Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Andrea Standeren

MongoDB in a Self-Managed Kubernetes Cluster

Deploying MongoDB in a Self-Managed Kubernetes Cluster without an Operator

Master's thesis in Computer Science
Supervisor: Svein Erik Bratsberg
June 2022

NTNU
Norwegian University of Science and Technology

Andrea Standeren

MongoDB in a Self-Managed Kubernetes Cluster

Deploying MongoDB in a Self-Managed Kubernetes Cluster without an Operator

Master's thesis in Computer Science
Supervisor: Svein Erik Bratsberg
June 2022

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Abstract

This thesis aims towards exploring the manual setup process required when deploying MongoDB in a self-managed Kubernetes cluster without the help of an operator. The alternative approach takes advantage of a Kubernetes cluster managed by a provider and applies application specific operators for configuring applications to run in a Kubernetes cluster. Mapping out the areas of complexity versus simplicity within the manual approach is a valuable contribution to the field of DevOps, Development and IT Operations, considering whether or not one should rely on third-parties or do it yourself. In the process of procuring hands-on experience to justify a recommendation of an approach over another, a self-managed Kubernetes cluster was initialized using Azure Virtual Machine (VM)s and kubeadm before deploying a replicaset of three MongoDB instances using a StatefulSet and a sidecar container. The thesis emphasizes all roadblocks met on the way to obtaining this architecture and reflects on what concepts that demand cutting-edge domain expertise. The journey revealed that the biggest challenge by resolving to manual approaches when working with such a tightly coupled system as Kubernetes is, is that there are unique configurations for every varying component. The thesis concluded that setting up a self-managed Kubernetes cluster is relative quick and easy while still possessing freedom and control over the cluster. On the other hand, not exploiting an operator for configuring a stateful application to run in a Kubernetes environment was not recommended.

Sammendrag

Denne oppgaven har som formål å utforske den manuelle oppsettprosessen som kreves ved implementering av MongoDB i et self-managed Kubernetes cluster uten hjelp fra en operator. Den alternative fremgangsmåten innebærer et provider-managed Kubernetes cluster og bruk av applikasjonsspesifikke operatorer for å konfigurere applikasjoner til å kjøre i et Kubernetes cluster. Kartlegging av områdene kompleksitet og enkelhet innenfor den manuelle fremgangsmåten er et verdifullt bidrag til feltet DevOps, Development og IT Operations, med tanke på om man bør belage seg på tredjeparter eller gjøre det selv. I prosessen med å skaffe praktisk erfaring for å rettferdiggjøre en anbefaling av en fremgangsmåte fremfor en annen, ble et self-managed Kubernetes cluster initialisert ved å bruke Azure VMs og kubeadm før et replikasett med tre MongoDB-forekomster ble implementert ved bruk av et StatefulSet og en sidecar container. Oppgaven legger vekt på alle veisperringer på veien mot å oppnå denne arkitekturen og reflekterer over hvilke konsepter som krever spisskompetanse innenfor domenet. Reisen viste at den største utfordringen ved å velge manuelle fremgangsmåter når man arbeider med et system som består av mange komponenter, er at det er unike konfigurasjoner for hver varierende komponent. Avhandlingen konkluderte med at det er relativt raskt og enkelt å sette opp et self-managed Kubernetes cluster, samtidig som man bevarer frihet og kontroll over clusteret. På den annen side ble det ikke anbefalt å unngå å benytte seg av en operator for å konfigurere en stateful applikasjon til å kjøre i et Kubernetes-miljø.

Acknowledgments

With this section I wish to express my gratitude towards people that have been central buildingblocks in developing this final construction being my master thesis. Throughout my journey of working on this thesis I have been continuously accompanied by my supervisor, Svein Erik Bratsberg, through both remote and physical guidance. He has been a source of motivation, self-confidence and courage. I also want to thank him for always being available and supportive.

Next, I want to thank Yngve Molnes, an experienced developer in the field of Kubernetes, who generously provided me with his knowledge and expertise. His technical guidance through pair programming and remote consultancy has been central for accomplishing the experiments connected to this thesis.

Lastly, I wish to thank my institute, IDI, at NTNU, for financing the resources obtained on Azure, which was essential for my research.

Contents

Abstract . . . iii
Sammendrag . . . v
Acknowledgments . . . vii
Contents . . . ix
Figures . . . xi
Tables . . . xiii
Code Listings . . . xv
Acronyms . . . xvii
1 Introduction . . . **1**
1.1 Thesis Goals . . . 1
1.2 Outline . . . 2
2 Background . . . **3**
2.1 Containers and Container Orchestration Tools Explained . . . 3
2.1.1 Containers . . . 4
2.1.2 Container Orchestration Tools . . . 4
2.2 Benefits of Containers and Container Orchestration . . . 6
2.3 Kubernetes Commercial Usage . . . 7
2.4 Evolution of Containerizing Stateful Applications . . . 8
2.4.1 Volume Plugins, Persistent Volumes, Persistent Volume Claims and Storage Classes . . . 9
2.4.2 StatefulSets . . . 10
2.4.3 Operators . . . 10
2.5 Alternatives for Cluster Initialization and Stateful Application Deployment . . . 11
2.5.1 Self-Managed or Provider-Managed Cluster . . . 11
2.5.2 Using an Operator or not . . . 14
2.6 Similar Research . . . 15
2.6.1 The Approach . . . 16
2.6.2 Results . . . 18
3 Method . . . **21**
3.1 Exploratory Environment . . . 22
3.2 Experimental Environment . . . 22
3.2.1 Cluster Setup . . . 22

3.2.2 Application Setup 24
3.2.3 General Considerations Concerning Manual Setup 35
3.3 Information Gathering 38
4 Conclusion 39
4.1 The Final Setup 39
4.2 Answers to the Research Questions 41
5 Future work 45
5.1 Comparing Manual and Automated Approach 45
5.2 Testing Different Operators' Complexity 46
5.3 Comparing Performance of Different Persistent Volume Kinds 46
5.4 Comparing the Manual Setup Process Between Different Databases 46
Bibliography 47
A Output from MongoDB 51
B Configuration Files on Master 55

Figures

2.1 Logic architecture of VM (left) and containers (right). 5
2.2 Overview of the experimental setup of MySQL. 16
2.3 Overview of the experimental setup of TiDB. 17
2.4 Overview of the experimental setup of CockroachDB. 17

3.1 Cluster setup of three Azure VMs. 25
3.2 MongoDB applied on cluster. 26
3.3 MongoDB status immediately after applying. 27
3.4 MongoDB applied to the cluster as a Deployment. 27
3.5 MongoDB applied to the cluster as a StatefulSet. 28
3.6 Initializing the MongoDB replicaset with commands in pod specification. 29
3.7 Output when running rs.status() command from MongoDB on the PRIMARY MongoDB instance on worker0. 31
3.8 Calico ippools output. 31
3.9 Output of route -n from master node before editing ippools. 32
3.10 Editing the configuration for Calico ippool. 33
3.11 Calico ippools output after edited ippool configuration. 34
3.12 Output of route -n from master node after editing ippools. 34
3.13 How internal pod-communication is achieved in an Azure/Calico environment. 35
3.14 Status of LoadBalancer service running in cluster. 35
3.15 External connection to MongoDB from the master node with port-forwarding. 35
3.16 A possible solution to gaining external connection to the application. 36

4.1 Final Kubernetes cluster architecture. 42

Tables

3.1 Inbound ports on the master node. 24
3.2 Inbound ports on the worker nodes. 24

Code Listings

2.1 Command for viewing environment variables that are set for the kube-apiserver. 13
2.2 Command for viewing environment variables that are set for the ETCD. 13

3.1 Command for viewing pods details. 26

A.1 Output from rs.status() from mongodb instance on worker0 after editing Calico ippools. 51
A.2 Snippet of output from container logs. 53

B.1 Custom dockerfile for MongoDB image with networking tools installed. 55
B.2 Headless service for MongoDB replicaset. 55
B.3 Secret for MongoDB credentials. 56
B.4 Statefulset for three replicas of MongoDB. 57
B.5 Role-based access control. 58
B.6 StorageClass. 59
B.7 Persistent Volume for worker 0. 59
B.8 Persistent Volume for worker 1. 59
B.9 Persistent Volume for master. 60

Acronyms

AKS Azure Kubernetes Service. 13, 25, 45

AWS Amazon Web Services. 4, 8, 9, 13, 38

CIDR Container Inter-Domain Routing. 23

CIS Center for Internet Security. 13

CNI Container Network Interface. 23, 24, 30, 32, 39, 40

DNS Domain Name System. 6, 10, 27, 33, 34

EC2 Elastic Compute Cloud. 8

EKS Amazon Elastic Kubernetes Service. 13, 25, 45

GKE Google Kubernetes Engine. 4, 13, 16, 25, 45

OS Operating System. 3, 4

PV Persistent Volume. 9, 10, 26–28, 42

PVC Persistent Volume Claim. 9, 10, 26–28, 40, 42

RBAC Role-based Access Control. 13, 29

VM Virtual Machine. iii, v, xi, 4, 5, 7, 22–24, 40, 41

YCSB Yahoo! Cloud Serving Benchmark. 46

Chapter 1

Introduction

If there would been a nomination of the hippest technology the past years, I would place my bet on one candidate for one of the top ratings - Kubernetes. Companies worldwide are thirsty after developers who posses knowledge in this domain. Kubernetes' ability to easily scale allows companies to deliver their services to their customers with a sky high availability. They host thousands of microservices in pods running in the cloud. Actually, the physical limit of pods running in a Kubernetes cluster is 150 000 pods![1] This appeals even to the companies that have a certain architecture or services which may not make Kubernetes the optimal hosting environment, but just because it is trending. The technology is mainly designed to handle stateless applications, but with the help of new Kubernetes concepts and big brains, stateful applications can also be hosted. However, incorporating this technology into your company's infrastructure – and let alone if your company's infrastructure rely on stateful applications – require high domain knowledge. The community highly recommends to take advantage of aids simplifying both cluster initialization and application deployment and management. This brings us to the essence of this thesis; figure out how difficult the manual approach really is.

1.1 Thesis Goals

This thesis encapsulates the process of an exploration of deploying a stateful application – MongoDB in particular – in a self-managed Kubernetes cluster, without the help of an operator. What these terms means and what the process involves will become clear in Chapter 2. Moreover, the goal of exploring this is to confirm or deny what seems to be the general opinion in the Kubernetes community - admittedly that one should utilize a provider-managed cluster and an application-specific operator when deploying a database in Kubernetes. We define the thesis goal achieved when the following questions can be adequately answered:

1. How much overhead is added to a setup process if choosing a self-managed

Kubernetes cluster?
2. Is the overhead added to the manual setup process above viable, with increased freedom taken into account, to consider the approach for your company or personal development?
3. How much overhead is added to a setup process if avoiding an operator in the deployment of MongoDB in a Kubernetes cluster?
4. Is the overhead added to the manual setup process above viable, with increased freedom taken into account, to consider the approach for your company or personal development?

1.2 Outline

This master thesis will begin by covering some basics concerning container and container orchestration technologies, before providing a section on how this technology is used commercially as of today. All this will be found in Chapter 2, Background, which is based on the specialization project delivery in the subject TDT4501 at NTNU Computer Science. This chapter also includes a section presenting the considerations that should be done when facing the choice between a self- or provider-managed cluster and using an operator or not. In the same chapter there will be a section presenting an example of a similar previous research done on database usage in the container orchestration field. Following up, Chapter 3, Method, will explain the approach of the experiment. It will cover the specifications of the experimental environment and how the environment was set up, including how the application – meaning MongoDB – was set up. Different approaches, as well as the challenges met, towards the desired application setup will be addressed in this section. Chapter 4, Conclusion, will discuss the setup process leading to a conclusion of the master thesis, whether or not you should seek help in a provider-managed cluster and whether or not you should apply an operator for your stateful applications running in Kubernetes. Finally, Chapter 5 will address some possible expansions of the experiment and alternative approaches to make within the specific field of databases in a containerized environment.

Chapter 2

Background

This chapter will cover the research made connected to the topic of this thesis - Exploring the setup of MongoDB in a self-managed Kubernetes cluster without the help of operators. First, it will cover what containers and container orchestration tools are, in particular Docker and Kubernetes, which are the technologies of the most relevance today. Following up is a section highlighting two different approaches on how a Kubernetes cluster can be initialized and how stateful application deployment can be done. Alternative approaches for cluster initialization comes in terms of managed or self-managed, while alternative approaches for stateful application deployment comes in terms of using an operator or not. Further, the benefits of such technologies will be discussed in addition to the fields where these technologies are not optimal yet. Then, a status as of today when it comes to container usage and container orchestration usage, as well as the commercial usage of databases in such environments, will be presented. Finally, a similar research done on the field of databases in a containerized environment will be addressed.

2.1 Containers and Container Orchestration Tools Explained

Application development has evolved over the years to be more complex, consisting of more components and dependencies, it can be written in an unlimited amount of different languages and there are many underlying Operating System (OS) to adapt to. This flexibility in the field of application development has on the other hand led to a time consuming and troublesome work for the operation teams. Allowing the application to be so depended on other libraries and on their underlying OS makes them difficult to update and will very often lead to questions like "why doesn't it work on my machine?". These struggles forced IT engineers to find a way to abstract the application from the underlying OS and make updates easier and faster by isolating the application. As a result, containers was considered an attractive technology to conquer this problem in the 2000s [2].

Containers was actually developed as early as in 1979, but it needed to take another 20 years before the technology saw its area of use. However, the technology became more available in 2008 through the inclusion of container functionality within the Linux kernel [3]. Also, the launch of Docker open-source container platform in 2013 gave containers even more attention. Containers are simple to maintain and manage in small quantity, but as organizations often has a need of scaling their applications to increase performance, the amount of containers can reach a number that is too high to manage manually. To ease the operation team for this, container orchestration tools were introduced. With such tools operation teams could run their application in an automated cluster that manages failover, updates, loadbalancing and scaling. To conclude, containers and container orchestration tools unarguable provide the industry with valuable implementations and features, but how does these technologies actually work?

2.1.1 Containers

Containers are lightweight units of software that packages code and dependencies that allow the program running inside to be isolated from its underlying environment. The term lightweight means that they do not require a dedicated OS per container - they share the underlying OS of the host server [4]. Being lightweight units of code they spin up fast and can be destroyed fast. In contrast, VMs, which possesses many of the same features as a container, must run with its own OS. See Figure 2.1 that illustrates the differences between a VM and a container. In commercial use today there are many vendors of container technology including Docker, Amazon Web Services (AWS) Fargate, Google Kubernetes Engine (GKE) and many more, where Docker is the most used one by far [5].

In order to run, or initiate, a container, an **image** must be created first. An image is made by building a dockerfile which describes – with command lines – how to start a container with an application inside it while referring to where the code for the application can be found on the local machine.

2.1.2 Container Orchestration Tools

A container orchestration tool is responsible for running some sort of application on a cluster of nodes without administrative impact from an operation team. There are many vendors on the market offering tools to manage a cluster of machines running many containers each. Kubernetes, Docker Swarm and Apache Mesos are some to mention the most common ones. Among these, Kubernetes is definitely the most popular. Before Kubernetes can spin up an application, a Kubernetes cluster must be deployed, which can consist of nodes being both physical and VMs. The cluster is bootstrapped by installing Kubernetes critical software, being kubeadm, kubelet and kubectl, on the nodes.

- **Kubeadm** is a tool tailored specifically for cluster bootstrapping, including

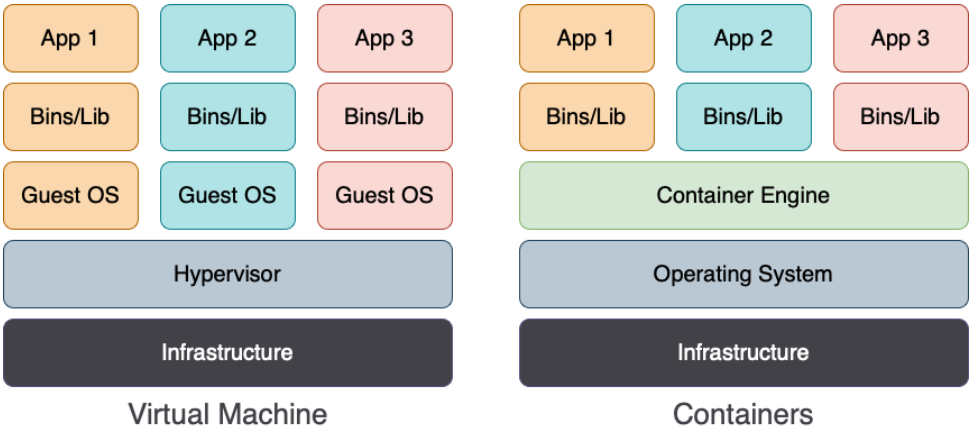


Figure 2.1: Logic architecture of VM (left) and containers (right).

initializing a cluster, joining a cluster, resetting a node, recreating a cluster join-token, upgrading a cluster and more.

- **Kubelet** is a piece of software running on each node that administrates that pods running on the particular node are satisfying the pod specifications – in shape of YAML files – applied to the pod through the apiserver.
- **Kubectl** is the command line tool used for managing the cluster and communicating to the cluster, issued from the master nodes' host terminal.

After installing the essentials, the cluster gets initialized by calling a *kubeadm init* command from a node designated as a master node. The maintaining nodes that should handle workloads, being workers, is then added to the cluster by copying and running a kubeadm join command that was issued from the master.

For Kubernetes to be able to automatic manage and maintain all the containers in the cluster, there are many configuration files needed, in addition to logic abstractions of software units. These configuration files are mostly written in the markup language YAML due to the in general higher user-friendliness, even though JSON can be used interchangeably in almost all scenarios. The fundamental files needed for an application to run and be accessed are a Deployment file and a service file [6]. However, a so-called StatefulSet configuration file can replace a Deployment, but more details on this down the line. It is also worth mentioning that configuration files that describe different concepts in the Kubernetes world can be grouped together in the same file, but following best practice you should only group configurations of concepts that have a close connection.

Deployments

A Deployment, in Kubernetes, is a description of a set of pods, where a pod is a set of containers running applications. The description provides e.g. information on how many pods to run, how many containers to run inside, what image the con-

tainers shall run, how much RAM and how much CPU the containers should use. When the Deployment is applied to the Kubernetes cluster all containers inside all the pods spins up automatically. From inside the pod, the containers are accessed by using the pods' IP-address, or localhost, and a corresponding port, which its port-mapping is defined in the Deployment YAML file through a NodePort field. Explicit access to a container from outside the pod is therefor not possible and must go through the pod's IP-address. Since pods are ephemeral, meaning that they get created and destroyed dynamically, they also change IP-addresses dynamically.

Services

A service, in Kubernetes, is a component in the cluster that is a workaround to handle the frequently changing pod IP-addresses when pods are communicating. To enable stable communication between two pods, e.g. a pod running frontend code and another pod running backend code, the frontend pod can communicate with the backend pod through the Domain Name System (DNS) name of the service [7]. The service will always be accessible for the application pod through this name since the service is configured with a static clusterIP. A DNS service in the cluster will map the DNS name to the IP-address of the service and route the request to the service. A selector on the service will use the portnumber triggered to select all the pods with the corresponding name, e.g. api, and pass the requests to the correct pods. In addition the service is able to load balance the traffic sent to the set of pods with the requested name to increase performance and availability.

Other Kubernetes concepts also exists, such as Secrets, ConfigMaps and a couple ones related to successfully running stateful applications. The latter being StatefulSets, Persistent Volumes, Persistent Volume Claims and Storage Classes, which are further explained in Section 2.4.

2.2 Benefits of Containers and Container Orchestration

As mentioned above a huge advantage of containers and container orchestration tools is to ease the management work for operation teams in terms of neglecting OSes from the development process, automatic updates, scaling and failover, but there are more benefits [8].

Rapid Development and Testing

Development and testing are two sides of the same story, at least in agile development. Developers continuously test new code snippets during development by running the application locally, or by having different kinds of tests that must be passed to be able to push new commits to a version controller tool like git. This process is simplified with containers because container images are quick and easy

to update with new code since they, being lightweighted, spin up fast. The co-operation between testers, developers and operators also gets improved by the guarantee that all units will run the application with the same requirements and dependencies so misunderstandings can be avoided.

Lower Costs

For organizations to develop applications today, they will probably need servers for running both a testing environment and a production environment. These servers may be VMs or physical machines, but anyway they will require more resources which implies higher costs. In addition, you might also get more power and storage than you actually need, which is a waste of money. In contrast, containers are open-source and free to use. The portability of containers is also a source of saving since you wont need to pay someone to configure the host machine and install all required dependencies.

Improved Security

Containers are secure by nature. Their ephemeral property, resulting in 54% of them existing for five minutes or less [9], provides them with the mechanism of quickly overcoming software vulnerabilities. Another property increasing security is the fact that they are isolated units. Despite these natural security measures, one should be careful with container *images*. The images are executable code and is a popular target for tampering [10].

2.3 Kubernetes Commercial Usage

The adoption of container technology in technical organizations is growing in an incredible speed. A prediction says that by the year 2025, 85% of organizations will run containers in production. This is an increase of more than 55% from the state of the art in container-usage in production[11]. Another survey can reveal that Kubernetes is on the top 3 appreciated platforms used by developers [12]. When it comes to what companies that actually use these technologies, there are many to mention. Some examples are Spotify, Pokemon Go, Tinder and MobiDev.

Spotify

Spotify, the world's largest music streaming service provider, is an early adopter of microservices and Docker as they introduced these in 2014 [13]. They actually developed their own container orchestration system named Helios where they hosted their services. However, having a small team inside Spotify working on optimizing Helois wasn't efficient enough considering the fast growth of end users. In 2019 they were able to migrate some of their workloads seamlessly over to Kubernetes, but the possibility to scale introduced a challenge in managing multiple

services. As a workaround, Spotify developed their own plugin for the Kubernetes API called Backstage which is a platform for building developer portals. Today the company has over 1600 production services running on Kubernetes.

Pokemon Go

With the use of Kubernetes, Pokemon Go could establish hubs all over the globe providing higher availability. This was quite helpful, taken in account, the more than 20 million daily users the company has.

Tinder

When switching their platforms to Kubernetes, Tinder were able to reduce the waiting time for Elastic Compute Cloud (EC2) instances down to seconds. An EC2 is a virtual server in AWS terminology where an AWS cloud compute server is provisioned to an AWS subscriber if requested [14]. Compared to the previous EC2 waiting time on multiple minutes, the company experienced a significant reduction in cost when migrating to Kubernetes.

MobiDev

MobiDev had another kind of issue to solve with the usage of Kubernetes. One of their projects, a Point of scale (POS) software and venue management system adopted by a lot of bars and restaurants, needed a new strategy [12]. They wanted to make predictions of sales for the next period for each venue by the use of an AI-based forecasting system. Kubernetes made this task solvable by providing real-time computing resources optimization by offering auto-scaling.

In general, Kubernetes and containers are suitable, and tempting, choices for developers and companies that needs to scale big. This property pairs very well with Machine Learning and Artificial Intelligence and is predicted to be a combination that will be seen quite frequently in the future [12].

On the other hand, there are scenarios where containers might not be the best choice. Examples of such scenarios are for projects that are simple and small, consumes a tiny user base or has a simple architecture. Another scenario where Kubernetes might not be the best choice, is when dealing with stateful applications – in particular – databases. At least this is the general perception in the tech community, but is it true?

2.4 Evolution of Containerizing Stateful Applications

Considering the ephemeral characteristic of containers, they clearly are not made for stateful applications where data must be stored persistently to ensure the correct behavior of such applications. From the early days of container technology,

developers were strictly advised to steer clear of running databases in Kubernetes [15]. Instead, many stateful applications needed to change their architecture to make the client responsible for remembering the state of the application by the use of session cookies and local storage. If the need of a database however was unavoidable, the solution was almost always to host the database on a server outside of the Kubernetes cluster. This was especially the case if the application was relying on transactions, meaning that the ACID property delivered by relational SQL databases was crucial [16]. When it comes to applications where one can accept *eventual* consistency, NoSQL databases has become acceptable to use in a containerized environment due to their ability to scale which is a necessity in such environments. Also, the newer database alternative, NewSQL databases, have properties that make them a promising option in the world of containers as they combine online ACID transaction processing with the speed and performance delivered by NoSQL. However, as container technology continues to mature, developers strive for adapting the data layer to make it an even better fit for containers. As a result of this dedicated work, distributed SQL databases was designed to seamlessly integrate with microservices like Kubernetes while still being compatible with the SQL language. Being distributed by nature makes it easier to manage in a clustered environment. In order to make this evolutionary change allowing databases to run in containers and container orchestrations, adaption on the data layer wasn't the only change needed. Kubernetes has extended their framework with additional API objects to better facilitate for persistent storage.

2.4.1 Volume Plugins, Persistent Volumes, Persistent Volume Claims and Storage Classes

Since containers have a short lifecycle, the responsibility for storing data in a persistent matter is outsourced to a third party through a volume plugin. Currently, Kubernetes supports 16 different kinds of volume plugins, whereas Azure Disk, AWS Elastic Block Store, Google Cloud Engine Persistent Disk and Network File System storage, are some examples [17]. The connection between the volume plugins and the cluster is achieved with objects called Persistent Volume (PV), Persistent Volume Claim (PVC) and in some cases Storage Classes. PVs are pieces of storage in the cluster that are provisioned manually by an administrator or automatically by the use of a so called *storage Class*. PVs are resources in the cluster in the same way as pods are, created by applying a configuration file, but they only consist of meta-data of the external storage unit, the volume plugins. When the pieces of storage are needed by an application a PVC is created that connects the application to a suitable persistent volume, depending on properties like size and access modes. Requested size and access mode can either be defined explicit in the fields of the PVCs YAML file or there can be a field in the PVCs YAML file pointing to a specific kind of Storage Class, implicitly defining the values of these properties. A Storage Class must have defined a provisioner which determines what volume plugin is used for provisioning PVs. All new updates to the applic-

ation that results in changes in the state will be added to the storage unit of the PV that is bound to the pod through the PVC. If, and when, the container running the stateful application, crashes, a new container running the same image is created and the storage needed to retain the same state as prior to the crash is mounted back to the stateful application. Where to mount the data back into the new pod is defined as part of the pod configuration file in shape of a path to the volume called "MountPath" under the definition of the containers [18]. Where the data that should be mounted back to this path is to be found, being defined by the PVC, is however often defined on a higher level of abstraction than the pod configurations – namely in the *StatefulSet* specifications.

2.4.2 StatefulSets

As an alternative to using a PVC that is independent from your stateful application running inside a pod, StatefulSets allow managing multiple replicas of the application and corresponding PVCs at the same time [19]. Running multiple replicas without a StatefulSet introduce operative overhead in terms of handling the Deployment related requirement that makes the application able to handle concurrent read-write operations of the same file. Instead of this approach, StatefulSets was introduced to replace Deployments for such applications. StatefulSets are similar to Deployments in the sense that they manage pods based on an identical container spec, but they differ in the sense that StatefulSets maintain an identity for all the pods [20]. In order to maintain the identities for the pods a headless service is required. A headless service is a service that does not have any clusterIP inside the pod. It creates internally necessary endpoints to expose pods with DNS names in order to reach a specific pod making it possible to consistently run the same pod on the same node as its corresponding PV retains.

2.4.3 Operators

An additional step in the journey towards containerizing stateful applications is the introduction of operators. Operators are application-specific software that automates the deployment and management process to applications on Kubernetes. This is especially the case when it comes to deploying stateful applications, such as databases. Since all stateful applications require a unique process of deployment and management that the vendors have the most domain knowledge about, the vendors themselves have to develop the operator for their specific stateful application. However, the quality and complexity of the different operators vary a lot. Some database operators offers nearly any features that simplifies the process, making them more troublesome than useful.

Operators work by exploiting the Kubernetes extension *Custom Resources*. A custom resource is a modification of the Kubernetes API involving an additional endpoint that can be custom configured. This resource can be interpreted in the same way as a pod or a StatefulSet. This is only a Kubernetes property that operat-

ors exploit, but a control loop mechanism is what they really are. A control loop mechanism is a Kubernetes core property that allows for its automation by always checking the state of the cluster and match it with the desired state stored in the ETCD [1] created from the applied YAML files. However, the control loop mechanism that the operators works with makes sure that the custom resources obtain the desired state in the cluster.

2.5 Alternatives for Cluster Initialization and Stateful Application Deployment

As mentioned Kubernetes is a sought tool in the tech field these days due to its ability to automate deployment in a large scale. However, this is not necessarily always true. Global Cloud providers, such as Google and Amazon, invest much resources in their provider-managed Kubernetes cluster solutions. Choosing a provider-managed cluster solution simplifies the whole process for users migrating into the Containerized world, but the choice require some sacrifice. Another concept that liberates the user for a lot of configuration and management when it comes to deploying a diversity of applications onto Kubernetes, is the operators. The alternative to exploit the aid provided by operators is to do everything manually, introducing a lot of overhead.

Following is an explanation of the pros and cons linked to the choices one has to make. The section is outlined with focus on one aspect at a time; freedom, simplicity, control, security and cost, first when discussing self-managed or provider-managed clusters, and again when discussing using an operator or not.

2.5.1 Self-Managed or Provider-Managed Cluster

Freedom

Building your Kubernetes cluster as a self-managed cluster gives you the freedom to configure the cluster to your desires [21]. You have direct access to the master node and the control plane. This implies that you can issue kubectl commands directly from the master in contrast to a provider-managed cluster where the provider have their own managed service to control the master node(s). By abstracting away this layer from the user, the user are not able to access or control any of the following;

- Kube-API-server: Validate and configure data for all objects that can be reached through the API.
- Kube-controller-manager: A daemon pod running in the control plane that is responsible for maintaining the desired state of the cluster by running a

[1] The distributed key-value store of Kubernetes, which ir's name comes from the directory /etc combined with "distributed".

non-terminating control loop over the API.
- ETCD: The Kubernetes specific datastore where all cluster state-related data is stored.
- Kube Scheduler: Responsible for assigning workloads to the workers in terms of pods based on data in ETCD.

More of the potential freedom obtained by managing your own cluster is also discarded when providers tends to inflict you with vendor lock-in. Even if intended or unintentional, vendors will adapt their services to fit better (or maybe *only* fit) with other services that they also offer. In addition, providers will not give you the freedom to use the latest Kubernetes releases if their services are not designed to handle them.

Simplicity

When considering the simplicity on the other hand, building your Kubernetes cluster as a provider-managed cluster simplifies the whole process from initializing to management and maintenance. Relying on a provider takes care of most of the duties required keeping the cluster running. Doing all the configurations yourself is almost doomed to be troublesome. The process will be way more time consuming than the provider-alternative since all necessary packages and tools must be downloaded and internal networking must be manually configured.

Control

Considering the control aspect of the alternatives, choosing provider-managed clusters forces you to give away some privacy since you are hosting all your workloads on the providers control. However, unless your cluster is used for confidential military workloads or building a cure to cancer that is worth a billion dollars, this is not the biggest sacrifice.

Security

Another important aspect providers will (or at least should) take care of for you is the security. Especially important is it to secure and not publicly expose the API server that acts as an entrypoint to all your nodes and the containers running on them [22]. Ensuring that the ETCD is not publicly exposed is also something the provider should handle for you. Attackers constantly have bots searching for open APIs to Kubernetes clusters and can use them for malicious actions. An example of such actions was when Tesla's erroneously open consoles was abused for illegal cryptocurrency mining operations. Considering the security from a self-managed cluster point of view, *doing-it-yourself* is obviously not to prefer when comparing to provider-managed, unless you are never suppose to expose your cluster for production anyway. Despite this, there are small measures to apply on your self-managed cluster to make it at least *more* secure. Running the command

in Code listing 2.1 provide you with an output of all environment variables that are set for the kube-apiserver. These could then be compared with the security measurements recommended by the Center for Internet Security (CIS) Foundation Benchmarks[2]. Ensuring that the ETCD is not exposed can also be done in a self-managed cluster by running the command in Code listing 2.2 and controlling that the security essential environment variables are set. The same can be done enabling Role-based Access Control (RBAC) as an authorization-mode will also make sure not all cluster administrators can access all services in your cluster. Especially should the cluster-critical namespace kube-system be restricted to certified cluster administrators. Another security issue that must be addressed when choosing a self-managed cluster is the default open network between pods in the cluster. This should be avoided by customizing the network configurations by creating the Kubernetes object called NetworkPolicy, when going into production. The simple reason for this is that if a malicious attacker unfortunately should get access to creating new pods the attacker is able to access the whole cluster through the open internal network.

Code listing 2.1: Command for viewing environment variables that are set for the kube-apiserver.

```
andrea@masternode$ ps -ef | grep kube-apiserver
```

Code listing 2.2: Command for viewing environment variables that are set for the ETCD.

```
andrea@masternode$ ps -ef | grep etcd
```

Cost

Of course, the benefits you get by choosing a provider-managed cluster surly does come at a cost. The services from the providers are expensive. On the other hand, even though a self-managed cluster is free, you do need to take into account the costs associated with continuous infrastructure and maintenance costs. If you do not obtain – or have employees that obtain – the knowledge and experience with managing a Kubernetes cluster, the general tip is to steer clear of doing it yourself.

The most common provider-managed Kubernetes clusters are the following:

- Azure Kubernetes Service (AKS), created, delivered and managed by Microsoft Azure.
- Amazon Elastic Kubernetes Service (EKS), created, delivered and managed by AWS.
- GKE, created, delivered and managed by Google Cloud Platform.

[2] `https://www.cisecurity.org/cis-benchmarks/`

2.5.2 Using an Operator or not

Freedom

Similar to taking the advantage of a provider for your cluster management in exchange for freedom, the same sacrifice has to be made when taking advantage of an operator for your application management. There is also a positive side of the freedom aspect when it comes to operators. It allows the operator developer to customize the feedback from the *kubectl describe [custom resource]* command, which provides the cluster administrators with the information that best suits the particular application. However, this kind of freedom is only experienced by the operator developers, not the end-users.

Simplicity

The most valuable advantage of using an operator is, without doubt, the simplicity. Deploying a stateful application without the use of an operator is very troublesome and time consuming and it requires a lot of domain knowledge connected to the exact database. Avoiding these challenges are the essence of an operator. Assuming that the operator provider has developed a good operator, it should ease the user of having to create, run and update the cluster of database instances, as well as having to synchronize the data. On the other hand, this approach does add another level of complexity to your deployments. You now need to check and maintain the controller as a part of the operator, to ensure that it runs as it should.

Control

When introducing an operator to your cluster, you do unfortunately need to let go of some control. The user will not have control over what exactly is deployed in its cluster - this is abstracted away by the operator. In addition, the operator strive to maintain a certain state so it may unwillingly overwrite user-added configurations.

Security

Security is always an aspect of high significance. However, using operators from trusted third-parties is per se not a security risk. The security risk is introduced when installing operators from non credible sources or when trying to build your own operator. Operators are very complex and difficult to design without introducing any risk of failures or security breaches considering that operators take advantage of core Kubernetes concepts; the API, controllers and Kubernetes resources. These concepts should not be tampered with, but treatd with caution.

Cost

When addressing the cost aspect of choosing an operator or not, there are few arguments towards avoiding it. Most operators are open source and those vendors

that offers commercial, complex operators to deploy their applications, tend to also offer a free, less complex operator. In addition, doing it yourself will cost both time and resources in order to collect all the necessary domain knowledge to manage without an operator. This involves both deploying the application as well as maintaining it.

Functionality

In addition to the above aspects to consider when standing upon the choice of applying an operator or not, there is one last major aspect to take into account. Applying an operator is actually critical for achieving the correct functionality of (most) databases in Kubernetes. Or at least if the intention of using Kubernetes for your application is to exploit its ability to scale. Kubernetes StatefulSet can not alone be let in charge of gracefully scale a set of database instances from, for example, 6 to 4. The StatefulSet is not aware of what database is running inside it, how the data is stored and what the procedure is for replicating the data in a correct way for that exact database. If this is something you want your cluster to manage, you will either need to create your own operator or use the operator provided by the database vendor.

2.6 Similar Research

In 2019 Olle Larsen, hereby referred to as Larsen, wrote his master thesis in cooperation with Umeå University in Sweden [23]. His thesis title was *RUNNING DATABASES IN A KUBERNETES CLUSTER* and it addressed how three different databases behaved in a Kubernetes environment by evaluating the impact some Kubernetes primitive operations had on each of them. In Larsen's case the operations to evaluate the impact from was scale in/out (horizontal scaling), scale up/down (vertical scaling), backup and version upgrade. Vertical scaling involves equipping the nodes with more computing power by for example increasing the amount of CPUs, RAM and/or memory. This process implies that the workload running on the particular node that is exposed to the vertical scaling, is moved to a new node that satisfies the new hardware specifications. Horizontal scaling means adding more instances to the system that can handle requests or workloads. The three databases Larsen investigated on was MySQL, TiDB and CockroachDB. The goal of Larsen's thesis was achieved if he could answer the following two questions:

- How large is the gap between Kubernetes capabilities and the requirement of stateful database services?
- How does different database solutions behave while being hosted in Kubernetes and operated on, and what is the impact of operating on both client performance and resource usage of the database?

2.6.1 The Approach

Larsen's approach to possessing results of the impact of the operations above started by deploying the databases on a managed Kubernetes cluster. He used the one provided by Google, GKE. When using GKE as the cluster provider he could also benefit from the features provided by the Google Cloud Platform, such as node pools. With node pools Larsen could dedicate different groups of machines to have different configurations. This allowed him to have dedicated nodes for monitoring, benchmarking operations and database workloads.

After having the cluster up and running, he deployed the three databases one by one using custom node pools for each database. This was necessary since he relied on having unique architectures for each database. Figures 2.2, 2.3 and 2.4 of MySQL-, TiDB- and CockroachDB-architecture respectively, illustrates how much the architectures of the three different databases varied. For the application deployment itself, Larsen utilized two custom operators to configure and manage two of the databases. For MySQL he used the Presslabs MySQL Operator, for TiDB he used the TiDB Operator and for CockroachDB he only utilized a StatefulSet since the single operator option for CockroachDB, named Rook, did not offer any additional valuable features.

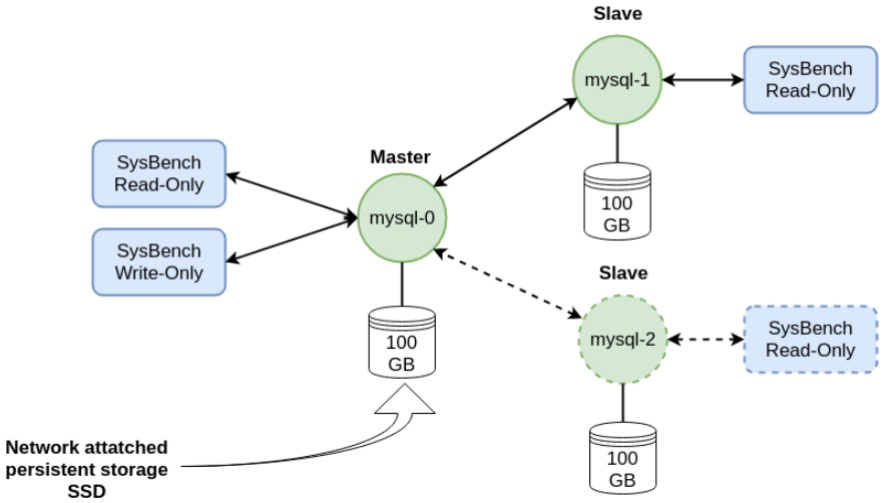


Figure 2.2: Overview of the experimental setup of MySQL.

For all the setups Larsen equipped the nodes dedicated to perform database-related tasks with a persistent storage in shape of a 100 GB, network attached, SSD persistent disk. The data that throughout the experiment was stored on these disks was generated by SysBench. SysBench is the tool Larsen used to benchmark the different databases in terms of client-sided metrics, being throughput and latency. Evaluating these metrics revealed how the client, being the database, handles the operations performed on the databases. In addition to generate the data and cap-

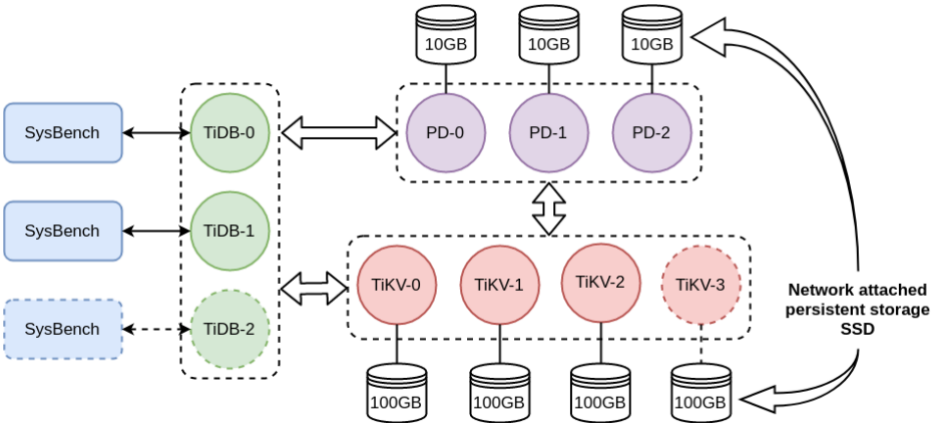


Figure 2.3: Overview of the experimental setup of TiDB.

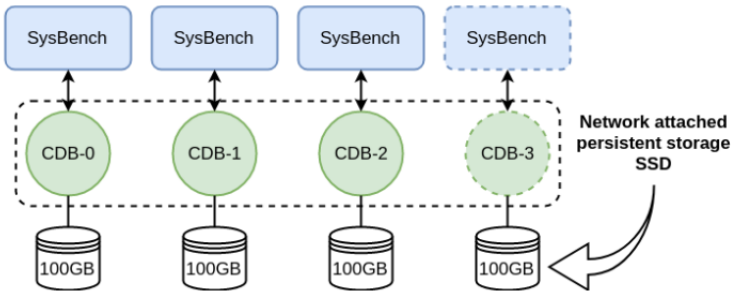


Figure 2.4: Overview of the experimental setup of CockroachDB.

ture the client-sided metrics, SysBench also injects the database with queries and transactions. The generated dataset consisted of 25 tables with 1 million rows per table, which is equivalent with approximately 6 GB.

While SysBench were able to gather the client-sided metrics, Larsen needed to set up a manual composition of services to derive the server-sided metrics. The services Larsen used in the composition was CAdvisor, Prometheus and Grafana. All the services were deployed inside each node handling database-operations. As CAdvisor only allows custom configuration when deployed as a stand-alone, the service was not deployed in a Kubernetes pod in contrast to Prometheus and Grafana. The two latter was easily deployed and configured with the Prometheus operator as an aid. CAdvisor acted as the metrics collector by observing and exporting the CPU, memory and network usage at fixed intervals from all containers running on the host machine. Notice that Larsen chose to observe the memory but when examining the results he discovered that the operations did not have a significant impact on the memory. Furthermore, Prometheus acted as the structuring unit that stored the collected metrics in a time series database. Lastly, Grafana

acted as the visualizer that queried the Prometheus database and aggregated the time series data before visualizing it with a set of graphs. With these services running, Larsen were able to observe and obtain how both the client-sided and the server-sided metrics was impacted when the system was exposed by the Kubernetes primitive operations mentioned initially in this section.

2.6.2 Results

Throughout his experiments Larsen were able to obtain valuable results. He presented his observations by focusing on one Kubernetes primitive operation at a time and discuss how the different databases' reactions to the particular operation varied.

When *scaling out* the Kubernetes cluster, Larsen could observe that the metrics that had the most impact on all the databases was network rates. Even though MySQL also experienced this impact, this database differed from the others when considering the lack of impact on the client performance. A limitation caused by the usage of the MySQL operator, which forced Larsen to reduce the number of transactions per second, may be the reason for the lack of impact on this metric. On the other hand it is quite clear why both CockroachDB and TiDB performed poorly in terms of network rates. Both databases is built on the principle of data sharding, meaning that they need to execute data rebalancing when database instances is added to or removed from the cluster. When the new node was added to the cluster it was the cause of high network *receive* rate and the three original nodes was the cause of high network *transmit* rate. This also caused both databases to experience high throughput and latency due to the new node immediately needed data from the other nodes to answer client requests. Time-wise the two databases were however very differentiable - CockroachDB spent 35 minutes on data rebalancing while TiDB only spent 5.

When *scaling in* the Kubernetes cluster, MySQL experienced no impact when removing a node due to no need for data rebalancing. MySQL only replicates the data that resides on one of the other nodes since its a relational database. CockroachDB and TiDB on the other hand experienced negative impact on both throughput and latency because of the same need to rebalance the data before the node could be terminated. Again TiDB proves to complete this process significantly faster than CockroachDB with times on 3 minutes and 9 minutes, respectively.

When *scaling up* the Kubernetes cluster, all databases applied a method called, rolling update, meaning that one by one instance of the database is terminated before deployed again on a new node with the improved computing power. All the databases had a positive reaction to the scale-up by being able to handle a higher throughput after the operation, which is an expected result. Another observation Larsen did was that there was a reduction in throughput for all databases while terminating each node since all clients that were connected to the database were not able to query the database during the rolling update. Hence, also an expected

impact.

When *scaling down* the Kubernetes cluster, all the databases reacted to the operation similarly to the scaling up, but in an inverted way. The throughput were now lowered after the rolling downgrade.

When performing a version upgrade on the Kubernetes cluster, similar observations as when scaling the nodes up and down was observed again. This is due to the method applied to the cluster during this process is the same as the method applied when scaling the nodes up and down; rolling update. The only difference is that the sate before and after this process is the same, in contrast to the vertical scaling where the throughput differs. A version upgrade involves replacing the database image running in the pod on the node to a newer (in most cases at least) image, which is in theory synonymous with a new version of the database software. In order to do so, the pods must terminate and be created again, hence the need for a rolling update.

When performing backup on the Kubernetes cluster, Larsen could observe different impact on the three databases. Again he could report lack of impact on the client-sided metrics of MySQL. CockroachDB and TiDB both experienced a negative impact in CPU usage since all the instances of the database needed to transmit data to a backup pod before this pod could stream the data to Google Cloud Storage. MySQL experienced some increase in network transmit rate because one of the nodes streamed backup to Google Cloud Storage.

Chapter 3

Method

In this chapter, the method applied in order to be able to answer the research questions presented in the introduction, is outlined in detail. The process is explained in a chronological order, from an exploratory testing environment in Minikube, to a complete and functional StatefulSet of three MongoDB instances running on a node each where the three nodes acts as members of a self-managed Kubernetes cluster. Throughout the explanation of the process, all challenges met and all possible solutions attempted, will be included. We will start by explaining how the exploratory testing went by. The second part, being the experimental environment, will be split up in two parts; the cluster setup and the application setup. While the cluster setup is quite straight forward, the application setup is more comprehensive, hence it will be split up in one sections per element in the setup process that turned out being, in some degree, complex. The elements that will be examined in detail are the following:

- MongoDB as a Deployment.
- MongoDB as a StatefulSet and volume assessments.
- Enabling communication between containers on different nodes.
- Enabling external connection.

After the setup outline, a section on some general considerations that was made throughout the process follows. Finally, a section highlighting the importance of exploiting valuable resources – especially in the tech world where trends and practices are changing faster than bitcoin changes in value – will find place. To be specific, the section explains how different knowledgeable people were invited to share their thoughts and ideas connected to problems encountered in this unorthodox approach of deploying a database in a Kubernetes cluster.

3.1 Exploratory Environment

As a part of the research phase in the subject TDT4501, Specialization Project, the *Minikube* platform were explored. Minikube is a technology made primarily for testing Kubernetes locally. The program allows a seamless setup of a single-node cluster where one can deploy running pods with kubectl, which is the command line interface to talk to Kubernetes.

Using Minikube made it easy to test the setup of a database – in this case MongoDB – in Kubernetes. While performing some simple queries, as well as manipulating the pods, it was possible to see that the volume plugins worked. Specifically, the database were created over three pods, data were inserted into a table in one pod, the pod were killed and restarted revealing that the previous inserted data were mounted back into the pod.

The exploratory phase could also reveal something else. After using Minikube, the plan was to test Kubernetes on the physical hardware assigned to the project by the supervisor. During this process a discovery was made. When bootstrapping the cluster an error occurred due to an insufficient amount of CPUs. The error stated the following: *[ERROR NumCPU]: the number of available CPUs 1 is less than the required 2*.

3.2 Experimental Environment

For the experimental environment, since the original plan expired due to insufficient amount of CPUs, cloud technology turned out to be the second best solution. In addition to being a platform for quick and simple access to VMs, it is also a more relevant approach in the today's industry. The choice of cloud technology fell on Microsoft Azure. Using this platform allows customizing a machine with a specific operating system, hardware specializations, disk type and size, as well as assigning a physical region of location. The machines customized for this experiment consisted of one master node, making up the Kubernetes control plane, and two worker nodes. The master node was a 2 CPU Linux machine with an Ubuntu 18.04 image, while the workers had 1 CPU each, and also Linux with Ubuntu 18.04 images. All machines was located in west Europe. For interacting with the machines SSH connection was set up using Azure portal. Also, inbound networking between the machines was set up using the portal. However, all direct interaction with the cluster was done using terminal commands, mainly via the control plane by passing kubectl commands from the master.

3.2.1 Cluster Setup

Before attacking the assignment on setting up the Kubernetes cluster without a provider in our back, one need to decide for an approach. All the alternative approaches involves using some sort of tools that installs necessary software on all

the machines that will be part of the cluster and run a diversity of commands that initializes and configures the cluster. The different approaches differs in the degree of manual impact required, and consequently the degree of freedom. More manual impact implies more freedom, and vice versa. The most manual approach involves using the command line tool, kubeadm, which is the lowest level of interaction that can be. Other approaches, being Kubespray and Kops, exploits kubeadm under the hood. Kubespray uses Ansible [1] for deployment and configuration, while Kops simplifies not only deployment and configuration, but also upgrading, maintaining and deleting the cluster. Kops is very valuable when aiming for a production-based, high available cluster. Seeing that this is an exploratory project, we have decided to go for the most manual approach; kubeadm.

Another consideration that one should make is what container runtime to choose. The common ones are containerd, CRI-O, Docker Engine and Mirantis Container Runtime. In context of this project the choice of container runtime is rather irrelevant since the choice does not impact the manual setup process in any way. Naturally enough the container runtime chosen in this project was Docker Engine, being the most common.

The process of setting up the cluster with kubeadm started trivially enough by possessing access to the three machines. This was done by storing their private SSH key, generated by Azure, on the local machine and SSH into the VMs. Then docker, kubectl, kubeadm and kubelet needed to be installed on all the machines before the cluster could be initialized from the master node with a *kubeadm init* command. A crucial part of this command is the option –pod-network-cidr. This option establishes a subnet on the cluster in terms of Container Inter-Domain Routing (CIDR) being 192.168.0.0/16, in this case, which is one of a few subnets that is accepted as a private address space. Also, swap was turned off, meaning that if the RAM memory space gets filled up, inactive pages *are not* moved to swap space to release memory to other processes. This is a common configuration when working with Kubernetes due to Kubernetes being a highly automated technology that determines the best available node on which to deploy newly created pods. If memory swapping is allowed to occur on a host system, this can lead to performance and stability issues within Kubernetes [25]. It is also necessary to open ports on the nodes so the Kubernetes components can communicate; this is especially important for the master which must open a port to the api-server, ETCD datastore, kube-scheduler and kube-controller which together acts as a cluster-brain. Tables 3.1 and 3.2 lists all the necessary ports that must be opened through the Azure portal and what their functions are. When initializing the cluster from the masternode using the masters private IP-address, the output is a *kubeadm join* command with a token that are run from the workers to add them to the cluster. Although being added, the workers maintains in a temporarily NOT READY state. Finally a Container Network Interface (CNI) is set up by applying an image for a

[1] Ansible is a software tool that provides simple but powerful automation for cross-platform computer support [24]

specific CNI provider on the masternode, which causes the workernodes to change state to READY.

Name	Port	Purpose
kube-controller-manager	10252	Lets the kube-controller-manager continuously control the cluster by moving the current state closer to the desired state
kube-scheduler	10251	Lets the kube-scheduler assign pods to nodes
kubelet-API	10250	Lets the node act as a Kubernetes node
etcd-server-client-API	2379-2380	Enables communication between kube-apiserver and ETCD
kubernetes-api-server	6443	Enables creation and manipulation of all Kubernetes components

Table 3.1: Inbound ports on the master node.

Name	Port	Purpose
nodeport-services	30000-32767	Enables external access to the services inside the cluster
kubelet-API	10250	Lets the node act as a Kubernetes node

Table 3.2: Inbound ports on the worker nodes.

Deploying a CNI to the cluster is a significant step for enabling network connectivity and network security policy enforcement between workloads [26]. Such interfaces are used when it comes to workloads for containers, VMs, and bare-metal, and there are many to choose from. In this experiment, Calico, was used as the CNI. When using a CNI in a Kubernetes environment Calico works as an interface between network providers and Kubernetes pod networking. Applying a Calico image to the cluster creates a new Deployment with a controller pod and a coredns pod, including one calico-node pod on each node in the cluster. A crucial effect of applying a CNI to your cluster is that future pods deployed on the cluster will get assigned IP-addresses in the established subnet from the kubeadm init command. Figure 3.1 illustrates how the machines in the cluster is configured at this stage.

3.2.2 Application Setup

The process of setting up MongoDB in the cluster is in theory mostly plug-n-play when using Kubernetes. This however, only applies if all the different YAML files contains all the necessary fields and if their values match the values of other necessary fields. With an architecture, a combination of resources and an application choice that in some way differs from – apparently – all existing tutorials, things

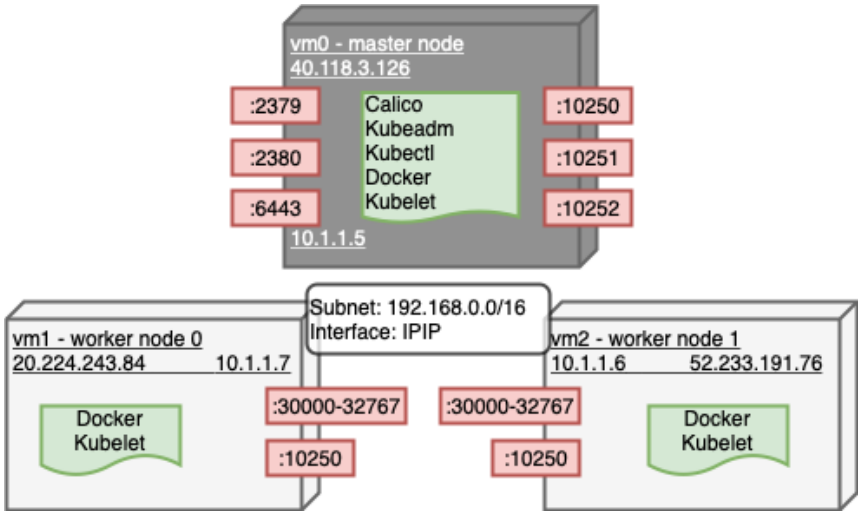


Figure 3.1: Cluster setup of three Azure VMs.

got difficult. First, deploying a self-managed Kubernetes cluster, without any third party vendor such as AKS, EKS or GKE, lacks documentation and troubleshooting tips. Yet we were able to set up a functioning cluster as explained above. Second, deploying databases at all in a self-managed Kubernetes cluster – let alone, MongoDB – is clearly not an everyday-task for developers, leaving behind little documentation on this topic as well. Third, deploying MongoDB in a self-managed Kubernetes cluster with a local disk as the volume plugin, is an even rarer topic with even rarer incidents of suitable tutorials. Also, enabling both internal and external communication for a MongoDB replicaset in a self-managed cluster with Calico as the CNI, is – not very surprisingly – a niche knowledge. Finally, there exists different approaches of running MongoDB on Kubernetes; as a StatefulSet, a Deployment or a ReplicaSet, which consequently has different standards of provisioning storage, but *what* differences are however not trivial. In summary, there does not exist a custom tutorial for this setup, so we needed to create our own.

MongoDB as a Deployment

Before applying a critical eye on the online tutorials, a beginners mistake is to follow them to the letter without investigating that the prerequisites are fulfilled and the tutorials goal matches yours. The first step in the right direction in the process of deploying a three-node replicated MongoDB application on a Kubernetes cluster was to mistakenly deploy a Kubernetes *Deployment* with 1 replica, meaning 1 pod, which of course was not the intention in a three-node cluster. A seemingly obvious solution was to adjust the replica amount to 3. See output after applying the Deployment in Figure 3.2. A first glance at the output of the command in the Code listing 3.1, which you can see in Figure 3.3, made us believe that all pods are running and the deployment was successful. This is in some way

true, but this setup does not satisfy the architecture that was aimed for, in terms of being a replicated MongoDB cluster. At this stage the three replicas are totally independent instances of MongoDB that will not be treated as one stateful application by the Kubernetes API. Hence, if one of the replicas would crash Kubernetes would spin up the replaced pod on a random node with a random name, providing no guarantees that the initial PV, that was once bounded to the PVC created along with this new pod, will find the PVC so the data can be mounted back into the pod. Another issue with this setup, was that all the pods used the same PVC and PV which will lead to data corruption and strange MongoDB behavior [27]. An example event that could arise is that the pod on worker0 adds a document to the database, leading to an update of the local volume on that node, and if that pod failovers to a different node it will mount back the data on the local volume on the new node - meaning that the previous added document will be out of reach until a pod is scheduled on the original node. The fact that all pods running on the workernodes would eventually die and spin up on the masternode, could also reveal that the usage of "HostPath" as the volume plugin for the PV (which was a spec designed for the original Deployment of 1 pod) only work in a single-node cluster, whereas in a multi-node cluster "local" volume should be the equivalent [17]. Lesson learned from this was that Kubernetes Deployments are suited for stateless applications or when multiple replicas of pods running stateful workloads can use the same volume. It is, however, possible to make this setup work with a workaround, but that would require a lot of manually management including creating a ReplicationController and a service for each replica. Such a solution is a pitfall of failure when scaling the set up and down. Figure 3.4 illustrates how the architecture became when deploying MongoDB to the cluster, even though it was not a functioning architecture.

```
andrea@master-node-0:~/deployment-kubernetes-mongodb$ kubectl apply -f .
deployment.apps/mongo-client created
deployment.apps/mongo created
service/mongo-nodeport-svc created
persistentvolume/mongo-data-pv created
persistentvolumeclaim/mongo-data created
secret/mongo-creds created
```

Figure 3.2: MongoDB applied on cluster.

Code listing 3.1: Command for viewing pods details.

```
andrea@masternode$ kubectl get pods -n <namespace> -o wide
```

MongoDB as a StatefulSet and Volume Assessments

Discarding Kubernetes Deployments as an approach to our application setup, the next obvious Kubernetes concept to apply is a StatefulSet. A StatefulSet is a Kubernetes object introduced with Kubernetes 1.5 that simplifies the running of stateful workloads in Kubernetes [28]. When applying a StatefulSet to a Kubernetes

```
andrea@master-node-0:~/deployment-kubernetes-mongodb$ kubectl get pods -n mongo-deployment -o wide
NAME                                READY   STATUS    RESTARTS   AGE     IP               NODE            NOMINATED NODE   READINESS GATES
mongo-6f78446cd8-fsvxj              1/1     Running   0          2m20s   192.168.196.184   master-node-0   <none>           <none>
mongo-6f78446cd8-fvn6l              1/1     Running   0          2m20s   192.168.204.75    worker0         <none>           <none>
mongo-6f78446cd8-jx4l4              1/1     Running   0          2m20s   192.168.235.153   worker1         <none>           <none>
mongo-client-6c7bc768c4-2gjhv       1/1     Running   0          2m20s   192.168.204.76    worker0         <none>           <none>
mongo-client-6c7bc768c4-mpsr6       1/1     Running   0          2m20s   192.168.196.183   master-node-0   <none>           <none>
mongo-client-6c7bc768c4-pbrv2       1/1     Running   0          2m20s   192.168.235.152   worker1         <none>           <none>
```

Figure 3.3: MongoDB status immediately after applying.

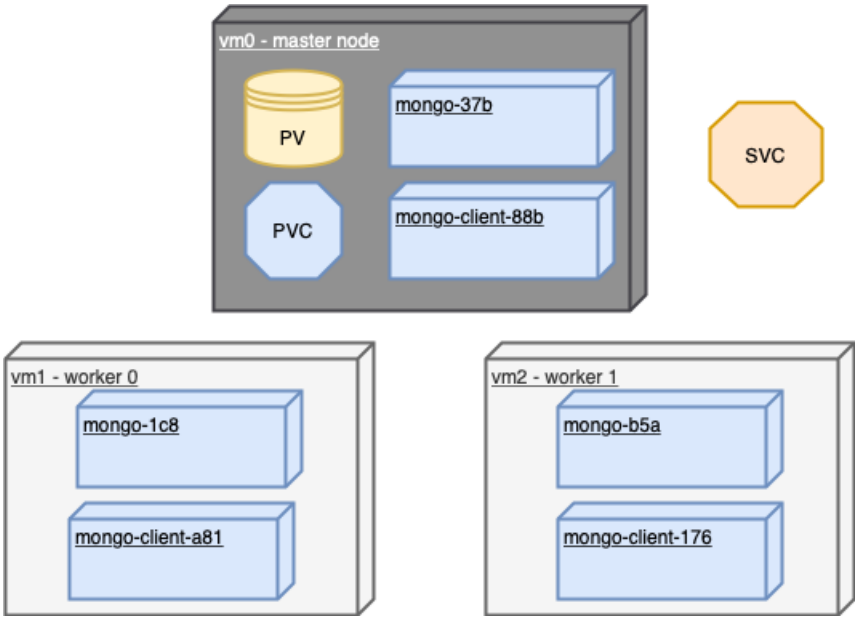


Figure 3.4: MongoDB applied to the cluster as a Deployment.

cluster, the process of how objects gets created differ from a Deployment or a ReplicaSet. A ReplicaSet is a Kubernetes object that creates identical individual pods, which is the lower-lever object that Deployments use to delegate the pod-creation. Being created as individual pods, these pods can all spin up separately. StatefulSets on the other hand has the unique property that its pods is assigned stable DNS names, which allows other Kubernetes object to address the pods by their pod-names. In contrast, pods created by Deployments gets randomly generated DNS names. The unit responsible for assigning the pods with stable DNS names is the Headless Service. A headless service is added to a cluster by applying a YAML file similar to a standard Kubernetes Service, except that its ClusterIP has the value *none*. Since a StatefulSet, in contrast to Deployments and ReplicaSets, creates its own uniquely named pods and corresponding PVCs, a StatefulSet is created replica by replica. This means that if the first replicas PVC binds to the, hypothetically, only existing PV, the following replica will enter an infinitely pending-mode and the third replica will never be created. Before being aware of this behavior of StatefulSets, the exact scenario did in fact apply. As an attempt to correct this state we created three identical PVs using "local" as the volume plugin. When spinning up the StatefulSet with this foundation, an error saying

that *a persistent volume with name local-pv already exists* stopped the process. Assigning the PVs unique names on the other hand resulted in the PVCs, that are made from a single PVC Template, did not find the general name of the PV that they were configured to bind to according to the pod specifications. The solution was to use a Storage Class that automatically will assign appropriate PVs to PVCs independently of their names not matching, along with the manually defined local PVs, while adding "storageClassName" to the the PVC template. When using local storage on each node as the volume plugin, the Storage Class provisioner one must specify is "no-provisioner". Applying the StatefulSet YAML to the cluster with the new starting point resulted in all pods having PVCs that were bound to their own PV, as intended, but unintentionally all pods did now run on the same node. Luckily applying NodeAffinity to the specifications of the PVs forced them to be bound to a specific node, which provided us with an ideal architecture of our purposes. Furthermore, there is a necessity of having an architecture that is able to assign a pod-personal volume for each pod, as achieved with StatefulSets compared to Deployments, when running MongoDB as a replicaset over multiple pods in Kubernetes. An illustration of the functioning StatefulSet along with the resulting volume assessment can be found in Figure 3.5. To see the complete configuration files of the StatefulSet, the headless service, the PV and the storageclass turn to Appendices B.4, B.2, B.7, B.8, B.9 and B.6.

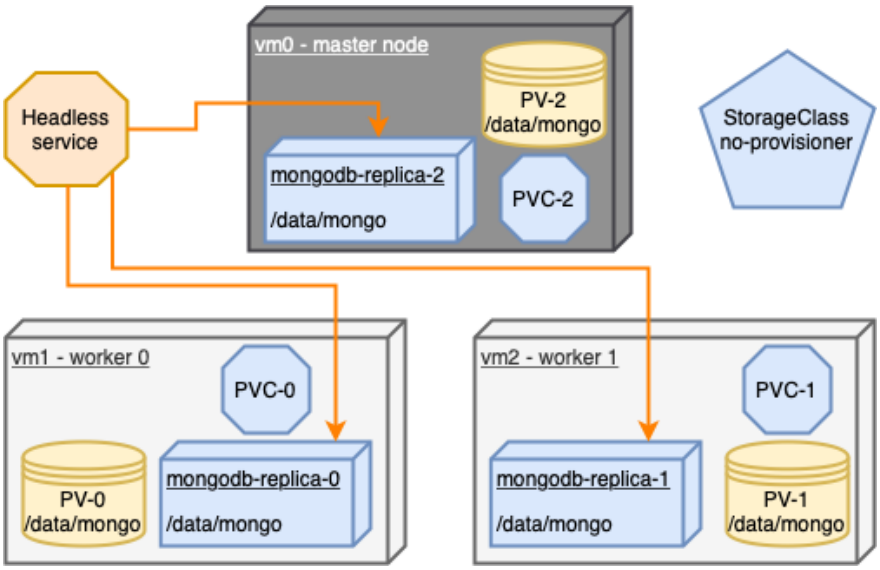


Figure 3.5: MongoDB applied to the cluster as a StatefulSet.

Enable Communication Between Containers on Different Nodes

Even though obtaining an architecture that confirms all theoretical checks of a checklist, the MongoDB replicaset was not working as expected. For the three MongoDB instances to communicate by demands from MongoDB and not through

the Kubernetes API, they need to be initialized as a replicaset. This is typically done from inside each MongoDB instance, but since this application is setup with Kubernetes, the command lines that issues this process can be done from the pod specifications, as showed in Figure 3.6 of a snippet of the pod specification. Before this was interpreted as a necessity, all seemingly unnecessary fields in the StatefulSet YAML was removed in order to have a minimum valuable product of the application that was functioning without introducing any other dependencies. However, when discovering that a replicaset needed to be initialized, the command lines above were added to the pod spec. Still, it turned out to be problematic adding the instances to the replicaset initialized from the host. The host were successfully defined as the primary instance, while the two other instances were not able to locate the host and hence did not become a part of the replicaset. When applying the command *rs.add("<pod-name>.<headless-service-name>:<port-number>")* MongoDB returned an error saying *NodeNotFound*. Considering that Kubernetes components communicate by referring to a service in the same namespace and the name of the pod followed by the port of the relevant container-image, this response was rather peculiar.

```
spec:
  terminationGracePeriodSeconds: 30
  containers:
  - name: mongodb
    image: docker.io/mongo:4.2
    env:
    command: ["/bin/sh"]
    args: ["-c", "mongod --replSet=rs0 --bind_ip_all"]
```

Figure 3.6: Initializing the MongoDB replicaset with commands in pod specification.

As described in Section 2, one can apply RBAC to the Kubernetes cluster that grants access to specific resources only to specific cluster administrators through components such as roles, role bindings and service accounts. In this stage of the process, RBAC was set up in the cluster to grant access to the StatefulSet. This did however not prevent the *NodeNotFound*-error from occurring when trying to add the secondary instances to the MongoDB replicaset. Refer to Appendix B.5 to see the complete YAML file of the RBAC specification.

Another possible workaround was investigated further in the process. Deploying a helper-container is common procedure in the Kubernetes world. Such helper-containers are often referred to as a *sidecar-container*. Sidecar containers can be used in many circumstances where it is convenient to abstract some of the Kubernetes specific additional functionality of the main container. This can be when a container should be coupled up with a storage unit, but one wishes to facilitate for

a situation where the application that runs in the main container can be replaced and the logic that connects the application to the storage unit must remain. Other usecases of a sidecar can be scheduling, synchronizing, back-up or authentication. A sidecar also allows replicas to be added and removed from the MongoDB replicaset automatically when scaling the StatefulSet. In this particular case we wish to facilitate for the situation where we have multiple MongoDB containers that must initialize a replicaset amongst them. Implementing a sidecar container in all the pods in our StatefulSet allows us, in theory, to abstract the manual workload connected to initializing the replicaset. In our project a MongoDB sidecar image created by cvallance were added to the pod spec in the StatefulSet YAML file. The StatefulSet was successfully applied and the new pods with two containers per pod were created and running in the cluster. Still, when running kubectl exec on the pod that was assigned the ordinal number 0 – meaning it was addressed with the role as the PRIMARY instance in the replica set – the replicaset was not successfully initialized. This was revealed by running rs.status() from within the MongoDB container. See Figure 3.7 for a snippet of the resulting output of the command. The list of members only consists of the IP-address belonging to the MongoDB instance running on worker 0. However, it is clear that the sidecar container did contribute to something. Before introducing the sidecar, either of the MongoDB-instances acted as a primary instance without manual configuration. Also, the logs of the pods could prove even more contribution. The logs from each of the pods were examined to see what configurations and commands that automatically was issued. Running the *kubectl logs <pod-name> -c <container-name>* command provides us with all terminal outputs from the container during the creation. From mongodb-replica-0 running on worker 0 we could see that lots of network connections was attempted established between the MongoDB pods and some localhost connections (probably to the sidecar container). In addition we see that the container prints a configuration field called *replSetReconfig* that lists all the members that are supposed to be added to the replicaset. In the end of the line we can also see an error message that tells us that the nodes was not able to reach each other within the time limit. To see the complete snippet of the logs from worker 0, turn to Appendix A.2.

After investigating the liveness of the network-connections between the pods, it was discovered that the error was caused by a problem on a higher level. The investigation involved installing a diversity of networking packages to troubleshoot the networking capability. A part of this stage involved updating the running version of Calico, the CNI in the cluster. When running *calicoctl get ippools -o wide* command, in order to see the ip-pools that Calico had established in the cluster, the system revealed that Calico client and the cluster used different versions; 3.22.0 and 3.22.1 respectively [2]. Both packages was updated to 3.23.0, but the error was still present. Anyway, after updating Calico, the output from the command above

[2]This command only works after applying the alias; alias calicoctl="kubectl exec -i -n kube-system calicoctl – /calicoctl" to the .profile-file on the node.

Figure 3.7: Output when running rs.status() command from MongoDB on the PRIMARY MongoDB instance on worker0.

got more informative as seen in Figure 3.8.

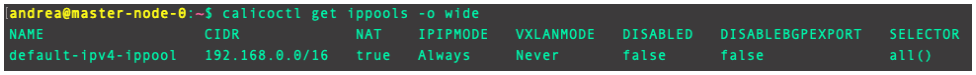


Figure 3.8: Calico ippools output.

When attempting to install debugging-tools on the pods, a groundbreaking observation came to our attention. It got clear that the pods running on the worker nodes in the cluster were not able to run apt-get, which implies that they were not able to install any packages. As an attempt to bypass this issue a custom MongoDB image was built which included commands that installed networking packages like net-tools and iproute2. The image was created from a manually written dockerfile which can be seen as a whole in Appendix B.1. After the image was built, it was pushed to a private repository on Dockerhub and added to the pod spec of the StatefulSet. Unfortunately, this attempt did not lead to any network access on the pods on the worker nodes even though they now were able to run commands like ip a and route -n. These commands could however reveal that the pods did not have any gateways connecting them together. As seen in Figure 3.9, the bottom two routes only reached the gateways to the local IP-addresses of the worker nodes (10.1.1.6 and 10.1.1.7) through the tunl0 interface. Hence, none

of the pods on the workers can be reached from the master node.

```
andrea@master-node-0:~$ route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         10.1.1.1        0.0.0.0         UG    100    0        0 eth0
10.1.1.0        0.0.0.0         255.255.255.0   U     0      0        0 eth0
168.63.129.16   10.1.1.1        255.255.255.255 UGH   100    0        0 eth0
169.254.169.254 10.1.1.1        255.255.255.255 UGH   100    0        0 eth0
172.17.0.0      0.0.0.0         255.255.0.0     U     0      0        0 docker0
192.168.0.0     -               255.255.255.0   !     0      -        0 -
192.168.2.0     10.1.1.6        255.255.255.255 UGH   0      0        0 tunl0
192.168.196.128 0.0.0.0         255.255.255.192 U     0      0        0 *
192.168.196.134 0.0.0.0         255.255.255.255 UH    0      0        0 caliade9e914151
192.168.196.136 0.0.0.0         255.255.255.255 UH    0      0        0 calidfa85360b17
192.168.196.152 0.0.0.0         255.255.255.255 UH    0      0        0 cali872fd71a728
192.168.196.153 0.0.0.0         255.255.255.255 UH    0      0        0 calif0d98d85877
192.168.204.64  10.1.1.7        255.255.255.192 UG    0      0        0 tunl0
192.168.235.128 10.1.1.6        255.255.255.192 UG    0      0        0 tunl0
```

Figure 3.9: Output of route -n from master node before editing ippools.

The interpretation of this result was that the networking on the pods on the worker nodes were not configured correctly. Another consequence of this was that the pod on the master node were unable to ping the pods on the worker nodes. In order to locate the root of this issue, network specifications on Azure, in the context of Calico as the CNI, was investigated. It became clear that this exact cooperation had contradictory networking configurations by default. A section of the documentation of Calico that describes its interaction with Azure says that *Calico in VXLAN mode is supported on Azure. However, IPIP packets are blocked by Azure network fabric* [29]. Revisiting Figure 3.8 we can see that Calico defaults to always allowing IPIPmode and never allowing VXLAN. This conflicts with Azure that blocks all IPIP packets which causes the networking issues from the pods on the worker nodes. By simply changing the configuration file for Calico ippools with *kubectl edit ippools.crd.projectcalico.org*, as seen in Figures 3.10 and 3.11, Calico will now do its internal container networking through the VXLAN interface instead of IPIP. Comparing Figure 3.9 with Figure 3.12 we can see that both the gateways and the interfaces of the two bottom routes have changed as a result of this.

After establishing a functioning internal container communication between the pods, the replicaset got initialized automatically thanks to the previously added MongoDB sidecar. When running rs.status() from one of the pods at this stage, all the pods' IP-addresses are listed as members. For the full output of this command see Appendix A.1. An illustration of the architecture at this stage is shown in Figure 3.13.

Enable External Connection

So far all interactions and management of the MongoDB service and the Kubernetes objects have been done from within the cluster - more specific with kubectl

```
apiVersion: crd.projectcalico.org/v1
kind: IPPool
metadata:
  annotations:
    projectcalico.org/metadata: '{"uid":"d
  creationTimestamp: "2022-05-15T16:01:58Z
  generation: 5
  name: default-ipv4-ippool
  resourceVersion: "11159401"
  uid: 3a1a9b7a-326f-4762-8104-5d0dcd4660c
spec:
  allowedUses:
  - Workload
  - Tunnel
  blockSize: 26
  cidr: 192.168.0.0/16
  ipipMode: Always # Change to Never
  natOutgoing: true
  nodeSelector: all()
  vxlanMode: Never # Change to Always
```

Figure 3.10: Editing the configuration for Calico ippool.

from the master-node. If it is desired to expose a Kubernetes service public, the first most important thing is to make sure the api is protected with authentication or other policies. At this stage of the process, this was however not a priority or a consideration at all. And since this project will remain as a testing environment running on Azure's machines, it is not an emergency anyway. The process of exposing the application, ignoring the security aspects, started unconsciously while initializing the cluster. A part of the initialization was to open inbound ports on the nodes through the Azure portal. When the ports 30000-32767 was opened for NodePort Services, the possibly created services, that – if existing – acts as an entry point to the application, is available. Although exposed with open port, no service, except from the headless service, was established at this point. The headless service can not act as a service since it does, by default, not obtain any clusterIP meaning it is not accessible by any instances either inside, nor outside of the cluster. The DNS server will not interpret the headless service as an actual service that clients can connect to pods through, so the DNS server responds to clients DNS lookups with the IPs of each individual pod instead. But in order to

```
andrea@master-node-0:~$ calicoctl get ippools -o wide
NAME                  CIDR            NAT    IPIPMODE   VXLANMODE   DISABLED   DISABLEBGPEXPORT   SELECTOR
default-ipv4-ippool   192.168.0.0/16   true   Never      Always      false      false              all()
```

Figure 3.11: Calico ippools output after edited ippool configuration.

```
andrea@master-node-0:~$ route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         10.1.1.1        0.0.0.0         UG    100    0        0 eth0
10.1.1.0        0.0.0.0         255.255.255.0   U     0      0        0 eth0
168.63.129.16   10.1.1.1        255.255.255.255 UGH   100    0        0 eth0
169.254.169.254 10.1.1.1        255.255.255.255 UGH   100    0        0 eth0
172.17.0.0      0.0.0.0         255.255.0.0     U     0      0        0 docker0
192.168.196.128 0.0.0.0         255.255.255.192 U     0      0        0 *
192.168.196.134 0.0.0.0         255.255.255.255 UH    0      0        0 caliade9e914151
192.168.196.136 0.0.0.0         255.255.255.255 UH    0      0        0 calidfa85360b17
192.168.196.152 0.0.0.0         255.255.255.255 UH    0      0        0 cali872fd71a728
192.168.196.153 0.0.0.0         255.255.255.255 UH    0      0        0 calif0d98d85877
192.168.204.64  192.168.204.79  255.255.255.192 UG    0      0        0 vxlan.calico
192.168.235.128 192.168.235.145 255.255.255.192 UG    0      0        0 vxlan.calico
```

Figure 3.12: Output of route -n from master node after editing ippools.

do a DNS lookup against the MongoDB application, the cluster needs a service inside the cluster to connect to first. A Kubernetes service can be configured with fours different types; ClusterIP, NodePort, LoadBalancer or ExternalName. The chosen type results in a certain degree of exposure, whereas the clusterIP-type maps to internal cluster exposure. NodePort maps to external cluster exposure by being an extension of the simultaneously created clusterIP services for each node. External traffic can access your service by connecting to the NodePort on any of the node's IPs, and then the node forwards the traffic to the service which handles it according to other configurations. The LoadBalancer type also maps to external cluster exposure but with yet another extension, cloud-providers custom LoadBalancer. The ExternalName type also maps to external cluster exposure, but to a DNS name like an externally running database. So in our case, we would prefer a LoadBalancer type, but since that type is highly dependent on a cloud-provider, it is not applicable in a self-managed cluster. Surely, we can create the LoadBalancer service, but it will not be assigned an external IP-address since we do not rely on a provider. Figure 3.14 illustrates how the LoadBalancer will not get any external IP-address, but still gets connected to the StatefulSet through the selector: app=mongo, which is equal to the label on the StatefulSet. Seen that the LoadBalancer would get an external IP-address, additional components would be required for correct behavior anyway. For example, if sending a write request to the LoadBalancer service, the service would not be able to determine if the request should be redirected to the primary or secondary MongoDB instances. In fact, the service would not even know what pods that corresponds to what MongoDB replica. To make this alternative architecture to function, we could have added an intermediary MongoDB client responsible for redirecting read/write requests to the correct instances. Turn to Figure 3.16 illustrating how the external connec-

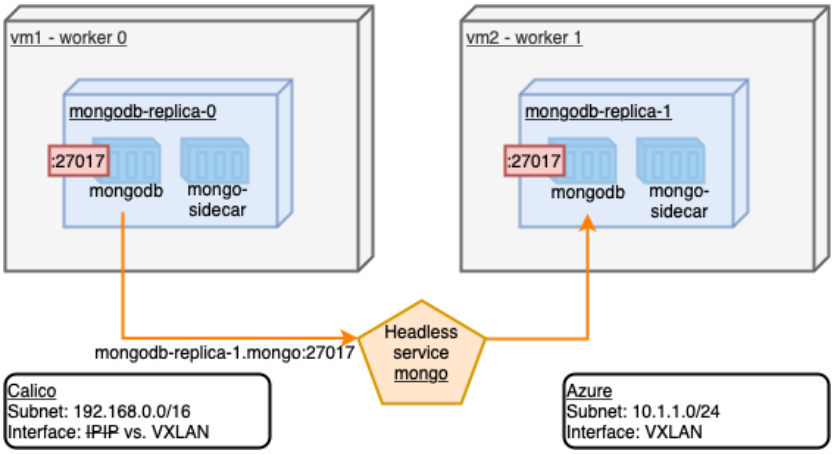


Figure 3.13: How internal pod-communication is achieved in an Azure/Calico environment.

tion could be achieved with this approach. However, one can exploit temporary solutions such as port-forwarding to verify that the application is accessible from outside the cluster as well. With port-forwarding a port on the master node is opened and connected to a desired port on the pod in the cluster. In this case the port 32000 was opened on the master and mapped to a pod on port 27017, which is the default MongoDB port. Figure 3.15 shows how the connection to MongoDB is established through port-forwarding from the master node without kubectl.

```
andrea@master-node-0:~/cluster-wide-config$ kubectl get svc -o wide
NAME         TYPE           CLUSTER-IP     EXTERNAL-IP   PORT(S)           AGE    SELECTOR
kubernetes   ClusterIP      10.96.0.1      <none>        443/TCP           57d    <none>
mongo        ClusterIP      None           <none>        27017/TCP         7d2h   app=mongo
mongo-lb     LoadBalancer   10.108.45.44   <pending>     27017:30165/TCP   21s    app=mongo
```

Figure 3.14: Status of LoadBalancer service running in cluster.

```
andrea@master-node-0:~$ mongo localhost:32000
MongoDB shell version v5.0.6
connecting to: mongodb://localhost:32000/test?compressors=disabled&gssapiServiceName=mongodb
```

Figure 3.15: External connection to MongoDB from the master node with port-forwarding.

3.2.3 General Considerations Concerning Manual Setup

Throughout this process we also brought some general considerations to the light, that might be trivial for an expert in operating MongoDB in Kubernetes.

One of these considerations was whether or not workloads should or can run on the master node, or if such nodes should be dedicated to running cluster administration tasks. We can confirm that it is no physical limitation connected to the master node that restricts it from running application specific workloads, but on

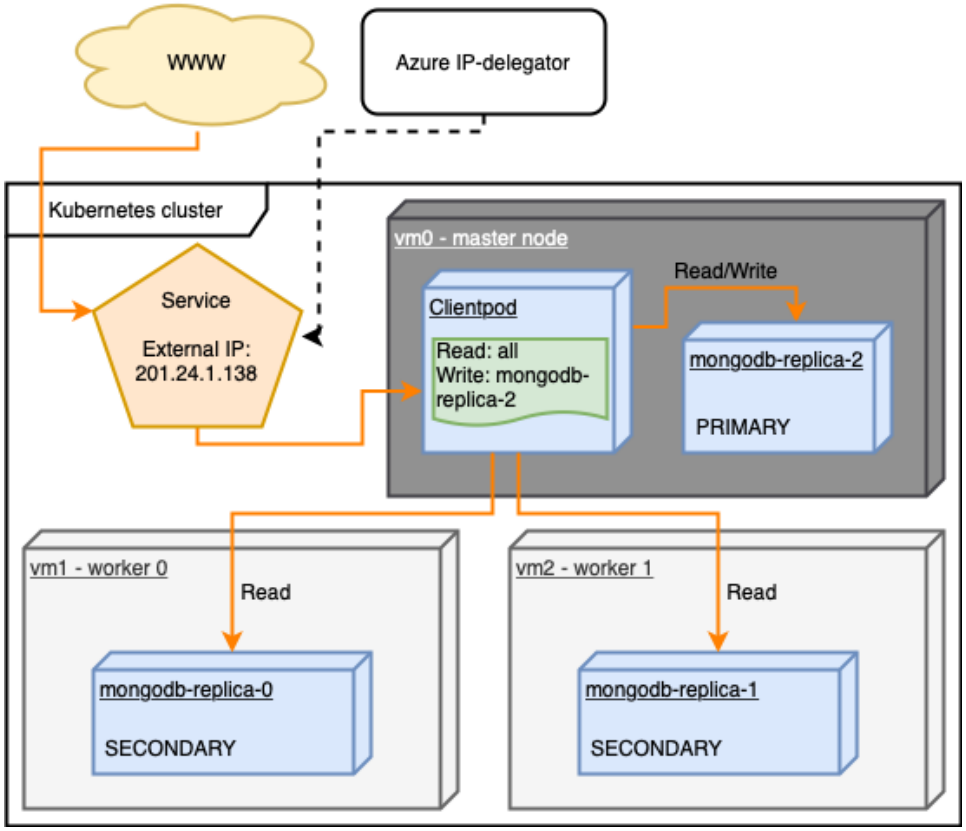


Figure 3.16: A possible solution to gaining external connection to the application.

the other hand it is not recommended in all circumstances. When operating on a small cluster running non-critical and minimal resources-heavy workloads, it is in general no problem to take advantage of the master to contribute with application workloads. Said differently, if the master node has sufficient resources available, in terms of CPU and RAM, that the critical administrative tasks are maintained, it can also run workloads.

Another consideration that must be addressed is whether to initialize MongoDB with replication or sharding. If MongoDB runs as a replicated database across multiple instances all the data residing on each instance is the same[30]. This is achieved by assigning one instance as a primary instance that handles all write and update requests and is responsible for replicating the new data to the other secondary instances. All instances can however respond to read requests. Sharding on the other hand is conceptually differing by splitting the data over multiple instances, resulting in all instances holding subsets of the whole dataset. In a sharded MongoDB database all instances can respond to both read and write requests, but if an instance receives a request asking for data that does not reside on that particular instance, it must ask the other instances for the requested data.

Also, the roles and function of the different instances vary from a replicated cluster, at least if one strive for configuring a sharded cluster following the practice used by MongoDBs own Kubernetes operator. Such a sharded cluster needs one or more configuration servers, mongos instances[3] and shard members, where a StatefulSet is deployed for all the configuration servers, another StatefulSet for all the mongos instances and one StatefulSet for each of the shard members. Which configuration to choose for your MongoDB application in a Kubernetes cluster is again depended on your usecase. If aiming for a simple highly available database one should implement the database as a replicaset. If in need of a larger database that handles horizontal scaling well one should implement the database as a sharded cluster. Evaluating the assumed amount of reads and writes is also of significance in this decision. If you expect your application to process many reads and few writes you should consider the replicated cluster since all instances can respond to read requests without interacting with other instances. A sharded cluster will most likely require a lot of inter-instance communication to respond to many read requests, hence a replicated cluster handles many reads better than a sharded cluster. If you in contract expect your application to process many writes and few reads, you should consider a sharded cluster since all instances can respond to write requests without interacting with other instances. In a replicated cluster on the other hand, the primary instance will act as a bottleneck in this scenario. Seen in context with the this exploratory project we chose the replicaset with its perk of involving a less complex setup. Since the database is not suppose to operate, the read and write ratio is of little importance.

Understanding the Kubernetes architecture in terms of pod and container connection and communication, is also a barrier to managing applications hosted in Kubernetes manually. When setting up applications in Kubernetes without using operators one need to – in some cases at least – get direct access to the application running in the container and manipulate it from its own shell interface. In order to do this one must *exec*[4] into the application. However it is not intuitive that one can exec into the pod and run commands that applies to the application running inside the container inside the pod. Even more confusion can be introduced when wanting access to a container from a worker node considering that kubectl is not installed on this node. In such cases docker, the container runtime, is applied directly to exec into the container. It soon became clear that when execing into the pods, Kubernetes takes care of the further connection to the container in the background and selects the main container. In practice, execing into the pod with kubectl and execing directly into the container with docker, is equal actions.

[3]Mongos instances are responsible for the interface between the client applications and the sharded cluster[31]

[4]Exec is a kubectl and docker option that lets you execute commands from inside a selected container.

3.3 Information Gathering

As part of the process some guidance were provided along the way. Considering that the technology applied in this thesis is quite immature there are limited sources of adequate knowledge out there to cover the specific topic in the already limited research field. However, we came across some knowledgeable people who were able to contribute, in some degree, in the technical challenges met in this thesis. In the tech community the eager to share knowledge is – from a subjective point of view – unique. At least, an earlier professor at the University at Tenerife, a colleague from Sweden and an employee from GeoData in Trondheim, were very willingly to share their thoughts on the subject and their ideas to solutions to the technical challenges met.

The professor at the university at Tenerife was helpful in exchanging thoughts within the very beginning of the project. Discussing the core of the topic considering managing databases in a Kubernetes cluster in general, were helpful in order to narrow the project down to something that would be feasible and still provide value to the tech community.

The colleague from Sweden who had some experience working on Kubernetes in past projects gladly shared his thoughts further down the line in the setup process. Being familiar with the Kubernetes concepts and some of the logic in how they are configured in terms of functionalities and network communication, his expertise was exploited when trying to enable external communication between the cluster and the outside world. Even though being knowledgeable he were only able to contribute with rubberducking and introducing tools to troubleshoot the issues. His expertise were unfortunately limited to Kubernetes clusters managed by providers where issues concerning external communication are handled by providers by allowing the Kubernetes service type LoadBalancer. In addition he had never worked with databases deployed in a Kubernetes cluster, and let alone, MongoDB. Actually he advised against running databases inside a Kubernetes cluster in general.

The employee from GeoData Trondheim on the other hand worked with managing a PostgreSQL database in a 100+ nodes large Kubernetes cluster on a daily basis. It is worth to mention that this database were hosted on an AWS managed cluster which makes the basis for comparison to this particular case slightly less valuable. Despite his lack of specific domain knowledge he confirms the impression of the unique willingness to share knowledge and to help out a fellow developer in the tech community. His expertise were helpful in the stage of the process involving enabling internal communication between the pods in order to successfully create the MongoDB replicaset. After some slow and ineffective correspondence over remote channels of communication, a physical meeting were established where the internal networking problem could be attempted solved in a much higher pace.

Chapter 4

Conclusion

In this chapter we will use the experiences and lessons learned from Chapter 3, Method, to present arguments constructing a conclusion to the research questions formulated in the introduction. First, we will summarize the manual process in setting up MongoDB in a self-managed Kubernetes cluster without an operator with a smooth transition to a presentation of the final setup. Following up, the research questions will be answered with a detailed justification, whereas the answers will reflect the actual conclusion to the master thesis.

4.1 The Final Setup

Considering each of the visited considerations from the method chapter one by one, we will now reflect on what resulted to be the actual implementation of the relevant consideration and why this one was chosen for this particular project. Then we will reflect on the lessons learned about the complexity or simplicity regarding the concept.

Cluster Setup

Regarding the cluster setup, there were few alternative approaches to choose from when settling for a manual and self-managed approach. In this project we built the cluster using kubeadm since it was the most metal-close approach available. This choice provided us with a lot of freedom and possibility to see all configurations and possibly do custom configurations, which was valuable when aiming for a full-fledged exploration. A part of the setup process also involved implementing a CNI to the cluster responsible for network connectivity between the cluster components. In this project Calico was installed to handle this task. We could also have chosen Flannel, but with the seemingly small differences the choice was irrelevant. Considering the complexity of this approach, it was pretty straight forward. The cluster setup involving kubeadm only consisted of a few commands and did

not require any special competence. Installing the CNI was also a smooth affair. On the other hand, one aspect of the manual cluster setup approach one should invest some extra evaluation into is the choice of VM vendor. Setting up a Kubernetes cluster manually will involve some manual configuration of the specification of the machines that can be done through an interactive portal, such as the Azure portal. Being familiar with the VM vendor of your choice is very valuable.

Deployment versus StatefulSet and Volume Assessments

Regarding the choice between a Deployment versus a StatefulSet, the final setup landed on the implementation of a StatefulSet. Considering that the application deployed on the cluster was a stateful application relying on a replicated architecture involving multiple pods on multiple nodes, the only viable choice was a StatefulSet. The only possible scenario where we could have used a Deployment for our stateful application is one where we only were suppose to deploy one pod. Complexity-wise this configuration was a bit more tricky than setting up the cluster. First of all it took some time to understand that using a Deployment with three replicas was impossible due to misleading error messages. After the Deployment option was discarded, difficulties with assigning volumes to the three pods correctly in the StatefulSet arised. Understanding how volumes work is a complex task and should be investigated thoroughly before implementing any stateful workloads on your cluster. One must evaluate what kind of volume; local, NFS, AzureDisk e.g., that suits your situation best, if you should utilize a storageclass or not, and if so, what provisioner should be used for that storageclass. In addition one should study how PVCs work and how they should be implemented when using a StatefulSet.

Communication Between Containers on Different Nodes

Regarding establishing communication between the containers on different nodes, the final setup relies on a MongoDB replicaset initialized by a sidecar container. In theory the internal pod-communication should have been established by Calico and realized by the pod-networking-cidr applied to the cluster when initializing it with kubeadm. Incompatibilities with the subnet configurations on Azure and on the cluster did however not reflect the same state in practice. The default configuration from Calico only allowing traffic on the IPIP-interface, while Azure blocks all traffic on the IPIP-interface was the root of the internal networking problem. This was fixed by manually editing the Calico ippool to allow traffic on the VXLAN-interface which consequently allowed the MongoDB sidecar to complete its automatic configuration of the replicaset. Before jumping into the Kubernetes world without being accompanied with a provider or an application operator, you should definitely have your networking skills and understanding at a high level. Puzzling together all the components of Kubernetes, which is a highly layered construction, with the correct IP-addresses, ports and subnets is difficult, especially since a lot of the communication lives in the shadow of Kubernetes behind NAT.

External Connection

Regarding acquiring external connection to the cluster, the final setup turned out not having any service exposing it to the outside world. The reason for this outcome was a combination of two things. First, it demanded unnecessary usage of time and money seen in context of what was necessary to do in order to answer the research questions. Second, considering the environment where the database is the only running instance in the cluster it would not make sense exposing it directly to the public without having any application in front.

General Considerations

Regarding the general considerations made in the context of this particular project, we stood up against the questions whether a master node can run workloads or not, to implement MongoDB as a replicated cluster or a sharded cluster and how to correctly connect to containers. In the final setup we have workloads running on the master, which is fine considering that the cluster is only an experimental cluster that is not suppose to run any production-workloads. MongoDB was implemented as a replicated cluster in contrast to a sharded cluster. This choice was based on simplicity grounds and with no regards to expected amount of reads and writes since the application will not be operating anyhow. When facing the question on how to correctly connect to the containers, it soon became clear that the two methods, execing into the pod or into the container, was equal. All these considerations did not have any big impact on the complexity on the setup. Hence, this is not something that should affect a decision between choosing a self-managed cluster or provider-managed cluster, or choosing an application operator or not.

As an attempt of summarizing all decisions made throughout this exploratory process and what architectural impacts they had on the final setup, see Figure 4.1.

4.2 Answers to the Research Questions

In order to develop a proper conclusion to this master thesis, we intend to answer the research questions formulated in Chapter 1, Introduction.

1. When choosing a self-managed cluster in contrast to a provider-managed cluster, some overhead is added. First, you manually need to obtain VMs or some other machines to run your cluster on. Second, you need to download all necessary tools on all the machines before initializing the cluster and add nodes to it. The steps needed to set up a provider-managed cluster on the other hand is usually a one-liner command.
2. Despite some added overhead, the process is quick and easy, and given the increased freedom and control you get, it is definitely worth considering choosing a self-managed cluster in contrast to a provider-managed cluster.

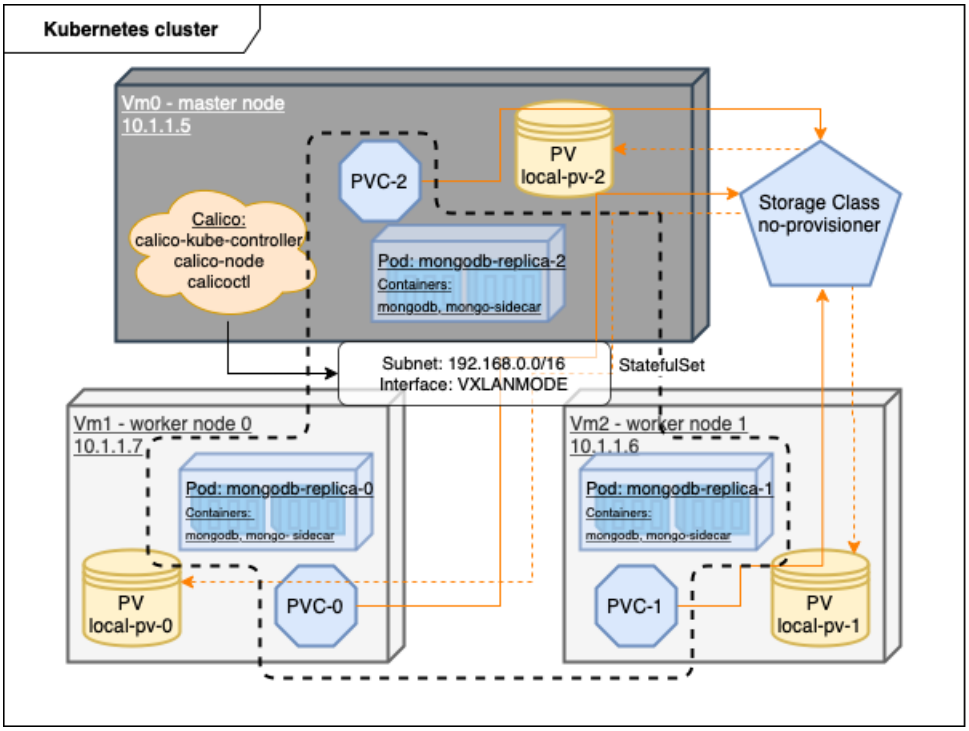


Figure 4.1: Final Kubernetes cluster architecture.

3. When avoiding an operator in the deployment of MongoDB in a Kubernetes cluster, a lot of overhead is added to the setup process. First, you need to create your own YAML-files and figure out what architecture that suits your environment the best. This involves gathering enough domain knowledge to know what fields must be added to the pod specifications of the StatefulSet, the PVC template specifications, the potential StorageClass specifications, the PV specifications and the service specifications. You also need to figure out if you need Secrets, ConfigMaps and RBAC. Second, you may need to write your own dockerfiles to create custom images. Third, you must understand how networking in Kubernetes works. Finally, you either need to utilize sidecars for automating some necessary database functionality or do it manually which may be very troublesome. The steps needed when exploiting an operator are not examined in detail, but assumingly they are easier and provide better functionality and conditions for scaling.
4. Taken into account all the overhead added, the freedom and control you gain when avoiding an operator is *not* worth the extra overhead and reduced functionality. Hence, it is not worth considering avoiding exploiting an application operator for running MongoDB in a Kubernetes cluster.

To summarize, Kubernetes is a highly coupled service of components and networking layers that can introduce a lot of complex errors that originates from config-

urations that are well hidden from the normal user in the many abstractions of Kubernetes. On the other hand, it is a very powerful tool that can ease your operating team for a lot of overhead in terms of maintaining and scaling an application in production. When that being said, it only implies if used correctly and monitored by experienced developers. When being faced with the choice between a self-managed Kubernetes cluster or a provider-managed Kubernetes cluster and using an operator or not, the decision is highly dependent on the prior statement; whether or not your operation team are experienced Kubernetes users. In addition, the decision is also dependent on how much freedom and personal configuration you need, and of course how much fundings you possess. However, put aside the organizational reasons for basing your decision and only focusing on the experiences made throughout this project, the conclusion is the following;

It is worth saving money and gain freedom by choosing a self-managed cluster and rather spend your money on an operator [1] *to easily and correctly set up a functioning and secure application, even though this limits your freedom in customizing the application configuration.*

[1] Many operators are even open source.

Chapter 5

Future work

This chapter will present some areas of further development that are closely related to the essence of the master thesis. The directions one can consider is the following:

- Comparing manual and automated approach.
- Testing different operators' complexity.
- Comparing performance of different persistent volume kinds.
- Comparing the manual setup process between different databases.

Of course, there are plenty more directions to choose from, but these are the ones that will be discussed in more depth in the below sections.

5.1 Comparing Manual and Automated Approach

This direction within future work is more of an extension of this thesis. In this thesis the focus have been on the manual setup of a stateful application in a self-managed cluster. When comparing that process with an automated approach, there are different aspects of the manual versus automated approach one can study. First, one can only focus on establishing the cluster in a manual way – as done in the thesis – compared to the quick and easy setup of a subset of provider-offered solutions like AKS, EKS and GKE. In that process one can study parameters such as simplicity, time and cost. A further extension of this research could be to study the usage of the different clusters - degree of freedom possessed, degree of control possessed, and degree of complexity versus simplicity. Second, one can focus on deploying a stateful application in a manual way – as done in this thesis – compared to deploying one with an operator. The same parameters as mentioned above can also be studied in this context, as well as extending by studying the differences in usage of the stateful application when deploying manual versus with an operator.

5.2 Testing Different Operators' Complexity

Another possible direction to go in, involve exploring the complexity offered by the different available operators. A starting point here could be to investigate the variety of operators for a single stateful application. In the instance of MongoDB as the stateful application one could compare the advanced enterprise operator and the less advanced community operator. Another perspective in the same direction could be to investigate how different operators designed for different stateful applications differ. Parameters to look into if so would again be time, freedom, control, cost and simplicity, as well as the ability to handle both horizontal and vertical scaling, backups, monitoring and version updates.

5.3 Comparing Performance of Different Persistent Volume Kinds

Directing focus towards the volume-aspect of the Kubernetes environment, one can examine different solutions to structure the volume. Kubernetes supports plenty of alternative PV types including awsElasticBlockStore, azureDisk, gcePersistentDisk, glusterfs, hostPath, local and nfs amongst others. It is also possible to investigate the performance impacts of introducing a storageclass. If even further investigation is desired, one could look into how varying the provisioner of the storageclass effects the performance or simplicity of setup and management.

5.4 Comparing the Manual Setup Process Between Different Databases

Finally, a last possible direction to go in suggested in this thesis, involve comparing the manual setup in a Kubernetes cluster with a few different databases. An idea could be to investigate three fundamentally different database systems, meaning e.g. one relational, one NoSQL and one distributed relational. Since being fundamentally different it is a fair assumption to make that they require fundamentally different configurations when setting them up without an operator. Taking it a step further one could also compare the performance with a benchmarking like Yahoo! Cloud Serving Benchmark (YCSB) or SysBench. Different database systems have properties that either makes them well suited to run in a Kubernetes environment or that makes them inappropriate. As en example, the ability to easily scale horizontally is a property that speaks well for running in Kubernetes. Based on this it is fair to believe that both NoSQL databases and distributed databases may benefit more from running in a Kubernetes environment than a relational database would.

Bibliography

[1] Kubernetes.io, *Considerations for large clusters*, https://kubernetes.io/docs/setup/best-practices/cluster-large/, Jun. 2021.

[2] B. Basyildiz, *A brief history of container technology*, https://www.section.io/engineering-education/history-of-container-technology/, Aug. 2019.

[3] IBM, *Container orchestration*, https://www.ibm.com/cloud/learn/container-orchestration, May 2021.

[4] Docker, *Use containers to build, share and run your applications*, https://www.docker.com/resources/what-container.

[5] Docker, *Top 10 best container software in 2021*, https://www.softwaretestinghelp.com/container-software/, Nov. 2021.

[6] M. Palmer, *Service - kubernetes guide with examples*, https://matthewpalmer.net/kubernetes-app-developer/articles/service-kubernetes-example-tutorial.html.

[7] M. Palmer, *Kubernetes networking guide for beginners*, https://matthewpalmer.net/kubernetes-app-developer/articles/kubernetes-networking-guide-beginners.html.

[8] R. Fattakhov, *Learn the basics and benefits of container orchestration*, https://www.parallels.com/blogs/ras/container-orchestration/, Nov. 2020.

[9] Sysdig, *Container and kubernetes security checklist*, https://sysdig.com/resources/whitepapers/s-container-and-kubernetes-security-checklist/?utm_source=google&utm_medium=cpc&utm_campaign=10874493555&adgroupid=106662533163&utm_content=473228537164&utm_term=containerization%20security&utm_position=&utm_device=c&utm_type=p&utm_geo=1010976&gclid=Cj0KCQjwrJOMBhCZARIsAGEd4VFe3kqT-ls6FKwaJx_e1kmYMw_1I5L6WmkSB-gYZKZnYXZJqAU7zg4aAuYLEALw_wcB.

[10] M. Mafy, *Learn the basics and benefits of container orchestration*, https://www.paloaltonetworks.com/blog/2020/05/containers-are-inherently-secure-reality-or-myth/, May 2020.

[11] B. Enterprise, *How container workloads are changing the future of cybersecurity*, https://businessinsights.bitdefender.com/how-container-workloads-are-changing-the-future-of-cybersecurity, Aug. 2021.

[12] A. Logvinenko, *Why, when and how to use kubernetes for web app development*, https://mobidev.biz/blog/when-why-how-use-kubernetes-app-development, May 2021.

[13] C. Gutierrez, *Spotify runs 1,600+ production services on kubernetes*, https://www.altoros.com/blog/spotify-runs-1600-production-services-on-kubernetes/, Sep. 2021.

[14] D. Taylor, *How to create ec2 instance in aws: Step by step tutorial*, https://www.guru99.com/creating-amazon-ec2-instance.html, Feb. 2022.

[15] B. Kurkchiev, *3 reasons to bring stateful applications to kubernetes*, https://thenewstack.io/3-reasons-to-bring-stateful-applications-to-kubernetes/, Aug. 2020.

[16] D. Seymour, *What makes a database a good fit to run in kubernetes?* https://softwareengineeringdaily.com/2020/09/22/what-makes-a-database-a-good-fit-to-run-in-kubernetes/, Sep. 2020.

[17] Kubernetes.io, *Persistent volumes*, https://kubernetes.io/docs/concepts/storage/persistent-volumes/, Sep. 2021.

[18] K. Davenport, *A basic guide to kubernetes storage: Pvs, pvcs, statefulsets and more*, https://portworx.com/blog/basic-guide-kubernetes-storage/, Dec. 2017.

[19] Z. Wen, *Persistent volume claim for statefulset*, https://zhimin-wen.medium.com/persistent-volume-claim-for-statefulset-8050e396cc51, Sep. 2018.

[20] Kubernetes.io, *Statefulsets*, https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/, Sep. 2021.

[21] C. Klinsmann, *Managed kubernetes vs self-managed kubernetes*, https://www.x-cellent.com/blog/managed-kubernetes-vs-self-managed-kubernetes/, Mar. 2022.

[22] M. Logan, *The basics of keeping kubernetes clusters secure*, https://www.trendmicro.com/vinfo/us/security/news/virtualization-and-cloud/the-basics-of-keeping-your-kubernetes-cluster-secure-part-1, Aug. 2020.

[23] O. Larsen, *Running databases in a kubernetes cluster*, https://www.diva-portal.org/smash/get/diva2:1369598/FULLTEXT01.pdf, 2019.

[24] opensource.com, *What is ansible*, https://opensource.com/resources/what-ansible.

[25] *Host os swap space must be disabled in nginx controller 2.8.0 and later*, https://support.f5.com/csp/article/K82655201, Nov. 2019.

[26] *Container network interface (cni) providers*, https://rancher.com/docs/rancher/v2.5/en/faq/networking/cni-providers/.

[27] *Can't run multiple mongodb docker container with same shared volume*, https://stackoverflow.com/questions/52660659/cant-run-multiple-mongodb-docker-container-with-same-shared-volume, Oct. 2018.

[28] *Running mongodb on kubernetes with statefulsets*, https://kubernetes.io/blog/2017/01/running-mongodb-on-kubernetes-with-statefulsets/, Jan. 2017.

[29] projectcalico.docs.tigera.io, *Azure*, https://projectcalico.docs.tigera.io/reference/public-cloud/azure?fbclid=IwAR0K4SagYXNjW9AsFOVeyChyNLtmX41d4hxLc_1cfbeuW-1LRRV7L9gaHKM.

[30] mongodb.com, *Mongodb database architecture in kubernetes*, https://www.mongodb.com/docs/kubernetes-operator/master/tutorial/mdb-resources-arch/.

[31] mongodb.com, *Mongodb documentation*, https://www.mongodb.com/docs/manual/reference/program/mongos/.

Appendix A

Output from MongoDB

Code listing A.1: Output from rs.status() from mongodb instance on worker0 after editing Calico ippools.

```
andrea@masternode$ kubectl exec -it mongodb-replica-0 -- mongo
rs0:SECONDARY> rs.status()
{
        "set" : "rs0",
        "date" : ISODate("2022-05-25T13:17:00.690Z"),
        "myState" : 2,
        "term" : NumberLong(3),
        "syncingTo" : "192.168.196.136:27017",
        "syncSourceHost" : "192.168.196.136:27017",
        "syncSourceId" : 0,
        "heartbeatIntervalMillis" : NumberLong(2000),
        "majorityVoteCount" : 2,
        "writeMajorityCount" : 2,
        "optimes" : {
                "lastCommittedOpTime" : {
                        "ts" : Timestamp(1653484611, 1),
                        "t" : NumberLong(3)
                },
                "lastCommittedWallTime" : ISODate("2022-05-25T13:16:51.621Z"),
                "readConcernMajorityOpTime" : {
                        "ts" : Timestamp(1653484611, 1),
                        "t" : NumberLong(3)
                },
                "readConcernMajorityWallTime" : ISODate("2022-05-25T13:16:51.621Z"),
                "appliedOpTime" : {
                        "ts" : Timestamp(1653484611, 1),
                        "t" : NumberLong(3)
                },
                "durableOpTime" : {
                        "ts" : Timestamp(1653484611, 1),
                        "t" : NumberLong(3)
                },
                "lastAppliedWallTime" : ISODate("2022-05-25T13:16:51.621Z"),
                "lastDurableWallTime" : ISODate("2022-05-25T13:16:51.621Z")
        },
        "lastStableRecoveryTimestamp" : Timestamp(1653484561, 1),
        "lastStableCheckpointTimestamp" : Timestamp(1653484561, 1),
        "members" : [
                {
                        "_id" : 0,
                        "name" : "192.168.196.136:27017",
                        "health" : 1,
                        "state" : 1,
                        "stateStr" : "PRIMARY",
                        "uptime" : 1207,
                        "optime" : {
                                "ts" : Timestamp(1653484611, 1),
                                "t" : NumberLong(3)
                        },
                        "optimeDurable" : {
                                "ts" : Timestamp(1653484611, 1),
                                "t" : NumberLong(3)
                        },
```

```
                        "optimeDate" : ISODate("2022-05-25T13:16:51Z"),
                        "optimeDurableDate" : ISODate("2022-05-25T13:16:51Z"),
                        "lastHeartbeat" : ISODate("2022-05-25T13:16:59.383Z"),
                        "lastHeartbeatRecv" : ISODate("2022-05-25T13:17:00.560Z"),
                        "pingMs" : NumberLong(1),
                        "lastHeartbeatMessage" : "",
                        "syncingTo" : "",
                        "syncSourceHost" : "",
                        "syncSourceId" : -1,
                        "infoMessage" : "",
                        "electionTime" : Timestamp(1653481651, 1),
                        "electionDate" : ISODate("2022-05-25T12:27:31Z"),
                        "configVersion" : 27721
                },
                {
                        "_id" : 1,
                        "name" : "192.168.204.72:27017",
                        "health" : 1,
                        "state" : 2,
                        "stateStr" : "SECONDARY",
                        "uptime" : 436171,
                        "optime" : {
                                "ts" : Timestamp(1653484611, 1),
                                "t" : NumberLong(3)
                        },
                        "optimeDate" : ISODate("2022-05-25T13:16:51Z"),
                        "syncingTo" : "192.168.196.136:27017",
                        "syncSourceHost" : "192.168.196.136:27017",
                        "syncSourceId" : 0,
                        "infoMessage" : "",
                        "configVersion" : 27721,
                        "self" : true,
                        "lastHeartbeatMessage" : ""
                },
                {
                        "_id" : 2,
                        "name" : "192.168.235.138:27017",
                        "health" : 1,
                        "state" : 2,
                        "stateStr" : "SECONDARY",
                        "uptime" : 1207,
                        "optime" : {
                                "ts" : Timestamp(1653484611, 1),
                                "t" : NumberLong(3)
                        },
                        "optimeDurable" : {
                                "ts" : Timestamp(1653484611, 1),
                                "t" : NumberLong(3)
                        },
                        "optimeDate" : ISODate("2022-05-25T13:16:51Z"),
                        "optimeDurableDate" : ISODate("2022-05-25T13:16:51Z"),
                        "lastHeartbeat" : ISODate("2022-05-25T13:16:59.524Z"),
                        "lastHeartbeatRecv" : ISODate("2022-05-25T13:16:59.523Z"),
                        "pingMs" : NumberLong(1),
                        "lastHeartbeatMessage" : "",
                        "syncingTo" : "192.168.196.136:27017",
                        "syncSourceHost" : "192.168.196.136:27017",
                        "syncSourceId" : 0,
                        "infoMessage" : "",
                        "configVersion" : 27721
                }
        ],
        "ok" : 1,
        "$clusterTime" : {
                "clusterTime" : Timestamp(1653484611, 1),
                "signature" : {
                        "hash" : BinData(0,"AAAAAAAAAAAAAAAAAAAAAAAAAAA="),
                        "keyId" : NumberLong(0)
                }
        },
        "operationTime" : Timestamp(1653484611, 1)
}
```

Code listing A.2: Snippet of output from container logs.

```
andrea@masternode$
[...] COMMAND [conn14] command admin.$cmd command:
replSetReconfig {
replSetReconfig: {
    _id: "rs0",
    version: 3,
    protocolVersion: 1,
    writeConcernMajorityJournalDefault: true,
    members: [
            {
        _id: 0,
        host: "192.168.204.69:27017",
        arbiterOnly: false,
        buildIndexes: true,
        hidden: false,
        priority: 1,
        tags: {},
        slaveDelay: 0,
        votes: 1
            },
            {
        _id: 1,
        host: "192.168.235.136:27017"
            },
            {
        _id: 2,
        host: "192.168.196.133:27017"
            }
        ],
    settings: {
        chainingAllowed: true,
        heartbeatIntervalMillis: 2000,
        heartbeatTimeoutSecs: 10,
        electionTimeoutMillis: 10000,
        catchUpTimeoutMillis: -1,
        catchUpTakeoverDelayMillis: 30000,
        getLastErrorModes: {},
        getLastErrorDefaults: {
            w: 1,
            wtimeout: 0
            },
        replicaSetId: ObjectId('628659272887d6d4980b7772')
        }
    },
force: false,
$db: "admin"
}
numYields: 0
ok: 0
errMsg:"Quorum check failed because not enough voting nodes responded; required 2 but only the following 1 
      voting nodes responded: 192.168.204.69: 27017; the following nodes did not respond affirmatively: 
      192.168.196.133:27017 failed with Couldn't get a connection within the time limit, 192.168.235.136: 
      27017 failed with Couldn't get a connection within the time limit"
errName:NodeNotFound errCode:74 reslen:588 locks: {} protocol:op_query 70074ms [...]
```

Appendix B

Configuration Files on Master

Code listing B.1: Custom dockerfile for MongoDB image with networking tools installed.

```
#
# MongoDB Dockerfile
#
# https://github.com/dockerfile/mongodb
#

# Pull base image.
FROM mongo:4.2

# Install MongoDB.
RUN \
  apt-get update && \
  apt-get install -y iproute2 && \
  apt-get install -y net-tools && \
  apt-get install -y dnsutils

# Define mountable directories.
# VOLUME ["/data/db"]
# Skip because its done from sts yaml

# Define working directory.
# WORKDIR /data
# Skip because its done from sts yaml

# Define default command.
CMD ["mongod"]

# Expose ports.
#   - 27017: process
#   - 28017: http
EXPOSE 27017
EXPOSE 28017
```

Code listing B.2: Headless service for MongoDB replicaset.

```
apiVersion: v1
kind: Service
metadata:
  name: mongo
  labels:
    app: mongo
spec:
  ports:
    - port: 27017
      targetPort: 27017
  clusterIP: None
  selector:
    app: mongo
```

Code listing B.3: Secret for MongoDB credentials.

```
apiVersion: v1
data:
  password: cGFzc3dvcmQxMjM=
  username: YWRtaW51c2Vy
kind: Secret
metadata:
  creationTimestamp: null
  name: mongo-creds
```

Code listing B.4: Statefulset for three replicas of MongoDB.

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: mongodb-replica
spec:
  serviceName: mongo
  replicas: 3
  selector:
    matchLabels:
      app: mongo
      role: mongodb
      environment: test
  template:
    metadata:
      labels:
        app: mongo
        selector: mongo
        role: mongodb
        environment: test
    spec:
      terminationGracePeriodSeconds: 30
      serviceAccount: mongo-account
      containers:
      - name: mongodb
        image: andreastanderen/mongo-sts-3
        command:
          - mongod
          - "--bind_ip_all"
          - "--replSet"
          - rs0
        ports:
        - name: mongo-port
          containerPort: 27017
        env:
        - name: MONGO_INITDB_ROOT_USERNAME
          valueFrom:
            secretKeyRef:
              name: mongo-creds
              key: username
        - name: MONGO_INITDB_ROOT_PASSWORD
          valueFrom:
            secretKeyRef:
              name: mongo-creds
              key: password
        volumeMounts:
        - name: mongo-data
          mountPath: /data/mongo
      - name: mongo-sidecar
        image: cvallance/mongo-k8s-sidecar
        env:
          - name: MONGO_SIDECAR_POD_LABELS
            value: "role=mongodb,environment=test"
  volumeClaimTemplates:
```

```
  - metadata:
      name: mongo-data
    spec:
      accessModes:
      - ReadWriteOnce
      storageClassName: manual
      resources:
        requests:
          storage: 1Gi
```

Code listing B.5: Role-based access control.

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: mongo-account

apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: mongo-role
rules:
- apiGroups: [""]
  resources: ["configmaps"]
  verbs: ["*"]
- apiGroups: [""]
  resources: ["deployments"]
  verbs: ["list", "watch"]
- apiGroups: [""]
  resources: ["services"]
  verbs: ["*"]
- apiGroups: [""]
  resources: ["statefulset"]
  verbs: ["*"]
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get","list", "watch"]

apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: mongo_role_binding
subjects:
- kind: ServiceAccount
  namespace: default
  name: mongo-account
roleRef:
  kind: ClusterRole
  name: mongo-role
  apiGroup: rbac.authorization.k8s.io
```

Code listing B.6: StorageClass.

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: manual
provisioner: kubernetes.io/no-provisioner
volumeBindingMode: WaitForFirstConsumer
```

Code listing B.7: Persistent Volume for worker 0.

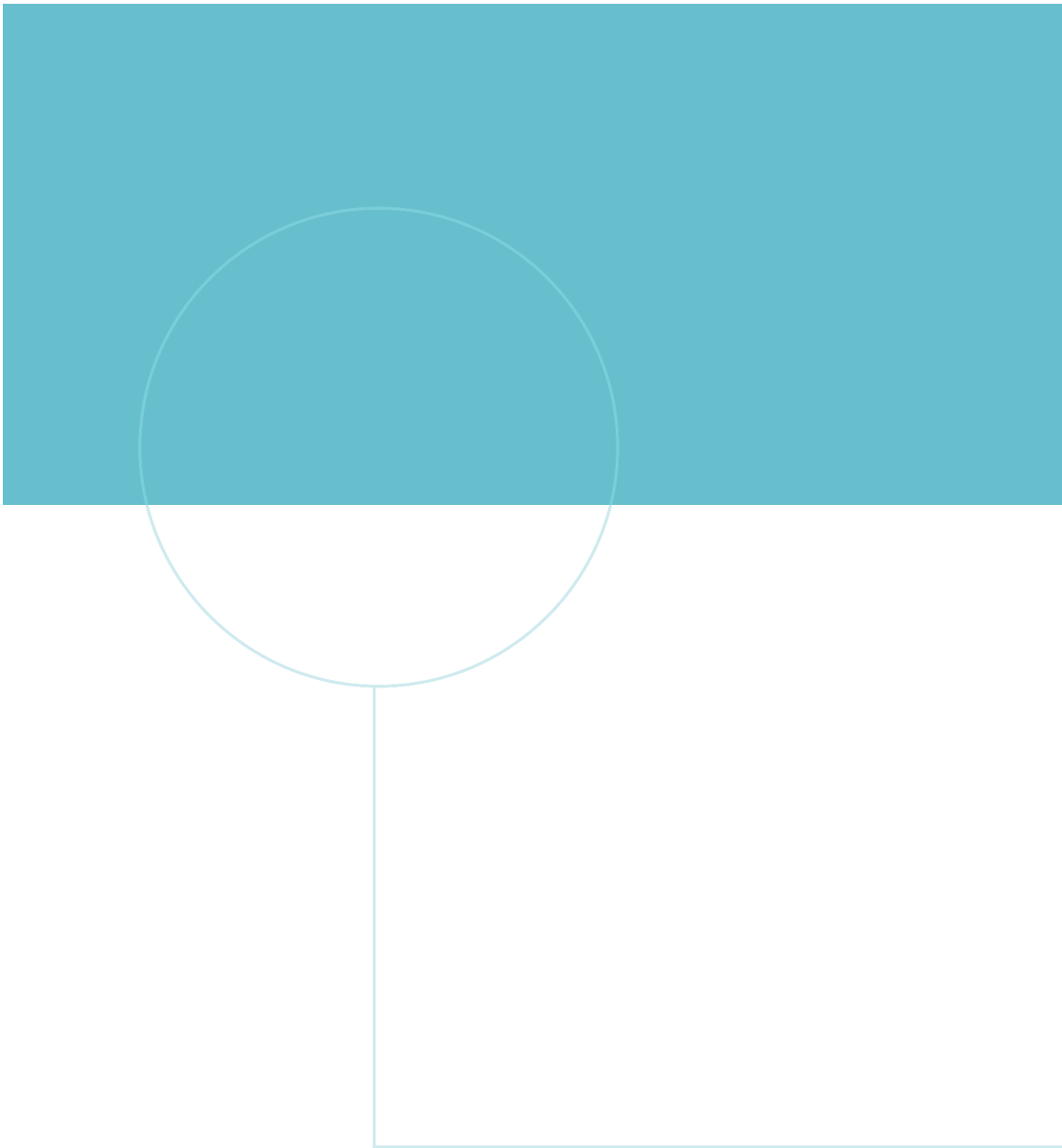
```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: local-pv-0
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  capacity:
    storage: 2Gi
  local:
    path: /data/mongo
  nodeAffinity:
    required:
      nodeSelectorTerms:
      - matchExpressions:
        - key: kubernetes.io/hostname
          operator: In
          values:
          - worker0
```

Code listing B.8: Persistent Volume for worker 1.

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: local-pv-1
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  capacity:
    storage: 2Gi
  local:
    path: /data/mongo
  nodeAffinity:
    required:
      nodeSelectorTerms:
      - matchExpressions:
        - key: kubernetes.io/hostname
          operator: In
          values:
          - worker1
```

Code listing B.9: Persistent Volume for master.

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: local-pv-2
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: manual
  capacity:
    storage: 2Gi
  local:
    path: /data/mongo
  nodeAffinity:
    required:
      nodeSelectorTerms:
      - matchExpressions:
        - key: kubernetes.io/hostname
          operator: In
          values:
          - master-node-0
```



NTNU

Kunnskap for en bedre verden