h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

# Darmstadt University of Applied Sciences

– Computer Science Department –

# Deployment, Monitoring and Evaluation of Microservices using Kubernetes and Open Source Technologies On-Premises

Bachelor Thesis

Bachelor of Science (B.Sc.)

vorgelegt am 01.03.2021 von

**Hamza Ben Zayed**

Matrikelnummer: 749657

| | | |
|---|---|---|
| Referent | : | Prof. Dr. Hans-Peter Wiedling |
| Korreferent | : | Prof. Dr. Steffen Lange |

# DECLARATION

Ich versichere hiermit, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt habe.

Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht.

Die Zeichnungen oder Abbildungen in dieser Arbeit sind von mir selbst erstellt worden oder mit einem entsprechenden Quellennachweis versehen.

Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

*Darmstadt, 01.03.2021*

Hamza Ben Zayed

ABSTRACT

Nowadays, containers and microservices architecture are the standards for building modern enterprise-grade software and this is due to the fact of the advantages that these technologies offer. For such applications, numerous microservices are in play and it's not an easy task to manage all of these microservices or containers manually, and here is the role of Kubernetes. With the rise of public cloud providers such as GCP etc, it's a matter of minutes till you have a fully managed and highly available Kubernetes cluster running on the cloud. However, not all enterprises can take advantage of these solutions, and this can be due to different reasons such as data compliance and privacy regulations or even the cost of the Kubernetes Platforms. Therefore this thesis investigates a POC for creating and setting up a highly available Kubernetes cluster on-premises using kubeadm, how we can use it to deploy a microservices application as well as an evaluation of the proposed solution using monitoring open source technologies on the cluster.

# ZUSAMMENFASSUNG

Heutzutage sind Container und Microservices-Architektur der Standard für den Aufbau moderner Enterprise-Grade-Software und das liegt an der Tatsache der Vorteile, die diese Technologien bieten. Bei solchen Anwendungen sind zahlreiche Microservices im Spiel, und es ist keine leichte Aufgabe, all diese Microservices oder Container manuell zu verwalten, und hier kommt Kubernetes ins Spiel. Mit dem Aufkommen von Public-Cloud-Anbietern wie GCP ect. ist es eine Frage von Minuten, bis Sie einen vollständig verwalteten und hochverfügbaren Kubernetes-Cluster in der Cloud laufen haben. Allerdings können nicht alle Unternehmen die Vorteile dieser Lösungen nutzen, was verschiedene Gründe haben kann, wie z. B. Daten-Compliance und Datenschutzbestimmungen oder auch die Kosten für die Kubernetes-Plattformen. Daher untersucht diese Arbeit einen POC für die Erstellung und Einrichtung eines hochverfügbaren Kubernetes-Clusters vor Ort, wie wir damit eine Microservices-Webanwendung bereitstellen können sowie eine Evaluierung der vorgeschlagenen Lösung unter Verwendung von Open-Source-Technologien zur Überwachung des Clusters.

# CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# 1 INTRODUCTION

Software engineering has evolved significantly and especially the SDLC (Software Development Life Cycle). Software engineers have been looking for alternatives to the classic methods such as the waterfall model and move towards more flexible and modern ways based on iterative work and especially more change tolerant, the result was the agile methodology with Ops into account where the software is engineered iteratively. However, it wasn't possible to take full advantage of these methodologies in the real world where software was built in a monolithic way. Therefore new approaches and software architectures have appeared such as SOA (Service Oriented Architecture) and MSA (Microservices Architecture) afterward, Where the software is divided into small and independent programs where each of these programs is responsible for a specific business domain of the application.

## 1.1 MOTIVATION

Meanwhile, the software delivery field has been evolving slowly. Hardware virtualization or VMs(virtual machines) were still the most used technique to deploy and ship software to production, no matter the size or the type of the application, even for microservices. This method has a few cons like resource usage and scalability. This was the case until a firm named docker[5] has come with the idea of packaging software in a lightweight fully isolated environment with a whole set of technologies that they have built around it, namely the container engine and the docker client what has made it a lot simple to use adapt.

With the rise of containerization, firms have started breaking their monolithic software and adapting their technology stacks to follow the MSA and the new software engineering standards. they end up running tens and even hundreds of containers and that was messy, insecure and not resilient. it also has made it a lot harder for administrators and engineers to manage all of the workloads, as a result, several container orchestration solutions have been built in order to make deploying and managing containerized workloads a lot more convenient and effective. With Kubernetes[23] winning the container orchestrators war all of the public cloud providers have built several products for it like Kubernetes Management engines what has made working with it easier, to the degree that it takes only a few steps to spin up a production-ready Kubernetes cluster. There are now even public cloud providers that have specialized in Kubernetes solutions and they are called KaaS(Kubernetes as a Service) platforms, rancher[9] is an example. Also the open-source community has been growing so fast and resulting in

very interesting and innovative tools and solutions. However, the problem is that a lot of firms cannot use and take advantage of the public cloud providers and there is several reasons for that such as data privacy regulations, multi-tenancy and business conflicts[6], cost optimization or even because the managers have invested in their own private cloud and want to make use of their investment. So how can these firms take advantage of the use of Kubernetes on-premises as a self-built production-grade cluster and the whole open source technologies around it. In addition, how efficient it would be for deploying microservices applications.

## 1.2 METHODOLOGY

This thesis uses an hybrid research strategy that consists of both quantitative as well as qualitative[4] strategies. A qualitative strategy is where a project is implemented in an exploratory way, this strategy is used for creating the self-hosted high available Kubernetes cluster. In addition, as per definition a quantitative strategy involves conducting experiments and measuring data and this is used for evaluation the cluster using several types of testing and monitoring technologies and then analyzing the data in an interpretative way. For the data collection open source tools are used, such as Grafana[8], Prometheus[21] and Linkerd[25]. It is especially done after conducting the following three experiments, security test, load test and chaos test. For the load and chaos experiments, open source monitoring tools, this is illustrated in the following figure:

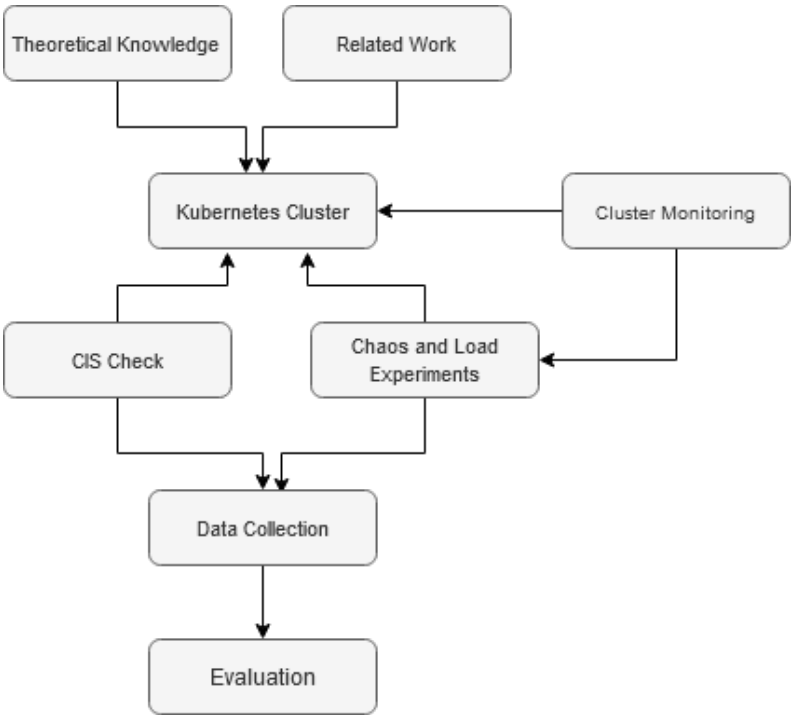


Figure 1.1: Methodology

## 1.3 THESIS PURPOSES

The purpose of this work is to investigate the creation of a production-grade Kubernetes cluster on-premises with providing an outline of the process of the cluster creation, where I go over all of the needed aspects such as infrastructure guidelines, security, networking and authorization policies. afterward, I will be deploying a microservices application on the cluster as an experiment along with setting up monitoring techniques using open source technologies.

Then the proposed solution will be evaluated using the collected data. For the evaluation of the proposed cluster implementation three criteria are take into account, these are cluster security, network latency and pods autoscalability.

- **Security**

Security should the first thing when building a production-grade infrastructure. In addition, Kubernetes is not secure by default. Therefore there are different ways to sneak into clusters when using the default configurations and vulnerabilities exposures are regularly found. It is necessary to check if security practices are implemented in order to protect the services and infrastructure.

- **Network latency**

Network latency is the delay of how much time it takes for data to be sent between two endpoints. It is a key criteria for every enterprise solution and it has very real business consequences for web services because of its direct influence on the user experience, as long latency leads users to simply abandon shopping carts, websites etc and look for the same service elsewhere.

- **Autoscaling**

Cluster elasticity is the scaling of the number of pods of a replicaset running on the cluster. It is a very important feature to have, when considering microservices and Kubernetes to deploy enterprise workloads. It allows cluster admins to administrate the cluster in more efficient way, as well as better performance and user experience.

The above mentioned objectives are approached as follows:
The second chapter is an introduction to Kubernetes architecture, fundamentals as well as the theoretical knowledge about Kubernetes networking and security aspects. In the third chapter, I provision the needed hardware infrastructure, create the Kubernetes cluster and discuss the deployment process in detail. In addition, I'll deploy a microservices application on the cluster. The fourth chapter is where I set up monitoring infrastructure and I evalu-

ate then the proposed cluster with conducting three experiments, security network latency and autoscaling experiment. During the experiments data will be collected using the setup monitoring tools and prepare it for the data analysis. After the data is collected, I will be evaluating the cluster and the application according to the collected data and the defined evaluation criteria in the introduction. The fifth chapter is then dedicated to the discussion. afterward, a conclusion is drawn and in the seventh chapter possible future works are discussed.

## 1.4 RELATED WORK

In this section, several scientific works about related topics such as microservices, containers and Kubernetes are presented and discussed.

**Networking in Containers and Container Clusters**
This work [18] provides insights into container networking strategies and configurations available in Docker. It also explains how they work and what limitations they have. The study also shows the importance of the networking layer of the containers when using Kubernetes. It goes further explaining networking goals for these clusters and a few of the possible networking strategies. This has contributed to this work in better understanding the networking on clusters when orchestrating workloads and what considerations should be taken as well as the choice of the networking plugins used.

**Containerisation and the PaaS Cloud**
This article [22] explores the underlying virtualization principles behind containers and explaining the inconsistencies with the virtual machines For deploying portable, interoperable applications in the cloud. It discusses also deployments on multi-cloud environments as well as how the distribution of the application can be done via lightweight software virtualization using containers to achieve better scalability and performance.

**A Systematization of Knowledge Related to Kubernetes Security Practices**
In this research [17] The authors provide a systematization of Kubernetes security practices that help in securing Kubernetes installations. Qualitative analysis is done on 104 Internet artifacts. As a result, several security best practices are identified and discussed, these include Authorization using roles for access control and implementing pod and network specific security policies. The research has contributed to this work with helping in understanding the security flaws that currently exists regarding Kubernetes and how to use authorization techniques to avoid these and have a more secure cluster.

**How to break a Monolith into Microservice**
In this work, [2] the author explains different patterns for refactoring applications and redesigning them to follow the microservices coupling. It also

addresses how containers have a principal role in migrating legacy systems and monolith into microservices and cloud-driven. Furthermore, the key role of Ops is described, when considering the MSA and a cloud-driven environment.

**Deploying a dockerized application with Kubernetes on GCP**
This is a study proposed by Botez et al. [1] that investigates the implementation of an Internet of Things application that is based on the MSA. The services are containerized using docker the standalone executables are then deployed on a Kubernetes cluster hosted on GCP (Google Cloud Platform). Also, the persistence layer of the web application is a cloud-based as it uses cloud SQL (Structured Query Language) as a DB, cloudSQL is GCPs product for SQL DB. The cluster is then monitored using Stackdriver, also a GCPs product for monitoring Kubernetes clusters and this enables useful features such as uptime Checks, Alerting, Tracing, Logging, and others.

In conclusion, the literature review shows the use of containerisation and orchestration techniques in modern software engineering has shown significant result in building production-grade software and infrastructure, however most of the conducted and available studies has been done following a cloud first approach. Therefore this study will focus essentially on building an orchestration infrastructure for deploying microservices applications with a cloud-agnostic approach, this means without using any public cloud solutions as Botez et al. [1] have done in their work.

# 2 KUBERNETES

To be able to design, manage and monitor Kubernetes clusters, it is quite important to have a good idea about how Kubernetes does work behind the scenes and not only about the fundamentals. Therefore this chapter explains both, the fundamentals of Kubernetes as well as under the hood aspects and workflow.

## 2.1 DEFINITION

Kubernetes is an open-source platform for automating deployment, scaling, and operations of containerized applications across clusters of hosts, providing container-centric infrastructure[11].

Kubernetes Principles:

| Principle | Description |
|---|---|
| Declarative over imperative | API driven application management<br>Agents monitor endpoints for state changes (real-time)<br>Controllers enforce desired state, self-healing |
| No hidden APIs | There no internal functionalities |
| Immutable | resources specifications and volumes cannot be modified |
| Workload portability | Decouple application logic from cluster implementation |

Table 2.1: Kubernetes Principles

Kubernetes Features:

- Scheduling: automated scheduling of workloads across the nodes

- Self-Healing: automated workload recovery and control

- Service Level Routing  Load Balancing

- Horizontal Scaling: automated scaling of workload

- Security: connections are secured using TLS (Transport Layer Security)

## 2.2 KUBERNETES ARCHITECTURE

Kubernetes is deployed in form of a cluster and it has Client-Server architecture. In Kubernetes, a Pod is the basic execution and work unit. The Pod is the smallest unit in the Kubernetes object model[11] that can be created or deployed. It represents processes running in the cluster. It encapsulates a container or multiple tightly coupled containers that share resources including network, storage and the node they are running on.

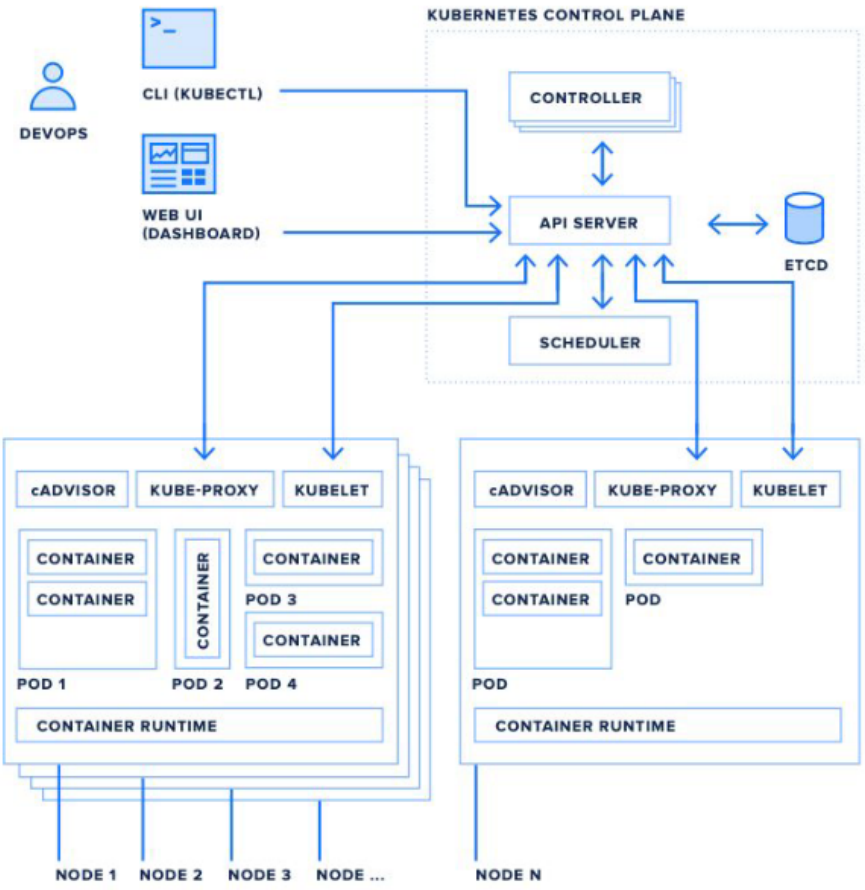


Figure 2.1: Kubernetes Architecture [19]

### 2.2.1 *Server: Control Plane*

The control plane is responsible for cluster management. This includes accepting clients requests (describing the desired state), scheduling containers and running control loops to drive the actual cluster state towards the desired one.
The following are the main elements of the control plane:

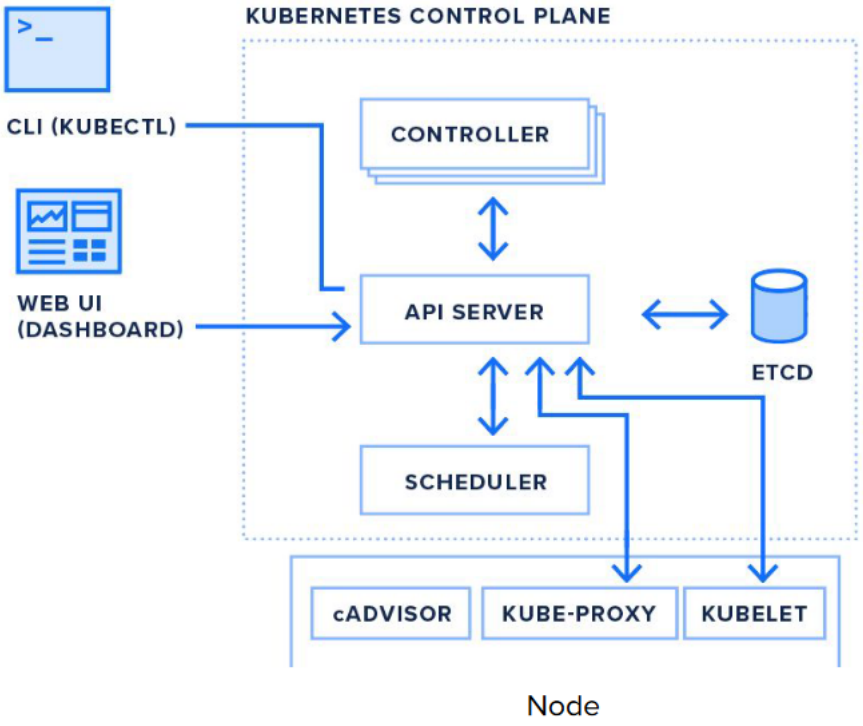


Figure 2.2: Kubernetes Control Plane [19]

#### 2.2.1.1 *API-Server*

It is a server that exposes REST API (Application Programming Interface) supporting basic CRUD operations on Kubernetes API objects[10] (such as pods, deployments, and services). This is the endpoint that a cluster administrator communicates with, Therefore it is considered as the frontend of the cluster. A cluster must have a minimum of one api-server available in order to function and it can be replicated over several nodes. The API-server is stateless[10]. Instead, it uses a distributed key-value storage system (Etcd) as its backend for storing the cluster state. To talk with the cluster (apiserver), we use the kubectl tool that generates for us the GRPC (Google Remote Procedure Call) calls.

#### 2.2.1.2 *Etcd*

It is a consistent, distributed and highly-available key-value store used as Kubernetes backing store for all cluster data[3].

#### 2.2.1.3 *Controller Manager*

The Kubernetes controller manager is a daemon that embeds the core control loops shipped with Kubernetes[11]. It runs the several control processes that watch attempt to move the actual state towards the desired state. Internally it consists of different Controllers such as:

- Deployment Controller: that ensures that the right number of replicas pods are running for each deployment.
- Endpoints Controller: that syncs and manages endpoints between services and Pods.
- namespace Controller: it manages and watches for namespaces.
- Node Controller: it mnges the worker nodes and watches for failures, etc.
- Service Account  Token Controller: create service and user account, creates and authorizes access tokens for the new namespaces.

#### 2.2.1.4 *Kube-DNS*

Each Kubernetes cluster has a process that is responsible for creating the DNS Pod and service. Afterward, it adds a configuration to the Kubelet (in each one of the worker nodes) to tell the containers to use the DNS Service's IP to resolve DNS names. Then other applications deployed in the cluster can look up the Services and Pods using their names.

#### 2.2.1.5 *Scheduler*

It takes care of Pod placement across the set of the available worker nodes, with the focus on balancing the resource consumption and not to place excessive loads on any of the nodes[13]. It runs as a process alongside the other master components such as the API server and is watching for Pods with an empty `PodSpec.NodeName`, and for each one, it posts a binding indicating where the Pod should be scheduled. At first, the scheduler does check for hardware and resource requirements and according to these requirements it filters the available nodes so that only nodes with sufficient resources are considered. Then each node is given a score using ranking functions, they are weighted by a positive number and the final score of each node is calculated by adding up all the weighted scores to find the best fit for the Pod. After the scores of all nodes are calculated, the node with the highest score is chosen as the host of the Pod. If there are more than one node with equal highest scores, a random one among them is chosen.

### 2.2.2 *Clients: Nodes*

The worker nodes are where the containerized applications and workloads, are running. A node is hardware agnostic, therefore it can be arbitrarily a physical machine (i.e. a server), or a virtual machine in a server. The following are the main elements of the control plane:

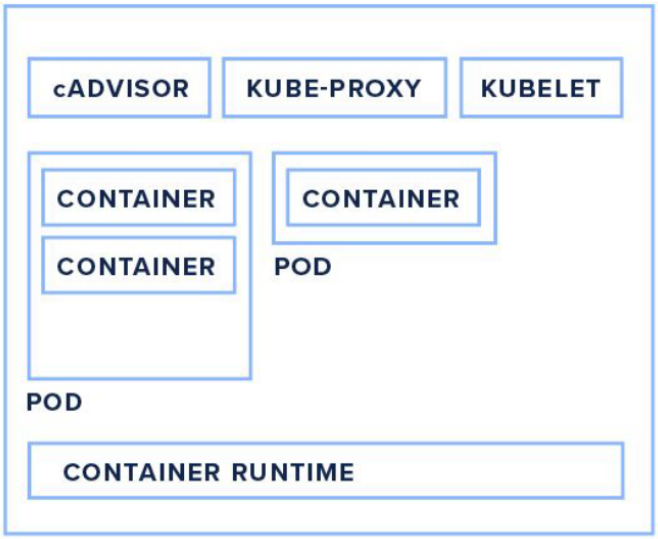


Figure 2.3: Kubernetes Nodes [19]

#### 2.2.2.1 *Kube-proxy*

It is a Pod deployed to each worker node, it is deployed under the kube-system namespace and is responsible for simple or round-robin forwarding of the traffic addressed to the virtual IP addresses of the Kubernete's service objects to the appropriate Pod. The kube-proxy has three modes as follows:

**- Userspace**
In this routing mode, a process that is running on the userspace is responsible for proxying and filtering the requests addresses to the node.

**- IP tables**
This mode uses Linux Netfilter rules to configure all routing for the Kubernetes services. This is the default mode used by the kube-proxy and most hosted platform. For load balancing this method uses an unweighted round-robin algorithm.

**- IPVS**
IP virtual server is a transport layer load router inside of Linux, It receives traffic on a virtual IP address and distributes it according to the available services. It has the same principle as the Kubernetes service object. However, it supports a lot more routing algorithms such as round-robin, least connections, IP hashing and a few others. It is more scalable than IP tables.

#### 2.2.2.2 *Kubelet*

The Kubelet is the key element that runs on each of the worker nodes. It talks to the API-server and manages the containers running on the node. It

uses the `PodSpec` JSON or YML object defined when creating the Pod object and keeps track of the desired state. The Kubelet uses the container runtime installed such as docker on the node to run the containers with the help of Container Runtime Interface (CRI).

## 2.3 NETWORKING

Kubernetes networking principles are [12] as follows:
'

- Pods can communicate with all other Pods without using NAT.

- all Nodes can communicate with all Pods without using NAT.

- The Pod private and public IP addresses are the the same.

**Kubernetes Networking Models**

In order to implement the mentioned networking principles Kubernetes defines four Networking models: [12]

- Highly-coupled container-to-container communications

- Pod-to-Pod communications

- Pod-to-Service communications

- External-to-internal communications

### 2.3.1 *Container to Container*

In Kubernetes containers within the same pod are designed to be tightly coupled, therefore they behave as if they are on the same host with regard to networking. They can reach each other using their port numbers on `localhost`. This offers simplicity, security(ports bound to `localhost` are visible within the pod but not outside) and performance. Besides, it simplifies migration for the applications, that are moving from the world of VM containerization.

### 2.3.2 *Pod to pod*

In Kubernetes, every created Pod is assigned an IP address (not machine-private). This is done with the help of the classless inter-domain routing, that is available on each node, it is a set of unique IP addresses. Therefore it is a NAT (Network Address Translation)-less, flat address space and each pod has its own IP address that other pods can know and simply communicate without proxies or translations. This done using Veth (Virtual Ethernet) pairs,

where the different processes are connected together going through the root network namespace. This enables naming and resolving strategies without the need for any configurations including self-registration techniques. However volumes like tmpfs (Temporary File System) are mostly used for this kind of task. This is different from the standard Docker model, because a container can not be reached on its private IP. Therefore we cant self-register anything from a docker container.

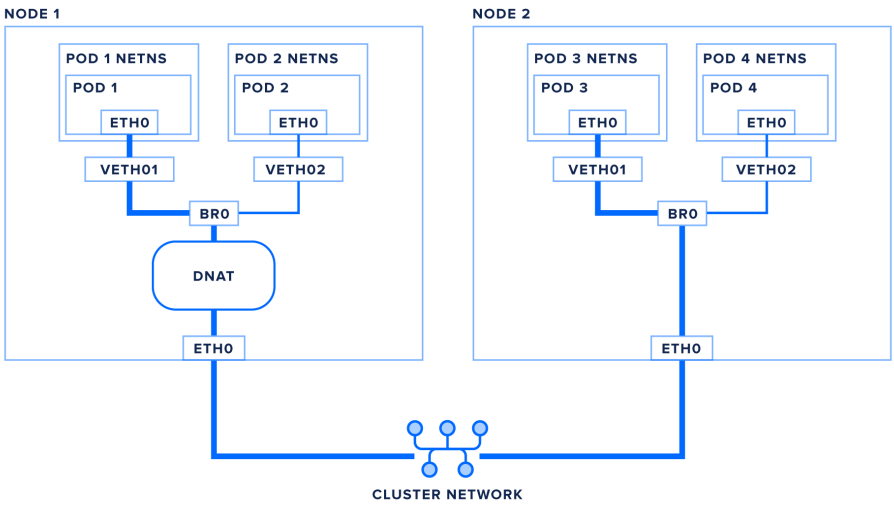


Figure 2.4: Pod to Pod Networking

### 2.3.3 *Pod to service*

In Kubernetes Pods are designed to changeable at any time this could be for scheduling purposes or because of failures. Therefore in order to enable high availability and ensure that communication with and between pods is maintained, Kubernetes uses special objects called services. The implementation of the service object creates a virtual IP for a group of pods which clients can access and which is transparently proxied to the right pods in a Service. This provides a highly-available load-balancing solution with low performance overhead by balancing client traffic from a node on that same node.

### 2.3.4 *External traffic to Pods*

Routing external traffic to the cluster in Kubernetes can be achieved using two methods, using the service object or the ingress resource.

To use the service Kubernetes object to expose the cluster to the outside world there are a few ways to do so. The first one is to use the standard cluster IP service type with a Kubernetes proxy that sits in front of the cluster. The second is `NodePort` service, it is Kubernetes's primitive way to get external traffic directly to a specific service. `NodePort`, as the name implies, opens a a specific port on all the Nodes, and the traffic sent to this port is forwarded. Another one is to use a service of type `LB` to expose the service to the internet, this is mostly done with the help of an external `LB` associated with a public cloud provider.

Ingress it is a resource and not a type of service contrary to services. An Ingress resource is essentially a collection of routing rules that can allow or deny users when trying to gain access to services running within a cluster. It sits in front of multiple services and acts as a "smart router" or entry point into the cluster. The behavior depends on the used ingress controller because it is the responsible object for applying those rules.

Because the preferred implementation depends on the environment, where the cluster is deployed and for more flexibility Kubernetes doesn't implement the specification discussed, it only exposes a CNI (Container Network Interface).

CNI implementations are called CNI plugins. They are responsible for allocating network interfaces for the containers and they must satisfy the CNI specification in how they receive and send data to Kubernetes. In Kubernetes containers are first created without a network interface, Afterward, Kubernetes calls the CNI Plugin. The Plugin configures the container networking and then return information about allocated network interfaces, IP addresses, etc.

## 2.4 KUBERNETES OBJECTS

This section is a brief introduction and description of the essential Kubernetes objects needed for a production-grade deployment of a stateful microservices application. Kubernetes Objects are persistent entities that represent the desired state of the cluster. They describe what and where containers are running, the resources that a container should have, and policies around how those containers behave such as restart policies, upgrades, and fault-tolerance. These objects are created typically using JSON or YML files and sent to the api-server using kubectl. In the architecture and networking section, I have already explained the Pod and service object.

### 2.4.1 *ReplicaSet*

It is a Kubernetes object that ensures that the desired number of Pod are replicas are available and running.

### 2.4.2 *Deployment*

Deployment It manages the deployment of the ReplicaSets using the deployment controller element of the control plane, which will capture changes and change the cluster state to the desired one. It's an improved version of the replication controller. Deployments can update, delete and rollback deployments in a declarative way.

### 2.4.3 *Stateful Set*

Stateful Set is the Deployment equivalent for deploying stateful applications. It provides guarantees about the identity of the Pods and the startup order. For the application, the Stateful Set manages one persistent volume per Pod. In addition, Stateful Sets uses headless services for networking, it is a service object with an undefined cluster IP.
Managing storage is a distinct problem from managing compute[14]. Therefore Kubernetes provides a way to decouple those two different tasks using PV and PVC.

### 2.4.4 *Persistent Volumes*

Kubernetes Persistent Volumes are cluster wide resources. they are storage objects with a life cycle independent of the pod that uses them[14]. They can use a wide variety of storage implementations such as DBs, NFS (Network File System) and others. PVs can be provisioned statically or dynamically, when the storage class claimed is not yet available in the cluster.

### 2.4.5 *Persistent Volume Claims*

The PVC is a request for storage by a user in the cluster. PVCs consumes PVs the same way as a Pod consumes Node resources[14]. When claiming a storage, we can define the access level and other properties.

### 2.4.6 *Secrets*

For production-grade applications, security is important, therefore sensitive data should not be saved in plain text. The solution is Kubernetes Secrets objects. Secrets enable us to use such information safely and reliably with the following properties:

- They are namespaced objects, they exist in the context of a namespace

- They can be accessed via volumes, environment variables, or containers

- The secret data on nodes is stored in tmpfs volumes

- A per-secret size limit of 1MB exists

### 2.4.7 *Configuration Object*

They are used to store insensitive data in key-value form, this can be environment-related data or any other. It provides a way to decouple the environment-specific data.

## 2.5 SECURITY

When thinking about security in Kubernetes there are a few aspects to think about such as container, pod, and cluster. To address these aspects Kubernetes provides several built-in resources that can be used to control the infrastructure and avoid security vulnerabilities.

**Security Context**
It is an object that can be defined when creating a pod using the security context field in the JSON or YML file. The settings specified in this field are applied to all the containers running in the pod. Using security context we can specify the privileges and resources that a container or a pod has the right to access or manipulate. The setting options include the following:

- Access and permission control, using PID (Process ID) and GID (Group ID).

- Privileges and unprivileged containers and pod

- Read-only mounting of the file system

Besides to the Security Context object Kubernetes also provides the pod Security Policy resource. It is a cluster-wide resource that controls security and sensitive aspects of the pod specification. It is an object that defines rules for the pod and acts as a guard.

Privileged: Unrestricted policy, providing highest level of permissions.

Default: Allows the default (minimally specified) Pod configuration.

Restricted: Heavily restricted policy

# 3 CLUSTER SETUP AND APPLICATION DEPLOYMENT

This chapter is divided into three sections as follows: cluster creation, application overview and application deployment.

## 3.1 CLUSTER CREATION

Setting up a production-ready Kubernetes cluster on-premises / private cloud is not an easy task and can be daunting. The most native way to spin up a Kubernetes cluster is to provision and configure the needed nodes and infrastructure, build and run the go binaries of the control plane elements as described in the second section, as foreground programs or using systemd. Then repeating the same process for the worker nodes elements. However, this is definitely not an ideal solution for production-grade use cases where resiliency, security and performance are necessary.

Therefore there are several tools that help in the process of creating self-hosted Kubernetes clusters for production use cases. For this work I have used kubeadm to install and configure the Kubernetes binaries. I have chosen Kubeadm because it is the only Kubernetes certified solution and is released by the Kubernetes team. Where it helps in bootstrapping the essential Kubernetes elements following security and performance best practices and with support to multiple features such as highly available clusters.

### 3.1.1 *Cluster Architecture*

While architecting the highly available Kubernetes cluster, I had to take several design decisions, these are explained as the follows:

**Cluster Topology**
For the cluster topology there are two options:

- Stacked Control plane nodes: In this topology, Etcd, the distributed key-value store of Kubernetes, is deployed on the same node as the control plane.
- External Etcd Cluster: This consists of deploying a separate Etcd cluster beside the Kubernetes cluster.

After researching the use cases of both topologies and studying several pieces of research done on this subject, this includes [15]. I have decided to choose the stacked control plane topology, because the Etcd node in this topology is deployed on the same node as the other control plane elements, therefore the Pod-to-Pod networking mode is used. This means there is no

need for any Virtual Ethernet Device or anything similar as the two pods are running on the same network interface. Also, an external Etcd cluster topology is generally meant for multi-cluster environment where there are several Kubernetes clusters in play and a standalone Etcd cluster could be used by all of the clusters to handle their states and this is not the case for this work.

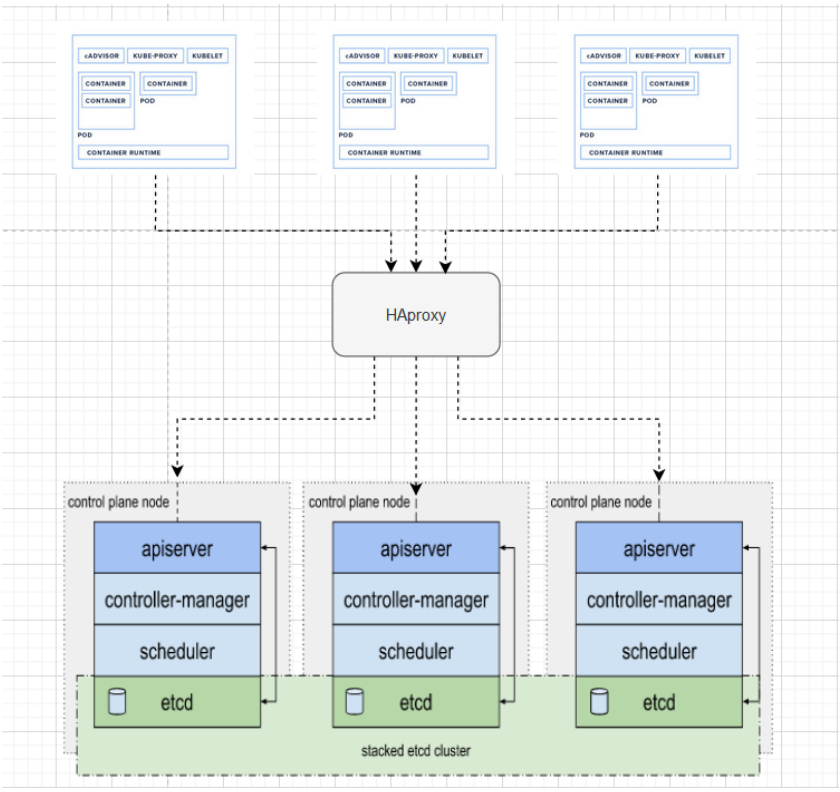


Figure 3.1: Cluster Architecture

**Control Plane Nodes**
It is important to provision the right number of nodes for the control plane, especially when the stack topology is opted because the control plane nodes also contain the Etcd instances and in order to decide how many control plane nodes should be provisioned to guarantee the high availability of the cluster, the workflow of Etcd should be taken into account. Etcd focuses on strong consistency, for that reason an Etcd cluster needs a majority of nodes, a quorum, to agree on updates to the cluster state. For a cluster with n members, a quorum is (n/2)+1, therefore the size of the Etcd cluster should be odd with the n is the number of the nodes that should be tolerated. As a result, I have decided to go with three nodes for the control plane.

**Worker Nodes**
The number of worker nodes that need to be provisioned depends on the workload, where we can easily add additional nodes or downsize the number of nodes in the cluster. For the worker nodes, I have also decided to go with three nodes.

**HAproxy**
Its a must for a highly available cluster to have several replicas of the proxy server, in order to avoid points of failure. Therefore I have provisioned two instances for the HAproxy servers.

**Provisioned Infrastructure**
This is an overview of the infrastructure I have used in this work:

Control Plane and Worker nodes:

- Number of VMs: each 3VMs

- dedicated CPUs: 2CPUs per VM

- dedicated RAM: 8GiB per VM

- OS: Linux Debian

HAproxy nodes:

- Number of VMs: 2VMs

- dedicated CPUs: 1CPU per VM

- dedicated RAM: 4Gib per VM

- OS:Linux Debian

Internet connectivity must be available for all of the nodes for pulling containers. Full network connectivity between machines in the cluster.

### 3.1.2 *Prerequisites and System Configuration*

Before actually setting up the cluster there are few configurations and modules needed.

**Swap**
Since the Kubernetes node agent doesn't support memory swap on the containers that it runs, it is very important to disable the memory swapp feature on the OS level. This can be done by modifying the `etc/fstab` file or by using the `swapoff` command. Notice that it is also possible to configure it not to fail on startup when the swap is activated, however this not a best practice.

```
sudo sed -i '/ swap / s/^\(.*\)$/#\1/g' /etc/fstab
sudo swapoff -a
```

Figure 3.2: Swap

**SELinux**
Security-Enhanced Linux, SELinux, is a Linux module that provides security policies for more security and access control. However new container runtimes don't fully support SElinux yet. Therefore, to avoid problems with the kubelet struggling to run the containers, I have put the SELinux module in a permissive mode. This is done, by using the `setenforce` command with the value 0 as an argument or by editing the SELinux config file.

```
sudo setenforce 0
sudo sed -i 's/^SELINUX=.*/SELINUX=permissive/g' /etc/selinux/config
```

Figure 3.3: SELinux

**Installations**
In this step, I have installed kubelet, kubeadm, and kubectl. One important thing is to hold the installed modules to prevent upgrades.

Afterward, I have installed and set up the container runtime that will be used on the nodes, for that I have decided to go with containerd since the Kubernetes team has announced that docker will be deprecated as a container runtime. Since docker configures the net.bridge and the IP forwarding under the hood, when using containerd I had to do that by myself in the `sysctl.conf` as well as in the `/proc/sys/net/ipv4/ip_forward`. Another important thing when configuring the container runtime is to set up the right cgroup, It has to be the same cgroup used by the node agent, in order for the kubelet to be able to create the containers. containerd and kubelet both use cgroupfs as a control group, therefore no additional configuration is needed. Afterward, I have enabled and restarted the `systemd` services for both the kubelet and containerd.

**Filesystem and Network Configuration**
There are two modules that should be loaded, the `overlay` and `br_netfilter` modules. The first is used for the `overlayfs` and the second for using the netfilter with the Linux host bridge network.

After the modules are loaded we should be configuring the systemctl to allow the packets traversing the bridge without being sent to iptables for processing. Afterward, I have added the above-loaded modules to the container runtime configuration. The shell script of this configuration is as follows:

### 3.1.3 *PKI and Certificates*

In Kubernetes Transport Layer Security (TLS) certificates are used to ensure the identity and the security of the c between all the elements whether within the control plane nodes or with the worker nodes. Therefore a Public Key Infrastructure (PKI), is needed for Kubernetes.

Actually, kubeadm is able to generate the necessary certificates and keys for us during the kubeadm init workflow. However, in this work I generate my own certificates to keep the private keys more secure by not storing them on the API server. Every Kubernetes cluster has a cluster root Certificate Authority (CA), it is used by the cluster elements to validate the API server's certificate, by the API server to validate kubelet client certificates, etc.
The needed certificates are the following:

- Client certificates for the kubelet to authenticate to the API server

- Server certificate for the API server endpoint

- Client certificates for administrators to authenticate to the API server

- Client certificates for the API server to talk to the kubelet

- Client certificate for the API server to talk to Etcd

- Client certificate for the controller manager to talk to the API server

- Client certificate/kubeconfig for the scheduler to talk to the API server.

There are a few methods to provision a Certificate Authority and generate the needed TLS certificates and keys. In this work I have used the single root CA method. In this methods the I have created a single root CA controlled by me. Then using this root CA I can then create multiple intermediate CAs, and delegate all further creation to Kubernetes itself, these are:

- `ca.crt,key` in kubernetes-ca: Kubernetes general CA

- `etcd/ca.crt,key` etcd-ca: For all Etcd-related functions

- `front-proxy-ca.crt,key` in kubernetes-proxy-ca: For the front-end proxy

- public/private key pair for signing service account tokens `sa.key/ sa.pub`

To create the root self-signed CAs, I have used OpenSSL and done the following steps: first I have created the CA configuration file of the certificate, then created the CA certificate signing request, CSR and afterward generated the CA certificate and private key: `ca-key.pem ca.pem`. The same steps are repeated for the etcd and front-proxy root CAs.

**The Kubelet Client Certificates** Since I have delegated the creation of the intermediate certificate to Kubernetes I do not have to create it myself, however it is a must to know that Kubernetes uses a special-purpose authorization mode called Node Authorizer, which specifically authorizes API requests made by the Kubelet. In order to be authorized by the Node Authorizer, the kubelet must use the credentials that identifies them as being in the system:nodes group, with `system:node:<node-name>` where the node name must be carefully set for each node in order to get authorized by the server. The full list of certificates and keys is the following figure:

```
root@m-node-1:/etc/kubernetes/pki# ls
apiserver-etcd-client.crt     apiserver-kubelet-client.key  ca.crt  front-proxy-ca.crt      front-proxy-client.key
apiserver-etcd-client.key     apiserver.crt                 ca.key  front-proxy-ca.key      sa.key
apiserver-kubelet-client.crt  apiserver.key                 etcd    front-proxy-client.crt  sa.pub
```

Figure 3.4: Cluster PKI

### 3.1.4 *Cluster Initialization*

The cluster initialization is done using kubeadm init command, where I have overriden the following arguments:

`-apiserver-bind-port` Port to bind the server to
`-control-plane-endpoint` stable IP address or DNS name
`-pod-network-cidr` Specify the range of IP addresses for the pod network
`-upload-certs` Upload control-plane certificates to the kubeadm-certs
When executing this command kubeadm does a series of checks where it validates the system after doing any changes. When the checks are valid, it begins with generating the self-signed CA as well as the intermediate certificates and keys, in this work this first step is skipped because I have provisioned the root CAs and provided them at texttt/etc/kubernetes/pki, so kubeadm can use these to generate the intermediate ones.

Afterward, the configuration files are generated for all the control plane elements(kube-server, Etcd, controllers and scheduler) as well as an administration configuration file. Then the kubelet service is started or restarted if it is already running and it begins with creating the static pods definition files of the control plane elements and starting these pods. After the kubelet mkes sure that all the static pods are running including the local Etcd, it generates bootstrap tokens used to join the cluster.

After the first control plane node is successfully initialized the bootstrap token is printed to the stdout along with the CA hash to verify the CA identity. I have saved these data into a shared file and then moved to repeat the process for the other two control plane nodes. After the nodes are configured

and have the necessary modules I have applied the kubeadm join command with the given token and hash as well as the overridden arguments I have used with the first node. Kubeadm join downloads first the cluster data and certificates using the given hash key and token, definition and configuration files and then the kubelet creates the static pods. Afterward, the TLS bootstrap is started and the local kubelet is configured to connect to the API server with assigned identity. I have done the same steps to join the worker nodes to the cluster and without specifying the control plane argument in order to tell kubeadm that the new node is a worker node. The kubeadm join workflow also differs, where the kubelet doesn't generate certificates, definition and configuration files for the static pods because there isn't any on the worker nodes as discussed in the second chapter. An example of the output of a worker node that has just joined the cluster is in the following figure:

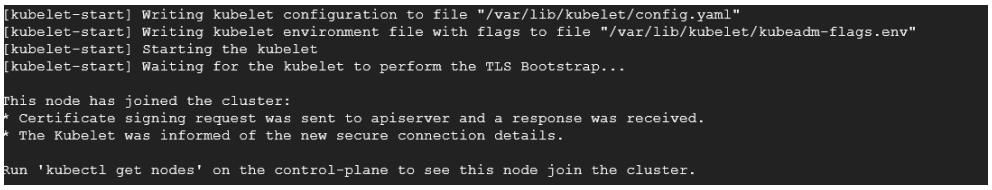


Figure 3.5: Worker Node Join Output

## 3.2 APPLICATION OVERVIEW

As discussed in the methodology section, in order to be able to better evaluate the cluster, I have deployed a microservices applications on the cluster. This section describes the application and its architecture.

The application that this work deploys on the cluster, is a microservices demo applications [7] that follows the cloud native approach. The application is ploygot, this means that the different applications are written in different programming languages and therefore the need to have them deployed as containerized workloads is necessary due to the fact of the several dependencies and runtime environment that each application needs. It uses GRPC and proto buffer for the internal connections and HTTP for connecting to the frontend microservice. I have built the containers locally using skaffold[7] and pushed them to a private container registry.

### 3.2.1 *Microservices List*

This is the list of the microservices involved in the application:

- Frontend Application
- Product Catalogue Application
- Cart Application
- Sugesstions Application
- Shipping Application
- Checkout Application
- Ads Application
- Currency Application
- Payment Application

### 3.2.2 *Overall Architecture*

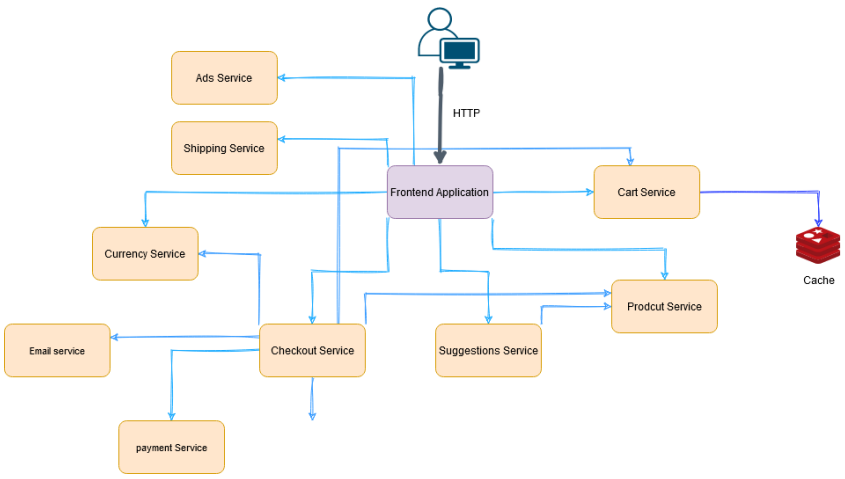


Figure 3.6: Application Architecture

## 3.3 APPLICATION DEPLOYMENT

In Kubernetes, applications and workloads are described in so-called description files. These can be in the form of JSON or YAML files. These files describe Kubernetes resources, objects and their configuration. The description files are then used with the kubectl to interact with the server and deploy the described workload. Kubernetes objects description files must have several fields and this includes the following:

- `apiVersion`
  The version of the Kubernetes API is used to create this object. The installed Kubernetes version must be taken into consideration when specifying this field.

- `kind`
  The kind of object to create for example Deployment, Service, etc

- `metadata`
  The data that uniquely identifies the object, including a name string, UID (User ID), and an optional namespace

- `spec`
  It is the detailed specification of the object. This field can have multiple entries and options and this depends on the type of the described object.

### 3.3.1 *Storage*

Since this is a self-hosted Kubernetes cluster with a cloud-agnostic approach, the cluster initially doesn't support any storage classes by default, contrary to Kubernetes clusters that are running on cloud environments where storage classes have been set for them already.

Therefore before starting with the actual deployment of the application, I have set up the needed storage classes. In order to have a production-grade storage setup, this work uses two types of storage provisioning, dynamic and static storage provisioning. For the static provisioning I have created a local-storage storage class and for the dynamic provisioning, I have created a dynamic storage provisioner with a storage class of type NFS. To set up the local-storage provisioning, I have used a public helm chart called local-storage provisioner. For the NFS storage provisioning, I have set up and configured an NFS server running on the control plane nodes and used a helm chart called nfs-client to support the dynamic provisioning of Persistent Volumes via Persistent Volume Claims.

### 3.3.2 *Cluster Architecture*

After the storage infrastructure is ready and the containers are already pushed to the container registry. I have started defining the application in the form of Kubernetes resources. The following figure illustrates the overall elements and resources I have used to deploy the application. For clarity reasons, I have used two worker nodes and five microservices in the figure, where each has one or two replications.

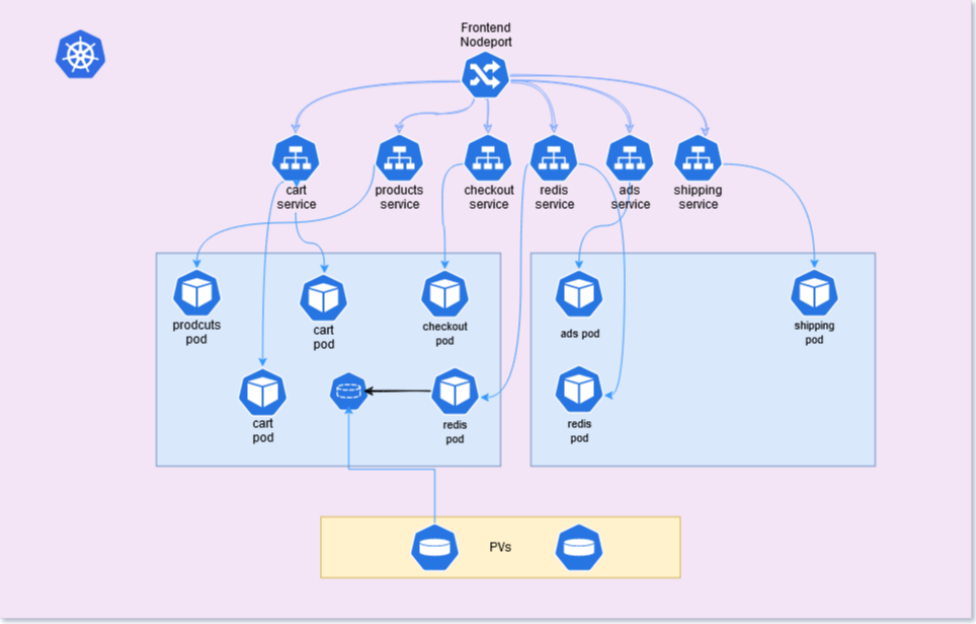


Figure 3.7: Cluster Overview

### 3.3.3 *Deployments*

The first resource I have defined is the Deployment resource. As discussed in the second chapter, using Deployments we can create Replica Sets of the defined container, which in its turn will create the specified amount of containers in the form of pods and manage them. I'm going to take the creation of a Deployment resource for the products microservice as an example to illustrate the process. Notice that the listings used in this chapter are minified and changed the ports.

```
apiVersion:  apps/v1
kind:  Deployment
metadata:
  name:  products
  labels:
    app:  products
spec:
  replicas: 1
  selector:
    matchLabels:
```

```
    app:  products
strategy:
  type: Recreate
template:
  metadata:
    labels:
      app:  products
  spec:
    containers:
    - name:  products
      env:
      - name:  INIT_DUMP
        value:  products-dump.js
      - name:  REDIS_PASSWORD
        valueFrom:
          secretKeyRef:
            name:  products
           key:  cache_password
      ports:
        - containerPort: 8500
      image:  products
    restartPolicy:  Always
```

The first two fields define the type of the resource, in this case `Deployment`, the version of the resource `apps/v1`, and its name `products`. Then in the `spec` field, I have defined the number of `replicas` that I want this Deployment to create and manage. It is also necessary for Deployments to have a `selector.matchLabels.app` that uniquely identifies the deployment object, this field should be identical with the pod name specified in the `template.metadata.labels.app`. The `template` field also has a `spec` key that takes an array of `containers`, where I have defined the container I want this pod to run as well as the ports to expose, the environment variables and the name. Afterward, I have applied the deployments using kubectl apply and the same for the other microservices.
This is the full list of deployments:

```
root@m-node-1:~# kubectl get deployments
NAME                    READY   UP-TO-DATE   AVAILABLE
adservice               1/1     1            1
cartservice             1/1     1            1
checkoutservice         1/1     1            1
currencyservice         1/1     1            1
emailservice            1/1     1            1
frontend                1/1     1            1
loadgenerator           1/1     1            1
paymentservice          1/1     1            1
productcatalogservice   1/1     1            1
recommendationservice   1/1     1            1
redis-cart              1/1     1            1
shippingservice         1/1     1            1
```

Figure 3.8: Deployments List

### 3.3.4 *Services*

Since Pods cannot be accessed directly, therefore and as illustrated in the figure of the cluster architecture, the next step is defining the necessary services to forward requests to the pods. this will expose and make them accessible in the cluster or to the outside world depending on the use case and the type of the service. In this work, I have used two types of Services, Node port and cluster IP. I have used these services to expose the different microservices to each other inside of the cluster, the following is an example of a service of type cluster IP that I have used in order to have the products catalog microservice exposed, so the frontend microservice for example can reach it and request the list of the available products.

```
apiVersion: v1
kind: Service
metadata:
  name:  products-svc
  labels:
    app:  products-svc
spec:
  ports:
  - port: 8500
    name:   products
  selector:
    app:  products
```

The second type is the Nodeport type, I have used this service in order to expose the frontend microservice to the outside world. When using the Node port service, the control plane allocates a static IP address on all of the worker nodes and exposes the container specified in the `selector` field of the definition on this port, we can also specify the port that we want to expose as I did in this work and illustrated in the following listing:

```
apiVersion: v1
kind: Service
metadata:
  name:  front-svc
  labels:
    app:  front-svc
spec:
  type: Nodeport
  ports:
  - port: 8500
    name:  frontend
  selector:
    app:  frontend
```

This is the full list of services:

```
root@m-node-1:~# kubectl get svc
NAME                    TYPE        CLUSTER-IP       EXTERNAL-IP
adservice               ClusterIP   10.107.125.254   <none>
cartservice             ClusterIP   10.102.236.17    <none>
checkoutservice         ClusterIP   10.102.123.37    <none>
currencyservice         ClusterIP   10.107.178.30    <none>
emailservice            ClusterIP   10.102.5.229     <none>
frontend                ClusterIP   10.101.53.238    <none>
frontend-external       NodePort    10.111.8.234     <none>
kubernetes              ClusterIP   10.96.0.1        <none>
nginx                   ClusterIP   None             <none>
paymentservice          ClusterIP   10.111.161.228   <none>
productcatalogservice   ClusterIP   10.101.153.197   <none>
recommendationservice   ClusterIP   10.111.239.152   <none>
redis-cart              ClusterIP   10.102.56.164    <none>
shippingservice         ClusterIP   10.102.218.8     <none>
```

Figure 3.9: Services List

### 3.3.5 *Stateful Set*

Since the application uses an in-memory DB to store the user's cart, the Redis deployment must be stateful and has a stable network and storage identifiers. Therefore I have used Stateful Set to deploy the Redis instances because it offers network and storage stability.

- Stable Network
The Pods in a Stateful Set are uniquely identified using an index that is assigned to each Pod by the Stateful Set controller. The Pod's names take the form `<stateful set name>-<index>`. Since the Redis Stateful Set has three replicas, it creates three Pods with following names, `redis-0`, `redis-1` and`redis-2`. The same names are also used for the hostname of each of the pods. Stateful Sets use a headless service to control the domain of its pods. The domain managed by this Service takes the form:`<service name>.-<namespace.svc.cluster.local` and the pod takes the following: `<podname>.-<governing service domain>`.default.cluster.local, where the governing service is defined by the `spec.serviceName` field on the Stateful Set definition.

```yaml
apiVersion: v1
kind: Service
metadata:
  name:  redis-hs
  labels:
    app:  redis-hs
spec:
  ports:
  - port: 80
    name:  redis
  clusterIP: None
  selector:
```

```
    app:  redis
```

Notice that for headless services it is necessary to give None as a value for the key`spec.clusterIP`

- Stable Storage

Since I have configured an NFS dynamic storage provisioner, Kubernetes will create one PersistentVolume for each `VolumeClaimTemplate`. In the Redis example below, each Pod will receive a single PersistentVolume with a storage class of `nfs` and 1 Gib of provisioned storage. When one of these Pods is (re)scheduled, its `volumeMounts` mount the PersistentVolumes associated with its PersistentVolume Claims. Plus the PersistentVolumes associated with the Pods PersistentVolume Claims is not deleted when the Pods or the Stateful Set are deleted. In order to define the above discussed resources, I have first created the following Headless Service to be responsible for the network identity of the `redis` Stateful Set:

After creating the healdess service, I have created the Stateful Set using the following definition:

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name:  redis
spec:
  serviceName: "redis-hs"
  replicas: 3
  template:
    metadata:
      labels:
        app:  redis
    spec:
      terminationGracePeriodSeconds: 10
      containers:
      - name:  redis
        image:  redis
        ports:
        - containerPort: 8500
          name: web
        volumeMounts:
        - name: redis-nfs
          mountPath: /mnt/redis
  volumeClaimTemplates:
  - metadata:
      name: redis-nfs
    spec:
      accessModes: [ "ReadWriteOnce" ]
      resources:
        requests:
          storage: 1Gi
```

One more thing to notice, is that the `spec.serviceName` key is used to bind the Stateful Set to the headless service and therefore the values has to be identical.

# 4 EVALUATION

In order to evaluate the proposed solution, this work conducts several types of observations and experiments on the control plane as well as the worker nodes, these are based on the criteria described in the first chapter and their categories.

## 4.1 SECURITY EXPERIMENT

In this experiment, the security evaluation criteria is addressed, it consists of a security check of the cluster core elements against the Center of Internet Security Benchmark for Kubernetes. It is one of the most popular Benchmarks created by the CIS and it is used to check and boost the security of Kubernetes clusters and ensure that best practices are applied. This includes several security aspects such as network policies, access control, admin privileges ,etc.

### 4.1.1 *API Server*

The CIS tests for the API-server are the following:

| Criteria | Value | Test |
|---|---|---|
| `-anonymous-auth` argument is set to `false` | `false` | valid |
| `-basic-auth-file` argument is not set | `null` | valid |
| `-insecure-allow-any-token` argument is not set | `null` | valid |
| `-kubelet-https` argument is set to `true` | `true` | valid |
| argument kubelet-client-key is set to appropriate | `true` | valid |
| argument kubelet-client-crt is set to appropriate | `true` | valid |
| `-insecure-bind-address` argument is not set | `null` | valid |
| admission control plugin `AlwaysAdmit` argument is not set | `null` | valid |
| `-authorization-mode` argument is not set to `AlwaysAllow` | `RB,Node` | valid |
| admission control plugin `AlwaysPullImages` argument is set | `AlwaysPullImages` | valid |
| admission control plugin `SecurityContextDeny` is set | true | not valid |
| admission control plugin NamespaceLifecycle is set | false | valid |
| `-token-auth-file` is not set | `null` | valid |

Table 4.1: CIS API - Server

### 4.1.2 *Scheduler*

The CIS checks for the scheduler are as follows:

| Criteria | Value | Test |
|---|---|---|
| `-profiling` argument is set to `false` | `false` | valid |
| `-address` argument is set to `localhost` | `localhost` | valid |

Table 4.2: CIS Scheduler

### 4.1.3 *Controller*

The CIS checks for the controller are as follows:

| Criteria | Value | Test |
|---|---|---|
| `-terminated-pod-gc-threshold` is set as appropriate | `10000` | valid |
| `-profiling` argument is set to `false` | false | valid |
| `-use-service-account-credentials` is set to `true` | false | valid |
| `-service-account-private-key-file` is set as appropriate | `/pki/sa.key` | valid |
| `-root-ca-file` is set as appropriate | `/pki/ca.crt` | valid |
| `-address` argument is set to `localhost` | `localhost` | valid |

Table 4.3: CIS - Controller

### 4.1.4 *Configuration Files*

The CIS checks for the configuration files are as follows:

| Criteria | Value | Test |
|---|---|---|
| etcd pod specification file ownership is set to root:root | `root:root` | valid |
| scheduler pod specification file ownership is set to root:root | `root:root` | valid |
| ownership of admin.conf is set to root:root | `root:root` | valid |
| API server pod specification file ownership is set to root:root | `root:root` | valid |
| controller manager pod specification ownership is set to root:root | `root:root` | valid |

Table 4.4: CIS - Configuration Files

### 4.1.5 *Etcd*

The CIS checks for Etcd are as follows:

| Criteria | Value | Test |
|---|---|---|
| `-cert-file` and `-key-file` are set as appropriate | `etc/kubernetes/pki/etcd` | valid |
| etcd data directory ownership is set to `etcd:etcd` | `etcd:etcd` | valid |
| –auto-tls argument is not set to true | `null` | valid |
| –peer-cert-file and –peer-key-file are set as appropriate | `true` | valid |
| –peer-client-cert-auth argument is set to true | `true` | valid |

Table 4.5: CIS - Etcd

### 4.1.6 *Kubelet*

The CIS tests for the kubelet are as follows:

| Criteria | Value | Test |
|---|---|---|
| –anonymous-auth argument is set to false | `false` | valid |
| –authorization-mode argument is not set to AlwaysAllow | `null` | valid |
| –client-ca-file argument is set as appropriate | `true` | valid |
| –read-only-port argument is set to appropriate | `0` | valid |
| –streaming-connection-idle-timeout argument is not set to 0 | `null` | valid |

Table 4.6: CIS - Kubelet

### 4.1.7 *Worker Nodes*

The CIS tests for the worker nodes are as follows:

| Criteria | Value | Test |
|---|---|---|
| kubelet service file ownership is set to root:root | `root:root` | valid |
| proxy kubeconfig file ownership is set to root:root | `root:root` | valid |
| kubelet configuration file ownership is set to root:root | `root:root` | valid |
| client certificate authorities file ownership is set to root:root | `root:root` | valid |
| kubelet.conf file ownership is set to root:root | `root:root` | valid |

Table 4.7: CIS - Worker Nodes

## 4.2 MONITORING

For monitoring the different aspects of the cluster infrastructure as well as the application, this thesis uses several monitoring open source technologies, essentially Prometheus, Grafana and Linkerd. These technologies and their installation are discussed in the reminder of this section. Prometheus is an open-source monitoring toolkit and it was chosen for this work because Kubernetes core elements expose their metrics at /metrics in a Prometheus format. it is also efficient due to its pull style in collecting metrics. Prometheus architecture consists of multiple applications, the following are the essential ones:

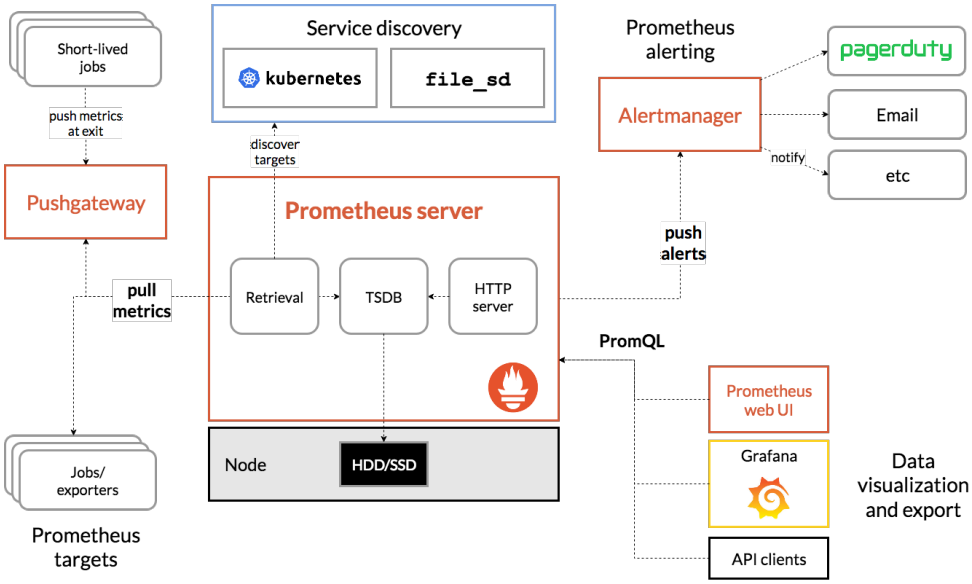


Figure 4.1: Prometheus Architecture[21]

- Server: It is the central element, it retrieves data using it's data retrieval worker that has access to the different application endpoints running on the cluster and saving those in the DB[21] . It has also an HTTP server that processes the query language to push the collected data to the different applications including dashboards for visualization purposes, as it is done in this work.

- Client Libraries: These applications provide support and integration for the different applications.

- Push gateway: As the name says, it is a gateway responsible for pushing all of the collected data to the database.

- Exporters: They are external applications that retrieve data in other formats, convert it and then expose it in Prometheus format to the /metrics endpoint.

Scrapping: Prometheus sends an HTTP request called a scrape in order to fetch metrics. It sends scrapes to targets based on their configuration. Each target, which can be statically configured or found dynamically, is scraped at a regular interval called a scrape interval. Each scrape reads the /metrics HTTP endpoint of the target to get the desired client metrics and persists those values in the Prometheus time-series database.

In Prometheus, there are several types of data, this work uses the following:

- Counter: The value of a counter will always increase. It can be increased or reset to zero. So, if a scrape fails, it only means a missed data point. Examples: total number of HTTP requests received or the number of exceptions.

- Gauge: A gauge is a snapshot at a well-specified point in time[21] . It can both increase or decrease. If a data fetch fails, you lose a sample; the next fetch might show a different value: example disk space, memory usage.

- Histogram: A histogram samples observations and counts them in configurable buckets. They are used for things like request duration or response sizes. They can be used to measure request duration for HTTP requests and in this case, it will have a set of buckets.

- Summary: It provides a total count of observations and a sum of all the observed values. It allows therefore averaging the values and it is used typically for observing request duration or response sizes.

Grafana:
In order to be able to better observe and analyze the experiments, this work uses Grafana as visualization tool[8] . Grafana is an open source analytic and visualization tool that has a full integration with Prometheus.

Installation:
This work uses the Prometheus operator for installing the discussed tools. it creates, manages and configures Prometheus monitoring instance and defines additional resources as extensions to Kubernetes, this includes, service monitor and metrics-server and a grafana instance. The next figure shows the used scripts used for the installation.

Linkerd:
Linkerd is a mesh tool for Kubernetes, it provides several features such as runtime debugging, observability, reliability and others, however for this work it used essentially for observing the meshed application, because it is one of linkerd's most powerful features as it has extensive set of tooling around observability. Linkerd adds a side-container to the meshed pods in

```
kubectl -n monitoring apply -f prometheus/prometheus-operator/
kubectl -n monitoring apply -f metrics-exporters-with-service-
monitors/node-exporter/
kubectl -n monitoring apply -f metrics-exporters-with-service-
monitors/kube-state-metrics/
kubectl -n monitoring apply -f alert-manager-with-service-monitor/
kubectl -n litmus apply -f metrics-exporters-with-service-
monitors/litmus-metrics/chaos-exporter/
kubectl -n litmus apply -f metrics-exporters-with-service-
monitors/litmus-metrics/litmus-event-router/
kubectl -n monitoring apply -f prometheus/prometheus-configuration/
kubectl -n monitoring apply -f grafana/
kubectl edit svc prometheus-k8s -n monitoring
```

Figure 4.2: Monitoring Installation

order to intercept all the in and outbound traffic. The inject command simply adds an annotation, linkerd.io/inject: enabled specifically this involves implementing an initContainer (linker-init) and container (linkerd-proxy) along with the original following the sidecar design.

## 4.3 LOAD EXPERIMENT

Load testing is used to determine the performance of the application and infrastructure and how well it behaves by simulating production-like traffic when it is being accessed simultaneously by a lot of users. With this experiment, the latency, response times, and success rate of the requests are evaluated.

### 4.3.1 *Experiment Design*

For generating the load this work has used k6, an open-source load testing tool. The script executed is written in nodejs and simulates what a real user would when it uses the application. In the load testing process, all the microservices are considered and tested. An overview is like the following, the virtual users hit the frontend application, browse the available products, adding products to their cart, and purchase, this also includes the Redis cache as well as the other microservices, such as the checkout, ads, delivery microservices.

### 4.3.2 *Experiment Settings*

- Virtual concurrent users: 50 virtual user
- Experiment Duration: 10minutes
- Load zone: Ashburn
- Ramping profile: Load

- Requests per execution: 45
- Requests types:GET, POST

### 4.3.3 *Experiment Analysis*


Figure 4.3: Latency Experiment - products microservice

As can be observed in the figure, line graphs are used to visualize the three golden metrics and these are the graphs of the products microservice, which I'm going to take as a reference for this section. The x-axis in all of the three graphs represents the time of the experiment and it is a five minutes interval, in other words, the second half of the whole experiment duration. Beginning with the request rate graph, as it represents the load generated during the experiment towards the products microservice. The line graph depicts a value point for each request, therefore the Y-axis is in requests per second. We can observe that the line graph can be divided into two sections, the first is uniform and steady, this is during the peak of the load and the second is at the last two minutes of the experiment where the RPS has significantly decreased, this is during the ramp down. The highest rps achieved was at 380rps and the lowest is 250rps.

Now when looking at the latency graph, where the latency is represented in milliseconds on the Y-axis. In this graph latency percentiles are used because the average latency can be misleading as it doesn't exactly reflect the user's experience or the real performance, and this can also be seen on the graph. There are three percentiles used P99, P95 and P50. For example, a P99 with a value point of 50ms means that 99 out of 100 users have experienced a latency of 50ms. I'm going to focus on the P99 because it is the tail of the distribution. As can be observed on the graph the P99 had an average of 45ms during the first four minutes, this where the RPS was at its highest and once the ramp down has started it started to fall down significantly till it has reached a P99 of 25ms.

The success rate graph reflects how many responses are HTTP status codes indicating a successful execution on the server, as well as any request that times out before receiving or completing its response. The Y-axis is represented in percentile and we can observe on the graph that during the experiment all the requests were executed with a success rate of 100 percent.

## 4.4 CHAOS AUTOSCALING EXPERIMENT

Chaos experiments are based on fault injection technique. It is a testing technique to test the resiliency and robustness of infrastructures and applications and how they deals with failures.There are a few tools used for chaos injection, this work has used litmus. Litmus is a cloud-native chaos engineering platform that orchestrates chaos on Kubernetes environments[16]. Litmus consists of Chaos Server, Runner and Engine, there roles are as follows:

- Chaos Server:
It is the central element that manages and watches for the chaos engine objects and sits in the middle between them and the chaos runner. It also triggers the chaos exporter to export the chaos details to be visualised.

- Chaos engine:
It links the targeted application to the chaos experiment and sits between them and the chaos server.

- Chaos Hub Is the public chaos repository, where the generic chaos experiments are hosted, this includes the ones used in this work.

Another optional elements is the chaos dashboard, it is used to visualize experiments, however this work has used grafana for the visualization of the experiments.
To install the chaos engine the following definition file is used:

```
kubectl apply -f https://litmuschaos.github.io/litmus/litmus-operator
```

Figure 4.4: Monitoring Installation

The chaos experiment that this work conducts is the autoscaling experiment and it aims to check how the cluster responds to the various amount of trafscalerI have chosen to conduct this experiments on the products microservice, since it is the one responsible for providing the data products data to the frontend, where the user experience should be at its best and be able to response to the high demnd, autoscale and keep the perfrmnce at the neccesary level as well as being able to scale down and minimize resource usage. For this type of tests litmus provide the Pod Autoscaler experiment. This experiment scales the application replicas specified and tests the autoscaling on the cluster.

First I have installed the experiment where the application is running. Then the chaos can be triggered with installing the chaos engine on the cluster. I have also created a service account as well as a cluster role and a cluster role binding, in order to provide the chaos engine with the needed roles permissions to access the Kubernetes server. Once the engine has the right permissions, I have defined the experiment using a YAML file with

kind of Chaosengine and specified the targeted deployment, the desired experiment type and the number of replicas to autoscale as illustrated in the listing below and applying it.

```
apiVersion: litmuschaos.io/v1alpha1
kind: ChaosEngine
metadata:
  name:  frontend-chaos
  namespace: default
spec:
  annotationCheck: 'false'
  engineState: 'active'
  appinfo:
    appns: 'default'
    applabel: 'app=frontend'
    appkind: 'deployment'
  chaosServiceAccount: pod-autoscaler-sa
  monitoring: false
  jobCleanUpPolicy: 'delete'
  experiments:
    - name: pod-autoscaler
      spec:
        components:
          env:
            # number of replicas to scale
            - name: REPLICA_COUNT
              value: '5'
```

### 4.4.1 *Experiment Design*

- Entry Criteria: Application pods are healthy on the respective nodes before chaos injection.
- Exit Criteria: The right number of replicas as specified in the YAML are running on the respective nodes.

### 4.4.2 *Experiment Settings*

- Targeted replicaset: frontend
- Initial replicas: 2

### 4.4.3 *Experiment Analysis*

In order to observe the results, I have built the Prometheus and grafana board below:


Figure 4.5: Prometheus - Autoscaling Experiment


Figure 4.6: Grafana - Autoscaling Experiment

As can be seen in the figure above the targeted replicaset has minimum of one replicaset, therefore the autoscaler has down scaled it to one replica as the load is at it's minimum. Then when I have launched the experiment and chaos started, we can easily observe how the replicas have started to be scaled up till by the autoscaler till they have reached the specified number of running replicas. Notice that the desired number of replicas represented by the green line as shown in the legend, it is on the 0 for the whole experiment, this is due to the fact that I haven't set any desired number on the replicaset resource so it is fully up to the autoscaler to decide what number of replicas should be running.

To get a better idea on the performance of the of the autoscaling on the worker nodes and how the pods are rescheduled, I have also used the following Prometheus queries to obtain the duration of pod startups when created by the kubelet, these are as follows:

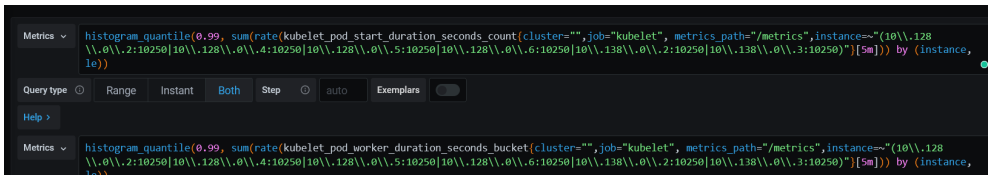


Figure 4.7: Grafana - Pods Startup Duration


Figure 4.8: Grafana - Pods Startup Duration

As shown in the above figure, the highest startup duration in the observed experiment was around 450ms, what can be considered as a decent amount to get a pod up and running.

# 5

# DISCUSSION

As demonstrated in the third chapter, setting up an in-house Kubernetes cluster using kubeadm with high availability and security best practices, is a lot more of a daunting task than using managed Kubernetes solutions. For instance, I have had to manage all of the underneath infrastructure such as cloud storage, routing, access control, TLS, hardware configuration, monitoring and the list goes on. This work has also demonstrated working with containerd as a container runtime instead of docker, what is considered as a bleeding edge setting for Kubernetes clusters. It also follows Kubernetes storage provisioning best practices because I have setup a dynamic storage provisioner of type NFS on the control plane nodes. In addition, explains how to achieve cluster high availability with using multi-nodes control plane with HAproxy instances and explains further the design decisions for the cluster topology and the chosen number of nodes and networking plugin.

As shown in the security experiment, the cluster validated almost all of the suggested security best practices according the CIS except for the this is due to the fact I have used security pod policy to enhance the security on pods level and to minimize their privileges in relation to the host system what helps in avoiding privilege escalation risks, therefore this feature has to be set to false.

As result, I can conclude that the cluster provides the needed level of security for a production use. However most of the security configurations were self configured because they weren't the default in kubeadm, what can be considered as a significant additional administrative overhead.

The second experiment has addressed the network latency evaluation criteria. Besides the importance of network latency for the user experience, it is also a very important indicator of the cluster performance and networking choices as well as the application. The latency experiment has shown that the proposed cluster implementation has the ability to deliver web services with a decent network latency. Where also the choices of the CNI plugin as well as the CNI data plane has a key role. As already mentioned in the chapter three, this implementation has used an overlay-less network to avoid additional en/decapsulation overhead. Another enhancement of the implemented solution, especially when aiming for scalability is to use extended Berkeley PF for the CNI. It should also be said that when scaling the cluster and adding additional worker nodes the CPU and memory usage should be taken into consideration because of the virtual Ethernet card used.

Pods Autoscalability is a key feature when deploying microservices, it is also challenging to have such features in a cloud-agnostic environments because these are generally meant for public cloud usage. Autoscaling ensures that the cluster responds quickly and efficiently to changes in demand with-

out causing scheduling errors, evicting Pods and using resources etc. As shown in the autoscaling experiment, the proposed cluster implementation has proven to be able to autoscale deployments and replicasets in in both directions up and down according to the inbound traffic. The considerable limitation of this implementation specifically and self-hosted Kubernetes deployments generally is Node Autoscaling, as hardware resources should be first provisioned and configured.

# 6

# CONCLUSION

This work was conducted in order to investigate the deployment of production-ready Kubernetes cluster using kubeadm with a private-cloud approach to host microservices applications, where all of the infrastructure is self-hosted. This was illustrated in the chapter three with providing an outline of architecturing, configuring and setting up Kubernetes in a high availability mode using kubeadm and open source technologies.

The second significant contribution is to evaluate the implemented solution, in order to provide measurable insights on how well can the proposed solution perform. What can help infrastructure engineers or anyone looking looking to setup Kubernetes without depending on public cloud providers offers, in the decision making when considering setting up high available self-manged Kubernetes cluster. Where the also the experiments have shown an encouraging insights regarding the performance and for adopting the cloud native technologies in private cloud environments, what open new perspectives in this direction, such as the discussed work in the next chapter.

# 7

# FUTURE WORK

This work has demonstrated an encouraging insights on adopting the cloud native technologies in private cloud environments. Therefore a very interesting work would be to investigate furthermore the adoption of serverless architecture on Kubernetes on-premises. A serverless architecture is a way to build and run applications and services without having to manage infrastructure[20]. The inspiration of the idea is a project called OpenFaas, It is a platform for building serverless functions with Docker and Kubernetes [24]. Faas stands for Function as a service and it an alternative that public cloud providers offers to Software as a Service.

# BIBLIOGRAPHY


[1] Robert Botez and V. Dobrota C. Iurian I. Ivanciu. "Deploying a Dockerized Application With Kubernetes on Google Cloud Platform." In: *2020 13th International Conference on Communications (COMM)* (2020), pp. 471–476.

[2] Z. Dehghani. *How to break a Monolith into Microservices*. Available at https://martinfowler.com/articles/break-monolith-into-microservices.html (Last Accessed:01.03.2021). 2018.

[3] *Distributed Key-Value Store DB*. URL: https://etcd.io/.

[4] Dr-Hassan Elkatawneh. "Comparing Qualitative and Quantitative Approaches." In: *Econometrics: Econometric Statistical Methods - General eJournal* (2016).

[5] *Empowering App Development for Developers*. Available at https://www.docker.com/ (Last Accessed:01.03.2021).

[6] *Getting your business GDPR ready for Cloud*. Available at https://www.rinodrive.com/getting-your-business-gdpr-ready-private-cloud-vs-public-cloud-explained/ (Last Accessed:01.02.2021).

[7] GoogleCloudPlatform. *GoogleCloudPlatform/microservices-demo*. URL: https://github.com/GoogleCloudPlatform/microservices-demo.

[8] *Grafana: The open observability platform*.

[9] *Innovate Everywhere*. Available at https://rancher.com/ (Last Accessed:01.03.2021)journal=Rancher Labs.

[10] *Kubernetes API-Server Design Proposal*.

[11] *Kubernetes Docs*. URL: https://kubernetes.io/docs/home/.

[12] *Kubernetes Networking Design Proposal*.

[13] *Kubernetes Scheduler Design Proposal*.

[14] *Kubernetes Storage(PV,PVC) Design Proposal*.

[15] W. Tarneberg andKlein L. Larsson, Erik Elmroth, and M. Kihl. "Impact of etcd deployment on Kubernetes, Istio, and application performance." In: *Software: Practice and Experience* 50 (2020), pp. 1986 –2007.

[16] *Litmus*. URL: https://litmuschaos.io/.

[17] Farzana A. Bhuiyan M. Shamim and A. Rahman. "XI Commandments of Kubernetes Security: A Systematization of Knowledge Related to Kubernetes Security Practices." In: *2020 IEEE Secure Development (SecDev)* (2020), p. 85.

[18] Victor Marmol and T. Hockin R. Jnagal. *Networking in Containers and Container Clusters*. 2015.

[19] Digital Ocean. *Getting-Started-with-Containers-and-Kubernetes*. 2020.

[20] *OpenFaas*. Available at https://www.openfaas.com/ (Last Accessed:01.03.2021).

[21] *Overview: Prometheus*. Available at https://prometheus.io/docs/introduction/overview/ (Last Accessed:01.03.2021).

[22] C. Pahl. "Containerization and the PaaS Cloud." In: *IEEE Cloud Computing* 2 (2015), pp. 24–31.

[23] *Production-Grade Container Orchestration*. Available at https://kubernetes.io/ (Last Accessed:01.03.2021).

[24] *Serverless Architecture, AWS*. Available at https://aws.amazon.com/fr/lambda/serverless-architectures-learn-more/ (Last Accessed:01.03.2021).

[25] *The world's lightest, fastest service mesh.* Available at https://linkerd.io/ (Last Accessed:01.03.2021).