



Performance evaluation of wireguard in kubernetes cluster.

Pavan Gunda
Sri Datta Voleti

This thesis is submitted to the Faculty of Computing at Blekinge Institute of Technology in partial fulfilment of the requirements for the degree of Master of Science in Telecommunication Systems. The thesis is equivalent to 20 weeks of full time studies.

The authors declare that they are the sole authors of this thesis and that they have not used any sources other than those listed in the bibliography and identified as references. They further declare that they have not submitted this thesis at any other institution to obtain a degree.

Contact Information:

Author(s):

Pavan Gunda

E-mail: pavangunda66@gmail.com

Sri Datta Voleti

E-mail: sridatta1996@gmail.com

University advisor:

Patrik Arlos

Department of Computer Science

Faculty of Computing
Blekinge Institute of Technology
SE-371 79 Karlskrona, Sweden

Internet : www.bth.se
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57

Abstract

Containerization has gained popularity for deploying applications in a lightweight environment. Kubernetes and docker have gained a lot of dominance for scalable deployments of applications in containers. Usually, kubernetes clusters are deployed within a single shared network. For high availability of the application, multiple kubernetes clusters are deployed in multiple regions, due to which the number of kubernetes clusters keeps on increasing over time. Maintaining and managing multiple kubernetes clusters is a challenging and time-consuming process for system administrators or DevOps engineers. These issues can be addressed by deploying a kubernetes cluster in a multi-region environment. A multi-region kubernetes deployment reduces the hassle of handling multiple kubernetes masters by having only one master with worker nodes spread across multiple regions. In this thesis, we investigated a multi-region kubernetes cluster's network performance by deploying a multi-region kubernetes cluster with worker nodes across multiple openstack regions and tunneled using wireguard(a VPN protocol). A literature review on the common factors that influence the network performance in a multi-region deployment is conducted for the network performance metrics. Then, we compared the request-response time of this multi-region kubernetes cluster with the regular kubernetes cluster to evaluate the performance of the deployed multi-region kubernetes cluster. The results obtained show that a kubernetes cluster with worker nodes in a single shared network has an average request response time of 2ms. In contrast, the kubernetes cluster with worker nodes in different openstack projects and regions has an average request-response time of 14.804 ms. This thesis aims to provide a performance comparison of the kubernetes cluster with and without wireguard, factors affecting the performance, and an in-depth understanding of concepts related to kubernetes and wireguard.

Keywords: Kubernetes, Docker, Wireguard, OpenStack, Performance analysis, container network interface, Network overlay

Acknowledgments

We would like to express our sincere gratitude to **Patrik Arlos** for his valuable support and inputs throughout the thesis. It would not have been possible to accomplish this without his supervision and help.

We would like to extend our deepest and most sincere regard to **Vida Ahmadi** for her continuous support, invaluable guidance and encouragement throughout the thesis. We would like to thank **Kim Hindart** for motivating us throughout the thesis.

Finally, we would like to extend our heart full love and gratitude to our parents, families, and friends for showing their moral support and backing us in all situations. This thesis would not be complete without their support. Thank you very much, everyone

Contents

Abstract	i
Acknowledgments	ii
1 Introduction	1
1.1 Aim and Objectives	3
1.2 Research questions	3
2 Related Work	5
3 Background	7
3.1 Cloud Computing	7
3.2 Virtualization	8
3.3 Containers	8
3.4 OpenStack	9
3.5 Docker	9
3.5.1 Namespaces	10
3.5.2 Cgroups	10
3.6 Kubernetes	10
3.6.1 Node	11
3.6.2 Pod	12
3.6.3 Deployment	12
3.6.4 Daemonset	12
3.6.5 Namespaces	12
3.6.6 Services	12
3.6.7 Kubernetes Ingress	13
3.6.8 Container Network Interface(CNI)	13
3.7 Wireguard	14
3.8 Rancher Kubernetes Engine(RKE)	14
3.9 Ansible	15
4 Method	16
4.1 Research Method	16
4.2 Literature review	17
4.2.1 Search strategy	17
4.2.2 Snowballing	18
4.2.3 Digital libraries	18
4.2.4 Inclusion-Exclusion Criteria	18

4.2.5	Data Extraction and analysis	19
4.3	Experimentation	19
5	Implementation of Experimental setup	21
5.1	Modeling the Service Architecture	21
5.2	Environment for Scenario 1	22
5.3	Environment For Scenario 2	23
5.4	Environment For Scenario 3	25
5.4.1	Hosting the Application In Three Kubernetes Clusters	26
5.4.2	Exposing The Deployed Application to Internet	27
5.5	Data Collection	28
6	Results and analysis	29
6.1	29
6.1.1	performance of kubernetes in a single shared network	29
6.1.2	performance of single region kubernetes with wireguard	31
6.1.3	Performance of multi-region kubernetes with wireguard	33
6.2	Research Questions and Answers	35
7	Discussion	36
8	Conclusion and Future Work	37
8.1	Conclusion	37
8.2	Future Work	38
	References	39
A	Deploy environment for scenario-1 using cluster.yaml	42
A.1	Ansible playbooks	45
A.1.1	Bash sript used for request response time	48

List of Figures

3.1	virtual machine architecture	8
3.2	container architecture	9
3.3	Architecture of a Kubernetes cluster	11
3.4	Node architecture	11
3.5	Inter pod communication	14
4.1	Research methodology	17
5.1	kubernetes client server model	22
5.2	Master and worker nodes and their assigned roles	22
5.3	Network architecture of scene-1	23
5.4	Wireguard network overlay	25
5.5	Description of a deployment	27
6.1	raw data of results from scenario-1	29
6.2	histogram plot of obtained data x-axis=response time in ms and y-axis=number of requests	30
6.3	ecdf for n=10000 x-axis=logarithmic response time in ms and y-axis=number of request	30
6.4	raw data of results from scenario-2	31
6.5	histogram plot of obtained data x-axis= response time in ms and y-axis=number of request	32
6.6	ecdf for n=10000 x-axis=logarithmic response time in ms and y-axis=number of request	32
6.7	raw data of results from scenario-3	33
6.8	histogram plot of obtained data x-axis=logarithmic response time in ms and y-axis=number of request	33
6.9	ecdf for n=10000 x-axis=logarithmic response time in ms and y-axis=number of request	34
8.1	average values response times in three scenarios	38
A.1	Successful cluster formation using RKE	43
A.2	deployment file	43
A.3	configuration file for wireguard	44
A.4	service file for NodePort	45
A.5	Ansible inventory list	46
A.6	A Kubernetes deployment from command line	47
A.7	Pods formed for a deployment	47

A.8 Output of Running Services	47
A.9 Kubernetes ingress resource sample yaml file	48

Rapid growth in the use of web applications through the internet has raised the demand for computational resources for application deployment and operation. Providing such resources using conventional computing methods would increase the requirement of expensive computing such as servers, routers, and switches. These issues led to a new way of managing resources called cloud computing. Cloud computing enabled computational resources such as virtual machines and containers to be rented and acquired on-demand, and the final cost depends on the quantity and duration of resources rented, also called as pay as you go model [3].

Cloud computing started as a service to provide on-demand virtual machines. Although virtual machines can be scaled up and down depending upon the number of users in minutes, it boots up with a full operating system installed, making it heavy and not cost-effective. On the other hand, containerization has gained significant popularity these days as containers are lightweight and portable and can be scaled within seconds compared to virtual machines [6]. Containers are considered more lightweight than virtual machines, as they share the kernel with host OS, while maintaining separate execution environments [18]. A Container in the context of cloud computing is an approach to operating system virtualization where user can run some code with its dependencies and resource files in an isolated environment.

However, there are challenges in controlling the demand for scaling in containers. Though container engines such as docker creates and runs containers [15], a container stops when the process running in the container exits. So, whenever we need to rerun a stopped container or scale the number of containers depending upon the number of users accessing the application, admins must manually input a command, which is not an ideal scenario. This is where we can use kubernetes to orchestrate the lifecycle of containers. Kubernetes is a container orchestration system for running and coordinating containerized applications across clusters of machines [3], which means it can auto-heal and auto-scale a set of containers depending upon the number of users accessing it.

However, Kubernetes comes with its limitations. One such restriction is, a Kubernetes cluster can only be deployed in a single shared network [23]. To have a multi-region deployment of an application, multiple Kubernetes clusters have to be deployed in multiple regions. In this scenario, sysadmins have to manually upgrade each cluster or manage all clusters using a solution called kubefed. Kubefed is a solution with a host cluster and member clusters (multiple clusters across multiple regions). We can manage the member clusters using the host cluster. This results in intricate architecture and a massive hassle in setup. Kubefed (also called Kubernetes

federation or synced solution) has its disadvantages.

The main disadvantages of Federation v2(kubefed) are

- It introduces an overarching federation control plane, turning the cluster into a massive meta-cluster which is hard to troubleshoot
- Highly sensitive to version each cluster is running on, making the maintenance challenging.

Due to above disadvantages, Kubefed (an alias of Federation v2) is still not a robust solution for deploying production grade clusters [23].

As a DevOps Engineer or a System Administrator, one may have to write scripts to update and upgrade the cluster and the deployed application frequently. As federation is not production-ready, one may have to use the traditional approach, that is, to run the scripts multiple times in multiple clusters and also monitor numerous clusters, which is a hassle, challenging, and time taking.

Some of the challenges are

- It is time-consuming to upgrade each cluster version
- Deploying the application in multiple clusters is a hassle as each cluster has different authorization certificates
- Troubleshooting is also time-consuming

Having a single Master with worker nodes across different cloud regions is an approach to reduce a lot of hassle while maintaining and managing the workloads and still being a multi-region highly available deployment. This multi-region deployment of kubernetes can be achieved by tunneling the worker nodes in multiple regions using a vpn protocol as an overlay network. There are multiple vpn protocols, wireguard is one such protocol, which is lighter and offers encryption with less overhead when compared to other vpn protocols such as IPsec and OpenVPN.

In this thesis work, we attempted to address the above challenges by creating and deploying a multi-region kubernetes cluster in OpenStack using wireguard as a network overlay within the kubernetes cluster (instead of using federated solutions with multiple masters or having the hassle of upgrading each cluster). We would then compare the performance of these clusters with a regular kubernetes cluster.

These kubernetes clusters are deployed in an OpenStack IaaS environment. openstack is an open-source cloud computing infrastructure project. An OpenStack region is a geographical region with a data center.

This thesis aims to investigate the performance of a kubernetes cluster with master in any desired openstack region and worker nodes spread across different openstack regions and tunneled using wireguard, when compared to a regular kubernetes cluster.

1.1 Aim and Objectives

The main aim of the thesis is to address the issues of maintaining and managing the kubernetes nodes along with addressing the Research gaps within the existing Scientific Studies (as described in chapter 3) by creating a multi-region kubernetes cluster using wireguard which has one master to maintain and manage the workloads across different openstack regions and perform experiments to conclude its performance. We are about to test the request-response times of kubernetes clusters to come to an understanding of their network performance.

The motivation behind performing this thesis is to evaluate how wireguard performs in Kubernetes as well as to reduce the hassle of maintaining multiple kubernetes clusters.

Below are the Objectives of our thesis

- 1: To Identify the factors that influence the Performance of a Kubernetes cluster in different scenarios
- 2: To Analyze the performance of a Kubernetes cluster within a single shared Private network. This is sometimes referenced as **Scenario-1** in the next chapters.
- 3: To Analyze the performance of a Kubernetes cluster with worker nodes across different Projects(Projects in openstack) in single cloud region connected with wireguard vpn. This is sometimes referenced as **Scenario-2** in the next chapters.
- 4: To Analyze the performance of a Kubernetes cluster with worker nodes across multiple cloud regions and in different Projects and connected using wireguard vpn. This is sometimes referenced as **Scenario-3** in the next chapters.

1.2 Research questions

In this thesis, we answer the following research questions formulated based upon the aims and objects mentioned in the section 1.1.

RQ1: What are the most common factors that influence the performance of Kubernetes cluster in multi cloud environment?

In order to create a multi-region kubernetes cluster and analyze the performance we need to know the factors that effect a kubernetes cluster when it is in multi-region environment along with wireguard, so that we can come to a conclusion on what metrics to be compared.

This helps us identify the metrics that are to be compared in order to conclude the results of the experiment.

RQ2: What is the performance of the Kubernetes cluster with worker nodes at different openstack regions and tunneled using wireguard

when compared to the performance of kubernetes cluster within a single shared private network.

After the deployment of above scenarios 1 & 2, we compare them with metrics found using RQ1 and conclude on which of the scenarios cluster is performing better.

RQ3: What is the performance of the Kubernetes cluster with workers nodes at multiple openstack regions and multiple openstack projects, tunneled using wireguard when compared to the other two scenarios

After the deployment of above scenarios 1 & 2 & 3, we compare them with metrics found using RQ1 and conclude on which of the scenarios cluster is performing better(similar to what we did for Finding RQ2).

Chapter 2

Related Work

This section describes the related work of this research. However, many researchers are working on the performance of applications deployed in Kubernetes. There are quite a few studies or nearly zero research conducted on the topic kubernetes over wireguard. The reason for this is, wireguard is a relatively new VPN protocol. To get this research done, we took the help of quite a few studies.

In this paper [16], author states that containers are lightweight, and due to their simplicity, containerization is by far the best way to run an application. In this paper [21], authors concluded that calico, when compared to other container network interfaces, has higher performance, we referenced this paper and used calico as our container network interface. In this paper, [12] authors designed and developed a multi-metric load balancer that lets users customize the load balancer according to our cluster's metrics. In this paper, [23] authors studied performances of various load balancers provided by cloud providers and developed a portable load balancer that can be used in any environment. It also concludes that kubernetes ingress is a better and efficient way to expose and application in the private cloud environment. Thanks to this research, we have concluded using Kubernetes ingress resource and NodePort to load balance the deployed service.

This paper [5] elaborates on the usability of the protocol and also provided us a comparison of all the available vpn protocols and concluded that wireguard has the best performance overall. In this paper, [14] author has deployed applications along with their configurations within a VPC at a cloud Provider and standardized the way of configuration management using ansible playbooks, this paper helped us how better configuration management can be done using ansible. In this paper [22] authors described the network analogy of all the container network interfaces and compared them, from which we had an opportunity to choose which container network Interface to use. In this paper [24] author has compared several operating systems and virtualization environments where docker can be best used and concluded linux operating system is an efficient way to run docker.

In this paper, [5] authors described the importance of wireguard protocol, and as wireguard is a Kernel-based virtual network interface, It is researched and concluded that wire guard is best performed when used in Linux based systems. In this paper [17] authors from the canonical team have researched on various kernels and concluded that the change in kernel affects the performance of the system and also, this paper addresses the kernel comparison under different workloads that focuses on response time, jitter, system throughput, scheduling the overhead, and having a balanced operational efficiency. In this paper [20], the authors stated that deploying

an application in kubernetes makes it highly available and resilient. The author here compared the deployments and performance of applications in different environments using prometheus and grafana and developed a new, better framework for monitoring the deployed application.

In this chapter, the key concepts and terminology related with this research are presented.

3.1 Cloud Computing

Cloud computing is a model for providing on-demand access to computing resources at different cloud providers' data centers over the internet. The need to store, process, and analyze large amounts of data is driving many organizations and individuals to adopt cloud computing [7]. Lack of computing resources, storage, reduced capital costs raised the deployments of a large number of scientific experimental applications that are increasingly processing and producing data. Cloud computing provides a total cost of ownership. It allows companies and organizations to focus on their core business without worrying about the issues, such as infrastructure, flexibility, and availability of resources [7]. Moreover, the combination of rich computational resources and scalable infrastructure, data storage offers an exceptional environment where scientists can perform their experiments without the hassle of setting up their environment from scratch.

Types of cloud deployment models

- **Public cloud** - whole computing resources are provided by a cloud Provider via internet with a pay per use policy [8].
- **Private cloud** - Hosting all your computing infrastructure in a private network by yourself or by cloud provider and is not shared, The security and control level is highest while such network.
- **Hybrid cloud** - Using both private and public clouds, depending on their use-cases. You host your most important applications on your servers to keep them more secure and secondary applications elsewhere

types of cloud computing services

- **IaaS** - Infrastructure as a service (IaaS) is provisioning instant computing infrastructure which can be managed over the internet. It is the basic category of cloud computing services.

- **PaaS** - Platform-as-a-service (PaaS) is the supply of instant on-demand environment for developing, testing, deploying, and managing software applications. It is designed to let developers focus of developing the application without worrying about setting up or managing the underlying infrastructure of servers, storage, network, and databases needed for development [8].
- **SaaS** - Software-as-a-service(SaaS) It is a method that allows users to access a cloud based application through internet on subscription basis [8] .

3.2 Virtualization

Virtualization is a technology used to abstract computer resources. In simple words, it is the separation of service from the underlying physical hardware. Thus it allows multiple virtual servers centralized onto a single physical machine. There are numerous virtualization techniques, but they all use a hypervisor that runs on the host operating system to set up the virtualization standards. Mostly virtualization is used to create virtualized servers for the IaaS environment due to its maturity and flexibility. But, it also has disadvantages such as slow bootup time, low efficiency, and it takes up a lot of system resources [8].

Figure 3.1 shows the basic architecture of a virtual machine, where every I/O operation has to pass through all of its components for the service in the Virtual Machine to run properly.

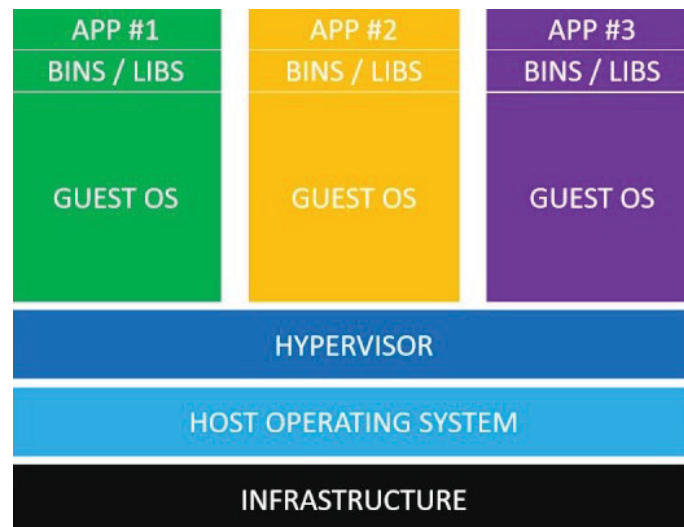


Figure 3.1: virtual machine architecture

3.3 Containers

A virtual machine can run an independent operating system through hardware virtualization, which requires an OS kernel, binaries, and related libraries as are necessary for executing applications for services [8]. Unlike Virtual Machines, Containers are less hardware-dependent than Virtual Machines. container shares the host's kernel

needed to run the application and only requires the program binaries and libraries. A container is a software package with code and all its dependencies that lets applications run quickly and reliably from one computing environment to another. A container image is a lightweight, standalone, executable software package that includes everything needed to run an application: code, runtime, system tools, system libraries, and settings.

Figure 3.2 represents the basic architecture of a container that does not depend on the host operating system, hence fast and lightweight.

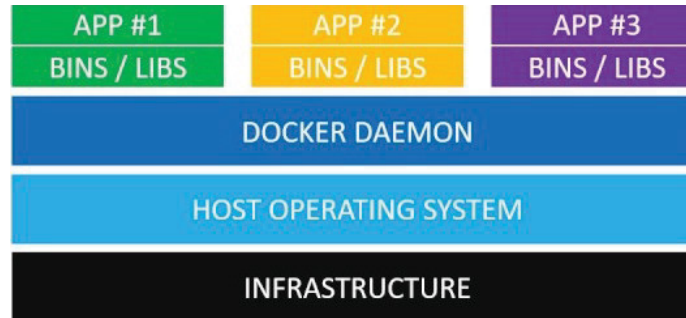


Figure 3.2: container architecture

3.4 OpenStack

Openstack is an open-source infrastructure-as-a-service(IaaS) deployment model which uses hypervisors and virtualization technology to manage several virtual servers at data centers through internet which makes horizontal scaling easy [11]. we used one such openstack cloud(city cloud)as an environment for the scenarios mentioned in 1.1.

3.5 Docker

Docker is a container runtime that is used to deploy applications in the form of containers. a container is a package with executable files and their dependencies to run an application quickly. Docker uses images to Spawn containers [6]. Multiple containers can be created using an image. Images can be saved in an online repository managed by docker, or we can make our docker registry [18]. In a container environment where applications are virtualized and executed, docker adds up an extra layer of deployment engine on top of it [16]. Docker is designed to give a quick and lightweight environment to quickly run containers and test them before they are put into production environment [16]. Docker allows the user to scale the container and move them into different hosts[3].

Docker uses dockerfile, a set of commands, and binaries in step-by-step docker rules to understand and create an image. Once the image is created, we can store it in a registry and distribute it to our Developing, Staging, and Deploying Environments.

3.5.1 Namespaces

Docker provides isolation using separate namespaces for containers. Namespaces enable creating an abstraction of a particular global resource and make it look like a separated instance to processes within a namespace. Consequently, several containers can use the same resource simultaneously without conflict.

3.5.2 Cgroups

Docker uses cgroups to group processes for system resource management, such as allocating CPU time, system memory, network bandwidth, or combinations of these among user-defined groups of tasks.

Although docker can create containers, whenever the containers are spun down, either due to application failure or due to network or central processing unit overload, the user has to spin up a new container manually. Due to the rise of containers and microservices, a container orchestration engine, a platform for container life cycle management such as container creation, deletion, and distribution, has been developed. The representative open-source software is kubernetes, docker swarm, etc. The main functions of these orchestration engines are scheduling, maintaining high availability support for container systems, health check of containers, distributed architecture, and load balancing for containers created in the cluster [16]. We are using one such container orchestration engine called kubernetes to deploy an application and conduct the experiment, as discussed in chapter 1.

3.6 Kubernetes

Kubernetes is a great standard for orchestrating container workloads. It enables containerization and deployment within the cloud infrastructure. It introduces the concept of pods, which is a group of one or more containers such as Docker containers within a shared network [3]. Kubernetes follows master/slave architecture. A Kubernetes cluster consists of master and worker nodes. Each pod runs on a worker node in a Kubernetes cluster [23].

Figure 3.3 is a Basic architecture of a Kubernetes cluster. Master node acts as a controller which consists of four components such as apiserver, scheduler, etcd, Controller manager. Apiserver provides an API(Application programming interface) for operations on various resources, Etcd acts as a default storage system where all the metadata and config files are stored, a scheduler is responsible for arranging pods in respective nodes, the controller manager takes the responsibility of executing various controllers and kubectl is a basic command-line tool for controlling Kubernetes clusters. Worker nodes are worker machines of Kubernetes cluster. It consists of kubelet, Kube proxy, pod. Kubelet is an agent that runs on each node and makes sure that containers are running in pods, Kube proxy, is a network proxy and responsible for handling Kubernetes service concept, where a pod is responsible for encapsulating one or more containers [3]. In our experiments, we use docker as a container engine, as docker containers, can be easily deployed in cloud-based environment [4].

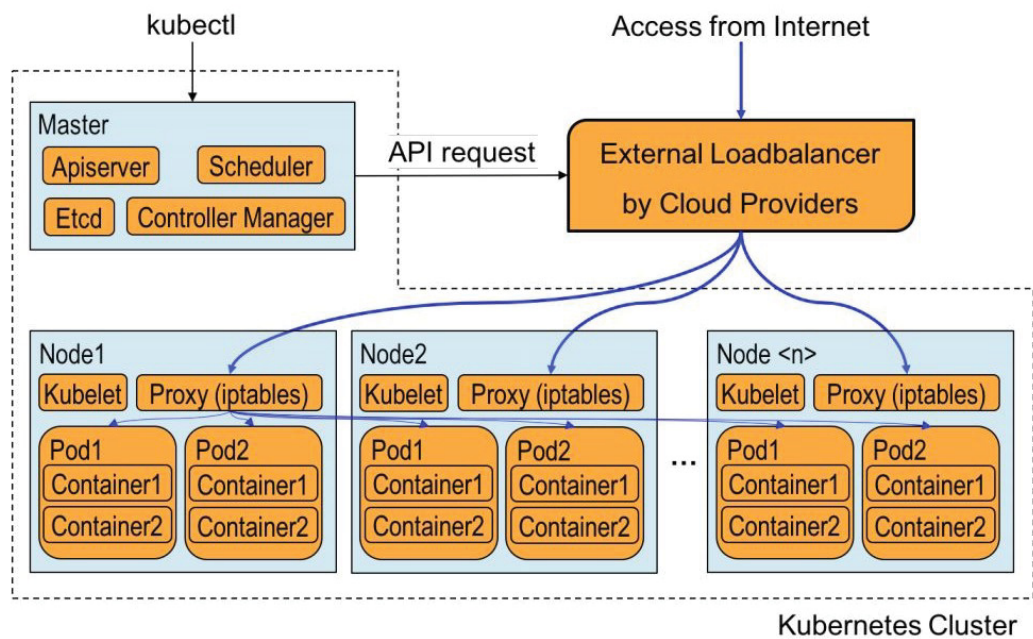


Figure 3.3: Architecture of a Kubernetes cluster [23]

3.6.1 Node

As mentioned above a node is a physical or a virtual machine that has all the components to manage pods. a node needs kubelet , kube proxy and docker as shown below 3.4

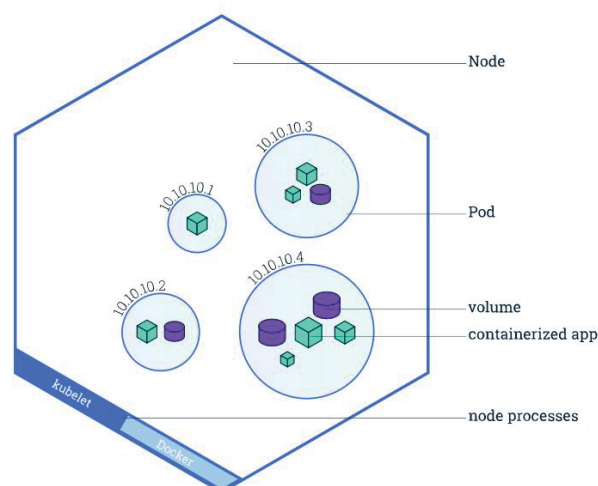


Figure 3.4: Node architecture

3.6.2 Pod

A pod is a collection of containers that run in the same network. Pod defines the run-time behavior of the containers and seen as a logical host on which the containers run, which lets containers in a pod share the same databases, which is an excellent place for deploying micro services [2].

3.6.3 Deployment

A deployment is used to deploy multiple pods. we can configure a deployment to contain replicas to ensure back-up instances in case the pod dies. In simple words, a kubernetes deployment ensures that a certain number of replicas are always available in a cluster, in case of pod death, it checks the current state(number of leftover pods) and tries to take it to the desired state(number of replicas configured in the deployment file). A deployment file also contains pod specs such as container image, volumes, ports, and other container specifications [1].

3.6.4 Daemonset

a daemonset in kubernetes is a configuration which runs exactly 1 pod in every kubernetes worker node. This is used to make sure the environment is stable and not two pods are deployed in one node, leaving another free. a daemonset ensures that each node is occupied.

3.6.5 Namespaces

Kubernetes also isolates system resources using namespaces as in docker. Kubernetes allows users to run deployments and services in different namespaces. A Kubernetes namespace enables users to divide cluster resources according to their requirements. A namespace is like a virtual cluster within a cluster.

3.6.6 Services

A Kubernetes service [13] provides gateway to access pods. A service in Kubernetes is assigned an IP address and the incoming traffic can be forwarded to as specified port of the pods defined in the service.

There are three types of Services

- **ClusterIP** - this is the default type, it exposes applications using clusters internal ip. Due to this, we can only reach the application from within the network.
- **NodePort** - NodePort exposes the service on each node's ipaddress at a static port. A clusterIP service is created automatically internally, and the nodeport will route to it. From outside the cluster, you can contact the NodePort service by using "<NodeIP>:<NodePort>".

- **LoadBalancer** - It exposes the service externally using the load balancer of your cloud provider. The external load balancer routes to your NodePort and ClusterIP services, which are created automatically.

Kimitoshi Takahashi [23] states that for a private network in openstack, the most beneficial way to expose a service is by using Kubernetes Ingress service and forward the traffic to NodePort.

3.6.7 Kubernetes Ingress

In Kubernetes, an Ingress is an API object that allows users to access your Kubernetes services from outside the Kubernetes cluster. You configure access by creating a set of rules that define which inbound connections reach which services. This lets you merge your routing rules into a single resource.

3.6.8 Container Network Interface(CNI)

Container Network interface is a combination of virtual network interfaces, bridges, and routing rules, allowing pods within a Kubernetes cluster(usually within the same namespace) to talk to each other.

There are multiple opensource projects for CNI plugins such as flannel, Calico, Weave, Canal. Picking an overlay network is a highly complex task. authors in this paper [10] confirm that every container network interface, when tested, added 0.2 ms of response time to the network. However, [21] Concludes that Calico has high performance and low latency compared to the canal and flannel.

Flannel uses an overlay network. An overlay network is a computer network layered on top of another network. This results in an extra latency, as the packet needs to pass through a different interface where it is fragmented and wrapped.

Calico

Calico is an open-source Container network interface project. Unlike Flannel and other CNI plugins, Calico does not use an overlay network. Instead, it configures a layer three network that uses border gateway protocol to route packets between hosts. This means that packets do not need to be wrapped in an extra layer of encapsulation when moving between hosts. The BGP routing mechanism can direct packets natively without extra wrapping packets in an additional layer of traffic, Thus reducing the latency.

Calico installs an agent called felix in all the nodes and a felix controller in the master node to handle interpod communication [22].

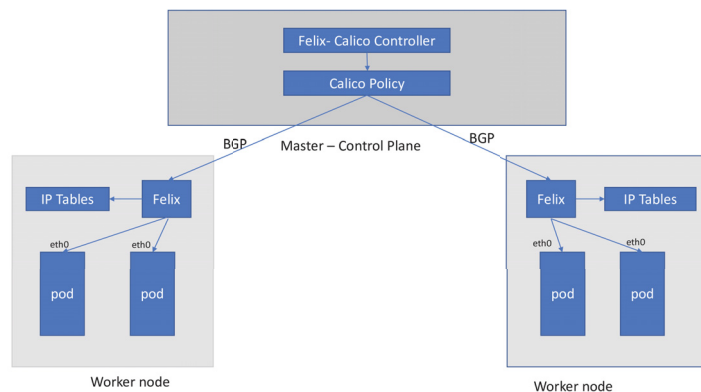


Figure 3.5: Inter pod communication

3.7 Wireguard

Wireguard is a secure network tunnel, operates at layer 3, and is implemented as a kernel virtual network interface for Linux. Compared to other network tunnel protocols, wireguard is more secure, more performant, and more comfortable to use. Wireguard interface is based on a fundamental principle of secure tunnels: an association between a peer public key and a tunnel source IP address. It uses a single round trip key exchange and provides strong perfect forward secrecy in addition to a Great deal of identity hiding. Transport speed in wireguard is accomplished using ChaCha20Poly1305 authenticated-encryption for encapsulation of packets in UDP. Improved IP-binding cookies help us mitigating DDoS attacks as well as a high degree of encryption and authentication. The overall design allows allocation of no resources in response to received packets, and from a systems perspective, there are multiple interesting Linux implementation techniques for queues and parallelism. Finally, wireguard is easy to implement in Linux due to its small 4000 lines of code, which makes it easy to audit and verify [5].

As wireguard can be implemented as a kernel virtual network interface, we are going to use this as host network between all the workers nodes and then install kubernetes within the tunneled nodes.

3.8 Rancher Kubernetes Engine(RKE)

Installing Kubernetes is a complicated task. We have to install and configure all the components in the cluster and configure certificates for authenticity. To simplify the installation of Kubernetes, various open-source Kubernetes distributions have been developed, such as Kubeadm, Kubespray, Kops, TKE (Tencent Kubernetes Engine), and RKE (Rancher Kubernetes Engine).

Out of all these kubernetes distributions, rancher kubernetes automates the creation of kubernetes using a simple YAML script. The cluster formed using RKE runs kuberenetes components as docker Containers, making it easier to deploy and maintain.

3.9 Ansible

Ansible is an agentless configuration management tool written in python and uses Playbooks written in YAML configuration. It uses SSH to connect to the hosts that are mentioned in the host configuration of Ansible. Ansible Works in Two Modes: ad-hoc mode and Playbooks. In Ad-hoc mode, Instructions to remote servers are passed as an argument, whereas in playbooks, they are written in YAML configuration. YAML is the abbreviation for Yet Another Markup Language. `/etc/ansible/hosts` is the file to describe the hosts you want to be configured using ansible [19]. We used ansible version 2.9 and python 3.7 for automation scripts.

4.1 Research Method

This research involves a systematic approach of gathering data, experimentation, and analysis to find the answers to the proposed research questions. The method applied is a “Hybrid model,” a combination of both quantitative and qualitative methods. A qualitative approach is used to answer research questions related to the factors that affect performance. A quantitative approach is used to answer the research questions related to the performance of Kubernetes as metrics. The knowledge gained and data gathered are used to form a robust Kubernetes cluster for experimentation and conclusion.

Literature review, case study, survey, and experiment are widely used research methods. A case study and survey are not suitable for our research because RQ1 requires peer-reviewed and controlled data. RQ2 and RQ3 need dependant variables to be measured.

A literature review is a process of gathering and accumulating existing data from relevant research papers to gain a deep understanding of the research area. A formal experiment is a process of manipulating independent variables and measure the dependent variables in a system.

From the above research questions, we can see that RQ1 can be answered by identifying factors affecting the performance. Hence, we used a literature review to gather and accumulate data from relevant papers. RQ2 and RQ3 can be answered using the formal experiment as they require measuring performance metrics and fall under quantitative research.

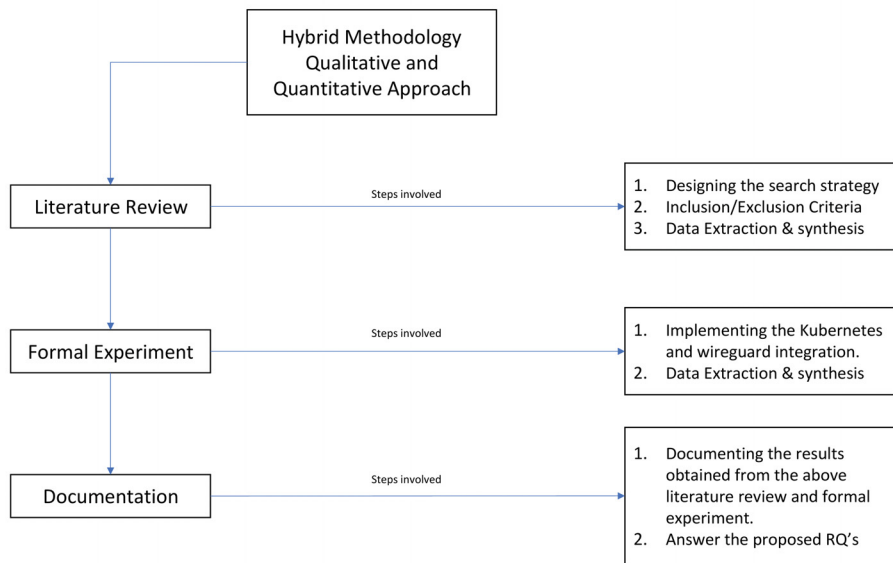


Figure 4.1: Research methodology

Below are the following methods depicted in Figure 4.1 that represent the research methodology of this thesis.

- Literature Review
- Formal experiment
- Documentation

4.2 Literature review

A literature review has been done to identify the factors that affect the performance of kubernetes with wireguard. A background study has also been done along with a literature review for a deep understanding of the concepts involved in this research. The answer to RQ1 is found using a Literature review. A series of steps such as search strategy, snowballing, inclusion-exclusion criteria are followed to filter the relevant papers.

4.2.1 Search strategy

This steps helps in finding and accumulating primary research papers. Results obtained from the literature review serve as base for our research. Some other relevant papers were obtained from snowballing. A search string was formulated to find the relevant research papers from the accesible databases.

("Performance analysis of kubernetes" OR "Performance evaluation of wireguard"
OR

```
"performance analysis of container network interface" OR "Kubernetes on  
openstack" OR  
"Kubernetes and wireguard" OR "network performance analysis of kubernetes  
cluster" OR "performance analysis of kubernetes network overlay") AND "Network  
performance of kubernetes with worker nodes in multiple openstack regions"
```

Keywords such as wireguard, docker, containers, virtual machines, container networking, kubernetes, cloud computing, performance, were also searched and gathered to get a good understanding of the experimental Factors.

4.2.2 Snowballing

As the research is fairly in the early stages, we used backward snowballing to search for more research papers from the references of the already found research papers to gain more insight.

4.2.3 Digital libraries

the following libraries were used to perform the search strategy

- ACM digital library
- IEEE
- Scopus
- Google scholar

4.2.4 Inclusion-Exclusion Criteria

This step is carried out to further refine our search to gather papers that address the research questions.

Inclusion Criteria

- Paper which are written in English.
- papers based on performance analysis of kubernetes
- papers based on kubernetes and cloud computing
- papers focusing on research questions

Exclusion Criteria

- Unpublished papers
- Papers that doesn't address the proposed research questions.
- Papers that doesn't focus on any of the keywords or search string

After this step, the abstracts and introduction of all papers found, are read and the papers that addressed our challenges and mitigations are selected.

The table 4.2.4 shows the papers that are filtered during the above processes and the total number of papers taken into consideration after the search strategy

Process	Results
Search & Snowballing	82 papers
Inclusion/Exclusion criteria	54 papers
Removed by title	38 papers
Reading abstract	24 papers
Total	24 papers

Table 4.1: papers filtered using search strategy

4.2.5 Data Extraction and analysis

This step focuses on extracting and gathering and analyzing the data from the gathered research papers. It is a Qualitative approach that provides us theoretical knowledge about the challenges we are dealing with. Also, this step provides the answer to our RQ1 i.e., the factors that effect the performance of kubernetes cluster in a multi-region deployment.

In this paper, data found from the resulted papers is accumulated and gathered for referncing during the experiment.

4.3 Experimentation

This is a quantitative approach. After identifying the factors and gaining knowledge on metrics that effect the performance of the Kubernetes cluster in a multi-region deployment through the literature review. We use those metrics to compare the multi region Kubernetes cluster with a regular Kubernetes cluster. This step gives us an outcome on the proposed RQ's 2& 3.

In order to achieve the objectives of the thesis we have divided the experiment into a set of tasks.

- Designing and setting up Kubernetes environment in openstack.
- Create a test container with suitable workload tests.
- Write deployment scripts for three scenarios using ansible.
- Deploy a single Kubernetes cluster in single cloud region within single shared networks using ansible.
- Deploy a single Kubernetes cluster with worker nodes across different projects in single cloud region connected with wireguard vpn.

- Deploy a single Kubernetes cluster with worker nodes across multiple cloud regions and in different projects and connected using wireguard vpn.
- Evaluate and compare the performance metrics Request-response time in each scenario.

After analysing and evaluating the results, graphs are plotted and conclusions are drawn from the results.

Chapter 5

Implementation of Experimental setup

This section deals with the process of implementation and the parameters considered while setting up the environment. This section consists of service architecture and hosting application in containers to perform tests and measure the performance metrics. In order to understand and compare the performance of the multi region kubernetes cluster, we took two other scenarios into consideration as explained in objectives and the environmental setup for all the three scenarios are elaborated below.

Performance metrics are affected by the system resources. To make sure that all the system resources are same for all the scenarios and nodes of all kubernetes clusters. Every cluster is made using master and worker nodes with the following specification.

System components	Server
operating system	Ubuntu 20.04.1 LTS (Focal Fossa)
RAM	8 GB
CPU	4 cores

To make sure every container in a pod gets equal amount of resources, the deployment file is configured to give exactly 1 cpu core to every container and only 3 containers are deployed in each kubernetes cluster, all of them are run as a daemonset, see section 3.6.4.

5.1 Modeling the Service Architecture

To deploy and analyze the performance of the services, an architecture model is required. In our scenario, we have chosen the client-server architecture, where the client is any user who sends requests, and the server is a load-balanced kubernetes node where the application is deployed. Figure A.5 shows the client-server model's architecture in our experimental setup.

Client/server works in a way where a client sends a request to the server over the internet or any network, it is then processed by the server, and the output is delivered to the client. A server can handle several clients simultaneously, and one a client can send requests to several servers at a time, each providing a different set of services [9].

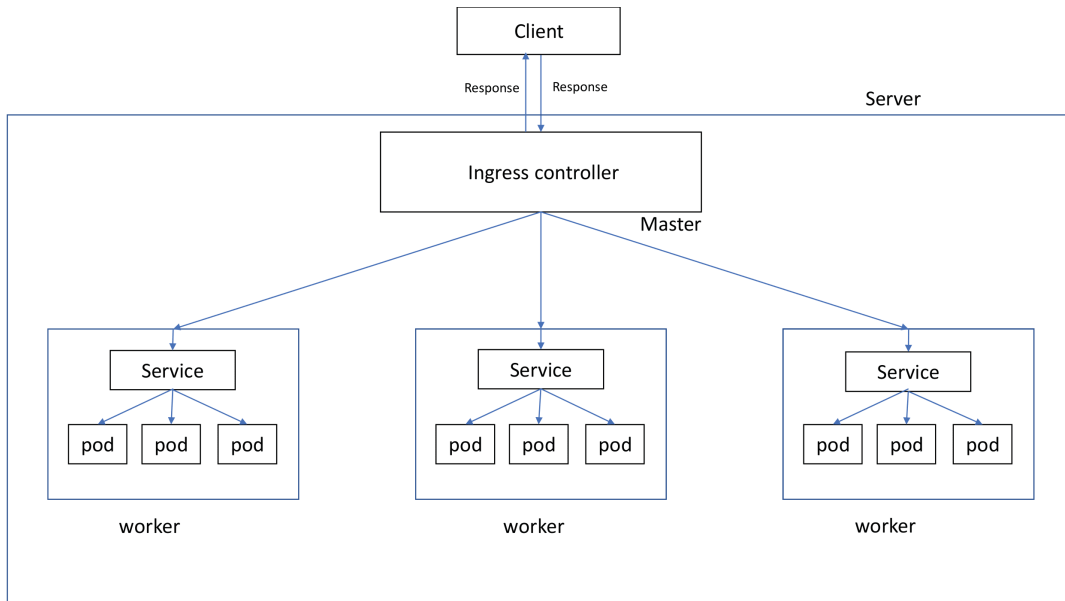


Figure 5.1: kubernetes client server model

5.2 Environment for Scenario 1

For scenario 1, as described in section 1.1 involves a kubernetes cluster with worker nodes in the same network. Therefore, we deployed a Kubernetes Cluster within a Single Project and single Region in openstack. To be specific, for scenario-1, we used karlskrona as the openstack region and all the virtual machines are located in karlskrona data center within same network and project. To create such a Kubernetes cluster, four instances were launched in openstack using our Host's SSH key for passwordless authentication along with Docker and RKE(rancher Kubernetes engine) installed in Each of them. A simple automation script in YAML Configuration is written with a preferred name of **cluster.yml**.

Once the kubernetes cluster is created, there is an output in the terminal as shown below A.1

and The Figure 5.2 shows a fully ready kubernetes cluster with master and worker nodes

```

root@k8s-s2-master:~# kubectl get nodes
NAME           STATUS    ROLES                                AGE     VERSION
10.0.1.1       Ready    controlplane,etcd,worker            4d5h    v1.17.6
10.0.1.2       Ready    worker                                4d5h    v1.17.6
10.0.1.3       Ready    worker                                4d5h    v1.17.6
10.0.1.4       Ready    worker                                4d5h    v1.17.6
  
```

Figure 5.2: Master and worker nodes and their assigned roles

Thus, we create a Kubernetes cluster for scenario 1. Creating a regular kubernetes cluster is done using rke binary and cluster automation file. But, creating a

kubernetes cluster within a tunneled network is an attempt made for research.

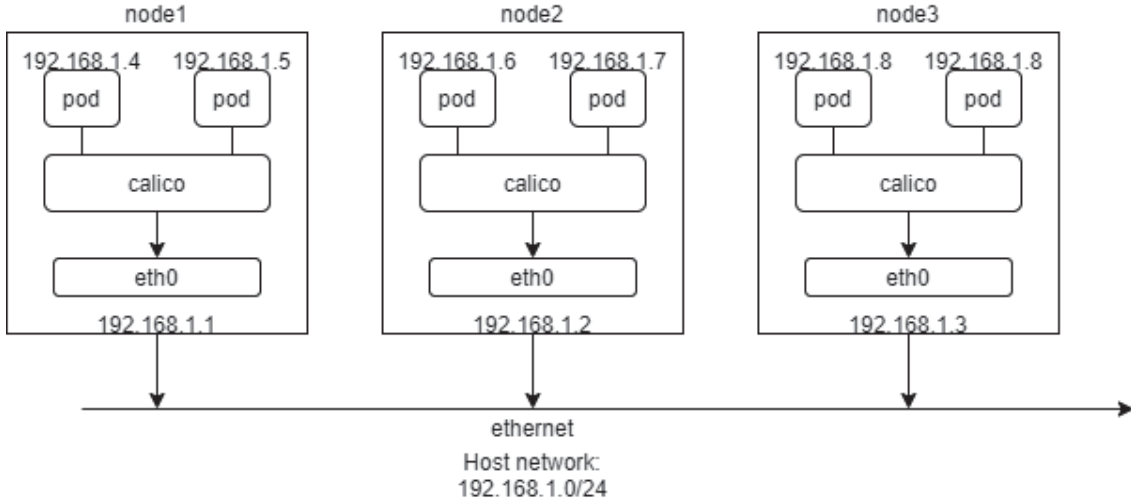


Figure 5.3: Network architecture of scene-1

Figure 5.3 is a basic architecture of the environment of scenario-1.

5.3 Environment For Scenario 2

Scenario2 is a deployment model as discussed in 1.1, a multi-project single region deployment, each project has a different network. In simple words, it has worker nodes in different networks, but they all are in the same OpenStack region. To be specific, we chose karlskrona as the openstack region but the virtual machines are in different openstack networks and projects. For this, We create 4 Instances, out of them, three are made sure to be present in three Projects(technically three networks) and the master can be in anyone of them. We apply floating IPs to all of them to SSH and configure the system using ansible and wireguard. Those instances are created using the host's SSH keys and are added To ansible Hosts to automate the configuration. wireguard is a virtual private network protocol that creates a virtual network interface(wg0) within the host and uses peer keys for data encryption and network tunneling by using any UDP port.

```
[Interface]
Address = {{ private_ip }}/24
PrivateKey = {{ private_key }}
postup= iptables
postdown= iptables
ListenPort = 51820

[Peer]
PublicKey = Public key of the peer
AllowedIps = allowed ips/32
Endpoint = public ip of the peer:51820
```

A basic wireguard configuration file Looks Like this 5.3 .

By using iptables see figure A.3, we can make wg0 the default ethernet port for data transfer during uptime of wireguard and once wireguard interface is down, eth0 will be the default port.

The above config file gives us a basic understanding of where Interface is for the host itself, and Peers are other instances that we want to tunnel. PostUp and PostDown are the ip tables for Interface. Postup Forwards the Inbound and Outbound traffic to Wg0 interface and PostDown is it's vice versa.

```

root@s2-master:~# wg
interface: wg0
  public key: aCC3dWJqs06ekIr4/1Js50G3+W7p2m201WnprgmglQc=
  private key: (hidden)
  listening port: 51820

peer: qZ24qRFq730EgzjHppncpCEhg0f79r2dM37zb+P13DE=
  endpoint: 188.95.226.5:51820
  allowed ips: 10.0.1.3/32
  latest handshake: 50 seconds ago
  transfer: 1.87 GiB received, 7.54 GiB sent

peer: 0e1o2zNgWwq9zqjeehtpqd/2Y6xzPnxNJwVv0GU/oRQ=
  endpoint: 188.95.226.238:51820
  allowed ips: 10.0.1.4/32
  latest handshake: 1 minute, 24 seconds ago
  transfer: 2.66 GiB received, 11.46 GiB sent

peer: 0/LKoYn8CFo0d0jF2yiloyamoDbJauI40fNf78xBohA=
  endpoint: 103.57.74.234:51820
  allowed ips: 10.0.1.2/32
  latest handshake: 1 minute, 55 seconds ago
  transfer: 2.21 GiB received, 9.22 GiB sent

```

Above text illustrates the output of wg interface and the tunneled peers. wireguard is running on port 51820 and we have created a subnet 10.0.0.1/32.

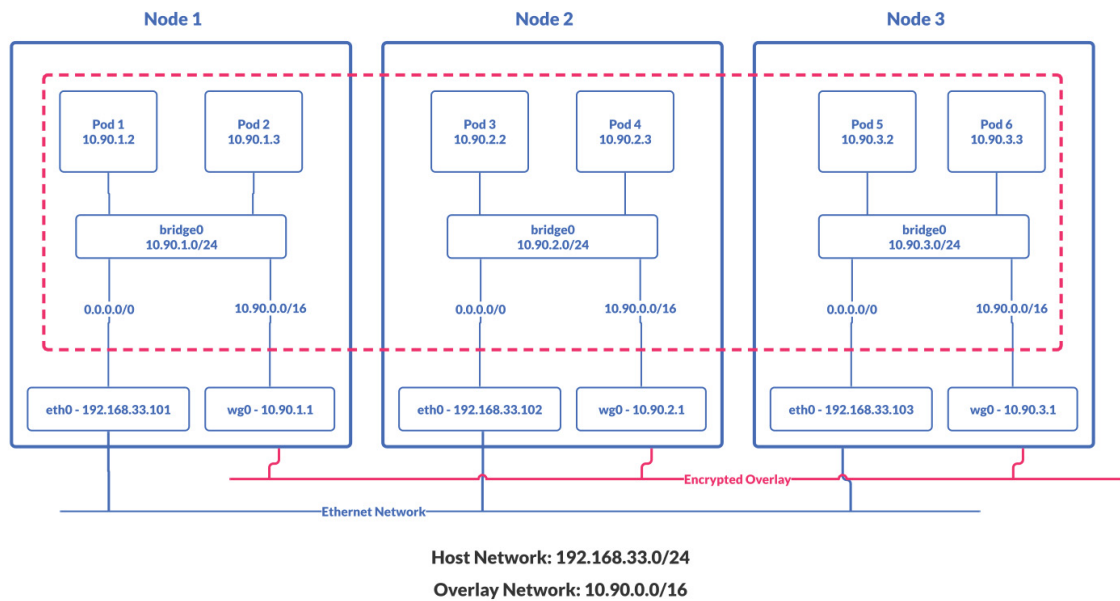


Figure 5.4: Wireguard network overlay

Figure 5.4 shows the CNI architecture of Kubernetes nodes. Installation of wireguard creates a new virtual network interface called wg0. Instead of ens3, wg0 becomes the network interface for calico traffic to pass through.

We Use ansible playbooks to automate the Above mentioned Setup in which, after installing wireguard, It uses RKE(Rancher Kubernetes Engine) and docker to create the kubernetes cluster within the tunneled network.

We use local inventory and the syntax is shown below

```
node-1 ansible_ssh_host=188.95.226.116
node-2 ansible_ssh_host=188.95.226.238
node-3 ansible_ssh_host=31.132.58.154
node-4 ansible_ssh_host=89.40.217.136
```

```
[wireguard:vars]
ansible_user=root
```

the above text illustrates the basic syntax of ansible hosts configuration. After adding the hosts, they are moved to inventory list once the SSH key is confirmed.

After the creation of cluster, we can see the output similar to what we saw in Scene 1.As, the default interface is wg0 calico uses wg0 as overlay network instead of ens3, to let nodes and pods communicate with each other.The next steps are similar to what we Did in Scenario 1. Thus, a testing environment for scenario 1 and 2 are created.

5.4 Environment For Scenario 3

Scenario 3 involves three openstack instances from three different projects and three openstack regions, i.e., all the three virtual machines are in three different networks.

We have created virtual machines in three regions(Karlskrona, Stockholm, Frankfurt), each in a separate project, making it a multi-region deployment. Once the virtual machines are created, we update the inventory of the ansible-playbook, see A.1 with the ip address of the workers nodes in different openstack regions and run the ansible playbook. Once the cluster is created the output is similar to 5.2.

After the environment for scenario3 is created, we need to deploy an application and expose it, and test the performance of the kubernetes cluster

5.4.1 Hosting the Application In Three Kubernetes Clusters

We deploy the container in a pod, which runs in a kubernetes cluster with its instances in openstack. A deployment file is written in YAML format and specify the container image, port, and which node it has to run on (if there are any specifications, we can label the node and point the deployment to the label, this procedure is called node affinity).

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web
spec:
  selector:
    matchLabels:
      app: web
  replicas: 3
  template:
    metadata:
      labels:
        app: web
    spec:
      containers:
      - name: blue
        image: pavangunda66/calc
        resources:
          requests:
            cpu: 1000m
            memory: 100Mi
```

the above script shows us the deployment file where cpu is given as 1000m which is 1000 millicores and equals to 1 core.

```

root@k8s-s2-master:~# kubectl describe deployments web1
Name: web1
Namespace: default
CreationTimestamp: Tue, 25 Aug 2020 16:07:23 +0000
Labels: <none>
Annotations: deployment.kubernetes.io/revision: 1
Selector: app=web1
Replicas: 4 desired | 4 updated | 4 total | 4 available | 0 unavailable
StrategyType: RollingUpdate
MinReadySeconds: 0
RollingUpdateStrategy: 25% max unavailable, 25% max surge
Pod Template:
  Labels: app=web1
  Containers:
    web1:
      Image: pavangunda66/calce
      Port: 80/TCP
      Host Port: 0/TCP
      Environment: <none>
      Mounts: <none>
      Volumes: <none>
Conditions:
  Type           Status  Reason
  ----           -
  Progressing    True    NewReplicaSetAvailable
  Available      True    MinimumReplicasAvailable
OldReplicaSets: <none>
NewReplicaSet: web1-69f644bc49 (4/4 replicas created)
Events:
  Type       Reason             Age   From                      Message
  ----       -
  Normal     ScalingReplicaSet  2m13s deployment-controller     Scaled up replica set web1-69f644bc49 to 1
  Normal     ScalingReplicaSet  102s  deployment-controller     Scaled up replica set web1-69f644bc49 to 4

```

Figure 5.5: Description of a deployment

Figure 5.5 describes the deployed application, It shows us the port where the container is exposed, volumes, strategy of the deployment which is Rolling Update (a process of updating an application serially).

5.4.2 Exposing The Deployed Application to Internet

We deploy applications in Kubernetes for its high availability and scalability and self healing nature. For user to access any application deployed in Kubernetes, It needs a kubernetes service (3.6.6) that exposes an application in kubernetes to the internet.

Exposing an application in Kubernetes is easier when managed kubernetes like EkS, AKS, etc., are used, as they have cloud controller manager installed as kubernetes component. If that is not the case, then exposing an application is quite challenging[23] as no cloud controller manager is running on our cluster and as open-stack octavia (LaaS) load balancer does not have native support to kubernetes , we choose to expose the application using NodePort and ingress as described in [23].

```

root@s2-master:~# kubectl get ingress
NAME           HOSTS      ADDRESS                                                                 PORTS   AGE
basic-ingress  *         10.0.1.1,10.0.1.2,10.0.1.3,10.0.1.4  80      40d

```

The above output illustrates that, using the YAML file, we have configured that any network request coming to any node ip, will be Load balanced with all of the worker nodes that has a pod with application running.

5.5 Data Collection

Once the environments for three scenarios are ready, and applications are running as deployments in Kubernetes clusters. We use the CURL tool's ability to calculate the total CURL response time of the application running in three clusters and compare them.

As mentioned in the 5.1, we use a client-server model. In our experiment, three kubernetes clusters, each having a master and an ingress controller load balancing the nodes. We test the request-response times of all three kubernetes clusters. We send 10000 curl requests to the master nodes of each cluster and take the average, min, max, standard deviation to conclude the average response times in 3 scenarios and evaluate the performance.

To avoid additional network related issues such as ISP bandwidth and latency of local computer, all the performance tests are done in a virtual machine in an isolated(one that does not include any of our experimental setups) openstack project and network and in openstack region karlskrona. All the kubernetes masters of three scenarios are located in different openstack projects and networks in openstack region karlskrona.

6.1

This section describes the main goal of the thesis i.e., performance evaluation of kubernetes cluster. Graphical representation of the experimental results is also shown in this section.

This section provides answers to the proposed research questions.

6.1.1 Performance evaluation of Kubernetes cluster with worker nodes in a single shared network in a single openstack region

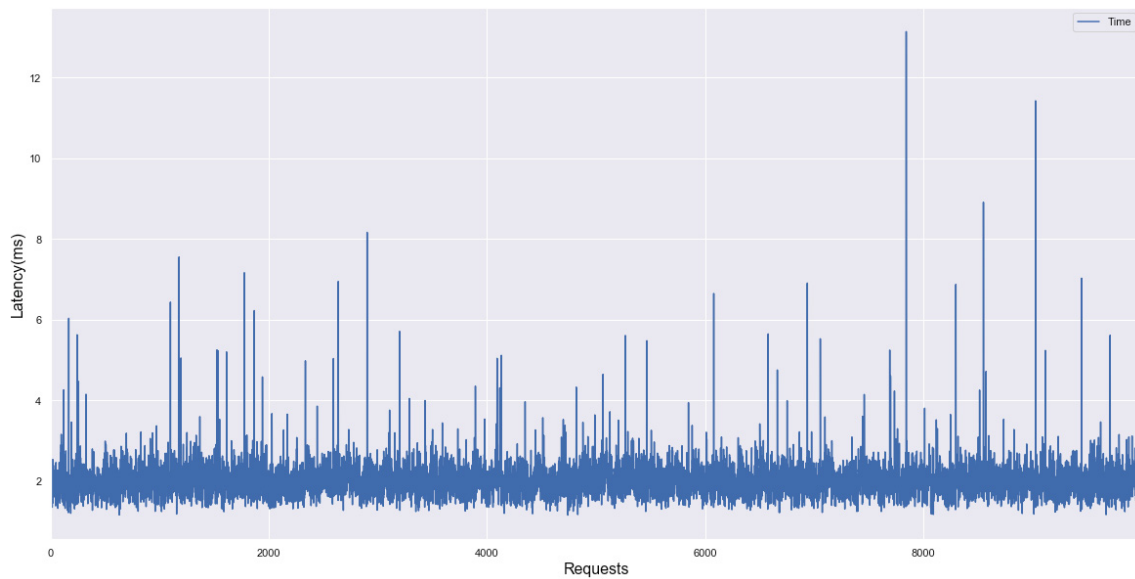


Figure 6.1: raw data of results from scenario-1

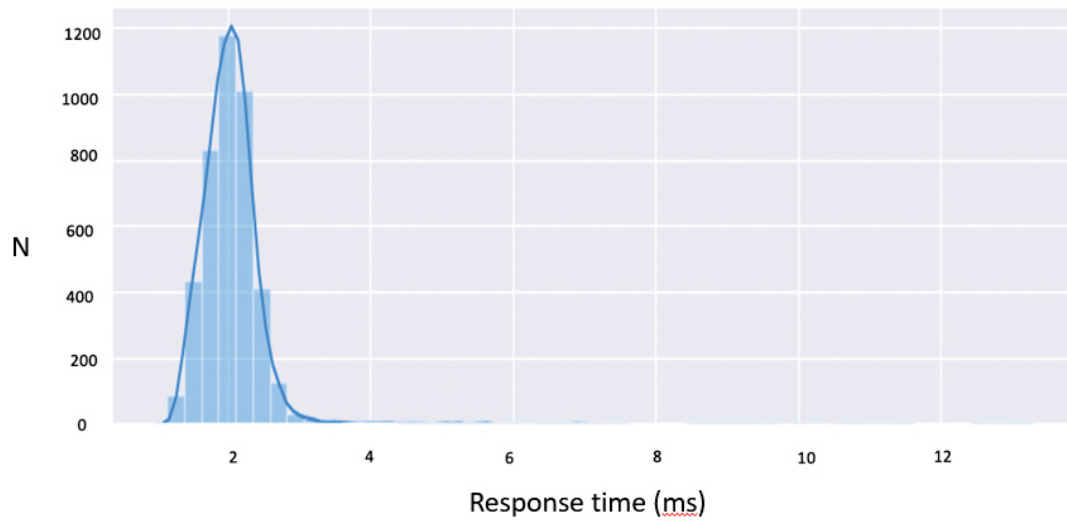


Figure 6.2: histogram plot of obtained data x-axis=response time in ms and y-axis=number of requests

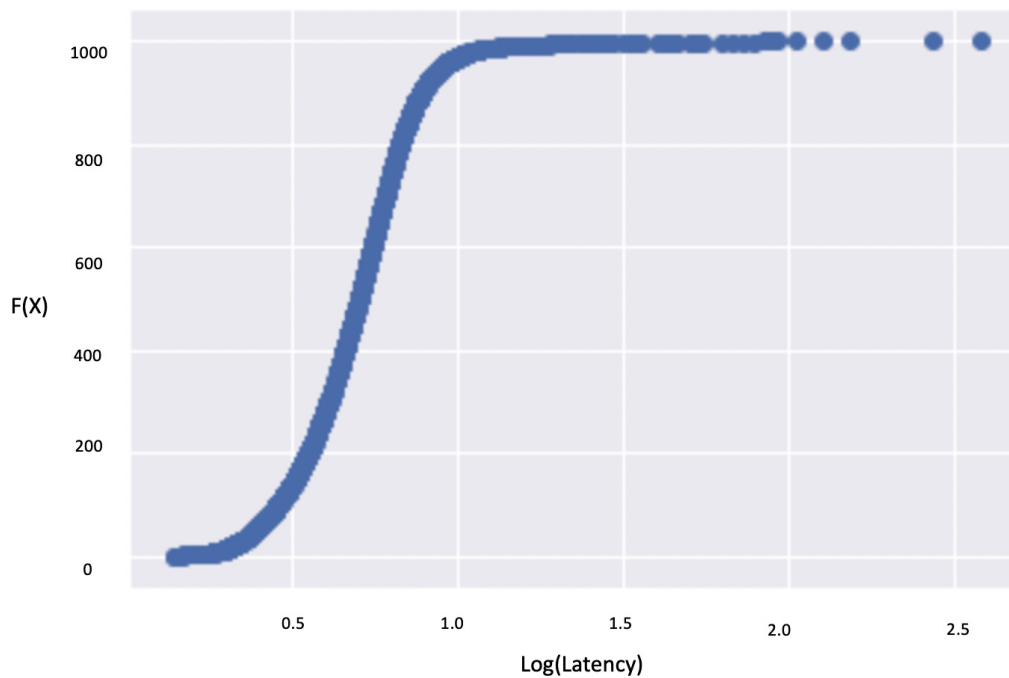


Figure 6.3: ecdf for $n=10000$ x-axis=logarithmic response time in ms and y-axis=number of request

Figure 6.1 shows the raw plot of 10000 requests vs their response time. From 6.2 we can obtain that, out of 10000 samples that are collected, 1200 data points have a value of nearly 2 milliseconds. The figure 6.2 is a histogram with bin-width of 0.3 milliseconds. In figure 6.3 the empirical cumulative distribution function is calculated and plotted. From the graph 6.3 we can observe that the function is nearly stable for a while at the range of 2 milliseconds i.e., 0.3 in log values. table 6.1 shows the

count	10000
average	2.0421
min	1.154
max	13.133
STD	0.442

Table 6.1: Statistical data

average value of the raw data which is 2.042 milliseconds along with min,max,std. The standard deviation shows us that how far each value lies with respect to mean i.e., in this scenario the deviation of each value with respect to mean is nearly 0.442 ms and the maximum value in all the requests sent is 13 ms, whereas the minimum value is 1.154. These state that all the values lie within 1 to 13 milliseconds range.

6.1.2 Performance evaluation of Kubernetes cluster with worker nodes in different openstack projects but in single openstack region and tunneled using wireguard

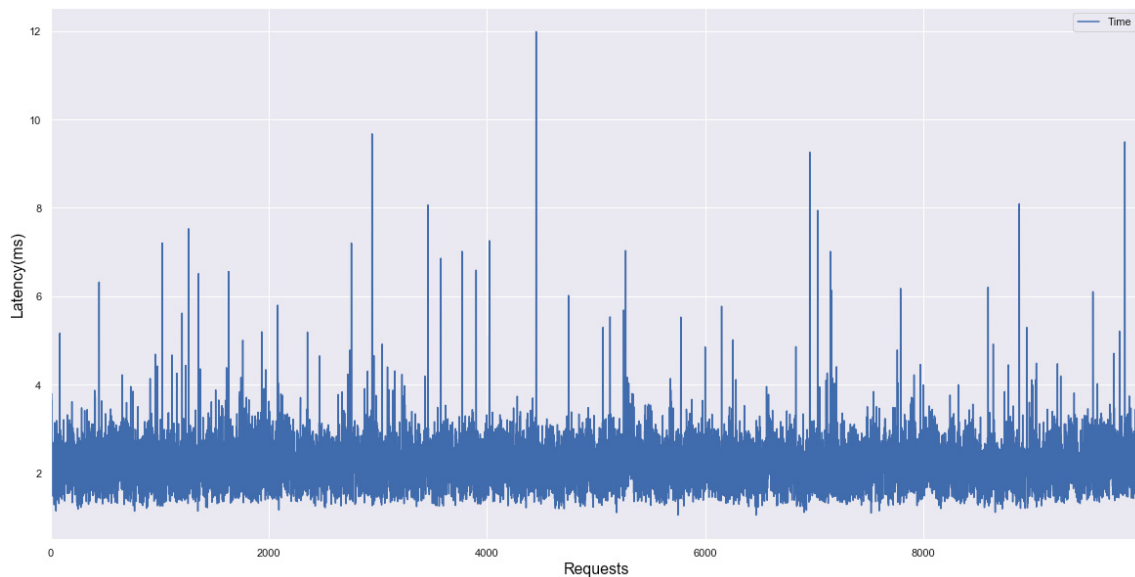


Figure 6.4: raw data of results from scenario-2

count	10000
average	2.3410
min	1.053
max	11.98
STD	0.602

Table 6.2: Statistical data

Figure 6.4 shows the raw plot of 10000 requests vs their response time .

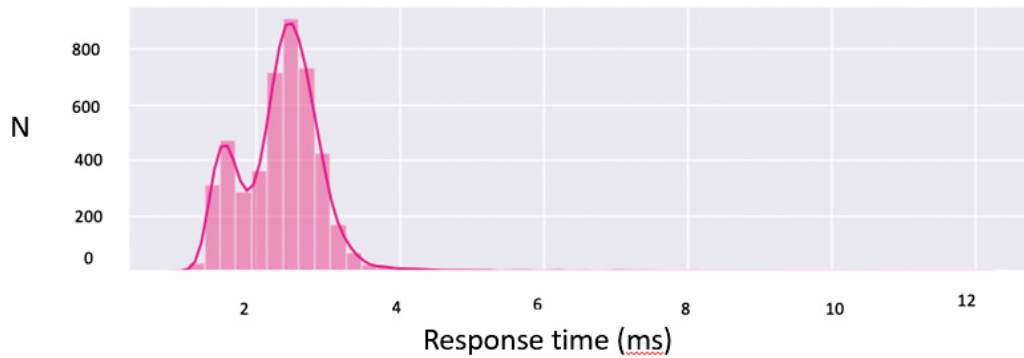


Figure 6.5: histogram plot of obtained data x-axis= response time in ms and y-axis=number of request

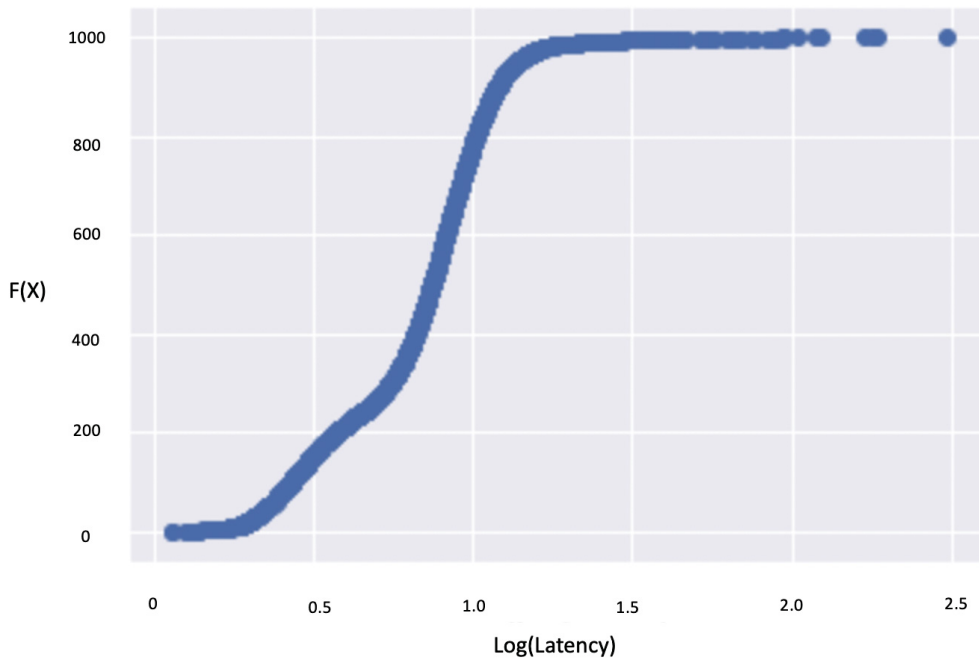


Figure 6.6: ecdf for $n=10000$ x-axis=logarithmic response time in ms and y-axis=number of request

Figure 6.5 represents that, out of 10000 samples y number fall under the range of x_1 to x_2 . x axis is the response time in milliseconds and y axis is the number of samples. From 6.5 we can understand that, out of 10000 samples that are collected, 800 data points have a value of nearly 2.3 milliseconds.

In figure 6.6 the empirical cumulative distribution function is calculated and plotted. From the graph 6.6 we can observe that the function is raising from 2 to 3 for a while at the range of 2 milliseconds i.e., at 0.3 log value in graph. table 6.2 shows the average value of the data which is 2.341 milliseconds along with min,max,

std. In this scenario the deviation of each value with respect to mean is nearly 0.602 ms and the maximum value in all the requests sent is 11 ms, whereas the minimum value is 1.154. These state that all the values lie within 1 to 11 milliseconds range.

6.1.3 Performance evaluation of Kubernetes cluster with worker nodes across multiple openstack projects and multiple openstack regions and tunneled using wireguard

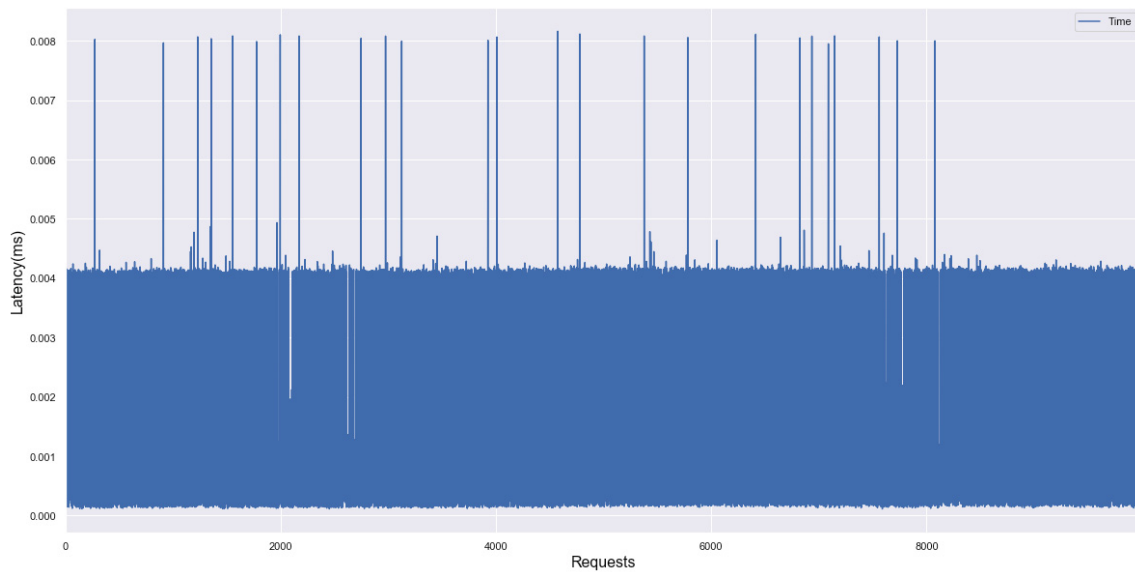


Figure 6.7: raw data of results from scenario-3

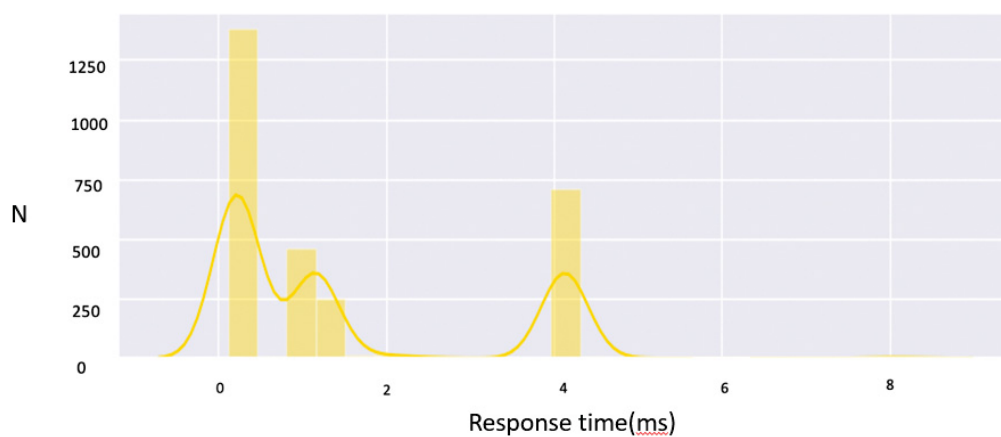


Figure 6.8: histogram plot of obtained data x-axis=logarithmic response time in ms and y-axis=number of request

Figure 6.7 shows the raw plot of 10000 requests vs their response time in ms. From 6.8 we can obtain that, out of 10000 samples that are collected, 1250 data points

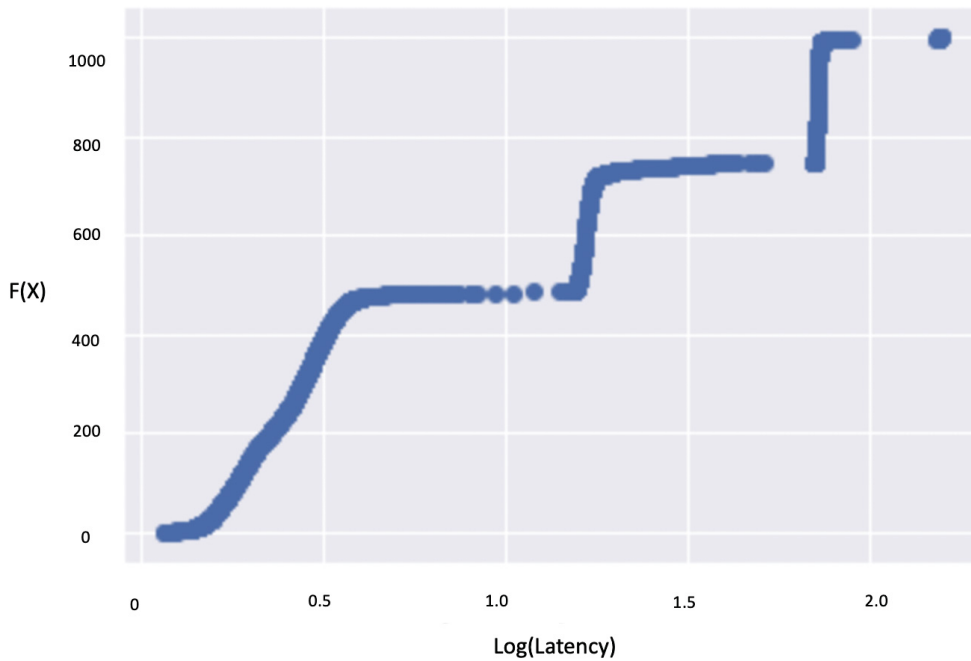


Figure 6.9: ecdf for $n=10000$ x-axis=logarithmic response time in ms and y-axis=number of request

count	10000
average	14.804
min	1.132
max	81.6
STD	16.353

Table 6.3: Statistical data

have a value in range of 0 to 20 milliseconds. In figure 6.9 the empirical cumulative distribution function is calculated and plotted. Table 6.3 shows the average value of the data which is 14.80 milliseconds along with min, max, std. The standard deviation shows us that how far each value lies with respect to mean i.e., in this scenario the deviation of each value with respect to mean is nearly 16.3 ms and the maximum value in all the requests sent is 81.6 ms, whereas the minimum value is 1.132. These state that all the values lie within 1 to 81.6 milliseconds range.

As we can see in figure A.5 , when a client sends an http request, it first passes through the master and goes to the readily available pod. All the masters of our kubernetes clusters are in the openstack region karlskrona. The virtual machine used to test the system's performance also resides in the openstack region karlskrona.

All the tests above are performed with an ubuntu instance in Karlskrona region for a stable experimental setup. However, the results vary when the tests are performed from other regions such as stockholm and frankfurt regions.

6.2 Research Questions and Answers

In this section, we discuss in detail the answers to our research questions from the knowledge and insight obtained from literature review and experiments.

RQ1: What are the most common factors that influence the performance of Kubernetes cluster in multi-region environment?

From the knowledge obtained from the literature review, response time, response time and reachability are the most common factors that influence the performance of kubernetes cluster. a vpn protocol can be influenced by level of encryption, physical location of the server, server overload. These factors helped in understanding which metrics should be compared and what parameters should be made stable in the environment [13].

As the factors affecting the performance are response time and reachability and response time, we decided to test the request-response time of the kubernetes cluster to conclude which scenario has better performance.

RQ2: What is the performance of the Kubernetes cluster with worker nodes at different openstack projects and in same openstack regions, tunneled using wireguard when compared to the performance of kubernetes cluster within a single shared private network.

From the above results see figure 6.1 and 6.4, we can come to an understanding that scenario-1 has a response time of 2.04 milliseconds and scenario-2 has 2.34 milliseconds of response time. Here, we can understand that worked nodes in different openstack projects and in same region, when tunneled using wireguard added an average response time of 0.3 milli seconds, i.e., the response time in kubernetes cluster with wireguard is more than that of the regular kubernetes cluster.

RQ3: What is the performance of the Kubernetes cluster with workers nodes at multiple openstack regions and multiple openstack projects, tunneled using wireguard when compared to scenario-1 and scenario-2

From the above results see figures 6.1, 6.4 and 6.7, we can understand that the average response time in three scenarios are 2.04 , 2.34 and 14.8. scenario 3 has reached a peak point of 80 milliseconds and standard deviation of 16 milliseconds i.e., when compared to scenario 1, scenario 3 has additional response time of 12.76 milli seconds and when compared to scenario 2, scenario 3 has additional response time of 12.56 milliseconds of response time. when worker nodes in different openstack regions and different openstack projects and different networks are tunneled, wireguard added an additional 12.76 milliseconds of response time along with encryption.

The results obtained from the research indicate that wireguard can be used to manage a multi-region kubernetes cluster. This helps in managing workloads in different networks and different openstack regions. When compared to other solutions such as kubefed, our solution is more efficient and with an acceptable response time. In this thesis, the system was set up in an openstack cloud provider deployment, the results may vary when virtual machines with higher resources are used (or) the change in cloud provider may affect the results, as they may have different latencies within their data centers. Though, we cannot state that these results are common in all cases such as high CPU VM's or different cloud providers, we can have a baseline concerning how good the wireguard performs when used in openstack kubernetes deployment.

One of the main contributions of the research work alongside identifying factors is the performance analysis and evaluation of the multi-region kubernetes cluster. This research gives a possibility of gaining knowledge on challenges while creating such kubernetes clusters and also the performance of wireguard in such kubernetes clusters. As this thesis is done in a real-time work environment i.e., on a cloud provider, this thesis also helps organizations to adapt to using this type of kubernetes environments to reduce hassle in their deployments. This thesis can also be referred in the future for gaining knowledge on performance of wireguard.

For testing the performance we chose request response time, we used a shell script see A.1.1, to automate sending the curl requests 10000 times. we used 10000 as a number to get the best average of requests in all the scenarios.

As mentioned in the implementation section, The regions considered for the environments of scenarios are Karlskrona, Frankfurt, Stockholm. Scenarios 1 & 2 are in Karlskrona region and the test systems are in Karlskrona region, therefore the response times on average are less. Whereas, scenario-3 has worker nodes in Karlskrona, Stockholm, Frankfurt and the response time are the averages of responses from all the three virtual machines from these regions, therefore we see a high response time. The lower the response time, the better the performance.

This chapter discusses the conclusions that are made from the answers of our research questions.

8.1 Conclusion

In this thesis, we tried to address the challenges and hassles faced during handling multiple kubernetes clusters, by configuring a single multi-region kubernetes cluster using wireguard as network overlay. we also tried to analyze the performance of the formed cluster and conclude whether the response times are acceptable or not.

To complete our objectives, we divided the environment into three scenarios and formulated 3 research questions. To answer RQ1 we used literature review. From the literature review, we can conclude that request-response time, wireguard encryption, geolocation of the server are the factors that influence the performance of kubernetes cluster in a multi-region deployment.

To answer RQ's 2 & 3, we have performed tests on the three scenarios. From results, we can see that the kubernetes cluster with worker nodes across multiple regions(scenario-3) has a response time of 14.8 milliseconds when compared to 2.04 milliseconds in regular kubernetes cluster(scenario-1) and 2.34 milli seconds in kubernetes cluster with worker nodes at multiple openstack projects and in same region(scenario-2). Having an average response time of 14 milli seconds is acceptable to use, as 14 ms is far less than 1 second.

From the experiment and the solutions to RQ's, as well as from the histogram 8.1, we can conclude that kubernetes cluster with worker nodes across different openstack regions performs with a response time that is acceptable and can be used for production setups by system administrators. From the results of scenario-2, we can see that wireguard adds a little or negligible amount of additional latency alongside encrypting the communication between the hosts. Therefore, we can also conclude wireguard can be used for encryption between hosts without affecting the latency of the system.

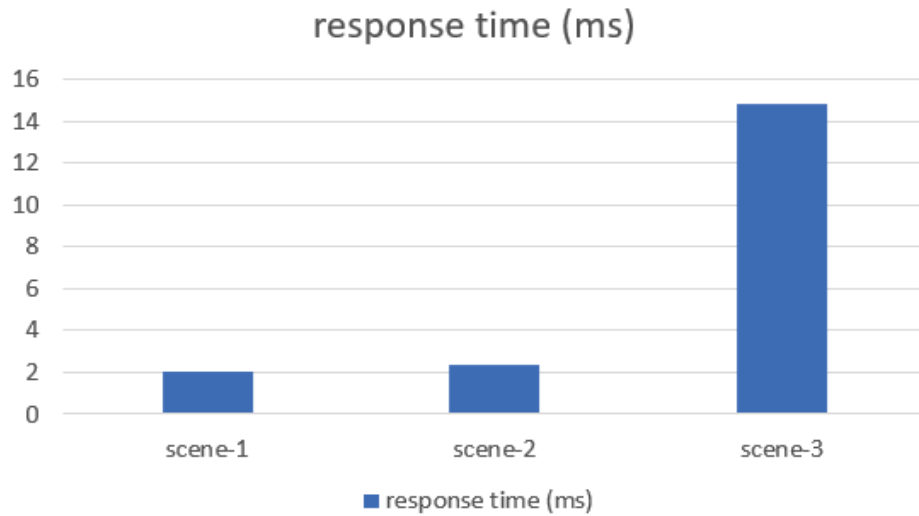


Figure 8.1: average values response times in three scenarios

8.2 Future Work

This section describes the extension of the current research that can be done using this research as a base. In our current scenario we have configured a multi-region kubernetes cluster with its worker nodes spread across different openstack regions and in different openstack projects.

we have multiple container network interfaces such as calico, flannel, weave. but they are not created for deploying kubernetes in multiple networks.

An exciting future work to this thesis can be, creating a container network interface plugin with wireguard that allows users to easily deploy kubernetes with worker nodes in multiple networks and multiple openstack regions. Manipulating the encryption of wireguard and analyze the performance of the kubernetes is also an exciting future work.

References

- [1] Deployment | kubernetes engine. [online]. available: <https://cloud.google.com/kubernetes-engine/docs/concepts/deployment> date visited: 12th september 2020.
- [2] Pods. [online].available:<https://kubernetes.io/docs/concepts/workloads/pods/pod/> date visited: 8th june 2020.
- [3] L. Abdollahi Vayghan, M. A. Saied, M. Toeroe, and F. Khendek. Deploying microservice based applications with kubernetes: Experiments and lessons learned. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 970–973, July 2018.
- [4] Babak Bashari.R, Harrison Bhatti, and Ahmadi. An introduction to docker and analysis of its performance. *IJCSNS International Journal of Computer Science and Network Security*, 173:8, 03 2017.
- [5] Jason A. Donenfeld. Wireguard: Next generation kernel network tunnel. In *NDSS*, 2017.
- [6] D. Géhberger, D. Balla, M. Maliosz, and C. Simon. Performance evaluation of low latency communication alternatives in a containerized cloud environment. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 9–16, July 2018.
- [7] Ibrahim Abaker Targio Hashem, Ibrar Yaqoob, Nor Badrul Anuar, Salimah Mokhtar, Abdullah Gani, and Samee Ullah Khan. The rise of “big data” on cloud computing: Review and open research issues. *Information Systems*, 47:98 – 115, 2015.
- [8] N. Jain and S. Choudhary. Overview of virtualization in cloud computing. In *2016 Symposium on Colossal Data Analysis and Networking (CDAN)*, pages 1–4, 2016.
- [9] Kai-Seung Siu and Hong-Yi Tzeng. On the latency in client/server networks. In *Proceedings of Fourth International Conference on Computer Communications and Networks - IC3N'95*, pages 88–91, 1995.
- [10] N. Kapočius. Performance studies of kubernetes network solutions. In *2020 IEEE Open Conference of Electrical, Electronic and Information Sciences (eS-tream)*, pages 1–6, 2020.

- [11] D. K. Konoor, R. Marathu, and P. Reddy. Secure openstack cloud with bandit. In *2016 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*, pages 178–181, 2016.
- [12] Q. Liu, E. Haihong, and M. Song. The design of multi-metric load balancer for kubernetes. In *2020 International Conference on Inventive Computation Technologies (ICICT)*, pages 1114–1117, 2020.
- [13] Steven Mackey, Ivan Mihov, Alex Nosenko, Francisco Vega, and Yuan Cheng. A performance comparison of wireguard and openvpn. 2020.
- [14] Nishant Kumar Singh, S. Thakur, H. Chaurasiya, and H. Nagdev. Automated provisioning of application in iaas cloud using ansible configuration management. In *2015 1st International Conference on Next Generation Computing Technologies (NGCT)*, pages 81–85, 2015.
- [15] M. Orzechowski, B. Balis, K. Pawlik, M. Pawlik, and M. Malawski. Transparent deployment of scientific workflows across clouds - kubernetes approach. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, pages 9–10, Dec 2018.
- [16] Y. Park, H. Yang, and Y. Kim. Performance analysis of cni (container networking interface) based container network. In *2018 International Conference on Information and Communication Technology Convergence (ICTC)*, pages 248–250, 2018.
- [17] J. Sahoo, S. Mohapatra, and R. Lath. Virtualization: A survey on concepts, taxonomy and associated security issues. In *2010 Second International Conference on Computer and Network Technology*, pages 222–226, 2010.
- [18] J. Shah and D. Dubaria. Building modern clouds: Using docker, kubernetes google cloud platform. In *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)*, pages 0184–0189, Jan 2019.
- [19] J. Shah, D. Dubaria, and J. Widhalm. A survey of devops tools for networking. In *2018 9th IEEE Annual Ubiquitous Computing, Electronics Mobile Communication Conference (UEMCON)*, pages 185–188, 2018.
- [20] N. Sukhija and E. Bautista. Towards a framework for monitoring and analyzing high performance computing environments using kubernetes and prometheus. In *2019 IEEE SmartWorld, Ubiquitous Intelligence Computing, Advanced Trusted Computing, Scalable Computing Communications, Cloud Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/S-CALCOM/UIC/ATC/CBDCoM/IOP/SCI)*, pages 257–262, 2019.
- [21] K. Suo, Y. Zhao, W. Chen, and J. Rao. An analysis and empirical study of container networks. In *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, pages 189–197, 2018.

- [22] K. Suo, Y. Zhao, W. Chen, and J. Rao. An analysis and empirical study of container networks. In *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, pages 189–197, 2018.
- [23] Kimitoshi Takahashi, Kento Aida, Tomoya Tanjo, and Hiroshi Sun. A portable load balancer for kubernetes cluster. pages 222–231, 01 2018.
- [24] X. Xie, P. Wang, and Q. Wang. The performance analysis of docker and rkt based on kubernetes. In *2017 13th International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery (ICNC-FSKD)*, pages 2137–2141, 2017.

Appendix A

Deploy environment for scenario-1 using cluster.yaml

```
cluster_name: rancher

nodes:
  - address: 10.0.1.1
    user: root
    role: [controlplane,worker,etcd]

  - address: 10.0.1.2
    user: root
    role: [worker]

  - address: 10.0.1.3
    user: root
    role: [worker]

  - address: 10.0.1.4
    user: root
    role: [worker]

services:
  etcd:
    snapshot: true
    creation: 6h
    retention: 24h

network:
  plugin: calico

ingress:
  provider: nginx
```

After the creation of the cluster.yml file, the command to create the kubernetes cluster is **rke up** while being in the Directory where RKE config file (cluster.yml) is present.

the above script gives us a basic idea of how the cluster automation configuration file for a Kubernetes cluster looks like. Nodes is the section where we have three arguments

```
INFO[0082] [addons] Executing deploy job..
INFO[0087] [ingress] ingress controller nginx is successfully deployed
INFO[0087] [addons] Setting up user addons
INFO[0087] [addons] Saving addon ConfigMap to Kubernetes
INFO[0087] [addons] Successfully Saved addon to Kubernetes ConfigMap: rke-user-addon
INFO[0087] [addons] Executing deploy job..
INFO[0092] [addons] User addons deployed successfully
INFO[0092] Finished building Kubernetes cluster successfully
```

Figure A.1: Successful cluster formation using RKE

- **address** - It is the External IP address of the Cluster to which the Host where Cluster.yml is going to Run has access to.
- **user** - the User to which our Host sends SSH request and Performs Operations
- **Roles** - Each node is Assigned a Kubernetes Role, Master has the roles: Control plane, etcd, Worker and the rest nodes have worker as their role.

Here is the Deployment File we used to deploy the application in our Cluster.

```
apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2
kind: Deployment
metadata:
  name: web
spec:
  selector:
    matchLabels:
      app: web
  replicas: 1
  template:
    metadata:
      labels:
        app: web
    spec:
      containers:
      - name: web
        image: pavangunda66/calculator
        ports:
        - name: http-server
          containerPort: 80
```

Figure A.2: deployment file

Figure A.2 illustrate a basic Deployment file used for deploying the Application, this works common in all the three Scenarios, as All of the Scenarios are using Same Kubernetes Version 1.17.6. Here, apiVersion is the version of api you want to use for the kind of Kubernetes object. Kind is the type of object we want to use. Metadata is the basic data of application, such as name etc, and the rest all of them are what its name suggests.

```
[Interface]
Address = {{ private_ip }}/24
PrivateKey = {{ private_key }}
PostUp = iptables
PostDown = iptables
ListenPort = 51820

[Peer]
PublicKey = Publickey of the peer
AllowedIps = Allowed ips/32
Endpoint = Publicip of the peer:51820
```

Figure A.3: configuration file for wireguard

below are its ip tables

- **PostUp** : iptables -t nat -A POSTROUTING -o ens3 -j MASQUERADE; iptables -t nat -I POSTROUTING 1 -s 10.0.1.0/24 -o ens3 -j MASQUERADE; iptables -I INPUT 1 -i wg0 -j ACCEPT; iptables -I FORWARD 1 -i ens3 -o wg0 -j ACCEPT; iptables -I FORWARD 1 -i wg0 -o ens3 -j ACCEPT; iptables -I INPUT 1 -i ens3 -p udp -dport 51820 -j ACCEPT
- **PostDown** : iptables -t nat -D POSTROUTING -o ens3 -j MASQUERADE; iptables -t nat -D POSTROUTING -s 10.0.1.0/24 -o ens3 -j MASQUERADE; iptables -D INPUT -i wg0 -j ACCEPT; iptables -D FORWARD -i ens3 -o wg0 -j ACCEPT; iptables -D FORWARD -i wg0 -o ens3 -j ACCEPT; iptables -D INPUT -i ens3 -p udp -dport 51820 -j ACCEPT

Public and Private Keys for wireguard Interface Can Be Created Using **CMD: wg genkey | tee privatekey | wg pubkey > publickey**

After Pasting the Config file is `/etc/wireguard`, We can now Start the Interface Using a Simple Command **wg-quick up wg0**. After starting the VPN Tunnel the interface Output Looks like Figure 5.4

```
apiVersion: v1
kind: Service
metadata:
  name: webapp
spec:
  type: NodePort
  selector:
    app: web
  ports:
    - port: 80
      targetPort: 80
```

Figure A.4: service file for NodePort

A.1 Ansible playbooks

Below is the ansible playbook used to create the environment of experiment

```
---
- hosts: wireguard
  any_errors_fatal: true
  roles:
    - { role: prepare }
    - { role: wireguard }

- hosts: node-1
  become: yes
  tasks:
    - name: rke install
      shell: wget https://github.com/rancher/rke/releases/download/v1.0.10/
            rke_linux-amd64

    - name: changing permission
      shell: chmod +x rke_linux-amd64

    - name: mv rke
      shell: sudo mv rke_linux-amd64 /usr/local/bin/rke

    - name: install kubect1
      shell: curl -LO
            https://storage.googleapis.com/kubernetes-release/release/'curl -s
            https://storage.googleapis.com/kubernetes-release/release/stable.txt'/'
```

```

bin/linux/amd64/kubectl

- name: mv kubectl
  shell: chmod +x ./kubectl

- name: install kubectl
  shell: sudo mv ./kubectl /usr/local/bin/kubectl

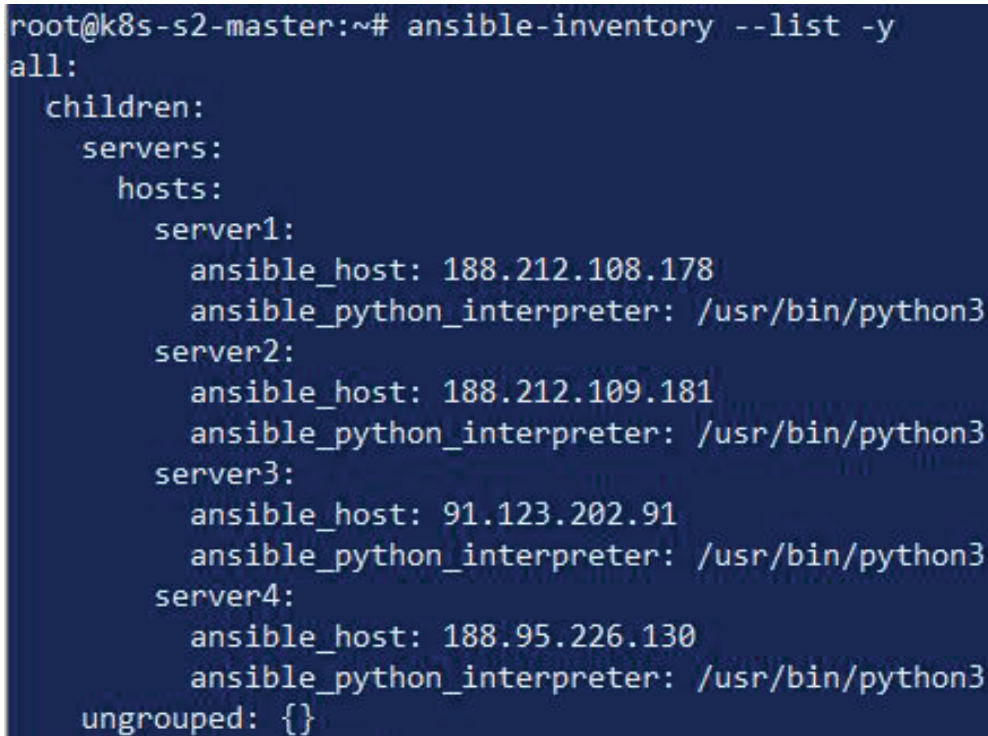
- name: rke up
  shell: rke up --config="cluster.yaml"

- name: rke up
  shell: mkdir ~/.kube

- name: mv config
  shell: mv kube_config_cluster.yaml config

- name: mv config file
  shell: mv config ~/.kube

```



```

root@k8s-s2-master:~# ansible-inventory --list -y
all:
  children:
    servers:
      hosts:
        server1:
          ansible_host: 188.212.108.178
          ansible_python_interpreter: /usr/bin/python3
        server2:
          ansible_host: 188.212.109.181
          ansible_python_interpreter: /usr/bin/python3
        server3:
          ansible_host: 91.123.202.91
          ansible_python_interpreter: /usr/bin/python3
        server4:
          ansible_host: 188.95.226.130
          ansible_python_interpreter: /usr/bin/python3
      ungrouped: {}

```

Figure A.5: Ansible inventory list

Figure 3.3 Ansible considers its hosts as its children and the parent as our host (where Ansible is running). Once the Ansible inventory list is filled we can run the command **ansible-playbook -i <name of the playbook>**, which starts off and ends when all the setup similar to scenario 2 and scenario 1 is configured.

Figure A.4 shows a simple YAML file that creates a Service with name "webapp" and type NodePort, we mention the app name in Selector and Port and Target port in Ports section, Targetport is port of the pod at which the application is running. Now, this creates a NodePort on all the nodes where the application is running.

```
root@k8s-s2-master:~# kubectl create -f dep.yaml
deployment.apps/blue created
root@k8s-s2-master:~# kubectl get deployments
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
blue	0/1	1	0	8s

Figure A.6: A Kubernetes deployment from command line

Figure A.6 shows output of **Kubectl get Deployments**, which in turn has pods running within the Same namespace.

```
root@k8s-s2-master:~# kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
web-77fd6b8995-84dgj	1/1	Running	0	6h21m
web-77fd6b8995-khqhg	1/1	Running	0	6h21m
web-77fd6b8995-192bz	1/1	Running	0	6h22m
web-77fd6b8995-mzxz2	1/1	Running	0	6h21m

Figure A.7: Pods formed for a deployment

Figure A.7 Shows us there are Pods In the same Namespace where the deployment is configured, Now we expose those Pods to The internet using a service as mentioned above.

```
root@k8s-s2-master:~# kubectl apply -f d.yml
service/webapp created
root@k8s-s2-master:~# kubectl get svc
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
kubernetes	ClusterIP	10.43.0.1	<none>	443/TCP	5d9h
web	NodePort	10.43.76.43	<none>	80:30160/TCP	11h

Figure A.8: Output of Running Services

Now, we Loadbalanced the created NodePorts using Ingress service of Kubernetes as described in 3.

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: basic-ingress
spec:
  backend:
    serviceName: web
    servicePort: 80
```

Figure A.9: Kubernetes ingress resource sample yaml file

Figure A.9 shows us a basic Ingress file that points any incoming traffic to all nodes that are running in the Kubernetes cluster.

A.1.1 Bash script used for request response time

```
#!/bin/bash

main () {
  local url=$1

  if [[ -z "$url" ]]; then
    echo "ERROR:
    An URL must be provided.

    Usage: check-res <url>

  Aborting.
  "
    exit 1
  fi

  print_header
  echo "REMOVING METRICS LOG FILE"
  rm -rf metrics.log
  echo "REMOVING ERRORS LOG FILE"
  rm -rf errors.log
  echo "RESPONSE_CODE | TIME_TOTAL(s) | POD_ID" >> metrics.log
  echo "RESPONSE_CODE | TIME_TOTAL(s) | POD_ID" >> errors.log
  for i in `seq 1 10000`; do
```

```

        make_request $url
done
rm -rf output.log
}

print_header () {
    echo "code,time_total"
}

make_request () {
    local url=$1
    http_status=$(curl --write-out %{http_code} --silent --output output.log $url)

    if [[ "$http_status" -ne 200 ]] ; then
        curl \
            --write-out "%{http_code} | %{time_total} | " \
            --silent \
            --output output.log \
            >> errors.log \
            "$url"
    else
        curl \
            --write-out "%{http_code} | %{time_total} | " \
            --silent \
            --output output.log \
            >> metrics.log \
            "$url"

        curl --silent $url | grep -o 'web[^;]*' >> metrics.log
    fi
}

main "$@"

```

