FACULTY OF COMPUTER SCIENCE AND INFORMATION TECHNOLOGY

UNIVERSITY OF MALAYA

WIA3003

ACADEMIC PROJECT II

AUTO SCALING AND LOAD BALANCING WITH HIGH AVAILABILITY SERVER USING KUBERNETES

CHAI YIOW YANG

COMPUTER SYSTEM AND NETWORK

WIC170016 / 17112536/1

SEMESTER 1, 2020/2021

SUPERVISOR: ASSOC. PROF DR ANG TAN FONG

ABSTRACT

The proposal describes on the features of Kubernetes along with load balancing and high availability features. The aim of this project is to implement auto scaling function HPA with metrics statistics, deploy a bare metal software load balancer that distributed the load equally among the pods and to develop an application that simulate Kubernetes failures by removing pods and node at certain time. As the application grows, so do the codebase. All the modules are maintained in the same codebase and are dependent to each other. It gradually becomes troublesome to manage as everyone commits in the same code. A minimal change in one module might affect other modules and may cause application deployment failure. This is where microservices come in to handle the dependencies of modules.

The project application uses Python programming language to call Application Programming Interface (API) from Kubernetes. The API called will be used to obtain the features from Kubernetes such as getting metrics of pods, nodes, scaling pods, deleting pods and many more. The application will be deployed into Kubernetes from Google Kubernetes. The accomplishment of this project is to expect to develop a server that has auto scaling and load balancing with high availability feature.

1.0 INTRODUCTION

Container

A container is a standard unit of software that packages up code and all its dependencies, so the application runs quickly and reliably from one computing environment to another.

Instead of virtualizing the hardware stack as with the virtual machines approach, containers virtualize at the operating system level, with multiple containers running atop the OS kernel directly. This means that containers are far more lightweight: they share the OS kernel, start much faster, and use a fraction of the memory compared to booting an entire OS.

There are many container formats available. Docker is a popular, open-source container format that is supported on Google Cloud Platform and by Google Kubernetes Engine.

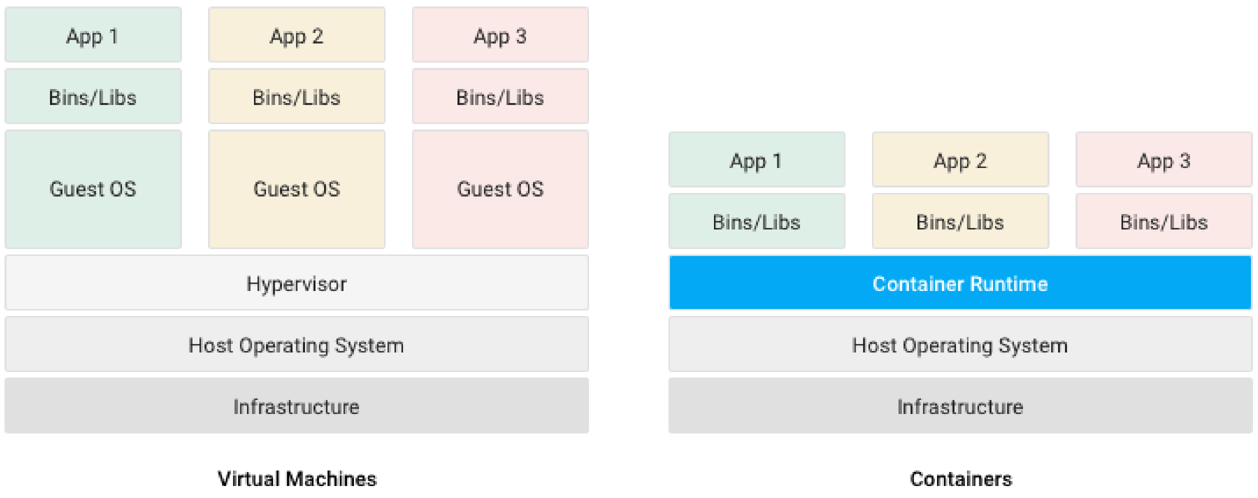


Figure 1: Comparison of abstractions of Virtual Machines and Containers

Docker

docker

Docker is a set of platform as a service (PaaS) products that uses OS-level virtualization to deliver software in packages called containers. Containers are isolated from one another and bundle their own software, libraries and configuration files. They can communicate with each other through well-defined channels. All containers are run by a single operating system kernel and therefore use fewer resources than virtual machines.

A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings.

A sample of simple PHP web server will be used in the testing of the Kubernetes features.

Kubernetes

kubernetes

Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications. It was originally designed by Google and is now maintained by the Cloud Native Computing Foundation. It aims to provide a "platform for automating deployment, scaling, and operations of application containers across clusters of hosts".

It has several features such as service discovery, load balancing, service topology, storage orchestration, self-healing, automated rollouts and rollbacks, secret and configuration management, batch execution, horizontal scaling and many more.

It can be used to run wide of container tools such as Docker.

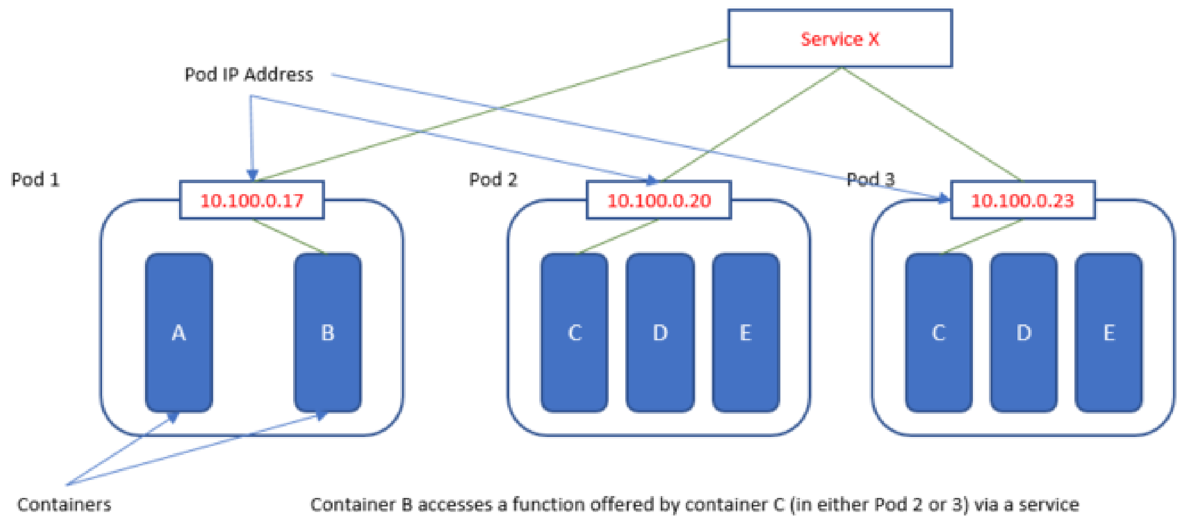

Service X
Pod IP Address
Pod 1
Pod 2
Pod 3
10.100.0.17
10.100.0.20
10.100.0.23
A
B
C
D
E
C
D
E
Containers
Container B accesses a function offered by container C (in either Pod 2 or 3) via a service

Figure 2 Kubernetes Architecture

Kubernetes has several objects, such as pods, replica sets, services, volumes, namespaces, ConfigMaps and Secrets, StatefulSets and DaemonSets. Pods are basically a containerized application that deploys in a Node.

This project will look into the features of Kubernetes, an open-source system for automating deployment, scaling, and management of containerized applications.

The features in Kubernetes such as high availability and high scalability of pods (containerized application). API of Kubernetes are provided to give flexibility to developers on leveraging the benefits from the application itself.

In order to maintain no downtime of the server, Kubernetes is used to implement a server that is reliable and always available to users.

High Availability

High availability ensures the application that users will use is always available. When the primary copy fails, another backup copy will takeover the role so the services can still serve to users.

Kubernetes High-Availability is about setting up Kubernetes, along with its supporting components in a way that there is no single point of failure. A single master cluster can easily fail, while a multi-master cluster uses multiple master nodes, each of which has access to same worker nodes. In a single master cluster, the important component like API server, controller manager lies only on the single master node and if it fails you cannot create more services, pods etc. However, in case of Kubernetes HA environment, these important components are replicated on multiple masters (usually three masters) and if any of the masters fail, the other masters keep the cluster up and running. There are many ways to setting up high availability depending on usage type.

Load Balancer

Kubernetes is the container orchestration system of choice for many enterprise deployments. That is a tribute to its reliability, flexibility, and broad range of features.

Load balancing is the process of efficiently distributing network traffic among multiple backend services and is a critical strategy for maximizing scalability and availability. In Kubernetes, there are a variety of choices for load balancing external traffic to pods, each with different tradeoffs.

The most basic type of load balancing in Kubernetes is actually load distribution, which is easy to implement at the dispatch level. Kubernetes uses two methods of load distribution, both of them operating through a feature called kube-proxy, which manages the virtual IPs used by services.

The default mode for kube-proxy is called iptables, which allows fairly sophisticated rule-based IP management. The native method for load distribution in iptables mode is random selection—an incoming request goes to a randomly chosen pod within a service. The older kube-proxy mode is userspace, which uses round-robin load distribution, allocating the next available pod on an IP list, then rotating the list.

However, neither of these methods is really load balancing. For true load balancing, the most popular, and in many ways, the most flexible method is Ingress, which operates by means of a controller in a specialized Kubernetes pod. The controller includes an Ingress resource—a set of rules governing traffic—and a daemon which applies those rules. The controller has its own built-in features for load balancing, with some reasonably sophisticated capabilities. In cloud environments such as Amazon Web Services, Google Cloud Platform or Microsoft Azure, load balancer is prepared by providers so users do not need tedious configurations. These are called external load balancer.

Load balancing can be applied in many ways, including in cloud or on premises data center. User can choose to use software load balancer such as NGINX, HAProxy, Traefik and many others. If administrator demand a LoadBalancer IP in Kubernetes service, they can opt for MetalLB which deploys inside Kubernetes cluster to act like external load balancer in cloud.

In this case, external software load balancer will be used, which is HAProxy Load Balancer.

Load balancing application will be integrated into a Kubernetes cluster.

1.1 PROBLEM STATEMENT

The problem statements where:

- HA mechanisms offered by Kubernetes while setting its monitoring intervals to their minimum values. It is not recommended and may lead to false node failure detection.
- It is daunting task to manually manage Kubernetes federated container clusters across all the service areas or to maintain the entire topology of cloud applications at a glance. Scalability is tedious.
- The Kubernetes does not include a load balancer and is heavily dependent on external load balancers that are set up on the fly by cloud providers through their application protocol interfaces (APIs).

1.2 OBJECTIVES

The objectives in this project is:

- To implement auto scaling function HPA with metrics statistics.
- To deploy and improve a bare metal software load balancer that distributed the load equally among the nodes.
- To develop an application that simulate Kubernetes failures by removing pods and node at certain time.

1.3 SCOPE

The project will be limited to load balancing and high availability function. Security and Persistent Storage are outside of the scope.

- Programming languages
  - Python – to write and call API from Kubernetes and create an application that simulates server failure
- Development tools
  - Kubernetes – to run Docker image and manage Kubernetes features

2.0 LITERATURE REVIEW

RESEARCH PAPER 1

L. Abdollahi Vayghan, M. A. Saied, M. Toeroe and F. Khendek, "Deploying Microservice Based Applications with Kubernetes: Experiments and Lessons Learned," 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), San Francisco, CA, 2018, pp. 970-973.

The paper discusses on the high availability (HA) features in Kubernetes which enables healing through its failure recovery actions. The feature reacts reasonably well to failure.

The goal of the paper is to devise architectures to enable high availability with Kubernetes for microservice based applications. Therefore, the researchers perform a quantitative evaluation of the availability achievable through Kubernetes under its default configuration. For this purpose, they have conducted experiments to measure availability metrics in different failure scenarios.

The metrics they used to evaluate Kubernetes from availability perspective are defined hereafter.

1. Reaction Time: The time between the failure event they introduce and the first reaction of Kubernetes that reflects the failure event was detected.
2. Repair Time: The time between the first reaction of Kubernetes and the repair of the failed pod.
3. Recovery Time: The time between the first reaction of Kubernetes and when the service is available again.
4. Outage Time: The duration in which the service was not available. It represents the sum of the reaction time and the recovery time.

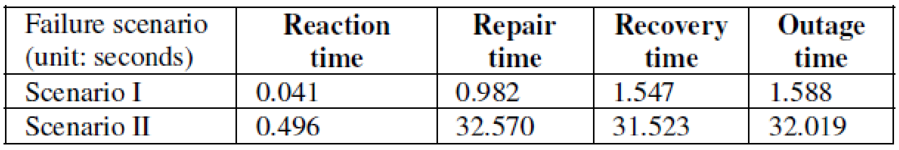
The architecture used in the experiments consist of:

- A cluster is composed of three VMs running on an OpenStack cloud, which is Ubuntu 16.04 OS on all VMs
- Docker 17.09 is running as the container engine.
- Kubernetes 1.8.2 is running on all nodes.
- Network Time Protocol (NTP) is used for time synchronization between nodes in the cluster.

- Microservices VLC video streaming.
- Pod template provided to the deployment controller contains
  - the container image of the streaming server, which once deployed will stream from a file.
- Condition where
  - the desired number of pods that the deployment controller needs to maintain is one.

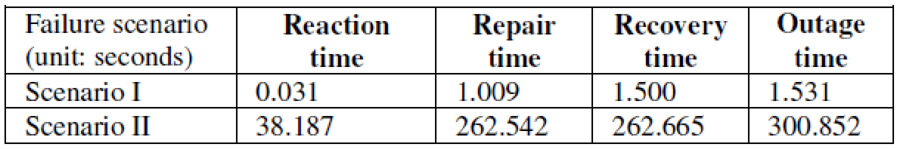
Two sets of failure scenarios will be demonstrated. In the first set, an application failure is due to a pod failure whereas in the second set it is due to a node failure. In each set, researchers distinguish between two failure scenarios. Scenario I designate a failure simulated by an administrative operation internal to Kubernetes while Scenario II is simulated by a trigger external to Kubernetes.

The results of the experiments for these failure scenarios are presented in Table I and Table II. Each scenario in the experiments has been repeated 10 times. The results presented in the tables are the averages of the 10 measurements. All measurements throughout the paper are reported in seconds.



Failure scenario (unit: seconds)	**Reaction time**	**Repair time**	**Recovery time**	**Outage time**
Scenario I	0.041	0.982	1.547	1.588
Scenario II	0.496	32.570	31.523	32.019

Table 1: Pod Failure where Scenario I is due to administrative pod termination by delete a pod while Scenario II due to pod process failure by killing the process from OS



Failure scenario (unit: seconds)	**Reaction time**	**Repair time**	**Recovery time**	**Outage time**
Scenario I	0.031	1.009	1.500	1.531
Scenario II	38.187	262.542	262.665	300.852

Table 2 Node Failure where Scenario I due to administrative deletion of a node by deleting a node wihle Scenario II due to externally triggered node failure by restarting the VM

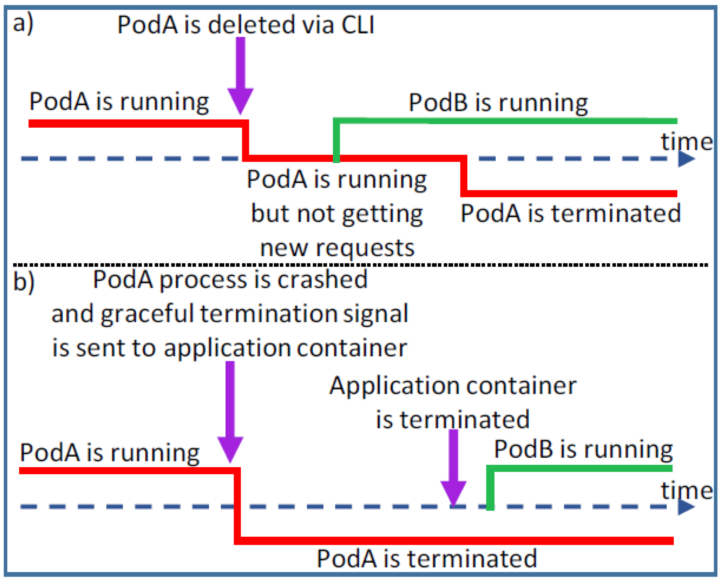

a)
PodA is deleted via CLI
PodA is running
PodB is running
time
PodA is running but not getting new requests
PodA is terminated
b)
PodA process is crashed and graceful termination signal is sent to application container
Application container is terminated
PodA is running
PodB is running
time
PodA is terminated

Figure 3: Private cloud exposing services via ingress

However, the downtime was significantly higher when the default configuration was used. Although the default configuration can be changed, figuring out how to reconfigure Kubernetes' reaction to node failures while avoiding network overhead and false positive reports can be complicated and requires a great effort.

High availability feature will be used in the research for maintaining server uptime.

RESEARCH PAPER 2

Kim, Dongmin & Muhammad, Hanif & Kim, Eunsam & Helal, Sumi & Lee, Choonhwa. (2019). TOSCA-Based and Federation-Aware Cloud Orchestration for Kubernetes Container Platform. Applied Sciences.

This research work proposes a method to automatically form and monitor Kubernetes Federation, given application topology descriptions in TOSCA (Topology and Orchestration Specification for Cloud Applications), by extending the orchestration tool that automatizes the modeling and instantiation of cloud applications. It also demonstrates the successful federation of the clusters according to the TOSCA specifications and verifies the auto-scaling capability of the configured system through a scenario in which the servers of a sample application are deployed and federated.

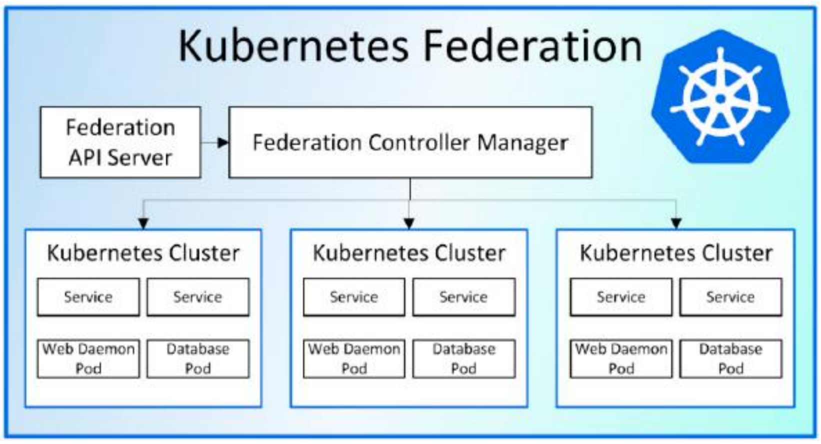


Figure 4 Kubernetes Federation Architecture

The container clusters could be automatically distributed and federated to the service areas of a cloud service provider and achieved efficient utilization of cluster computing resources using cloud management tools.

The research work also describes Horizontal Pod Auto-scaler (HPA) which allows active auto-scaling number of pods by defining Kubernetes HPA definition. The number of pods scaled up or down according to incoming traffic.

In the evaluation, a simple web game server scenario has been devised to prove the operability of the proposed system and to verify its auto-scaling ability. Figure 5 illustrates such an execution environment setup across multiple clusters. User interactions with the game are being handled by the load balancer to automatically add or remove web pods in both the Kubernetes clusters in Tokyo, Japan, and Oregon, US according to the workload and number of online users in the system. Figure 5 presents the skeleton code of the corresponding TOSCA descriptions of the game server federation. In the figure, Kubernetes Federation and cluster components are defined at element 1 through element 3, and HorizontalPodAutoscaler is defined as a TOSCA Node at element 6. Cloudify Manager 4.2 (Cloudify, New York, NY, USA, 2017) was used to run in a virtual machine configured with CentOS 7 x64 as its operating system hosted by VirtualBox in the Ubuntu 16.04 LTS x64 environment. Kubernetes 1.8 was used run on Google Cloud Platform, and Kubernetes clusters were built in the Tokyo, Japan and Oregon, US areas. Each cluster contains two nodes; the node type is n1-standard-2 (vCPU 2, RAM 7.5 GB), and Nginx is used as Web daemon, and MongoDB as the database server by default. Each node has a web pod, database pod, and a persistent data volume. Each web pod and DB pod have assigned a unique IP address through which it can be accessed accordingly. The number of nodes is later scaled up or down according to the incoming traffic.

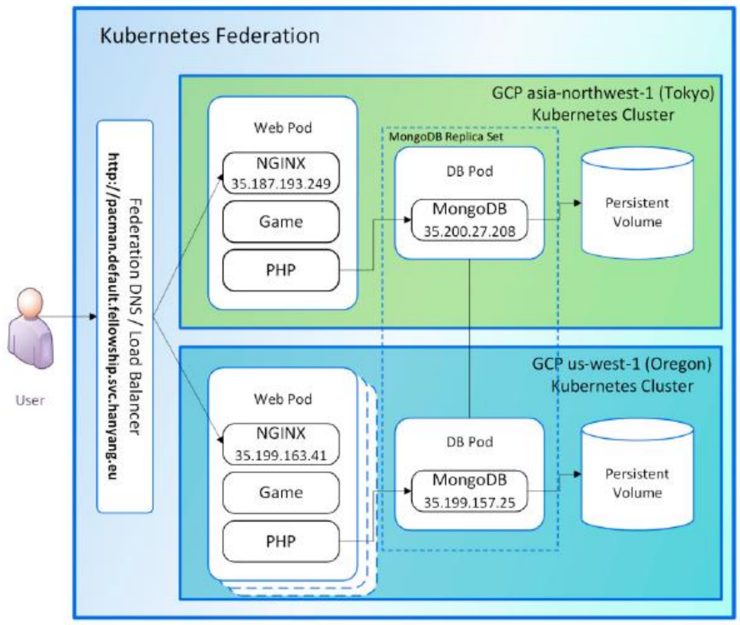


Figure 5 Kubernetes Federation Evaluation Setup

As the Kubernetes clusters are federated, the game service is automatically provided by a cluster in the other service area in the event of a sudden spike in the workload. The computing power must be increased by adding more pods to the available node pool, if necessary.

To describe the operation, firstly, in the normal operation status of the application, the number of nodes in the Tokyo cluster is one. There is also a single pod in the Oregon cluster as shown in the figure. After a sudden increase in incoming client requests to the cluster, the system automatically adjusts the number of Pacman pods in the Oregon cluster to handle the workload surge smoothly, while maintaining the system performance. The graph compares CPU usage in the federated clusters under normal and heavy load cases. As the input grows beyond the capacity, the federated HPA kicks in to add more pods to the Oregon cluster. In the experimental run, up to four pods are allocated to distribute the load increase among them, which is indicated in the case of "Heavy Load with HPA". The "Heavy Load" case represents a single pod case for the same load. It is noted that the target CPU usage for HPA is set to 500 millicores in the experiment.

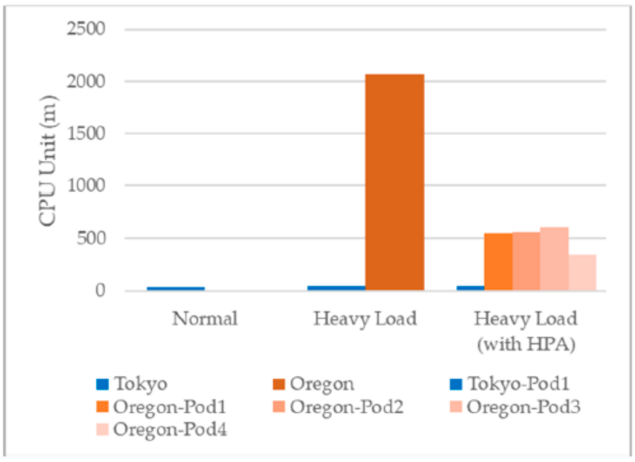


Figure 6: CPU usage of the clusters across the region

By using the prototype, it was verified that container clusters could be automatically distributed and federated to the service areas of a cloud service provider. The system proposed in this research article allows active auto-scaling using Kubernetes federation HPA.

An application will be developed and will be used in the research for managing Kubernetes and auto scaler function.

RESEARCH PAPER 3

Takahashi, Kimitoshi & Aida, Kento & Tanjo, Tomoya & Sun, Hiroshi. (2018). A Portable Load Balancer for Kubernetes Cluster. 222-231. 10.1145/3149457.3149473.

The paper emphasize that Kubernetes does not include a load balancer and is heavily dependent on external load balancers that are set up on the fly by cloud providers through their application protocol interfaces (APIs). Hence, a portable load balancer will be created by researchers and test its capabilities.

Since Kubernetes relies on external load balancers provided by cloud providers, it is difficult to use in environments where there are no supported load balancers. This is particularly true for on-premises data centers, or for all but the largest cloud providers.

In this paper, the researchers proposed a portable load balancer that was usable in any environment, and hence facilitated web services migration. They implemented a containerized software load balancer that is run by Kubernetes as a part of container cluster, using Linux kernel's Internet Protocol Virtual Server (IPVS). Then researchers compared the performance of our proposed load balancer with existing iptables Destination Network Address Translation (DNAT) and the Nginx load balancers.

The research proposed IPVS portable load balancer that was usable in any environment, and hence facilitated web services migration.

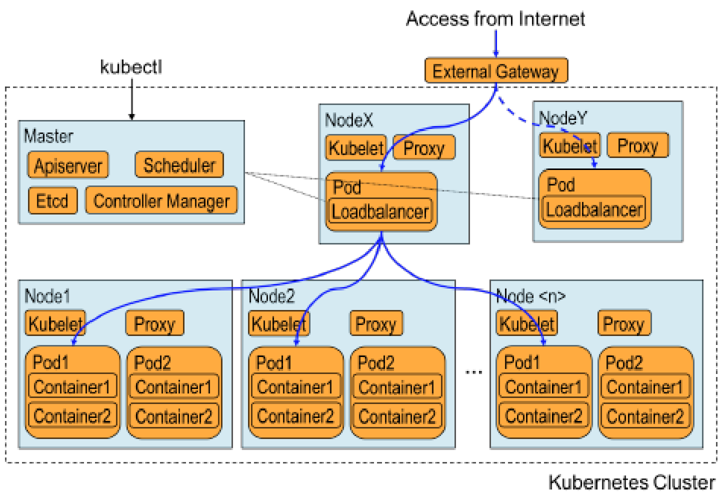


Figure 7: Kubernetes cluster with proposed load balancer

Figure 7 shows the proposed Kubernetes cluster architecture, which has the following characteristics:

1. Each load balancer itself is run as a pod by Kubernetes.
2. Load balancer configurations are dynamically updated based on information about running pods.

The proposed load balancer can resolve the conventional architecture problems, as follows: Since the load balancer itself is containerized, load balancer can run in any environment including on-premise data centers, even without external load balancers that is supported by Kubernetes. The incoming traffic is directly distributed to designated pods by the load balancer. It makes the administration, e.g. finding malfunctions, easier.

The researchers designed the proposed load balancer using three components, IPVS, keepalived, and a controller. These components are placed in a Docker container image. The IPVS is a Layer-4 load balancer capability, which is included in the Linux kernel 2.6.0 released in 2003 or later, to distribute incoming Transmission Control Protocol (TCP) traffic to real servers. For example, IPVS distributes incoming Hypertext Transfer Protocol (HTTP) traffic destined for a single destination IP address, to multiple HTTP servers (e.g. Apache HTTP or nginx) running on multiple nodes in order to improve the performance of web services. Keepalived is a management program that performs health checking for real servers and manage IPVS balancing rules in the kernel accordingly. It is often used together with IPVS to facilitate ease of use. The controller is a daemon that periodically monitors the pod information on the master and performs various actions when such information changes. Kubernetes provides ingress controller framework as the Go Language (Golang) package to implement such controllers. Researchers have implemented a controller program that will feed pod state changes to keepalived using this framework.

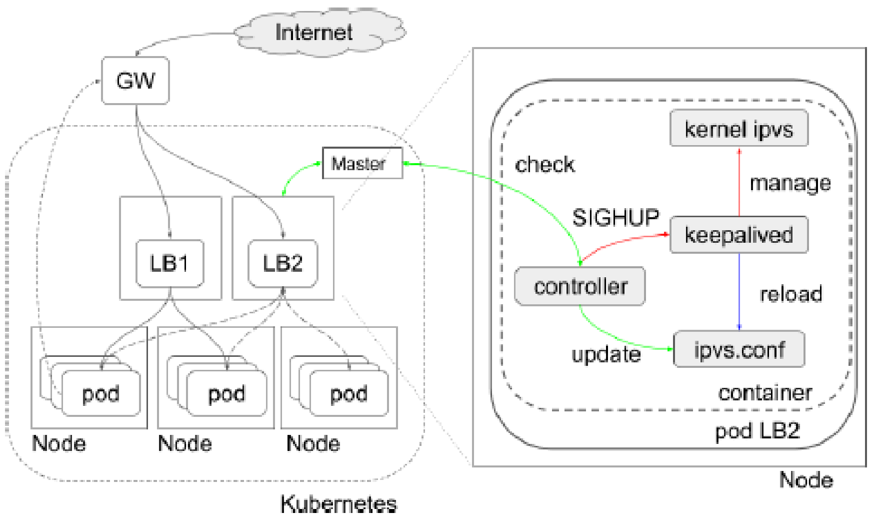


Figure 8: Load Balancer setup detailed

The proposed load balancer needs to dynamically reconfigure the IPVS balancing rules whenever pods are created/deleted. Figure 8 is a schematic diagram to show the dynamic reconfiguration of the IPVS rules. The right part of the figure shows the enlarged view of one of the nodes where the load balancer pod (LB2) is deployed. Two daemon programs, controller and keepalived, run in the container inside the LB2 pod are illustrated. The keepalived manages Linux kernel's IPVS rules depending on the ipvs.conf configuration file. It is also capable of health-checking the liveliness of real server, which is represented as a combination of the IP addresses and port numbers of the target pods. If the health check to a real server fails, keepalived will remove that real server from the IPVS rules. The controller monitors information concerning the running pods of a service in the Kubernetes cluster by consulting the apiserver running on the master. Whenever pods are created or deleted, the controller will automatically regenerate an appropriate ipvs.conf and issue SIGHUP to keepalived. Then, keepalived will reload the ipvs.conf and modify the kernel's IPVS rules accordingly. The actual controller is implemented using the Kubernetes ingress controller framework. By importing existing Golang package, "k8s.io/ingress /core/pkg/ingress", it could simplify the implementation.

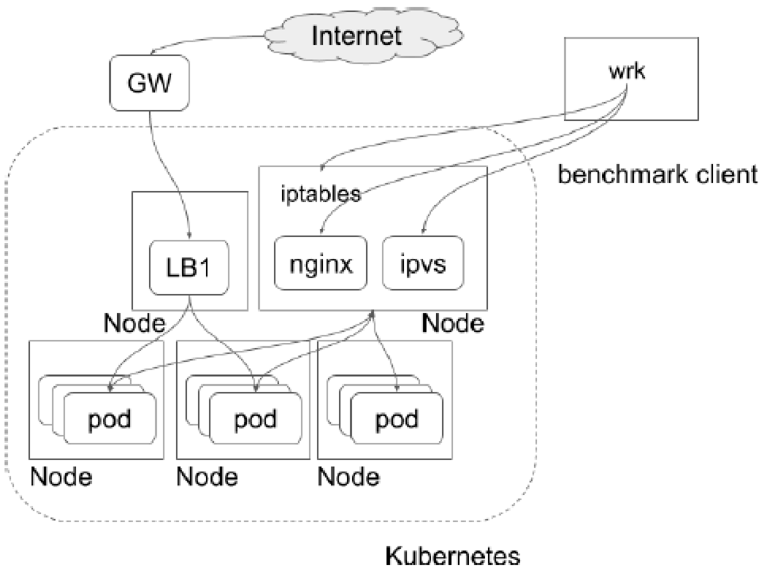
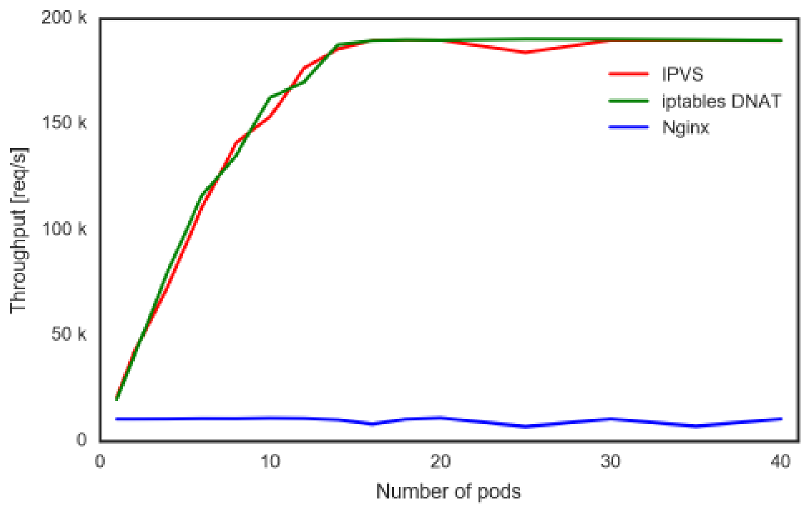


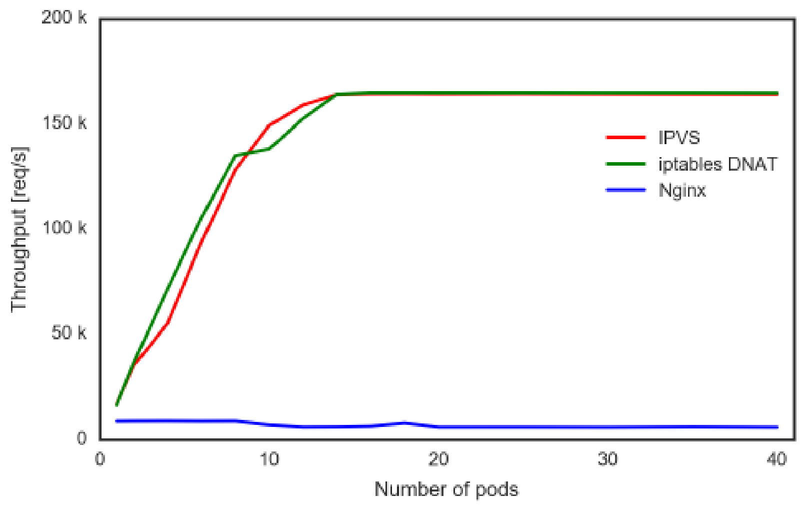
Figure 9: Load Balancer setup and benchmark

The performance of the load balancers was measured using the wrk. Figure 8 illustrates a schematic diagram of the experimental setup. Multiple pods are deployed on multiple nodes in the Kubernetes cluster. In each pod, a Nginx web server that returns the IP address of the pod are running. Then set up the IPVS, iptables DNAT, and Nginx load balancers on one of the nodes (the top right node in the Figure 6). The throughput also measured, Request/sec, of the web service running on the Kubernetes cluster as follows: The HTTP GET requests are sent out by the wrk on the client machine toward the nodes, using destination IP addresses and port numbers that are chosen based on the type of the load balancer on which the measurement is performed. The load balancer on the node then distributes the requests to the pods. Each pod will return HTTP responses to the load balancer, after which the load balancer returns them to the client. Based on the number of responses received by wrk on the client, load balancer performance, in terms of Request/sec can

Network overlay, flannel, was used to build the Kubernetes cluster in the experiment. Flannel has three types of backend, i.e., operating modes, named host-gw, vxlan, and udp.



(host-gw)



(vxlan)

Figure 10: Performance of Load Balancers with (RSS, RPS) = (off on)

Figure 9 compares the performance measurement results among the IPVS, iptables DNAT, and Nginx load balancers with the condition of "(RSS, RPS) = (off on)". The proposed IPVS load balancer exhibits almost equivalent performance as the existing iptables DNAT based load balancer. The Nginx based load balancer shows no performance improvement even though the number of the Nginx web server pods is increased. It is understandable because the performance of the Nginx as a load balancer is expected to be similar to the performance as a web server.

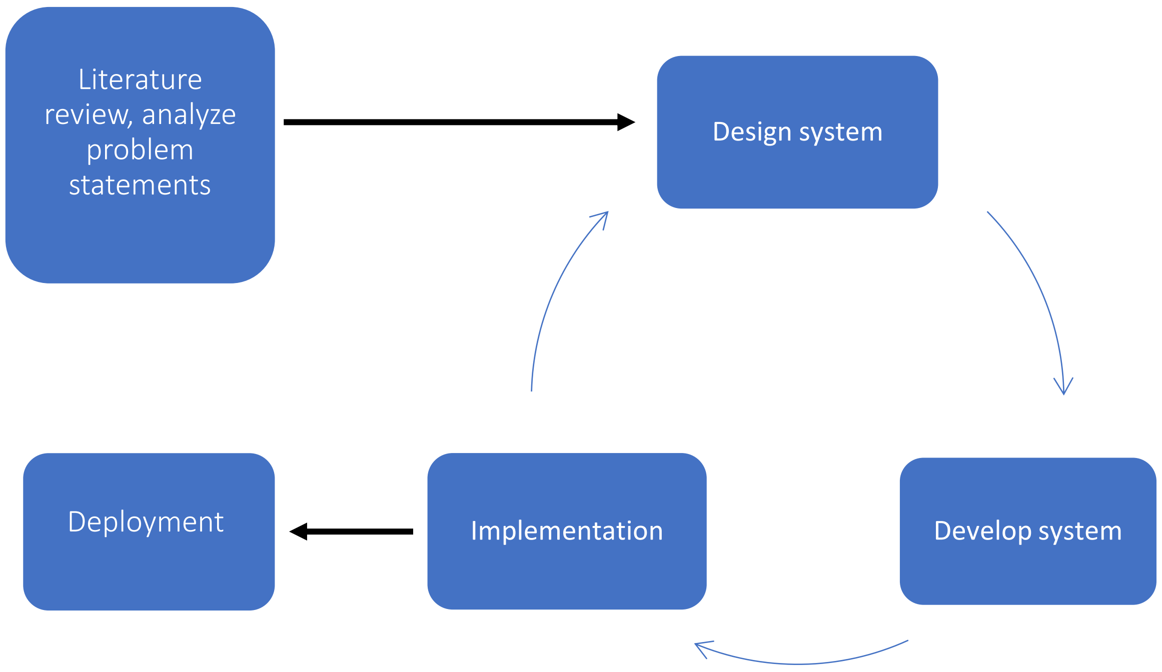
The experimental results indicate that the IPVS based load balancer in container improves the portability of the Kubernetes cluster system while it shows the similar performance levels as the existing iptables DNAT based load balancer. Nginx did not show performance improvement even though number of pods increases.

Experimental results indicate that the IPVS based load balancer in container improves the portability of the Kubernetes cluster system while it shows the similar performance levels as the existing iptables DNAT based load balancer. They also clarified that choosing the right operating modes of overlay networks is important for the performance of load balancers. For example, in the case of flannel, only the vxlan and udp backend operation modes could be used in the cloud environment, and the udp backend significantly degraded their performance. Furthermore, they also learned that the distribution of packet processing among multiple CPUs was very important to obtain the maximum performance levels from load balancers.

The load balancer will be used to balancing utilization of pods and high availability of pods.

3.0 RESEARCH METHODOLOGY

To develop the features into the server, Rapid Application Deployment (RAD) is used.



Literature review and analyze problem statements.

In this phase, analysis is done on features of Kubernetes. Literature review on three articles are chosen to review which are closely related to the project. These papers will be selected and referred to obtain ideas. The papers explain on the design and development of Kubernetes features along with the benchmarking result before implementation and after implementation of the features such as high availability, pod auto scaler and software load balancer.

After analyzing three papers, problem statements will be analyzed to solve the issues made by researchers on these papers.

Design System

This phase where an early system architecture will be developed to fit these features into the server.

Develop System

All the installation of the Kubernetes components will be done in this phase. This includes the plugin, sub module, YAML file deployment and container image. Deploying load balancer, auto scaler configuration files and creating an application.

Implementation

Coding and implementation of API will be performed and if necessary, refine the application to increase the performance. Application such as Horizontal Pod Autoscaler and software load balancer will be implemented and integrated. If there are needs of improvement and redesign of small part of module, system will have to be redesign again. Testing will also be done in this phase and along with getting the measurables.

After completion of implementation, proceed to deployment.

Deployment

The Kubernetes system will be deployed and run in a bare-metal hypervisor on-premises server or public cloud such as AWS, GCP or Azure. The system should be able to load balance traffic and ensure high availability of the server. Once deployed, a simple application that is written in Python will be used to simulate failure of the system.

4.0 SYSTEM ANALYSIS AND DESIGN

In this section, all the requirements for the developing system have been stated out and defined thoroughly. By understanding and analyzing the background and the objective of the project, the goal of the project could be determined. Thus, the requirements during the system development stages will be learned to make sure that the system able to run and process.

4.1 SYSTEM REQUIREMENTS

Several applications will be used for testing:

- Kubernetes
- Kubeadm cluster manager
- Kubectl
- Ubuntu OS (20.04)
- Ubuntu OS (18.04)
- Sample PHP docker image for load testing
- Flannel network overlay for Kubernetes
- Text editor for creating YAML file
- Kubernetes Python client
- MetalLB bare-metal load balancer
- NGINX Ingress controller

To use the following tools, isolated environment will be set up such as using a hypervisor Virtualbox and an external network to home network.

4.2 SYSTEM ARCHITECTURE

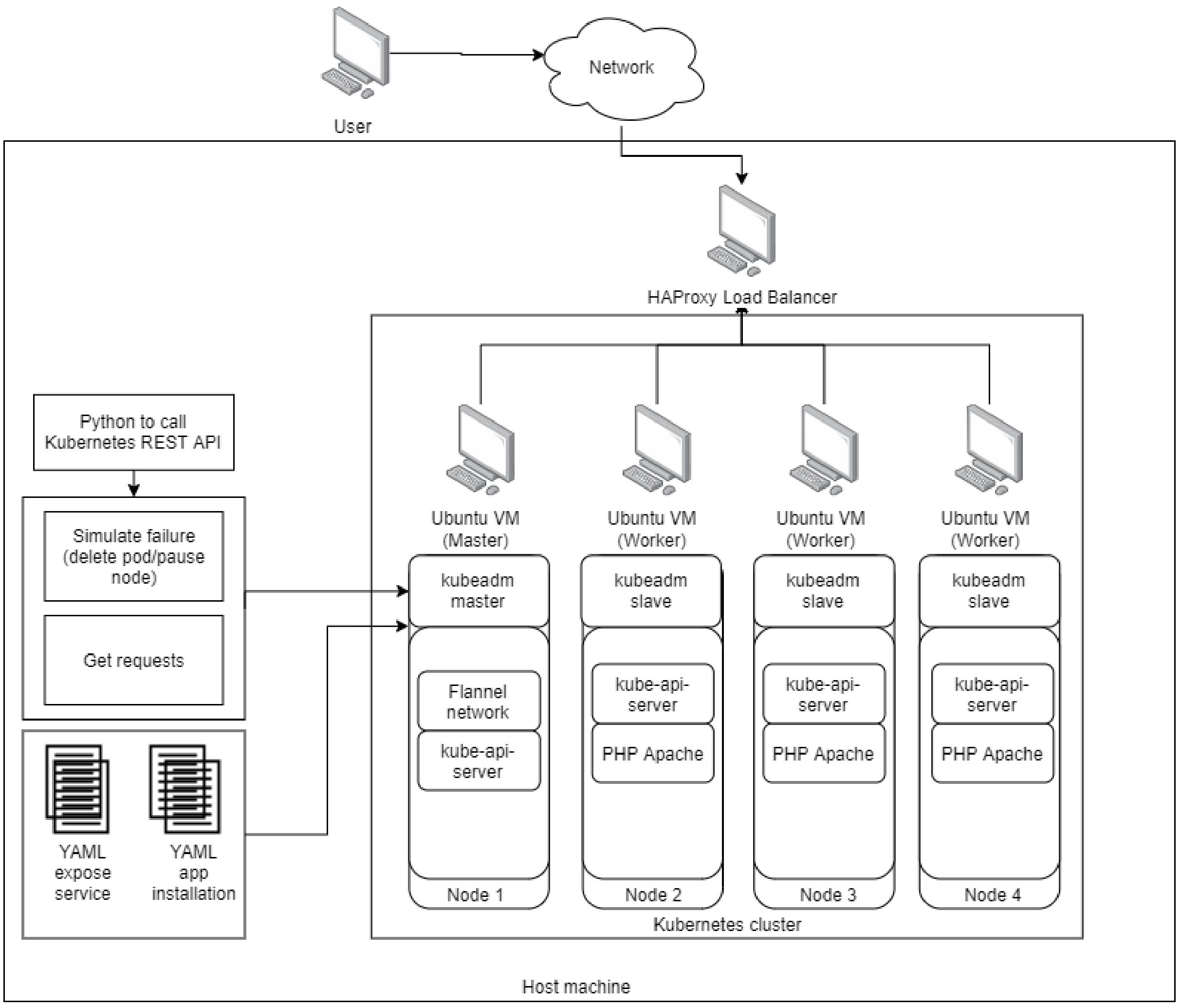


Figure 11: Kubernetes setup

The figure above shows on how the Kubernetes will setup and along with other processes. Three virtual machines installed with Ubuntu 20.04 in Oracle Virtualbox, included with Kubernetes cluster management called kubeadm, one master and two slaves. YAML files are defined and created so to be deployed into the master node. API token will also be generated to allow calling using Python through REST or requests. The nodes are bridged to one network adapter to be connected to one network.

4.3 FUNCTIONAL REQUIREMENTS

The server should be able to:

- Scale the number of pods by increasing or decreasing depending on metrics usage
- Load balancing between pods and nodes to get equal usage
- Ensure high availability when the pods fail. A pod will be resurrected when another pod fails and failover when node is down. Minimal down time should be reasonable and not too long
- Show the usage of pods
- Show load balancer statistics
- Get API request from client to simulate failure

4.4 KUBERNETES SERVICES

In order to expose the application to the users, Kubernetes uses service such as ClusterIP, NodePort and LoadBalancer. NodePort will be used for implementation. The diagrams below show a service NodePort 32002 is exposed to users.

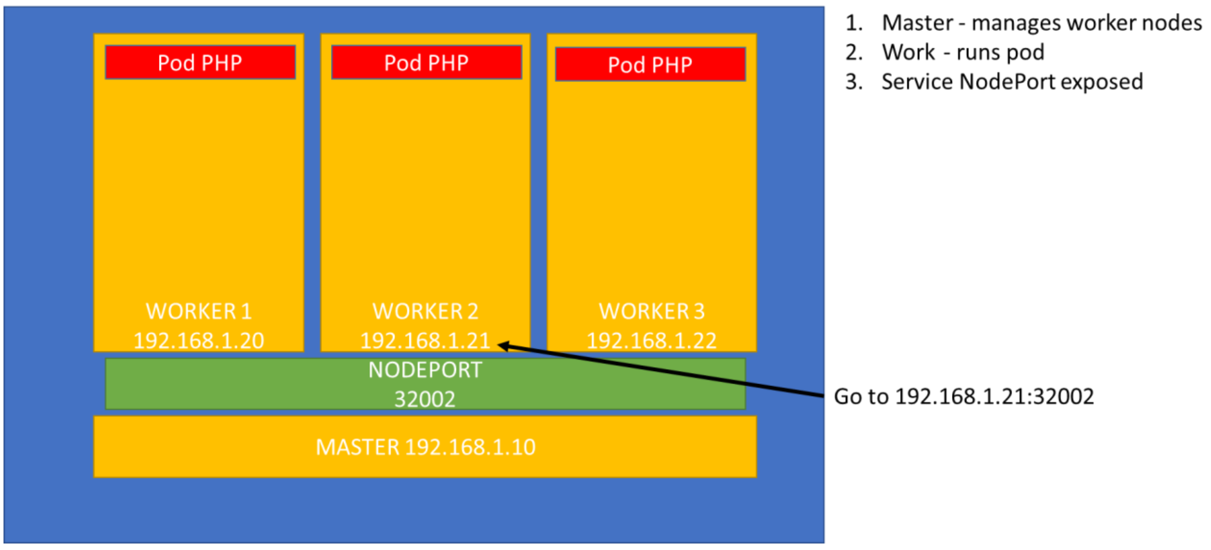


Figure 12: A user goes into Worker 2 IP addresses to get requests

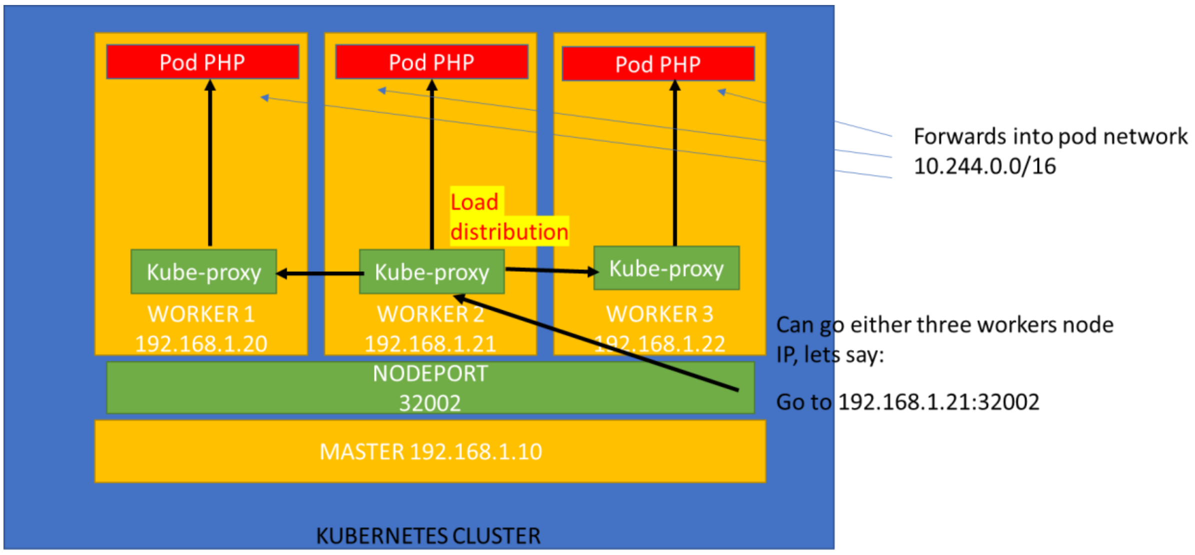


Figure 13: When a user reaches Worker 2, the kube-proxy distribute the load to pod

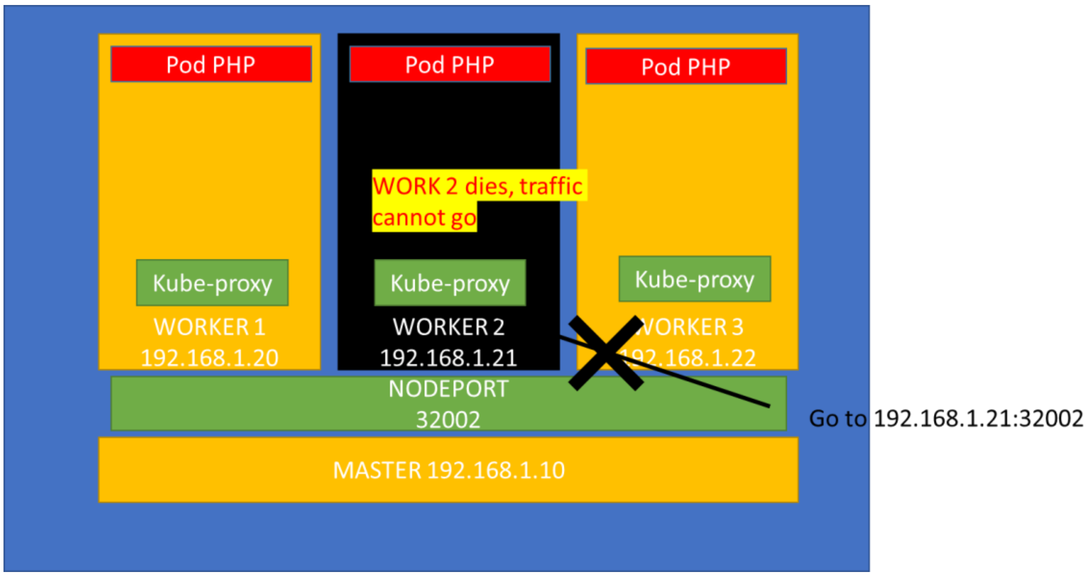


Figure 14: Worker 2 dies and the traffic unable to get requests from the server. This means failure to the server.

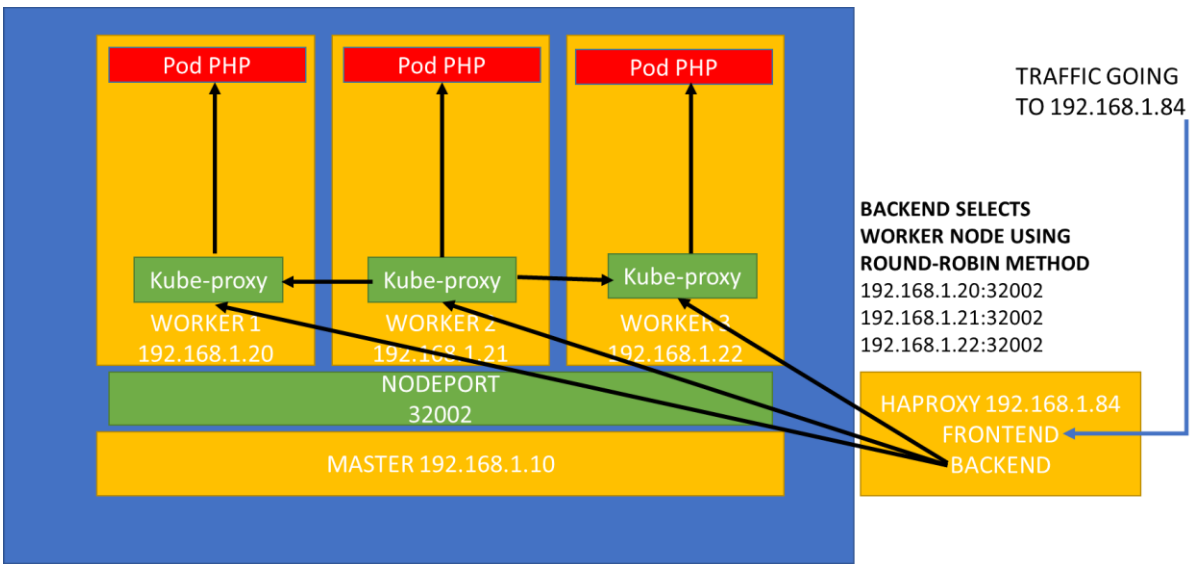


Figure 15: Using HAProxy will solve the failure issue. HAProxy provides load balancing and high availability features for server by forwarding traffic to the servers.

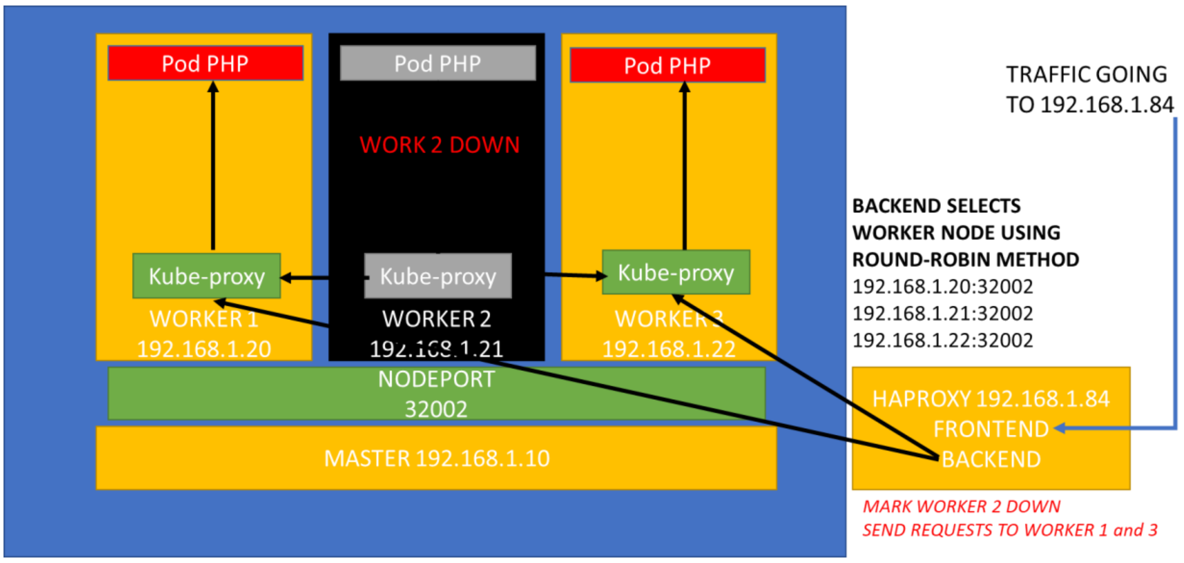


Figure 16: HAProxy detects Worker 2 is down and proceed to forward traffic to Worker 1 and Worker 3.

5.0 SYSTEM DEVELOPMENT

The system development phase will explain on how the server will be built.

5.1 SYSTEM IMPLEMENTATION

Using the tools as mentioned in Chapter 4, the system will be installed, configured, and implemented.

Components of the system includes:

Load Balancer

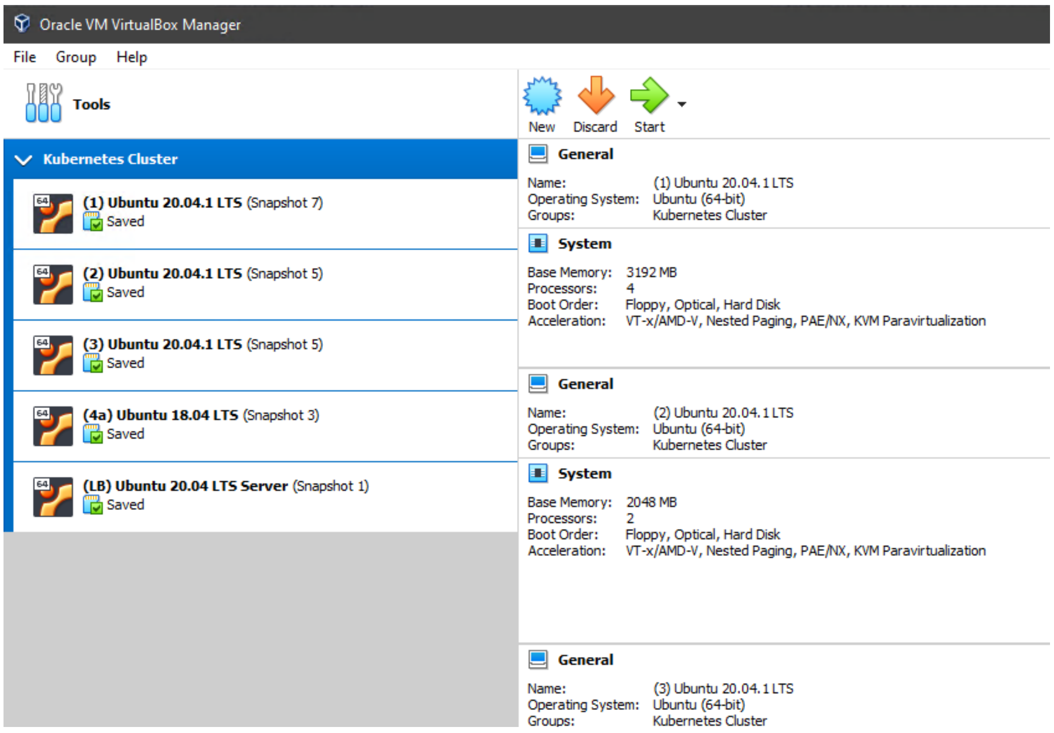
- HAProxy (High-Performance Load Balancer)

HAPROXY

Horizontal Pod Autoscaler

- Taint
- Pod Affinity
- Descheduler
- Kube-controller configuration file
- Replica
- CronJob

Five virtual machines provisioned with 4 CPU, 3 GB and 2 CPU, 2 GB for another three virtual machines. One virtual machine with 2 CPU, 1 GB for load balancer application. Kubeadm cluster management installed on four machines, consists of 1 master node and 3 worker nodes.



Kubernetes Cluster

(1) Ubuntu 20.04.1 LTS (Snapshot 7)
Saved

(2) Ubuntu 20.04.1 LTS (Snapshot 5)
Saved

(3) Ubuntu 20.04.1 LTS (Snapshot 5)
Saved

(4a) Ubuntu 18.04 LTS (Snapshot 3)
Saved

(LB) Ubuntu 20.04 LTS Server (Snapshot 1)
Saved

General

Name:	(1) Ubuntu 20.04.1 LTS
Operating System:	Ubuntu (64-bit)
Groups:	Kubernetes Cluster

System

Base Memory:	3192 MB
Processors:	4
Boot Order:	Floppy, Optical, Hard Disk
Acceleration:	VT-x/AMD-V, Nested Paging, PAE/NX, KVM Paravirtualization

General

Name:	(2) Ubuntu 20.04.1 LTS
Operating System:	Ubuntu (64-bit)
Groups:	Kubernetes Cluster

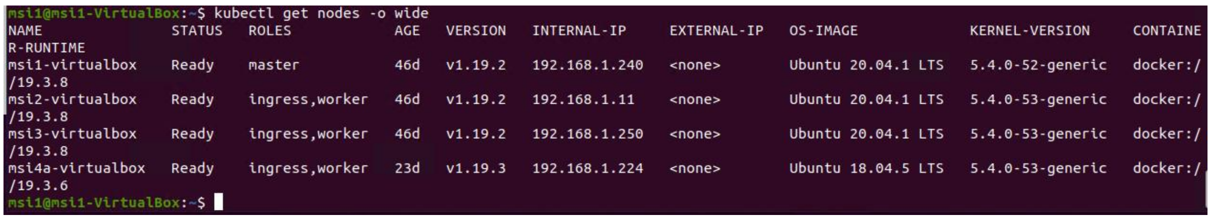
System

Base Memory:	2048 MB
Processors:	2
Boot Order:	Floppy, Optical, Hard Disk
Acceleration:	VT-x/AMD-V, Nested Paging, PAE/NX, KVM Paravirtualization

General

Name:	(3) Ubuntu 20.04.1 LTS
Operating System:	Ubuntu (64-bit)
Groups:	Kubernetes Cluster

Figure 17: Installed virtual machines using Virtualbox



```
msi1@msi1-VirtualBox:~$ kubectl get nodes -o wide
NAME             STATUS   ROLES            AGE   VERSION   INTERNAL-IP     EXTERNAL-IP   OS-IMAGE             KERNEL-VERSION     CONTAINE
R-RUNTIME
msi1-virtualbox   Ready    master           46d   v1.19.2   192.168.1.240   <none>        Ubuntu 20.04.1 LTS   5.4.0-52-generic   docker:/
/19.3.8
msi2-virtualbox   Ready    ingress,worker   46d   v1.19.2   192.168.1.11    <none>        Ubuntu 20.04.1 LTS   5.4.0-53-generic   docker:/
/19.3.8
msi3-virtualbox   Ready    ingress,worker   46d   v1.19.2   192.168.1.250   <none>        Ubuntu 20.04.1 LTS   5.4.0-53-generic   docker:/
/19.3.8
msi4a-virtualbox  Ready    ingress,worker   23d   v1.19.3   192.168.1.224   <none>        Ubuntu 18.04.5 LTS   5.4.0-53-generic   docker:/
/19.3.6
msi1@msi1-VirtualBox:~$
```

Figure 18: List of nodes and other details

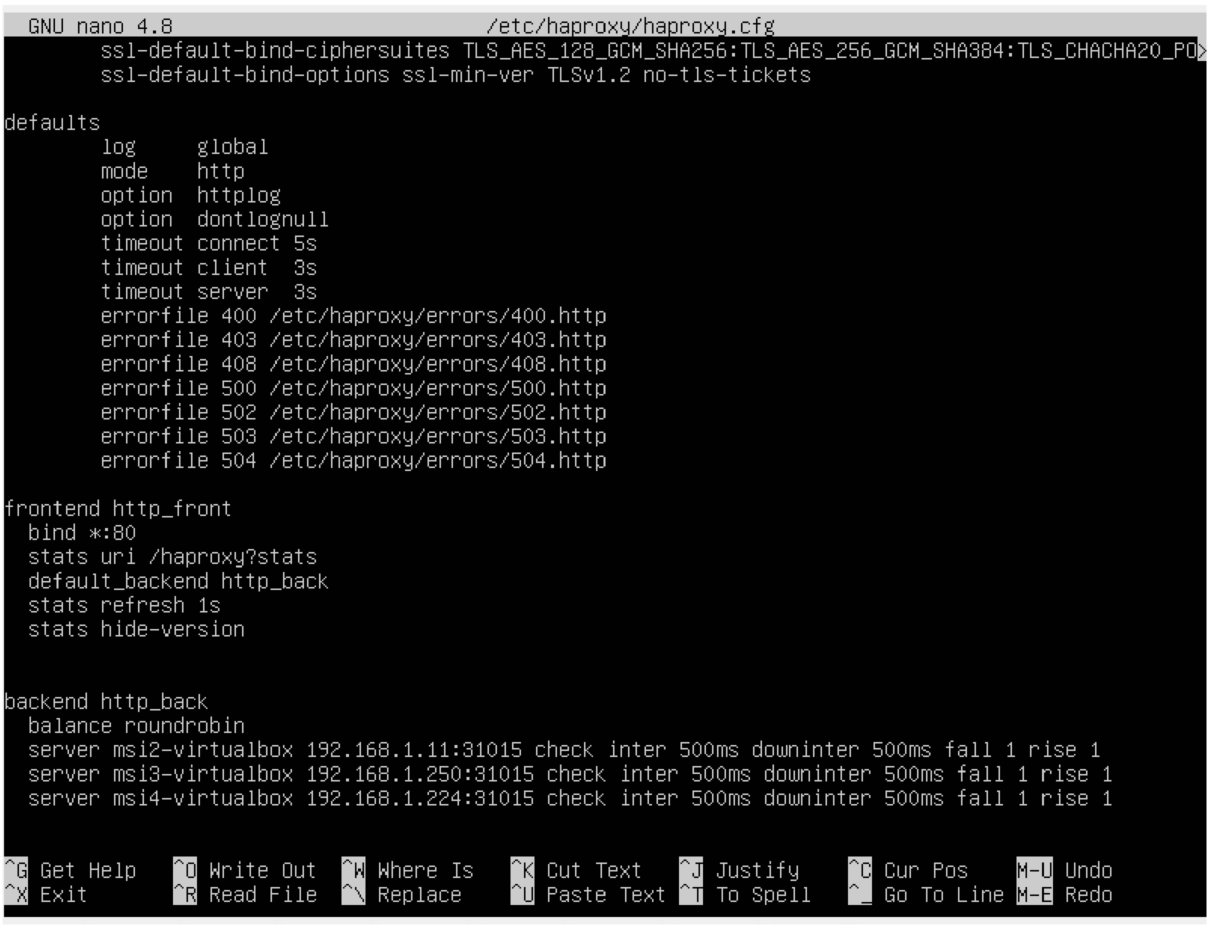
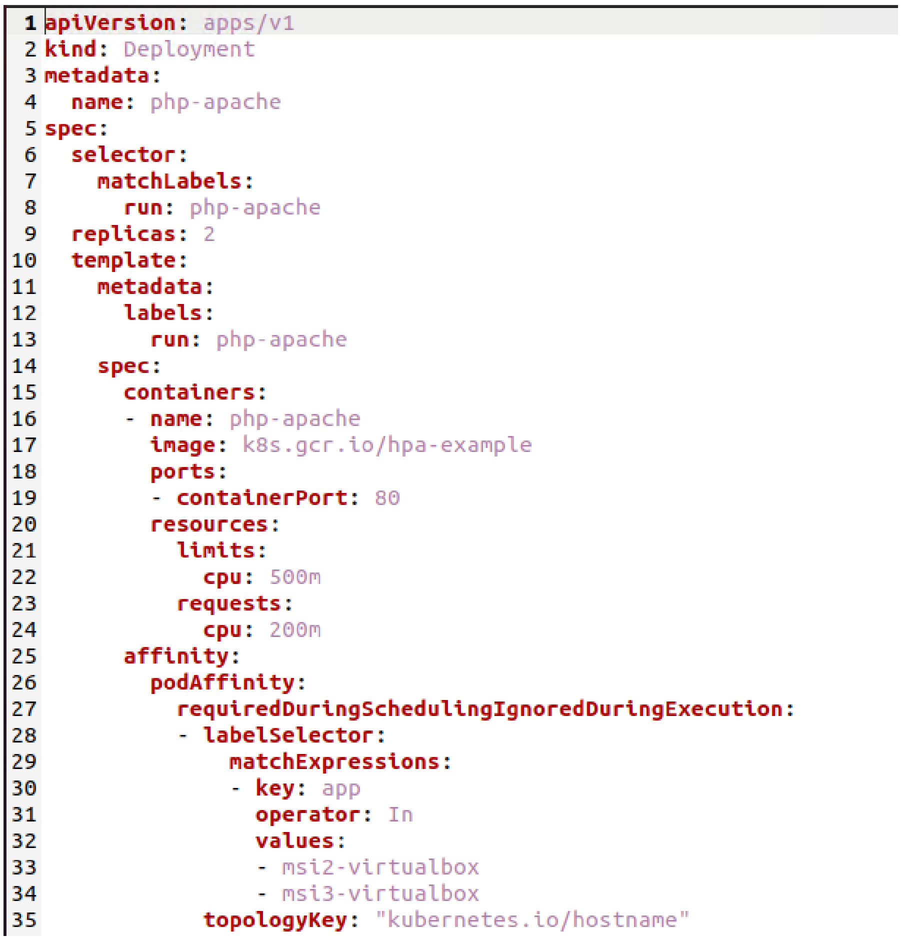

```
  GNU nano 4.8                          /etc/haproxy/haproxy.cfg
        ssl-default-bind-ciphersuites TLS_AES_128_GCM_SHA256:TLS_AES_256_GCM_SHA384:TLS_CHACHA20_PO>
        ssl-default-bind-options ssl-min-ver TLSv1.2 no-tls-tickets

defaults
        log     global
        mode    http
        option  httplog
        option  dontlognull
        timeout connect 5s
        timeout client  3s
        timeout server  3s
        errorfile 400 /etc/haproxy/errors/400.http
        errorfile 403 /etc/haproxy/errors/403.http
        errorfile 408 /etc/haproxy/errors/408.http
        errorfile 500 /etc/haproxy/errors/500.http
        errorfile 502 /etc/haproxy/errors/502.http
        errorfile 503 /etc/haproxy/errors/503.http
        errorfile 504 /etc/haproxy/errors/504.http

frontend http_front
  bind *:80
  stats uri /haproxy?stats
  default_backend http_back
  stats refresh 1s
  stats hide-version


backend http_back
  balance roundrobin
  server msi2-virtualbox 192.168.1.11:31015 check inter 500ms downinter 500ms fall 1 rise 1
  server msi3-virtualbox 192.168.1.250:31015 check inter 500ms downinter 500ms fall 1 rise 1
  server msi4-virtualbox 192.168.1.224:31015 check inter 500ms downinter 500ms fall 1 rise 1


^G Get Help    ^O Write Out   ^W Where Is    ^K Cut Text    ^J Justify     ^C Cur Pos     M-U Undo
^X Exit        ^R Read File   ^\ Replace     ^U Paste Text  ^T To Spell    ^_ Go To Line  M-E Redo
```

Figure 19: Configuration file of HAProxy in a mchine

5.2 YAML IMPLEMENTATION

YAML file stands for Yet Another Markup Language, human readable text-based format to specify configuration information. YAML files will be used to deploy Deployment, Services, DaemonSet, Service, Ingress and other kinds. Some YAML files can be directly imported from repository (GitHub, Community web). To deploy an application based on needs, the file have been modified such as replicas, port number and name.



```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: php-apache
spec:
  selector:
    matchLabels:
      run: php-apache
  replicas: 2
  template:
    metadata:
      labels:
        run: php-apache
    spec:
      containers:
      - name: php-apache
        image: k8s.gcr.io/hpa-example
        ports:
        - containerPort: 80
        resources:
          limits:
            cpu: 500m
          requests:
            cpu: 200m
      affinity:
        podAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
              matchExpressions:
              - key: app
                operator: In
                values:
                - msi2-virtualbox
                - msi3-virtualbox
            topologyKey: "kubernetes.io/hostname"
```

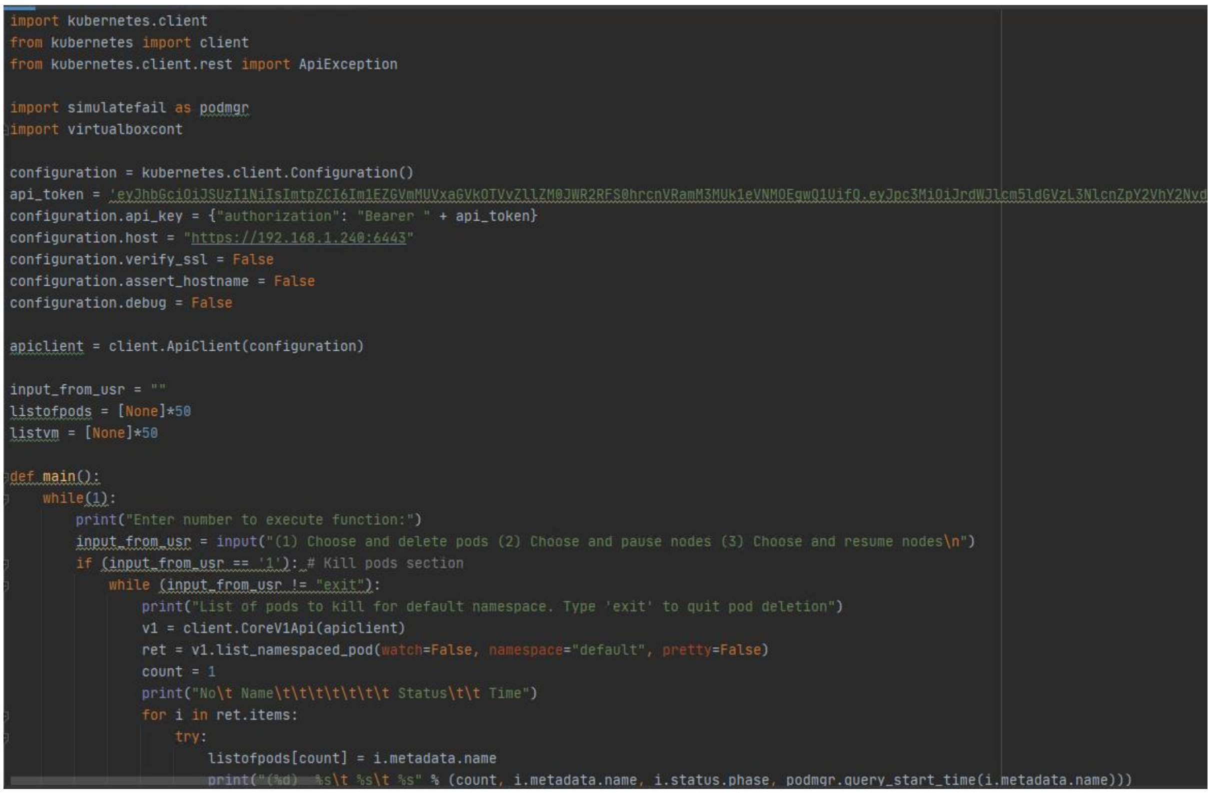
Figure 20: YAML template for Deployment of php-apache

5.3 KUBERNETES PYTHON API AND VIRTUALBOX API

Official Kubernetes client can be used to call API via Python. Administrator can use API for deployment, debugging, diagnostic and others. In this project, API will be used to simulate failure by deleting pods, pause and resume VM.

Function call includes:

- delete_namespaced_pod
- list_namespaced_pod
- list_pod_for_all_namespaces
- virtualbox



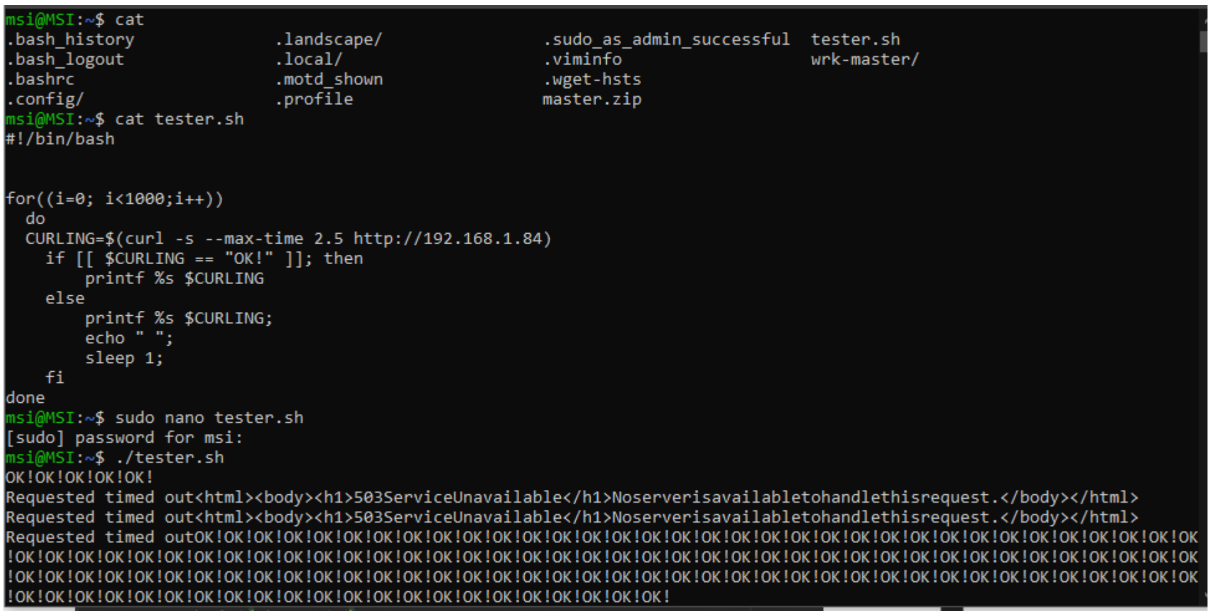
```python
import kubernetes.client
from kubernetes import client
from kubernetes.client.rest import ApiException

import simulatefail as podmgr
import virtualboxcont

configuration = kubernetes.client.Configuration()
api_token = 'eyJhbGciOiJSUzI1NiIsImtpZCI6Im1EZGVmMUVxaGVkOTVvZllZM0JWR2RFS0hrcnVRamM3MUk1eVNMOEgwQ1UifQ.eyJpc3MiOiJrdWJlcm5ldGVzL3NlcnZpY2VhY2Nvd
configuration.api_key = {"authorization": "Bearer " + api_token}
configuration.host = "https://192.168.1.240:6443"
configuration.verify_ssl = False
configuration.assert_hostname = False
configuration.debug = False

apiclient = client.ApiClient(configuration)

input_from_usr = ""
listofpods = [None]*50
listvm = [None]*50

def main():
    while(1):
        print("Enter number to execute function:")
        input_from_usr = input("(1) Choose and delete pods (2) Choose and pause nodes (3) Choose and resume nodes\n")
        if (input_from_usr == '1'): # Kill pods section
            while (input_from_usr != "exit"):
                print("List of pods to kill for default namespace. Type 'exit' to quit pod deletion")
                v1 = client.CoreV1Api(apiclient)
                ret = v1.list_namespaced_pod(watch=False, namespace="default", pretty=False)
                count = 1
                print("No\t Name\t\t\t\t\t\t\t Status\t\t Time")
                for i in ret.items:
                    try:
                        listofpods[count] = i.metadata.name
                        print("(%d)  %s\t %s\t %s" % (count, i.metadata.name, i.status.phase, podmgr.query_start_time(i.metadata.name)))
```

Figure 21: Inside Python IDE to call Kubernetes API

6.0 SYSTEM TESTING AND EVALUATION

Using a separate client, Windows Subsystem Linux and bash script to run curl command to get requests from server:

```
#!/bin/bash


for((i=0; i<1000;i++))
  do
  CURLING=$(curl -s --max-time 2.5 http://192.168.1.84)
    if [[ $CURLING == "OK!" ]]; then
        printf %s $CURLING
    else
        printf %s $CURLING;
        echo " ";
        sleep 1;
    fi
done
```

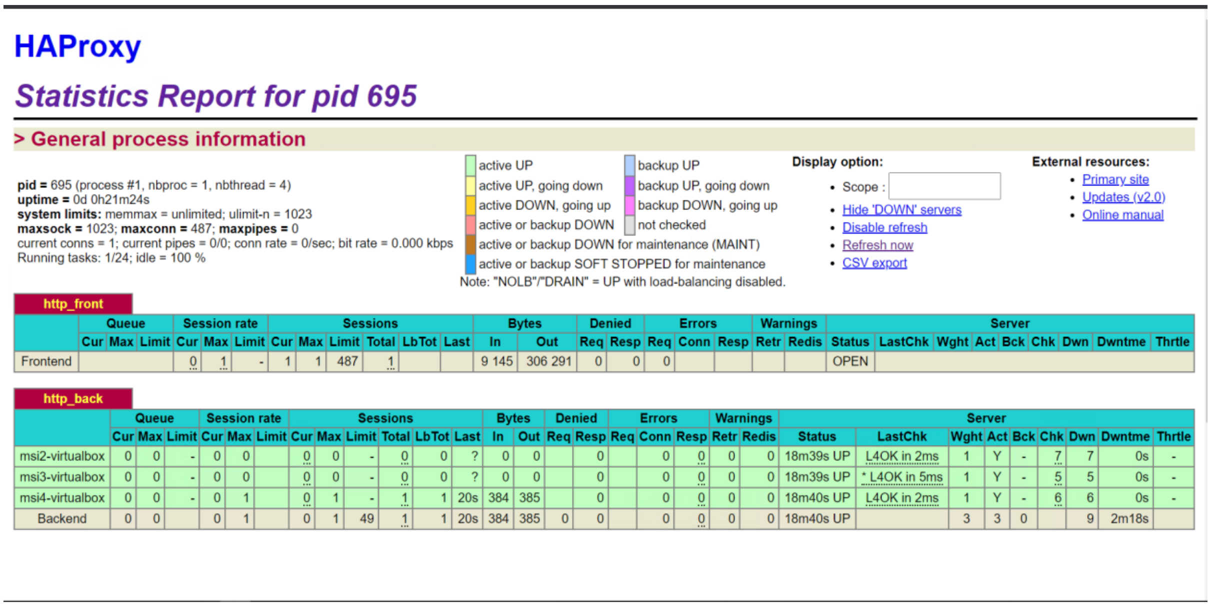


```
msi@MSI:~$ cat
.bash_history             .landscape/               .sudo_as_admin_successful  tester.sh
.bash_logout              .local/                   .viminfo                   wrk-master/
.bashrc                   .motd_shown               .wget-hsts
.config/                  .profile                  master.zip
msi@MSI:~$ cat tester.sh
#!/bin/bash


for((i=0; i<1000;i++))
  do
  CURLING=$(curl -s --max-time 2.5 http://192.168.1.84)
    if [[ $CURLING == "OK!" ]]; then
        printf %s $CURLING
    else
        printf %s $CURLING;
        echo " ";
        sleep 1;
    fi
done
msi@MSI:~$ sudo nano tester.sh
[sudo] password for msi:
msi@MSI:~$ ./tester.sh
OK!OK!OK!OK!OK!
Requested timed out<html><body><h1>503ServiceUnavailable</h1>Noserverisavailabletohandlethisrequest.</body></html>
Requested timed out<html><body><h1>503ServiceUnavailable</h1>Noserverisavailabletohandlethisrequest.</body></html>
Requested timed outOK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK
!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK
!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK
!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!
```

Figure 22: Running and sending traffic to load balancer server

6.1 LOAD BALANCER TEST

One configuration will be using without load balancer, where it will expose deployment to NodePort. HAProxy will be used. The load of pods will be inspected through kubectl command. CPU usage and memory usage should be almost equal among the pods in the Deployment. HAProxy should able to distribute traffic through the worker nodes using round robin method. HAProxy has ability to perform health check and identify which node has been failed and subsequently directs the traffic to active nodes.



HAProxy

Statistics Report for pid 695

> General process information

pid = 695 (process #1, nbproc = 1, nbthread = 4)
uptime = 0d 0h21m24s
system limits: memmax = unlimited; ulimit-n = 1023
maxsock = 1023; **maxconn =** 487; **maxpipes =** 0
current conns = 1; current pipes = 0/0; conn rate = 0/sec; bit rate = 0.000 kbps
Running tasks: 1/24; idle = 100 %

🟩	active UP	🟦	backup UP
🟨	active UP, going down	🟪	backup UP, going down
🟧	active DOWN, going up	🟪	backup DOWN, going up
🟥	active or backup DOWN	⬜	not checked
🟫	active or backup DOWN for maintenance (MAINT)		
🟦	active or backup SOFT STOPPED for maintenance		

Note: "NOLB"/"DRAIN" = UP with load-balancing disabled.

Display option:
- Scope :
- Hide 'DOWN' servers
- Disable refresh
- Refresh now
- CSV export

External resources:
- Primary site
- Updates (v2.0)
- Online manual

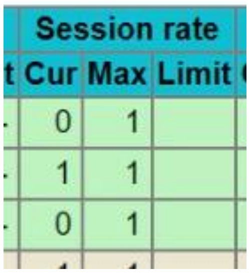
http_front

	Queue			Session rate			Sessions						Bytes		Denied		Errors			Warnings		Server								
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk	Wght	Act	Bck	Chk	Dwn	Dwntme	Thrtle
Frontend				0	1	-	1	1	487	1			9 145	306 291	0	0	0					OPEN								

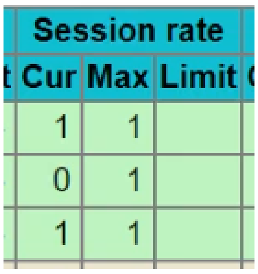
http_back

	Queue			Session rate			Sessions						Bytes		Denied		Errors			Warnings		Server								
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk	Wght	Act	Bck	Chk	Dwn	Dwntme	Thrtle
msi2-virtualbox	0	0	-	0	0		0	0	-	0	0	?	0	0		0		0	0	0	0	18m39s UP	L4OK in 2ms	1	Y	-	7	7	0s	-
msi3-virtualbox	0	0	-	0	0		0	0	-	0	0	?	0	0		0		0	0	0	0	18m39s UP	* L4OK in 5ms	1	Y	-	5	5	0s	-
msi4-virtualbox	0	0	-	0	1		0	1	-	1	1	20s	384	385		0		0	0	0	0	18m40s UP	L4OK in 2ms	1	Y	-	6	6	0s	-
Backend	0	0		0	1		0	1	49	1	1	20s	384	385	0	0		0	0	0	0	18m40s UP		3	3	0		9	2m18s	

Figure 23: HAProxy Statistics page

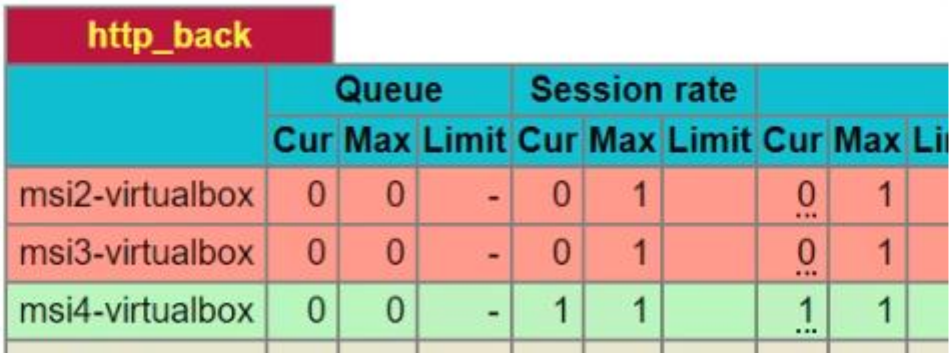


Session rate		
Cur	Max	Limit
0	1	
1	1	
0	1	



Session rate		
Cur	Max	Limit
1	1	
0	1	
1	1	

Figure 24: Load distribution in session rate



http_back

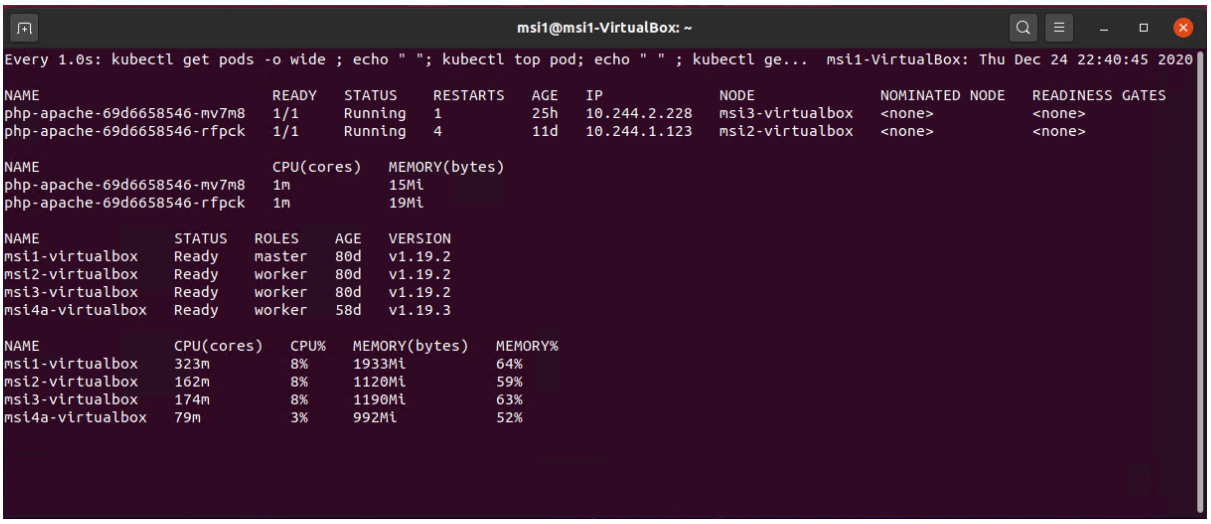
	Queue			Session rate					
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Li
msi2-virtualbox	0	0	-	0	1		0	1	
msi3-virtualbox	0	0	-	0	1		0	1	
msi4-virtualbox	0	0	-	1	1		1	1	

Figure 25: Two nodes are down, and traffic directs to msi4 node

6.2 HIGH AVAILABILITY TEST

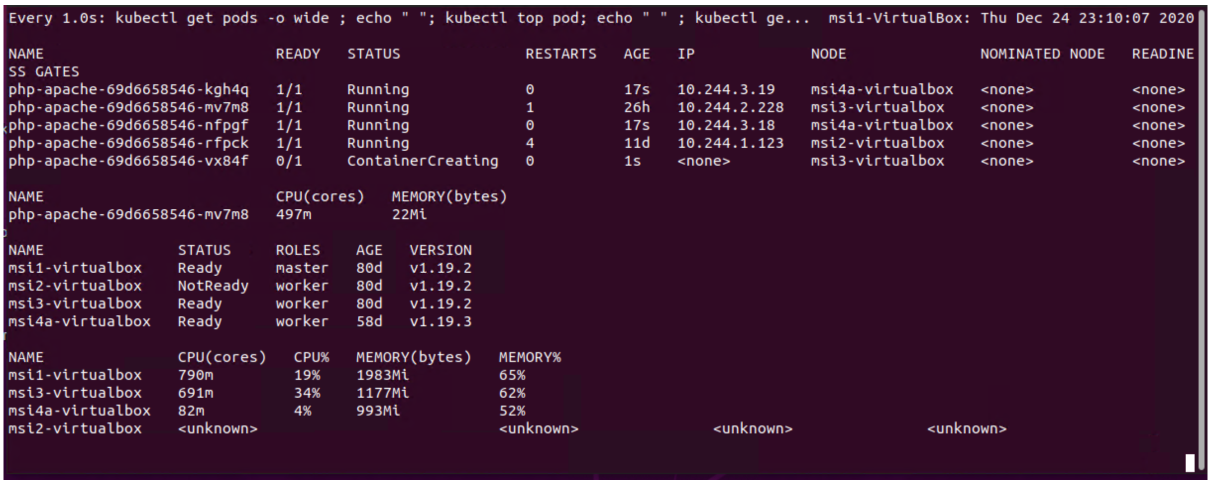
One deployment will create a High Pod Autoscaler and another deployment will not have HPA. The pods cannot be scaled up when HPA is not configured. When load is applied to Deployment, the pods should scale up depending on CPU usage.

When scaled up to high number of replicas, the pods can switch over to healthy pod when a pod fails. When node fails, it should be able to switch to other healthy node at minimum downtime.



```
Every 1.0s: kubectl get pods -o wide ; echo " "; kubectl top pod; echo " " ; kubectl ge...  msi1-VirtualBox: Thu Dec 24 22:40:45 2020

NAME                          READY   STATUS    RESTARTS   AGE   IP             NODE              NOMINATED NODE   READINESS GATES
php-apache-69d6658546-mv7m8   1/1     Running   1          25h   10.244.2.228   msi3-virtualbox   <none>           <none>
php-apache-69d6658546-rfpck   1/1     Running   4          11d   10.244.1.123   msi2-virtualbox   <none>           <none>

NAME                          CPU(cores)   MEMORY(bytes)
php-apache-69d6658546-mv7m8   1m           15Mi
php-apache-69d6658546-rfpck   1m           19Mi

NAME               STATUS   ROLES    AGE   VERSION
msi1-virtualbox    Ready    master   80d   v1.19.2
msi2-virtualbox    Ready    worker   80d   v1.19.2
msi3-virtualbox    Ready    worker   80d   v1.19.2
msi4a-virtualbox   Ready    worker   58d   v1.19.3

NAME               CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
msi1-virtualbox    323m         8%     1933Mi          64%
msi2-virtualbox    162m         8%     1120Mi          59%
msi3-virtualbox    174m         8%     1190Mi          63%
msi4a-virtualbox   79m          3%     992Mi           52%
```

Figure 26: Viewing statistics of pods and nodes



```
Every 1.0s: kubectl get pods -o wide ; echo " "; kubectl top pod; echo " " ; kubectl ge...  msi1-VirtualBox: Thu Dec 24 23:10:07 2020

NAME                          READY   STATUS              RESTARTS   AGE   IP             NODE               NOMINATED NODE   READINE
SS GATES
php-apache-69d6658546-kgh4q   1/1     Running             0          17s   10.244.3.19    msi4a-virtualbox   <none>           <none>
php-apache-69d6658546-mv7m8   1/1     Running             1          26h   10.244.2.228   msi3-virtualbox    <none>           <none>
php-apache-69d6658546-nfpgf   1/1     Running             0          17s   10.244.3.18    msi4a-virtualbox   <none>           <none>
php-apache-69d6658546-rfpck   1/1     Running             4          11d   10.244.1.123   msi2-virtualbox    <none>           <none>
php-apache-69d6658546-vx84f   0/1     ContainerCreating   0          1s    <none>         msi3-virtualbox    <none>           <none>

NAME                          CPU(cores)   MEMORY(bytes)
php-apache-69d6658546-mv7m8   497m         22Mi

NAME               STATUS     ROLES    AGE   VERSION
msi1-virtualbox    Ready      master   80d   v1.19.2
msi2-virtualbox    NotReady   worker   80d   v1.19.2
msi3-virtualbox    Ready      worker   80d   v1.19.2
msi4a-virtualbox   Ready      worker   58d   v1.19.3

NAME               CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
msi1-virtualbox    790m         19%    1983Mi          65%
msi3-virtualbox    691m         34%    1177Mi          62%
msi4a-virtualbox   82m          4%     993Mi           52%
msi2-virtualbox    <unknown>                           <unknown>               <unknown>               <unknown>
```

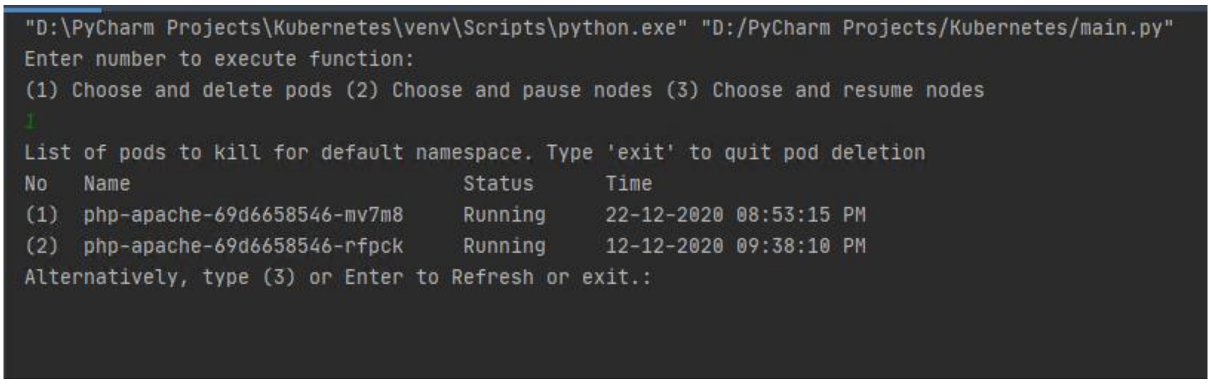
Figure 27: Pods creating when load increases

6.3 SIMULATE FAILURE VIA API CALL

The application will have almost no or minimal interruption after failure of pods or nodes. The server still able to respond to query after interruption and recovery.

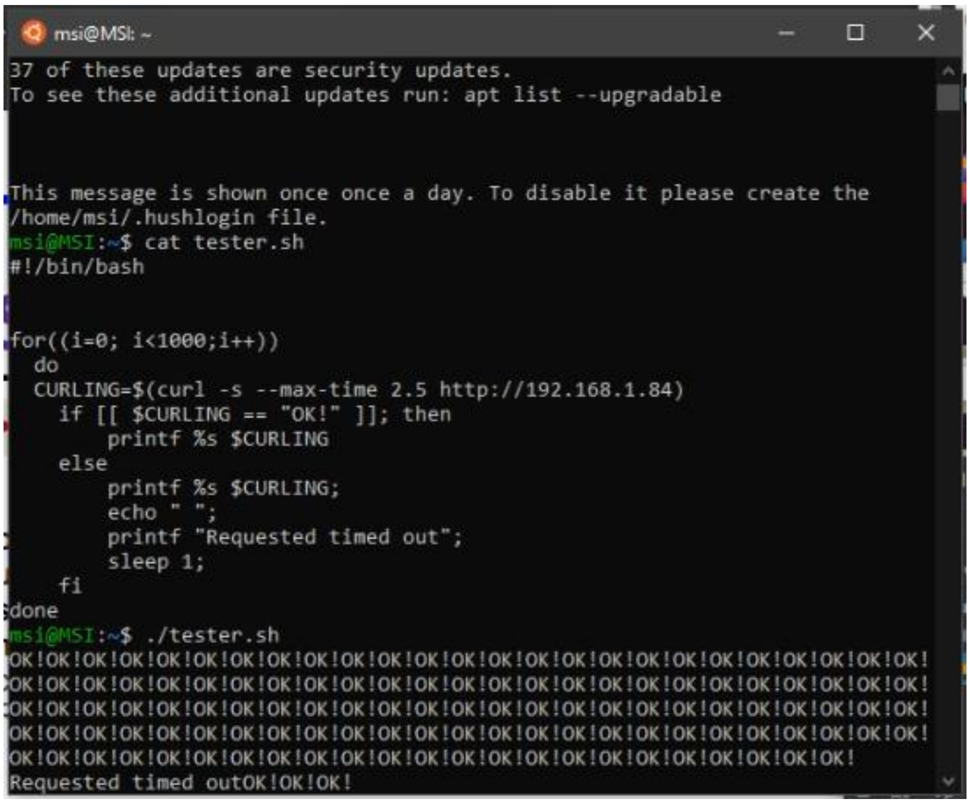
6.3.1 DELETING POD (SIMULATE LOGICAL FAILURE)

Calling an API from Kubernetes client to delete the pod, one by one, and the pods will resurrect again, which show self healing capabilities. The pods should restore back to desired number of replicas. Minor downtime should be noticed.



```
"D:\PyCharm Projects\Kubernetes\venv\Scripts\python.exe" "D:/PyCharm Projects/Kubernetes/main.py"
Enter number to execute function:
(1) Choose and delete pods (2) Choose and pause nodes (3) Choose and resume nodes
1
List of pods to kill for default namespace. Type 'exit' to quit pod deletion
No   Name                            Status     Time
(1)  php-apache-69d6658546-mv7m8     Running    22-12-2020 08:53:15 PM
(2)  php-apache-69d6658546-rfpck     Running    12-12-2020 09:38:10 PM
Alternatively, type (3) or Enter to Refresh or exit.:
```

Figure 28: Inside section (1) to kill pods

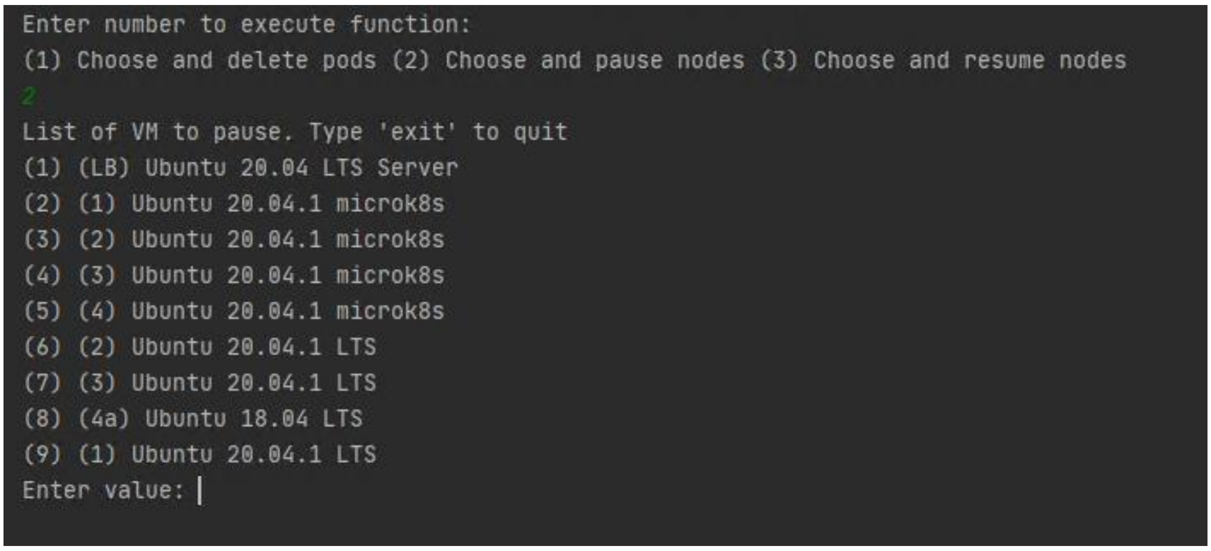


```
msi@MSI: ~
37 of these updates are security updates.
To see these additional updates run: apt list --upgradable


This message is shown once once a day. To disable it please create the
/home/msi/.hushlogin file.
msi@MSI:~$ cat tester.sh
#!/bin/bash


for((i=0; i<1000;i++))
  do
  CURLING=$(curl -s --max-time 2.5 http://192.168.1.84)
    if [[ $CURLING == "OK!" ]]; then
        printf %s $CURLING
    else
        printf %s $CURLING;
        echo " ";
        printf "Requested timed out";
        sleep 1;
    fi
done
msi@MSI:~$ ./tester.sh
OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!
OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!
OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!
OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!
OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!
Requested timed outOK!OK!OK!
```

Figure 29: Requested timed out should be less and able to forward traffic when pods deleting

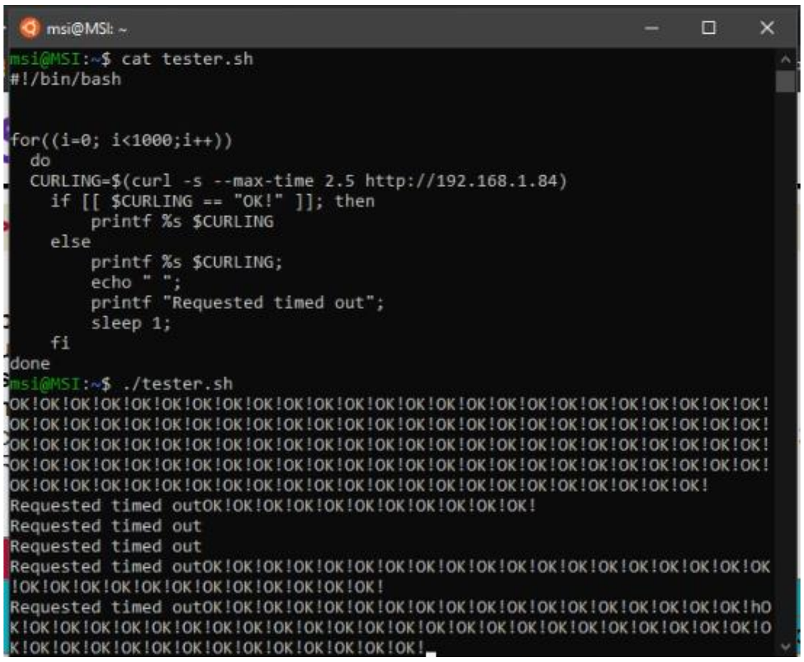
6.3.2 PAUSING NODE (SIMULATE PHYSICAL FAILURE)

Calling Virtualbox API that communicates via Microsoft COM to pause the worker nodes. (2) Ubuntu 20.04 LTS, (3) Ubuntu 20.04 LTS and (4) Ubuntu 20.04 LTS. When a node goes down, the load balancer should able to detect down node and switch to other worker nodes. The traffic can reach into the pods. Minimal down time should be noticed. Node failure can also be identified with HAProxy statistics page.



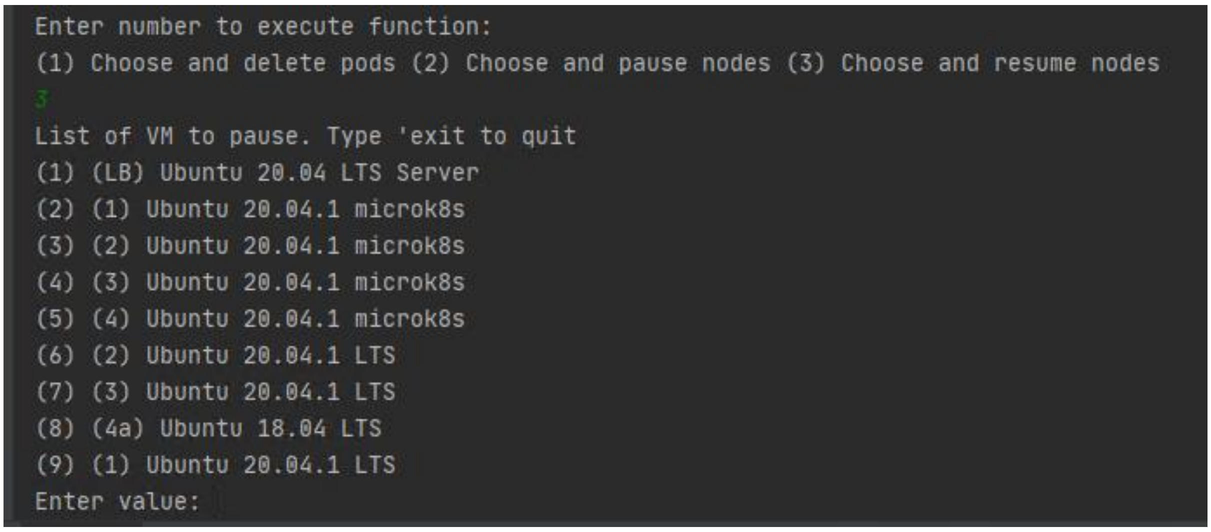
```
Enter number to execute function:
(1) Choose and delete pods (2) Choose and pause nodes (3) Choose and resume nodes
2
List of VM to pause. Type 'exit' to quit
(1) (LB) Ubuntu 20.04 LTS Server
(2) (1) Ubuntu 20.04.1 microk8s
(3) (2) Ubuntu 20.04.1 microk8s
(4) (3) Ubuntu 20.04.1 microk8s
(5) (4) Ubuntu 20.04.1 microk8s
(6) (2) Ubuntu 20.04.1 LTS
(7) (3) Ubuntu 20.04.1 LTS
(8) (4a) Ubuntu 18.04 LTS
(9) (1) Ubuntu 20.04.1 LTS
Enter value: |
```

Figure 30: Inside section (2) to pause nodes



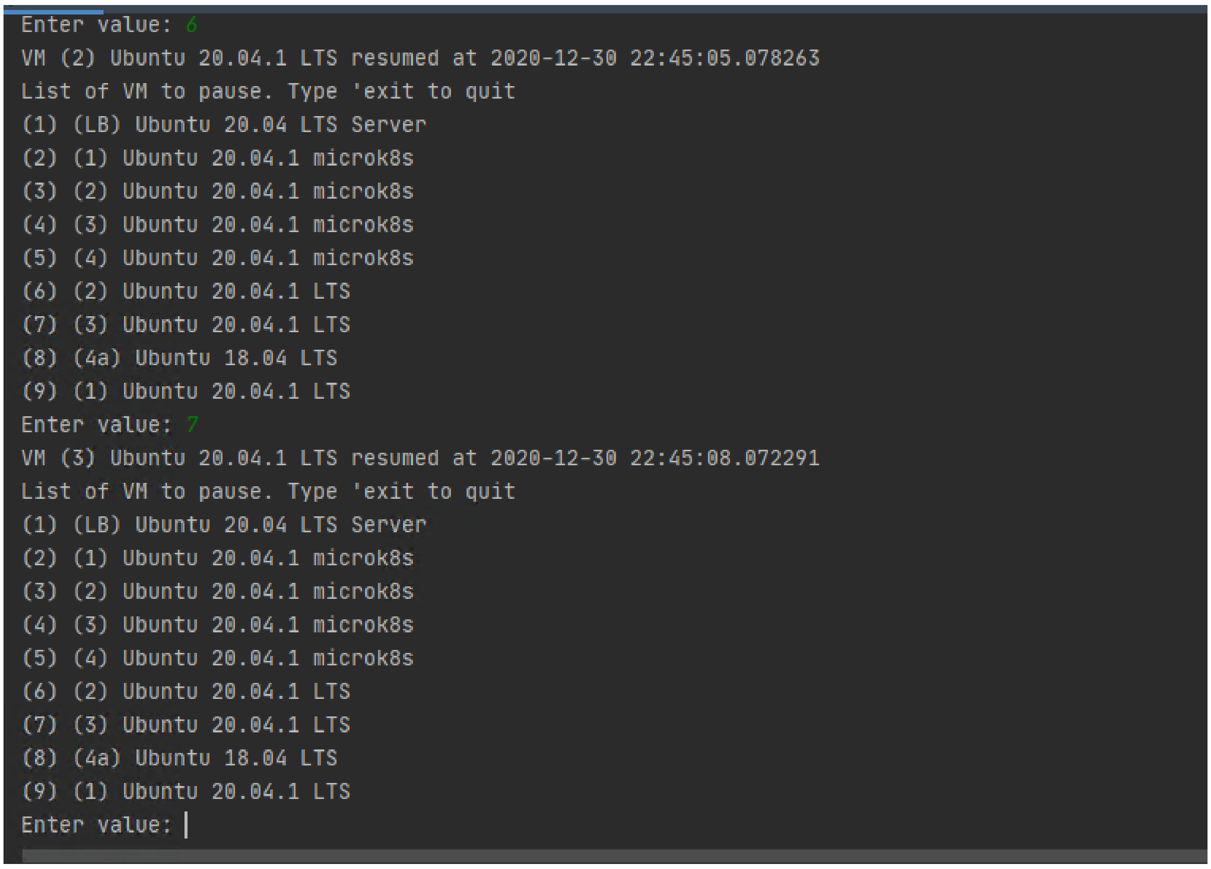
```
msi@MSI: ~
msi@MSI:~$ cat tester.sh
#!/bin/bash


for((i=0; i<1000;i++))
  do
  CURLING=$(curl -s --max-time 2.5 http://192.168.1.84)
    if [[ $CURLING == "OK!" ]]; then
        printf %s $CURLING
    else
        printf %s $CURLING;
        echo " ";
        printf "Requested timed out";
        sleep 1;
    fi
done
msi@MSI:~$ ./tester.sh
OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!
OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!
OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!
OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!
OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!
Requested timed outOK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!
Requested timed out
Requested timed out
Requested timed outOK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK
!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!
Requested timed outOK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!hO
K!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!O
K!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!
```

Figure 31: Requested timed out and downtime should be reasonable and able to resume sending traffic afterwards



```
Enter number to execute function:
(1) Choose and delete pods (2) Choose and pause nodes (3) Choose and resume nodes
3
List of VM to pause. Type 'exit to quit
(1) (LB) Ubuntu 20.04 LTS Server
(2) (1) Ubuntu 20.04.1 microk8s
(3) (2) Ubuntu 20.04.1 microk8s
(4) (3) Ubuntu 20.04.1 microk8s
(5) (4) Ubuntu 20.04.1 microk8s
(6) (2) Ubuntu 20.04.1 LTS
(7) (3) Ubuntu 20.04.1 LTS
(8) (4a) Ubuntu 18.04 LTS
(9) (1) Ubuntu 20.04.1 LTS
Enter value:
```

Figure 32: Inside section (3) to resume nodes

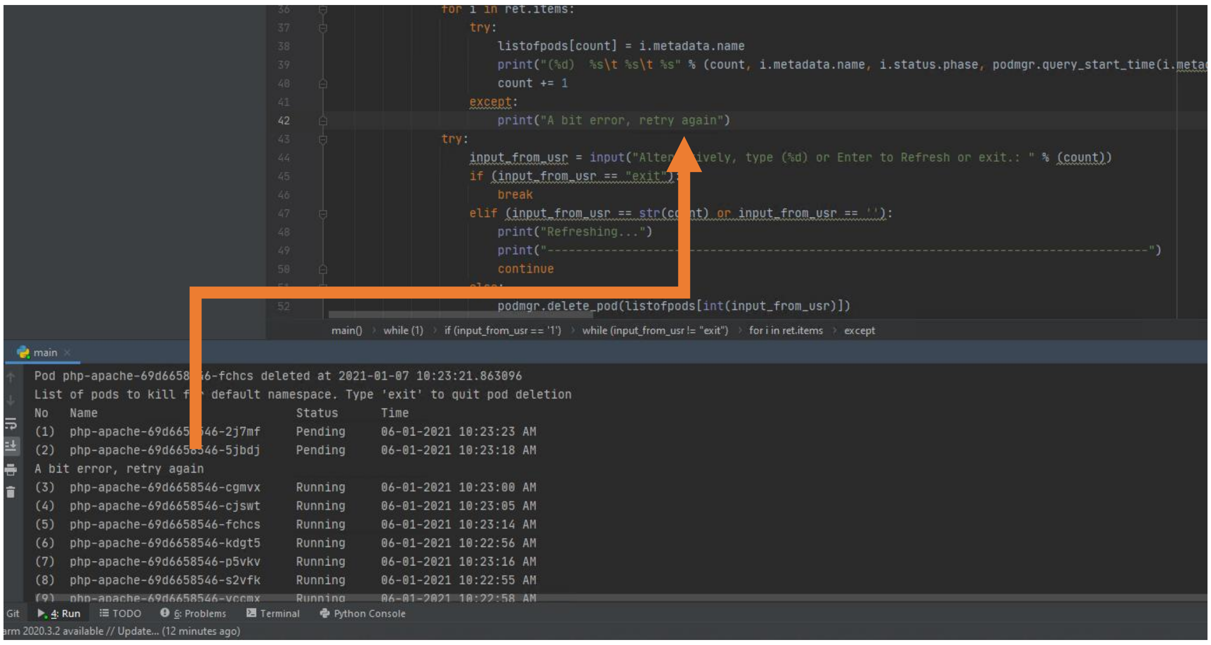


```
Enter value: 6
VM (2) Ubuntu 20.04.1 LTS resumed at 2020-12-30 22:45:05.078263
List of VM to pause. Type 'exit to quit
(1) (LB) Ubuntu 20.04 LTS Server
(2) (1) Ubuntu 20.04.1 microk8s
(3) (2) Ubuntu 20.04.1 microk8s
(4) (3) Ubuntu 20.04.1 microk8s
(5) (4) Ubuntu 20.04.1 microk8s
(6) (2) Ubuntu 20.04.1 LTS
(7) (3) Ubuntu 20.04.1 LTS
(8) (4a) Ubuntu 18.04 LTS
(9) (1) Ubuntu 20.04.1 LTS
Enter value: 7
VM (3) Ubuntu 20.04.1 LTS resumed at 2020-12-30 22:45:08.072291
List of VM to pause. Type 'exit to quit
(1) (LB) Ubuntu 20.04 LTS Server
(2) (1) Ubuntu 20.04.1 microk8s
(3) (2) Ubuntu 20.04.1 microk8s
(4) (3) Ubuntu 20.04.1 microk8s
(5) (4) Ubuntu 20.04.1 microk8s
(6) (2) Ubuntu 20.04.1 LTS
(7) (3) Ubuntu 20.04.1 LTS
(8) (4a) Ubuntu 18.04 LTS
(9) (1) Ubuntu 20.04.1 LTS
Enter value:
```

Figure 33: Execute resume nodes

6.4 ERROR HANDLING

Error handling includes Kubernetes self-healing pods, HAProxy backend server redirect and Python API try-catch-except.



```python
            for i in ret.items:
                try:
                    listofpods[count] = i.metadata.name
                    print("(%d)  %s\t %s\t %s" % (count, i.metadata.name, i.status.phase, podmgr.query_start_time(i.metad
                    count += 1
                except:
                    print("A bit error, retry again")
            try:
                input_from_usr = input("Alternatively, type (%d) or Enter to Refresh or exit.: " % (count))
                if (input_from_usr == "exit"):
                    break
                elif (input_from_usr == str(count) or input_from_usr == ''):
                    print("Refreshing...")
                    print("-----------------------------------------------------------------------------------------")
                    continue
                else:
                    podmgr.delete_pod(listofpods[int(input_from_usr)])
```

main()
while (1)
if (input_from_usr == '1')
while (input_from_usr != "exit")
for i in ret.items
except
main
Git
4: Run
TODO
6: Problems
Terminal
Python Console
arm 2020.3.2 available // Update... (12 minutes ago)

```
Pod php-apache-69d6658546-fchcs deleted at 2021-01-07 10:23:21.863096
List of pods to kill for default namespace. Type 'exit' to quit pod deletion
No   Name                              Status      Time
(1)  php-apache-69d6658546-2j7mf       Pending     06-01-2021 10:23:23 AM
(2)  php-apache-69d6658546-5jbdj       Pending     06-01-2021 10:23:18 AM
A bit error, retry again
(3)  php-apache-69d6658546-cgmvx       Running     06-01-2021 10:23:00 AM
(4)  php-apache-69d6658546-cjswt       Running     06-01-2021 10:23:05 AM
(5)  php-apache-69d6658546-fchcs       Running     06-01-2021 10:23:14 AM
(6)  php-apache-69d6658546-kdgt5       Running     06-01-2021 10:22:56 AM
(7)  php-apache-69d6658546-p5vkv       Running     06-01-2021 10:23:16 AM
(8)  php-apache-69d6658546-s2vfk       Running     06-01-2021 10:22:55 AM
(9)  php-apache-69d6658546-vccmx       Running     06-01-2021 10:22:58 AM
```

Figure 34: When API fails to call, it will show an error during failure simulation.

7.0 CONCLUSION

At the end of this project, a highly scalable, load balanced and high availability server will be developed. Auto scaling function HPA shows scaling of pods when load increases and a load balancer distributes traffic by round robin method and redirects traffic to other nodes when a node failed. Simulating Kubernetes failures successfully performed by deleting pods and pausing/resuming nodes, while maintaining server uptime.

REFERENCES

Production-Grade Container Orchestration. (n.d.). Retrieved from https://kubernetes.io/

L. Abdollahi Vayghan, M. A. Saied, M. Toeroe and F. Khendek, "Deploying Microservice Based Applications with Kubernetes: Experiments and Lessons Learned," 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), San Francisco, CA, 2018, pp. 970-973.

Kim, Dongmin & Muhammad, Hanif & Kim, Eunsam & Helal, Sumi & Lee, Choonhwa. (2019). TOSCA-Based and Federation-Aware Cloud Orchestration for Kubernetes Container Platform. Applied Sciences.

Takahashi, Kimitoshi & Aida, Kento & Tanjo, Tomoya & Sun, Hiroshi. (2018). A Portable Load Balancer for Kubernetes Cluster. 222-231. 10.1145/3149457.3149473.

Technologies, V. (2019, October 15). Demystifying High Availability in Kubernetes Using Kubeadm. Retrieved from https://medium.com/velotio-perspectives/demystifying-high-availability-in-kubernetes-using-kubeadm-3d83ed8c458b

HAProxy Technologies (2020, November 02). Retrieved from https://www.haproxy.com/