.14: hap.py: Config builder of state for HAProxy integration

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  # Integration to HAProxy
```

```
5  import basefunc
6  from lib.service import Service, Service_tree
7  from lib.endpoint import Endpoint
8  from haproxy.haproxy import HAproxy
9  from lib import config as cfg
10 containerconfig = cfg.get_container_config()
11 containerhostnames = containerconfig.get('main', 'hostnames').split(',')
12 containerhosts = containerconfig.get('main', 'hosts').split(',')
13
14 hostdict = {}
15 for i, item in enumerate(containerhostnames):
16     hostdict[item] = containerhosts[i]
17
18 session = basefunc.session
19
20
21 def build_containerdict(containers):
22     containerlist = []
23     for c in containers:
24         tmpc = c.get_state()
25         tmpc['hostip'] = hostdict[c.stack.host]
26         containerlist.append(tmpc)
27     return containerlist
28
29
30 def build_trees():
31     every = {}
32     leafs = session.query(Service_tree).all()
33     for leaf in leafs:
34         every[leaf.child.name] = {'endpoint': leaf.endpoint.name,
35                                   'port': leaf.port}
36         containers = []
37         for stack in leaf.child.stacks:
38             containers.append(stack.container[leaf.stackpos])
39
40         every[leaf.child.name]['containers'] = build_containerdict(stack.container)
41
42     return every
43
44
45 def rebuild_hap_config():
46     hap = HAproxy()
47     tree = build_trees()
48     # The new config file is written
49     # BUT: Needs a restart + handling of discrepanies
50     hap.compile(tree)
51
52 if __name__ == '__main__':
53     hap = HAproxy()
54     tree = build_trees()
55     #print tree
56     hap.compile(tree)
```

A.2 The webapp

The following code relates to the application written to test out the deployment with the new prototype. This is a simple python web application using the framework flask. The code is also available on github: https://github.com/larhauga/webapp The github repository is then pulled into the automated build repository at Docker Hub: https://registry.hub.docker.com/u/larhauga/webapp/

Listing A.15: webapp.py: Simple website

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import ConfigParser
import requests
import argparse
from flask import Flask
app = Flask(__name__)

config = ConfigParser.ConfigParser()
config.read('etc/app.cfg')


def get_version():
    version = None
    with open('etc/version.txt', 'r') as f:
        version = f.read().strip().split()
    return version


@app.route("/")
def front():
    version = get_version()

    return "Service %s. Version %s" % (version[0], version[1])


@app.route('/api')
def depend():
    txt = "Service %s: %s<br />" % (str(get_version()[0]),
                                    str(get_version()[1]))
    txt += "<b>Sub services:</b><br />"
    for service in config.get('main', 'service').split(','):
        try:
            r = requests.get(config.get(service, 'url'), timeout=1)
            txt += r.text
        except:
            txt += "No connection to configured subservice"
            print "Something went wrong"
        txt += "<br />"
    return txt


@app.route('/health')
def healthcheck():
    return "Up and ready"
```

```
47
48 if __name__ == '__main__':
49     parser = argparse.ArgumentParser()
50     parser.add_argument('-p', '--port', type=int, required=True)
51     args = parser.parse_args()
52     app.debug = config.get('main', 'debug')
53     app.run(host='0.0.0.0', port=args.port)
```

A.3 HAProxy Config updater

The HAProxy config updater is a Python application that were built in part during a previous course, and modified to meet the requirements of this thesis, where it enables the integration of HAProxy. This enables the generation of configuration based on the state, and dynamic replicable configuration is achieved. The prototype uses this through the integration with the file hap.py A.14.

Since this is not created in hole during this thesis, and implements only a small part, it is not included here.

The code can be found on github:
https://github.com/larhauga/haproxy_config_updater