

Oleg Mironov

DevOps Pipeline with Docker

Helsinki Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Thesis

15 April 2018

Author(s) Title	Oleg Mironov DevOps Pipeline with Docker
Number of Pages Date	55 pages + 11 appendices 2 December 2017
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Software Engineering
Instructor(s)	Marko Klemetti, CTO Erik Pätynen, Senior Lecturer
<p>Software complexity and size are growing at the ever-increasing rate. This study attempts to use modern DevOps practices and technologies to create a reusable pipeline template capable of meeting the demands of modern software and strengthen up the development practices. Such pipeline should largely be reusable and flexible as well as being able to be scaled and maintained with high reliability.</p> <p>Theoretical research on the topic of how DevOps came to be and what it means is an important part of this project. Then a set of technologies that reflect the state of DevOps today was carefully studied and analysed. Docker and its fast-growing ecosystem is the core technology of this project. With Docker and some other technologies, a set of configuration files could be used to run, develop, test and deploy an application. Such approach allows maximum automation while ensuring transparency of those automation processes.</p> <p>The result of this project is a fully working pipeline setup that is fully automated and is able to support a fast-paced software development. The pipeline is built against a reference project. Most of the pipeline is configured with a set of different configuration files meaning that from a fresh start it could be brought up with minimal human interaction. It covers all parts of a reference application lifespan from a development environment to a potential production deployment. There is a set of technologies used in the pipeline such as Jenkins, Docker and container orchestration with Rancher.</p>	
Keywords	Docker, Jenkins, Rancher, DevOps, Pipeline, Automation, Continuous Integration, Continuous Delivery

Contents

1	Introduction	1
2	Theoretical Background	3
2.1	Continuous Integration	3
2.2	Continuous Delivery and Continuous Deployment	5
2.3	DevOps	7
2.4	Virtualization	10
2.5	Docker	12
2.6	Container Orchestration	15
2.7	Container Management Software	17
3	Project Methods and Materials	19
3.1	Project Workflow and Methods	19
3.2	Requirements	19
3.3	Tools and Technologies	20
4	Pipeline Design and Implementation	22
4.1	Reference Project Description	22
4.2	Pipeline Flow	23
4.3	Implementation Plan	26
4.4	Dockerizing Voting App	27
4.5	Local Container Orchestration	30
4.6	Virtual Machines with Ansible and Vagrant	34
4.7	Configuring Rancher Environment	36
5	Results and Discussion	49
5.1	Summary of Results	49
5.2	Evaluation of Results	50
5.3	Project Challenges	52
6	Conclusions	55
	References	56
	Appendix 1. GitHub Web Links	60

Appendix 2. Voting App Interface Screenshots	61
Appendix 3. Rancher docker-compose.dev.yml.	63
Appendix 4. Rancher rancher-compose.dev.yml	65
Appendix 5. Rancher load balancer UI.	67
Appendix 6. Add Backend Webhook.	68
Appendix 7. Unlock Jenkins.	69
Appendix 8. Select Jenkins Plugins.	70
Appendix 9. Selecting Jenkins job type.	71
Appendix 10. Adding DockerHub credentials to Jenkins.	72
Appendix 11. Final Jenkinsfile.	73

Abbreviations

CD	Continuous Delivery
CDt	Continuous Deployment
CI	Continuous Integration
DevOps	Development Operations
OS	Operating System
QA	Quality Assurance
VM	Virtual Machine
UI	User Interface

1 Introduction

Throughout its lifetime any application would likely be run in many environments, such as local, development and production. Today, there are a lot of different tools that are claimed to optimise and simplify all stages of deployment and, especially, a production deployment. With the recent introduction of Docker engine and an ever-growing ecosystem around it, it is becoming increasingly important to find out a universal deployment pattern that could be reliably used for modern web projects.

However, one will need to go more in the past to better understand the grounds for modern accepted DevOps practices. The DevOps as a culture and concept will be described in detail in later sections. Then, continuous integration is a concept that has been around from 1991 when it was first introduced by Grady Booch [1] and which is at the cornerstone of this thesis. Combined with newer concept of Continuous Delivery, which was first acknowledged in 2011 after the publication of “Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation” by Jez Humble and David Farley [2], the grounds for building modern software pipeline are being formed.

Then Docker has been introduced in March 2013 [3]. Since then, the adoption of the technology has reached a pace that is uncommon for operations part of the industry. May 2015 to May 2016 adoption of Docker has seen an increase of 30% that mostly came from large enterprises [4]. Docker and the ecosystem around it bring significant benefits to the users. These benefits will be discussed in further chapters. However, being such a new trend means documentation is sparse and good practices are not so immutable compared to more matured IT technologies and methodologies.

Such immaturity can result into using this, as it would first seem, beneficial technology to cause more harm than good. Environments powered by Docker can become less secure, harder to maintain and less reliable with lower uptime. As with any new technology caution should be applied and new solid practices should be developed.

The main goal of this thesis is to deeply study and analyse the booming Docker ecosystem, combine it with established Continuous Integration and Continuous Delivery paradigms and come up with those very practices that could be safely used in software pipelines to achieve solid efficiency, reliability and flexibility. The output of this project

should be laid in a form of documentation, configuration files and scripts that together would form a template for production ready environments from scratch for a typical web project.

In the next section the theoretical background of the topic will be covered. Above mentioned technologies and patterns are of the main focus. Then, the third section is mainly targeted to tools and techniques used to achieve the goal of a stable environment template.

Section 4 presents the result of work - this very environment setup, all the methodologies and recommendation developed during this research. Following next the discussion section will cover main use cases, advantages and disadvantages of developed template. Finally, the conclusion section takes a challenge of summarising, analysing and finishing this thesis.

2 Theoretical Background

2.1 Continuous Integration

Ever since the introduction of first computers and software, the development of those was becoming a larger and larger process. Size and complexity of computer software was growing leading to at least larger development teams and greater dependencies. As it is seen from a Visual Capitalist graph, Windows NT 3.1 released in 1993 had about 4.5 million lines of code, whilst the latest 2017 Firefox has about 10 million lines of code [5]. Even browsers of today are more complex than desktop operating systems of 25 years ago. Then, 2009 Windows 7 is estimated to 40 million lines - a growth of almost 9 times in just two and a half decades.

Such a rapid growth of complexity could not go unnoticed. Prior to Continuous Integration (or CI) introduction, applications were built and integrated together at random moments, often by the end of development cycles. This led to a variety of problems such as code conflicts and dependency errors. Vivek Ganesan writes in his 2017 book that back in times when Waterfall was the main software development methodology there was a designated integration stage, where software written in different parts by many of developers would be merged together [6]. Due to software design inconsistencies and miscommunication it was common that integration would not go smoothly and cause bugs and critical malfunctions, further adding to complexity and extending development cycle.

CI introduction in 1991 is one of the first attempts to address the issue of integration during a lifespan of the project. Continuous Integration is one of the initial 12 practices of extreme programming [7]. In different use cases and organizations practices behind CI may vary but it will still mainly consist of:

- Committing code changes to main repository at least once a day by every contributor.
- Those changes usually are first verified by each team member in local development environment.
- When pushed to code repository, new changes are verified by an automated build that should contain tests.
- If any change causes a build or tests fail, then this should be immediately addressed and fixed.
- Often today, if the build was successful new code will be deployed to staging

environment, which is a clone of production one.

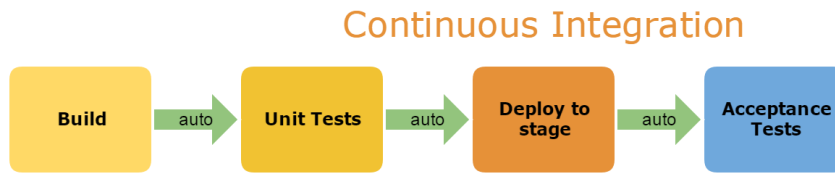


Figure 1. Continuous Integration Flow. Modified from Pecanac [8].

The figure 1 above displays a typical flow of CI. Upon code being committed, the software is usually built, then some local tests such as unit tests are run. If build and tests are had been run successfully, then deployment to staging or test environment is performed following by acceptance tests to verify that application including possible third-party integrations is acting as expected. There are many different variations of this flow, but the main idea would always be the same - integrate, build and test software.

Figure 1 also shows “auto” in between stages, which means that those stages should run automatically without human intervention. However, nothing becomes automated by itself. As CI has been around for a few decades, there are now dozens of automation solutions that have been developed as proprietary or open source and meant to run as a software as a service (SaaS) or on-premise software. This type of a software used in support of CI workflow is called consistently and simple - Continuous Integration servers.

Figure 2 displays most popular CI servers in Java community in 2016:

- Jenkins with 60% is by far the most popular tool, which is open source and distributed as an on-premise software. Initially forked from Hudson and then developed by community and CloudBees company. [9]
- Bamboo, which is on the second place with just 9% is the closest competitor to Jenkins. It is being developed by Atlassian and is both proprietary and needs to be licensed per organization. It needs to be Installed on premises with SaaS options available [10].
- Teamcity occupies 3rd place by usage number. It is similar to Bamboo being a closed code software, which needs to be licensed and installed on premises [11].

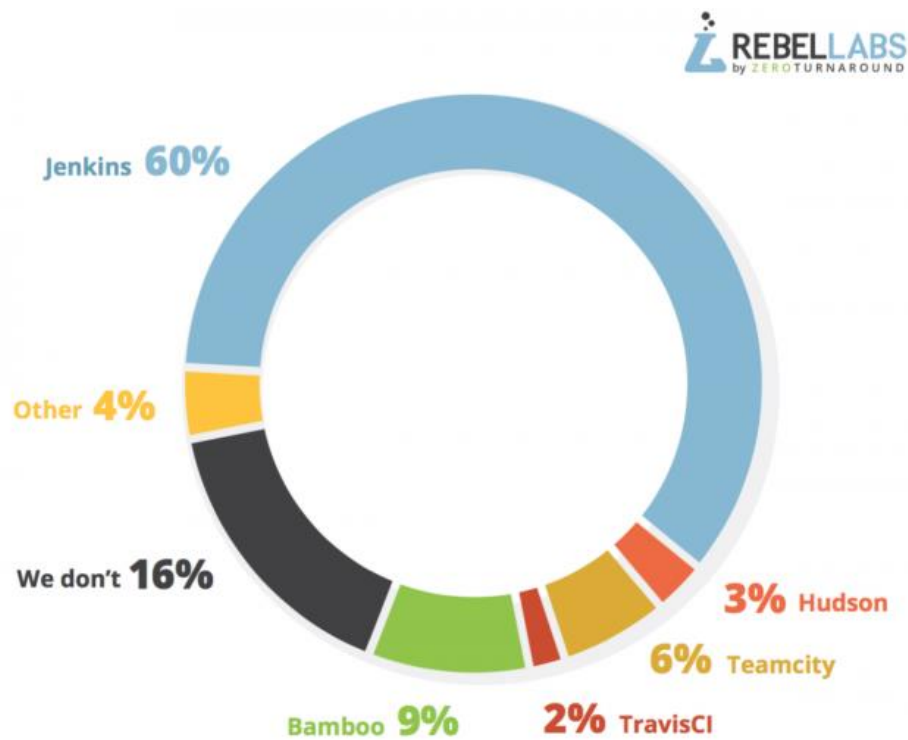


Figure 2. Continuous Integration Server Usage. Reprinted from Maple [12].

Figure 2 displays that only 16% of respondents reported to not use any CI servers, while 60% prefers to use Jenkins. CI server is an important tool for building a valid CI workflow allowing automation of builds, tests and deployments. Deployment automation is a whole topic on its own that will be discussed next.

2.2 Continuous Delivery and Continuous Deployment

Automated deployment practices come under two terms of continuous delivery and continuous deployment. These two will be referred as CD and CDt respectively.

First of all, what is the definition of CD and CDt and what is the difference between them? According to Caum blog post for a major automation tool Puppet, Continuous Delivery is a set of practices that ensure quick, constant and reliable delivery of latest code changes to a production like environment. [13]

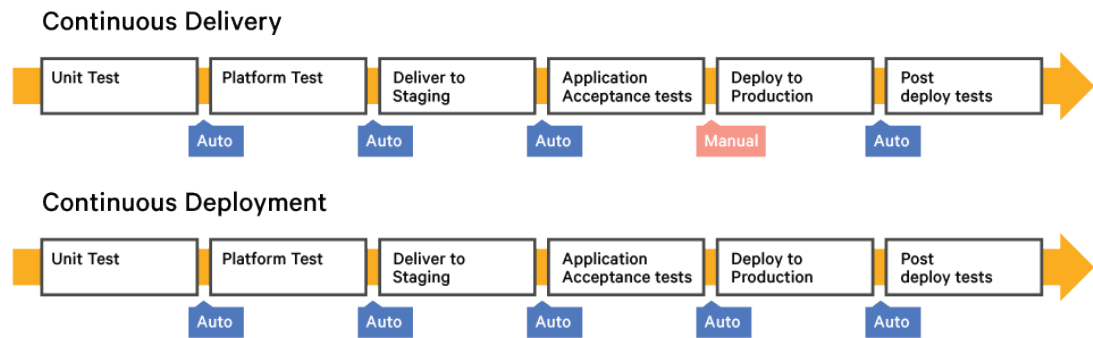


Figure 3. Continuous Delivery and Continuous Deployment Structure. Reprinted from Caum [13].

Analysing the figure 3, it is obvious that difference between the two is not a major one. Continuous deployment simply adds one more layer of automation. Unlike continuous delivery, where final decision of production deployment is manual, continuous deployment is completely automated.

It is clearly seen from figure 1 and figure 3 display of “continuous” concepts that one comes after another. And it is true that continuous delivery will not work without implemented CI pipeline. Once CI is implemented it is within reason to upgrade a pipeline to comply with continuous delivery definition. However, with introduction of continuous deployment, upgrade does not seem to be that obvious.

In practice the definition of CD is implemented in the following way. When new code is pushed to the main repository branch such as “master”, a new job is triggered in the CI server. The code is build and tested similarly to continuous integration way. Then code is deployed to staging environment and automated tests are run. It is followed by manually triggering a production deploy, which is usually highly automated except for triggering. The process as displayed on figure 3 is commonly finished with post deployment tests to verify that production environment is healthy.

The only change that continuous deployment introduces is removing the manual triggering of the production deployment step. Usually, production deployment step is already automated as part of continuous delivery. Therefore, technical implementation of completely automated production deployment is not difficult. The problems that appear from continuous delivery are rather business related or come from the way the software itself has been developed.

One common business case when continuous deployment may contradict with business goals is when a certain feature should only be released at some particular date [13] or should there be a set of new features released. Generally, there could be dozens of different business reasons why deployment should be human controlled.

Apart from business reasons there is whole set of technical ones regarding the software itself. Software needs to adhere to a whole set of specifications to allow safe use of CDt. The most important one is that application needs to be highly testable and largely covered with tests. All new functionality must be covered with tests prior to publishing code to main branch. Human factor should be excluded - all members of a team should obey this rule. Another important issue is that reliability of CI plus CDt pipeline should be higher as what happens in the pipeline is not directly monitored. [14] All this leads to a conclusion, that continuous deployment, while being a valid option, is not an obvious improvement on top of continuous delivery.

There is still one major question that needs to be resolved about Continuous Delivery. Where did it come from and what is its relation to DevOps? The origin of this term is from Agile Manifesto of 2001, first principle of which states: "Our highest priority is to satisfy the customer through early and continuous delivery of valuable software." [15] A few years later DevOps is being introduced by Patrick Debois at an Agile Conference [16]. The question of relation between Continuous Delivery (deployment was not around at that point yet) simultaneously arises as soon as one is introduced to DevOps, which at first glance may look very similar. Next section will look at this relation and define the DevOps itself.

2.3 DevOps

"Agile Infrastructure & Operations" was the original name of a methodology that was presented by Patrick Debois in 2008 [16]. Then, a new term for this very idea was introduced at Devopsdays 2009 conference in Belgium - DevOps, which stands for development operations [17]. Initial 2008 presentation addressed 2 main issues. First, even though back in 2008 Agile was an established concept, it was only applied to developers. Operations and IT parts of the companies were not part of the Agile workflows. Then, the second issue that is closely related to first one is the fact that these 3 teams are separated from each other. On the contrary, their actual work is directly dependent on each other and interconnected. The proposition was to include

infrastructure personnel to development teams and make them interact as well as introduce frequent, for example nightly builds.

Over the years DevOps methodology emerged dramatically and it is now much larger than it was initially presented in 2008. DevOps today combines a company culture, different workflows and technologies. What crucial here is that culture is, probably, more important among others.

“Even with the best tools, DevOps is just another buzzword if you don't have the right culture.” as written by Wilsenach [18]. The underlying goal of DevOps is plain and simple - remove the boundaries between different departments involved into software development process. Shared responsibility and close collaboration are the two main factors that drive DevOps culture. If application is to be transferred at some point of development to another department for deployment and maintenance the sense of ownership and responsibility is lost. On the other hand, introducing autonomous teams, where each team member is responsible for the whole application might just be a trigger to empower personnel for better quality of both software itself and deployment practices. [18] As seen in previous sections, automation is a major part of efficient development and running of the applications, and integrated teams would know best how to implement automation as much as possible as it will directly affect their daily working routine.

Initially, DevOps main goal, as its very name suggests (development operations), was to link applications developers and IT personnel responsible for actual running the application and maintenance. However, a by-product of DevOps turned out to be that, effectively, a traditional quality assurance (or QA) methods, which required a lot of repetitive manual labour, also started to fade away. Essentially, this leads to the fact that organizations, which have adopted DevOps, do no longer require traditional QA according to Yigal [19]. Yigal's article states that using DevOps implies using CI and CD, hence tests automation should already be implemented. Figure 4 below illustrates that today DevOps is at the very centre of the three: development, operations and quality assurance.

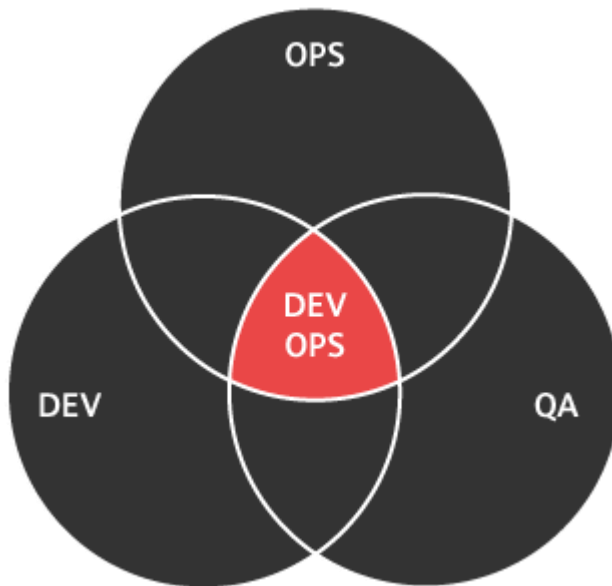


Figure 4. DevOps diagram. Reprinted from Soapui [20].

After looking at the figure 4, it is reasonable to come back to the question of relation between DevOps and CD. When technically implemented, the output of both may look similar - output is software development process, where building, deployment and testing is automated. However, the key difference between the two is that DevOps is more of a philosophy, while CD consists of concrete practices and has a much more tangible outcome. DevOps is very unlikely to be embraced by the company without implementing CD. On other hand CD could be used by traditional hierarchical organization without major challenges. [21] In the essence, DevOps affects a whole company culture by forcing automation, common responsibility and optimising most of software development processes, while CD just solves one particular problem of software deployability.

A recent 2017 DevOps report by Puppet has a few important numbers backing up the advantages of DevOps. According the figure 5 below, high DevOps performers have seen a 440 times growth in lead time for changes and 96 times decrease in mean time to recover. This means that product deployments could be made on demand as current high performers can achieve multiple deploys per day. On the other hand, recovery time has decreased to less than an hour on average. [22]

IT performance metrics	2016	2017
Deployment frequency	200x more frequent	46x more frequent
Lead time for changes	2,555x faster	440x faster
Mean time to recover (MTTR)	24x faster	96x faster
Change failure rate	3x lower (1/3 as likely)	5x lower (1/5 as likely)

Figure 5. Changes in IT performance of high performers. Reprinted from Forsgren et. al. [22]

The metrics above clearly show that DevOps brings a lot of agility to software development processes while improving overall reliability. However, DevOps is just a methodology, which implies heavy use of certain tools some of which are modern while others have been known in the industry for decades. Next subsections of theoretical background will mainly focus on those technologies.

2.4 Virtualization

In the 1960s computers were very expensive while operating systems were complex to use and not scalable. This resulted into the problem that allowing multiple users to use the same computer at the same time was not possible. The solution to this problem was called a time-sharing system at a time. The implementation of such system was difficult, and IBM addressed it by inventing of what would become the first virtualization tool and virtualization as a concept. IBM took a novel approach of creating virtual machines - VM per each user. [23]

The underlying software that enables virtualization and VMs is called hypervisor. The hypervisor is a middle layer between VMs and available computer resources such as CPU, GPU etc. It allocates resources to VMs that are running on the computer, but it also isolates one VM from another allowing them to exist independently of each other. [24] There are two types of hypervisors that are displayed on figure 6.

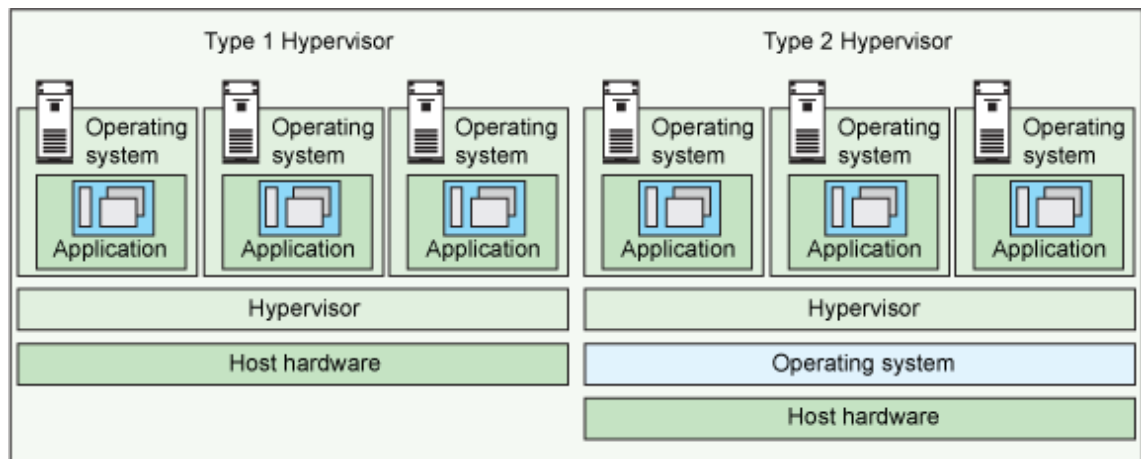


Figure 6. Type 1 and type 2 hypervisors. Reprinted from Tholeti [25].

Hypervisors from diagram above look very similar with the exception of operating system layer present in type 2 hypervisor. The first hypervisors developed by IBM were Type 1 hypervisors [26].

Type 1 hypervisors are so called bare metal. They are low level and often are a part of hardware itself. These hypervisors are very efficient as there is no host operating system overhead. Stability is also very high for computers that run type 1 hypervisor. The main disadvantage of such hypervisors is their complexity, they are difficult to start using and mainly matter when performance is of utmost importance. On the contrary, type 2 hypervisors need a host operating system (or OS) to run. They are essentially just another software that is running under OS. The main advantage of type 2 over type 1 is ease of use - user with basic computer skills can install a software based on type 2 hypervisors, one of which could be VirtualBox. Type 2 hypervisors are significantly slower due to the host OS and are not as reliable because if host OS fails all the running VMs will fail too. [24]

Back to the topic of general description of virtualization, its classification varies a lot depending on the source. One of the possible classifications is displayed below in figure 7. The initial virtualization developed by IBM and described above falls into hardware virtualization. As listed on the figure below, there are at least 7 common types of virtualization that all have common grounds in that first one developed by IBM. Nevertheless, the other virtualization types will not be discussed further as they are out of scope of this thesis. However, the two subtypes of hardware virtualization will be. Those are full virtualization and operating-system-level virtualization.

Virtualization						
Hardware <ul style="list-style-type: none"> • Full • Bare-Metal • Hosted • Partial • Para 	Network <ul style="list-style-type: none"> • Internal Network Virtualization • External Network Virtualization 	Storage <ul style="list-style-type: none"> • Block Virtualization • File Virtualization 	Memory <ul style="list-style-type: none"> • Application Level Integration • OS Level Integration 	Software <ul style="list-style-type: none"> • OS Level • Application • Service 	Data <ul style="list-style-type: none"> • Database 	Desktop <ul style="list-style-type: none"> • Virtual desktop infrastructure • Hosted Virtual Desktop

Figure 7. Classification of virtualization. Reprinted from Bhupender [27].

Full hardware virtualization is what makes possible virtual machines powered by software like VirtualBox that is very commonly used today. It is a cornerstone of today's cloud computing ever increasing server utilization and flexibility [28]. It is also popular among software developers themselves allowing them to run software in virtual machines, while keeping the host OS clean and giving a possibility to switch between very different projects many times a day [29]. However, a full virtualization is a resource heavy tool which basically leads to running multiple operating systems simultaneously on the same computer. This is where operating-system-level virtualization comes in with tools like Docker. This type of virtualization and its key representative Docker will be discussed in the next section.

2.5 Docker

Operating-system-level virtualization, or as it is also called, container-based virtualization, is a lightweight version of full virtualization. It does not imply running the full VM unlike full virtualization. Hypervisors are not used for providing resources to VMs. In fact, a VM in context of operating-system-level virtualization is called a container, which is this very virtual environment. With the absence of hypervisors, resource sharing is implemented differently - via host OS kernel. Kernel itself is providing process isolation and management of resources. Therefore, container from the inside looks very much like a normal operating system with file system, processes, memory and etc. [30]

CONTAINERS VS VMs

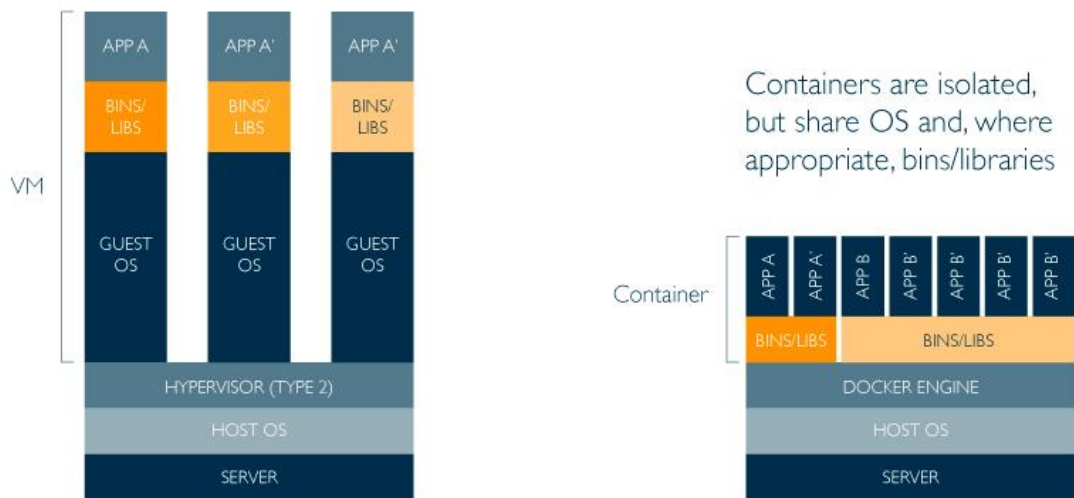


Figure 8. Containers compared to VMs. Reprinted from O'Reilly [31].

The figure 8 above displays the operating principle of container virtualization compared to traditional full virtualization used with Hypervisor 2. It is clearly seen that containers displayed on the right are more lightweight and do not require running the full OS. Hence, the main advantage of containers is their sheer performance. The performance is explained by no need to wait for resources allocation from hypervisor as well as by the simple fact that less software is run in order to make container work. More importantly, such virtualization causes less overhead on the increase of scale. [31]

Coming back to the heading of this subsection, Docker, there have been developments of operating-system-level virtualization from as early as 1979. Then in the 2000s containers were slowly growing in popularity. There are 3 events that lead to containers growing so popular. First, introduction of control groups and merging them into Linux kernel, which allowed Linux native containers in the first place. Then, in 2008, Linux Containers or LXC are created, which is a first complete implementation of container management for Linux. All of this laid ground for a first Docker release in 2013, which was initially based on LXC. [32]

In 2016, 460 thousand applications were hosted with Docker, while in 2014 the figure was only 3100 [33]. So, what differs Docker from all the other tools that were available prior and why it became so popular so quickly and widely accepted in the industry? The time of first release is an important reason in its own right. Back in 2013 cloud computing trend was gaining momentum, so Docker got introduced just at the right time to get into

the flow of success. Old container tools were already on the market but none of them were industry standard, therefore, Docker, which was a newer technology had got a chance for attention. [33]

Nevertheless, right time is not the only reason why Docker is the container technology of choice for building a modern DevOps pipeline. First of all, Docker is significantly easier to use than its older counterparts. One of the complications of operating-system-level virtualization, which Docker represents, is the fact that base images used for running containers need to be built in a specific way that container vendor supports. Docker addresses this problem by bringing DockerHub - a central repository that contains those very images. In autumn 2016, there were already around 400 thousand public images [34].

This abundance of pre-built images allows commands like the one above. The command displayed in listing would pull an nginx image from DockerHub and start a container. Then it will map port 80 in the container to port 80 on the host. Depending on the speed of internet connection setting up basic nginx takes just seconds.

```
docker run -p 80:80 nginx
```

Listing 1. Running nginx with Docker.

This functionality brings benefits not only to deployment but to a local development as well. With just one command similar to the one above, a full on isolated container can be started with an application already running. But Docker would not be complete without a whole set of different viable tools for local use or production. For example, a Dockerfile, which contain instructions to how an application should be built to run in Docker. More importantly, Docker tackles the issue of container orchestration that allows running a whole set of supporting software like database, backend and frontend that together combine into one application. A more detailed overview and use of these tools will be discussed in the next sections.

In the scope of this thesis, container orchestration is the last piece of a puzzle that is missing from starting to build a DevOps pipeline. Therefore, the next subsection will discuss the topic of orchestrating the containers.

2.6 Container Orchestration

Container orchestration is a major part of Docker ecosystem. Orchestration and how accessible it has become is one of the reasons for this thesis in the first place. It addresses multiple issues from being a means of documentation to providing a way to running applications across multiple servers in a more manageable way than traditional container free environment.

So, what does container orchestration stand for? Container orchestration manages how containers are created, updated and it controls their reliability. The control of container communication is also handled by an orchestration tool. [35] This kind of features becomes needed along with growing loads to servers where application is running. At a certain load point a single server will not be able to sustain to an application. At this point application will have to become distributed across more than one server. This is where a list of challenges like load balancing, health checks and zero-downtime deployment comes into the scene and container orchestration addresses all of them [36].

The architecture overview displayed below in figure 9 is an example of Kubernetes cluster, which is a typical example of container orchestrated environment. If such an environment is to be built without a container orchestration tool, then the challenges mentioned in previous paragraph need to be tackled. Figure 9 uses 3 worker hosts called Kubernetes nodes and a master server node, which controls the whole setup. In an orchestration free environment, the very first problem that one will have to solve, is how those independent containers communicate. Orchestration tools usually solve those problems via running an orchestrated environment in one private network that the orchestration tool helps to set up, which is much less time consuming than doing such a network configuration manually via native Linux tools.

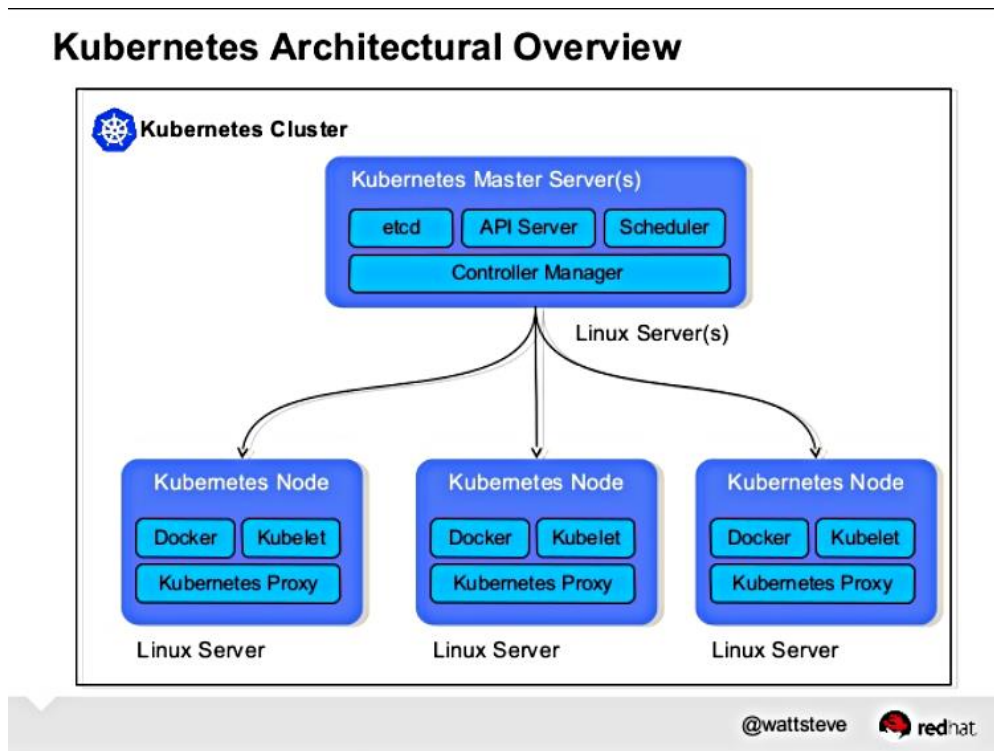


Figure 9. Kubernetes Architecture Overview. Reprinted from Schroder [37].

The second large challenge is the maintenance of a multi-server container environment. Container orchestration tool on figure 9 has a master server that provides a set of APIs that could be used to control the whole environment just like it was running on a single server. Deployment, load-balancing, monitoring and health-checks are usually either built-in into container orchestration tool or easier to implement compared to manual approach.

As container orchestration has been developing rapidly last few years, there are now tools available and all of them are built with Docker in mind. The major tools are:

- Kubernetes - Developed by Google Kubernetes is very popular with a share of 32% in 2016 [38]. Google Cloud platform, a major cloud platform, implements Kubernetes as a container service [39] but it could also be run standalone.
- Amazon EC2 Container Service (ECS) - with a share of 29% in 2016 ECS is another key player on container orchestration market [38]. It has many built-in features and largely benefits from being a part of Amazon infrastructure. Those features include identity and access management, auto scaling, load balancing, Docker images repository and etc. [40].
- Docker Swarm - this tool is native to Docker with a share of 34% in 2016 [38]. It is one of the easiest orchestration tools to start with possessing good quality documentation.

- Cattle - developed by Rancher Labs and is highly integrated with native Docker methodology. It is a base orchestration for Rancher container management server. [41]

Container orchestration offers significant advantages to adopters. However, there is a major issue that they require a steep learning curve. This could be addressed by adopting a container management server or running orchestration at a cloud platform like Google Cloud Platform. Such an orchestration platform allows to abstract large parts of tedious setup work. The next subsection researchers such platforms.

2.7 Container Management Software

Although orchestration tools provide highly transparent APIs and are already a step forward compared to traditional deployments, there is still a sizable level of complexity. Therefore, there is another layer that has recently been introduced called container management software. It stands for simplifying administration of a containerized environment providing means of creating, destructing, deploying and scaling of containers [42].

As the container and server count in an orchestrated environment grows, the management tools of Kubernetes and Docker Swarm are becoming insufficient. Understanding and managing relationships between clusters of serves and containers becomes error prone. In addition to, each of the applications running in the environment can have dozens of different settings. Hence, it becomes clear that container management software is the required addition for an efficient and reliable container orchestration. [42]

This management software can be categorised into 2 types, where one is a part of cloud infrastructure like AWS ECS or Google Container Engine and the second are standalone applications like Rancher or OpenShift [42]. Both types have considerable advantages and disadvantages. Container platform as a part of a large cloud provider has many features such as auto-scaling. However, if there is a need to change a cloud provider, such a migration could be both costly and time consuming. The infrastructure will have to be rebuilt with a whole different container management software. On the other hand, the second type, which is a self-hosted container management server does not lock a user to a particular cloud. Such management software could be installed at any server

or even on the local virtual machine. The main disadvantage of this approach is that depending on the container management software, some advanced features like auto-scaling could be limited, as there is no direct connection to cloud infrastructure. Additionally, self-hosted management software has one more advantage - it requires less prior knowledge unlike cloud provider one where knowledge of provider's infrastructure is required.

According to Rancher's own documentation, there are just a few steps that need to be performed in order to start with a basic container orchestrated environment. One would need an Ubuntu server, which has Docker running. Then, run Rancher server, which is shipped as Docker image. After that a Rancher agent would need to be started on the server and right after that environment is ready for adding applications to environments. [43]

Container management software is the last piece of a puzzle required for building a reusable template for a DevOps pipeline. The next sections will focus of work methods used in producing such a template and a description of actual project output.

3 Project Methods and Materials

3.1 Project Workflow and Methods

This project workflow consists of 6 stages displayed on figure 10 below.

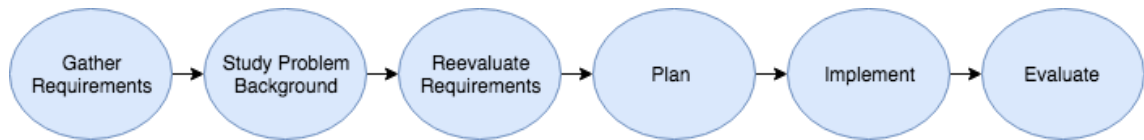


Figure 10. Project workflow.

The first stage is gathering the requirements, which are the base for the whole project. The main requirement of this project that was briefly mentioned in introduction section is to build a template for a DevOps pipeline implementing Docker. This led to a comprehensive analysis of available information on modern practices for automation of software development processes. Stage 2 - studying the background information, extensively affects the requirements for a template pipeline. As it is clearly seen from previous section there are a lot of options for building a pipeline, therefore the requirements are re-evaluated in stage 3 according to analysed background implementation with the main goal of maximum reusability.

After the requirements are updated, a high-level plan of implementation should be developed. This plan will contain diagrams and flows of pipeline demonstrating and describing purposes for all methodologies and technologies. Once the plan is ready, the actual implementation should start that would represent the end output of this project.

The final 6th stage will conclude and analyse the results of this project providing a discussion for project strong and weak spots as well as proposing future updates and improvements.

3.2 Requirements

The main two requirements proposed in introduction remain the same - resulting pipeline should be reusable as much as possible as well as use Docker. The latter requirements are based on theoretical overview conducted in previous section.

First of all, pipeline should be built primarily via open source or free software where applicable. Therefore, the technologies selection listed in the next subsection should mostly consist of those software.

Then the pipeline should be built according to latest CI, CDt and DevOps trends and be automated. When pipeline is configured, the only action required from the developers should be to push code to version control, while the rest should be handled by pipeline.

As Docker containers are the cornerstone of this project, all the latest tools such as container orchestration and container management software should be implemented as part of the pipeline. This would also allow scalability, which is another requirement. Pipeline should be easily adjusted to support environments both small and large, therefore the use of container orchestration is essential. Container orchestration will also bring in such features as load balancing and zero-downtime deployments.

Finally, a non-directly related pipeline requirement is that it should be possible to run all components of the pipeline locally via the means of virtual machines. This will enable quick testing without the need of running actual servers and will be a medium for an actual pipeline display.

3.3 Tools and Technologies

To begin with, in order to start building a pipeline, a reference project is required, against which, the pipeline will be built. Such project is provided by Eficode Oy and it will be described in better detail in the next section. Nevertheless, the project will be put in Git version control. In addition to, the project will have to be containerized and this is where Docker will be used.

Jenkins server will be used for implementing CI and CDt . It is a popular open-source solution for automation of software development that has a wide functionality and goes hand in hand with modern DevOps agenda.

Container management software will be Rancher server. Rancher has multiple advantages such as being free and open-source as well as good documentation and moderate learning curve. Rancher supports all the needed features such load-balancing and extensive management capabilities.

One of the requirements of orchestrated container environment is that Docker images need to be stored at some repository and be available for pulling on demand. DockerHub provides unlimited amount of storage of Docker images as far as they remain publicly accessible. Therefore, DockerHub will be the default repository for storing application images.

Lastly, there is a need for servers, where the pipeline infrastructure will run. VirtualBox and Vagrant paired with Ansible provisioning tool will be used for this purpose. VirtualBox is a widely-known free to use software for running full desktop virtualization. Vagrant, on the other hand, is a popular set of APIs that allows create and manage virtual machines from command line. Ansible is software that allows storing OS provision information in a form of configuration files, which can be reused or extended as many times as needed.

The tools that were introduced here should allow building this very pipeline with Docker that meets the requirements of a modern DevOps pipeline. The next section will describe the pipeline and how it was built.

4 Pipeline Design and Implementation

This section of this paper is where the actual description of pipeline design and implementation will take place. First, the architecture of reference web project and its architecture will be discussed. Next, high level pipeline architecture will be presented, following with implementation plan. Lastly, the actual implemented pipeline will be presented and explained. Both, pipeline template and Voting App are stored on GitHub and web links to them can be obtained from appendix 1.

4.1 Reference Project Description

The reference project is provided by Eficode Oy. The name of the project is Voting App. The main use case behind this project is deciding a winner project in a competition like a hackathon. Application is very minimalistic and has just 3 views.

The views of Voting App are as following, and the screenshots can be seen from appendix 2:

- /new.html - This is where a new project can be created by submitting project name and project team.
- / - The main voting view. Voter needs to first select three projects of preference, which will get 5, 3 and 1 points respectively, then enter the email and submit. One voter is only allowed to vote once.
- results.html/ - The results view, where the list of projects with corresponding points can be seen.

The high-level architecture of Voting App is presented on figure 11 and is very typical for a web project. There is a frontend that makes API requests to a backend, which stores information in the database.



Figure 11. Voting App high level architecture.

There is a following technology stack used in this web project:

- Frontend - JavaScript with Riot.js and Express.js.
- Backend - Node.js with Koa framework and Sequelize for object-relational

mapping or ORM.

- Database - PostgreSQL, which is a popular SQL database engine.

Voting app is a small project with just a few hundred lines of code. However, the size of the project is irrelevant for the end goal of this thesis - a reusable pipeline template. Voting App has all the components that are making up a typical web project. Therefore, the next subsection will discuss a pipeline high level architecture and pinpoint the exact purposes for its every component and technology.

4.2 Pipeline Flow

The final pipeline should follow a set of criteria, where the main one is automation. Once developer has delivered new code to version control, the pipeline should automatically handle the update accordingly. On high level, pipeline will follow continuous deployment pattern described on figure 3. Continuous deployment is selected to achieve the maximum level of automation maturity.

The general flow of the diagram can be observed below in figure 12. First, developer uploads new code to version control. Then, the CI server or Jenkins server, in case of this pipeline implementation, will see the new change in version control and will start a new build. All branches of the version control will be covered by Jenkins server and change in any of them will be triggering the pipeline job.

After the build has started, the CI server will build Voting App, which is the reference project for this pipeline. Then, if build is successful, the pipeline will continue to run tests, or in case of build failing, the pipeline will finish with error. Running tests against Voting App is the next stage. The pipeline will continue or finish with error depending on the results of the tests.

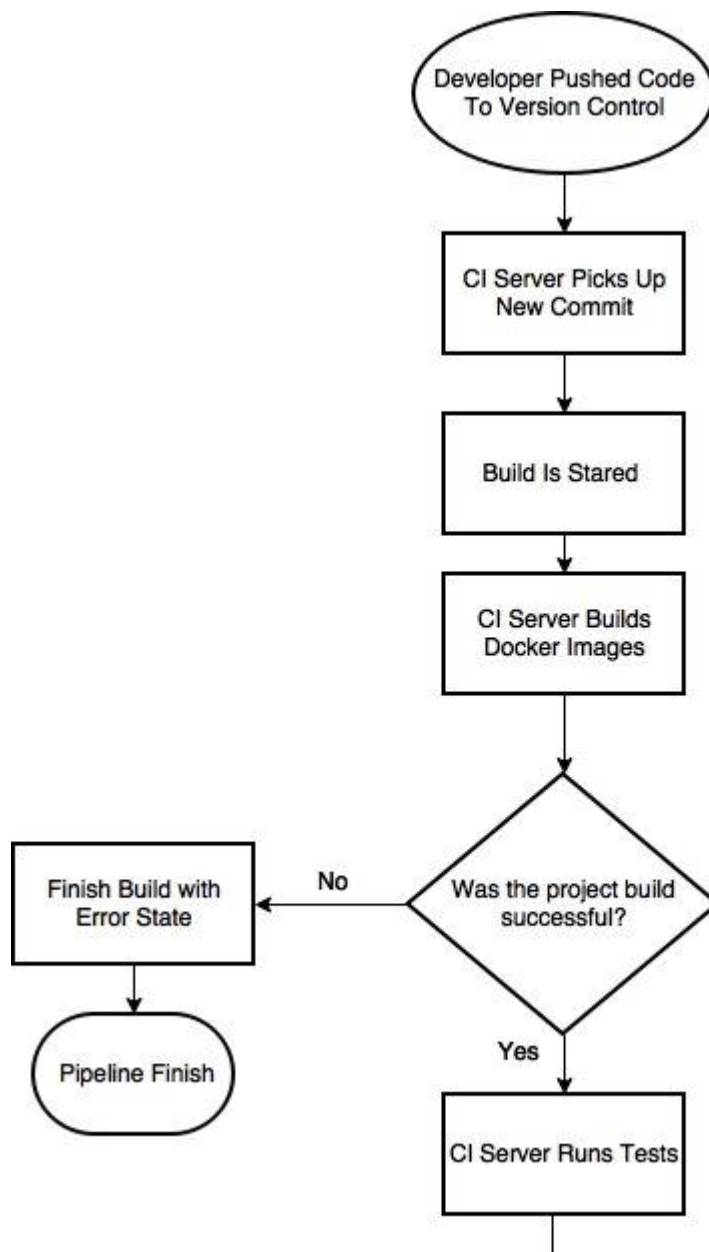
If tests were run successfully the pipeline should check the branch of the current job. If current branch is designated for deploying, then Jenkins will continue to upload of resulting Docker images from build step. If not, then the pipeline will finish with success status.

If upload of Docker images is needed, then Jenkins will try to send those images to a corresponding DockerHub repository that would require authentication. On upload fail

the build will finish with error, while on success Jenkins will send a command to Rancher server, which will automatically do everything necessary to update Voting App running environments according to new Docker images.

On receiving the command from Jenkins, Rancher will pull new images from DockerHub and update the running Voting App accordingly. The application should experience zero or close to zero downtime.

Finally, the pipeline will be finished for all possible build scenarios.



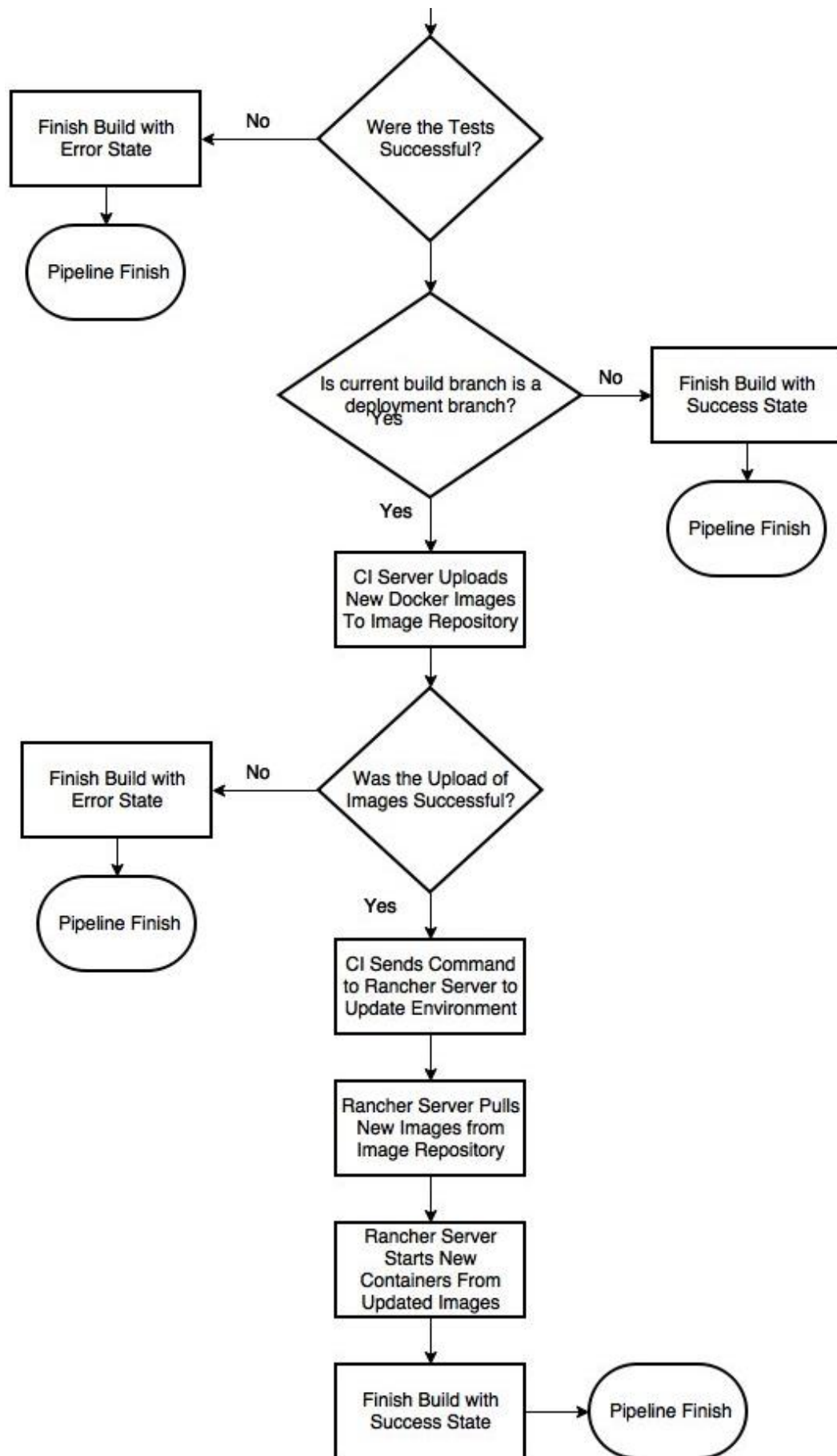


Figure 12. Pipeline Flow.

4.3 Implementation Plan

As seen from diagram above, resulting pipeline will be of noticeable complexity and will consist of multiple components. Hence, in this subsection the order of implementation covering all parts of pipeline will be discussed.

At first, Voting App will need to be dockerized. This means that configurations for building project components, such as frontend, backend and database will have to be made. Then Voting App will need to be prepared for local orchestration that will be needed for building Docker images and running tests in Jenkins. Local orchestration will also serve as a starting point for a creating a one with Rancher.

The next step would be setting up Rancher environment for the project. This will include installing Rancher server and starting application environment, uploading Docker images to DockerHub and making a configuration to enable Rancher to run Voting App.

After the Rancher environment and the project are ready, the next step would be automating the process of deployment. This would mean that Jenkins server will need to be installed and configured. Then Jenkins job will need to be created for Voting App. This job will behave according to figure 12 in the previous subsection. This is the last step to making the pipeline operational.

The pipeline infrastructure from the user point of view should look like it is displayed on figure 13. There will be a virtual machine running Jenkins Server, a main Rancher node and an n number of Rancher nodes, where the n could be changed dynamically on user preference.

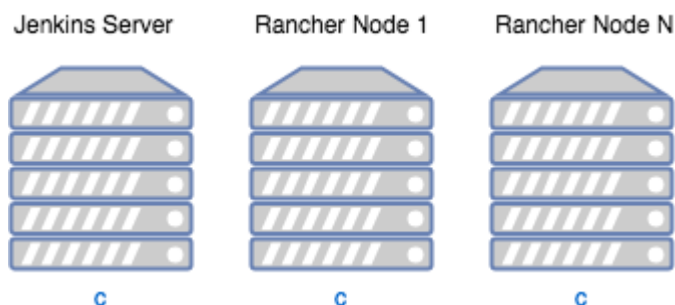


Figure 13. Pipeline Infrastructure.

4.4 Dockerizing Voting App

4.4.1 Docker Glossary

The two main building blocks of an application powered by Docker are images and containers.

Docker images are the base for containers. According to Docker's own documentation, an image can be reduced to a set of layered filesystems that are stacked on top of other, which together make up the combined content of an image [44]. On the other hand, the second building block - container, is a runtime instance of a Docker image. Container consists of a Docker image, an execution environment and a standard set of instructions. [44]

The first step then, to dockerize the project, would be to build Docker images for all the components: frontend, backend and database. The instructions for making images are stored in a so called Dockerfile [45]. Dockerfile is a text file that contains all commands that user needs to run in order to build the image. The instruction structure goes as displayed below in listing 2.

Comment

INSTRUCTION arguments

Listing 2. Dockerfile instruction structure.

First goes the instruction to execute and then the arguments that need to be passed to the instruction. Some of the more important commands are listed in the table below:

Instruction	Example Argument	Description
FROM	nginx	The base image for the image that is being build.
RUN	apt-get update	A command that needs to run on image file system that results into some changes in the file system. Installing application dependencies as an example.
WORKDIR	/home/user/project	The working directory of the image. Basically, it sets a root directory for the image.

COPY	./home/user/project	Allows copying files and directories from host filesystem into image filesystem.
EXPOSE	3000	The port that should be exposed in the container when it is running the image.
CMD	./start_project.sh	Allows to specify the default application that will be running in the container when it is using the image. If default application is crushed or finished executing the container will stop.

Table 1. Dockerfile basic instructions overview. Data gathered from Dockerfile reference [45].

It is clearly seen from the table above that Docker images and containers have a very close relationship as they are a part of a single workflow - preparing and running an application as a Docker container. Nevertheless, it is time to come back to Voting App and prepare it for running with Docker, which is the topic of next subsection.

4.4.2 Dockerizing Voting App

Both frontend and backend of Voting App are Node.js applications. The typical installation for such applications looks the following way:

- Install Node.js runtime environment.
- Use Node.js native package manager `npm` and run `npm install` in the root of the project.
- Run `npm start` command that will start the application.

The listing 3 below displays a Dockerfile definition that was developed to build Docker image for backend. First, the base image is selected with `FROM` instruction and `node:argon` argument. DockerHub provides many preset images from Nginx and Jenkins to Python and Node.js. `node:argon` here stands for the name of the image and tag on DockerHub, where `node` is the name and `argon` is a tag (of version) of this node image. Argon tag is selected as this is the node version that satisfies the dependencies of Voting App backend.

```
FROM node:argon

# Create app directory
RUN mkdir -p /usr/src/app
WORKDIR /usr/src/app

# Install app dependencies
COPY package.json /usr/src/app/
RUN npm install

# Bundle app source
COPY . /usr/src/app

EXPOSE 8080

CMD [ "npm", "start" ]
```

Listing 3. Voting App Backend Dockerfile.

The second block in developed Dockerfile starts from comment “Create app directory”. This is where *RUN* instruction is used to run Linux shell command to create a directory path */usr/src/app*, where the application is going to be stored. It is followed with *WORKDIR* command that sets the working directory of an image to the newly created application path.

The third block is responsible for installing application dependencies. A typical place for storing dependencies definitions in Node.js projects is a *package.json* file. It is copied from host to image application path with a *COPY* instruction as seen above. Then, *RUN* instruction will execute *npm install* in image to install the required packages according to *package.json* file.

Last 3 sections are finishing up the preparation of the image. First, the application source code is copied from host to image application directory. Then port 8080 will be exposed, which is an application public port defined in a source code. Finally, a command to execute on container startup is defined, which is *npm start*.

The listing above is for a backend image. However, the frontend image is almost exactly the same with a different exposed port (3000).

In order to build and test the images only two commands are required. These commands are shown in listing 4. First command will build the image according to instructions in Dockerfile. The `-t` parameter stands for defining the image name. In case shown in listing 4, the image name would be `votingapp_backend`. Then, with the second command, the container from this image would be run. In this command `-p` stands for port mapping in format `host:container` and the `votingapp_backend` stands for the name of the image to run.

```
docker build -t votingapp_backend .  
docker run -p 3000:3000 votingapp_backend
```

Listing 4. Build and run Voting App frontend with Docker.

With the approach described in previous paragraph, the whole Voting App or any application can be assembled. However, it will require tedious work on building and managing container/images individually. Then the communication between containers would be another issue that would need to be solved. The database building and creating is not specified in this part of the implementation on purpose. The reason for this is that all these issues will be addressed in the next subsection covering local container orchestration.

4.5 Local Container Orchestration

Docker Compose is a native Docker tool that was built just for the purpose of building and running Docker images on a single host. It covers many aspects of an application from ports mapping and injection of environment variables to defining dependent services. The configuration for building the environment is stored in YAML files. It is important to note that for the most part it is possible to configure container environment without Docker Compose but it largely simplifies this process, therefore it is used in this pipeline. [46]

Docker Compose has a large number of features allowing to create the environments of great complexity. However, as Voting App is a very simple project and the goal of this thesis is to produce a reusable base template for DevOps pipeline, only the features that are actually used in this project will be described. The listing 5 below displays the end result docker-compose yml file that was produced for Voting App.

```
version: '3'
```

```

services:
  backend:
    build: backend
    command: npm start
    entrypoint: ./wait-for-db.sh
    environment:
      - DATABASE_URL=postgres://votingapp:votingapp@db/votingapp
    depends_on:
      - db
    ports:
      - "8080:8080"

  frontend:
    build: frontend
    ports:
      - "3000:3000"

  db:
    image: postgres:9.4
    environment:
      - POSTGRES_USER=votingapp
      - POSTGRES_PASSWORD=votingapp
      - POSTGRES_DB=votingapp
    volumes:
      - db_data:/var/lib/postgresql/data

volumes:
  db_data:

```

Listing 5. Voting App docker-compose.yml.

Docker-compose.yml always starts with version. Depending on the specified version a different set of options will be available to the user. Then, there are services and volumes. Services are, essentially, a set of definitions that are applied to containers when they start. In the example above, there are 3 services that will respectively correspond to 3 containers. The high-level volumes definition declares different types of persistent data volumes that could be reused by different services. [47]

Going deeper into services, each of them has at least an image or build tag. In the example above *build: backend* and *build: frontend* tell Docker Compose to build Dockerfiles in the *project_root/backend* and *project_root/frontend* directories respectively, while *docker-compose.yml* is located in the root of the project. The image

key, like the one in the *db* service, is set to *postgres:9.4*. This means that in order to create *db* service Docker Compose will download PostgreSQL with 9.4 tag from DockerHub. Therefore, *db* service in this implementation does not require a Dockerfile, instead, a prebuilt image from Docker library is used.

Next, the backend has two keys of *command* and *entrypoint* but those will be addressed in the next paragraphs. *Environment* keys present in backend and db services correspond to the environment variables that need to be passed to the containers. The *DATABASE_URL* variable tells backend the address of the database container. Figure 14 displays the structure of this URL. The beginning of it is typical, containing protocol and username/password pair. The host address, *db/votingapp*, however is the point of interest. “*db*” part refers to the name of the database service in *docker-compose.yml*. This means that all the containers in the Docker Compose orchestrated environment can refer to each other by the name of the service defined in the *docker-compose.yml*. Such service discovery is implemented through a default overlay network, which also contains a DNS server. This network is automatically created by Docker Compose for all orchestrated environments.

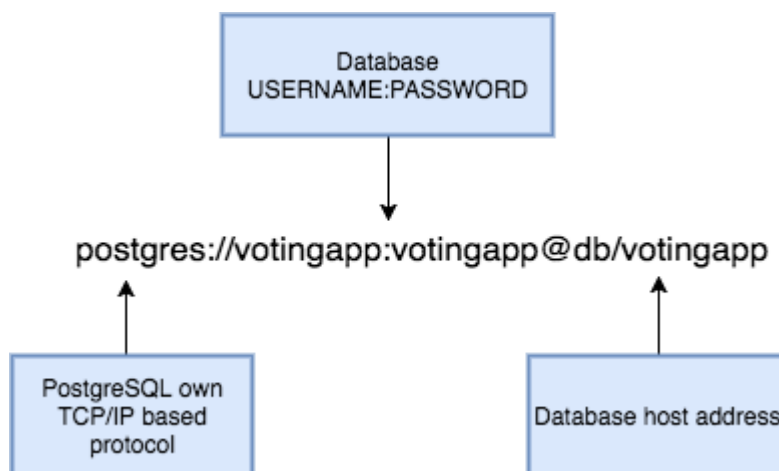


Figure 14. PostgreSQL URL structure.

Next key is only present in backend service and it is *depends_on*. The value of this key is *db*. This tells Docker Compose to start *backend* container only after *db* container has been started. This is implemented due to the fact that backend service depends on the database already running or otherwise the backend will crash. Following *ports* key that also present in the frontend service is similar to the normal Docker port mapping. The same format *host:container* is preserved.

Lastly, the *volumes* key in the *services*. Database service has it with this value *db_data:/var/lib/postgresql/data*. On the left-hand side is the *db_data* named volume defined in global *volumes* key. Named volume work the way that Docker Compose will reserve a directory somewhere on the host machine and store persistent data there. On the right-hand side of the expression is the path in the container where to make the mapping to. It is important to note that the mapping is a two-way mapping.

It is now time to come back to explaining *command* and *entrypoint* keys present in the *backend* service. One important concept about Docker is that it is not aware of what is going on inside the container processes. It is important here because backend container depends on the database one and even if *depends_on* key is set, there is still a possibility that the backend Node.js server will start before the PostgreSQL will be fully initialized and ready to accept requests. This is where *entrypoint* paired with *command* becomes highly useful. *Entrypoint* behaves as an enter script and in this case a shell script runs a *netcat* command in a loop with delay, waiting for a database port to open. When it is opened a *command*, key will be run, and the container will start. By default, Docker Compose does not run the *entrypoint*, therefore if it is set, then *command* needs to be set too, and these two keys behave like an override for the *CMD* instruction set in the original backend Dockerfile. The updated Dockerfile for the backend with installation of *netcat* and the content of *wait-for-db.sh* script could be found in Voting App repository, links to which could be found from appendix 1.

In order to start the resulting Docker Compose environment it is enough to run just one command in the root directory of Voting App. This command is displayed in a listing 6.

```
docker-compose up --build
```

Listing 6. Docker Compose build and start command.

The *--build* parameter in this command tells Docker Compose to rebuild Docker images and update containers if they are already running and there have been changes detected in the underlying Docker Images. The result of this command is three containers running in one Docker Compose environment. The next subsection will discuss the preparing for running Rancher environment and Jenkins Server in the previously discussed local environment based on virtual machines.

4.6 Virtual Machines with Ansible and Vagrant

To enable an easy access to the end result pipeline, 3 virtual machines will be created. Those virtual machines will be made with Ansible and Vagrant.

Vagrant is a tool for creating and managing virtual machines. It behaves as a wrapper on top of the standard virtual machine providers such as VirtualBox. In fact, Vagrant does not provide any built-in virtual machine implementation. In order to utilize Vagrant, one or many virtual machine providers need to be installed. [48] In case of this project, VirtualBox will be the provider of virtual machines.

Vagrant environments are defined with Vagrantfile, which contains instructions to creating virtual machines. Below is the Vagrantfile that will be used for all 3 virtual machines. The only part that will differ between environments is the IP address defined on the 3rd line.

```
Vagrant.configure("2") do |config|
  config.vm.box      = "ubuntu/trusty64"
  config.vm.network "private_network", ip: "192.168.50.5"

  config.vm.provision :ansible do |ansible|
    ansible.playbook = "ansible/provision.yml"
    ansible.host_key_checking = false
  end

  config.vm.provider "virtualbox" do |v|
    v.memory = 1024
    v.cpus = 2
  end
end
```

Listing 7. Vagrantfile.

The Vagrantfile above defines the virtual machine image that will be downloaded from Vagrant repository and sets the default network. Then, Ansible block tells Vagrant to use Ansible as a provision tool. Last block defines VirtualBox as the provider and sets virtual memory to 1024 MB and CPU to 2 virtual cores.

Ansible is a provisioning tool, which works according to infrastructure-as-a-code principle. This principle means that server configuration needs to be developed once and

put into Ansible configuration files. Then Ansible can be used to provision remote or local servers any number of times from the single set of configuration files. [49] This gradually decreases server setup time and provides a documentation close to human readable format. The major advantage of Ansible to competitors is the fact that end servers do not need any special agents installed. Ansible will push its modules to the server on demand depending on type of provisioning required. [49]

```

---
- hosts: all
  become: yes
  become_method: sudo
  roles:
    - common
    - jenkins
    - docker
    - swap
    - finishing

```

Listing 8. Jenkins Node provision.yml.

The code above displays Ansible endpoint YAML file - *provision.yml*. This file tells Ansible to provision all available hosts, which are the virtual machines in this particular case. It also gives an instruction to run all commands as root user and then there is list of roles that Ansible should execute. Roles are a key concept of Ansible. Each role usually corresponds to one piece of software or a task that needs to be run. In this case common dependencies will be installed, following with Jenkins Server and Docker. Then swap file will be added for better virtual machine performance and lastly the *finishing* role will trigger a virtual machine reboot.

Below is the role that will provision the Jenkins Server. *apt*, *apt_repository* and others are the examples of Ansible modules that will handle the installation. According to the YAML file above, a *daemon* dependency will be installed, which is required by Jenkins server. Then Java 8 is installed from official repository. It is followed by the installation of Jenkins Server 2.79 from a *.deb* package. The last command will start Jenkins.

```

- name: Install daemon
  apt: name=daemon update_cache=yes state=latest

- name: Add Official Java Repository
  apt_repository:
    repo: ppa:webupd8team/java

```



```

state: present

- name: Accept Java 8 License
  become: yes
  debconf: name='oracle-java8-installer' question='shared/accepted-oracle-license-v1-1' value='true' vtype='select'

- name: Install Java 8
  apt: name=oracle-java8-installer update_cache=yes state=latest

- name: Get Jenkins Deb Package
  get_url:
    url: http://pkg.jenkins-ci.org/debian/binary/jenkins_2.79_all.deb
    dest: /opt/jenkins_2.79_all.deb

- name: Install Jenkins
  apt: deb="/opt/jenkins_2.79_all.deb"

- name: Start Jenkins
  command: /etc/init.d/jenkins start

```

Listing 9. Jenkins Ansible role.

All it takes to start an Ansible provisioned Vagrant host is a command that is displayed in listing 10. This will bring up a virtual machine and then Ansible will provision according to *provision.yml* file and selected roles. All 3 resulting virtual machines will be configured the same way. Web link to the pipeline repository containing the rest of Ansible and Vagrant configuration for all 3 hosts can be found from appendix 1.

```
vagrant up
```

Listing 10. Vagrant command to start virtual machine.

4.7 Configuring Rancher Environment

In this subsection Rancher server will be installed and configured. After it is done Voting App configuration will be developed to make it run with Rancher.

4.7.1 Configuring Rancher Server

Rancher is shipped in the form of a Docker image [43]. The command used in Ansible to

provision Rancher Server could be seen below in listing 11. This command will start container with the name “rancher” from the image *rancher/server:v1.6.10*. The MySQL data from Rancher will be stored in */opt/rancher/mysql* on virtual machine file system. In case if container crashes, Docker is instructed to restart it right away. Public port 8080 is mapped to the container. Lastly, the *-d* option stands for running the container in a detached or, in other words, background mode.

```
docker run -d --name rancher -v /opt/rancher/mysql:/var/lib/mysql --restart=always -p 8080:8080 rancher/server:v1.6.10
```

Listing 11. Jenkins Ansible role.

When server, starts it is available on *http://192.168.50.4:8080*. First of all, a host registration URL is required. The default value provided by Rancher will be used for registration URL. The screen demonstrating the Rancher UI could be seen on figure 15 below.

The screenshot shows the Rancher UI interface for adding a host. At the top, there is a navigation bar with 'Environment', 'Default', 'STACKS', 'CATALOG', 'INFRASTRUCTURE', 'ADMIN!', and 'API'. Below the navigation bar, a warning message is displayed: 'Before adding your first service or launching a container, you'll need to add a Linux host with a supported version of Docker. Add a host'. The main content area is titled 'Hosts: Add Host'. Underneath, there is a section for 'Host Registration URL' with the question 'What base URL should hosts use to connect to the Rancher API?'. There are two radio button options: 'This site's address:' (selected) and 'Something else:'. The 'This site's address:' option has a text input field containing 'http://192.168.50.4:8080'. The 'Something else:' option has a text input field containing 'e.g. http://example.com:8080'. Below the input fields, there is a note: 'Don't include /v1 or any other path, but if you are doing SSL termination in front of Rancher, be sure to use https://'. At the bottom of the form, there is a blue 'Save' button. A teal information box at the bottom of the form contains an information icon and the text: 'Are you sure all the hosts you will create will be able to reach http://192.168.50.4:8080? It looks like a private IP or local network.'

Figure 15. Selecting host registration URL in Rancher.

The following screenshot below displays a screen where Rancher first host should be added. On the top of the screen there are presets for adding hosts in different cloud computing providers and a “Custom” preset that would work for any Linux server and it

is the one used here. In the list below under number 2, there is a message specifying that in order for Rancher hosts to work correctly, UDP ports 500 and 4500 need to be open. Configuration option number 4 allows to enter a custom public IP address that should be assigned to the host. Finally, under number 5 there is a command that needs to be copied and executed on the target host, which will start the Rancher agent and add the host into the environment. The operation of adding hosts will be repeated twice to add both of the hosts.

Hosts: Add Host

Manage available machine drivers

- 1 Start up a Linux machine somewhere and install a [supported version of Docker](#) on it.
- 2 Make sure any security groups or firewalls allow traffic:
 - o From and To all other hosts on **UDP** ports **500** and **4500** (for IPsec networking)
- 3 Optional: Add labels to be applied to the host.
 - + Add Label
- 4 Specify the public IP that should be registered for this host. If left empty, Rancher will auto-detect the IP to use. This generally works for machines with unique public IPs, but will not work if the machine is behind a firewall/NAT or if it is the same machine that is running the **rancher/server** container.
- 5 Copy, paste, and run the command below to register the host with Rancher:


```
sudo docker run --rm --privileged -v /var/run/docker.sock:/var/run/docker.sock -v /var/lib/rancher/r:/var/lib/rancher rancher/agent:v1.2.6 http://192.168.50.4:8080/v1/scripts/4211B8729BACC153CEE5:1483142400000:KdVTnnKwaIbe3xa2jC411ezVtXE
```
- 6 Click close below. The new host should pop up on the **Hosts** screen within a minute.

Close

Figure 16. Selecting host registration in Rancher.

By the completion of adding hosts, Rancher environment will be almost ready for adding a real project, which is Voting App in case of this thesis. The screenshot below on figure

17 displays the two resulting hosts running on two virtual machines as part of one virtual environment. So far, the orchestration tool that will be used in the environment was not mentioned. By default, Rancher uses its own native Cattle orchestration and this is the one that is used in the environment that have just been created. Nevertheless, the next step is to prepare Voting App project for running in a Rancher environment.

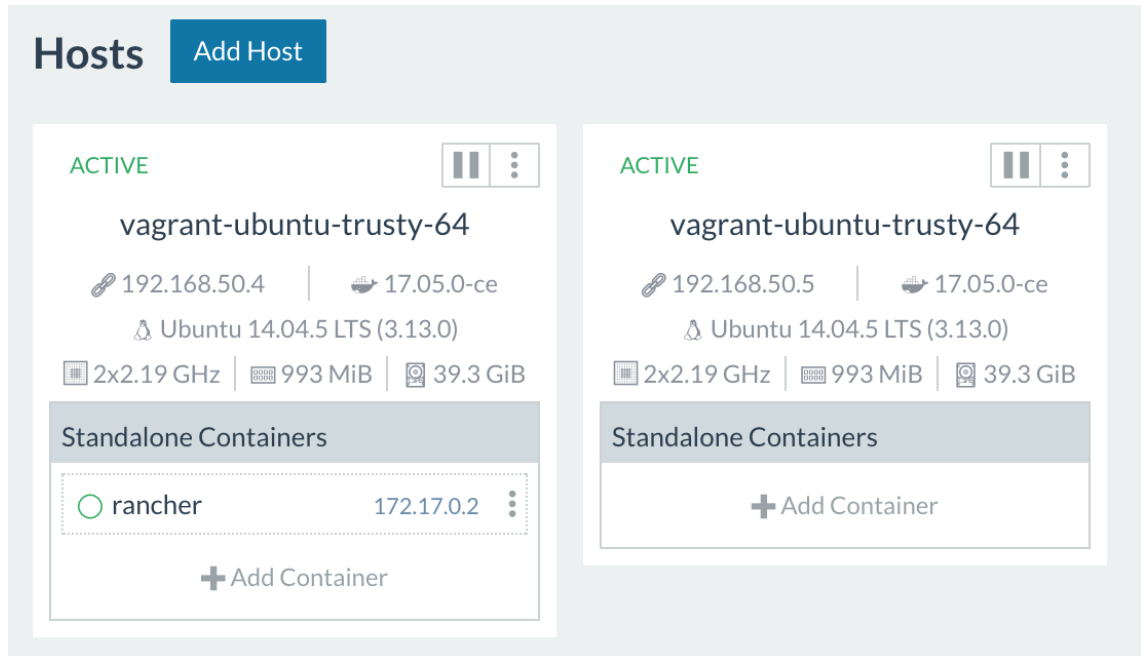


Figure 17. Two resulting hosts in Rancher environment.

4.7.2 Preparing and Running Voting App in Rancher

The first step to prepare Voting App for deployment with Rancher is to push the corresponding images to repository. DockerHub conveniently provides free of charge unlimited storage for public images. Two Docker repositories were created for this purpose. They are:

- allhaker/votingapp_backend - for Voting App backend.
- /allhaker/votingapp_frontend - for Voting App frontend

As Jenkins server is not yet running, the Docker images were pushed manually using Docker CLI. First `docker login` command, displayed on listing 12, was used to login to DockerHub. Then, the resulting images from Docker Compose build were each tagged with new image repository and then pushed. Docker tag and push for Voting App frontend could be seen below on the listing 12.

```
docker login
```

```
docker tag devopspipelinedemoproject_frontend allhaker/votingapp_frontend
docker push allhaker/votingapp_frontend
```

Listing 12. Tagging and pushing Voting App frontend.

After the images are in the DockerHub, the next step takes place in Rancher. Under the “Infrastructure” menu there is an item “Registres”, which links to the page where registries can be added to Rancher. A DockerHub registry for Voting App repository is added there with corresponding credentials.

The Rancher orchestrated environment is now ready for adding a stack. A stack in Rancher terminology is an application that is run and managed by Rancher. Rancher stacks are configured with *docker-compose.yml* and *rancher-compose.yml*. The *docker-compose.yml* is very similar to the original implementation by Docker Compose with some Rancher specific modifications. *rancher-compose.yml* on the other hand is Rancher’s own configuration file that allows specifying extra parameters like health-checks, load balancers and etc.

Full versions of Voting App YAML files could be found in the appendixes 3 and 4. Here, only the most important differences from original *docker-compose.yml* file will be covered. First of all, *image* keys for frontend and backend are replaced with corresponding paths for Docker images on DockerHub.

Then, there is a new key *labels*. Labels are used to pass some specific parameters to Rancher about how to orchestrate a stack. For example, *io.rancher.container.pull_image: always* and *io.rancher.scheduler.global: 'true'* stand for always downloading new Docker image from the image registry and running the container globally with one instance on each of the available hosts respectively. The database service also has the *labels*, which tells Rancher to start database container only on a host that is labelled *Database=true*. The database host is Host 2 with IP address of 192.168.50.5. This is done due to the fact that database needs a volume and therefore it should always be started only on a host where this volume will be preserved.

There are also two brand new services in the YAML file. They correspond to a load-balancer and a special instance of the backend that will execute a script on the backend that will in turn initialize the database with default values and shutdown.

The relation between containers could be observed on figure 18 below. *backend* and

seed-db services depend on and link to the database. The *balancer* service, which corresponds to load-balancer, links to both *frontend* and *backend* services. In this application load-balancer serves two main purposes - it provides a single public endpoint for frontend and backend and directs traffic equally to containers. As it is seen from figure 18, there are two instances of frontend and backend, therefore they need special software in front of them to coordinate where traffic goes, and the load-balancer solves this issue.

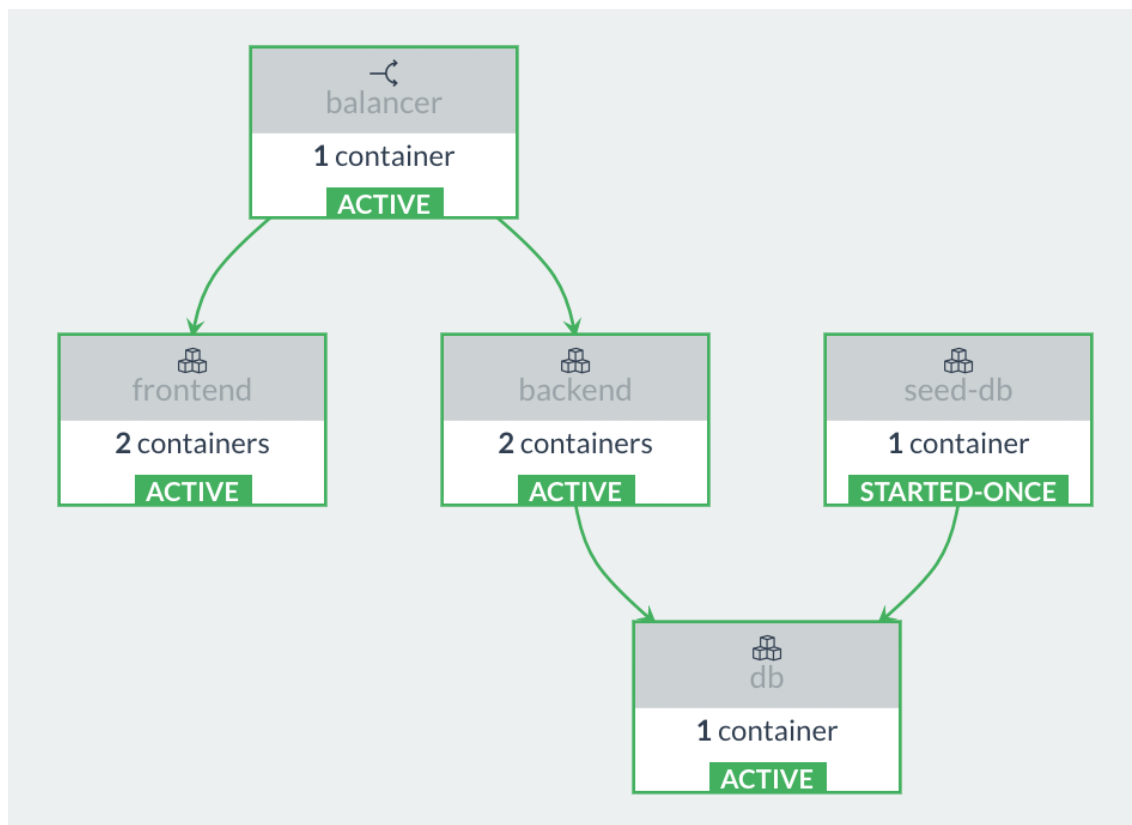


Figure 18. Voting App Rancher link graph.

Generally, load balancers are a difficult software to configure correctly. However, Rancher uses its own implementation of HAProxy load-balancer. This means that Rancher has a special UI that can be used for configuring the load-balancer. This UI is displayed on appendix 5. Load-balancer is configured with this UI. Public port is set to HTTP port 80 for both frontend and backend. Then, the backend URL starts from */api/v1* and frontend with just */*. Target containers are set to *backend* and *frontend* respectively followed with corresponding port numbers of the running applications in those containers.

Before continuing to the next section one more piece of configuration is required. When Jenkins server is up and running, it would require a medium for triggering Rancher environment upgrade. For this very purpose Rancher has got an implementation of

webhooks. First, labels *service: backend* and *service: frontend* need to be assigned to backend and frontend respectively. Then, under “API” tab, “Webhooks” menu item should be selected in Rancher UI. After that a new webhook could be created. The screenshot of creating a webhook for backend service could be seen in appendix 6. First, webhook name is supplied. Then a kind of webhook is selected, which in this case is “Upgrade a Service”. Next, image tag of target service is set. The following “Service Selector” section is where previously mentioned labels are utilized. This is needed in order to enable Rancher to find the right service to upgrade. Last three options are specifying the exact procedure of an upgrade and they are left to default settings. After all these settings are specified a webhook can be created. The same webhook creation procedure was repeated with corresponding settings for the frontend service.

The Rancher is now fully configured to ensure the needs of the pipeline. Next subsection will present a short overview of the resulting Rancher environment.

4.7.3 Environment Overview

The resulting environment overview is displayed on the figure 19 below. There are 3 virtual machines which are Jenkins Server as well as Host 1 and Host 2 of Rancher environment. The IP addresses are displayed on the diagram as well.

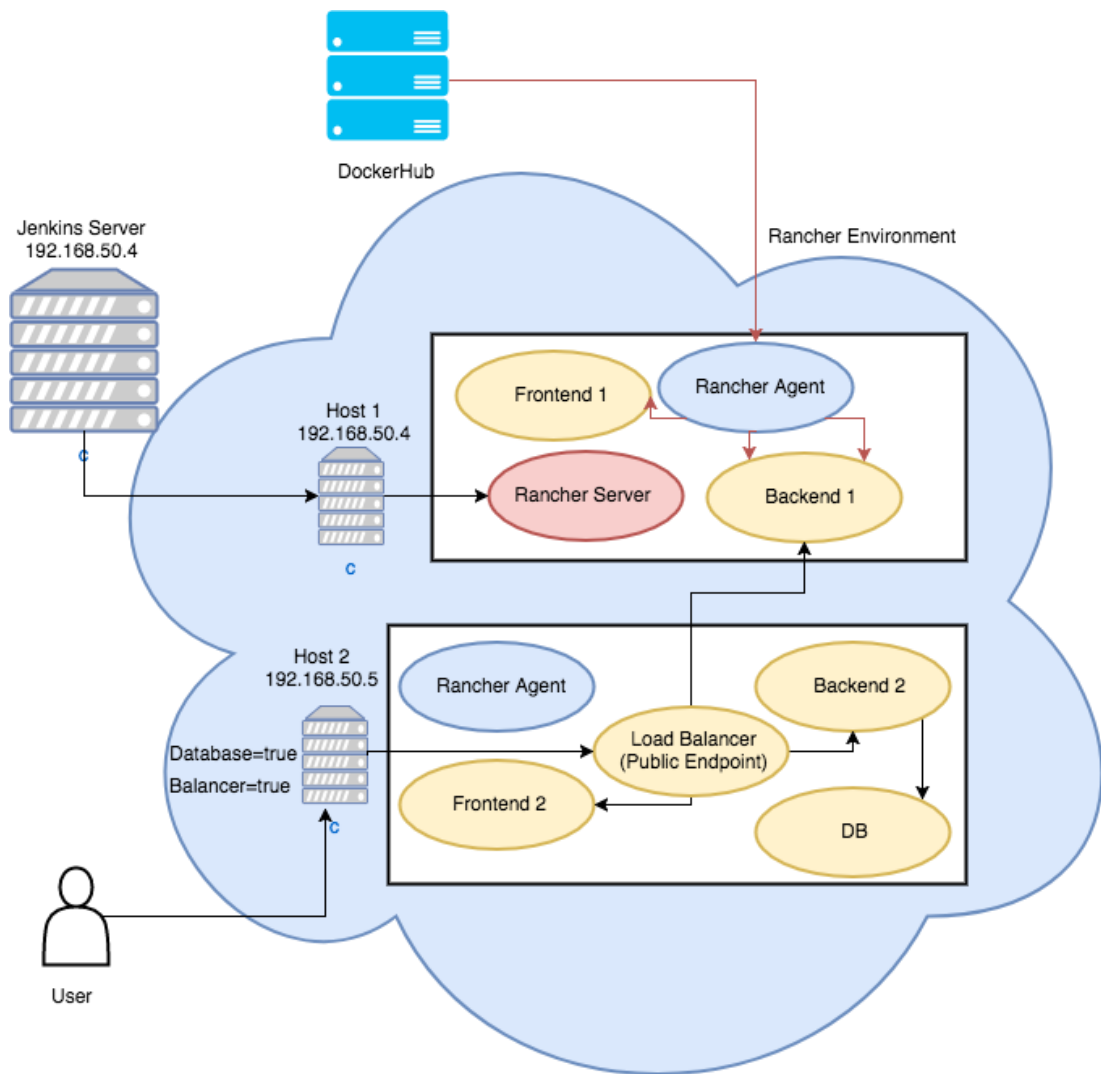


Figure 19. Resulting environment architecture.

Rancher server is running on Host 1 alongside with Rancher agent and one instance of each frontend and backend. Host 2 has *database* and *load-balancer* labels that allow scheduling database and load-balancer containers only to Host 2. Load-balancer presence on Host 2 means that it behaves as a public endpoint for users. Host 2 also has Rancher agent and instances of frontend and backend.

The communication of containers between each other is also displayed on the diagram. When requests reach load-balancer, they are directed to one of the frontend or backend containers. Both backend instances, if needed, can call the single database instance on Host 2.

Rancher agent containers on both hosts manage application containers. If container

upgrade is requested, agents will take care of downloading corresponding images from DockerHub and start new containers accordingly. Adding DockerHub credentials to Rancher was described in previous subsection.

The last missing part of the environment is the delivery automation of the Voting App new versions. Hence, the next subsection will describe the configuration of Jenkins to enable continuous integration and continuous deployment functionality for the pipeline.

4.8 Implementing Jenkins Pipeline

Implementing the CI and CD flows with Jenkins server is the last missing piece of this DevOps pipeline template. This was being completed in two steps. First, Jenkins was configured and then a Jenkins job was developed according to figure 12 described in section 4.2

4.8.1 Configuring Jenkins

Jenkins virtual machine is created the same way as Rancher hosts, via Vagrant and Ansible. When virtual machine is started, Jenkins would be available on `http://192.168.50.3:8080/`. To begin with, it has to be unlocked. According to the instructions on Jenkins unlock screen that are presented in appendix 7, there is a special file in Jenkins directory that contains a default unlock password. Commands in the listing below were executed in order to fetch the password. First command prints the list of virtual machines including their identifiers (ids). The second command connects via SSH to a virtual machine with *id* of `3e7ecc2` that corresponds to Jenkins server VM and then a `cat` command is run on a virtual machine to print password to host computer terminal.

```
vagrant global-status  
vagrant ssh 3e7ecc2 -c "sudo cat /var/lib/jenkins/secrets/initialAdminPassword"
```

Listing 13. Unlocking Jenkins Server.

Once Jenkins is unlocked, it prompts the initial plugin installation screen that could be observed on the appendix 8. Jenkins is a very modular software and without plugins it loses most of the functionality. Therefore, a default offered set of plugins is installed.

Jenkins can run tasks for a multitude of projects. For every set of tasks Jenkins has a

job. Therefore, Voting App needs a job in Jenkins. The job creation screen could be seen in appendix 9. There are several types of available jobs. However, multibranch pipeline would suit best for purposes of this thesis. This job type requires access to the repository of project of interest and the instructions for how Jenkins should run the job are taken from special *Jenkinsfile* configuration file.

There are two main advantages that multibranch pipeline job type has. First, Jenkinsfile have to be stored in the project source itself, therefore it is easily accessible by project developers, and this complies with DevOps flow described earlier. Second, all project branches that have Jenkinsfile present are automatically covered by Jenkins job, which, in turn if automated tests coverage is extensive can help to find bugs on the earlier stage.

After the job type and name defined, Jenkins would offer to enter some settings for the job. In this case only the source of the project need to be configured. One of the default Jenkins plugins was the GitHub plugin, which is pulling Voting App source code. Therefore, under the branch sources, a GitHub source is selected in the UI. Then, owner is set to *allhaker* (this is a GitHub username of the researcher) and Voting App is selected from the list. The result of this could be seen on the screenshot below. The rest of the job settings are left as default.

The screenshot shows the 'Branch Sources' configuration page in Jenkins. It features a 'GitHub' section with the following fields:

- Credentials:** A dropdown menu showing '- none -' and an 'Add' button with a key icon.
- Owner:** A text input field containing the username 'allhaker'.
- Repository:** A dropdown menu showing 'devops-pipeline-demo-project'.

There are blue question mark icons to the right of the Credentials, Owner, and Repository fields. A yellow warning triangle with the text 'Credentials are recommended' is positioned between the Credentials and Owner fields.

Figure 20. Selecting Jenkins project source.

On submit event, Jenkins will pull the repository and finish with message that no *Jenkinsfile* is found in any of Voting App branches. Hence, this *Jenkinsfile* will need to be implemented and it will contain the definition of the pipeline. However, before proceeding one more piece of configuration is required. Jenkins job will be pushing Docker images to DockerHub registry. This means that Jenkins will need to store the credentials for this registry.

Jenkins offers a built-in credentials storage. The new credential screen could be accessed navigating in the UI. First, “Credentials” menu item from Jenkins menu on the main page needs to be selected. Then, “Jenkins” credentials need to be selected following by selecting “Global credentials (unrestricted)” domain on the next screen. Lastly, “Add Credentials” would need to be selected from the menu on the left of the screen. New credentials screen could be observed on appendix 10. Username and password that are typed in are the same as the ones in Rancher. “ID” field provides a way to access credentials in the Jenkins job and is set to “docker-login”.

With Jenkins job configured and DockerHub credentials being prepared it is now time to continue to the next step - implementing a *Jenkinsfile*.

4.8.2 Developing Jenkinsfile

Jenkinsfile is a text file that contains instructions to how Jenkins should execute a pipeline build. *Jenkinsfile* uses Groovy programming language as a base. There are two main ways in which *Jenkinsfile* can be written: declarative and scripted. Declarative option is a newer one and has higher readability. On the contrary, a scripted one allows more freedom in running custom commands as part of the build. [50] This thesis implements declarative *Jenkinsfile* for the reason of simplicity and better readability. The full version of the *Jenkinsfile* could be observed in the appendix 11.

The very first line of the file defines a global *compose* variable that helps to reduce the unnecessary use of the repeating plain text. After that, the actual pipeline starts with *pipeline* directive that wraps the whole code. On the next level there are 4 directives.

First directive, *agent*, specifies on which Jenkins agent the build should be executed. In case of this pipeline there is only one agent - the default built-in Jenkins agent. That is why the value of the directive is *any*. *options* can have a large number of different settings defined for this pipeline, for instance, a build timeout. The next directive is the one where the actual build stages are defined and where the pipeline code is located. Last directive, *post*, is similar to *stages*, except it is triggered only after all build stages are complete and where errors could be caught or clean up operations could be performed.

Going one level deeper into *stages* directive, there are multiple *stage* directives that take stage name as an argument. Each *stage* has *steps* directive which is where the actual pipeline commands are located. First stage checks out the code from version control.

Checkout stage is standard and usually present in most Jenkinsfiles. The actual build starts from “Build” stage.

In the *steps* directive of “Build” stage there is a *sh* directive with a string argument. *sh* directive will tell Jenkins to run the argument script as shell script on the server, where Jenkins is running. In this case it will trigger a Docker Compose to build Docker images. The next stage will run mocha tests that are a part of Voting App backend. If tests fail or if any other stage in the pipeline will finish with error result, Jenkins server will abort the build and finish with error.

“Tag and Push Docker Images to DockerHub” stage introduces a new directive *when*. This allows making some stages optional. The use case for this is that Jenkins job was configured so that any branch of Voting App, if it has the *Jenkinsfile* and commits are being made there, will be covered by pipeline and Jenkins will be executing builds. However, the desired behaviour is to have built and test stages present in all branches but deployment only for master branch. Hence, optional pipeline stages serve this purpose. The *expression* directive inside *when* will evaluate Jenkins environment variable that contains current branch name against “master” and the stage will only be run if job branch is master.

Inside already discussed *steps* directive lies *withCredentials* directive. Its purpose is to fetch credentials from Jenkins local storage of available credentials and inject them into usable variables. The credentials in question are the ones for DockerHub repository of Voting App and they were added in the previous section. Credentials are then used in headless Docker login command. The rest of the stage consists of already described *sh* directives that will tag and push Docker images of Voting App to repository and the stage will be finished with a logout command.

The next stage “Upgrade Rancher Environment” just as previous one runs on the master branch only. This is where webhook configured in Rancher will be utilised. There is a new *script* directive here. It allows running Groovy code and defining variables. There were two webhooks established in Rancher. Therefore, there are two *curl* shell commands in this stage executing *curl* to trigger webhooks. First, http headers and http method are supplied to *curl*. Then goes the payload that specifies image tags that correspond to target containers and repository name of the image to update previous one to. Lastly, the *curl* contains the webhook URL provided by Rancher. After the two webhooks are executed, Rancher will start an upgrade procedure. There is a

disadvantage here caused by the very nature of webhooks. Jenkins pipeline will have no means of knowing whether the Rancher Voting App environment deployment went successful or not.

After all stages are complete, the pipeline will finish with success status and Jenkins will trigger a post step, which is in this case will bring down Docker containers and their corresponding images.

Installing and configuring Jenkins server and implementing *Jenkinsfile* was the last step to finalise the pipeline. The next section will present the results and their analysis.

5 Results and Discussion

5.1 Summary of Results

The resulting pipeline, first of all, follows the principles of DevOps, which, as the title of this thesis suggests, was one of the main goals. It covers all the parts of an application lifecycle. Voting App was fully dockerized and full local development environment could be started with just one Docker Compose command. If Docker is used by a whole development team of a project dockerized in this way, then each team member will have exactly the same environment, which mitigates the chances for random dependencies or underlying programming language versions conflicts.

There is a full CI and CD workflow built as part of a pipeline. Jenkins job covers builds, tests and deployments. It is completely automated and does not require special attention. On the other hand, Rancher server handles the live environment of the application. With the container orchestration application could be easily scaled on demand. The main advantage of Docker here is that it still remains the single software shipment medium. Docker images that run by developers are the same as those run in Jenkins or Rancher.

The demo setup which consists of 3 virtual machines is built with Vagrant and Ansible. Those could be started in just minutes. Having all configuration files of a pipeline already in place, it takes less than an hour to configure Jenkins and Rancher with their graphical user interfaces. Therefore, pipeline is very portable and environment agnostic, meaning that it can be hosted locally, on bare metal servers or in a cloud provider infrastructure.

The second main goal of creating the pipeline template that could be reused in the future is achieved. Docker, Docker Compose, Jenkins, Rancher, Ansible and Vagrant configuration files can, to a large extent, be reused. If a new project needs a pipeline and this project is a Node.js one, then all the files are almost 100% reusable. More than that, Jenkins and Rancher can host as many projects as one would like if there is enough computing power. This brings reusability to an even higher level.

Another result, which was not initially at the core of the pipeline, is that with such an abundance of configuration files that cover all the major parts of an application, they work as a self-documentation. If one is familiar with core technologies used in this pipeline, then, by exploring the configuration files, it is possible to deconstruct how all the parts of

the pipeline work to the very root of it.

5.2 Evaluation of Results

The evaluation needs to be started from analysis of to which extent DevOps principles are followed by the pipeline. The culture of DevOps, essentially, is about centralising development, operations and quality assurance as one concept. All of those are covered by the different parts of the pipeline. Most importantly, there is no physical division of what potential developers are able to do with the pipeline. All the application specific YAML files are in the repository and directly accessible by developers.

If there is a part of the pipeline that might need to be handled by the operations personnel, then this is infrastructure. The server infrastructure is implemented via VMs created and managed by a Vagrant and Ansible pair. However, this is far from being the most important part of the project, but rather a way to provide infrastructure. Rancher and Jenkins, that are running in the infrastructure, need to be configured only once and then any multitude of projects can be run on this infrastructure.

On the contrary, application specific configuration files are located in the repository. Developers will be the ones, who are managing them. There is no need to contact operations staff in order to configure the new application in server environment. Dockerfiles and Docker Compose YAML files contain all the required instructions to prepare an application and then ship it in the form of Docker images via the pipeline. The possibility of misunderstanding between different teams resulting in overhead is highly reduced.

There is a total of 269 lines of Dockerfiles, Docker Compose, Rancher and Jenkins configuration files in the pipeline. In case if a similar Node.js project would need to be converted to the developed pipeline, in the best-case scenario, 191 lines of configuration files will remain the same. This is the rate of 71% of configuration being completely reusable. The Dockerfiles can be copied into a new project without changes. In the worst-case scenario, only about a half of the configuration will have to be removed, being the worst-case scenario is a project based on a completely different programming language and architecture.

The average time for Jenkins pipeline job to finish all the stages is 47.5 seconds

calculated from 4 consecutive builds. In case if unexpected slowdown occurs, this figure is rounded up to 60 seconds. The Rancher takes about 30 seconds to update all the containers. However, new Jenkins build can be started before Rancher will finish environment upgrade. If there are 7.5 hours in a typical working day, which corresponds to 450 minutes, then 450 deployments could be made in a single day. Such frequent deployments are, probably, not business viable but it shows that the achieved pipeline is very flexible for any kind of deployment schedule.

During the upgrade of Voting App the total downtime is equal to zero. There is noticeable delay in getting requests from both frontend and backend if the requests are being made at exactly the moment, when one of the containers is being shut and another one is being started. **Nonetheless**, all the requests are reaching the end user, which, everything considered, can be called a zero downtime deployment.

There was also a critical error test conducted. Voting App runs on 2 servers in Rancher environment, which are Host 1 and 2. Host 2 houses load-balancer and database. This is a single point of failure. If Host 2 goes down, then Voting App will no longer be accessible. However, in case if Host 1 goes down, which was simulated by powering down the virtual machine, the application will remain operational. There was a noticeable slowdown in time of reply from the application in the first 30 seconds after Host 1 was brought down. Yet, application remained operational and stabilised after these 30 seconds. Therefore, resulting environment can be called reliable.

Despite all the benefits, pipeline possess a set of disadvantages. First of all, it is the very complexity of the pipeline. There are a lot of different technologies used, which all require corresponding configuration files. One will need to spend a considerable amount of time to figure out, how all of these technologies work together and form a pipeline. However, this flow would need to be learnt just once and there is fair amount of abstraction available to the pipeline user. Those mainly come from running the whole environment locally with Docker Compose and Rancher. Both of them automatically create application network with Rancher also allowing to configure complex load-balancing and health checks from the graphical user interface.

Another limitation of this pipeline is the reference project - Voting App. It is a small and limited project. Additional studies are needed in order to make pipeline more usable for larger projects. There is also a fairly new microservice architecture, which requires a completely different treatment by the pipeline. This issue is unresolved in this pipeline.

5.3 Project Challenges

There have been a set of challenges during the pipeline planning and implementation. The most important one is the pipeline complexity itself. Pipeline uses many different technologies from Docker to Jenkins. All of these technologies have their own specifics that needed to have been learnt before actual planning and implementation. There was a steep learning curve for the researcher of this thesis. Even when a familiarity with technologies was acquired, it was still challenging to put all this knowledge together into one solid pipeline.

Theoretical research had its difficulties as well. Definitions for CI, CD and DevOps, which serve as the starting point for the pipeline, are vague, and interpretation depends on the source. These methodologies are changing the software development processes, while changes in the processes affects the definition of these methodologies. It is an infinite loop, where there are constant changes. Therefore, for the most accurate and up to date definitions of CI, CD and DevOps, technical blogs of major companies working with the three, such as Ansible and Puppet, were used.

Best practices for using Docker or Rancher are also difficult to come by. These tools are less than a decade old, hence by the book practices simply do not exist. Furthermore, Docker ecosystem is changing at a very fast pace, which leads to additional challenges. An article from 2 years ago can be very far away from the implementation of today. This problem was resolved by trial and error, where different options from varying sources and own attempts were tried.

Jenkins, even though an established software, is changing all the time. The Jenkins Multibranch Pipeline that is used in this project was dramatically changed during the implementation of this project. First research on building of the pipeline was started by the researcher in late 2016. When a first draft of Jenkinsfile had already been ready, a new version of Pipeline Plugin was released. The released 2.5 version carried a whole set of changes implementing two new syntaxes for Jenkinsfile, which are declarative and scripted. Old syntax was not deprecated right away but Jenkins log was prompting messages informing that the old syntax will be deprecated in newer versions. This required a complete rewrite of Jenkinsfile and an additional learning.

Outlining this subsection, it is worth to mention that there was not a single challenge that would cause long-term delays. Taking into account that pipeline consists of many technologies, each of them individually is not that difficult. In addition to, the reference project is small and there are still hundreds of more features that the pipeline can implement. Further development, however, is the topic of the next subsection.

5.4 Further Development

Without changing the structure of a pipeline, a whole set of improvements could be conducted. First of all, a history of Docker images. As of now, Voting App images are always pushed with the *latest* tag. A history of images could be useful for rollbacks, for example. Currently this is not supported. Such history could be implemented with, for example, adding a git commit hash to the image tag.

An additional stage of post deployment tests could also be implemented. There are currently no means to automatically verify if the deployment was successful. For a real production environment this feature need to be present to ensure reliability of deployments. Even a few post-deployment smoke tests against the updated environment could largely decrease the chance from fatal failure going unnoticed after deployment.

Usually, highly automated environments, like the one presented in this thesis, require a solid layer of monitoring. Software monitoring is a large topic on its own and therefore would need to be researched to bring improvements to the pipeline. There dozens of available tools that can monitor the health of an application and report problems online so that they can be immediately fixed.

Rancher environment, even though quite reliable, as it was seen from a test when one host was artificially brought down, still contains a fatal flaw. There is a single point of failure on the host 2, which contains load-balancer and database. Eliminating this flaw would require additional research. Netflix, which has a very large infrastructure, even introduced a concept of “Chaos Monkey”, that would be randomly attacking different parts of the infrastructure trying to cause a fatal failure [51]. This case could be used as a reference for improving reliability of this pipeline.

Last major improvement that could be made is running the pipeline in an auto-scaling cluster. However, this will require remaking most of the pipeline and ultimately result in

a different project. Nevertheless, having a second template targeted for Amazon or Google cloud could be a worthwhile addition. Implementing such cluster without the cloud providers would be a costly venture as it would require a large number of servers and a lot of complex configuration. With cloud, however, such an implementation should be viable and, in the end, would, most likely, result in better efficiency and reliability. There is, of course, a disadvantage that such pipeline would require choosing a specific target cloud platform, which will make the pipeline less reliable. Nonetheless, the benefits may outweigh this, but this is a topic for additional research.

6 Conclusions

Designing and implementing a reusable DevOps pipeline that takes full advantage of the Docker ecosystem was the main goal of this thesis. During the project, a comprehensive research was conducted starting from the very root of this project, the concepts of CI, CD and CDt. The relation between those and DevOps as well as the origin of the DevOps itself were studied. It was followed by an overview of virtualization and Docker with its ecosystem as a representative of virtualization.

The final implementation fulfils the main goal of reusability. Some parts of the pipeline are 100% reusable, while others could have a reusability rate of up to 71%. Pipeline follows the DevOps culture of automation and collaboration. The structure of the pipeline is very rigid and specific providing a solid support for a reference project. Most of the configuration for the pipeline is stored in a clearly defined configuration files.

If the pipeline is to be used for real life projects, it could largely improve software development processes. Whether it is local, development or production environment, pipeline takes it all into account. Deployments could be as frequent as needed. Additionally, Rancher environment could be easily scaled up by adding more servers to reply to a growth in the application loads.

The resulting pipeline fulfils all the requirements defined within Requirements subsection. There are noticeable disadvantages, like an already mentioned single point of failure, as well as an overall application complexity. However, the benefits of pipeline flexibility and transparency outweigh these disadvantages. Pipeline could likely be simplified, and the single point of failure could be addressed and improved. Apart from these, there are other minor and major improvements that could increase pipeline viability such as post-deployment testing and auto-scaling pipeline in the infrastructure of the cloud provider.

Finally, DevOps and the technologies around it are all very young. There are new technologies and approaches appearing regularly. Therefore, there will always be a room for improvement and additional research to support this pipeline as a universal reusable template.

References

1. Booch G. Object Oriented Design: With Applications. Benjamin-Cummings Publishing Company; 1991. 209 p.
2. Smith S. Introducing Continuous Delivery - DZone DevOps [Internet]. dzone.com. 2014 [cited 2017 Aug 24]. Available from: <https://dzone.com/articles/introducing-continuous>
3. Avram A. Docker: Automated and Consistent Software Deployments [Internet]. InfoQ. [cited 2017 Mar 30]. Available from: <https://www.infoq.com/news/2013/03/Docker>
4. Arijis P. Docker Usage Stats: Adoption Up in the Enterprise and Production - DZone Cloud [Internet]. dzone.com. 2016 [cited 2017 Aug 24]. Available from: <https://dzone.com/articles/docker-usage-statistics-increased-adoption-by-ente>
5. Desjardins J. Infographic: How Many Millions of Lines of Code Does It Take? [Internet]. Visual Capitalist. 2017 [cited 2017 Aug 26]. Available from: <http://www.visualcapitalist.com/millions-lines-of-code/>
6. Ganesan V. Blameless Continuous Integration: A Small Step Towards Psychological Safety of Agile Teams. Partridge Publishing; 2017. 5 p.
7. Fowler M. Continuous Integration [Internet]. martinfowler.com. [cited 2017 Aug 27]. Available from: <https://martinfowler.com/articles/continuousIntegration.html>
8. Pecanac V. What is Continuous Integration and why do you need it? - Code Maze [Internet]. Code Maze. 2016 [cited 2017 Oct 8]. Available from: <https://www.code-maze.com/what-is-continuous-integration/>
9. About Jenkins [Internet]. CloudBees. 2014 [cited 2017 Aug 27]. Available from: <https://www.cloudbees.com/jenkins/about>
10. Atlassian. Bamboo - Continuous integration, deployment & release management | Atlassian [Internet]. Atlassian. [cited 2017 Aug 27]. Available from: <https://www.atlassian.com/software/bamboo>
11. TeamCity: Distributed Build Management and CI Server by JetBrains [Internet]. JetBrains. [cited 2017 Aug 27]. Available from: <https://www.jetbrains.com/teamcity/>
12. Maple S, Simon Mapledirector of, Advocate OS. Java Tools and Technologies Landscape Report 2016 | [Internet]. zereturnaround.com. 2016 [cited 2017 Aug 27]. Available from: <https://zereturnaround.com/rebellabs/java-tools-and-technologies-landscape-2016/>
13. Caum C. Continuous Delivery Vs. Continuous Deployment: What's the Diff? [Internet]. Puppet. [cited 2017 Aug 29]. Available from: <https://puppet.com/blog/continuous-delivery-vs-continuous-deployment-what-s-diff>
14. Butterly J. 8 Principles of Continuous Delivery [Internet]. DevOpsNet. 2011 [cited 2017 Oct 8]. Available from: <https://devopsnet.com/2011/08/04/continuous-delivery/>

15. van Bennekum Alistair Cockburn Ward Cunningham Martin Fowler James Grenning Jim Highsmith Andrew Hunt Ron Jeffries Jon Kern Brian Marick Robert C. Martin Steve Mellor Ken Schwaber Jeff Sutherland Dave Thomas KBMBA. Principles behind the Agile Manifesto [Internet]. 2001 [cited 2017 Aug 30]. Available from: <http://agilemanifesto.org/iso/en/principles.html>
16. Debois P. Agile Infrastructure & Operations [Internet]. 2008 [cited 2017 Aug 30]. Available from: <http://www.jedi.be/presentations/agile-infrastructure-agile-2008.pdf>
17. DevOps dictionary definition | DevOps defined [Internet]. [cited 2017 Aug 30]. Available from: <http://www.yourdictionary.com/devops>
18. Wilsenach R. bliki: DevOpsCulture [Internet]. martinowler.com. 2015 [cited 2017 Oct 8]. Available from: <https://martinowler.com/bliki/DevOpsCulture.html>
19. Yigal A. How DevOps is Killing QA - DevOps.com [Internet]. DevOps.com. 2016 [cited 2017 Oct 11]. Available from: <https://devops.com/devops-killed-qa/>
20. DevOps Growth and the Affect on Testing | SoapUI Testing Dojo [Internet]. Soapui. [cited 2017 Oct 12]. Available from: <https://www.soapui.org/testing-dojoworld-of-api-testing/dev-ops-trends.html>
21. Yehuda Y. What's the relationship between DevOps and Continuous Delivery? [Internet]. IT Pro Portal. 2015 [cited 2017 Oct 12]. Available from: <https://www.itproportal.com/2015/06/09/whats-relationship-between-devops-and-continuous-delivery/>
22. 2017 State of DevOps Report [Internet]. Puppet. 2017 [cited 2017 Oct 16]. Available from: <https://puppet.com/resources/whitepaper/state-of-devops-report>
23. Brodtkin J. With long history of virtualization behind it, IBM looks to the future [Internet]. Network World. 2009 [cited 2017 Oct 22]. Available from: <https://www.networkworld.com/article/2254433/virtualization/with-long-history-of-virtualization-behind-it--ibm-looks-to-the-future.html>
24. Garrison J. What Is a Virtual Machine Hypervisor? [Internet]. How-To Geek. 2016 [cited 2017 Oct 22]. Available from: <https://www.howtogeek.com/66734/htg-explains-what-is-a-hypervisor/>
25. Tholeti BP. Learn about hypervisors, system virtualization, and how it works in a cloud environment [Internet]. 2011 [cited 2017 Oct 22]. Available from: <https://www.ibm.com/developerworks/cloud/library/cl-hypervisorcompare/index.html>
26. Meier S. IBM Redbooks | IBM Systems Virtualization: Servers, Storage, and Software [Internet]. 2008 [cited 2017 Oct 22]. Available from: <http://www.redbooks.ibm.com/abstracts/redp4396.html?Open>
27. Bhupender. Types of Virtualization in Cloud Computing- An Overview [Internet]. ZNetLive Blog - A Guide to Domains, Web Hosting & Cloud. 2016 [cited 2017 Oct 22]. Available from: <https://www.znetlive.com/blog/virtualization-in-cloud-computing/>
28. Vaughan-Nichols SJ. Amazon EC2 cloud is made up of almost half-a-million Linux servers | ZDNet [Internet]. ZDNet. 2012 [cited 2017 Nov 12]. Available from: <http://www.zdnet.com/article/amazon-ec2-cloud-is-made-up-of-almost-half-a-million-linux-servers/>

29. Olson N. Tips for a Virtual Development Environment - Intertech Blog [Internet]. Intertech Blog. 2016 [cited 2017 Nov 12]. Available from: <https://www.intertech.com/Blog/tips-for-a-virtual-development-environment/>
30. Brockmeier J. Containers vs. Hypervisors: Choosing the Best Virtualization Technology [Internet]. Linux.com | The source for Linux information. 2010 [cited 2017 Oct 23]. Available from: <https://www.linux.com/news/containers-vs-hypervisors-choosing-best-virtualization-technology>
31. O'Reilly D. The Drawbacks of Running Containers on Bare Metal Servers [Internet]. Morpheus. Morpheus Data, LLC; 2017 [cited 2017 Oct 23]. Available from: <https://www.morpheusdata.com/blog/2017-04-28-the-drawbacks-of-running-containers-on-bare-metal-servers>
32. Strotmann J. Free Infographic: A history of #containerization technology! [Internet]. Plesk. Plesk; 2016 [cited 2017 Oct 23]. Available from: <https://www.plesk.com/blog/business-industry/infographic-brief-history-linux-containerization/>
33. Tozzi C. What's Docker's Market Share Today? Good Question - Container Journal [Internet]. Container Journal. 2017 [cited 2017 Oct 26]. Available from: <https://containerjournal.com/2017/06/01/whats-dockers-market-share-today-good-question/>
34. Currie A. How Many Public Images are there on Docker Hub? – Microscaling Systems – Medium [Internet]. Medium. Microscaling Systems; 2016 [cited 2017 Nov 12]. Available from: <https://medium.com/microscaling-systems/how-many-public-images-are-there-on-docker-hub-bcdd2f7d6100>
35. Containers and Orchestration Explained [Internet]. MongoDB. [cited 2017 Oct 26]. Available from: <https://www.mongodb.com/containers-and-orchestration-explained>
36. Chifor A. Container Orchestration with Kubernetes: An Overview [Internet]. Medium. Onfido Tech; 2017 [cited 2017 Oct 26]. Available from: <https://medium.com/onfido-tech/container-orchestration-with-kubernetes-an-overview-da1d39ff2f91>
37. Schroder C. What Makes Up a Kubernetes Cluster? [Internet]. Linux.com | The source for Linux information. 2017 [cited 2017 Oct 26]. Available from: <https://www.linux.com/news/learn/chapter/intro-to-kubernetes/2017/4/what-makes-kubernetes-cluster>
38. The Evolution of the Modern Software Supply Chain - The Docker Survey, 2016 [Internet]. Docker. 2016 [cited 2017 Oct 28]. Available from: <https://www.docker.com/survey-2016>
39. Google Container Engine Documentation | Container Engine | Google Cloud Platform [Internet]. Google Cloud Platform. [cited 2017 Oct 28]. Available from: <https://cloud.google.com/container-engine/docs/>
40. What is Amazon EC2 Container Service? - Amazon EC2 Container Service [Internet]. [cited 2017 Oct 28]. Available from: <http://docs.aws.amazon.com/AmazonECS/latest/developerguide/Welcome.html>
41. Oliver K. Unleash the Cattle: The Rancher Container Orchestration Platform Now Generally Available - The New Stack [Internet]. The New Stack. 2016 [cited 2017

- Oct 28]. Available from: <https://thenewstack.io/unleash-cattle-rancher-container-platform-reaches-general-availability/>
42. What is container management software? - Definition from WhatIs.com [Internet]. SearchITOperations. [cited 2017 Oct 28]. Available from: <http://searchitoperations.techtarget.com/definition/container-management-software>
 43. Quick Start Guide [Internet]. [cited 2017 Oct 28]. Available from: <http://rancher.com/docs/rancher/latest/en/quick-start-guide/>
 44. Docker glossary [Internet]. Docker Documentation. 2017 [cited 2017 Nov 2]. Available from: <https://docs.docker.com/glossary/>
 45. Dockerfile reference [Internet]. Docker Documentation. 2017 [cited 2017 Nov 28]. Available from: <https://docs.docker.com/engine/reference/builder/>
 46. Overview of Docker Compose [Internet]. Docker Documentation. 2017 [cited 2017 Nov 3]. Available from: <https://docs.docker.com/compose/overview/>
 47. Compose file version 3 reference [Internet]. Docker Documentation. 2017 [cited 2017 Nov 3]. Available from: <https://docs.docker.com/compose/compose-file/>
 48. Introduction - Vagrant by HashiCorp [Internet]. Vagrant by HashiCorp. [cited 2017 Nov 4]. Available from: <https://www.vagrantup.com/intro/index.html>
 49. How Ansible Works | Ansible.com [Internet]. [cited 2017 Nov 4]. Available from: <https://www.ansible.com/how-ansible-works>
 50. Using a Jenkinsfile [Internet]. Using a Jenkinsfile. [cited 2017 Nov 7]. Available from: <https://jenkins.io/doc/book/pipeline/jenkinsfile/index.html>
 51. Brodtkin J. Netflix attacks own network with “Chaos Monkey”—and now you can too [Internet]. Ars Technica. 2012 [cited 2017 Nov 16]. Available from: <https://arstechnica.com/information-technology/2012/07/netflix-attacks-own-network-with-chaos-monkey-and-now-you-can-too/>

Appendix 1. GitHub Web Links

Pipeline Teamplate - <https://github.com/allhaker/devops-pipeline-ansible-template>

Voting App - <https://github.com/allhaker/devops-pipeline-demo-project>

Appendix 2. Voting App Interface Screenshots

New Project

Vote for your three favorites

Test Ranchering

john

Gitviz

Joe, donald

5

Saatiobot

kostya

Pydamsa

ivan

3

Watson

emma

#thankssoftware

brad

Good Enough Auction

ahmed, ll

1

Triviabot

nastya, alex, tapio

New Test Project

The Test Team

Results List

Gitviz joe, donald	5
Pydamsa ivan	3
Good Enough Auction ahmed, li	1
Watson emma	0
#thankssoftware brad	0
New Test Project The Test Team	0
Test Ranchering john	0
Saatiobot kostya	0
Triviabot nastya, alex, tapio	0

Appendix 3. Rancher docker-compose.dev.yml.

```

version: '2'
services:
  balancer:
    image: rancher/lb-service-haproxy:v0.7.9
    ports:
      - 80:80/tcp
    labels:
      io.rancher.container.agent.role: environmentAdmin
      io.rancher.container.create_agent: 'true'
      io.rancher.scheduler.affinity:host_label: balancer=true
  seed-db:
    image: allhaker/votingapp_backend:development
    environment:
      DATABASE_URL: postgres://votingapp:votingapp@db/votingapp
    links:
      - db:db
    command:
      - node
      - db/seeds.js
    labels:
      io.rancher.container.start_once: 'true'
  backend:
    image: allhaker/votingapp_backend:development
    environment:
      DATABASE_URL: postgres://votingapp:votingapp@db/votingapp
    links:
      - db:db
    labels:
      io.rancher.container.pull_image: always
      io.rancher.scheduler.global: 'true'
      service: backend
  frontend:
    image: allhaker/votingapp_frontend:development
    command:
      - npm
      - run
      - serve
    labels:
      io.rancher.container.pull_image: always
      io.rancher.scheduler.global: 'true'
      service: backend
  db:

```

```
image: postgres:9.4
environment:
  POSTGRES_DB: votingapp
  POSTGRES_PASSWORD: votingapp
  POSTGRES_USER: votingapp
volumes:
- /opt/docker/votingapp_data:/var/lib/postgresql/data
labels:
  io.rancher.scheduler.affinity:host_label: Database=true
```

Appendix 4. Rancher rancher-compose.dev.yml


```
version: '2'
services:
  balancer:
    scale: 1
    start_on_create: true
  lb_config:
    certs: []
    port_rules:
      - path: /api/v1
        priority: 1
        protocol: http
        service: backend
        source_port: 80
        target_port: 8080
      - path: /
        priority: 2
        protocol: http
        service: frontend
        source_port: 80
        target_port: 3000
    health_check:
      healthy_threshold: 2
      response_timeout: 2000
      port: 42
      unhealthy_threshold: 3
      interval: 2000
      strategy: recreate
  seed-db:
    scale: 1
    start_on_create: true
  backend:
    start_on_create: true
    health_check:
      healthy_threshold: 2
      response_timeout: 2000
      port: 8080
      unhealthy_threshold: 3
      initializing_timeout: 20000
      interval: 2000
      strategy: recreate
      reinitializing_timeout: 60000
  frontend:
```

```
start_on_create: true
health_check:
  healthy_threshold: 2
  response_timeout: 2000
  port: 3000
  unhealthy_threshold: 3
  initializing_timeout: 10000
  interval: 2000
  strategy: recreate
  reinitializing_timeout: 60000
db:
  scale: 1
  start_on_create: true
```

Appendix 5. Rancher load balancer UI.

Edit Load Balancer ⓘ

Scale

1 

Name

Description

Port Rules ⊕ Add Service Rule ⊕ Add Selector Rule

Access*	Protocol*	Request Host	Port*	Path	Target*	Port*
<input type="text" value="Public"/>	<input type="text" value="HTTP"/>	<input type="text" value="e.g. example.cor"/>	<input type="text" value="80"/>	<input type="text" value="/api/v1"/>	<input type="text" value="backend"/>	<input type="text" value="8080"/> <input type="button" value="−"/>
<input type="text" value="Public"/>	<input type="text" value="HTTP"/>	<input type="text" value="e.g. example.cor"/>	<input type="text" value="80"/>	<input type="text" value="/"/>	<input type="text" value="frontend"/>	<input type="text" value="3000"/> <input type="button" value="−"/>

Host and Path rules are matched top-to-bottom in the order shown. Backends will be named randomly by default; to customize the generated backends, provide a name and then refer to that in the custom haproxy.cfg. [Show custom backend names.](#) [Show host IP address options.](#)

SSL Termination
Stickiness
Custom haproxy.cfg
Labels
Scheduling

There are no SSL/TLS ports configured.

Appendix 6. Add Backend Webhook.


Name*

Kind

Image Tag*

Only registry pushes to the given tag will cause a service upgrade.

Service Selector*

 Add Selector Label

Key		Value
<input type="text" value="service"/>	=	<input type="text" value="backend"/>

ProTip: Paste one or more lines of key=value pairs into any key field for easy bulk entry.

Services matching the given label will be upgraded

Batch Size

Batch Interval

 sec

Start Behavior

Start before Stopping

Appendix 7. Unlock Jenkins.

Getting Started

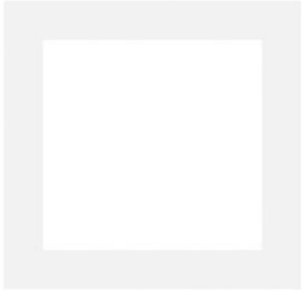
Unlock Jenkins

To ensure Jenkins is securely set up by the administrator, a password has been written to the log ([not sure where to find it?](#)) and this file on the server:

```
/var/lib/jenkins/secrets/initialAdminPassword
```

Please copy the password from either location and paste it below.

Administrator password



[Continue](#)

Appendix 8. Select Jenkins Plugins.

Getting Started ✕

Customize Jenkins

Plugins extend Jenkins with additional features to support many different needs.

Install suggested plugins

Install plugins the Jenkins community finds most useful.

Select plugins to install

Select and install plugins most suitable for your needs.




Jenkins 2.79


Appendix 9. Selecting Jenkins job type.

Enter an item name


voting-app




Freestyle project
This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any build system, and this can be even used for something other than software build.




Pipeline
Orchestrates long-running activities that can span multiple build slaves. Suitable for building pipelines (formerly known as workflows) and/or organizing complex activities that do not easily fit in free-style job type.




Multi-configuration project
Suitable for projects that need a large number of different configurations, such as testing on multiple environments, platform-specific builds, etc.



Folder
Creates a container that stores nested items in it. Useful for grouping things together. Unlike view, which is just a filter, a folder creates a separate namespace, so you can have multiple things of the same name as long as they are in different folders.




GitHub Organization
Scans a GitHub organization (or user account) for all repositories matching some defined markers.



Multibranch Pipeline
Creates a set of Pipeline projects according to detected branches in one SCM repository.

if you want to create a new item from other existing, you can use this option:



Copy from

OK

Appendix 10. Adding DockerHub credentials to Jenkins.

 [Back to Global credentials \(unrestricted\)](#)

 **Update**


 **Delete**

 **Move**

Scope 

Username 

Password 

ID 

Description 

Save

Appendix 11. Final Jenkinsfile.

```

def compose = "docker-compose -f docker-compose.yml -f docker-compose.test.yml -p
votingapp"

pipeline {
  agent any
  options {
    timeout(time: 20, unit: 'MINUTES')
  }

  stages {
    stage('Checkout') {
      steps {
        checkout scm
      }
    }

    stage('Build') {
      steps {
        sh "${compose} build --pull"
      }
    }

    stage('Test Backend (Mocha)') {
      steps {
        sh "${compose} run mocha"
      }
    }

    stage('Tag and Push Docker Images to DockerHub') {
      when {
        expression { env.BRANCH_NAME == 'master' }
      }

      steps {
        withCredentials([[ $class: 'UsernamePasswordMultiBinding', credentialsId:
'docker-login',
        usernameVariable: 'USERNAME', passwordVariable: 'PASSWORD']]) {
          // Login to Docker Registry
          sh "docker login -u ${USERNAME} -p ${PASSWORD}"
        }

        // Tag Docker Images
        sh "docker tag votingapp_frontend allhaker/votingapp_frontend"
        sh "docker tag votingapp_backend allhaker/votingapp_backend"

        // Push Docker Images
        sh "docker push allhaker/votingapp_frontend"
        sh "docker push allhaker/votingapp_backend"

        // Logout from Docker Registry
        sh "docker logout"
      }
    }

    stage('Upgrade Rancher Environment') {
      when {
        expression { env.BRANCH_NAME == 'master' }
      }

      steps {
        script {
          env.rancherHeaders = '-H "Content-Type: application/json" -X POST'
        }
      }
    }
  }
}

```

```

        env.rancherJSONFrontend =
'{"push_data":{"tag":"development"},"repository":{"repo_name":"allhaker/votingapp_fro
ntend"}}'
        env.rancherURLFrontend = 'http://192.168.50.4:8080/v1-
webhooks/endpoint?key=APIKEY '

        env.rancherJSONBackend =
'{"push_data":{"tag":"development"},"repository":{"repo_name":"allhaker/votingapp_bac
kend"}}'
        env.rancherURLBackend = 'http://192.168.50.4:8080/v1-
webhooks/endpoint?key=APIKEY '
    }

    sh "curl ${env.rancherHeaders} -d '${env.rancherJSONFrontend}'
'${env.rancherURLFrontend}'"
    sh "curl ${env.rancherHeaders} -d '${env.rancherJSONBackend}'
'${env.rancherURLBackend}'"
    }
}

post {
    always {
        sh "${compose} down -v"
    }
}
}

```