Deploy an Application to a Kubernetes Cluster using Workflow

Introduction

In modern software development, deploying applications manually on servers is inefficient and difficult to maintain. Applications must be portable, scalable, and highly available to handle real-time user traffic. To solve these challenges, containerization and container orchestration technologies are widely used in the industry. Docker is a containerization platform that allows applications and their dependencies to be packaged into lightweight containers. These containers ensure consistency across development, testing, and production environments. However, managing multiple containers manually becomes complex as the application grows. Kubernetes is a container orchestration platform that automates deployment, scaling, networking, and management of containerized applications. It ensures high availability by distributing application instances across multiple nodes and automatically recovering failed containers.

In this project, a web application named Akrithi is containerized using Docker and deployed on a Kubernetes cluster created using AWS EC2 virtual machines. The cluster consists of one master node and multiple worker nodes. Kubernetes services and Ingress controller are used to expose the application to external users.

The project demonstrates a real-world DevOps workflow including infrastructure setup, containerization, cluster configuration, application deployment, and external access through load-balanced routing. This implementation helps understand how modern cloud-native applications are deployed in production environments.

Problem Statement

Deploying applications on a single server leads to downtime, poor scalability, and difficult maintenance. As user traffic increases, manual management of multiple instances becomes complex and unreliable. Modern applications require high availability, easy updates, and consistent execution across environments. Therefore, a containerized and automated deployment system is needed to manage applications efficiently and ensure continuous availability.

Objective

The objective of this project is to deploy a containerized web application on a Kubernetes cluster created on AWS EC2 instances. The application is built using static web files and served through Docker containers. Kubernetes is used for orchestration, scaling, networking, and external access using NodePort and Ingress.

- To containerize the Akrithi web application using Docker
- To create a Kubernetes cluster using AWS EC2 instances
- To deploy and manage application pods using Kubernetes
- To expose the application using Service and Ingress
- To achieve scalability and high availability
- To demonstrate a real-time DevOps deployment workflow

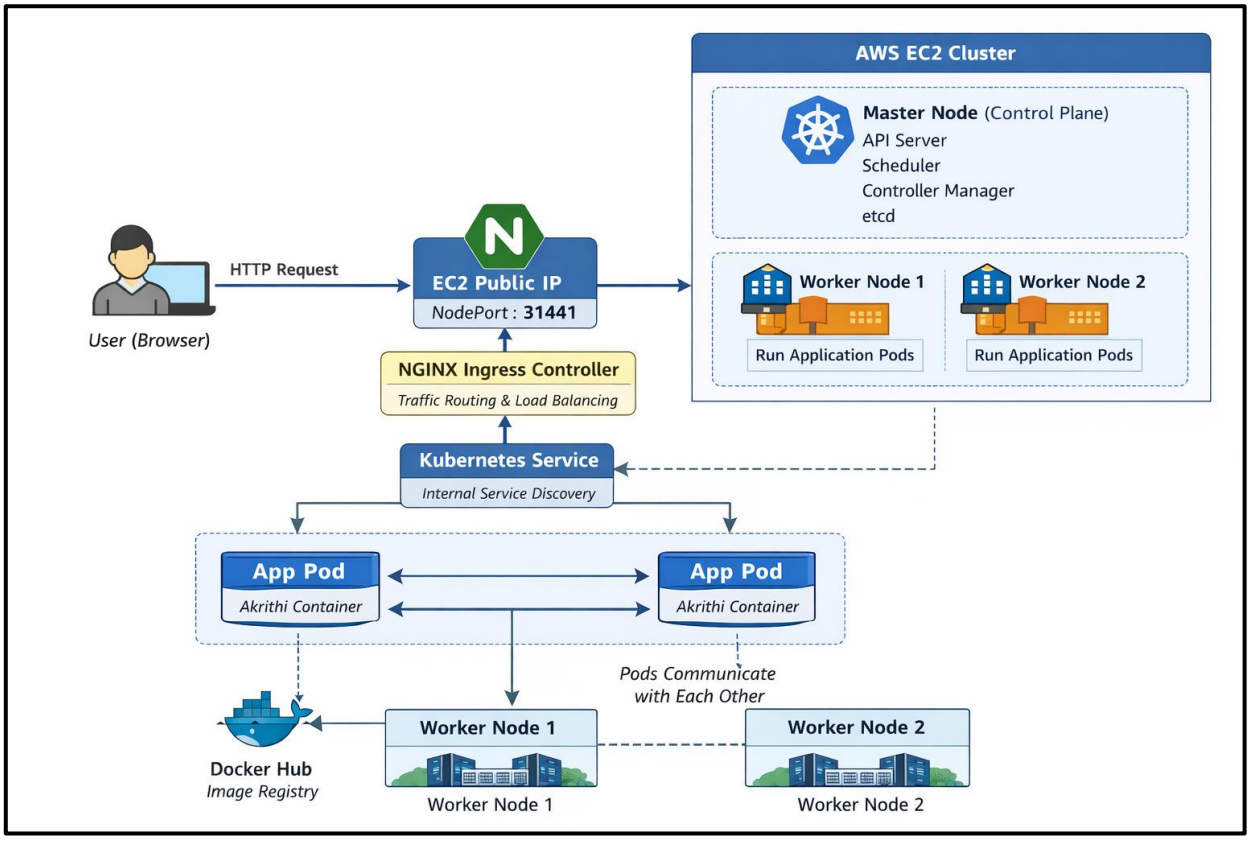
Architecture Overview

The system follows a client–server architecture deployed on a Kubernetes cluster hosted on AWS EC2 instances. The cluster consists of one master node and two worker nodes. The master node manages scheduling, cluster state, and networking, while worker nodes run the application containers inside pods.

The Akrithi web application is first containerized using Docker and stored in Docker Hub. Kubernetes pulls the image from Docker Hub and runs multiple replicas of the application as pods across worker nodes to ensure high availability.

An internal Kubernetes Service provides stable communication to the pods, and the Nginx Ingress Controller acts as an entry point for external users. User requests from the browser reach the Ingress, which routes traffic to the service and then to the running application pods.

This architecture ensures load balancing, fault tolerance, and scalable deployment of the application.



Components Used

The following components were used in the implementation of this project:

1. **Cloud Platform**
   - **AWS EC2** – Hosts Kubernetes cluster (1 Master + 2 Worker nodes)
2. **Containerization**
   - **Docker** – Builds container image of Akrithi website
   - **Docker Hub** – Stores the application image (adithi741/akrithi)
3. **Container Orchestration**
   - **Kubernetes (kubeadm cluster)** – Manages deployment, scaling and networking

- **kubectl** – Command-line tool to interact with cluster

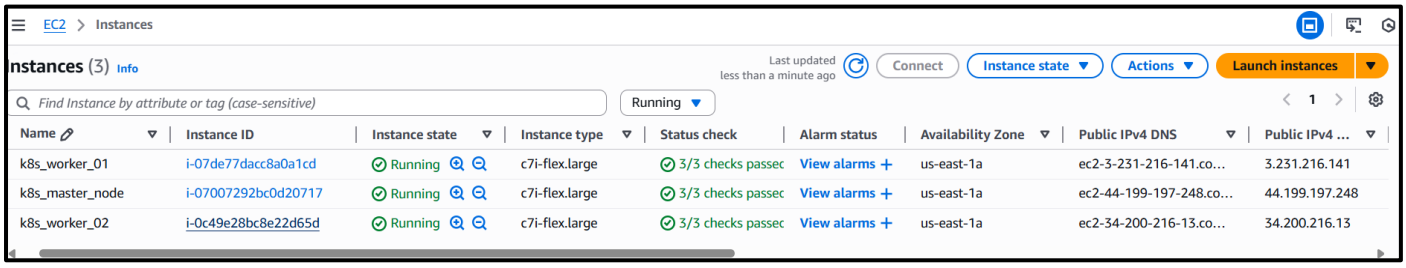
4. **Kubernetes Objects**
   - **Pod** – Runs Akrithi container
   - **Deployment** – Maintains replica pods (High availability)
   - **Service (NodePort)** – Exposes application externally
   - **Service (ClusterIP)** – Internal communication
   - **Ingress Controller (NGINX)** – HTTP routing & load balancing
   - **Ingress Resource** – URL-based routing rules
5. **Networking**
   - **Calico CNI Plugin** – Pod-to-Pod communication network
   - **Kube-proxy** – Service load balancing inside cluster
6. **Version Control**
   - **Git & GitHub** – Source code repository (Akrithi website)

Step-by-Step Implementation

Step 1: Create AWS EC2 Instances

a. Open AWS Console and go to EC2 service

- Launch 3 Ubuntu servers
  - 1 server → Master node (controls the cluster)
  - 2 servers → Worker nodes (run the website)
- Choose instance type: c7i-flex.large
- Create a key pair and download it (used to login)



b. Configure Security Group

Allow these ports:

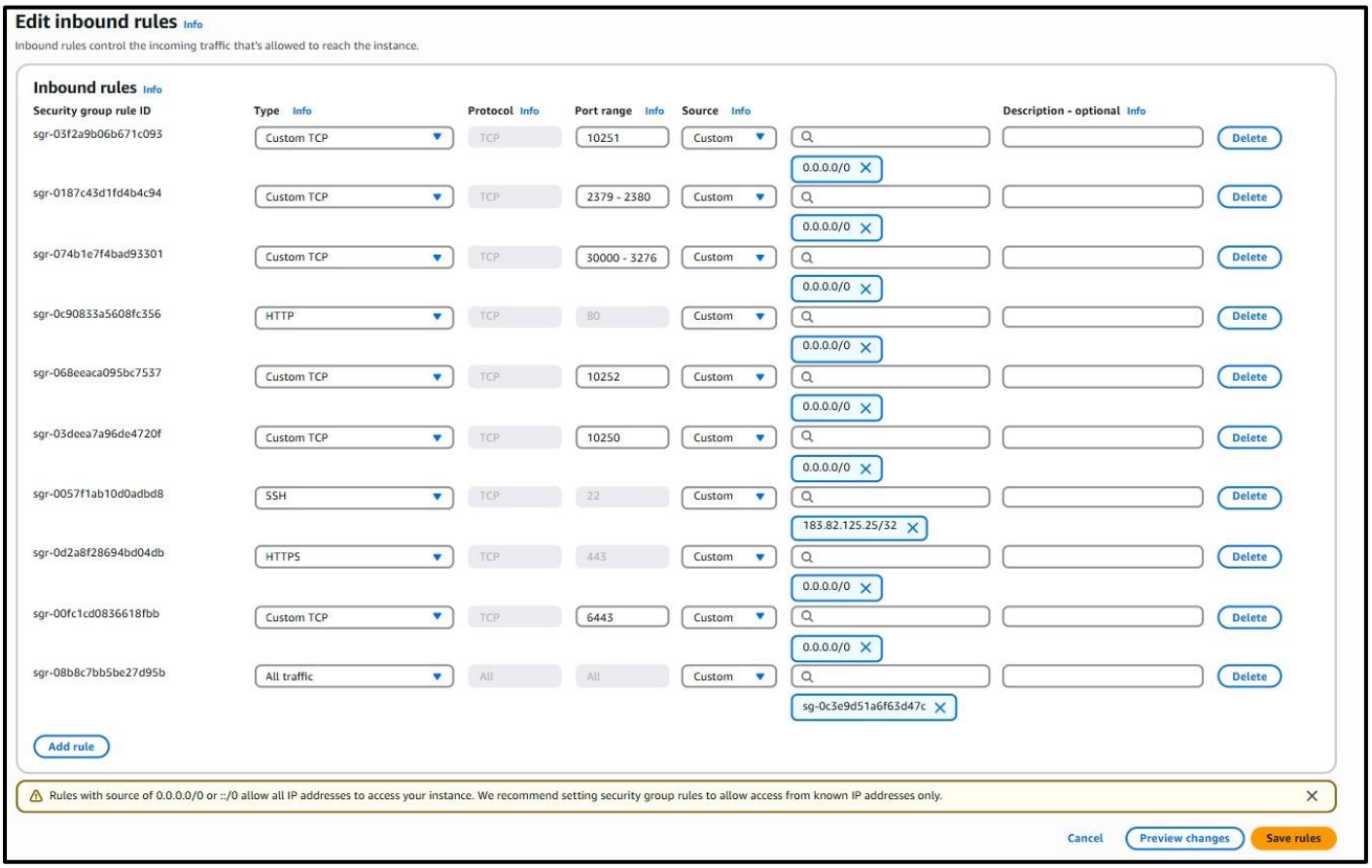
- **SSH (22)** → Your IP

  Used to connect to server using MobaXterm / SSH
- **Custom TCP (6443)** → Anywhere (0.0.0.0/0)

  Used for Kubernetes API communication (master ↔ workers)
- **Custom TCP (2379-2380)** → Anywhere

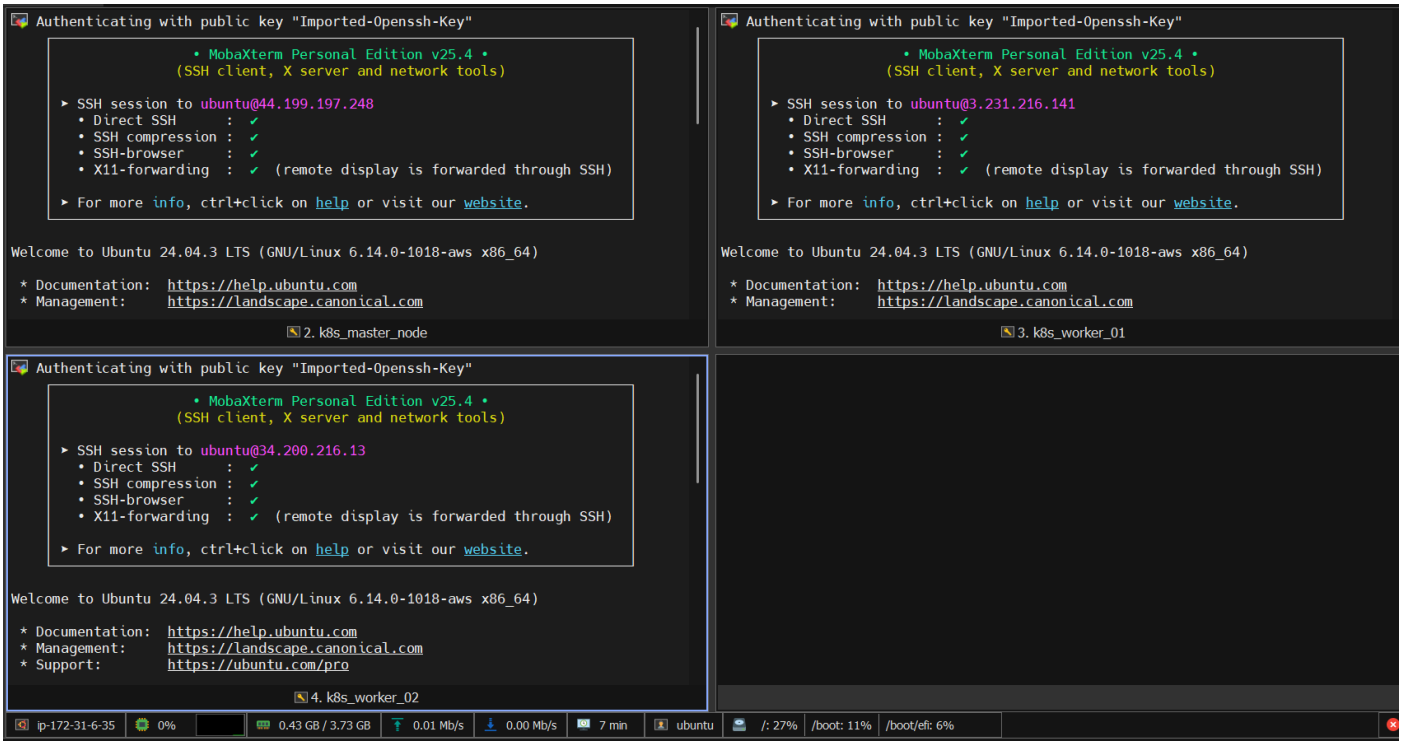
  Used by etcd database inside master node

- **Custom TCP (10250)** → Anywhere
  Used for communication between control plane and worker nodes
- **Custom TCP (10251)** → Anywhere
  Used by kube-scheduler
- **Custom TCP (10252)** → Anywhere
  Used by controller-manager
- **Custom TCP (30000-32767)** → Anywhere
  Required for NodePort services (to open website in browser)
- **HTTP (80)** → Anywhere
  Used for accessing application through Ingress
- **HTTPS (443)** → Anywhere
  Used when website is accessed securely using https://
- **All Traffic → Same Security Group**
  Allows communication between cluster nodes internally



c. Connect to servers

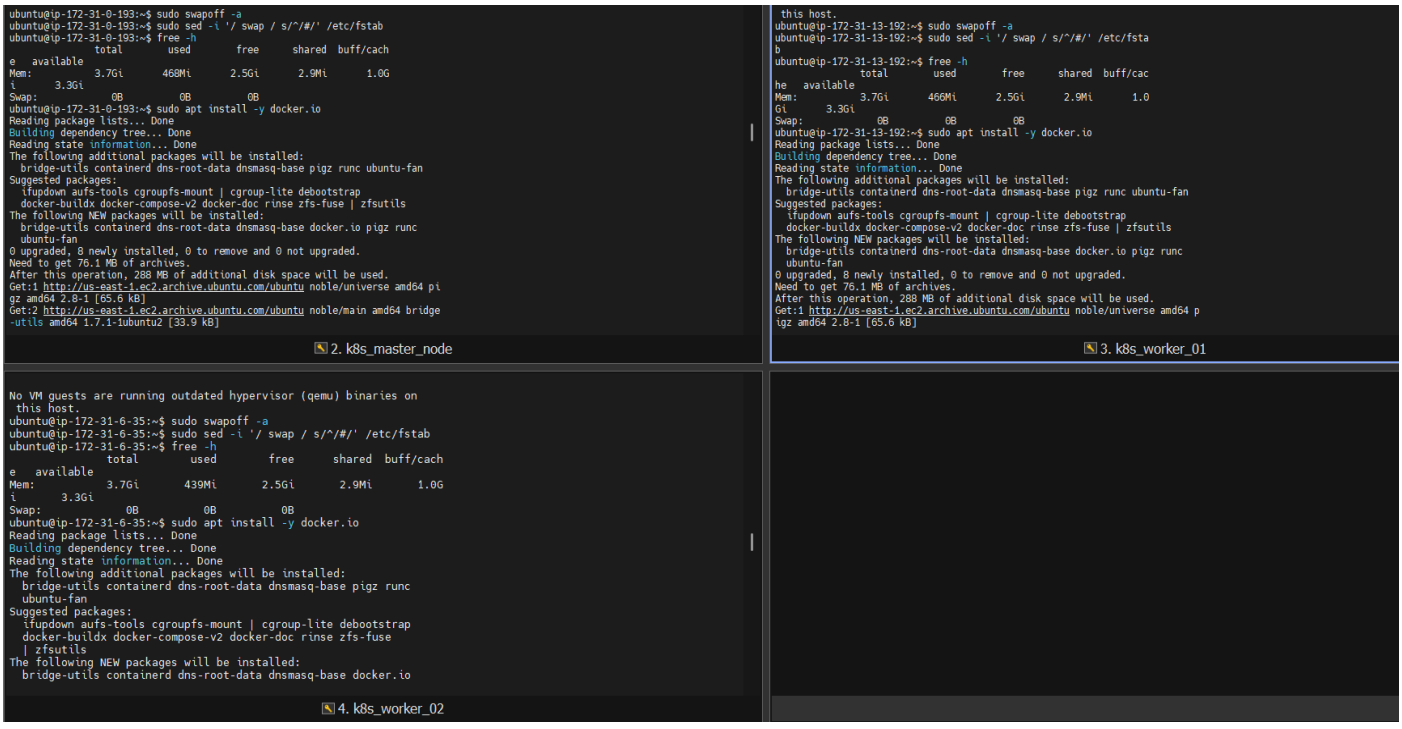
Use MobaXterm / SSH to connect to all 3 machines using public IP



Step 2: Install Docker

sudo swapoff -a

sudo sed -i '/ swap / s/^/#/' /etc/fstab



sudo apt update

sudo apt install docker.io -y

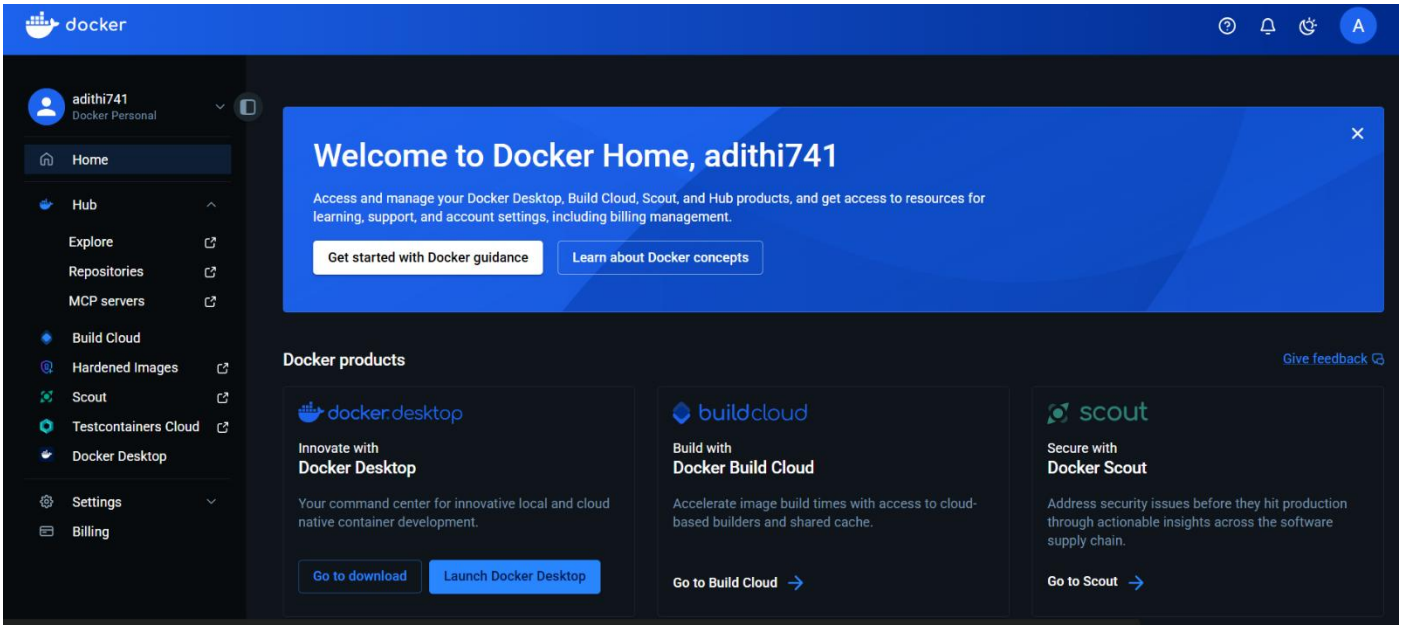
sudo systemctl enable docker

sudo systemctl start docker

```
ubuntu@ip-172-31-0-193:~$ sudo swapoff -a
ubuntu@ip-172-31-0-193:~$ sudo sed -i '/ swap / s/^/#/' /etc/fstab
ubuntu@ip-172-31-0-193:~$ free -h
               total        used        free      shared  buff/cach
e   available
Mem:           3.7Gi       468Mi       2.5Gi       2.9Mi       1.0G
i       3.3Gi
Swap:             0B          0B          0B
ubuntu@ip-172-31-0-193:~$ sudo apt install -y docker.io
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
The following additional packages will be installed:
  bridge-utils containerd dns-root-data dnsmasq-base pigz runc ubuntu-fan
Suggested packages:
  ifupdown aufs-tools cgroupfs-mount | cgroup-lite debootstrap
  docker-buildx docker-compose-v2 docker-doc rinse zfs-fuse | zfsutils
The following NEW packages will be installed:
  bridge-utils containerd dns-root-data dnsmasq-base docker.io pigz runc
  ubuntu-fan
0 upgraded, 8 newly installed, 0 to remove and 0 not upgraded.
```

```
ubuntu@ip-172-31-0-193:~$ sudo systemctl enable docker
ubuntu@ip-172-31-0-193:~$ sudo systemctl start docker
ubuntu@ip-172-31-0-193:~$ sudo mkdir -p /etc/containerd
ubuntu@ip-172-31-0-193:~$ containerd config default | sudo tee /etc/containe
rd/config.toml
disabled_plugins = []
imports = []
oom_score = 0
```

```
ubuntu@ip-172-31-0-193:~$ sudo nano /etc/containerd/config.toml
ubuntu@ip-172-31-0-193:~$ sudo systemctl restart containerd
ubuntu@ip-172-31-0-193:~$ sudo systemctl restart docker
ubuntu@ip-172-31-0-193:~$ sudo systemctl status containerd
● containerd.service - containerd container runtime
     Loaded: loaded (/usr/lib/systemd/system/containerd.service; enabled; p>
     Active: active (running) since Fri 2026-02-06 20:56:35 UTC; 13s ago
       Docs: https://containerd.io
    Process: 18598 ExecStartPre=/sbin/modprobe overlay (code=exited, status>
   Main PID: 18600 (containerd)
      Tasks: 7
     Memory: 13.6M (peak: 14.1M)
        CPU: 64ms
     CGroup: /system.slice/containerd.service
             └─18600 /usr/bin/containerd

Feb 06 20:56:35 ip-172-31-0-193 containerd[18600]: time="2026-02-06T20:56:3>
Feb 06 20:56:35 ip-172-31-0-193 containerd[18600]: time="2026-02-06T20:56:3>
Feb 06 20:56:35 ip-172-31-0-193 containerd[18600]: time="2026-02-06T20:56:3>
Feb 06 20:56:35 ip-172-31-0-193 containerd[18600]: time="2026-02-06T20:56:3>
Feb 06 20:56:35 ip-172-31-0-193 containerd[18600]: time="2026-02-06T20:56:3>
Feb 06 20:56:35 ip-172-31-0-193 containerd[18600]: time="2026-02-06T20:56:3>
Feb 06 20:56:35 ip-172-31-0-193 containerd[18600]: time="2026-02-06T20:56:3>
Feb 06 20:56:35 ip-172-31-0-193 containerd[18600]: time="2026-02-06T20:56:3>
Feb 06 20:56:35 ip-172-31-0-193 systemd[1]: Started containerd.service - co>
Feb 06 20:56:35 ip-172-31-0-193 containerd[18600]: time="2026-02-06T20:56:3>

ubuntu@ip-172-31-0-193:~$ sudo usermod -aG docker $USER
ubuntu@ip-172-31-0-193:~$ newgrp docker
ubuntu@ip-172-31-0-193:~$ docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES
ubuntu@ip-172-31-0-193:~$ sudo systemctl status containerd | grep running
     Active: active (running) since Fri 2026-02-06 20:56:35 UTC; 6min ago
ubuntu@ip-172-31-0-193:~$ sudo apt install -y apt-transport-https ca-certifi
cates curl gpg
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
ca-certificates is already the newest version (20240203).
ca-certificates set to manually installed.
```



Step 3: Install Kubernetes Components (All Nodes)

Kubernetes prerequisites and tools such as kubeadm, kubelet, and kubectl are installed on both the master and worker nodes. Swap memory is disabled to ensure proper Kubernetes operation. These components provide the foundation required to create and manage the Kubernetes cluster.

- sudo apt install -y apt-transport-https ca-certificates curl gpg
- curl -fsSL https://pkgs.k8s.io/core:/stable:/v1.29/deb/Release.key \

| sudo gpg --dearmor -o /etc/apt/keyrings/kubernetes-apt-keyring.gpg

- echo "deb [signed-by=/etc/apt/keyrings/kubernetes-apt-keyring.gpg] \

https://pkgs.k8s.io/core:/stable:/v1.29/deb/ /" \

| sudo tee /etc/apt/sources.list.d/kubernetes.list

```
ubuntu@ip-172-31-0-193:~$ curl -fsSL https://pkgs.k8s.io/core:/stable:/v1.29
/deb/Release.key \
| sudo gpg --dearmor -o /etc/apt/keyrings/kubernetes-apt-keyring.gpg
ubuntu@ip-172-31-0-193:~$ echo "deb [signed-by=/etc/apt/keyrings/kubernetes-
apt-keyring.gpg] \
https://pkgs.k8s.io/core:/stable:/v1.29/deb/ /" \
| sudo tee /etc/apt/sources.list.d/kubernetes.list
deb [signed-by=/etc/apt/keyrings/kubernetes-apt-keyring.gpg] https://pkgs.k8
s.io/core:/stable:/v1.29/deb/ /
ubuntu@ip-172-31-0-193:~$ sudo apt update
Hit:1 http://us-east-1.ec2.archive.ubuntu.com/ubuntu noble InRelease
Get:2 http://us-east-1.ec2.archive.ubuntu.com/ubuntu noble-updates InRelease
 [126 kB]
```

```
ubuntu@ip-172-31-0-193:~$ sudo apt-mark hold kubelet kubeadm kubectl
kubelet set on hold.
kubeadm set on hold.
kubectl set on hold.
ubuntu@ip-172-31-0-193:~$ sudo systemctl enable kubelet
ubuntu@ip-172-31-0-193:~$ kubeadm version
kubeadm version: &version.Info{Major:"1", Minor:"29", GitVersion:"v1.29.15",
 GitCommit:"0d0f172cdf9fd42d6feee3467374b58d3e168df0", GitTreeState:"clean",
 BuildDate:"2025-03-11T17:46:36Z", GoVersion:"go1.23.6", Compiler:"gc", Plat
form:"linux/amd64"}
ubuntu@ip-172-31-0-193:~$ kubectl version --client
Client Version: v1.29.15
```

Step 4: Initializing the Master Master Node

The Java web application source code was cloned from a GitHub repository using Git.

The project structure was reviewed to confirm all required files were present before building the application.

- sudo kubeadm init --pod-network-cidr=192.168.0.0/16
- mkdir -p $HOME/.kube

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

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

```
ubuntu@ip-172-31-0-193:~$ sudo kubeadm init --pod-network-cidr=192.168.0.0/1
6
I0206 21:10:14.484081   22216 version.go:256] remote version is much newer:
v1.35.0; falling back to: stable-1.29
[init] Using Kubernetes version: v1.29.15
[preflight] Running pre-flight checks
[preflight] Pulling images required for setting up a Kubernetes cluster
```

```
kubeadm join 172.31.0.193:6443 --token 32yee3.k8d8ruw8xdle6md6 \
        --discovery-token-ca-cert-hash sha256:2d6777a9b3675725278f82146aa7c3
5a8d9e5ecd46ab77e9fccd3734106263f3
ubuntu@ip-172-31-0-193:~$ mkdir -p $HOME/.kube
ubuntu@ip-172-31-0-193:~$ sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/
config
ubuntu@ip-172-31-0-193:~$ sudo chown $(id -u):$(id -g) $HOME/.kube/config
ubuntu@ip-172-31-0-193:~$ kubectl get nodes
NAME               STATUS     ROLES           AGE    VERSION
ip-172-31-0-193    NotReady   control-plane   109s   v1.29.15
ubuntu@ip-172-31-0-193:~$ kubectl apply -f https://raw.githubusercontent.com
/projectcalico/calico/v3.27.0/manifests/calico.yaml
poddisruptionbudget.policy/calico-kube-controllers created
```

Step 5: Joining Worker Node to Master Node

The Kubernetes master node is initialized using kubeadm, which sets up the control plane components. A pod network CIDR is specified, and a network plugin is installed to enable communication between pods. This step establishes the core Kubernetes cluster.

- kubeadm join 172.31.xx.xx:6443 --token abcdef.123456789 \

--discovery-token-ca-cert-hash sha256:xxxxxxxxxxxxxxxx

- kubectl get nodes

```
ubuntu@ip-172-31-0-193:~$ sudo modprobe overlay
ubuntu@ip-172-31-0-193:~$ sudo modprobe br_netfilter
ubuntu@ip-172-31-0-193:~$ cat <<EOF | sudo tee /etc/modules-load.d/k8s.conf
overlay
br_netfilter
```

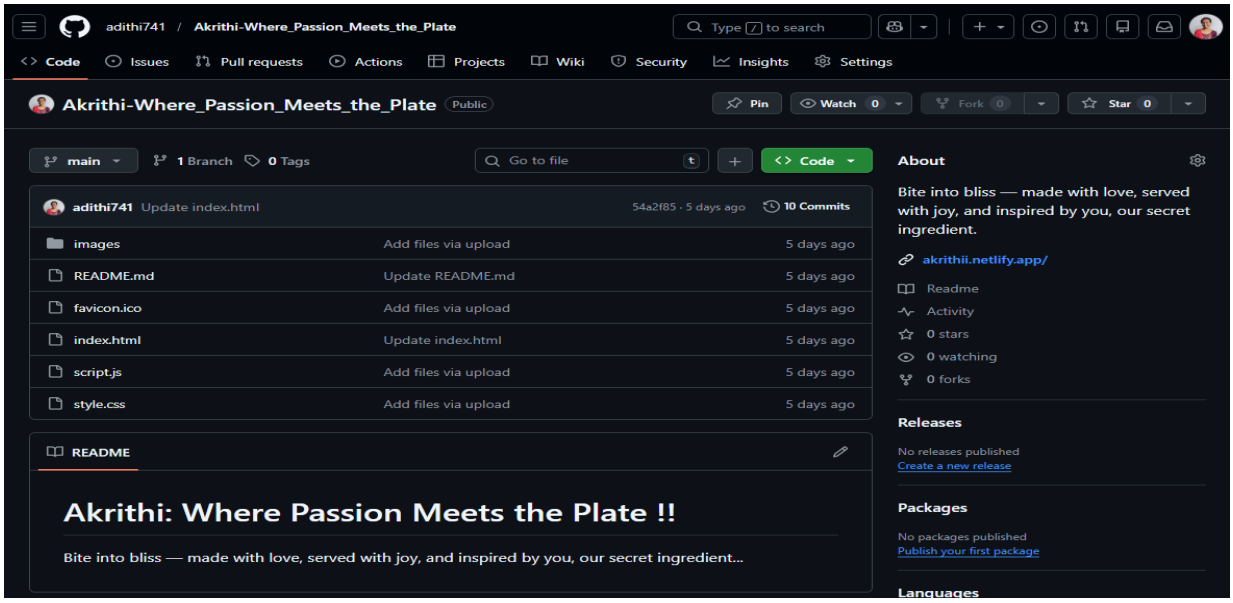
```
ubuntu@ip-172-31-13-192:~$ ls /proc/sys/net/bridge
bridge-nf-call-arptables  bridge-nf-filter-pppoe-tagged
bridge-nf-call-ip6tables  bridge-nf-filter-vlan-tagged
bridge-nf-call-iptables   bridge-nf-pass-vlan-input-dev
ubuntu@ip-172-31-13-192:~$ sudo kubeadm join 172.31.0.193:6443 --token 32ye
e3.k8d8ruw8xdle6md6 \
--discovery-token-ca-cert-hash sha256:2d6777a9b3675725278f82146aa7c35a8d9e5
ecd46ab77e9fccd3734106263f3
[preflight] Running pre-flight checks
```

```
Your Kubernetes control-plane has initialized successfully!

To start using your cluster, you need to run the following as a regular user
:

  mkdir -p $HOME/.kube
  sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
  sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

```
ubuntu@ip-172-31-0-193:~$ kubectl get pods -n kube-system
NAME                                       READY   STATUS    RESTARTS   AGE
calico-kube-controllers-5fc7d6cf67-sbwpg   1/1     Running   0          34s
calico-node-rxtnn                          1/1     Running   0          34s
coredns-76f75df574-gk6cj                   1/1     Running   0          2m17
s
coredns-76f75df574-hpl8l                   1/1     Running   0          2m17
s
etcd-ip-172-31-0-193                       1/1     Running   0          2m32
s
kube-apiserver-ip-172-31-0-193             1/1     Running   0          2m32
s
kube-controller-manager-ip-172-31-0-193    1/1     Running   0          2m32
s
kube-proxy-mm2kr                           1/1     Running   0          2m18
s
kube-scheduler-ip-172-31-0-193             1/1     Running   0          2m33
s
ubuntu@ip-172-31-0-193:~$ kubectl get nodes
NAME              STATUS   ROLES           AGE     VERSION
ip-172-31-0-193   Ready    control-plane   2m59s   v1.29.15
ubuntu@ip-172-31-0-193:~$ kubectl get nodes
NAME              STATUS   ROLES           AGE     VERSION
ip-172-31-0-193   Ready    control-plane   6m23s   v1.29.15
ip-172-31-13-192  Ready    <none>          14s     v1.29.15
ip-172-31-6-35    Ready    <none>          11s     v1.29.15
```

Step 6: Clone Application Code

The application source code is cloned from GitHub to the master node. GitHub acts as the version control system and stores application files, Docker configuration, and Helm charts required for deployment.

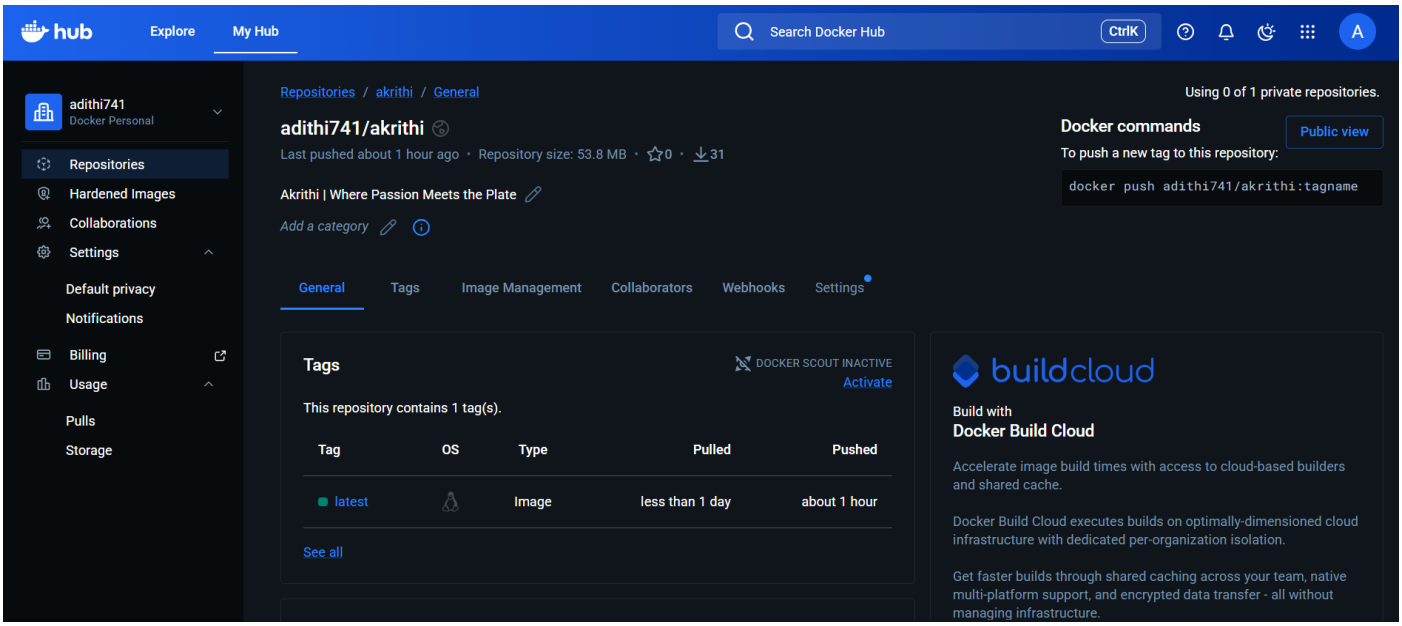


```
sudo apt install git -y
cd ~
git clone https://github.com/adithi741/Akrithi-Where_Passion_Meets_the_Plate.git
cd Akrithi-Where_Passion_Meets_the_Plate
ls
```

Step 7: Build and Pushing Docker Image

A Docker image is built using the application source code and pushed to Docker Hub. The image serves as the deployable artifact that Kubernetes pulls during application deployment.

```
nano Dockerfile
docker build -t adithi741/akrithi:latest .
docker login -u adithi741
docker push adithi741/akrithi:latest
```



Step 8: Kubernetes Deployment & Service

Kubernetes deployment and service manifests were created and applied. The deployment ensured multiple replicas of the application, while the service enabled internal communication.

```
nano deployment.yaml
nano service.yaml
kubectl apply -f deployment.yaml
kubectl apply -f service.yaml
kubectl get pods -o wide
kubectl get svc akrithi-service
```

Step 9: Install NGINX Ingress Controller

The NGINX Ingress Controller was installed to manage external HTTP traffic and route requests to the application service.

```
  55  kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/main/
deploy/static/provider/cloud/deploy.yaml
  56  kubectl get pods -n ingress-nginx
```

Step 10: Configure Ingress Resource

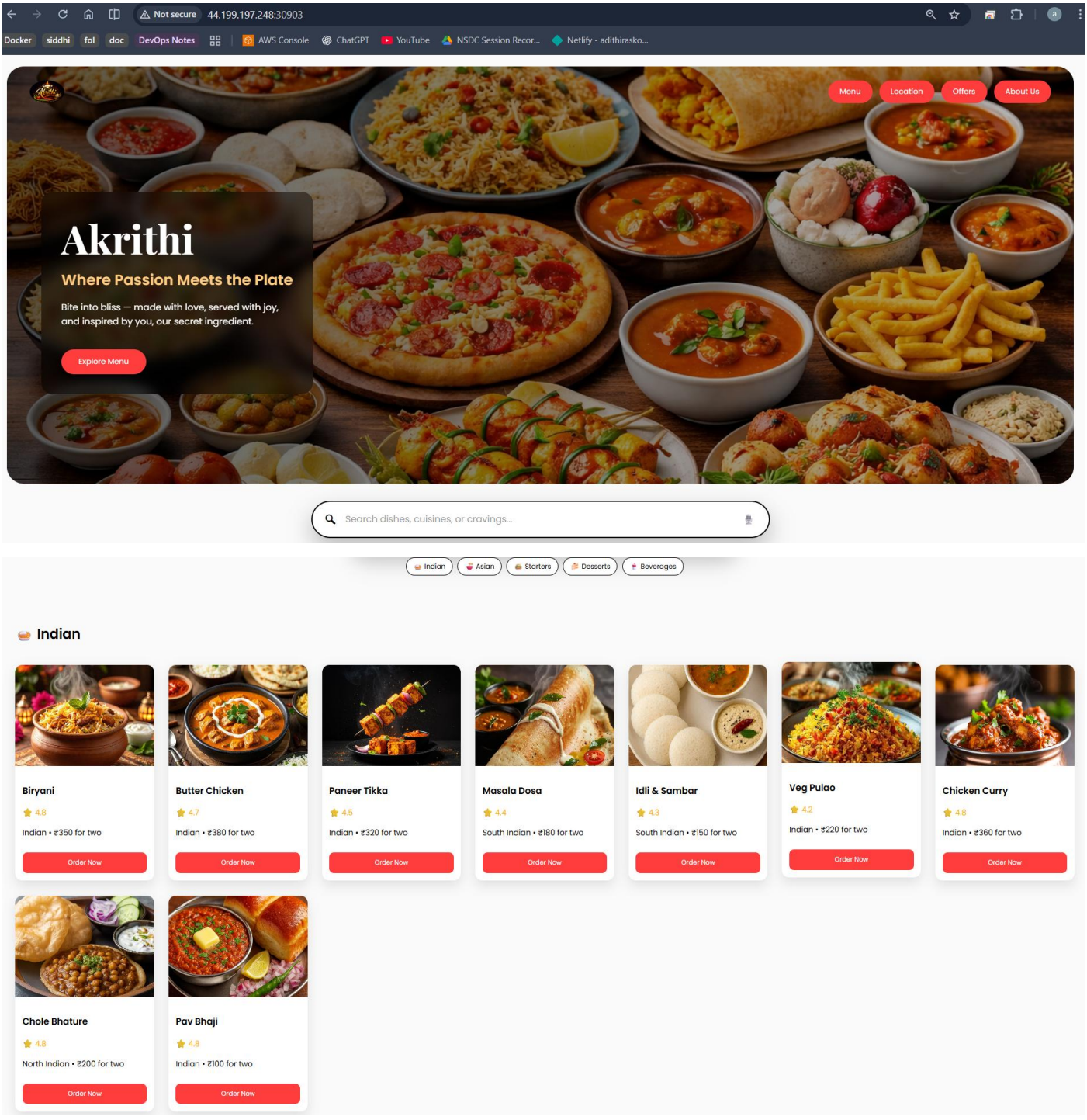
An ingress resource was created to define routing rules that forward external requests to the application service.

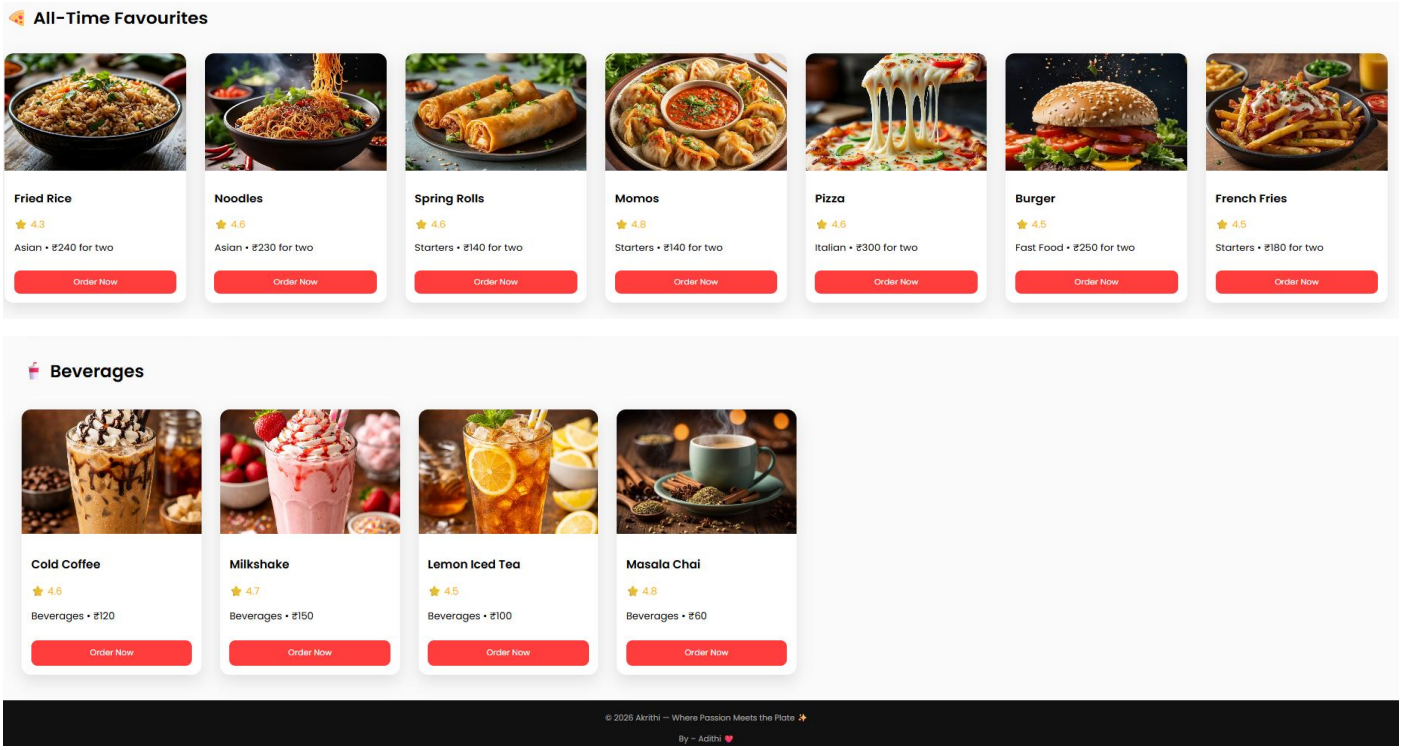
```
nano ingress.yaml
kubectl apply -f ingress.yaml
kubectl get ingress
```

Step 11: Access Application

The ingress service NodePort was identified, and the application was accessed using the worker node's public IP address and exposed port.

```
kubectl get svc -n ingress-nginx
```





Project Outcome

The project successfully deployed a containerized application to a Kubernetes cluster using a CI/CD workflow. Docker was used to package the application, Kubernetes managed deployment and scalability, Helm simplified resource management, and NGINX Ingress enabled external access. The implementation demonstrated a complete cloud-native deployment workflow with reliable application availability and scalability.

Conclusion

This project demonstrated the successful deployment of a containerized application on a Kubernetes cluster using a CI/CD workflow. By integrating Docker for containerization, Kubernetes for orchestration, Helm for simplified deployment, and NGINX Ingress for external access, the project showcased key DevOps and cloud-native concepts. The implementation highlights how modern deployment workflows improve application scalability, reliability, and manageability in real-world environments.

By -
R. Adithi
Aspiring DevOps Engineer
Batch - 06