



Bachelor Degree Project

Container orchestration

- the migration path to Kubernetes



Authors: Johan Andersson & Fredrik
Norrmann
Supervisor: Jesper Andersson
External Supervisors: Ulrik Sjölin &
Daniel Ohlsson
Semester: VT/HT 2020
Subject: Computer Science

Abstract

As IT platforms grow larger and more complex, so does the underlying infrastructure. Virtualization is an essential factor for more efficient resource allocation, improving both the management and environmental impact. It allows more robust solutions and facilitates the use of IaC (infrastructure as code). Many systems developed today consist of containerized microservices. Considered the standard of container orchestration, Kubernetes is the natural next step for many companies. But how do we move on from previous solutions to a Kubernetes cluster? We found that there are not a lot of detailed enough guidelines available, and set out to gain more knowledge by diving into the subject - implementing prototypes that would act as a foundation for a resulting guideline of how it can be done.

Keywords: virtualization, container, Docker, Docker Compose, container orchestration, Kubernetes, Ansible, Terraform, Internet-of-Things, deployment, scaling, migration patterns, Rancher, Helm, IaC (infrastructure as code)

Preface

This project has acted as a valuable and rich experience, letting us delve deeper into such knowledge that we previously only grazed the surface of.

We would like to extend a big thank you to Jesper Andersson, our supervisor during this thesis work, for guidance with mainly the formulation of this report and also the direction of the subject.

We also want to thank Sensative AB immensely for the opportunity to do this work in collaboration with them. Special thanks to Daniel Ohlsson - acting as an external supervisor at the company - for the vast amounts of guidance, support, and insight he has provided.

Contents

1 Introduction	5
1.1 Background	6
1.2 Related work	7
1.3 Problem formulation	8
1.4 Objectives	9
1.5 Scope/Limitation	10
1.6 Target group	10
1.7 Outline	11
2 Method	12
2.1 Problem Identification & Motivation	12
2.2 Scientific methods	12
2.2.1 Document studies - O1, O2, O3	12
2.2.2 Case study - O1, O2, O3	12
2.2.3 Prototyping - O3, O4, O5, O6	13
2.2.4 Interviews and Discussions - O1-O7	13
2.2.5 Compilation of Guide - O7	13
2.3 Reliability and Validity	13
2.4 Ethical Considerations	13
2.5 Data management policy	14
3 Virtualization & Orchestration	15
3.1 Virtualization	15
3.1.1 Hardware virtualization	15
3.1.2 Hypervisor	15
3.1.3 Container-based virtualization	17
3.2 Container Orchestration	17
3.3 Docker	18
3.3.1 Docker Compose	18
3.3.2 Docker SWARM	18
3.4 Kubernetes	18
3.4.1 Nodes - Masters and workers	19
3.4.2 Services & Pods	20
3.4.3 Namespaces & Network Policies	21
3.4.4 Ingress & Loadbalancers	21

3.4.5 Kubectl, yaml, and Configurations	22
4 Implementation - experimental environment	23
4.1 Underlying Infrastructure	23
4.1.1 OpenStack	23
4.1.2 Terraform	23
4.1.3 Ansible	23
4.2 Kubernetes Installation	24
4.2.1 RKE	24
4.2.2 Plugins	25
4.3 First prototype	26
4.3.1 NGINX-Ingress and cert-manager	26
4.3.2 kubectl and yaml-configurations	27
4.4 Namespaces and NetworkPolicies	28
4.5 Second prototype and Helm charts	30
4.5.1 Helm and charts	30
4.5.2 ReplicaSet	34
4.5.3 StatefulSet	34
4.5.4 External LoadBalancer	35
5 Results & Analysis	36
5.1 Underlying infrastructure and provisioning	36
5.1.1 OpenStack	36
5.1.2 Terraform	36
5.1.3 Ansible	37
5.2 Kubernetes Installation	37
5.2.1 RKE	37
5.2.2 Nodes	38
5.2.3 Additional configurations	38
5.3 Useful plugins	38
5.3.1 OpenStack Cloud Controller Manager	38
5.3.2 CSI Cinder plugin	39
5.3.3 Kubernetes Dashboard	39
5.4 Ingress and external load balancers	39
5.5 Namespace independency	40
5.6 Application configurations	40

5.7 Dealing with redundancy	40
5.7.1 Helm charts	40
5.8 Analysis	41
5.8.1 Resources	41
5.8.2 Deployment	41
5.8.3 Maintenance	42
6 Conclusions and Future Work	43
6.1 Conclusions	43
6.1.1 Reflections on possible improvements	43
6.1.2 Benefits of the results	44
6.2 Future possibilities	45
6.3 Further work	45
References	47

1 Introduction

This chapter covers a broad overview and introduction to our work, and the area it covers - container orchestration and migration patterns.

In the modern world, demands on IT infrastructure have snowballed. With more and more technologies presented and incorporated in a vast number of devices, ranging from your everyday computer and cellphone to new appliances such as fridges, toasters, and windows, it is not hard to see why this is the case.

These present and future demands have led to a need for new solutions for both deploying and managing the infrastructure of IT platforms. Through packaging the system's components in so-called containers, deployment can be streamlined. Instead of installing a service directly on the server, we run a container containing the system itself and its dependencies. But how do we manage all these containers? This is where orchestration tools like Docker Compose and Kubernetes come into play, allowing us to deploy and manage our containers in an efficient way.

1.1 Background

Some basic concepts need to be understood before venturing further into this project.

Virtualization is a technique used to utilize existing hardware more effectively. In short, it translates to putting a layer on top of the hardware and dividing this into smaller independent processes that emulate hardware, such as servers. For the end-user, it still appears like a fully functional hardware (server). To deploy and manage several linked virtual machines in large and dynamic environments, orchestration techniques such as Docker and Kubernetes are frequently used.

We have an interest in present and future solutions for infrastructure deployment, especially the use of containerization, container orchestration, and automation to minimize manual labor - one of our passions. Therefore, we want to delve deeper into this, gaining more knowledge about the technique itself. Considering Kubernetes is basically the industry standard today (or at least, perceived as it), this also leads to an interest in how we can migrate existing container orchestration to it.

While researching potential projects, we contacted Sensative AB [1]. Sensative develops IoT solutions such as sensors for windows and sinks. This, in conjunction with their platform called Yggio [2], acts as an integration layer between these devices and different services. During our

discussions, they expressed a need for more dynamic and scalable infrastructure.

Today the deployment is done via Docker Compose [3], but the company expresses a desire for migration to a Kubernetes-based solution. This demand stems from a growing business with an increasing number of customers. The growth has led to the deployment of more instances of the platform and a lack of overview as well as management difficulties.

This opportunity was used as the main case for the project, where we were able to implement prototypes of solutions in close collaboration with the company.

As a starting point for this project, we need to look at what has been done before - if there is anything available related to what we want to achieve. This information could be presented in articles, papers, tutorials, guides, or similar.

1.2 Related work

After extensive research we were still unsuccessful in finding published articles that are entirely related to the exact subject we are working with - migration from Docker to Kubernetes and the paths to take. We did however manage to find some articles about the technicalities around Docker as well as Kubernetes as a whole, and what improvements the latter provides.

However, browsing online articles and blogs, the information tends to be more vast closer to our niched subject.

It is clear to see what positive changes a migration to Kubernetes can bring, ranging from easier management of the infrastructure - e.g., scalability possibilities - to actual hardware performance and smaller costs due to less load. Previous research shows that Kubernetes can provide significantly reduced workloads compared to solutions using Docker, similar containerized solutions, or non-containerized solutions. This is in conjunction with the ease of scalability and overall management. [4]

Orchestration of containers in Docker can be done in a couple of different ways, namely using Compose [3] or SWARM [5]. The former provides a simple way to run a multi-container application on a single host, while SWARM can be used to scale it across one or more servers. Compose is the simple “plug and play” alternative, but lacks a lot of the management and scalability options [6]. While SWARM, on the other hand, is a bonafide orchestration tool, compared to Kubernetes, it still offers limited functionality and fault tolerance. The scaling is more of a manual nature. While many today strive to automate as much as possible, the manual labor required for

things such as scaling is a significant drawback for SWARM.

Kubernetes, having the most functionality, is also the more complex of the tools. The installation and setup process can be quite complex and involve a steep learning curve compared to SWARM, which has an easy and fast setup. However, the community around Kubernetes is huge, and the Cloud Native Computing Foundation backs it, all the while being an open-source tool [7][8].

To circumvent, or at least ease, the complexity of Kubernetes, there are a lot of useful tools that can be used for different steps of the process, provided by both Kubernetes itself and other parties. One popular example is Rancher [9] and its rke [10] tool, which helps with the initial installation and configuration of a cluster.

The information we could find just touched the surface of our goals and handled either too broad or too narrow aspects of the problem - often with very niche tools, problems and solutions presented. A large amount of knowledge needed is not presented adequately.

Through this project and the insight we will gain, we hope to be able to provide useful information about what challenges need to be overcome and what the solutions can look like for future migrations. This should probably be of great interest to the developer community as a whole, as there are some examples and tutorials online but not as complete or in-depth as one may want.

1.3 Problem formulation

So, how do we move even further forward in our journey to secure, as well as streamline, both the deployment and management of modern-day IT platforms? Scalability is also of great concern as these platforms grow, and at the same time, see clear patterns in usage and downtime - where specific time periods can result in big traffic spikes potentially causing bottlenecks & slowdowns. While other time slots are much less occupied, we clearly see a need not only to invest in more resources - but also to downscale these resources, enabling a more cost-efficient infrastructure. In addition to this, there is also the problem of failures, where restarts and other management of the platform are needed. Preferably, this would all be done in a mostly automated way to minimize both downtime of the service as well as needed manual labor.

For companies already running container orchestration via Docker, the

next logical step for development would be Kubernetes, as it provides more advanced automation, like mass deployments, redundancy, automatic restart of falling containers, auto-scaling and load balancing. Kubernetes is nowadays commonly perceived as the industry standard for container orchestration [11].

From what we have found, there is a lack of properly extensive documentation for Kubernetes in the way of guides/tutorials, templates, and examples on how to set up and configure it in a good way. Every Kubernetes cluster is unique in the way it needs to be configured, depending on how and for what it is used.

This poses the following questions and problems:

- How does one move from Docker Compose to Kubernetes - what could a possible migration path look like? - *Method or means of development*
- What range of tools is available to help with the setup - what impact could different techniques/tools have on the migration? - *Design, evaluation, or analysis of a particular instance*
- What challenges and corresponding solutions can be found? - *Method for analysis; Generalization or characterization; Feasibility*

These questions make up the sum of what we want to try to achieve with this project: making a simple guide/tutorial/example of how an existing software, already consisting of containerized services, can be deployed in Kubernetes - and how to set up and configure the cluster. In addition to this making a shallow comparison with other possible solutions, such as SWARM.

1.4 Objectives

01	Research about possible migration patterns/solutions and techniques - <i>Procedure or technique</i>
02	Compile a list of possible solutions and techniques needed for prototyping
03	Implement small scale prototype involving the deployment of parts of the infrastructure - <i>Specific solution</i>
04	Analyze and evaluate the impact/result of prototype
05	Implement larger scale prototype involving the deployment of the infrastructure - <i>Specific solution</i>
06	Analyze and evaluate the impact/result of the updated prototype - <i>Answer or judgement</i>

O7	Complete guide/example with results from prototyping - <i>Report</i>
-----------	---

What we expect to get as a result of this project is a presentation of examples of the different challenges one might encounter when migrating IT infrastructure from Docker Compose to Kubernetes and what solutions can be found for these challenges, respectively. With these results, we want to compile a small guide of example configurations that would hopefully prove useful in future similar projects.

Another expectation is that we will be able to implement a sufficient, working prototype for our solution that can act as a basis for continued work.

Our personal thoughts about this are that there are many different paths to take to reach the end result, not one be-all-end-all solution for everyone, but rather different solutions that can act as the “best one” per each case. What we hope to see with this report is that it can act as a starting ground in trying to find the most fitting solution for specific scenarios.

1.5 Scope/Limitation

This project can, in theory, hold limitless potential regarding the scope since there are many different migration paths and techniques to choose between and utilize. However, since our resources, especially time, are of a limited nature, this means that some priorities and limitations have to be set.

The main focus of the end result will be a simple functioning prototype, in addition to the aforementioned presentation of migration patterns, their challenges, and their own solutions. The prototyping, together with the configuration guide will in themselves also require a somewhat limited scope, as we will not be able to explore all the available options. The decision has been made to mainly focus on Docker Compose as a basis of migration, with the possible addition of SWARM if time allows. Down the line, this means that we will be able to present one or two possible solutions that we can invest more time into, which hopefully will lead to a more detailed end result.

1.6 Target group

On a large scale, the main target group would be SME and other businesses that want to migrate their infrastructure to Kubernetes, provided that they already use Docker - as the migration from Docker to Kubernetes is the subject of this report.

The other target group would be the company Sensative itself, as in conjunction with the studies of this report, we will try to find the basis of a solution for their case.

Finally, as mentioned in the motivation segment, the developer community as a whole could be considered a target group - however, this could be argued to be linked with the main target group.

1.7 Outline

The remainder of this report is structured as follows. Section 2 describes the method(s) used for the project, how we answer and solve the problem(s) of the project, and how we value the reliability of our solution. Section 3 provides a deep dive describing the different techniques involved in the project more extensively. Section 4 describes the implementation of our solution. In section 5, the objective results achieved through our work are presented, as well as subjective opinions and conclusions drawn from the result. Lastly, the report is concluded in section 6 and presents whether our results are relevant to science, industry, or society in the future. It also presents some possible future work to be done.

2 Method

The project will be done in several somewhat incremental steps. As an entry point, information needs to be gathered about virtualization, container orchestration, and specifically Kubernetes - as well as Docker. In combination with case studies and discussions with the company, the main task is to implement prototypes and conduct experiments with these. The final result is projected to be a simple implementation guide.

2.1 Problem Identification & Motivation

This project's main method will consist of a vast number of controlled experiments, mainly by implementing different prototypes. In addition to this, there may be a smaller number of interviews and discussions with the developers at hand. The experiments will be conducted on the prototypes being developed during the span of the project, and during the process of implementing them. Additional techniques to be used in comparisons are studied through case studies.

Information about challenges, errors, and linked solutions is to be gathered and compiled in a guide. The idea is for this to be a guide of examples in the end, but mainly a presentation of one or two ways the migration can be done is what we strive for.

We think this is the best way to reach the goals we set and answer the questions, as we need hands-on experience with Kubernetes to get to know it properly and be able to produce a decent enough guide.

2.2 Scientific methods

To reach the goals/objectives we set - analyzing the company's needs and expectations, gathering enough information to be able to provide a sufficient guideline - the following scientific methods are needed.

2.2.1 Document studies - O1, O2, O3

The central part of the initial work, but something that has also accompanied the work through the whole project, is gathering and analyzing existing data. This data has to the vast majority consisted of official documentation regarding Kubernetes and the associated techniques. This has laid the basis for the experiments and prototypes.

2.2.2 Case study - O1, O2, O3

To be able to make comparisons to other possible techniques than

Kubernetes, we have conducted a case study to gather information about them rather than doing additional prototyping and experiments as this does not fit in the scope of the project. The case study is compiled using official documentation and possible guides/tutorials.

2.2.3 Prototyping - O3, O4, O5, O6

As the most significant part of the project, we have experimented a lot with prototyping different installations/setups and configurations of a Kubernetes cluster, using several different tools and techniques available. These prototypes and experiments have given us more knowledge about the possibilities of different solutions and how it can be done.

2.2.4 Interviews and Discussions - O1-O7

Throughout the project, we have had continuous informal interviews and discussions with the company, mainly with our external supervisor Daniel Ohlsson. This has been done to share ideas and gather requirements for the solution, the goals and plans of the company, what they are looking for.

2.2.5 Compilation of Guide - O7

As a result of all the data gathering, prototyping and experiments, we conduct a small guide mainly consisting of the steps we have taken and the solutions we have used, as well as suggestions of other possible solutions that we have found. Here we also motivate and discuss the different techniques and tools we used.

2.3 Reliability and Validity

Since we expect there to be a number of different ways to conduct a migration of this kind, and not a single one set in stone as the most correct one, we cannot guarantee that the resulting solution will be the exact same or even the most fitting for everyone. However, as stated, our goal is not to find the “be-all-end-all” solution but rather to provide a window into *how*, *why*, and *when* it can be done. We want to describe what the challenges can be and give examples of how they can be overcome. Simply put, give an example solution that can be used as a foundation and inspiration for future projects. The reliability and validity of this is something we think will be of no major concern.

2.4 Ethical Considerations

As this project has mainly consisted of the development and experimentation

of implementing a number of prototypes, we do not have any direct ethical considerations. The discussions and interviews with representatives from the company, Sensative, do not bring any privacy concerns about anonymity or the likes. The employees in question are the external supervisors mentioned in the report.

2.5 Data management policy

As Yggio is not an open source platform and the repository is not public, we are not allowed to disclose any of the configuration files directly, other than the examples provided throughout the report. We are therefore not able to provide any links to files or repositories.

3 Virtualization & Orchestration

To better understand the project as a whole, the main subject, and the techniques involved, we need to dive a bit deeper and explain some basic concepts.

3.1 Virtualization

Virtualization is the act of creating virtual versions of something [12] - we tend to think about it as hardware virtualization, but there are many different types of virtualization techniques. The main goal of virtualization is a more effective and streamlined use of resources, as well as easier management of large systems.

In this segment, we will dive deeper into the different virtualization techniques available today.

3.1.1 Hardware virtualization

Hardware virtualization, or platform virtualization, refers to the creation of virtual machines that acts just like a real computer with its own operating system [12]. All hardware is virtualized by software that we call a hypervisor.

Because all hardware is virtualized, we can freely choose what operating system we want to run, as well as specifying dedicated processing power, RAM, and the likes for each machine.

This is the most versatile and straight forward way of using virtualization; however, it also brings with it a lot of overhead.

3.1.2 Hypervisor

As mentioned above, when using hardware virtualization, a software called hypervisor is needed for running the virtualization layer. Furthermore, there are two types of hypervisors [13] available.

The type 1 hypervisor, as depicted in Figure 3.1, runs directly on the system hardware [13]. Famous examples of this kind of hypervisor are VMware ESX and Microsoft Hyper-V.

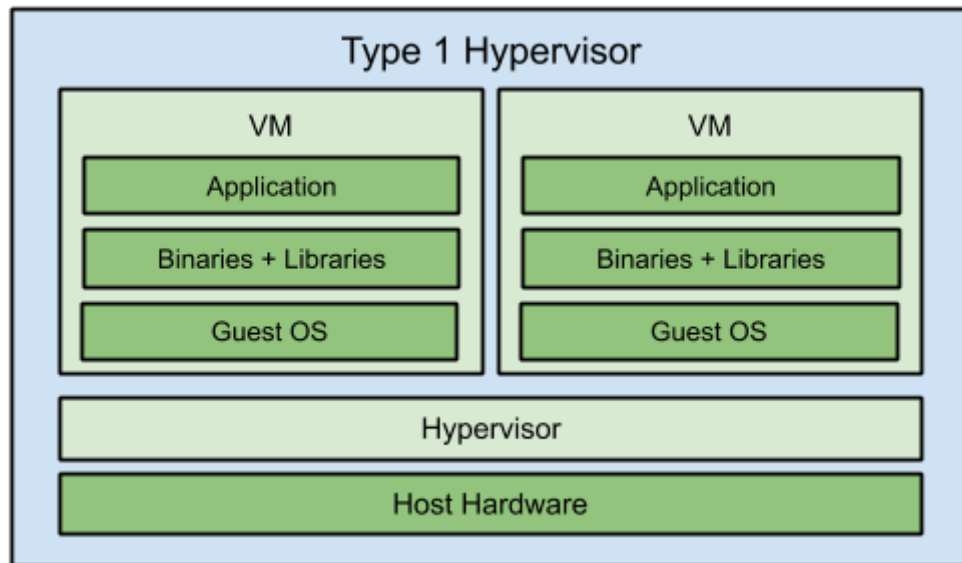


Figure 3.1 - Overview of a Type 1 Hypervisor

The other kind of hypervisor, simply called type 2 hypervisor, is running on a host operating system [13] that provides virtualization services. This can be I/O devices and memory management. Examples of the type 2 hypervisor, shown in Figure 3.2, are KVM, Oracle VM VirtualBox, and VMWare Workstation.

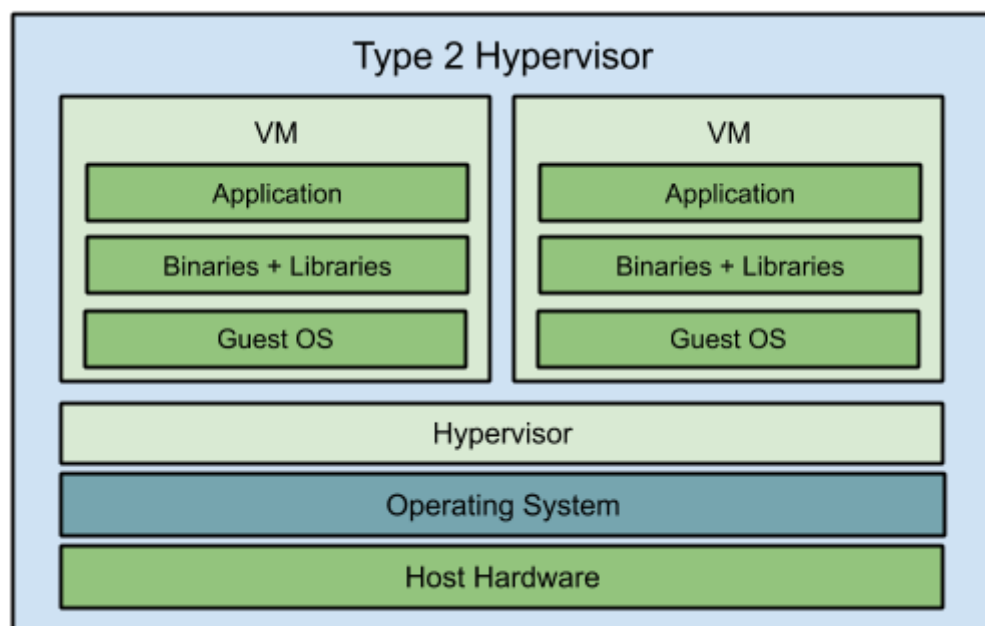


Figure 3.2 - Overview of a Type 2 Hypervisor

3.1.3 Container-based virtualization

A growing technique for deploying applications and software systems today, microservices translates to splitting up a software into different smaller services [14] that are then dependent on each other. These so-called microservices can then be run independent of each other in standalone environments - so-called containers [15]. Using this technique allows for easier scaling [14] as well as updates of individual parts without affecting the larger service as a whole.

Container-based virtualization (Figure 3.3) is a very effective [16] virtualization technique and has a lot less overhead compared to hardware virtualization. It is, however, less flexible, as the core is shared with the host; therefore, OS choice is non-existent. This can be circumvented by so-called nested virtualization [12], where the container engine is run on top of another virtualization layer.

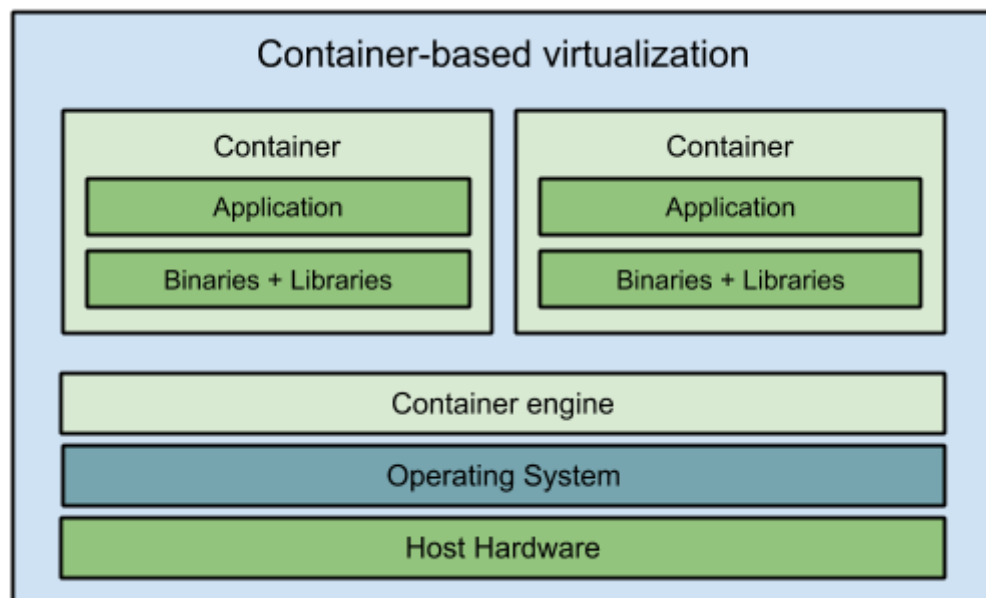


Figure 3.3 - Overview of container-based virtualization

3.2 Container Orchestration

Container orchestration is the process of automating all aspects of container coordination and management, with a particular focus on the management of container lifecycles and their dynamic environments [17]. It enables controlling and automating a number of tasks, such as the provisioning and deployment of the containers, load balancing, redundancy and availability, allocation of resources and much more.

3.3 Docker

Docker is a tool that provides the functionality to package software in host-independent containers [15] that use OS-level virtualization (container-based virtualization). These containers are, as mentioned, independent from both the host machine itself, and each other, and bundles their own software with its dependencies. Through defined channels, they can also communicate with each other.

The containers are configured in yaml-files, which can be saved as images. These images can then be saved in repositories and used as base images [18] - starting points - for new containers/images. One example is the standard Nginx Docker image, which can be used as a foundation for a container that will run with Nginx.

3.3.1 Docker Compose

Docker Compose provides a way of orchestrating multiple containers [3] dependent on each other. Examples include a service dependent on a database, a service using some kind of caching. This dependency makes the service unusable on its own - and it has to be managed in some way.

It is a standalone binary [19] that will run a multi-container environment on a single Docker host. Docker Compose can also be beneficial to use in single-container scenarios. This way, we get the possibility to store configuration settings, such as environment variables, in the docker-compose yaml file.

3.3.2 Docker SWARM

Docker SWARM is built into the Docker CLI and is used to scale a multi-container application across one or more servers; in other words, native clustering functionality [5].

SWARM enables cluster deployment using standard docker commands, and thanks to this, it is relatively simple to migrate to Docker SWARM from a regular Docker deployment. It also supports using a docker-compose configuration to deploy to the cluster. Docker SWARM only has one [20] type of node that can act as manager, leader, and worker.

3.4 Kubernetes

The open-source project Kubernetes, or k8s, was initially designed by Google and is now maintained by the Cloud Native Computing Foundation [21]. It is a powerful and, in some ways, quite a complex container orchestration system that allows automation of application deployment, scaling, and

management [21]. It works with Docker, amongst other container tools, and is offered as a PaaS (Platform as a Service) or SaaS (Software as a Service) on most cloud services.

Compared to Docker SWARM, Kubernetes offers a lot more and more complex, functionalities, and possibilities. The complexity demands more of the infrastructure and user, as it involves a lot more configuration and options.

The foundation of Kubernetes consists of two types [22] of nodes - masters and workers.

3.4.1 Nodes - Masters and workers

The aforementioned types of nodes in Kubernetes - masters and workers - have totally different duties in the cluster. Together these nodes form what is called a cluster (see Figure 3.4).

The master node is responsible for maintaining the desired state [23] of the cluster, and all interactions with the cluster are piped via a master node. The master contains the *etcd* [22] service, which is a database with the task of keeping track of the state of the cluster. It also contains the *API-server* [22], which is used to communicate with the cluster, a *scheduler*, and the *controller-manager* [23]. The latter two keep the cluster in the desired state and schedule new pods (containers) if needed. To achieve high availability, it is possible to run multiple master nodes in a cluster. All master nodes are then replicated [23], eliminating a single point of failure.

On the other hand, worker nodes are responsible for running the pods (containers) [22] in the desired state. All worker nodes run a service called *kubelet* [24], which communicates with the master node(s), receives pod specs, and ensures that they are running in the desired state. It also ensures that the pods are healthy and recreates them if needed.

The worker nodes also run a *proxy service* [24] responsible for the network rules on the nodes. This *proxy service* enables network communication to and from pods - both inside the cluster and to the outside world.

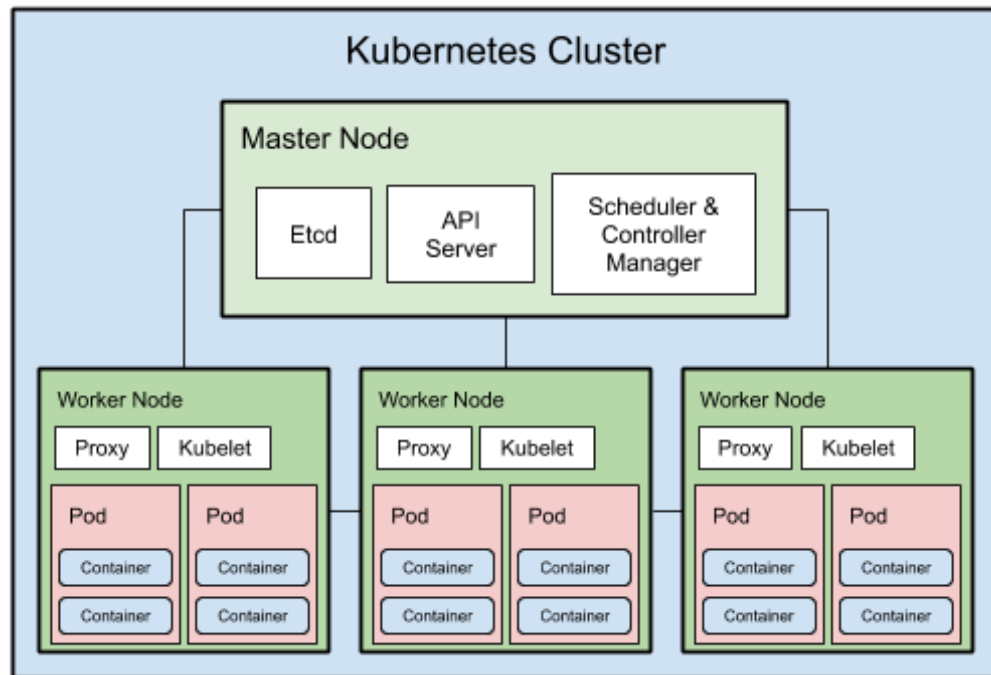


Figure 3.4 - Master and worker nodes in a Kubernetes cluster

3.4.2 Services & Pods

A Kubernetes service [25] is the resource that acts as an entry point to an application. In this resource, the allowed communication and the type of service are specified.

In Kubernetes, there are several types [25] of services that specify the behavior of the network communication. The most common is the cluster-IP [25] service, this service delivers a single internal static IP for traffic to the pod(s) it serves. It also provides internal load balancing if it serves more than one pod. The cluster-IP can also be configured as a headless service [25]. In this form, the service only provides a list of IP-addresses to the serving pod(s). Cluster-IP only allows internal cluster communication. To expose an application to the outside world, services such as ingress [26] and load balancer are needed. See section 3.4.4 for more information.

The applications themselves are run on one or more pods. If multiple pods are configured to be run for one application, they will be run as so-called replicas [27]. These replicas act as multiple, simultaneously running copies of the application.

With stateless microservices, specifying the number of desired replicas - paired with a service - easily enables a load-balanced application without a single point of failure inside the application itself (see Figure 3.5). If one replica - pod - fails, the others will still make sure that the application continues to be reachable, all the while the failing part will be recreated by

the cluster. Thanks to this, better uptime can be guaranteed, and maintenance becomes less critical as a failure will not bring down the whole application at once.

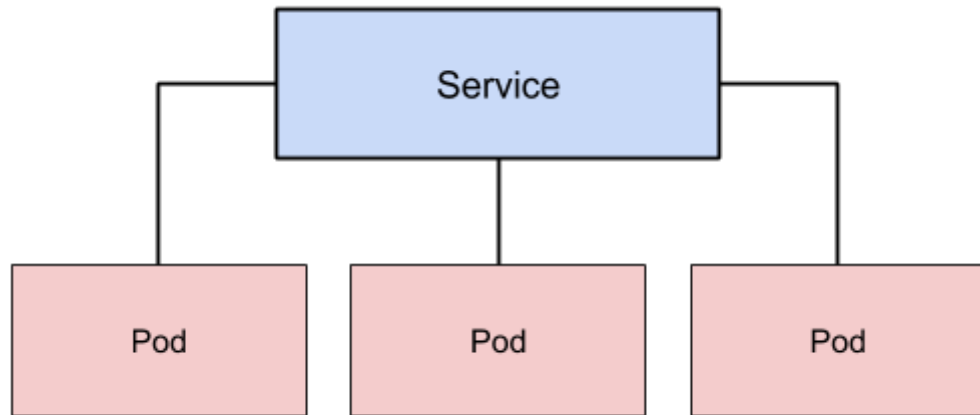


Figure 3.5 - A service and its pods in a Kubernetes cluster

3.4.3 Namespaces & Network Policies

Kubernetes supports multiple virtual clusters [28] backed by a single physical cluster. Namespaces enable this way of dividing a cluster into virtual clusters.

Network Policies [29] enable configuration of network traffic on a namespace level. By combining namespaces and network policies, we are able to isolate namespaces from each other and run completely independent virtual clusters.

Using the above solution, it is also possible to run multiple applications and multiple instances of the same application in the same cluster. This enables companies to host specifically customized instances for their customers in the same cluster, easing up the management of all the different deployments. In addition to this, namespaces can be used to host, for example, devtest, staging, and production environments in a single cluster - easing up the management even further.

This way of dividing the cluster into virtual clusters provides the possibility of gathering multiple application instances in one place. This in turn could lead to better structure, isolation and overview of the clusters' different resources and applications.

3.4.4 Ingress & Loadbalancers

In a complex and multi hosting environment like Kubernetes, we need a clear way of routing traffic into the cluster. A lot of the applications are reachable through the web, and we need an efficient way of routing and managing this traffic.

The Ingress [26] service helps us achieve this by offering name-based routing, automated TLS certificate creation, as well as port

mapping/forwarding. The ingress operates as an internal HTTP(s) router, easing web traffic management.

To expose this web traffic to the outside world via an external IP address, a service of type load balancer [25] can be bound to the ingress. This provides an external IP that forwards traffic to the ingress, which in turn handles internal routing. Another solution could be using a so-called NodePort, which could be an easier/smooth solution if only one replica were to be used. NodePorts does not provide a dedicated external IP to the service, however. With this solution the external IP would be the IP address of the worker node itself, and the port would by default be between 30000-32767.

One, or multiple, depending on the desired configuration, so-called Ingress Controllers need to be running on the cluster to enable and manage the ingress resources. Some well known and popular examples are the Nginx Ingress-Controller, Traefik, and Proxy HA.

3.4.5 Kubectl, yaml, and Configurations

Kubectl [30] is a command-line tool for controlling a Kubernetes cluster and is the standard tool used for this purpose. Kubectl needs a kubeconfig file to be able to do this. The kubeconfig is typically received when creating a Kubernetes cluster but may depend on the tools used during the creation.

It is also possible to create a customized [31] kubeconfig file on a running cluster - this can be useful to restrict access.

All configuration done on the Kubernetes cluster is defined in the yaml format and applied via the kubectl CLI. Resources to be deployed in the cluster are also specified in the yaml format and applied with kubectl.

The kubectl CLI communicates with the API-server running on the master node(s) in the cluster.

4 Implementation - experimental environment

In this section, we describe the solution(s) we have implemented, how it has been implemented and what decisions were made. We have tried to use free and open tools for every step of the way.

The purpose of this implementation is to gain more knowledge, insight, and experience about Kubernetes and its quirks. This will enable us to answer the problem, and provide a simple guideline of how a cluster can be set up.

4.1 Underlying Infrastructure

To run Kubernetes, the underlying infrastructure and provisioning need to be specified and configured accordingly. See Figure 4.1 for an overview of the underlying infrastructure.

4.1.1 OpenStack

The company we have been working with during this project, Sensative, is using OpenStack [32] as their cloud platform; therefore, we had access to the OpenStack environment for our prototyping and experiments.

4.1.2 Terraform

Since the goal is for the infrastructure to be as automatic as possible - infrastructure as code - we used Terraform [33] for provisioning of servers in OpenStack. This tool was also something already tested and suggested by the company.

4.1.3 Ansible

For the configuration of the servers provisioned with Terraform, we have used Ansible [34] with playbooks provided by Sensative to get the correct configurations for their platform.

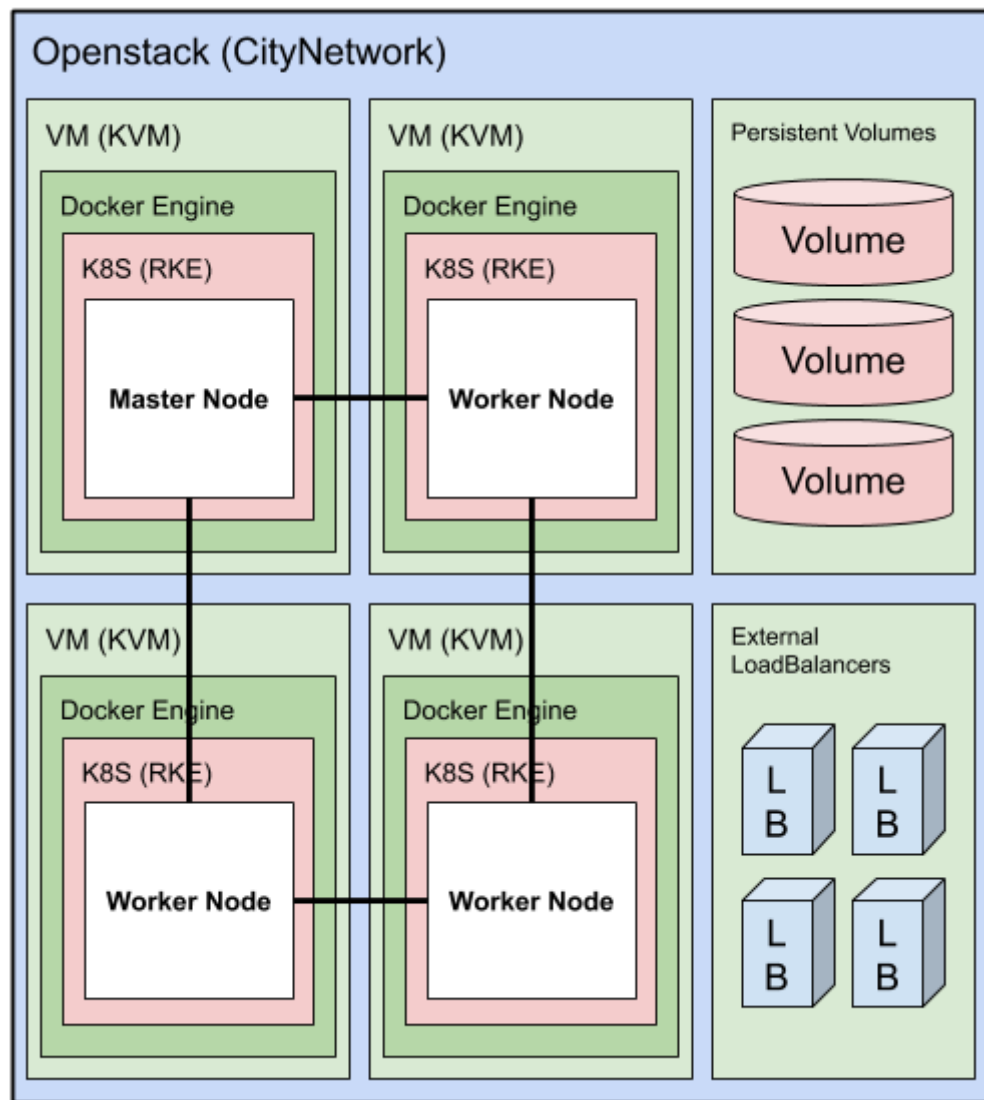


Figure 4.1 - Overview of the provisioning architecture

4.2 Kubernetes Installation

4.2.1 RKE

To install a Kubernetes cluster on the servers, we took advantage of the free tool RKE [10] (Rancher Kubernetes Engine) provided by Rancher. It is a simple CLI tool that takes input in the form of a configuration file in yaml format (see code block 4.1 for minimal example), using this to set up the cluster as wanted. Utilizing this tool, we were able to automate the installation process of the cluster instead of manually doing it.

```

0:  nodes:
1:      - address: 1.2.3.4
2:      user: ubuntu
3:      role:
4:          - controlplane
5:          - etcd
6:          - worker

```

Simple rke config example with a master/worker-node

Code block 4.1 - Example of minimal RKE configuration

4.2.2 Plugins

To enable communication between the Kubernetes cluster and the underlying OpenStack API, we have taken advantage of the official external tools provided.

The OpenStack Cloud Controller Manager [35] is a plugin that replaces the now deprecated old implementation integrated into Kubernetes; it has the task of establishing and managing the communication between the cluster and OpenStack.

The ability to dynamically provision volumes for persistent data storage in OpenStack is essential to circumvent manual operations. For this to work as intended, we used the CSI Cinder plugin [36], also an official plugin. This plugin works in symbiosis with the OpenStack Cloud Controller Manager.

Finally, we decided to install the official Kubernetes Dashboard [37], providing a GUI with an overview of the cluster and its components (see Figure 4.2). Something that has proven to be valuable during the prototyping and experiments.

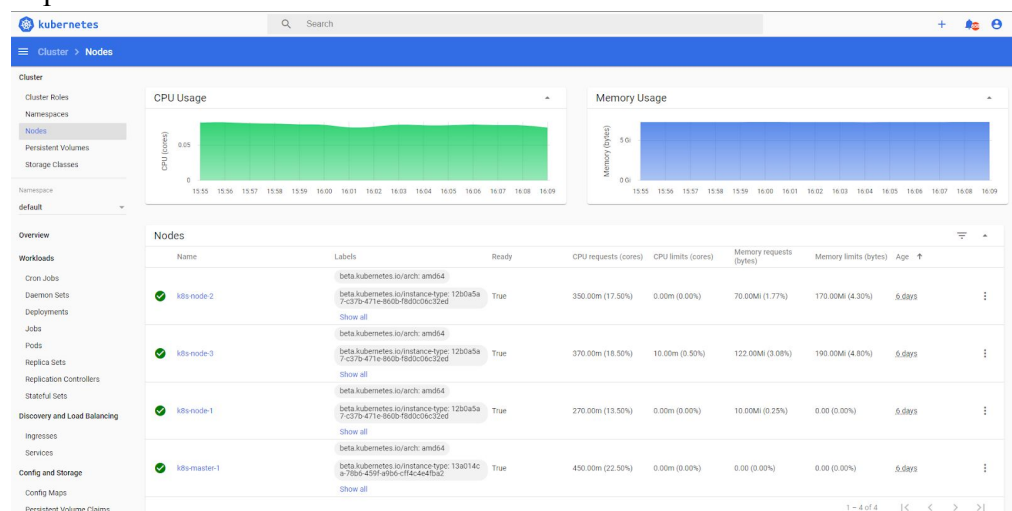


Figure 4.2 Example of the Kubernetes Dashboard overview

4.3 First prototype

An initial wish and suggestion from Sensative for the first prototype were configuring a barebones deployment of their platform Yggio [2] in Kubernetes. This simple deployment would consist of the bare minimum microservices (using Docker images provided by the company) to make it run, an ingress handling the HTTP traffic and DNS names as well as using cert-manager to set up TLS certificates.

This first prototype was deployed as one instance in the default namespace, with one master and two worker nodes in the cluster.

4.3.1 NGINX-Ingress and cert-manager

To handle the automatic creation of TLS certificates via Let's Encrypt [38], we used cert-manager - an x509 certificate management controller for Kubernetes. It helps us automate the certificate management, providing “certificates as a service” [39]. It has support for Let's Encrypt, a free, open and automated certificate provider, and we have chosen it to use for our certificates.

With cert-manager, we gain access to new resource types in our cluster, such as Issuers. We chose to use the type ClusterIssuer [40] (*see code block 4.2*), making it available for the whole cluster. This Issuer can then be used by our ingress resources to create the certificates needed automatically.

```
0:  apiVersion: cert-manager.io/v1alpha2
1:  kind: ClusterIssuer
2:  metadata:
3:    name: letsencrypt-staging
4:  spec:
5:    acme:
6:      server: https://acme-staging-v02.api.letsencrypt.org/directory
7:      email: ja222um@student.lnu.se
8:      privateKeySecretRef:
9:        name: letsencrypt-staging
10:     solvers:
11:       - http01:
12:         ingress:
13:           class: nginx
```



Set to letsencrypt's staging server for testing

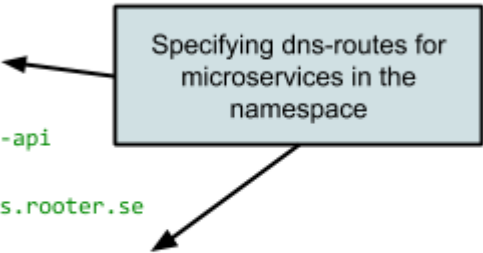
Code block 4.2 - ClusterIssuer configuration

To be able to route HTTP traffic based on hostnames to the platform's endpoints, we decided to use the proven NGINX-Ingress and Ingress-Controller [41]. The controller is installed in the cluster and handles the ingress resources we specify.

The ingress resource is configured in a yaml file (*see code block 4.3*),

where we specify the hostnames and their corresponding paths, services, and ports as well as the Issuer to be used for certificates.

```
0:  apiVersion: networking.k8s.io/v1beta1
1:  kind: Ingress
2:  metadata:
3:    name: nginx-ingress
4:    annotations:
5:      kubernetes.io/ingress.class: nginx
6:      cert-manager.io/cluster-issuer: "letsencrypt-staging"
7:  spec:
8:    tls:
9:      - hosts:
10:         - controlpanel.dev.k8s.rooter.se
11:         secretName: controlpanel-cert
12:      - hosts:
13:         - api.dev.k8s.rooter.se
14:         secretName: restapi-cert
15:    rules:
16:      - host: api.dev.k8s.rooter.se
17:        http:
18:          paths:
19:            - path: /
20:              backend:
21:                serviceName: rest-api
22:                servicePort: 80
23:      - host: controlpanel.dev.k8s.rooter.se
24:        http:
25:          paths:
26:            - path: /
27:              backend:
28:                serviceName: control-panel
29:                servicePort: 80
```



Specifying dns-routes for
microservices in the
namespace

Code block 4.3 - NGINX-Ingress configuration

4.3.2 kubectl and yaml-configurations

The standard way to deploy an application in Kubernetes is using yaml-files to configure the deployment and service of it (*see code block 4.4*). Here we specify the metadata, containers with corresponding Docker images, ports, and possible environment variables.

```

0:  apiVersion: apps/v1
1:  kind: Deployment
2:  metadata:
3:    name: control-panel
4:    labels:
5:      app: control-panel
6:  spec:
7:    selector:
8:      matchLabels:
9:        app: control-panel
10:   strategy:
11:     type: Recreate
12:   template:
13:     metadata:
14:       labels:
15:         app: control-panel
16:     spec:
17:       containers:
18:         - image: <link to image>
19:           name: control-panel
20:           ports:
21:             - containerPort: 80
22:               name: http
23:           env:
24:             - name: YGGIO_API_URL
25:               value: https://api.dev.k8s.rooter.se
26:             - name: YGGIO_DOMAIN_URL
27:               value: k8s.rooter.se
28:             - name: YGGIO_RULES_FE_URL
29:               value: https://rules.dev.k8s.rooter.se/start
30:       imagePullSecrets:
31:         - name: regcred

```

Code block 4.4 - Deployment yaml file using kubectl

To then install this microservice in the cluster, we used the kubectl command seen in code block 4.5.

```
kubectl apply -f deployment.yaml
```

Code block 4.5 - Applying a deployment file using kubectl

4.4 Namespaces and NetworkPolicies

The next natural step for our prototyping was being able to deploy multiple instances of the platform in the same cluster, taking advantage of namespaces [28] (see figure 4.3). This has also been an expressed wish from Sensative, as they want to handle all their deployments in one cluster; for production,

staging, and customers.

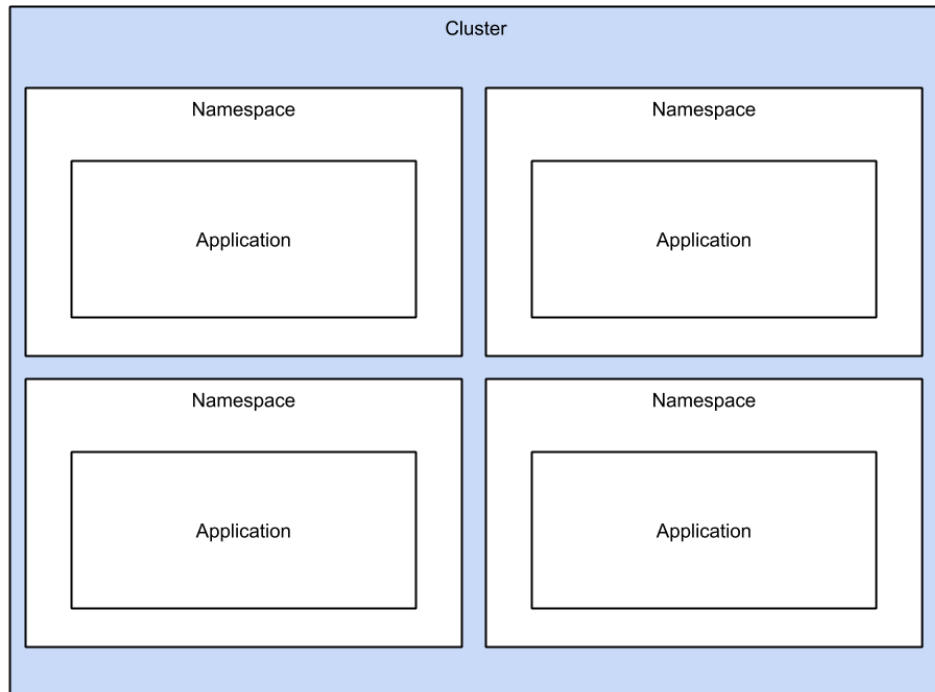


Figure 4.3 - Namespaces

We configured the namespaces in their own yaml-files (see code block 4.6), to get a good structure and overview of the components in our cluster. It is possible to create namespaces using a `kubectl` command directly, but we think that this leads to a lack of proper structure.

```
0:   apiVersion: v1
1:   kind: Namespace
2:   metadata:
3:     name: dev
4:     labels:
5:       name: dev
```

Code block 4.6 Namespace configuration

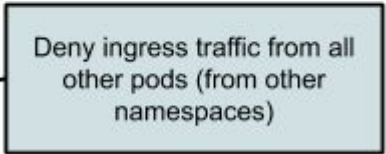
An essential aspect of using namespaces for multiple different deployments in one cluster, be it the same application or different, is the traffic between the namespaces. Since, in this case, we are deploying the same application in multiple instances, it is crucial to make sure that every instance is isolated from others. To ensure this, we used NetworkPolicies [29] to configure allowed and disallowed ingress and egress traffic (see code block 4.7).

```

0:  apiVersion: networking.k8s.io/v1
1:  kind: NetworkPolicy
2:  metadata:
3:    namespace: dev
4:    name: deny-from-other-namespaces
5:  spec:
6:    podSelector:
7:      matchLabels:
8:    ingress:
9:      - from:
10:         - podSelector: {}

```

Deny ingress traffic from all other pods (from other namespaces)



Code block 4.7 NetworkPolicy example

4.5 Second prototype and Helm charts

For further development of the implementation, the goals set for the next prototype was:

- Remove redundancy
- Run several replicas of the microservices, including the stateful ones
- Faster/easier deployment of new instances
- Using an external IP instead of the node IPs
- Better structuring

The previous prototype had too much redundancy in the form of using copies of the same configuration files for every namespace deployment, and it also required applying every file via the kubectl CLI.

To remove any single point of failure, the aim was to utilize the functionality of running multiple replicas [27] of every microservice.

We also wanted to stop using the IP addresses of the nodes themselves, and instead use an external IP for the hostnames.

Additionally, the cluster configuration was changed during the implementation of this prototype, to consist of one master and three worker nodes.

4.5.1 Helm and charts

Helm [42] is a tool that provides functionality to facilitate the management of complex Kubernetes applications. This is done by introducing Kubernetes packages, called charts [43]. These charts enable packaging and distribution of Kubernetes-ready applications. Helm charts provides the use of variables in configuration files, enabling dynamic and reusable configuration files by utilizing the Go Template Engine.

We took advantage of mainly the template engine to improve the structure of our implementation and remove practically all redundancy, as well as easing up the deployment process by a lot.

We solved this by implementing a main “umbrella” chart for the whole

platform containing all the microservices as sub-charts - dependencies (see code block 4.8).

```
0:  apiVersion: v2
1:  name: services-platform
2:  description: Kubernetes deployment of Yggio
3:  type: application
4:  version: 0.1.0
5:  deprecated: false
6:  keywords:
7:    - IoT
8:      Yggio
9:      Kubernetes
10:     Sensative
11:  maintainers:
12:    - name: Johan Andersson
13:      email: ja222um@student.lnu.se
14:      url: www.rooter.se
15:    - name: Fredrik Norrman
16:      email: fn222ep@student.lnu.se
17:  dependencies:
18:    - name: control-panel
19:      condition: control-panel.enabled
20:      import-values:
21:        - child: deployment
22:          parent: deployment

43:    - name: rest-api
44:      condition: rest-api.enabled
45:      import-values:
46:        - child: deployment
47:          parent: deployment
```

References to the
microservices'
subcharts

Flag to enable/disable
specific service in
deployment

Code block 4.8 - Main umbrella chart for the platform

The sub-charts hold the default values for their deployments in their own values-files, and we then use namespace-specific value-files to inject the configurations per namespace basis (see code block 4.9). In addition to this, we added a condition for the sub-charts, which enables the possibility to choose which dependencies to use per namespace basis.


```

0:  resources: {}
1:
2:  ingress:
3:    enabled: true
4:    production: false
5:    spec:
6:      hosts:
7:        - control-panel:
8:            hostName: controlpanel.dev.k8s.rooter.se
9:            secretName: controlpanel-cert
10:           serviceName: control-panel
11:           servicePort: 80
12:        - rest-api:
13:            hostName: api.dev.k8s.rooter.se
14:            secretName: restapi-cert
15:            serviceName: rest-api
16:            servicePort: 80
17:
18:  control-panel:
19:    enabled: true
20:    deployment:
21:      replicas: 3
22:      image: <link to image>
23:      yggioApiUrl: https://api.dev.k8s.rooter.se
24:      yggioDomainUrl: dev.k8s.rooter.se
25:      yggioRulesUrl: https://rules.dev.k8s.rooter.se/start
26:
27:  rest-api:
28:    enabled: true
29:    deployment:
30:      replicas: 3
31:      image: <link to image>
32:      yggioDomainUrl: dev.k8s.rooter.se
33:      yggioLocationManagerUrl:
34:        https://locationmanager.dev.k8s.rooter.se
35:      yggioRulesUrl: https://rules.dev.k8s.rooter.se

```

Namespace-specific values for the ingress resource, such as dns-names

Namespace-specific values for each microservice are set, injected further into the corresponding subchart

Code block 4.9 - Example of values-file for a namespace

Using this approach, we were able to remove a lot of the redundancy and keep just one set of the configuration files for the whole platform. Practically the only files needed per namespace are the values-files, which just contains the minimum required values needed to set for a new instance. To deploy the application in a new namespace, it is as simple as using the command seen in code block 4.10.

```
helm install -f env/dev/values.yaml dev-platform ./services-platform -n
dev
```

Code block 4.10 Installation command, pointing to the values-file, naming the deployment, pointing to the main chart, and specifying the namespace.

The values are then injected through the main chart to the sub-charts own values-files, which in turn are then echoed into the necessary configuration files (see code block 4.11).

```
0:  apiVersion: apps/v1
1:  kind: Deployment
2:  metadata:
3:    name: control-panel
4:    labels:
5:      {{- include "control-panel.labels" . | nindent 4 }}
6:  spec:
7:    replicas: {{ .Values.deployment.replicas }}
8:    selector:
9:      matchLabels:
10:        {{- include "control-panel.selectorLabels" . | nindent 6 }}
11:  template:
12:    metadata:
13:      labels:
14:        {{- include "control-panel.selectorLabels" . | nindent 8 }}
15:    spec:
16:      containers:
17:        - name: {{ .Chart.Name }}
18:          image: "{{ .Values.deployment.image }}"
19:          ports:
20:            - name: {{ .Values.deployment.portName }}
21:              containerPort: {{ .Values.deployment.containerPort }}
22:          env:
23:            - name: YGGIO_API_URL
24:              value: {{ .Values.deployment.yggioApiUrl }}
25:            - name: YGGIO_DOMAIN_URL
26:              value: {{ .Values.deployment.yggioDomainUrl }}
27:            - name: YGGIO_RULES_FE_URL
28:              value: {{ .Values.deployment.yggioRulesUrl }}
29:          {{- with .Values.imagePullSecrets }}
30:            imagePullSecrets:
31:              {{- toYaml . | nindent 8 }}
32:          {{- end }}
```

Using Go Language to inject values from external sources, such as the values-files

Code block 4.11 - Example of a configuration file where values are echoed using the Go Language

4.5.2 ReplicaSet

One of the main functionalities of Kubernetes is the ability to run several containers per image, which removes the problem of single-point-of-failure. This is relatively easy to achieve with stateless applications, and can simply be configured by specifying the number of replicas wanted in the configuration file for the deployment [23]. The deployment will then create a number of pods accordingly, which the service resource for the microservice then utilizes for load balancing (see figure 4.4).

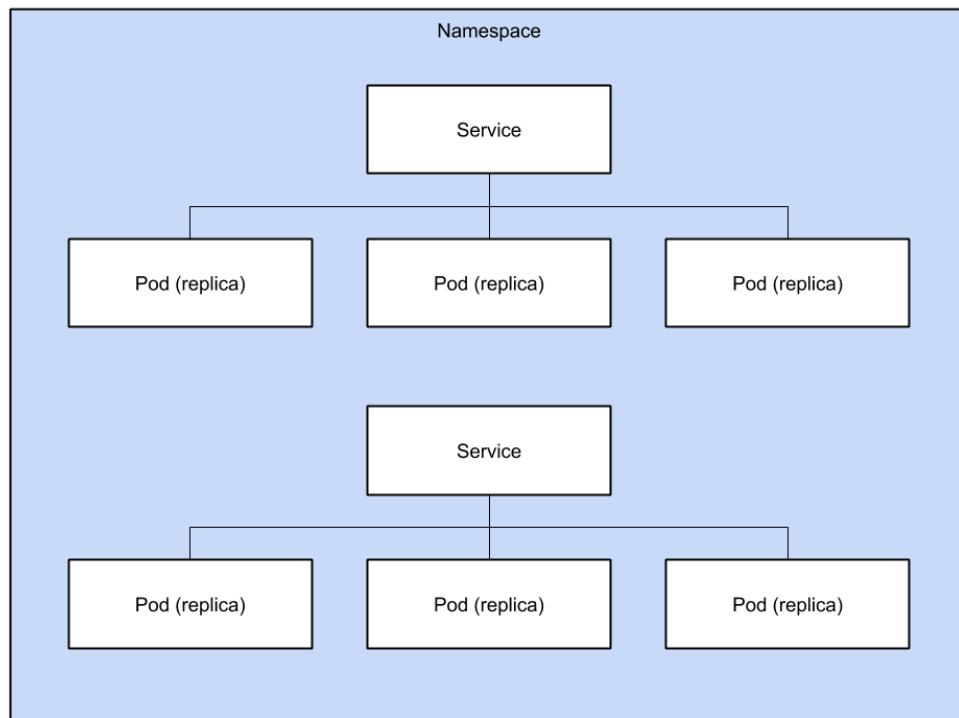


Figure 4.4 - Services and pods per microservice in a namespace

4.5.3 StatefulSet

To deploy several replicas of a microservice that has a state is a lot more complex compared to stateless. Due to Yggio depending on applications such as RabbitMQ and MongoDB, this was something we still needed to investigate. The balancing of the state, for example, the persistent data stored in MongoDB, is up to the application itself and requires the implementation to support it. It is then possible to deploy it using a StatefulSet [23]. Thankfully, both RabbitMQ and MongoDB have ready-to-go solutions for this, which are even available as Helm Charts [43].

To be able to automate the deployment as we wanted, we still used our own sub-charts acting as the dependency for the main chart as a middle layer,

putting the provided charts by RabbitMQ and MongoDB as dependencies to our respective sub-charts.

4.5.4 External LoadBalancer

To direct external traffic into our cluster, without using external IP addresses directly on the nodes themselves, we implemented external load balancers via the OpenStack API. Thanks to already configuring the cluster with plugins such as the OpenStack Cloud Controller Manager [35], we were able to configure the load balancers as resources directly in Kubernetes.

For each deployment in an independent namespace, a new ingress controller is deployed. This controller handles both the ingress resource for the namespace in question, as well as the connection to an external load balancer. The latter is deployed automatically with the application, provisioned in OpenStack via the Cloud Controller Manager, and directs both the web traffic and MQTT traffic for RabbitMQ through the ingress resource (see Figure 4.5).

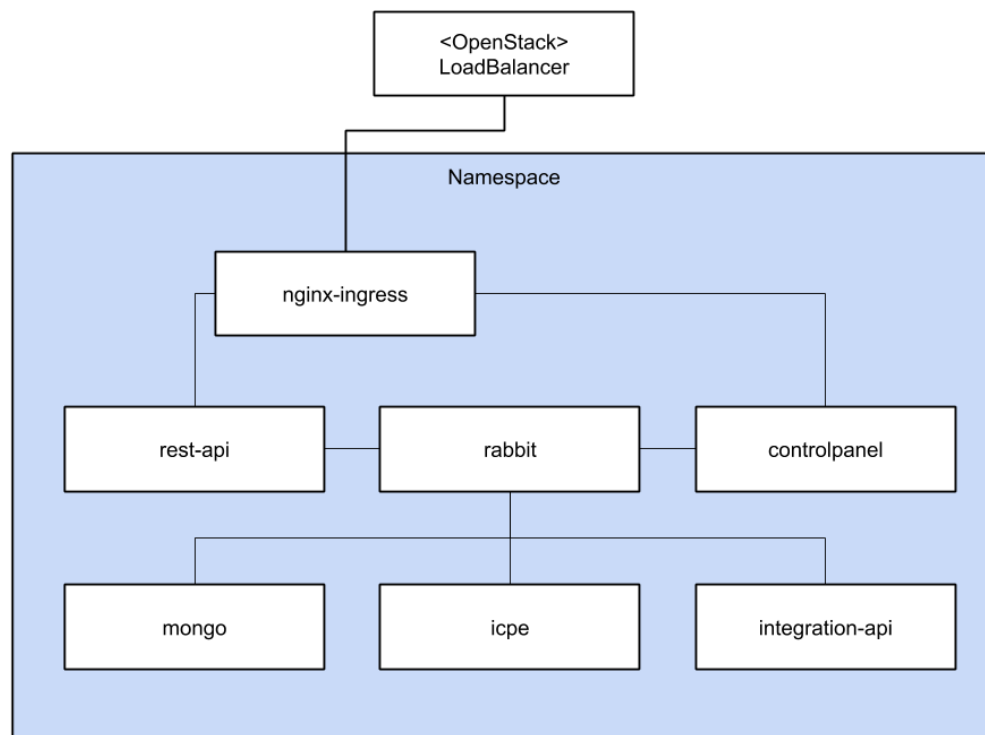


Figure 4.5 - Overview of a namespace with the external load balancers

5 Results & Analysis

The prototypes and corresponding experiments implemented and conducted during this project have resulted in some possible use cases for the company in the future. In addition to this, the implementations we have done can also act as a basic guide for other developers and companies looking for a similar solution - a foundation for future projects.

The following subsections will provide insights and pointers on how to achieve a scalable, mostly automated, Kubernetes cluster from scratch. It will be possible to deploy multiple applications separated from each other, with unique addresses for external access.

5.1 Underlying infrastructure and provisioning

To be able to run a Kubernetes cluster, we need computer resources to run it on. In our case, we are using OpenStack as our cloud platform, and to get a consistent and repeatable result, we heavily advocate the use of infrastructure as code tooling. This also eliminates human errors in addition to making it very easy to reinstall the underlying infrastructure. In this project, we used Terraform to provision our servers and Ansible to configure them to be ready for Kubernetes.

5.1.1 OpenStack

OpenStack is a free and open-source cloud computing platform containing all the functionality needed to develop automated infrastructure on top of.

When choosing a cloud platform, it is important to look into the support for various infrastructure as code tools. Configuring everything manually is very time-consuming in the long run and requires a lot of manual labor, which translates to money and locked up staff. It also introduces a greater risk of human errors, which is just a matter of time before it happens in most situations.

Infrastructure as code also helps us eliminate configuration drifting and snowflake servers.

5.1.2 Terraform

Terraform is a free open source infrastructure as code software, and was used in this project for the provisioning of servers.

Terraform is a pretty straight forward tool to use for this task, and supports many different cloud platforms - OpenStack being one of them.

It enables us to use a high-level configuration language (Hashicorp Configuration Language) to define our desired computer resources. With the help of variables supported by Terraform, we can make this configuration a

lot easier to customize and maintain. It is strongly recommended to version control these configuration files, just like regular software code. When the configuration is done, all that is needed now is a single command to start the magic. Even the novice user now possesses the ability to provision servers.

5.1.3 Ansible

Ansible is an automation engine that can be used to provision cloud services automatically, configure devices, deploy applications, and much more.

We utilized Ansible and its Ansible Playbooks (using yaml) for automatic configuration management of the devices created by Terraform. Our Ansible Playbooks configured the servers to run Docker, on top of which the Kubernetes Cluster will be installed. Just like with Terraform, it is highly suggested to version control the configuration files - in this case, the Playbooks.

5.2 Kubernetes Installation

Kubernetes itself *could* be installed manually; however, this is a fairly daunting and complicated task. For this reason, a number of tools are available, providing a more straightforward and automated installation process to get started with a cluster. Using any of these tools enables us to yet again take advantage of the infrastructure as a code mindset, and version control our configuration files.

The installation could be streamlined by using Ansible through the whole process, removing the need of additional tools. This would require creating our own Ansible scripts for the Kubernetes installation itself, which due to the time constraints we chose to not do in favor of using a more plug-and-play tool. Another possible solution could be to run the chosen tool with Ansible. However, our focus with the project was more on the implementation of the cluster itself and not optimizing the whole installation process.

5.2.1 RKE

RKE (Rancher Kubernetes Engine) is a free to use Kubernetes installer by Rancher, that installs a cluster according to provided specifications on Linux hosts.

The desired configuration of a cluster is specified in a file named `cluster.yml`. Everything needed for the cluster can be specified here, ranging from the number of nodes and their individual configurations to various services et cetera. If needed, additional resources to be installed with the cluster can be added as well.

We picked RKE out of all available options due to it being free and simple to use yet highly customizable, suiting our needs for this project well.

5.2.2 Nodes

Our configuration of choice landed on deploying one master node accompanied by three worker nodes, for a couple of reasons.

Normally, it would be recommended to run multiple master nodes for the sheer reason of eliminating single point of failure. However, since this project is still in the development stage, one master was sufficient enough for further testing purposes.

When deciding on the number of worker nodes needed, we had to factor in the requirements of the software to be deployed in the cluster. The main reason to run multiple worker nodes in the first place is to benefit from the fail-tolerant and horizontal scaling, deploying multiple replicas of the applications. The reason for running an odd number of worker nodes is due to our use of stateful services, which in most cases are recommended to be run on an odd number of replicas and nodes [45]. We settled on three to not go overboard since it is a testing environment, but still having multiple nodes to get the aforementioned benefits.

During initial prototyping, the nodes were configured with very slim computer resources; 1 CPU core and 2 GB RAM per node. When scaling up the cluster and deploying more instances, this had to be increased to ensure stability. We finally settled on 2 CPU cores per node, and 4 GB RAM.

5.2.3 Additional configurations

To be able to use the external and non-deprecated cloud controller manager for integration with OpenStack, we configured the kubelet service accordingly. This was done by deactivating the standard internal controller manager by telling the kubelet service to expect an external controller.

Since one of the requirements was separate ingress controllers and load balancers per namespace, we had to deactivate the ingress controller supplied by the default RKE configuration.

5.3 Useful plugins

Kubernetes comes with a lot of functionality built-in per default, but there are vast possibilities to further extend and modify this functionality according to customized needs. To improve the ability of automation and the overview as well as management of the cluster, we used a couple of plugins.

5.3.1 OpenStack Cloud Controller Manager

Kubernetes, by default, does indeed have the ability to communicate with the OpenStack API, enabling automatic provisioning defined in the Kubernetes configuration. The already built-in method for this is now deprecated due to a wish to separate it from the Kubernetes core, and instead make this an

external functionality as this provides more streamlined updates and faster adoption of new functionality provided by cloud providers.

To take advantage of a more up-to-date controller manager, while simultaneously future-proofing the cluster by moving away from deprecated solutions, we decided to install the external OpenStack Cloud Controller Manager.

5.3.2 CSI Cinder plugin

One of the main goals of moving on to a Kubernetes based infrastructure for Sensative is to eliminate manual steps, making the installation and management process of applications as automated as possible.

The application deployed in this cluster uses stateful services, such as MongoDB, that require provisioning of volumes in OpenStack. Usually this would have to be a manual step of creating the volumes via the OpenStack API or via for example Terraform. We wanted this to happen dynamically when deploying new instances, however. In other words, we wanted it to be managed by Kubernetes automatically.

Through research about the topic, we discovered the CSI Cinder plugin, which instead enables automated provisioning of volumes in OpenStack directly from the cluster.

By specifying storage class resources, we can define the specifics of the volumes to be created. The storage class is then used in conjunction with a persistent volume claim resource while deploying the application to provision new volumes in OpenStack dynamically.

5.3.3 Kubernetes Dashboard

The Kubernetes Dashboard is an official plugin, but not installed per default. It consists of a web application providing a clean GUI for an overview of the cluster. Having the dashboard available not only facilitates manual troubleshooting when needed, but also the management of the cluster overall - in combination with the automation of IaC. Mainly, it provides a graphical overview of the cluster.

5.4 Ingress and external load balancers

To be able to manage and route web traffic to applications in the cluster, we took advantage of the Nginx Ingress Controller in combination with external load balancers provisioned in OpenStack.

Due to the ingress controller only being able to be connected to one load balancer, we decided to deploy one controller per namespace - as one of the requirements was to use one load balancer per namespace. This requirement stems from the need of one IP address per application deployment.

Thanks to the external cloud controller, the provisioning of load balancers

in OpenStack is, as mentioned, done automatically when deploying into a new namespace.

5.5 Namespace independency

The aim was to deploy multiple instances of the application(s) in the cluster, each with their own differing configurations, without interference between the different instances.

We solved this by defining a number of namespaces in the cluster and network policies to block traffic between them. Namespaces can be created when deploying resources, but due to structure and version control, we decided to define them as resources in yaml files. The network policies are defined in one yaml file per rule for better separation.

Thereafter, the platform can be deployed namespaced.

5.6 Application configurations

The separate microservices that make up the sum of the platform are each configured with a service resource and a related deployment resource.

In the service resource, the metadata and ports are specified. In the deployment resource, we define the number of replicas desired - in our case, three, for good mapping with the number of worker nodes in the cluster. Details such as the Docker image to be used as well as possible environment variables to be passed to this are also defined here.

5.7 Dealing with redundancy

Over time as the project grew in size and complexity, we started to notice a rapidly growing problem of duplicated code and redundancy.

The standard way of defining cluster resources works in such a way that all resources require their own configuration file. In our case, running the same application(s) with different configurations would have led to the majority of our configuration files being duplicated. This resulted in a lot of redundant code, with very tiny differences. This could have been a nightmare to manage in the end, and probably very time consuming to keep track of. Mainly if the cluster is used to deploy a multitude of instances of the same application with minor differences. If for example one change is needed to be done to the configuration of all instances, then every file for every instance would have had to be edited. This turned out to be the case with our prototypes.

The answer to this problem in our case is Helm charts.

5.7.1 Helm charts

Helm enables the creation of dynamic configuration files and the use of

variables. Helm is using the Go Template Engine to achieve this, generating configuration files based on the defined variables. With Helm, we only need to create one value file per deployment instead of a whole package of the application. Helm then translates this into configuration files and deploys them to the cluster. All that is needed is to specify a value file when deploying, and Helm takes care of the rest.

Our solution involved using a main - so-called umbrella chart - for the application/platform, with all the required microservices defined as dependencies in the form of sub-charts. This way, we get a reusable Helm chart for the application.

5.8 Analysis

5.8.1 Resources

Hardware resources are utilized more effectively with a Kubernetes-based infrastructure versus the previous solution the company had in place. This would be beneficial for Sensative due to the money savings that could be made, and overall beneficial for the environment.

Instead of multiple different and separated servers, each deploying their own Docker Compose, everything could be deployed more cohesively through the Kubernetes cluster - using a configured number of dedicated servers.

5.8.2 Deployment

Previously, the application was deployed on separate servers using unique Docker Compose-configurations for each deployment. With a Kubernetes cluster in combination with the chart modeling of Helm, deployment is both made more automatic and easier. This also leads to less vulnerability to human errors when deploying. In this project's prototypes, deployment is done by configuring values-files and using a single helm install-command to deploy a new instance.

With infrastructure as code and utilizing Helm for configuration files, we also gain the positive effect of versioning, and in turn, easier rollbacks when needed. This also leads to better traceability of possible errors.

Sensative would, with these prototypes as a basis, be able to deploy multiple instances of their application(s) in a single Kubernetes cluster - all separated and independent by using namespaces. This could, for example, be deployments to different customers, staging, and production deployments.

Using the plugins for OpenStack - the underlying provisioning used by Sensative - it is also possible to use dynamic provisioning of volumes for services that use persistent data, removing yet another manual step of new deployments. This integration between Kubernetes and OpenStack also

enables the use of external load balancers for application instances, making it possible to assign unique IP addresses for each instance while still being run on the same servers.

5.8.3 Maintenance

Having all deployments of their application(s) in a cluster gives a vastly better overview compared to isolated, independent servers. With the use of the Dashboard GUI in combination with automated processes of the cluster, everything is easily monitored from a single place, freeing up human resources and manual labor.

With their current solution, Sensative is forced to use so-called server guards - meaning, individuals of the staff taking turns to be on call 24/7 if something were to fail. This is expensive, as it ties up parts of the staff in periods and requires more money investments. Some form of human server guard would still be needed if a major failure were to occur, but running a more robust infrastructure utilizing container orchestration - with multiple replicas of each microservice - does indeed provide higher uptime of the application. This means that for minor errors, the server guard response time is not as critical.

Using Kubernetes, the services can not only be configured to be self-healing - auto-deploying new instances when crashed - but also run on several replicas. E.g., a microservice can run with several copies so that if one fails - it still runs. Running several replicas also provides internal load balancing of the services. All in all, this removes the problem of single-point-of-failures on the application layer and between its microservices, and makes sure high uptime is provided. One single-point-of-failure still exists in the form of the external load balancers, but this is still a huge improvement compared to having multiple SPF overall.

The cluster can be configured to run on any number of desired servers - so-called nodes. Deployed services and their replicas are all spread out over the nodes. If one node - server - fails, this ensures that the application(s) will keep running, removing unnecessary and unwanted downtime. Yet again, we see another possible savings factor: less manual work is needed, and customers will likely be happier. This could be argued to equal more long term customers as well as new ones.

6 Conclusions and Future Work

6.1 Conclusions

An important realization is that there exists no definite solution to the problem we set out to “solve”. That is, how to move on to and set up a Kubernetes based infrastructure. Rather, there are many options and paths to choose between, with many different factors affecting the final choices. However, there are some ground rules about different aspects that can be followed, such as the use of namespaces for platform isolation, using ingress resources to route web traffic to applications, et cetera.

This could be one of the reasons why we found it to be the case that guidelines, tutorials and the likes available online were lacking in information. Every implementation of a Kubernetes cluster is unique, based on the requirements set and expected use. The more complex of a platform to be deployed, as well as the more diverse functionality needed of the cluster, the more complex the configuration and set up of the cluster turns out.

Still, we think that bigger and more general guidelines should be possible to present. They need not be specified in every minor detail but done in a (broader) way as to act as a starting point for new configurations of varying sizes.

We gained some compelling insights and general knowledge about *how* a cluster *can be* configured with some basic concepts in mind through our research and prototyping with different cluster configurations and implementations.

Our hope is that even though some platform-specific details are presented, the resulting guideline and implementation examples we reached are still in a general enough form to act as a foundation for future configurations.

6.1.1 Reflections on possible improvements

Even though we reached our goal of implementing a Kubernetes cluster, gaining a lot of knowledge in the process, we are aware of a number of possible improvements that could be made. These improvements apply to both the implementation itself as well as the resulting guideline.

Our implementation could have been tested more thoroughly to for example observe the behavior under more heavy load, and what implications that would have on the solution itself. Would the cluster be stable as is, or would something have to be done differently?

An aspect we only just grazed the surface of is node failure - one of the servers going down. As one of the incentives for a move to a Kubernetes based solution is the more automated and self-healing infrastructure possibilities, correctly configuring the cluster to handle node failures and

recovery is of great importance. Testing the implementation of the cluster while forcing a node failure and observing the behavior of both the cluster and the applications running on it is something that we could have done more.

Hand in hand with the above mentioned failure of a node, is the scaling of a cluster - both horizontal and vertical. We focused more on scaling the microservices themselves with different numbers of replicas, but lacked in scaling the cluster itself with trying different numbers of nodes in a live setting.

Although the tools we ended up using for our prototyping - e.g., Terraform, Ansible, rke, nginx-ingress, et cetera - proved useful, we could have set more time aside to explore the different alternatives available.

The migration pattern could have been explored with more starting points in mind, and not only assume a previous Docker Compose basis. What would have to be done if a different containerization technique acted as a base, what if there is no containerization to begin with? Alternatives such as Docker SWARM could also have been included more in the prototyping, as well as prototyping on different cloud computing platforms other than OpenStack.

All in all, there is a lot more knowledge and experience to gain about the different aspects of Kubernetes that could have benefitted our end result - the guidelines.

6.1.2 Benefits of the results

The general benefits of the result are to get inspiration of how a Kubernetes cluster can be installed and configured. It also shows a brief overview of some tooling available that makes it possible to automate time consuming steps in the process. By automating we not only save time but also minimize the human factor of introducing errors, and gives us the ability to version control our configuration.

Depending on the software intended to migrate to Kubernetes, there are more specific benefits from the result.

If the intended software is going to be deployed multiple times with different configurations, Helm will help with minimizing duplicated code and facilitate the management of the configuration and deployments. We came to the conclusion that in our use case it could be very difficult to manage and maintain this type of configuration by only using the default configuration options. Helm enables the use of value files, which in turn enables a dynamic way to generate configuration based on these files. In the value file we assign variables configuration specific values which are used to generate unique configuration files. This could eliminate a lot of redundant code, and greatly facilitate maintenance. This also facilitates future needs, as all that is required for a new deployment is an additional value file. In our use case we found

this to be true. For more complex systems where multiple applications or multiple instances of the same application are deployed, Helm could prove to be a valuable tool. If the system is of a relatively simple nature, there could be the risk of over-engineering the implementation with Helm.

Before making the decision to start using Helm, one should be aware that this will lead to changes in the administrative workflow. The biggest change is a clear increase in the complexity of the configuration syntax, which supports both variables and functions to name a few. This allows for very complex and dynamic configurations, as these configuration files can be reused - making it possible to deploy the same software several times with different configurations. This can be done without introducing duplicated configuration files, which leads to a much more efficient maintenance work. The biggest contributing factor to this efficiency improvement is the ability to share configuration files, which provides a highly efficient file structure with all files gathered in one place.

We must be aware that we are adding another tool in our already well-filled toolbox, and it will be another tool that we must learn to use. However, the benefits this provides outweigh all the negatives, and makes it well worth learning. When operating multiple instances of the same software with different configurations, we are of the firm opinion that this should be done with the help of Helm. Without Helm we would have received large amounts of duplicate configurations which are almost identical. It would also have made future configuration changes difficult in the sense of error prone, and time consuming, since it would require editing multiple configuration files for each desired change.

6.2 Future possibilities

Migrating to a Kubernetes-based infrastructure could be seen as future-proofing the platform, especially for companies such as Sensative that already have a microservices-model in place for their application(s). Since it also likely will lead to less downtime and less manual work in the DevOps department, it can possibly open up new possibilities for other work.

There are endless possibilities for different solutions in a Kubernetes cluster. A lot of previously manual work can be configured to be automated. One example is integrating a pipeline with, for example, Gitlab. This could be used for automated deployment of new instances when new versions of services are pushed to the git repository.

6.3 Further work

With this project and what we have found out during it, regarding the tools, techniques, and possible solutions of implementing a Kubernetes cluster as a basic foundation, we hope that it could be easier for companies and

developers to realize what possibilities and positives it can provide for their infrastructure. Also acting as a simple starting point and guide, it could make the initial steps easier.

References

- [1] "Sensative » Making sense of IoT", Sensative, 2020. [Online]. Available: <https://sensative.com/>. [Accessed: 02- Jul- 2020].
- [2] "Yggio IoT Interoperability Platform by Sensative", Sensative, 2020. [Online]. Available: <https://sensative.com/yggio/>. [Accessed: 02- Jul- 2020].
- [3] "Overview of Docker Compose", Docker Documentation, 2020. [Online]. Available: <https://docs.docker.com/compose/>. [Accessed: 08- Jun- 2020].
- [4] "Server Scalability Using Kubernetes", IEEE Xplore Digital Library 2019. [Online]. Available: <https://ieeexplore-ieee-org.proxy.lnu.se/document/9024501> [Accessed: 15-mar-2020]
- [5] "Swarm mode overview", Docker Documentation, 2020. [Online]. Available: <https://docs.docker.com/engine/swarm/>. [Accessed: 08- Jun- 2020].
- [6] "Docker Compose vs Docker Swarm", linuxhint 2019. [Online]. Available: https://linuxhint.com/docker_compose_vs_docker_swarm/ [Accessed: 15-mar-2020]
- [7] "Docker Compose vs Kubernetes", stackshare 2020. [Online]. Available: <https://stackshare.io/stackups/docker-compose-vs-kubernetes> [Accessed: 15-mar-2020]
- [8] "Kubernetes Vs. Docker Swarm: A Comparison of Containerization Platforms", VEXXHOST 2017. [Online]. Available: <https://vexxhost.com/blog/kubernetes-vs-docker-swarm-containerization-platforms/> [Accessed: 17-mar-2020]
- [9] "Run Kubernetes Everywhere", Rancher Labs, 2020. [Online]. Available: <https://rancher.com/>. [Accessed: 02- Jul- 2020].
- [10] "Overview of RKE", Rancher Labs, 2020. [Online]. Available: <https://rancher.com/docs/rke/latest/en/>. [Accessed: 08- Jun- 2020].

- [11] "CNCF Survey: Use of Cloud Native Technologies in Production Has Grown Over 200%", CLOUD NATIVE COMPUTING FOUNDATION 2018. [Online]. Available:
[\[https://www.cncf.io/blog/2018/08/29/cncf-survey-use-of-cloud-native-technologies-in-production-has-grown-over-200-percent/\]](https://www.cncf.io/blog/2018/08/29/cncf-survey-use-of-cloud-native-technologies-in-production-has-grown-over-200-percent/) [Accessed: 24-mar-2020]
- [12] "Virtualization", En.wikipedia.org, 2020. [Online]. Available:
<https://en.wikipedia.org/wiki/Virtualization>. [Accessed: 08- Jun- 2020].
- [13] "What is Hypervisor and what types of hypervisors are there?", VapourApps Private Cloud, 2020. [Online]. Available:
<https://vapour-apps.com/what-is-hypervisor/>. [Accessed: 08- Jun- 2020].
- [14] "Microservices", En.wikipedia.org, 2020. [Online]. Available:
<https://en.wikipedia.org/wiki/Microservices>. [Accessed: 08- Jun- 2020].
- [15] "What is a Container? | Docker", Docker, 2020. [Online]. Available:
<https://www.docker.com/resources/what-container>. [Accessed: 08- Jun- 2020].
- [16] "OS-level virtualization", En.wikipedia.org, 2020. [Online]. Available:
https://en.wikipedia.org/wiki/OS-level_virtualization. [Accessed: 08- Jun- 2020].
- [17] "What is Container Orchestration? Definition & Related FAQs | Avi Networks", Avi Networks, 2020. [Online]. Available:
<https://avinetworks.com/glossary/container-orchestration/>. [Accessed: 08- Jun- 2020].
- [18] "docker images", Docker Documentation, 2020. [Online]. Available:
<https://docs.docker.com/engine/reference/commandline/images/>. [Accessed: 08- Jun- 2020].
- [19] "Install Docker Compose", Docker Documentation, 2020. [Online]. Available:
<https://docs.docker.com/compose/install/>. [Accessed: 08- Jun- 2020].
- [20] "Swarm mode key concepts", Docker Documentation, 2020. [Online]. Available:
<https://docs.docker.com/engine/swarm/key-concepts/>. [Accessed: 08- Jun- 2020].

- [21] "Kubernetes", En.wikipedia.org, 2020. [Online]. Available: <https://en.wikipedia.org/wiki/Kubernetes>. [Accessed: 08- Jun- 2020].
- [22] "4. Kubernetes Cluster Architecture and Considerations — Trident documentation", Netapp-trident.readthedocs.io, 2020. [Online]. Available: https://netapp-trident.readthedocs.io/en/stable-v19.01/dag/kubernetes/kubernetes_cluster_architecture_considerations.html. [Accessed: 08- Jun- 2020].
- [23] "Concepts", Kubernetes.io, 2020. [Online]. Available: <https://kubernetes.io/docs/concepts/>. [Accessed: 08- Jun- 2020].
- [24] "Kubernetes Components", Kubernetes.io, 2020. [Online]. Available: <https://kubernetes.io/docs/concepts/overview/components/>. [Accessed: 08- Jun- 2020].
- [25] "Service", Kubernetes.io, 2020. [Online]. Available: <https://kubernetes.io/docs/concepts/services-networking/service/>. [Accessed: 08- Jun- 2020].
- [26] "Ingress", Kubernetes.io, 2020. [Online]. Available: <https://kubernetes.io/docs/concepts/services-networking/ingress/>. [Accessed: 08- Jun- 2020].
- [27] "ReplicaSet", Kubernetes.io, 2020. [Online]. Available: <https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/>. [Accessed: 08- Jun- 2020].
- [28] "Namespaces", Kubernetes.io, 2020. [Online]. Available: <https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/>. [Accessed: 08- Jun- 2020].
- [29] "Network Policies", Kubernetes.io, 2020. [Online]. Available: <https://kubernetes.io/docs/concepts/services-networking/network-policies/>. [Accessed: 08- Jun- 2020].
- [30] "Install and Set Up kubectl", Kubernetes.io, 2020. [Online]. Available: <https://kubernetes.io/docs/tasks/tools/install-kubectl/>. [Accessed: 08- Jun- 2020].

- [31] "Organizing Cluster Access Using kubeconfig Files", Kubernetes.io, 2020. [Online]. Available: <https://kubernetes.io/docs/concepts/configuration/organize-cluster-access-kubeconfig/>. [Accessed: 08- Jun- 2020].
- [32] "Build the future of Open Infrastructure.", OpenStack, 2020. [Online]. Available: <https://www.openstack.org/>. [Accessed: 08- Jun- 2020].
- [33] "Terraform by HashiCorp", Terraform by HashiCorp, 2020. [Online]. Available: <https://www.terraform.io/>. [Accessed: 08- Jun- 2020].
- [34] "Ansible is Simple IT Automation", Ansible.com, 2020. [Online]. Available: <https://www.ansible.com/>. [Accessed: 08- Jun- 2020].
- [35] "kubernetes/cloud-provider-openstack", GitHub, 2020. [Online]. Available: <https://github.com/kubernetes/cloud-provider-openstack/blob/master/docs/using-openstack-cloud-controller-manager.md>. [Accessed: 08- Jun- 2020].
- [36] "kubernetes/cloud-provider-openstack", GitHub, 2020. [Online]. Available: <https://github.com/kubernetes/cloud-provider-openstack/blob/master/docs/using-cinder-csi-plugin.md>. [Accessed: 08- Jun- 2020].
- [37] "Web UI (Dashboard)", Kubernetes.io, 2020. [Online]. Available: <https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/>. [Accessed: 08- Jun- 2020].
- [38] "Let's Encrypt - Gratis SSL/TLS-certifikat", Letsencrypt.org, 2020. [Online]. Available: <https://letsencrypt.org/sv/>. [Accessed: 08- Jun- 2020].
- [39] "cert-manager x509 certificate management for Kubernetes", cert-manager 2020. [Online]. Available: <https://cert-manager.io/> [Accessed: 15-mar-2020]
- [40] "ClusterIssuers — cert-manager documentation", Docs.cert-manager.io, 2020. [Online]. Available: <https://docs.cert-manager.io/en/release-0.11/reference/clusterissuers.html>. [Accessed: 08- Jun- 2020].

[41] "NGINX Ingress Controller for Kubernetes | NGINX", NGINX, 2020. [Online]. Available: <https://www.nginx.com/products/nginx/kubernetes-ingress-controller/>. [Accessed: 08- Jun- 2020].

[42] "Helm", Helm.sh, 2020. [Online]. Available: <https://helm.sh/>. [Accessed: 08- Jun- 2020].

[43] "Charts", Helm.sh, 2020. [Online]. Available: <https://helm.sh/docs/topics/charts/>. [Accessed: 08- Jun- 2020].

[45] "Clustering Guide — RabbitMQ", Rabbitmq.com, 2020. [Online]. Available: <https://www.rabbitmq.com/clustering.html#node-count>. [Accessed: 08- Jun- 2020].