***Project Report***

***on***

***Kubernetes cluster with High Availability***

***and***

***Scalability of web server.***

******

*Submitted in partial fulfillment for the award of*

***Post Graduate Diploma in High Performance Computing System***

***Administration*** *from* ***C-DAC ACTS (Pune)***

***Guidedby: Ms.Tejaswini Apate***

*Presented by :*

***Miss Pranjali Patil (****230940127009****)***

***Miss Sushmita Diwakar (****230940127051****)***

### *Miss Sukanya Mane (230940127005)*

### *Mr. Trishna Dhruw (230940127053)*

***Mr. Tarun Shori (****230940127052****)***

# *Centre of Development of Advanced Computing (C-DAC), Pune*



CERTIFICATE

# *TO WHOMSOEVER IT MAY CONCERN*

***This is to certify that***

***Miss Pranjali Patil***

***Miss Sushmita Diwakar***

### *Miss Sukanya Mane*

### *Mr. Trishna Dhruw*

***Mr. Tarun Shori***

***have successfully completed their project on***

***Kubernetes cluster with High***

***Availability and scalability of web server***

***Under the Guidance of Ms. Tejaswini apate***

***Project Guide Project Supervisor***

# *HOD ACTS*

***Mr.***

****

PG-DHPCSA

ACKNOWLEDGEMENT

This project “ Kubernetes cluster with High Availability and scalability of web server**”** was a

great learning experience for us and we are submitting this work to Advanced

Computing Training School (CDAC ACTS).

We all are very glad to mention the name of Ms.Tejaswini apate for his valuable guidance to

work on this project. Overcome various obstacles and intricacies during the course of project work.

We are highly grateful to HPC tech team (ACTS training Centre, C- DAC), For his valuable

guidance and support whenever necessary while doing this course Post Graduate Diploma in

High Performance Computing System Administration (PG- DHPCSA) Through CDAC

ACTS , Pune.

Our most heartfelt thank goes to Ms. Swati salunkhe (Course Coordinator, PG- DHPCSA)

who gave all the required support and kind coordination to provide all the necessities like

required hardware, internet facility and extra Lab hours to complete the project and throughout

the course up to the last day here in C-DAC ACTS, Pune.

### *From:*

***Miss Pranjali Patil (****230940127009****)***

***Miss Sushmita Diwakar (****230940127051****)***

### *Miss Sukanya Mane (230940127005)*

### *Mr. Trishna Dhruw (230940127053)*

***Mr. Tarun Shori (****230940127052****)***

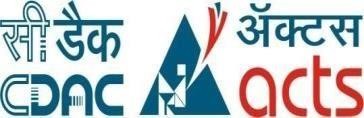


TABLE OF CONTENTS

1. Abstract
2. Introduction
3. Use Cases
4. Work Flow
5. System Requirements
6. Software Requirements
7. Setting Up The AWS Instances
8. Connect The Instances Via SSH
9. Kubernetes Installation
10. Create Docker Image And Push To Docker Hub
11. Create Persistent Volume And Persistent Volume Claim
12. Create Deployment And Services
13. Apply YAML Manifest For The “My-Web”
14. Kubernetes Dashboard Setup
15. Validation
16. References And Bibliography
17. Project Link
18. Limitations
19. Conclusion

Abstract

This project focuses on deploying and managing a highly available web service on a Kubernetes cluster hosted on the AWS platform. The web service is built using the Apache HTTP Server (httpd) and is deployed using Kubernetes Deployments, Services, Persistent Volumes, and Persistent Volume Claims. The project aims to demonstrate the high availability and scalability features provided by Kubernetes in a real-world scenario.

The key components of the project include:

1. Setting up an AWS Kubernetes cluster: The project starts by provisioning an AWS Kubernetes cluster using tools self-managed Kubernetes on AWS EC2 instances.

2. Deploying the web service: The Apache HTTP Server (httpd) is containerized and deployed as a Kubernetes Deployment. Multiple replicas of the httpd Deployment are created to ensure high availability.

3. Configuring Persistent Volumes: Persistent Volumes (PVs) and Persistent Volume Claims (PVCs) are used to provide persistent storage for the web service. PVs are configured to store data on AWS EBS volumes or other suitable storage solutions.

4. Scaling the web service: The project demonstrates how to scale the number of replicas of the httpd Deployment dynamically based on demand. This showcases Kubernetes' ability to handle increased traffic and ensure consistent performance.

6. Testing high availability: To test the high availability of the web service, the project includes scripts to simulate node failures in the Kubernetes cluster. The scripts drain and uncordon nodes to observe how Kubernetes manages pod scheduling and maintains service availability.

Overall, this project serves as a practical demonstration of deploying and managing a highly available web service on Kubernetes using AWS infrastructure. It highlights the benefits of using Kubernetes for container orchestration and showcases its capabilities in ensuring service availability, scalability, and resilience.

Introduction

In today's era of cloud-native applications, ensuring high availability and scalability of web services is crucial for businesses to meet the demands of their users while maintaining reliability and performance. Kubernetes, as a powerful container orchestration platform, provides robust solutions for deploying, managing, and scaling applications in a cloud-native environment.

This This project entails the creation of a robust Kubernetes cluster deployed on AWS instances. Leveraging Docker containers within Kubernetes pods, the Apache Web server is orchestrated to ensure both high availability and scalability. Through meticulous configuration and management, this deployment architecture promises to elevate the reliability and efficiency of web services while maximizing resource utilization.

Use Cases

1. Implement High Availability:

- Configured Kubernetes Pod Replication Controllers or Deployments to ensure redundancy.

2. Enable Scaling:

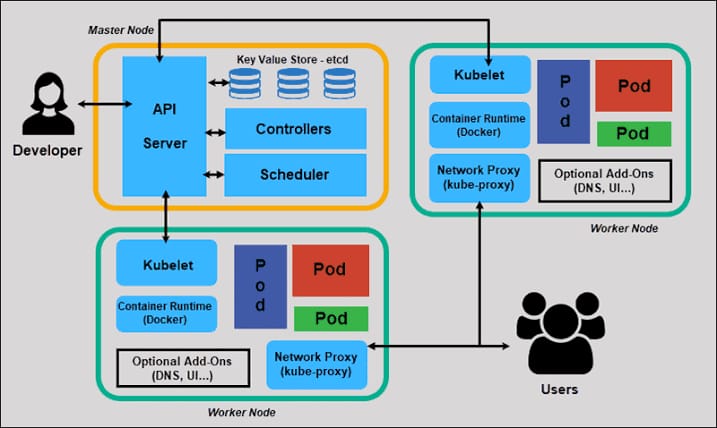
- Implement Kubernetes Horizontal Pod Autoscaler (HPA) to automatically scale the web server based on demand.

- Adjust HPA settings for CPU or memory utilization thresholds.

3. Configure Load Balancing:

- Used Kubernetes Services to expose the web server, automatically creating an internal load balancer.

Workflow



System Requirements

**For all the Nodes :**

* **RAM: 4 GB**
* **Storage: 15 GB**
* **Processors: 2 cores**
* **OS : AWS Linux**

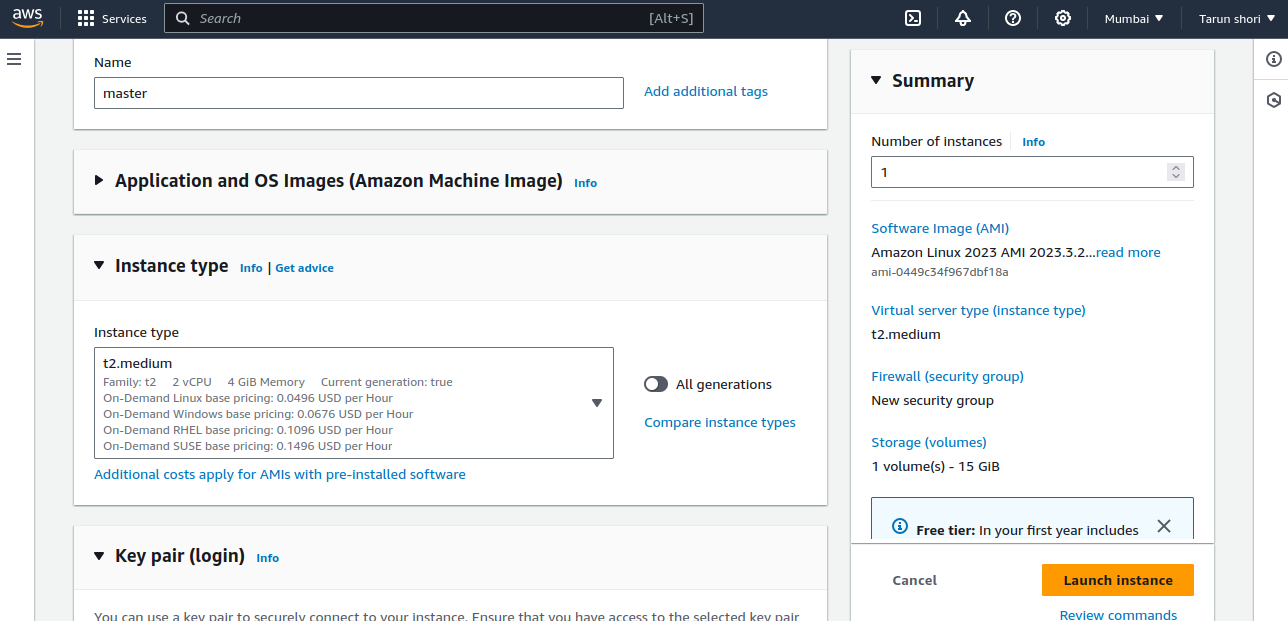
Software Requirements

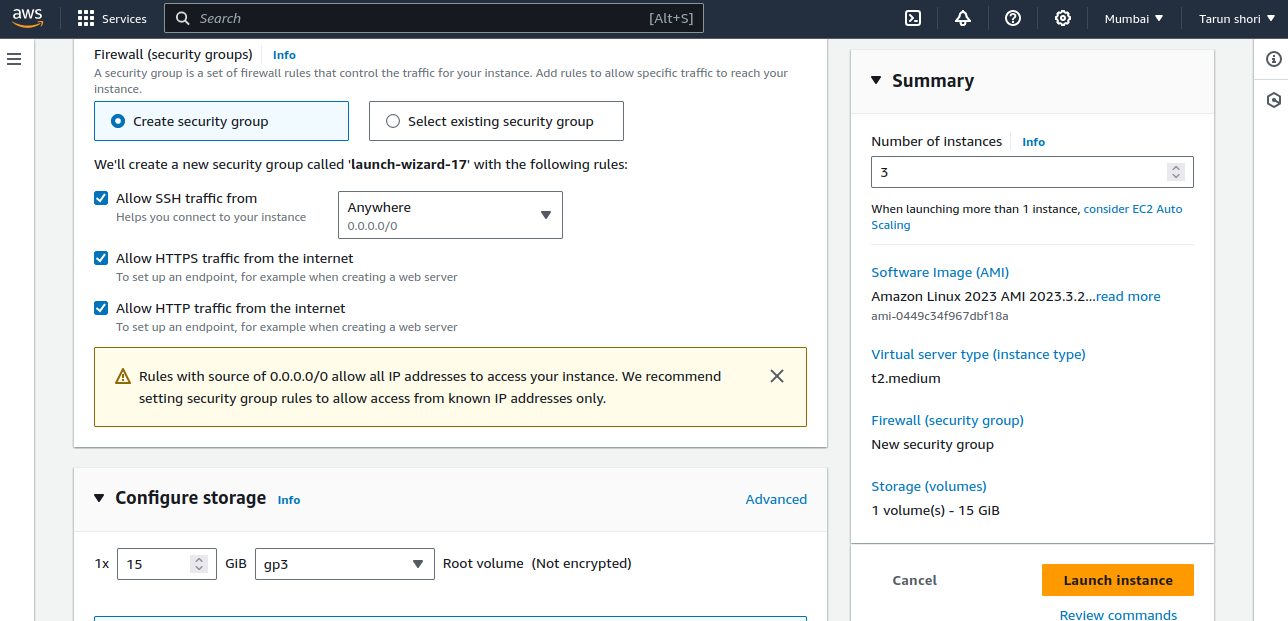
* Kubernetes Tool - Kubeadm
* Container Runtime – Containerd
* Networking Plugin- Calico
* Repository – Docker hub
* Web server -Apache2 – Httpd
* Cloud Provider - AWS

Setting up the AWS Instances

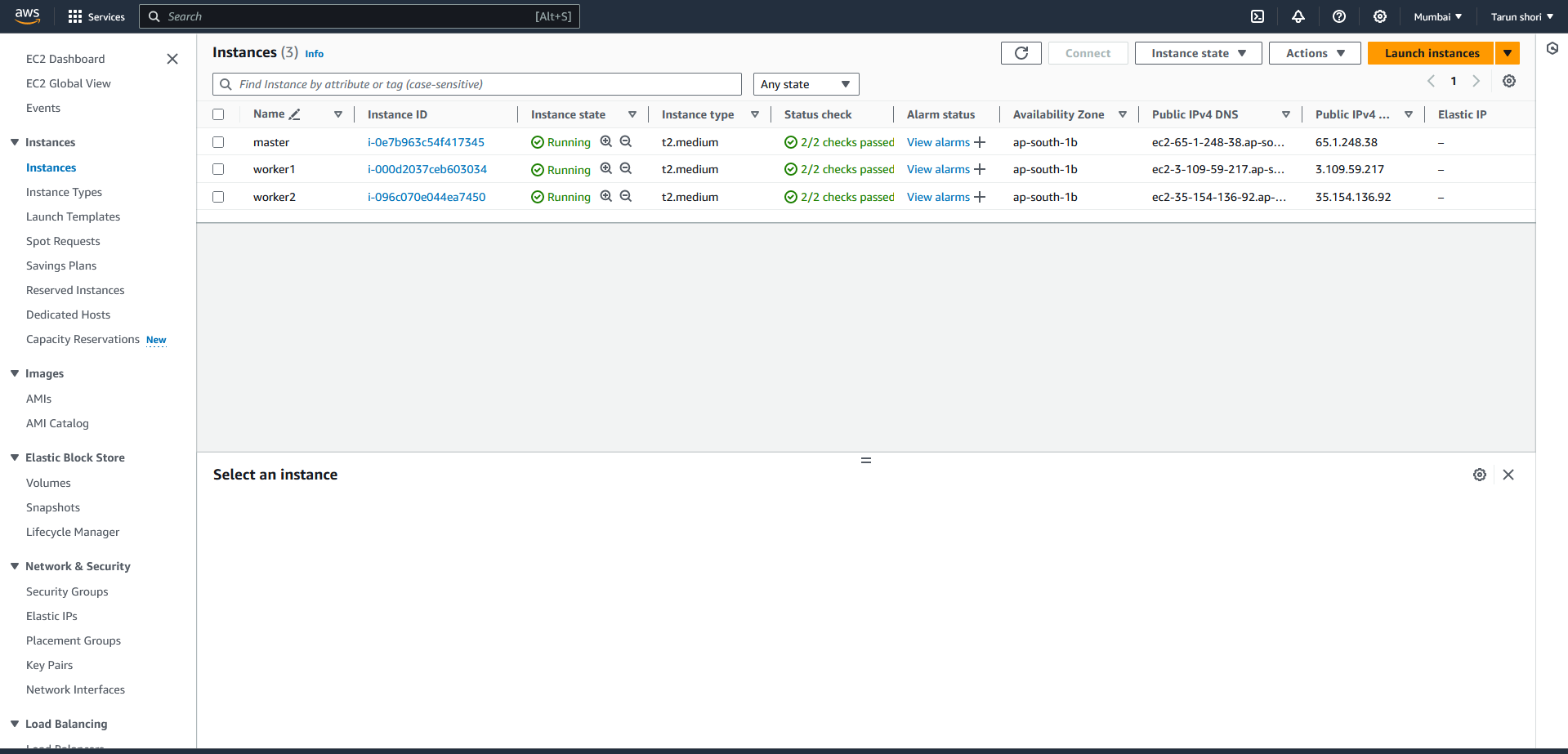
* create a Ec2 instance for master and worker nodes

Step 1: configure all the necessary configurations for Ec2 instances



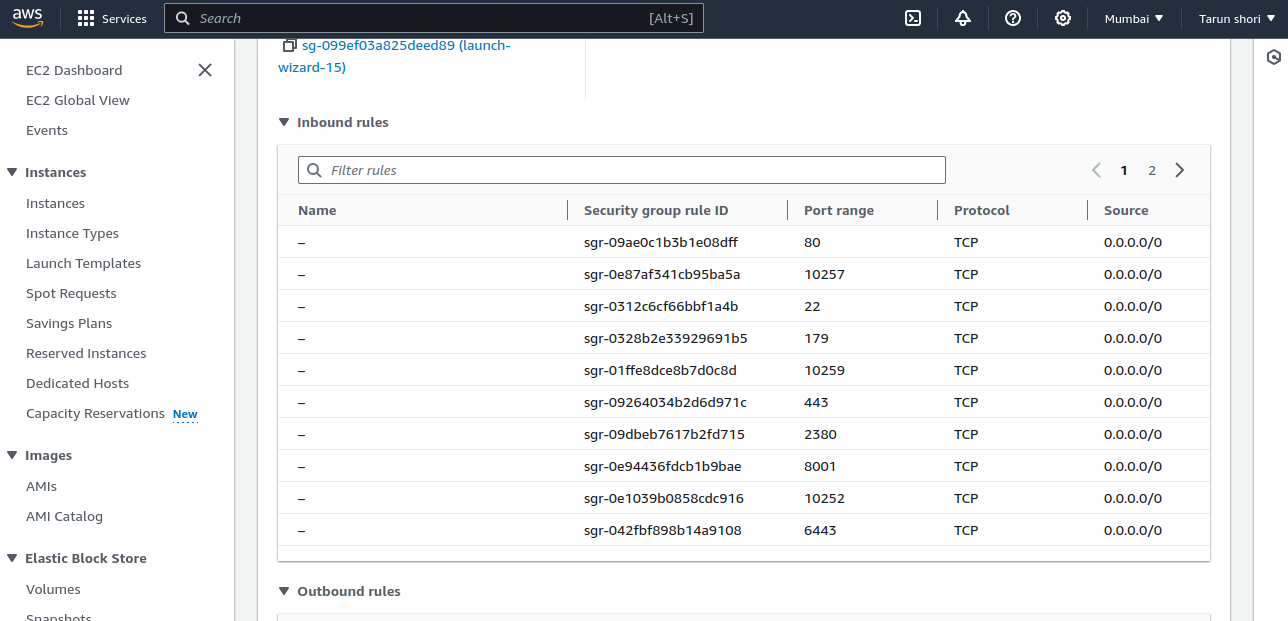


Step 2: Final step is to confirm the configuration and click Launch instance

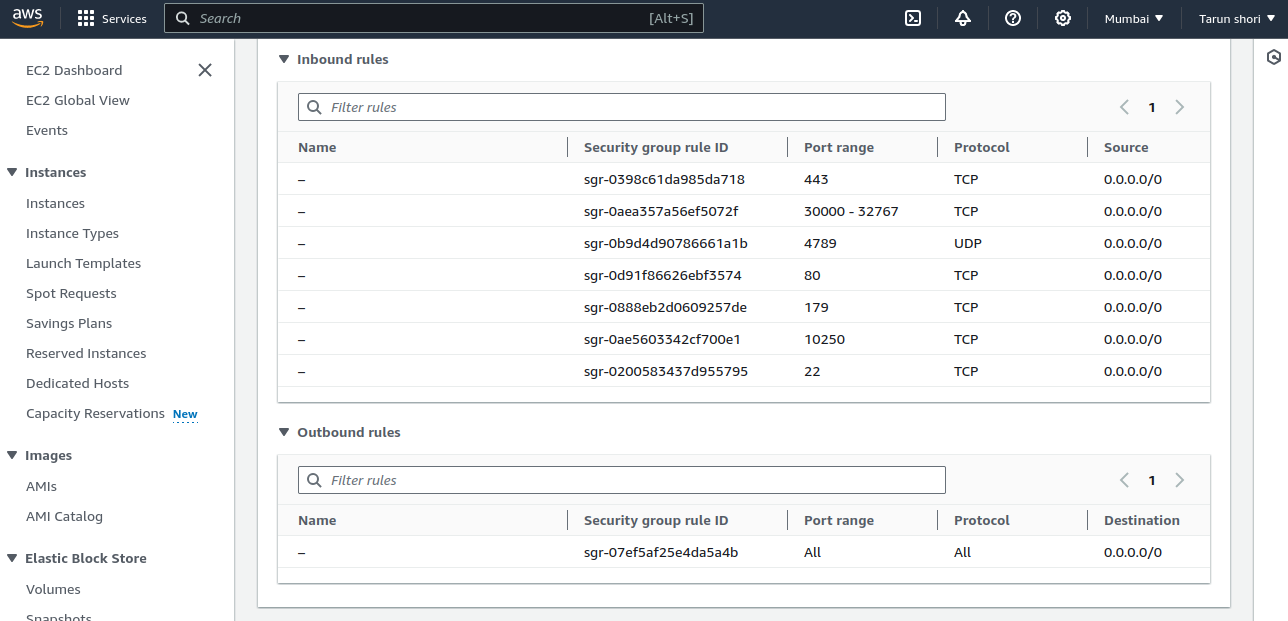


Step 3: Create Security group for instances for master and worker nodes.

For master node :



For worker nodes :



Connect the instances via SSH



Kubernetes Installation

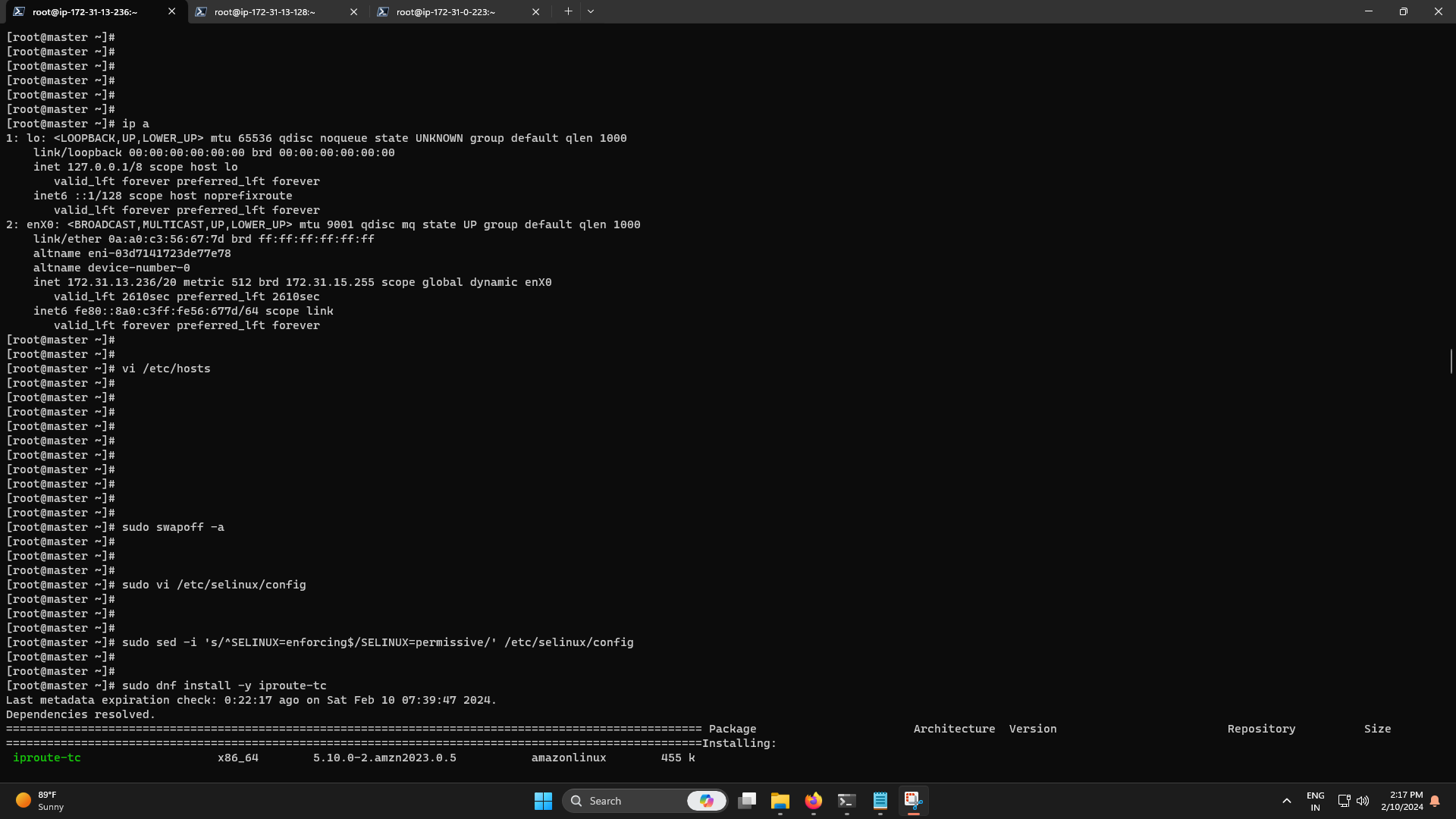
NOTE: Steps 1 to 6 should be applied to both the Master and the worker node.

## Step 1) Disable swap space

For best performance, Kubernetes requires that swap is disabled on the host system. This is because memory swapping can significantly lead to instability and performance degradation.

To disable swap space, run the command:

$ sudo swapoff -a



To make the changes persistent, edit the /etc/fstab file and remove or comment out the line with the swap entry and save the changes.

$ sudo sed -i '/swap/d' /etc/fstab



## Step 2) Disable SELinux

Additionally, we need to disable SELinux and set it to ‘permissive’ in order to allow smooth communication between the nodes and the pods.

To achieve this, open the SELinux configuration file.

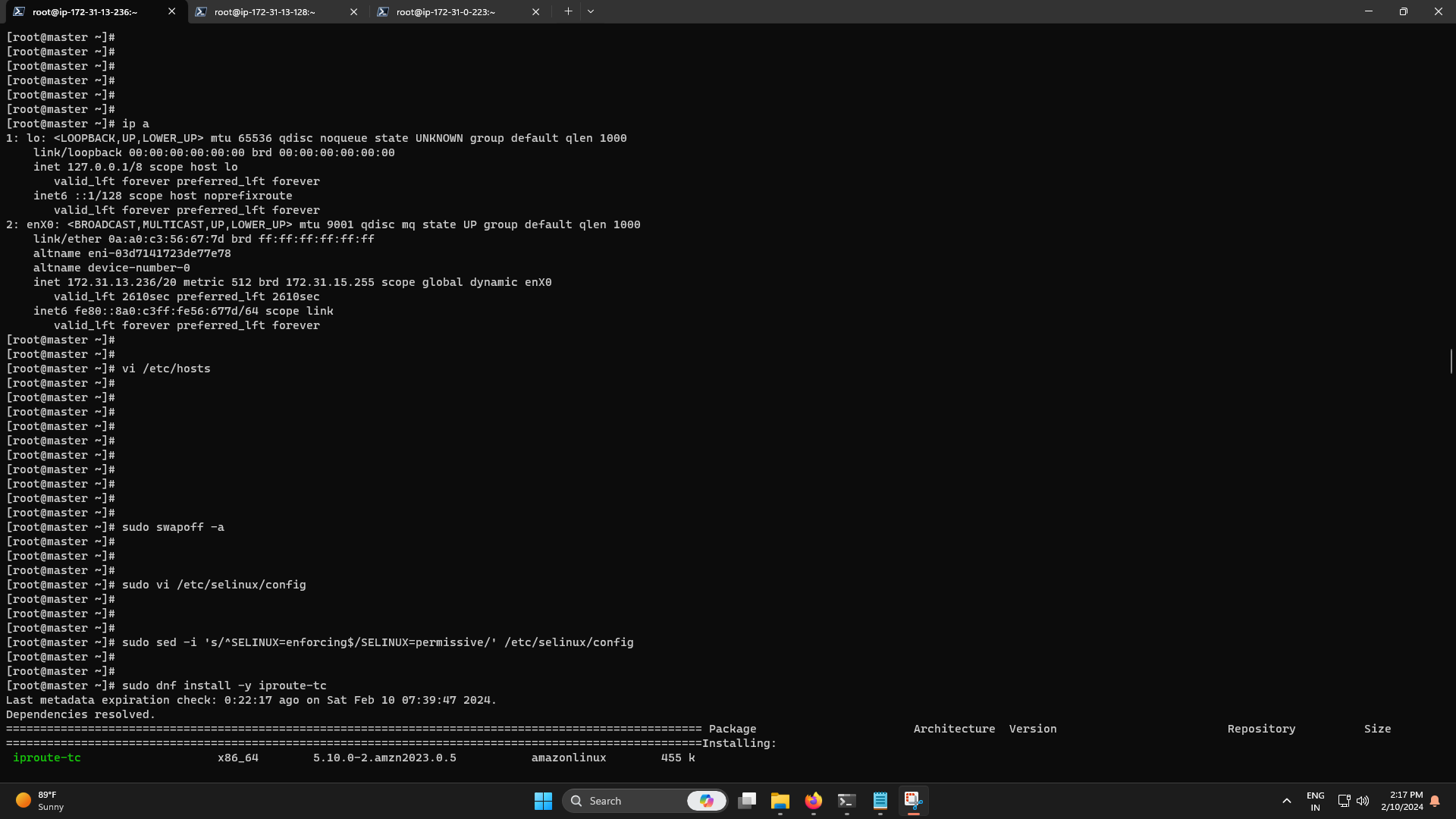
$ sudo vi /etc/selinux/config

Change the SELINUX value from enforcing to permissive.

SELINUX=permissive

Alternatively, you use the sed command as follows.

$ sudo sed -i 's/^SELINUX=enforcing$/SELINUX=permissive/' /etc/selinux/config



## Step 3) Configure networking in master and worker node

Some additional network configuration is required for your master and worker nodes to communicate effectively. On each node, edit the  /etc/hosts file.

$ sudo vi /etc/hosts

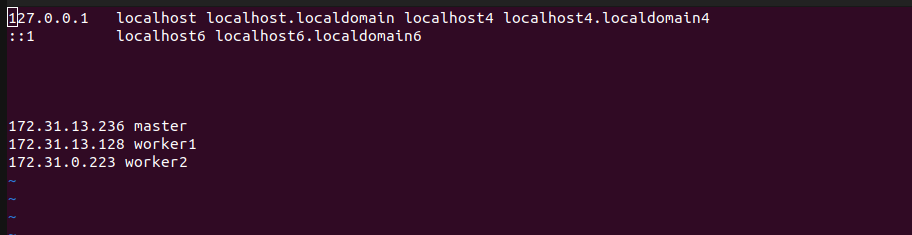
Next, update the entries as shown

172.31.13.236 master // For the Master node

172.31.13.128 worker1 // For the Worker node

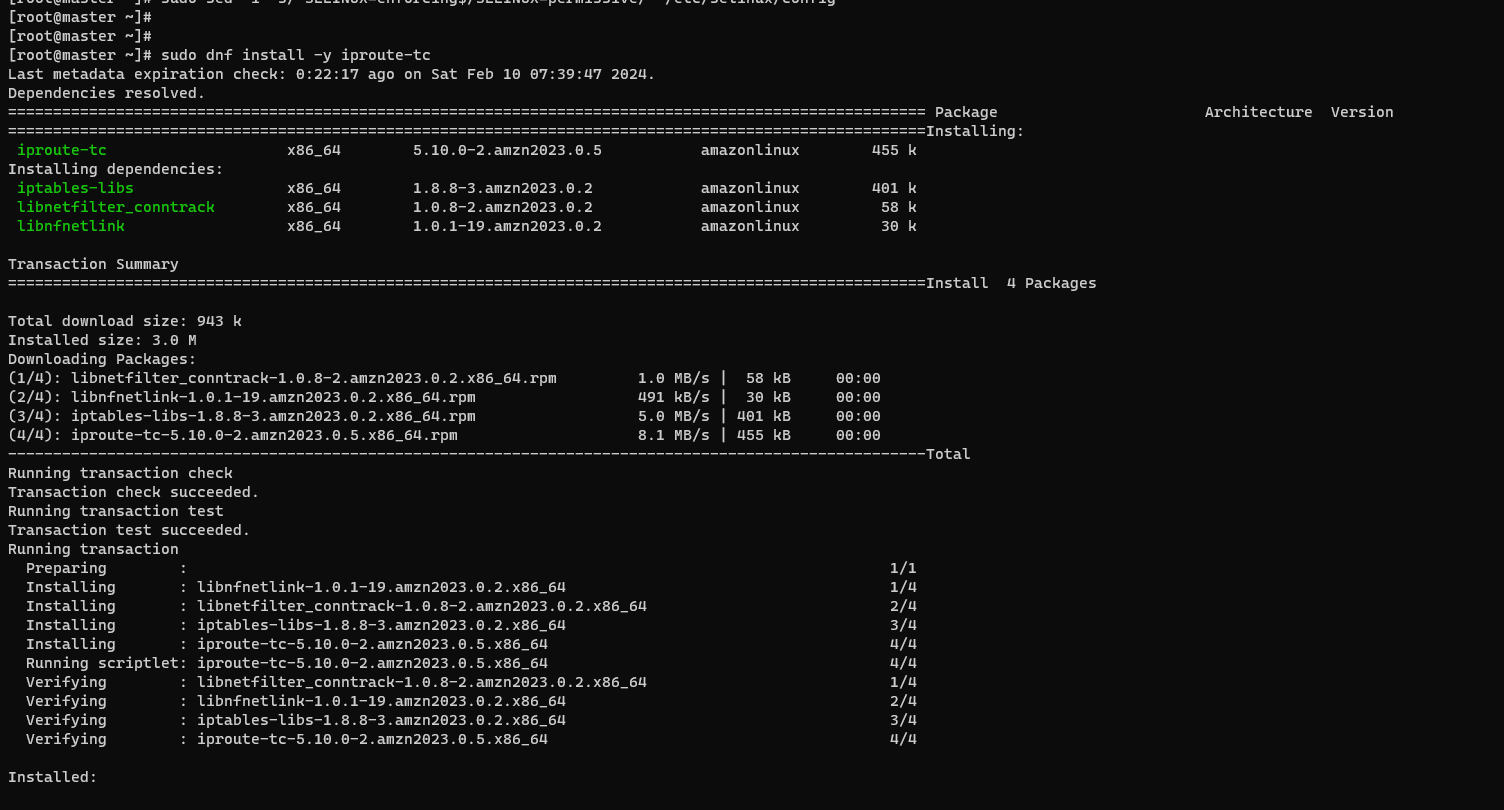
172.31.0.223 worker2 // For the Worker node





Save and exit the configuration file. Next, install the traffic control utility package:

$ sudo dnf install -y iproute-tc



For seamless communication between the Master and worker node, configure the firewall and allow some pertinent ports and services as outlined below.

On Master node, allow following ports,

$ sudo firewall-cmd --permanent –add-port=6443/tcp

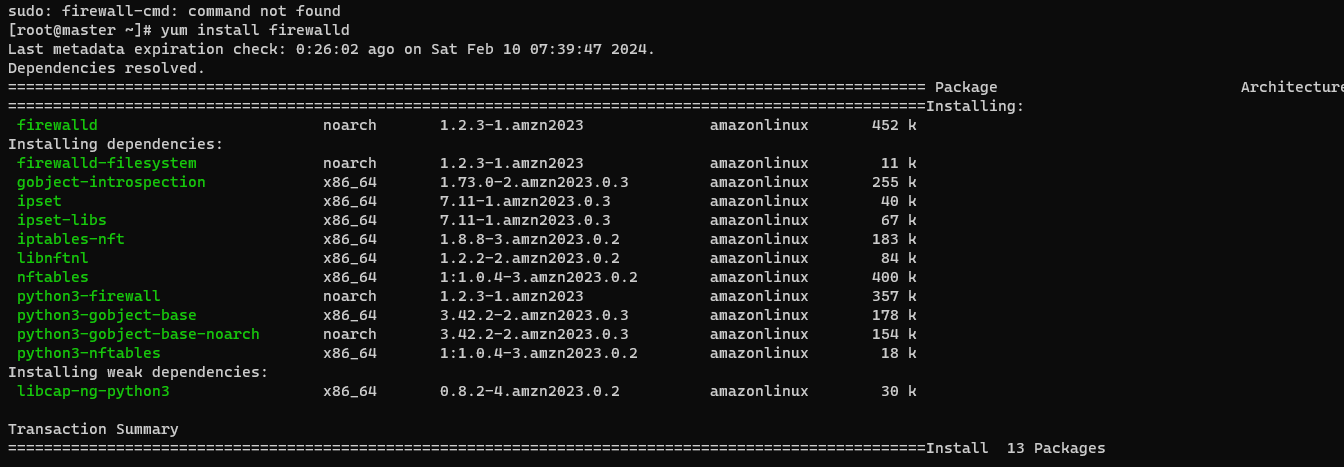
$ sudo firewall-cmd --permanent –add-port=2379-2380/tcp

$ sudo firewall-cmd --permanent –add-port=10250/tcp

$ sudo firewall-cmd --permanent –add-port=10251/tcp

$ sudo firewall-cmd --permanent –add-port=10252/tcp

$ sudo firewall-cmd --reload



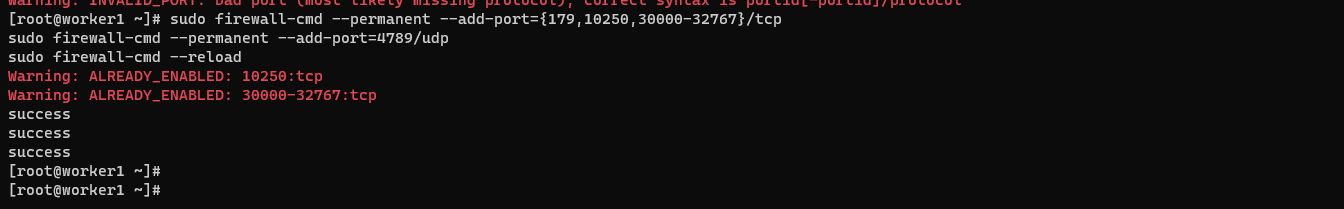


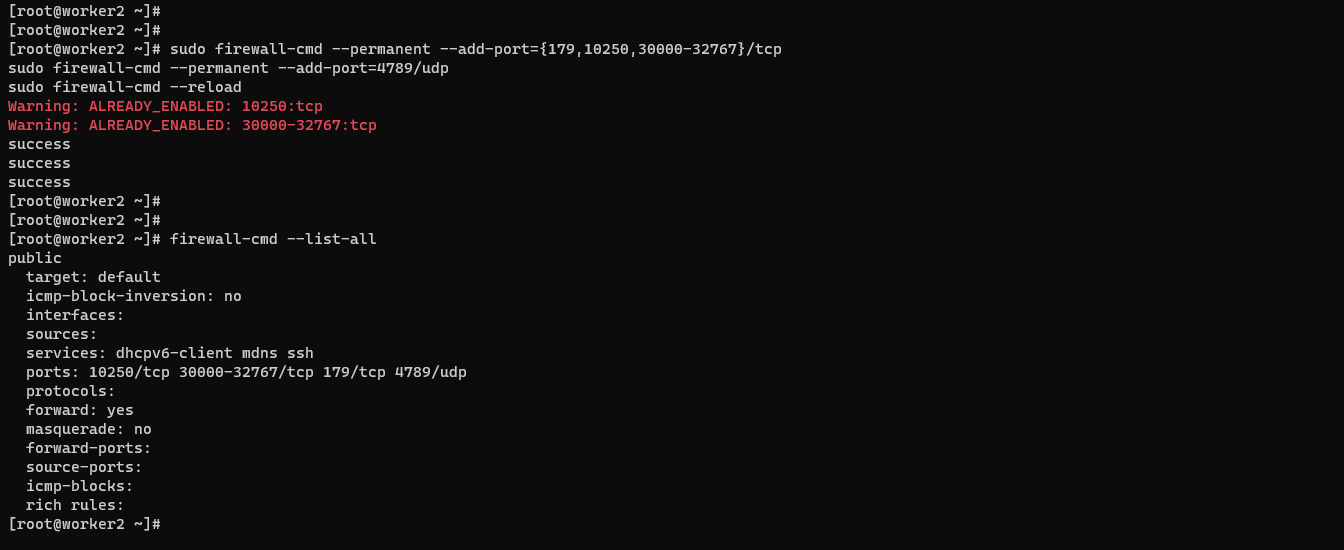
On Worker node, allow following ports,

$ sudo firewall-cmd --permanent –add-port=10250/tcp

$ sudo firewall-cmd --permanent --add-port=30000-32767/tcp

$ sudo firewall-cmd --reload





## Step 5) Install Containerd container runtime

Kubernetes requires a container runtime for pods to run. Kubernetes 1.23 and later versions require that you install a container runtime that confirms with the [Container Runtime](https://kubernetes.io/docs/concepts/overview/components/" \l "container-runtime) Interface.

In this guide, we will install Containerd which is a high-level container runtime. To do so, we need to enable two crucial kernel modules – overlay and br\_netfilter modules.

To achieve this, we need to configure the prerequisites as follows:

First, create a modules configuration file for Kubernetes:

$ sudo tee /etc/modules-load.d/containerd.conf <<EOF

overlay

br\_netfilter

EOF

Then load both modules using the modprobe command.

$ sudo modprobe overlay

$ sudo modprobe br\_netfilter



Next, configure the required sysctl parameters as follows:

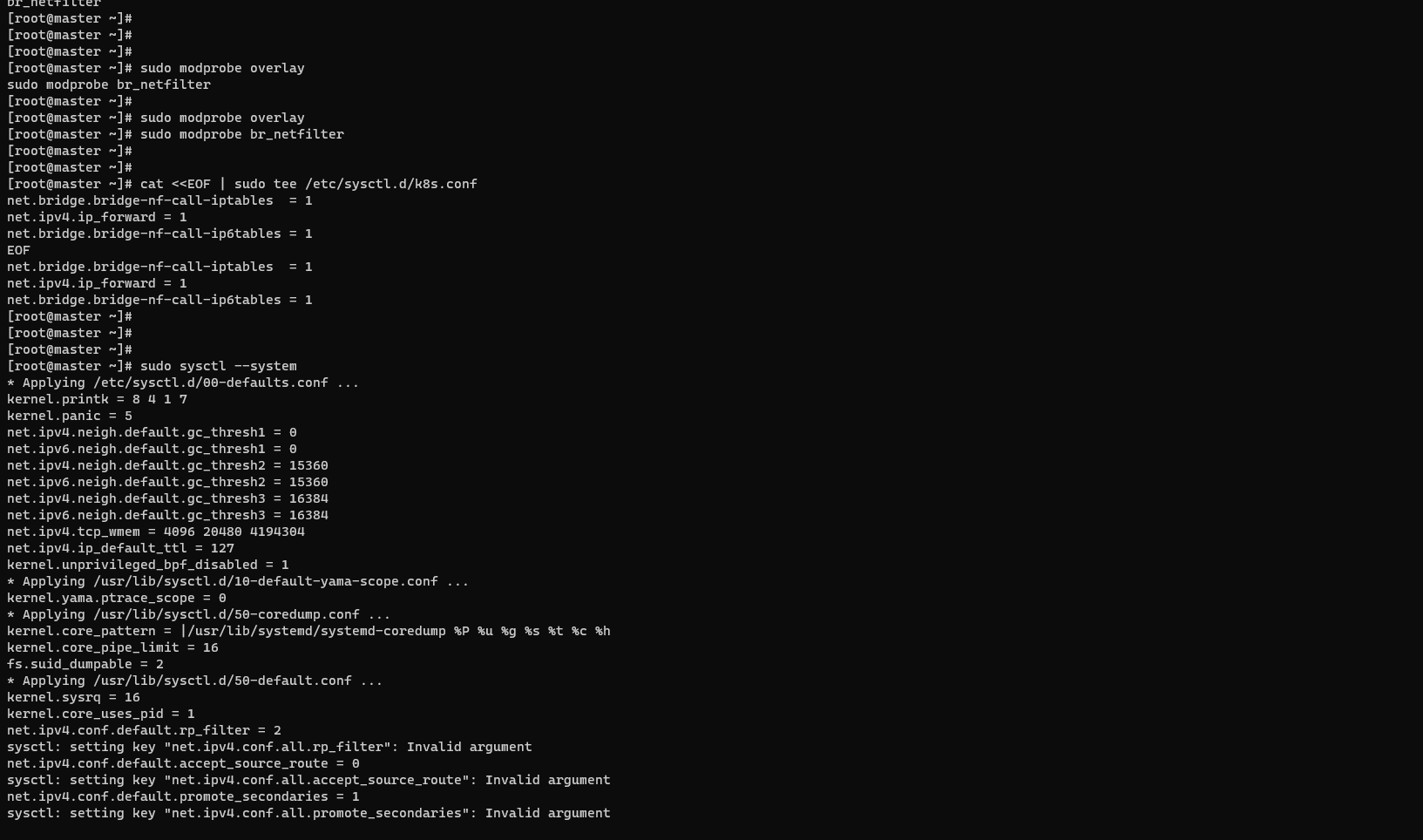
cat <<EOF | sudo tee /etc/sysctl.d/k8s.conf

net.bridge.bridge-nf-call-iptables = 1

net.ipv4.ip\_forward = 1

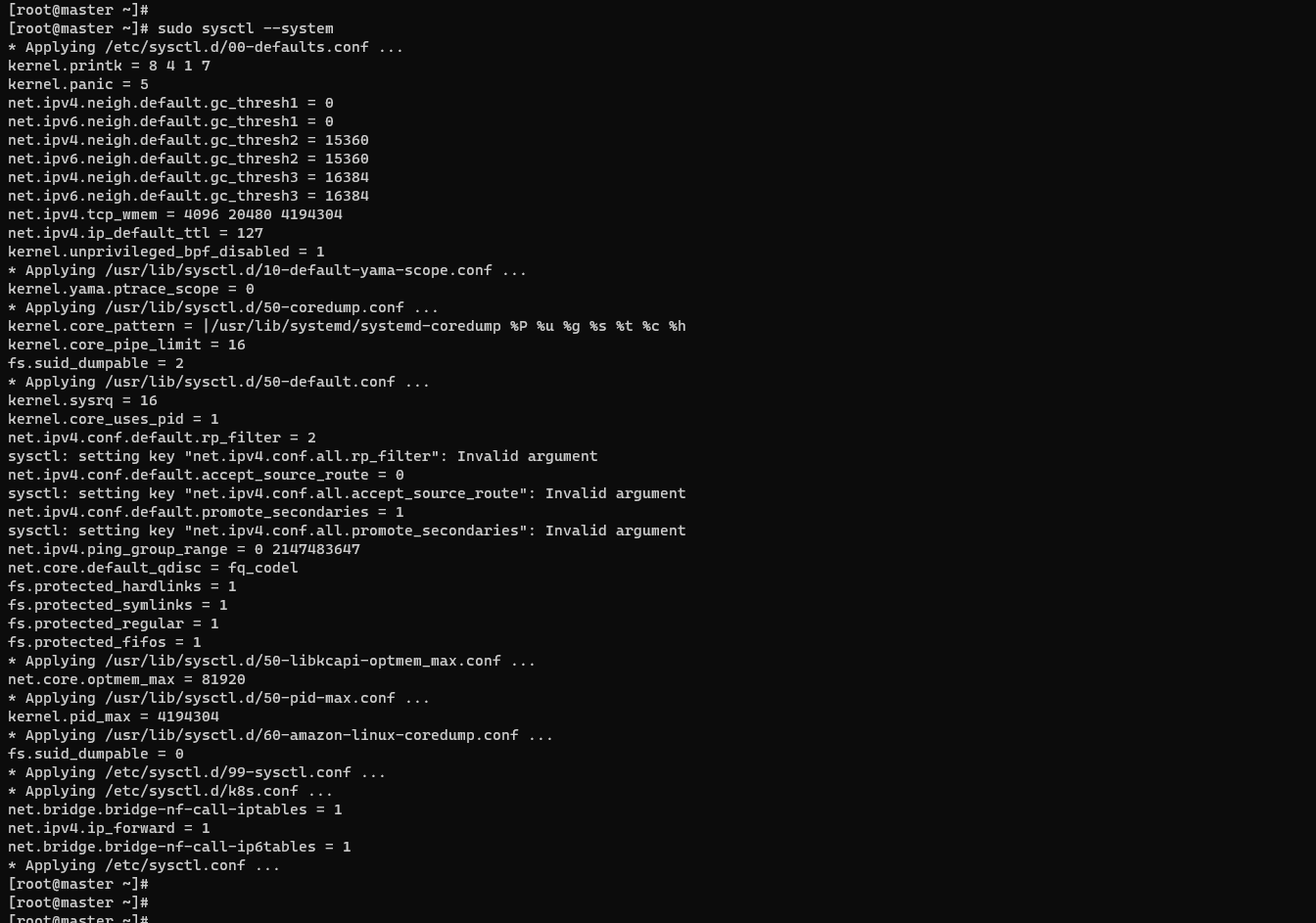
net.bridge.bridge-nf-call-ip6tables = 1

EOF



Save the changes and exit. To confirm the changes have been applied, run the command:

$ sudo sysctl –system



Install containerd runtime

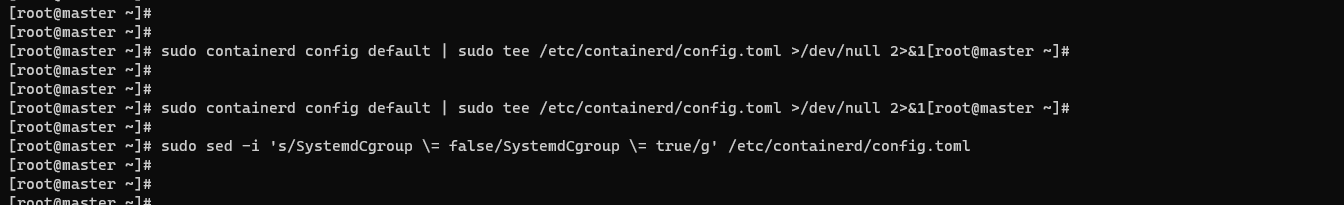
$ sudo dnf config-manager --add-repo https://download.docker.com/linux/centos/docker-ce.repo

$ sudo yum install containerd.io -y

$ sudo containerd config default | sudo tee /etc/containerd/config.toml >/dev/null 2>&1

$ sudo sed -i 's/SystemdCgroup \= false/SystemdCgroup \= true/g' /etc/containerd/config.toml

Start and enable the containerd service and also check the status of the service.

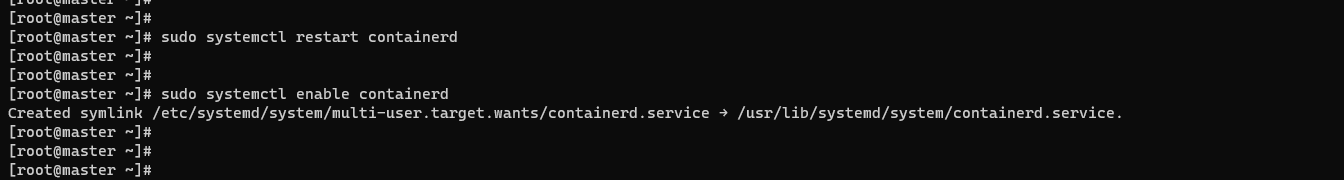




$ sudo systemctl start containerd

$ sudo systemctl enable containerd

$ sudo systemctl status containerd



## Step 6)  Install Kubernetes Packages

With everything required for Kubernetes to work installed, let us go ahead and install Kubernetes packages like kubelet, kubeadm and kubectl. Create a Kubernetes repository file.

$ sudo vi /etc/yum.repos.d/kubernetes.repo

And add the following lines.

[kubernetes]

name=Kubernetes

baseurl=https://packages.cloud.google.com/yum/repos/kubernetes-el7-x86\_64

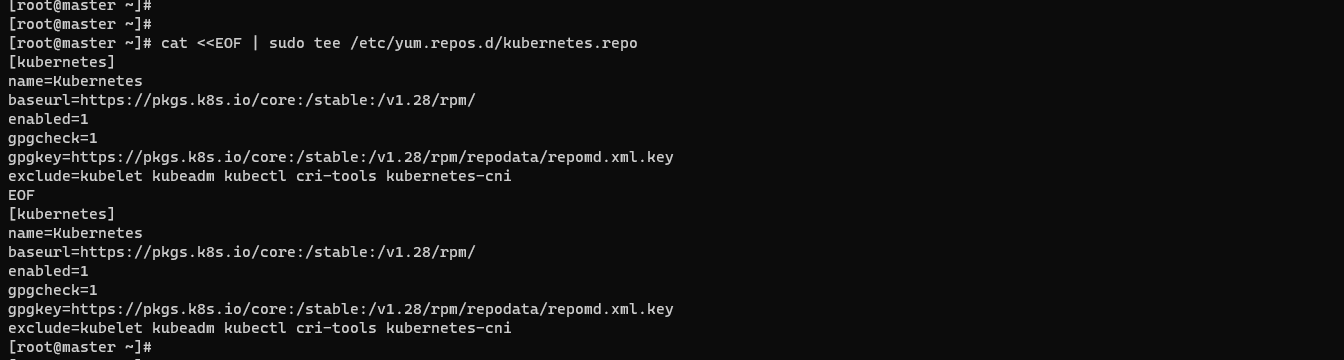
enabled=1

gpgcheck=1

repo\_gpgcheck=1

gpgkey=https://packages.cloud.google.com/yum/doc/yum-key.gpg https://packages.cloud.google.com/yum/doc/rpm-package-key.gpg

exclude=kubelet kubeadm kubectl



Finally, install k8s package as follows.

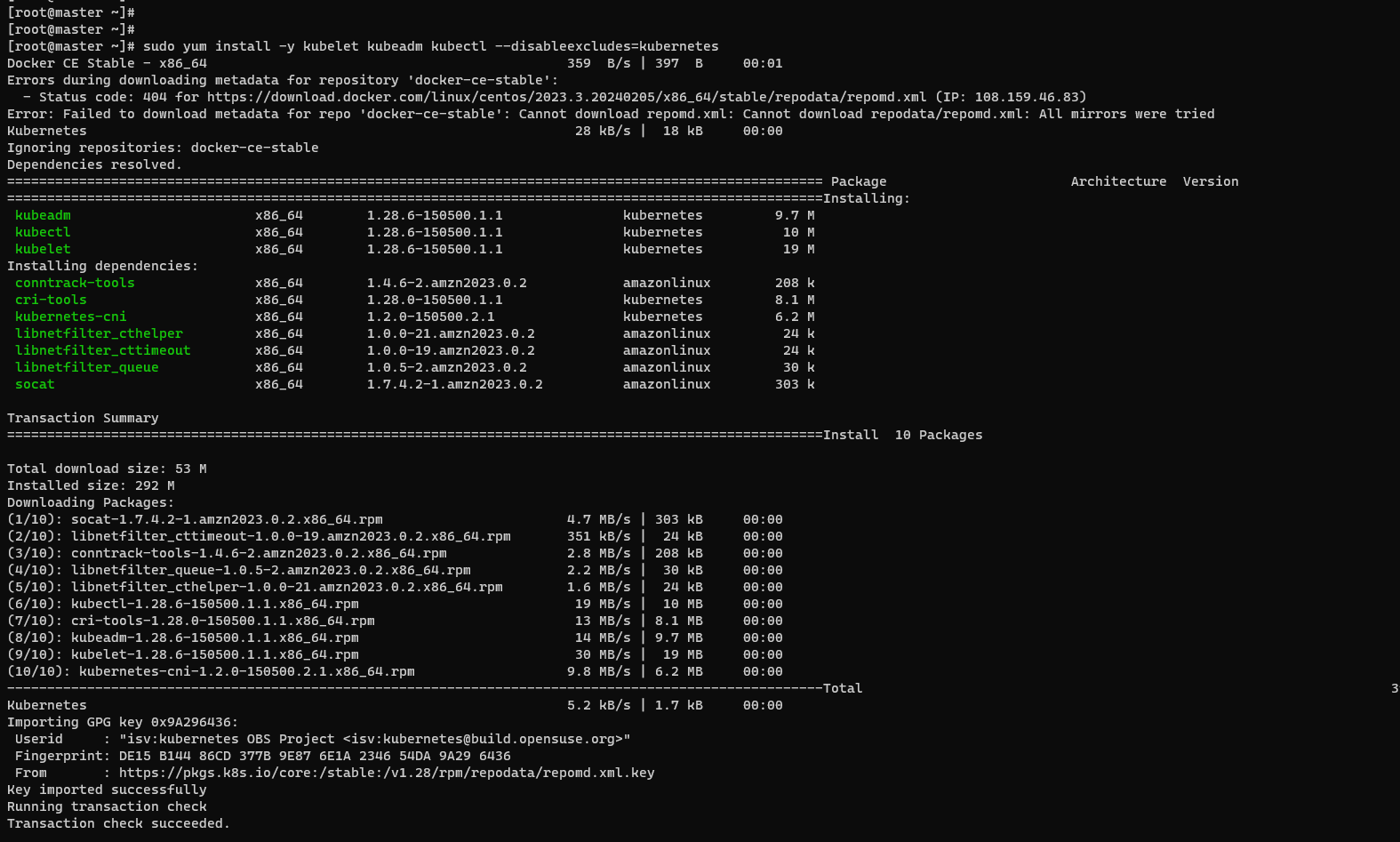
$ sudo dnf install -y kubelet kubeadm kubectl disableexcludes=kubernetes

Once installed, be sure to enable and start Kubelet service.

$ sudo systemctl enable kubelet

$ sudo systemctl start kubelet

At this juncture, we are all set to install Kubernetes cluster.



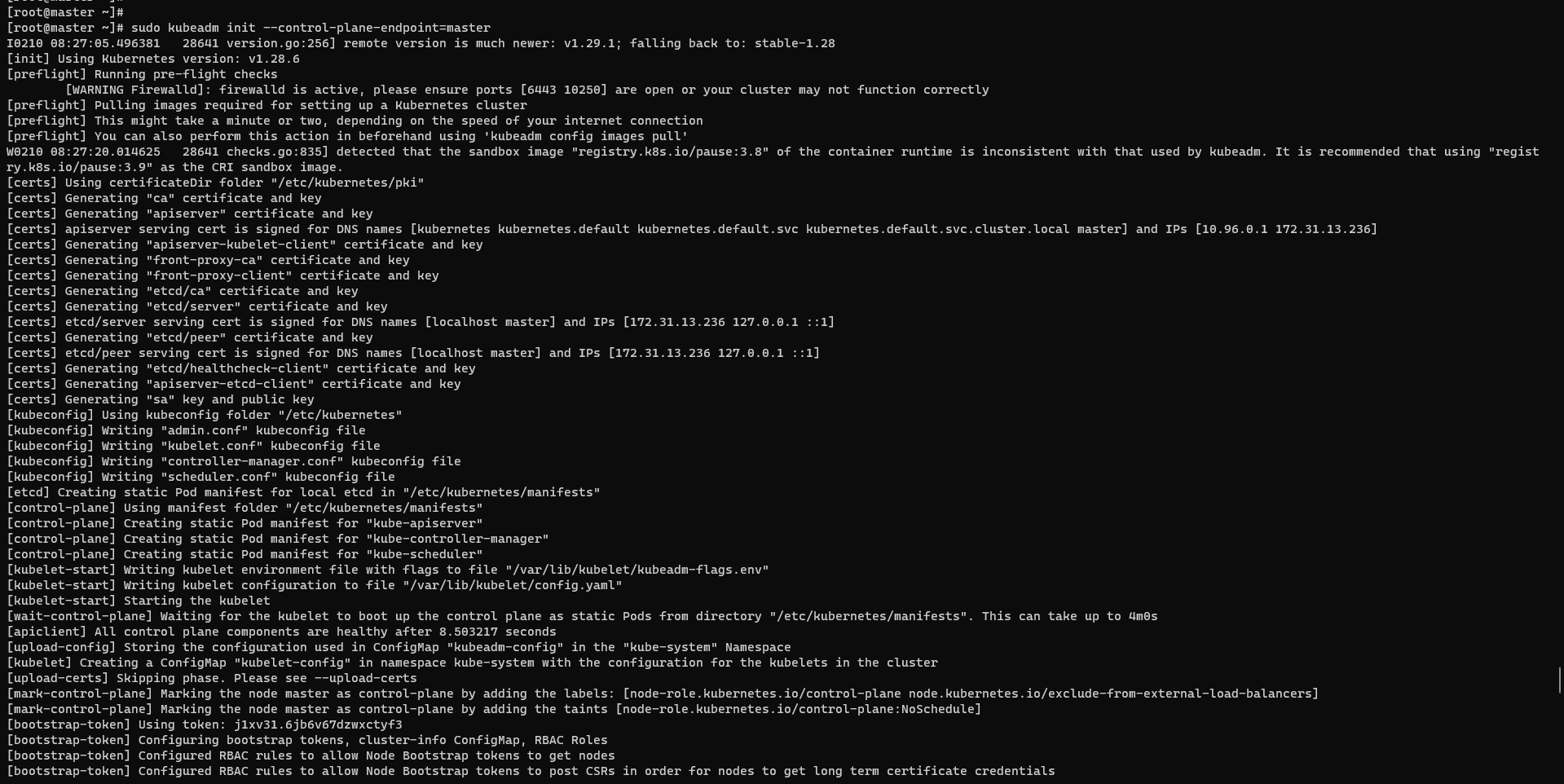


## Step 7)  Create a Kubernetes cluster

We are going to initialize a Kubernetes cluster using the kubeadm command as follows. This initializes a control plane in the master node.

$ sudo kubeadm init --control-plane-endpoint=master

Once the control plane is created, you will be required to carry out some additional commands to start using the cluster.





Therefore, run the commands, sequentially.

$ mkdir -p $HOME/.kube

$ sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config

$ sudo chown $(id -u):$(id -g) $HOME/.kube/config



## Step 8) Install Calico Pod Network Add-on

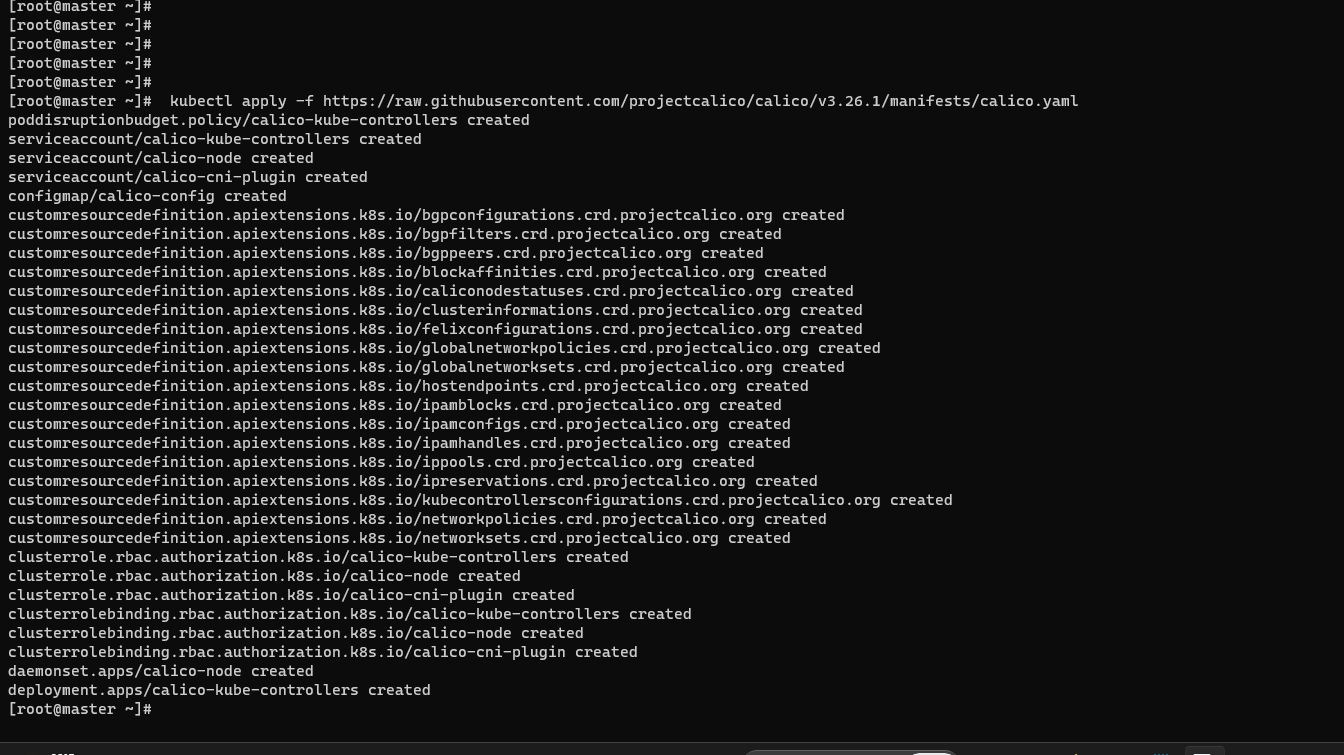
The next step is to install Calico CNI (Container Network Interface). It is an opensource project used to provide container networking and security. After Installing Calico CNI, nodes state will change to Ready state, DNS service inside the cluster would be functional and containers can start communicating with each other.

Calico provides scalability, high performance, and interoperability with existing Kubernetes workloads. It can be deployed on-premises and on popular cloud technologies such as AWS .

To install Calico CNI, run the following command from the master node

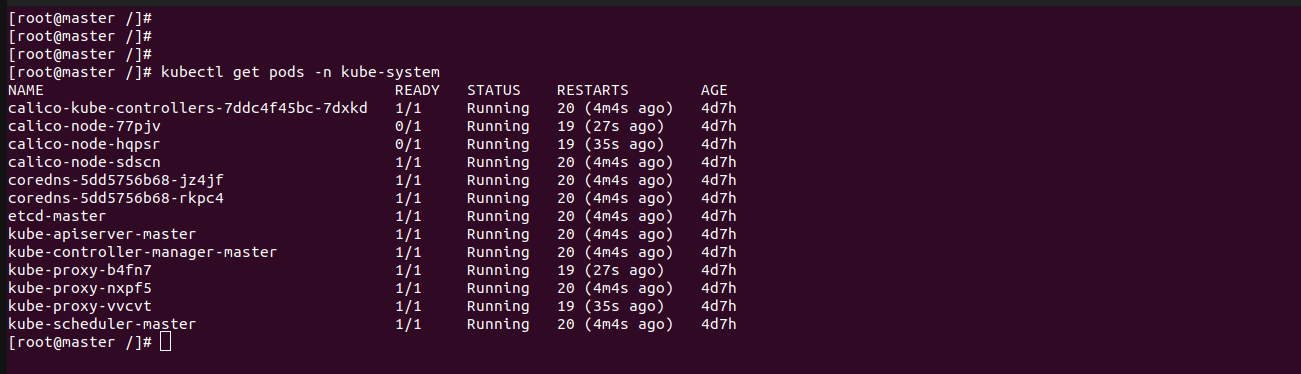
$ kubectl apply -f

https://raw.githubusercontent.com/projectcalico/calico/v3.26.1/manifests/calico.yaml



To confirm if the pods have started, run the command:

$ kubectl get pods -n kube-system



## Step 9) Adding worker node to the cluster

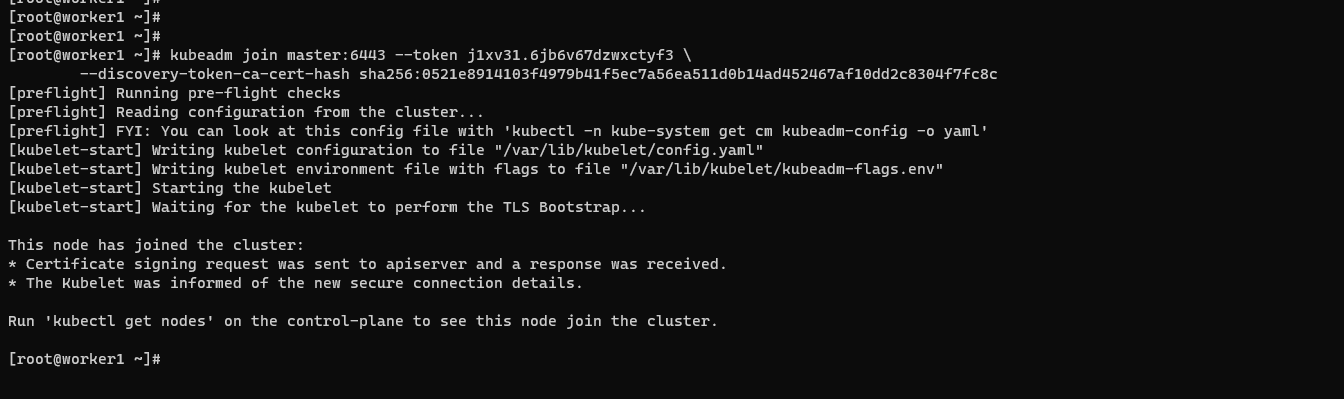
To add the worker node to the Kubernetes cluster, follow step 1 up until Step 6.  Once you are done, run the command generated by the master node for joining a worker node to the cluster. In our case, this will be:

$ sudo kubeadm join master:6443 --token cqb8vy.iicmmqrb1m8u9cob --discovery-token-ca-cert-hash

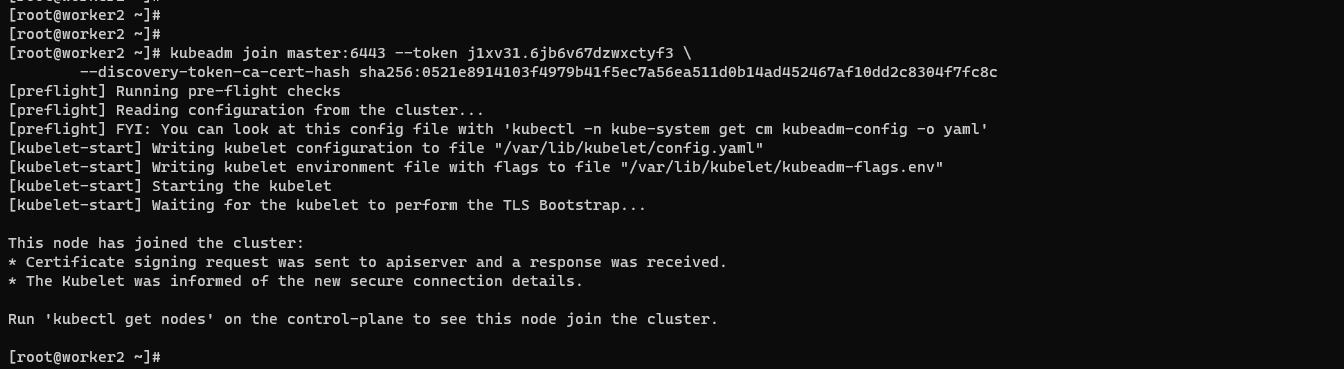
sha256:79748a56f603e6cc57f67bf90b7db5aebe090107d540d6cc8a8f65b785de7543

If all goes well, you should get the notification that the node has joined the cluster. Repeat the same procedure for other nodes in case you have multiple worker nodes

On worker1:



On Worker2 :



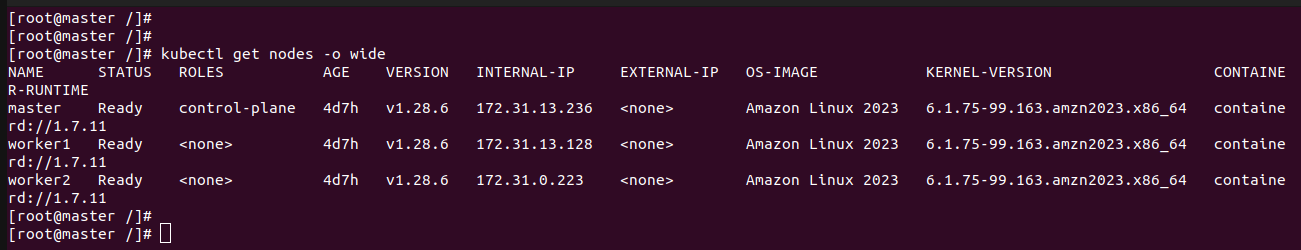
Now, head back to the master node and, once again, verify the nodes in your

cluster. This time around, the worker node will appear in the list on nodes in the

cluster,

In addition, you can retrieve more information using the -o wide options.

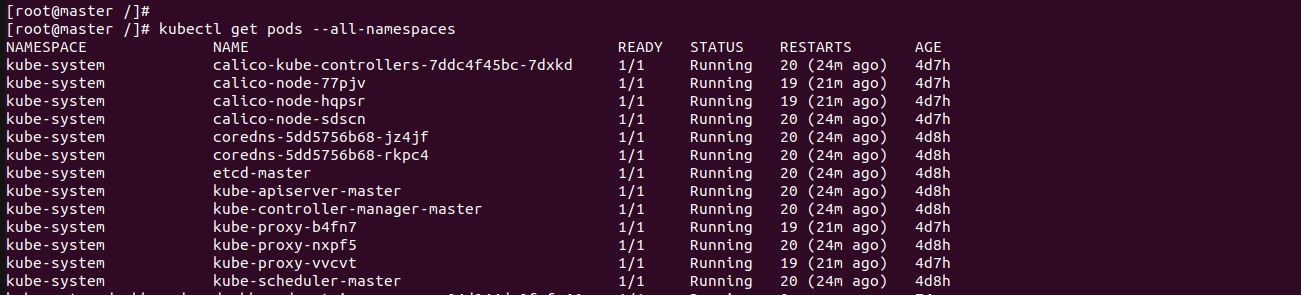
$ kubectl get nodes -o wide



The above output confirms that the master node is ready. Additionally, you can

check the pod namespaces:

$ kubectl get pods –all-namespaces



$ kubectl get nodes



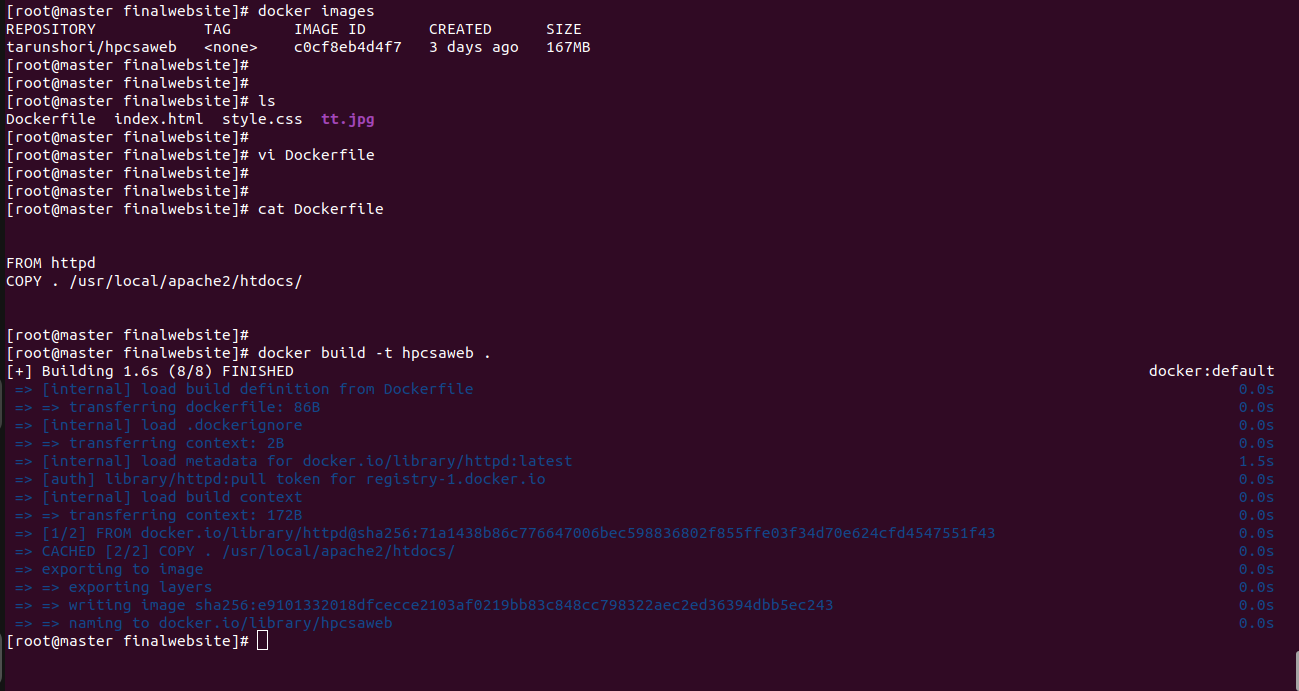
Create Docker Image And Push To Docker Hub

1.Create a Dockerfile: Create a Dockerfile in our project directory with the

necessary instructions to build your Docker image.

2.Build the Docker image: navigate to the directory containing your Dockerfile

and execute the following command to build the Docker image.



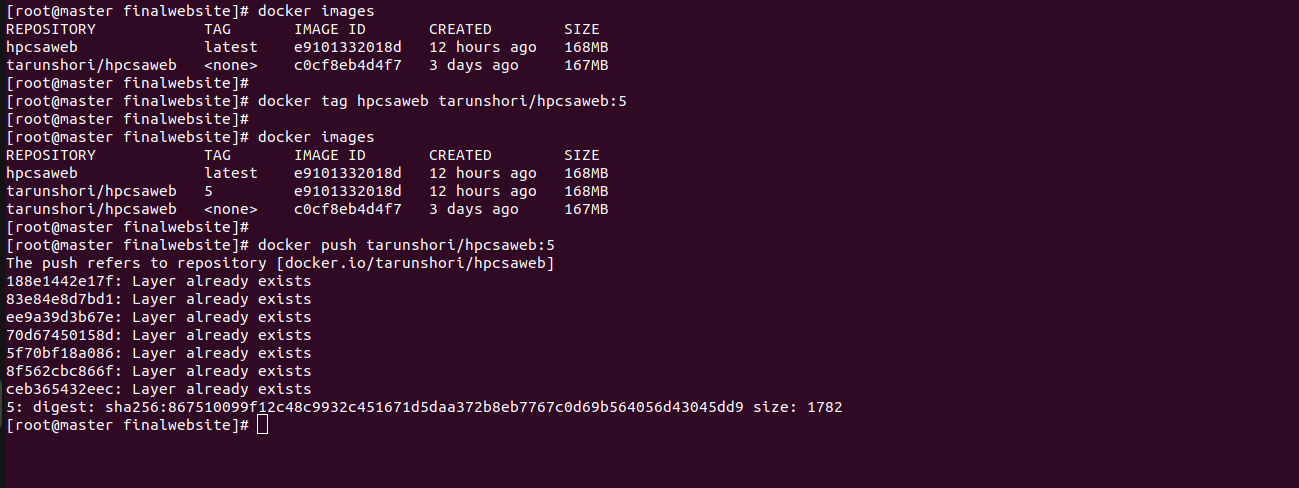
3.Tag the Docker image: After building the Docker image, need to tag it with

the appropriate version.

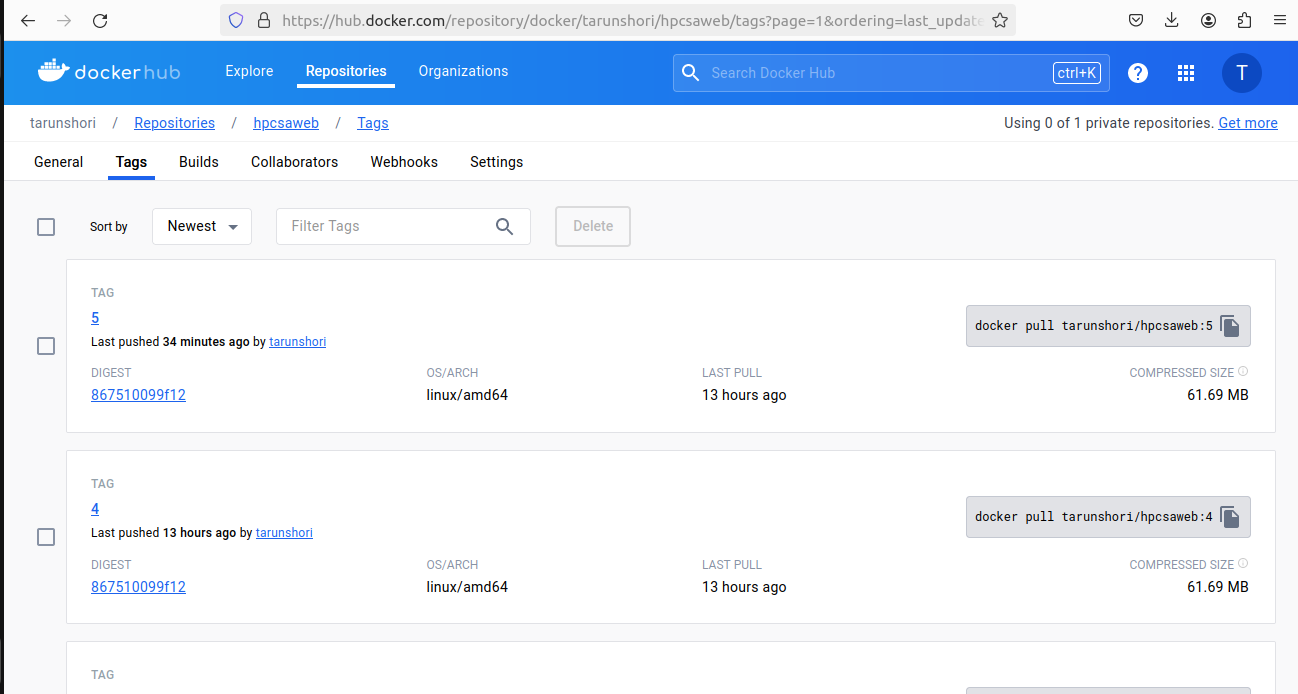
4.Push the Docker image to Docker Hub: Log in to Docker Hub account using

the docker login command, then push the tagged image to Docker Hub using the

docker push command.



5.Verify on Docker Hub: After pushing the image, verify that it's available on Docker Hub by visiting repository's page on the Docker Hub website.



Create PersistentVolume And PersistentVolumeClaim

1.create PersistentVolume:



This YAML manifest defines a PersistentVolume (PV) named `web-pv` within

the `project-website` namespace. Here's a breakdown of its components:

1. apiVersion: v1

- Specifies the Kubernetes API version being used for this resource.

2. kind: PersistentVolume

- Indicates that this resource is a PersistentVolume.

3. metadata:

- name: web-pv

- Provides a name for the PersistentVolume.

- namespace: project-website

- Specifies the namespace in which the PersistentVolume resides.

4. spec:

- capacity:

- storage: 4Gi

- Specifies the storage capacity of the PersistentVolume, which is 4

gigabytes.

5. accessModes:

- ReadWriteMany

- Specifies that the volume can be mounted as read-write by multiple nodes

simultaneously.

- local:

- path: /data/

- Specifies the path on the local node where the volume is located.

6. Node Affinity:

-Specifies that the PV is bound to nodes with the hostname "worker1" or

"worker2".

7. volumeMode: Filesystem

- Indicates that the PersistentVolume will be formatted with a filesystem.

8. storageClassName: standard

- Specifies the StorageClass to use for provisioning the PersistentVolume.

This PersistentVolume manifest defines a volume named `web-pv` with a capacity of 4 gigabytes, using the standard StorageClass. It allows multiple nodes to mount the volume as read-write. The volume is hosted locally at the directory `/data/` on the node.

2.create PersistentVolumeClaim :



This YAML manifest defines a PersistentVolumeClaim (PVC) named `web-pvc` within the `project-website` namespace. Here's a breakdown of its components:

1. apiVersion: v1

- Specifies the Kubernetes API version being used for this resource.

2. kind: PersistentVolumeClaim

- Indicates that this resource is a PersistentVolumeClaim.

3. metadata:

- name: web-pvc

- Provides a name for the PersistentVolumeClaim.

- namespace: project-website

- Specifies the namespace in which the PersistentVolumeClaim resides.

4. spec:

- volumeName: web-pv

- Specifies the name of the PersistentVolume to claim (`web-pv` in this case).

- storageClassName: standard

- Specifies the StorageClass to use for provisioning the

PersistentVolumeClaim.

- accessModes:

- - ReadWriteMany

- Specifies that the volume can be mounted as read-write by multiple nodes

simultaneously.

- resources:

- requests:

- storage: 2Gi

- Specifies the amount of storage requested for the PersistentVolumeClaim,

which is 2 gigabytes.

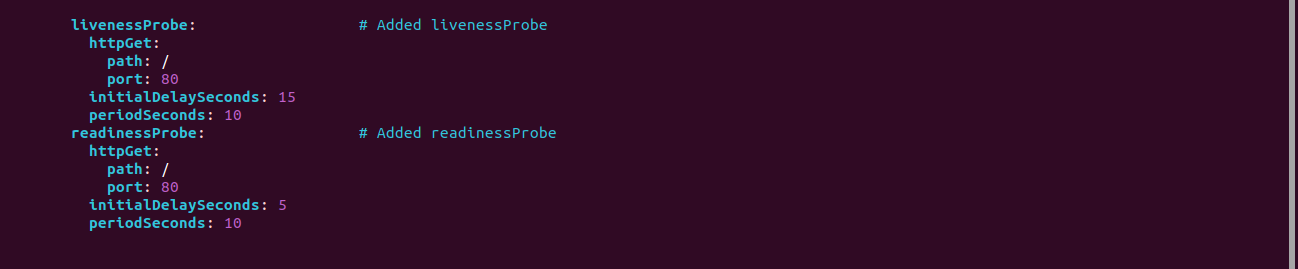
This PersistentVolumeClaim manifest requests a volume named `web-pv` with a capacity of 2 gigabytes, using the standard StorageClass. It allows multiple nodes to mount the volume as read-write.

Create Deployment and Service

run an application by creating a Kubernetes Deployment,Services,Horizontal Pod Autoscaler and other object. we describe all configurations in a YAML file.

1. Create a **Deployment** based on the YAML file:





1. Name: `my-web`

- Identifies the deployment within the Kubernetes cluster.

2. Namespace: `project-website`

- Specifies the namespace where the deployment resides.

3. Replicas: `3`

- Indicates the desired number of replica pods for the deployment.

4. Selector:

- `matchLabels: app=my-web`

- Specifies the labels used to match the pods controlled by this deployment.

5. Template:

- Defines the pod template for the deployment.

6. Containers:

- `web-container`

- Name of the container within the pod.

- `docker.io/tarunshori/hpcsaweb`

- Docker image used to create the container.

- Ports:

- `containerPort: 80`

- Exposes port 80 within the container.

7. Liveness Probe:

- livenessProbe:

- This section indicates the beginning of the livenessProbe

- configuration.httpGet: Specifies that an HTTP GET :

-request will be used for probing the container.

- Path: /

- Defines the path to which the HTTP GET request will be sent. In this case, it's

the root path /.

-port: 80:

-Specifies the port on which the HTTP GET request will be sent. Port 80 is

commonly used for HTTP traffic.

- InitialDelaySeconds: 15

- Specifies the number of seconds to wait after the container starts before the

first liveness probe is performed. This delay allows the container to initialize.

- periodSeconds: 10

- Specifies the interval between successive liveness probe executions. In this

case, a new probe will be executed every 10 seconds.

8. Readiness Probe:

- readinessProbe :

- This section indicates the beginning of the readiness Probe configuration.

- httpGet:

- Specifies that an HTTP GET request will be used for probing the container's

readiness.

- path: /:

-Defines the path to which the HTTP GET request will be sent. Here, it's the

root path /.

- port: 80:

- Specifies the port on which the HTTP GET request will be sent. Again, port 80

is used for HTTP traffic.

- initialDelaySeconds: 5

-Specifies the number of seconds to wait after the container starts before the

first readiness probe is performed. This allows the container some time to

become ready to serve traffic.

- periodSeconds: 10:

-Specifies the interval between successive readiness probe executions. In this

case, a new probe will be executed every 10 seconds.

9. Volumes:

- `data-volume`

- Name of the volume.

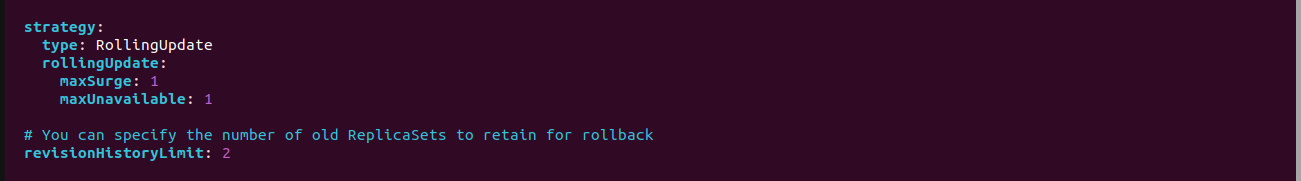
- Persistent Volume Claim: `web-pvc`

- Reference to the PersistentVolumeClaim used by the volume.

- Mount Path: `/data/`

- Specifies the mount path within the container.

2. Create a **Update Strategy** based on the YAML file:



1. Update Strategy:

- `type: RollingUpdate`

- Defines the update strategy for the deployment.

- `rollingUpdate:`

- Specifies the parameters for rolling updates.

- `maxSurge: 1`

- Maximum number of additional pods that can be created during an update.

- `maxUnavailable: 1`

- Maximum number of pods that can be unavailable during an update.

2. Revision History Limit:`2`

- Specifies the number of old ReplicaSets to retain for rollback purposes.

3. Create a **Horizontal Pod Autoscaler** based on the YAML file:



1. Name: `web-autoscaler`

- Identifies the HorizontalPodAutoscaler within the Kubernetes cluster.

2. Scale Target Reference:

- `apiVersion: apps/v1`

- `kind: Deployment`

- `name: my-web`

- Specifies the deployment to scale based on CPU utilization.

3. Minimum Replicas: `3`

- Specifies the minimum number of pods to maintain.

4. Maximum Replicas: `10`

- Specifies the maximum number of pods to scale up to.

5. Target CPU Utilization:`70%`

- Sets the target CPU utilization percentage to trigger scaling.

4. Create a **Service** based on the YAML file:



1. Name:`web-service`

- Identifies the service within the Kubernetes cluster.

2. Namespace: `project-website`

- Specifies the namespace where the service resides.

3. Type: `NodePort`

- Exposes the service on a static port on each node's IP.

4. Selector:

- `app: my-web`

- Routes traffic to pods with the label `app=my-web`.

5. Ports:

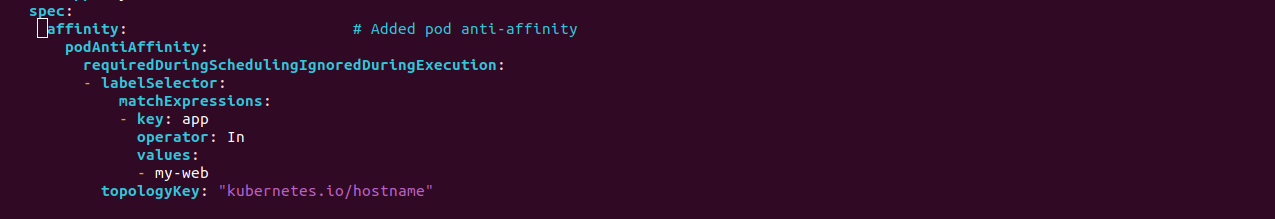
- `protocol: TCP`

- Specifies the protocol used for the port.

- `port: 80`

- Exposes port 80 within the cluster.

5. Create **Pod anti-affinity**  on the Deployment YAML file:



1. Label Selector: Ensure that the label selector (app: my-web) matches the

labels applied to your pods correctly. If the label is not applied or does not

match, the anti-affinity rule won't have any effect.

2. Topology Key: The topologyKey specifies the key of the node label to use

for anti-affinity. In this case, it's set to "kubernetes.io/hostname",

which means Kubernetes will consider the hostname of nodes when applying the

anti-affinity rule. Make sure that your nodes have unique hostnames for this rule

to work effectively.

3. Cluster Size: In a small cluster with only a few nodes, applying pod anti-affinity may limit the scheduling options for Kubernetes, potentially affecting the distribution of pods across nodes. Consider the impact on resource utilization and scheduling efficiency before applying anti-affinity rules in such environments.

Apply YAML Manifest For The "My-Web"

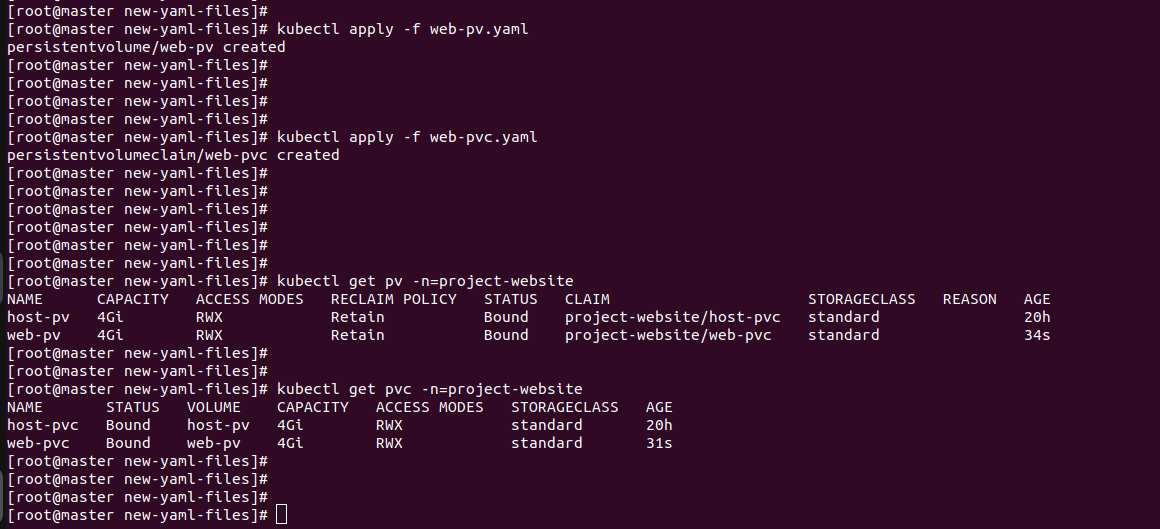
1. Apply YAML Manifest :

$ kubectl apply -f web-pv.yaml

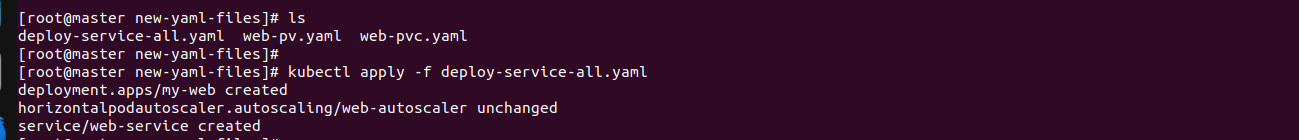
$ kubectl apply -f web-pvc.yaml

$ kubectl get pv -n=project-website

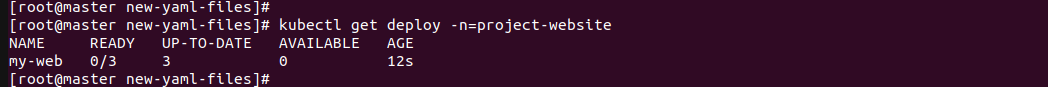
$ kubectl get pvc -n=project-website



$ kubectl apply -f deploy-service-all.yaml



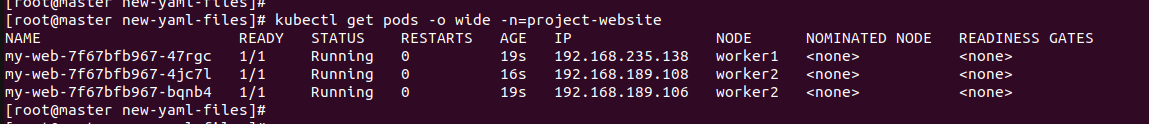
2. Display information about the Deployment:



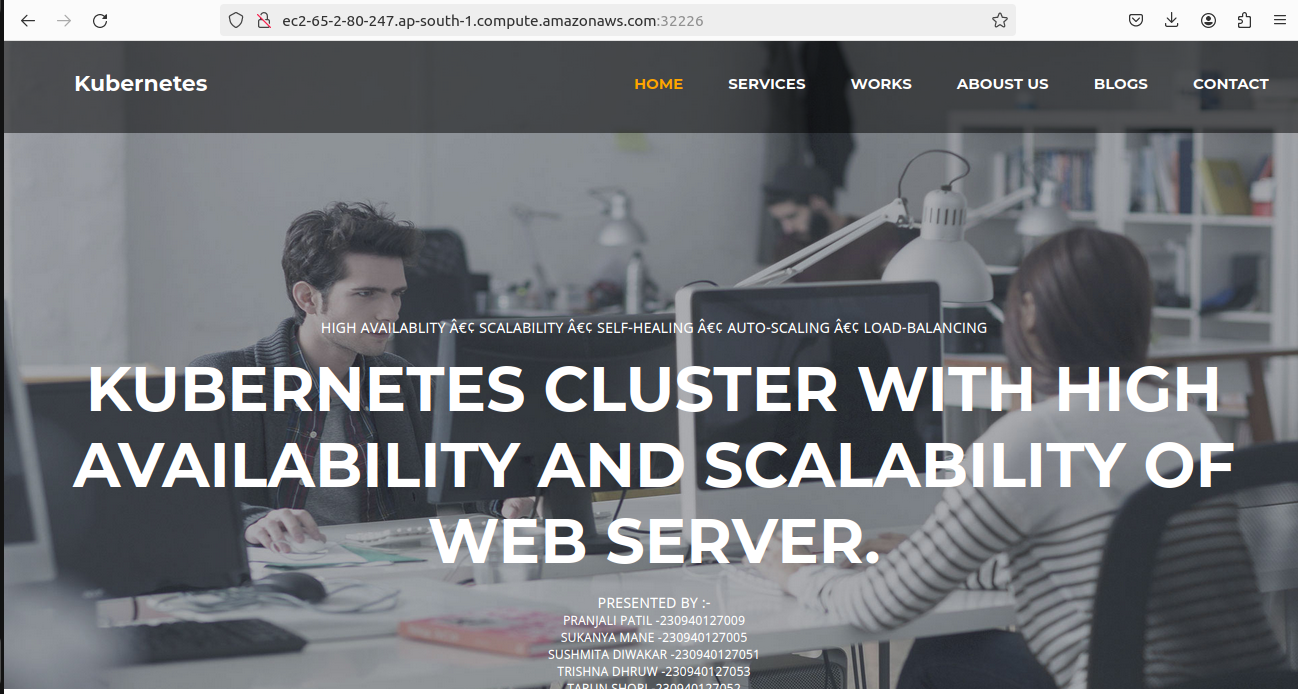
3. Display information about the Service:



4. List the Pods created by the deployment :



5. Access web-server from outside of the cluster using Node Port :

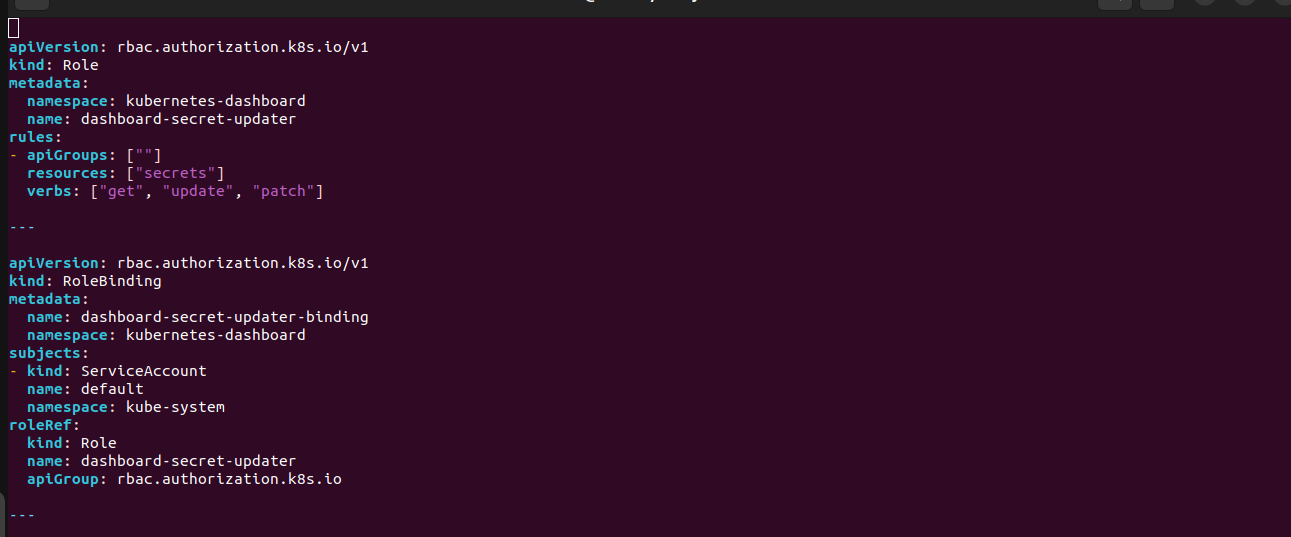


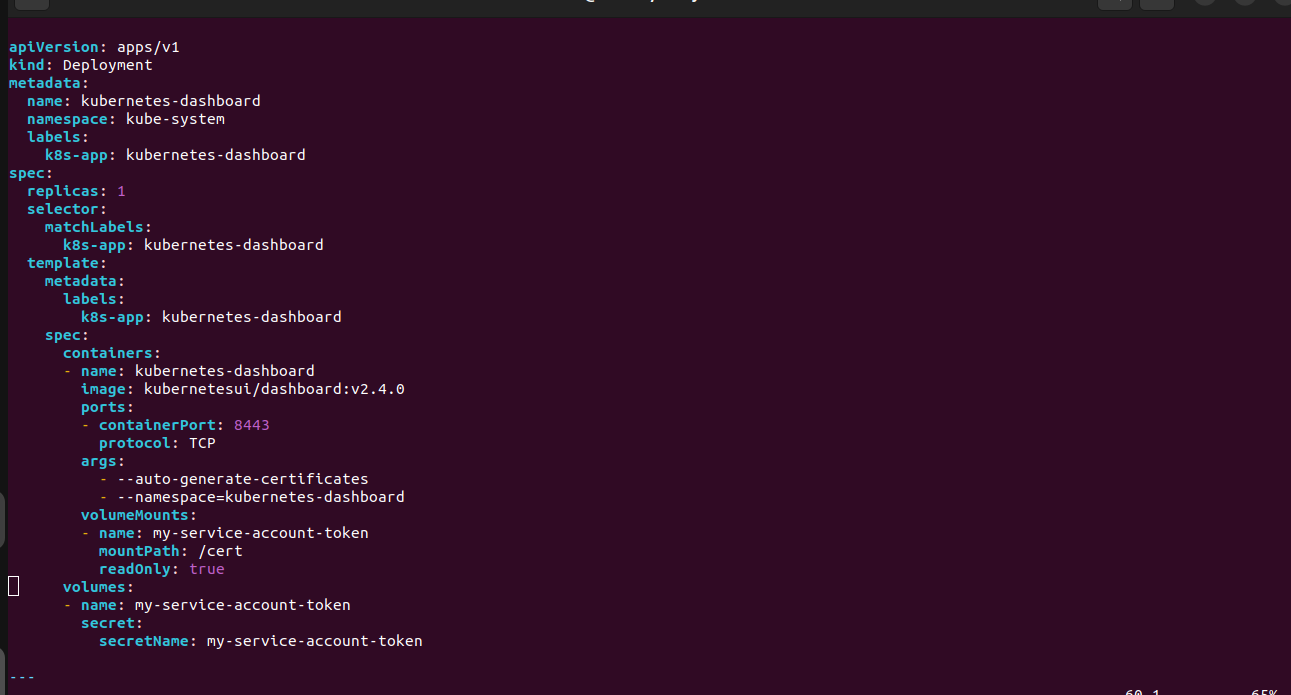
Kubernetes Dashboard Setup

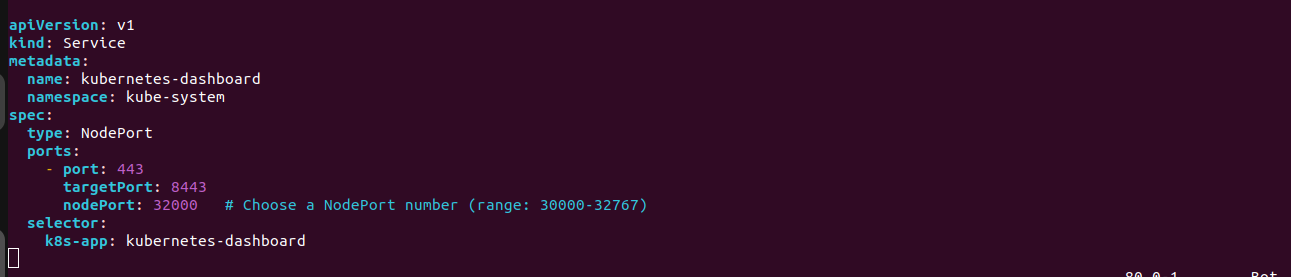
The dashboard is a web-based Kubernetes user interface. we can use the dashboard to deploy containerized applications to a Kubernetes cluster, troubleshoot your containerized application, and manage the cluster resources. You can use the dashboard to get an overview of the applications running on your cluster, as well as for creating or modifying individual Kubernetes resources (such as Deployments, Jobs, DaemonSets, etc). For example, you can scale a Deployment, initiate a rolling update, restart a POD, or deploy new applications using a deploy wizard.

1. The Dashboard UI is not deployed by default. To deploy it, run the following command:

Create a new Directory for Dashboard and create a yaml file for ServiceAccount, ClusterRoleBinding,Role, Deployment and Service.







This YAML file describes a set of Kubernetes resources for deploying the Kubernetes Dashboard application with RBAC (Role-Based Access Control) configuration. Let's break down each section:

1. Role:

- apiVersion: rbac.authorization.k8s.io/v1

- kind: Role

- metadata: Specifies metadata for the Role.

- namespace: The namespace where the Role is defined (`kubernetes

-dashboard`).

- name: The name of the Role (`dashboard-secret-updater`).

- rules: Defines permissions/rules for the Role.

- Grants permission to access, update, and patch secrets within the same

namespace (`kubernetes-dashboard`).

2. RoleBinding:

- apiVersion: rbac.authorization.k8s.io/v1

- kind: RoleBinding

- metadata: Specifies metadata for the RoleBinding.

- name: The name of the RoleBinding (`dashboard-secret-updater-binding`).

- namespace: The namespace where the RoleBinding is defined (`kubernetes-dashboard`).

- subjects: Defines the subjects (users, groups, or service accounts) that the

RoleBinding applies to.

- In this case, it applies to the `default` service account in the `kube-system`

namespace.

- roleRef: Specifies the Role reference.

- kind: Specifies the kind of resource being referenced (`Role`).

- name: Specifies the name of the Role (`dashboard-secret-updater`).

- apiGroup: Specifies the API group (`rbac.authorization.k8s.io`).

3. Deployment:

- apiVersion: apps/v1

- kind: Deployment

- metadata: Specifies metadata for the Deployment.

- name: The name of the Deployment (`kubernetes-dashboard`).

- namespace: The namespace where the Deployment is defined (`kube-

system`).

- labels: Labels to identify the Deployment (`k8s-app: kubernetes-dashboard`).

- spec: Specifies the desired state for the Deployment.

- replicas: Number of desired replicas (1).

- selector: Specifies how the Deployment finds which Pods to manage.

- template: Defines the Pod template for the Deployment.

- metadata: Metadata for the Pod template.

- spec: Specification for the Pod template.

- containers: List of containers to run in the Pod.

- Specifies the container named `kubernetes-dashboard`.

- image: Docker image to use (`kubernetesui/dashboard:v2.4.0`).

- ports: Specifies the port mapping for the container (`containerPort: 8443`).

- args: Arguments to the container.

- volumeMounts: Mounts a volume into the container.

- volumes: Defines a volume for the Pod (`my-service-account-token`).

4. Service:

- apiVersion: v1

- kind: Service

- metadata: Specifies metadata for the Service.

- name: The name of the Service (`kubernetes-dashboard`).

- namespace: The namespace where the Service is defined (`kube-system`).

- spec: Specifies the desired state for the Service.

- type: The type of Service (`NodePort`).

- ports: Specifies the port mapping for the Service.

- Exposes port 443 on the Service and targets port 8443 on Pods.

- Specifies a NodePort number (32000) for accessing the Service externally.

- selector: Selects the Pods to which the Service applies based on labels.

This YAML file defines RBAC rules, a Deployment for running the Kubernetes Dashboard application, and a Service for exposing it externally.

2. Then apply yaml file :

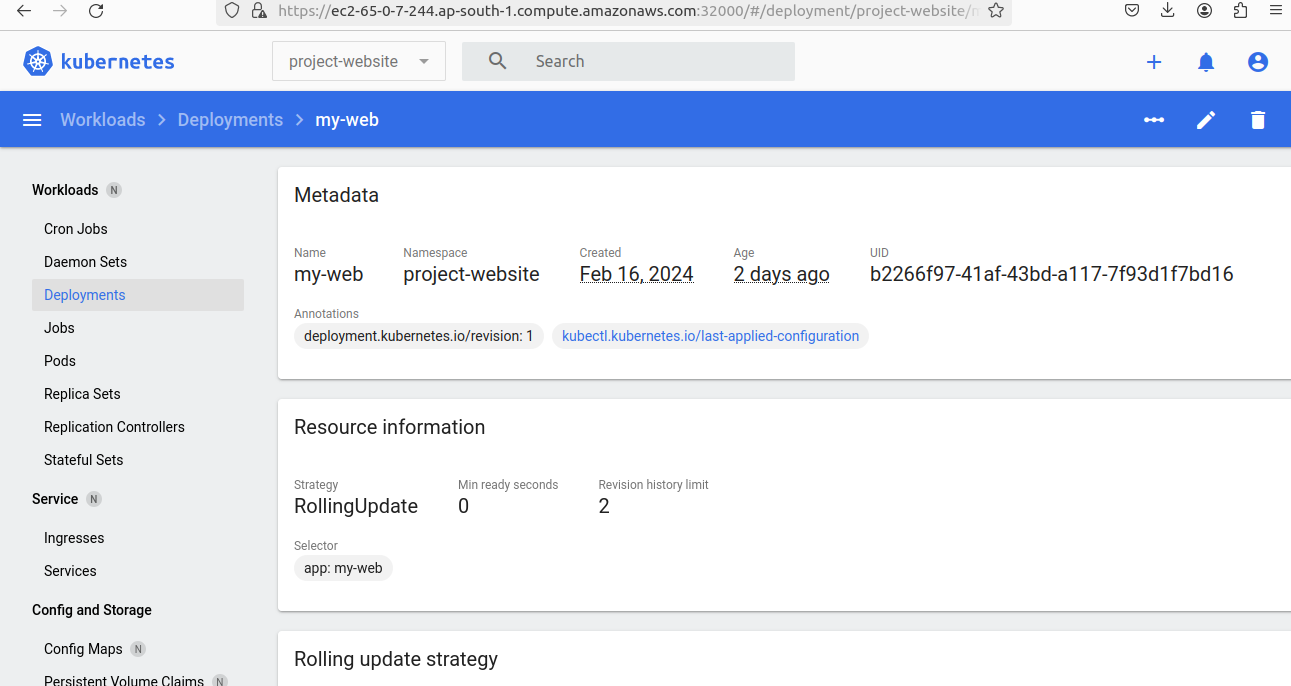
$ kubectl apply -f sa-dashboard.yaml

3. Get the token for authentication :

$ kubectl get secrets -n kube-system

$ kubectl describe secret my-service-account-token -n kube-system

4. Copy the token and paste in the kubernetes dashboard url with Nodeport.

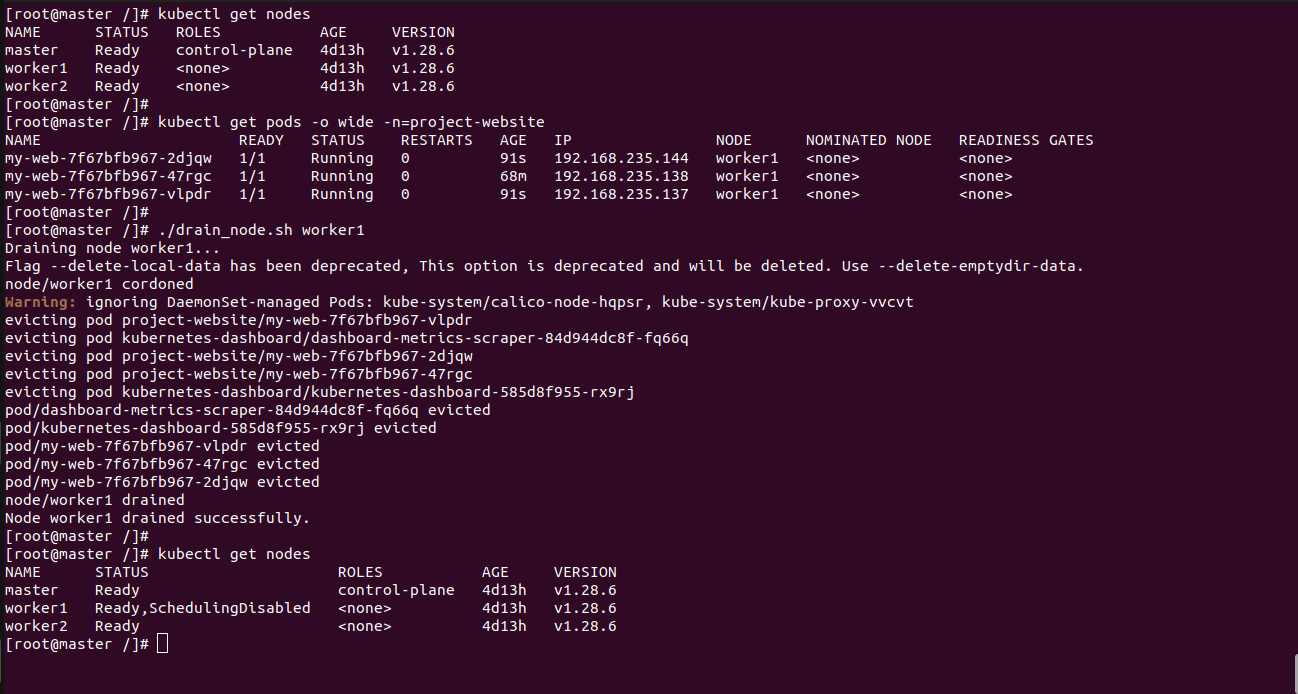


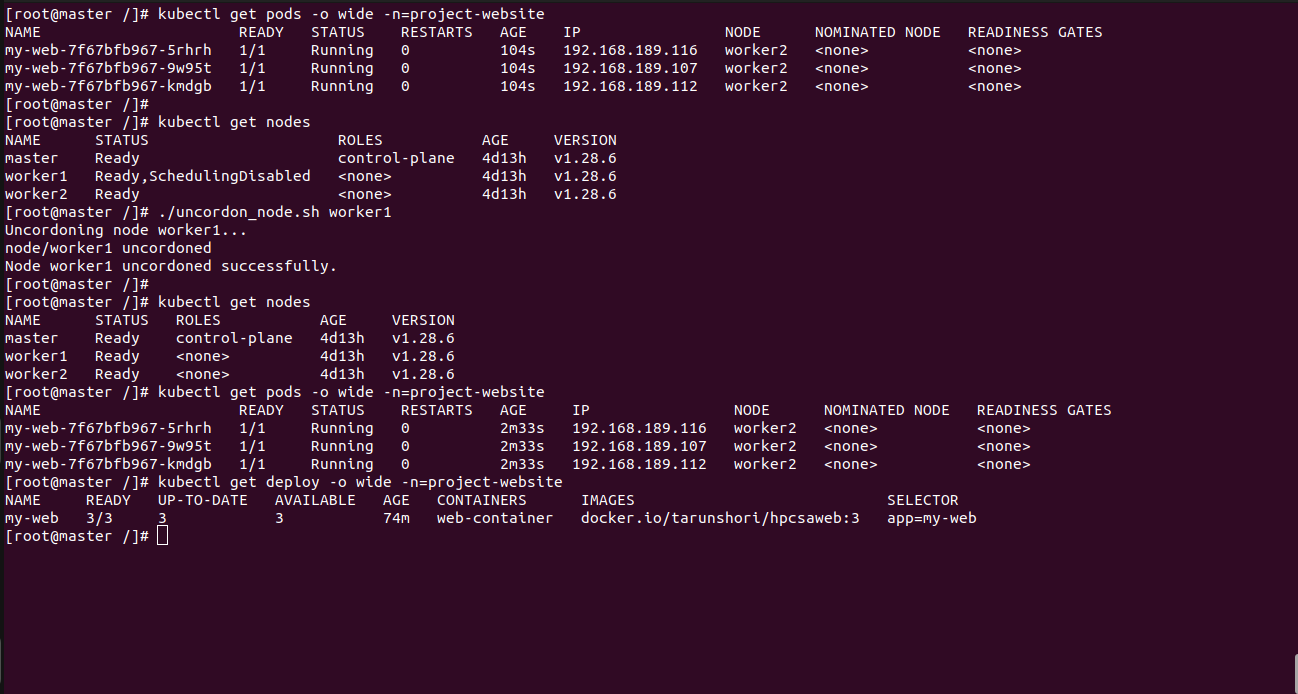
Validation

1. Verify **High** **Availability** of Pods if worker node is Failed : ReplicaSets or Deployments to ensure that

multiple replicas of your application pods are running across multiple nodes in the cluster. ReplicaSets

manage the lifecycle of replicated pods and ensure that a specified number of pod replicas are running at all times.





2. Verify **Scalability** of Pods : Horizontal Pod Autoscaling (HPA): Horizontal

Pod Autoscaling to

automatically scale the number of pods in a deployment based on observed CPU

or custom metrics.

* HorizontalPodAutoscaler (HPA) resource that specifies the deployment to autoscale and the desired target CPU utilization or custom metric threshold.
* Configured HPA to scale within specified minimum and maximum replica counts.
* Monitor the HPA's behavior as load increases or decreases to ensure that pods are scaled up or down accordingly.

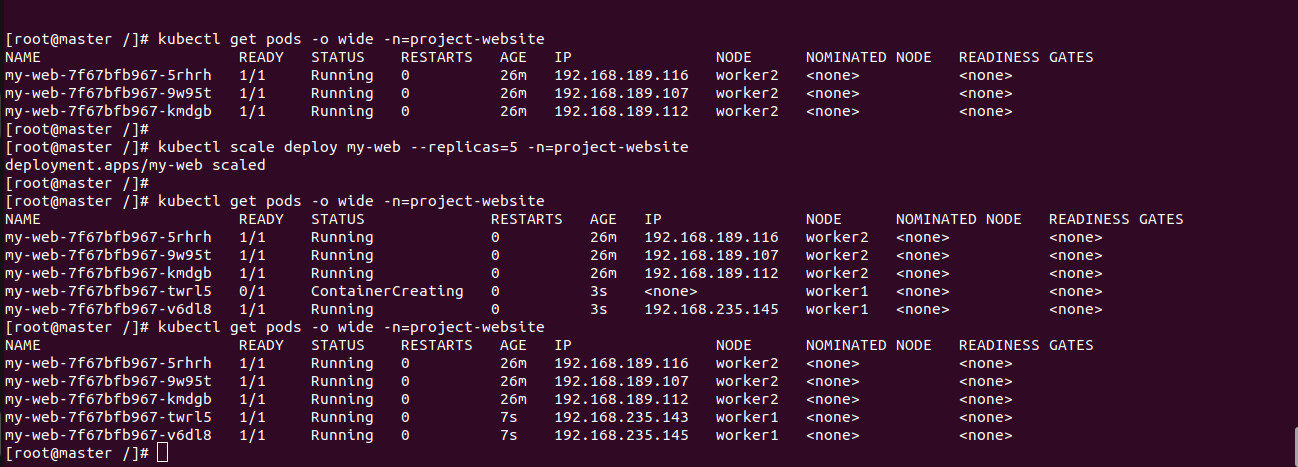
3. verify Manual Scaling: Experiment with manually scaling the number of

replicas in your deployment

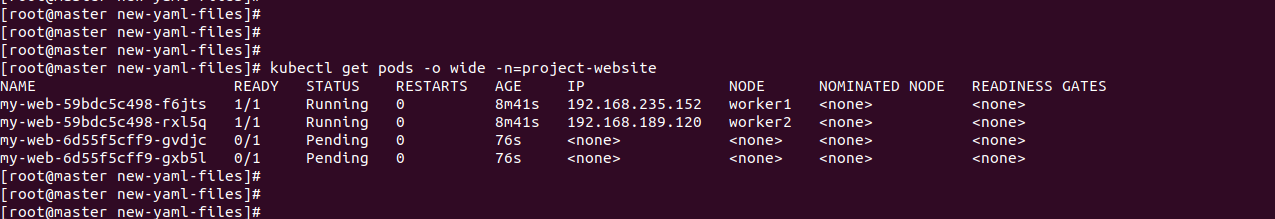
using the kubectl scale command. Verify that pods are added or removed as

expected and that the

application remains responsive during the scaling process.



4.Testing anti-affinity:test affinity and anti-affinity rules thoroughly to ensure they have the desired effect without causing unintended consequences. Monitor pod scheduling and cluster behavior after applying these rules to verify their effectiveness.



References & Bibliography

1. Kubernetes Documentation

https://kubernetes.io/docs/home/

1. Containerd Documentation

<https://containerd.io/docs/>

1. AWS Cloud Documentation

<https://docs.aws.amazon.com/>

1. Calico Documentation

https://docs.tigera.io/

Project Link

Github: https://github.com/tarun-code/KUBERNETES-CLUSTER-PROJECT

Limitations

1. Vendor lock-in: While Kubernetes itself is open-source and portable, using it on AWS may tie you to AWS-specific services and features, making it harder to migrate to another cloud provider in the future.

2. Service limitations: Some AWS services may not be fully compatible or optimized for Kubernetes. For example, AWS Load Balancers may require additional configuration to work seamlessly with Kubernetes services.

3. Regional availability: Not all AWS services and features are available in every AWS region, which could impact your Kubernetes deployment's flexibility and availability

4. Effectiveness: Pod anti-affinity may not be as effective in a two-node cluster compared to larger clusters with more nodes. With only two nodes available, there are fewer options for Kubernetes to schedule pods away from each other while still maintaining resource balance.

Conclusion

deploying a Kubernetes cluster on AWS with a focus on achieving high

availability, load balancing, and scaling of web server applications offers

numerous benefits but also presents certain challenges and limitations.

Throughout this project, we have explored the following key points:

1. Benefits: Kubernetes provides a powerful platform for automating the

deployment, scaling, and management of containerized applications. By

leveraging AWS infrastructure, we can take advantage of scalable compute

resources, managed services, and global reach to build resilient and efficient

Kubernetes clusters.

2. Challenges : Deploying and managing a Kubernetes cluster on

AWS involves addressing various challenges, including complexity, cost,

resource management, networking considerations, and security concerns. It

requires expertise in Kubernetes administration, AWS services, infrastructure

management, and DevOps practices.

3. Best Practices: Adhering to best practices for Kubernetes cluster design,

architecture, configuration, and operation is essential for ensuring reliability,

performance, and security. This includes proper resource allocation, networking

setup, security hardening, monitoring, logging, and disaster recovery planning.

4. Continuous Improvement: Deploying a Kubernetes cluster on AWS is an

iterative process that requires continuous monitoring, optimization, and refinement. By regularly reviewing cluster performance, analyzing metrics,

identifying bottlenecks, and implementing optimizations, we can enhance the

efficiency and resilience of our Kubernetes deployments over time.

In conclusion, deploying a Kubernetes cluster on AWS for high availability, load

balancing, and scaling of web server applications is a complex but rewarding

endeavor. By addressing challenges, leveraging best practices, and embracing

automation and scalability principles, organizations can build robust and resilient

Kubernetes environments that meet the demands of modern cloud-native

applications.