End-to-End CI/CD Pipeline for Secure Deployment of Wisecow on Kubernetes with TLS Encryption

This repository contains the Wisecow application, a simple web-based service that combines the fortune and cowsay utilities to display random quotes in a fun, ASCII-art format. This README provides instructions for cloning, testing, containerizing, deploying to Kubernetes using K3s, and setting up a CI/CD pipeline with GitHub Actions.

Prerequisites

- Git installed on your system
- Docker installed for containerization
- Docker Hub account for pushing images
- K3s for Kubernetes deployment
- GitHub account for repository and CI/CD setup
- Basic knowledge of terminal commands, Docker, and Kubernetes

Cloning the Repository

I started by cloning the project repository:

```
git clone https://github.com/nyrahul/wisecow
```

Next, I created my own GitHub repository:

https://github.com/shefeekar/wise-cow

I then pushed the project files into my repository.

Before proceeding further, I verified that the application worked correctly in my local environment. Following the repository instructions, I installed the required

dependencies:

```
sudo apt install fortune-mod cowsay -y
```

After installation, I confirmed that the program was running successfully at:

http://localhost:4499

The application worked as expected.

---

containerising the cowsay application

```
FROM ubuntu:latest
WORKDIR /app
COPY .  /app
RUN  apt-get update && apt-get install -y fortune-mod cowsay netcat-open
bsd && rm -rf /var/lib/apt/lists/*
RUN  chmod +x /app/wisecow.sh
ENV PATH="/usr/games:${PATH}"
EXPOSE 4499
CMD ["bash", "wisecow.sh"]
```

Install K3s and Deploy an Application on Kubernetes

K3s is a lightweight Kubernetes distribution, mainly used for edge, IoT, and small-resource environments. It provides a fully functional Kubernetes cluster with minimal setup.

`curl -sfL https://get.k3s.io | sh -`

`sudo systemctl status k3s`

check the status and controll nod is working

`kubectl get nodes -o wide`

NAME STATUS ROLES AGE VERSION INTERNAL-IP EXTERNAL-IP OS-IMAGE KERNEL-VERSION CONTAINER-RUNTIME
mail.shefeekar.online Ready control-plane,master 30h v1.33.4+k3s1 10.126.5.131 <none> Ubuntu 24.04.3 LTS 6.14.0-32-generic containerd://2.0.5-k3s2

deploy wise cow application using k8 deployment template

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: wisecow-deployment
  labels:
    app: wisecow
spec:
  replicas: 3
  selector:
    matchLabels:
      app: wisecow
  template:
    metadata:
      labels:
        app: wisecow
    spec:
      containers:
      - name: wisecow
        image: shefeekar/wisecow:latest
        imagePullPolicy: Always
        ports:
        - containerPort: 4499
```

and the serviece template called service.yml create

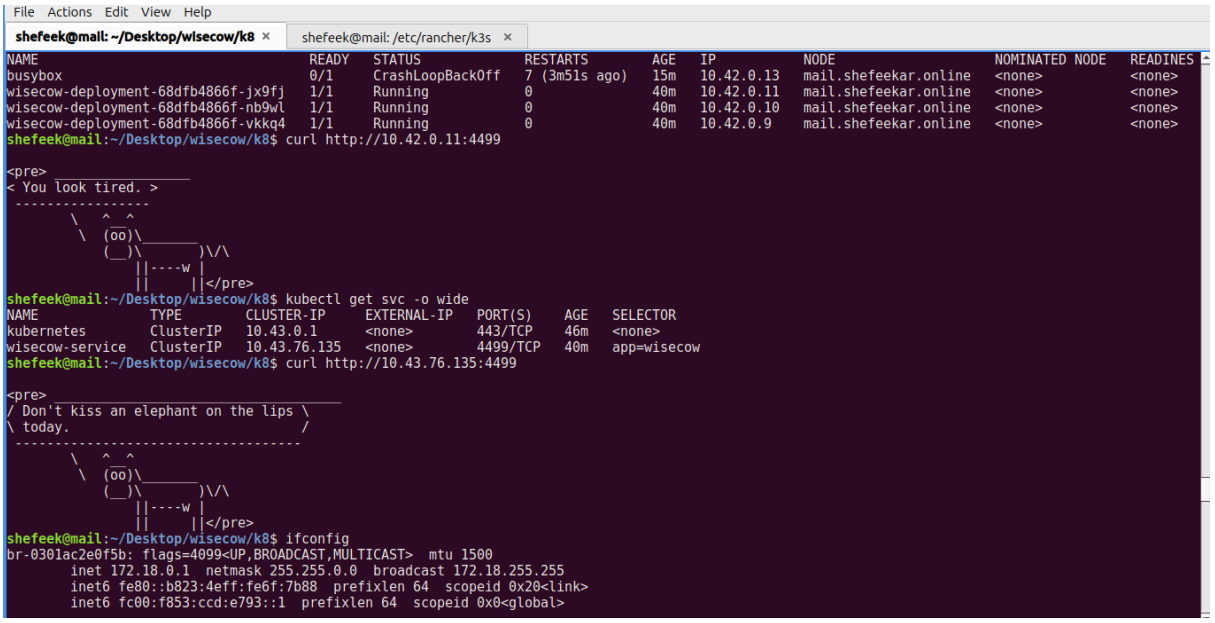
```
apiVersion: v1
kind: Service
metadata:
  name: wisecow-service
spec:
  selector:
    app: wisecow
```

```
  type: NodePort
  ports:
    - protocol: TCP
      port: 4499
      targetPort: 4499
      nodePort: 30499
```

Apply the Deployment

`kubectl apply -f  deployment.yml`

`kubectl apply -f service.yml`

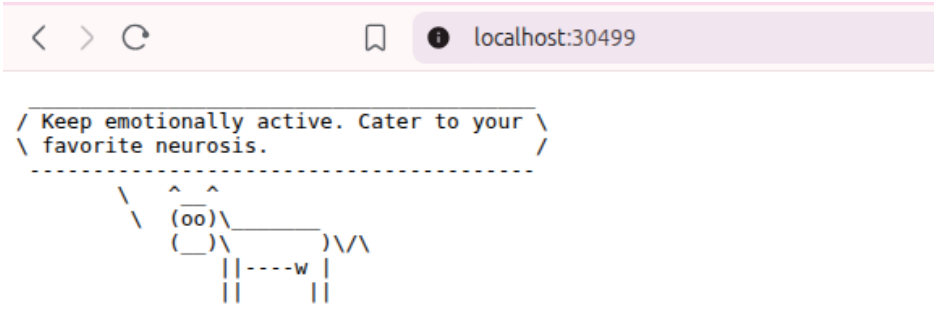


ensure that the application worked as expected

check connection to serviece and connection to the pod ip

`curl http://10.42.0.11:4499`

`curl` `http://10.43.76.135:4499`



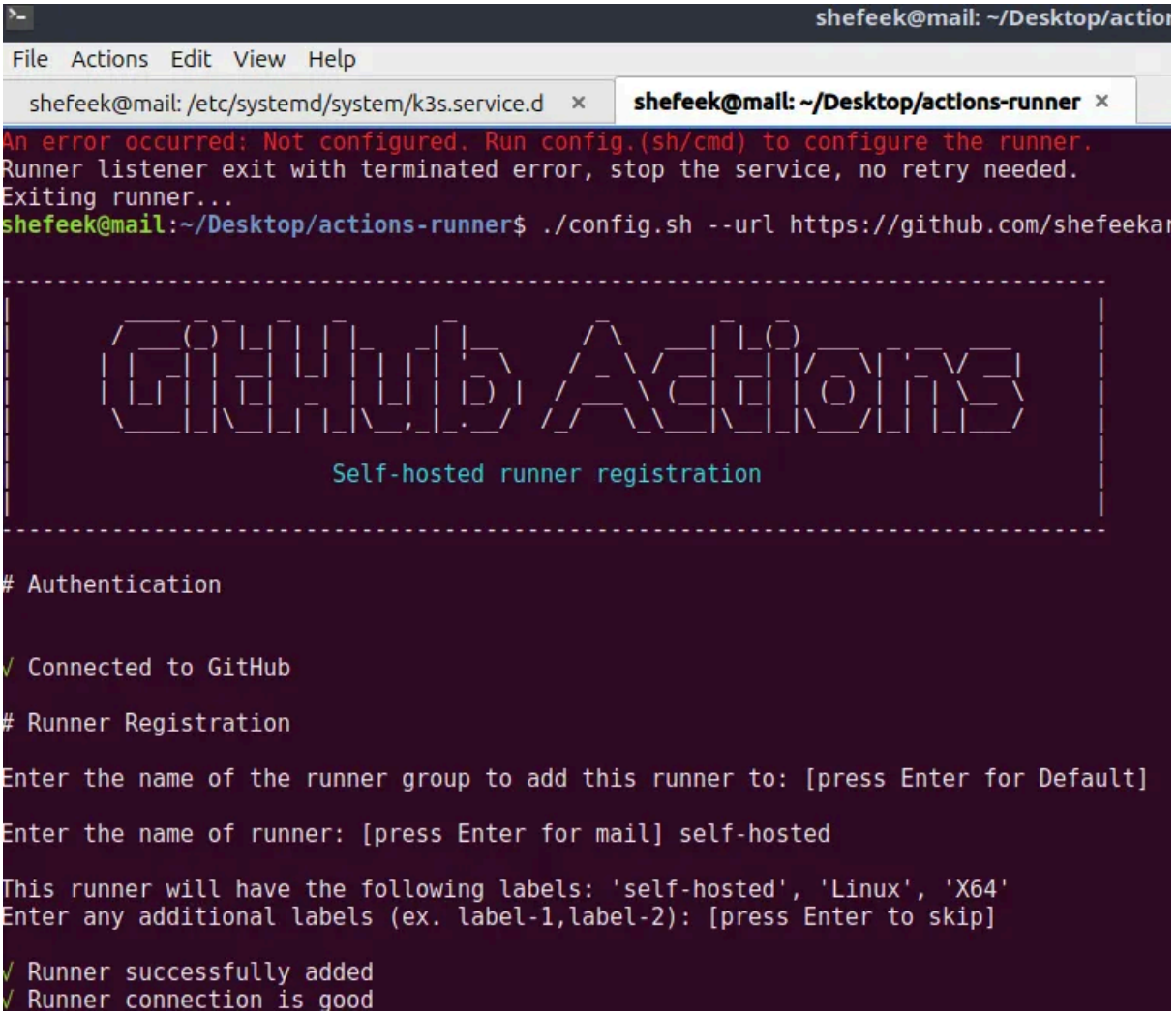
ensure the notde port is correctly serving the content in the port localhost:30499

setup git hub selfhosted runner

Since I don’t have access to a public cloud environment for deploying my application, I will set up a **self-hosted runner** in GitHub Actions. This runner will execute my workflows directly on my own machine. By doing so, I can use the GitHub Actions workflow file to build and deploy the application locally through the self-hosted runner.

Building a self-hosted GitHub Actions runner according to the official GitHub documentation.

https://docs.github.com/en/actions/how-tos/manage-runners/self-hosted-runners/add-runners



Create CI/CD Using GitHub Workflows

First, generate a new access token for Docker Hub. This token will be used for authentication when pushing images. Store the token securely in **GitHub Actions Secrets**, along with your Docker Hub username.

Next, configure a GitHub Actions workflow that builds a Docker image and pushes it to your Docker Hub repository whenever code is pushed to the `master` branch.

This ensures that the CI/CD pipeline automatically builds and publishes the latest image to Docker Hub using your stored credentials.

The workflow executes two main jobs sequentially: `docker` (for CI: build and push) and `deploy` (for CD: deployment to K8s).

1. The `docker` Job (CI - Build and Push)

- **Goal:** Build the Docker image for the application and push it to Docker Hub.
- **Environment:** Runs on a standard `ubuntu-latest` GitHub-hosted runner.
- **Steps:**
  - **Checkout:** Fetches the code from the repository.
  - **Login to Docker Hub:** Authenticates using a **variable** for the username (`vars.DOCKERHUB_USERNAME`) and a **secret** for the password/token (`secrets.DOCKERHUB_TOKEN`).
  - **Setup QEMU and Buildx:** Configures the environment to allow building multi-architecture Docker images.
  - **Build and Push:** Builds the Docker image based on the checked-out code and pushes it to the Docker repository `shefeekar/wisecow`.
  - **Image Tags:** The image is tagged with two versions: `latest` and the specific **Git commit SHA** (`github.sha`), ensuring both a stable and a unique version are available.

2. The `deploy` Job (CD - Kubernetes Deployment)

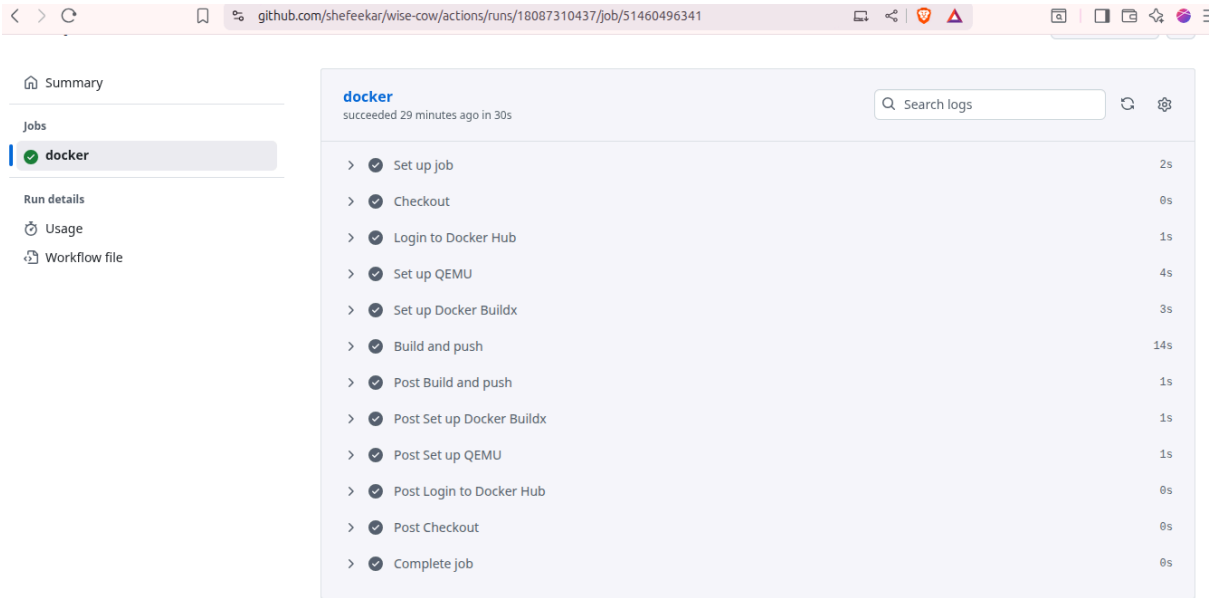
- **Goal:** Deploy the newly pushed Docker image to a Kubernetes cluster.
- **Prerequisite:** This job `needs: docker` and will only start after the `docker` job successfully completes.
- **Environment:** Runs on a `self-hosted` runner, which is assumed to have access to the target Kubernetes cluster (likely **K3s** based on the config paths).
  - It is associated with a `production` environment, with the provided URL as `http://localhost:30499`.
- **Steps:**
  - **Checkout Repository:** Fetches the repository code again (needed for the manifest files).
  - **Prepare Kubeconfig:** This critical step **configures access to the K3s cluster** by:
    - Copying the K3s configuration file (`/etc/rancher/k3s/k3s.yaml`) to the default Kubernetes configuration location (`~/.kube/config`).

        - Setting the correct **ownership** (`chown`) and **permissions** (`chmod 600`) to ensure the runner user can access it.
    - **Verify Kubeconfig Access:** Checks the cluster connection by running `kubectl cluster-info` and `kubectl get nodes` .
    - **Apply Kubernetes Manifests:** Uses `kubectl apply` to deploy or update the application using the configuration files located at specific absolute paths:
        - `/home/shefeek/Desktop/wisecow/k8/deployment.yml`
        - `/home/shefeek/Desktop/wisecow/k8/service.yml`
    - **Wait for Deployment Rollout:** Waits for up to 5 minutes to confirm that the `wisecow-deployment` is fully updated and all new pods are running successfully.
    - **Post-Deployment Verification:** Fetches and displays details about the `wisecow-service` .

```
name: CI/CD Pipeline

on:
  push:
    branches:
      - master

jobs:
  docker:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout
        uses: actions/checkout@v4

      - name: Login to Docker Hub
        uses: docker/login-action@v3
        with:
          username: ${{ vars.DOCKERHUB_USERNAME }}
          password: ${{ secrets.DOCKERHUB_TOKEN }}

      - name: Set up QEMU
```

```
      uses: docker/setup-qemu-action@v3

    - name: Set up Docker Buildx
      uses: docker/setup-buildx-action@v3

    - name: Build and Push
      uses: docker/build-push-action@v6
      with:
        context: .
        push: true
        tags: |
          shefeekar/wisecow:latest
          shefeekar/wisecow:${{ github.sha }}

  deploy:
    runs-on: self-hosted
    needs: docker
    environment:
      name: production
      url: http://localhost:30499

    steps:
      - name: Checkout Repository
        uses: actions/checkout@v4

      - name: Prepare Kubeconfig
        # This step fixes the "permission denied" error by copying the K3s co
nfig
        # to the runner's default KUBECONFIG location (~/.kube/config) and
        # setting the necessary permissions for the runner user.
        run: |
          sudo mkdir -p ~/.kube
          sudo cp /etc/rancher/k3s/k3s.yaml ~/.kube/config
          sudo chown $(id -u):$(id -g) ~/.kube/config
          sudo chmod 600 ~/.kube/config

      - name: Verify Kubeconfig Access
        # KUBECONFIG export is no longer needed since the file is in the defa
```

```
ult location
        run: |
          kubectl cluster-info
          kubectl get nodes

      - name: Apply Kubernetes Manifests
        run: |
          kubectl apply -f  k8/deployment.yml
          kubectl apply -f  k8/serviece.yml
      - name: Wait for Deployment Rollout
        run: |
          kubectl rollout status deployment/wisecow-deployment --timeout=5
m

      - name: Post-Deployment Verification
        run: |
          kubectl get svc wisecow-service -o wide
```



TLS Implementation

i didn’t have domain

I am adding an **NGINX reverse proxy sidecar container** with a self-signed TLS certificate to my existing *wisecow* Deployment.

This setup allows HTTPS traffic to terminate at NGINX before being forwarded to the main *wisecow* application container.

Since I don't have a domain, I am using a **self-signed TLS certificate** to overcome that limitation. The first step is to generate the self-signed certificate.

Step 1 — Create a Self-Signed Certificate

```
openssl req -x509 -nodes -days 365 \
-newkey rsa:2048 \
-keyout tls.key \
-out tls.crt \
-subj "/CN=wisecow.local/O=wisecow"
```

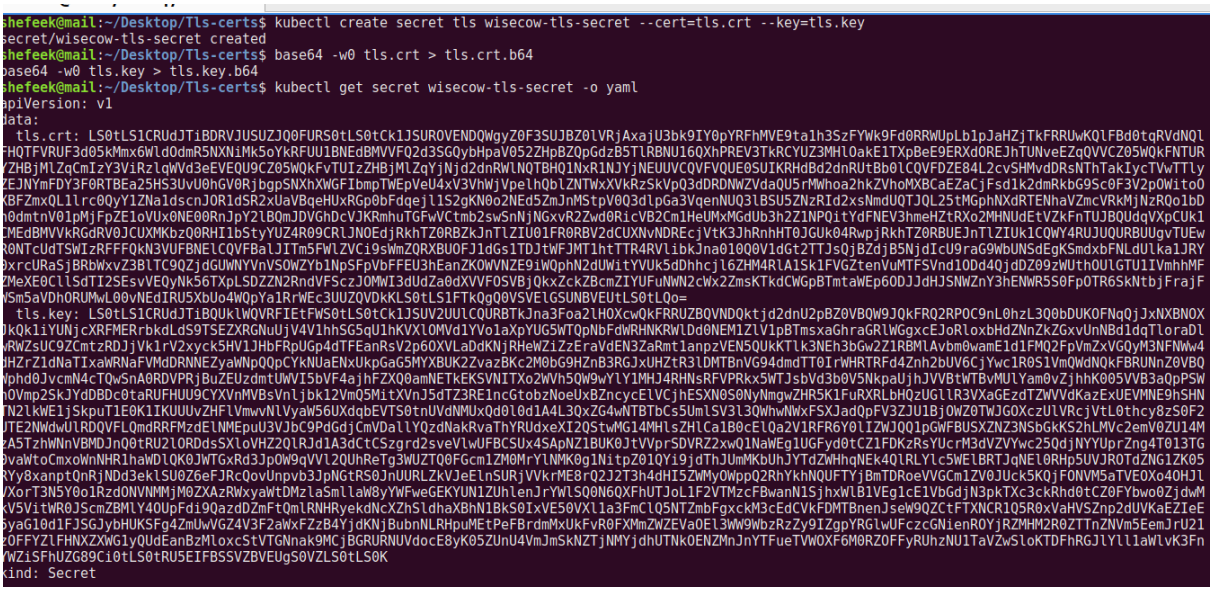
Then create a Kubernetes secret:

kubectl create secret tls wisecow-tls-secret \
--cert=tls.crt \
--key=tls.key

enable base 64 encryption

```
base64 -w0 tls.crt > tls.crt.b64
base64 -w0 tls.key > tls.key.b64
```

exract yaml file and save it name ***"secret.yml"***



Create an NGINX ConfigMap

I am creating a **ConfigMap** to store the NGINX configuration, which will be mounted into the pod so that the NGINX sidecar can use it.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: nginx-config
data:
  default.conf: |
    server {
      listen 443 ssl;
      server_name _;

      ssl_certificate /etc/nginx/ssl/tls.crt;
      ssl_certificate_key /etc/nginx/ssl/tls.key;

      location / {
        proxy_pass http://127.0.0.1:4499;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
```

```
    }
  }
```

```
shefeek@mail:~/Downloads/wise-cow/k8$ kubectl describe configmap nginx-config
Name:         nginx-config
Namespace:    default
Labels:       <none>
Annotations:  <none>

Data
====
default.conf:
----
server {
  listen 443 ssl;
  server_name _;

  ssl_certificate /etc/nginx/ssl/tls.crt;
  ssl_certificate_key /etc/nginx/ssl/tls.key;

  location / {
    proxy_pass http://127.0.0.1:4499;
    proxy_set_header Host $host;
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header X-Forwarded-Proto $scheme;
  }
}


BinaryData
====

Events:  <none>
```

Update the wisecow deployment yaml to append nginx revproxy

there are two containers. The first one is actual `wisecow` application, which is running on port 4499. The second one is an `nginx` sidecar container.

That NGINX container is acting like a reverse proxy. It listens on port 443, which is the HTTPS port. To handle HTTPS properly, it needs a certificate and a config file. So, Kubernetes mounts a TLS secret into the NGINX container at `/etc/nginx/ssl`, and it also mounts an NGINX config file from a ConfigMap into `/etc/nginx/conf.d/default.conf`.

With that in place, whenever NGINX receives traffic on port 443, it decrypts it using the self-signed certificate, then forwards the request straight to the wisecow app on port 4499 inside the same pod.

So, at this stage, pods are running, NGINX is ready to accept HTTPS traffic, and wisecow is behind it. YAML doesn't actually expose NodePort 30443 yet. That part will only happen when apply the Service manifest, which tells Kubernetes how outside traffic should reach port 443 of NGINX in each pod.

```
apiVersion: apps/v1
kind: Deployment
```

```
metadata:
  name: wisecow-deployment
  labels:
    app: wisecow
spec:
  replicas: 3
  selector:
    matchLabels:
      app: wisecow
  template:
    metadata:
      labels:
        app: wisecow
    spec:
      volumes:
        - name: tls-cert
          secret:
            secretName: wisecow-tls-secret
        - name: nginx-config
          configMap:
            name: nginx-config
      containers:
        - name: wisecow
          image: shefeekar/wisecow:v1.0
          imagePullPolicy: Always
          ports:
            - containerPort: 4499
        - name: nginx
          image: nginx:latest
          ports:
            - containerPort: 443
          volumeMounts:
            - name: tls-cert
              mountPath: /etc/nginx/ssl
              readOnly: true
            - name: nginx-config
              mountPath: /etc/nginx/conf.d/default.conf
              subPath: default.conf
```

```
shefeek@mail:~/Downloads/wise-cow/k8$ kubectl apply -f wisecow-deployment.yml
deployment.apps/wisecow-deployment created
shefeek@mail:~/Downloads/wise-cow/k8$ kubectl get pods
NAME                                 READY   STATUS    RESTARTS   AGE
wisecow-deployment-dbd59c4dd-g7x2n   2/2     Running   0          15s
wisecow-deployment-dbd59c4dd-pbqg6   2/2     Running   0          15s
wisecow-deployment-dbd59c4dd-wczvb   2/2     Running   0          15s
shefeek@mail:~/Downloads/wise-cow/k8$
```

kubectl apply -f wisecow-deployment.yml

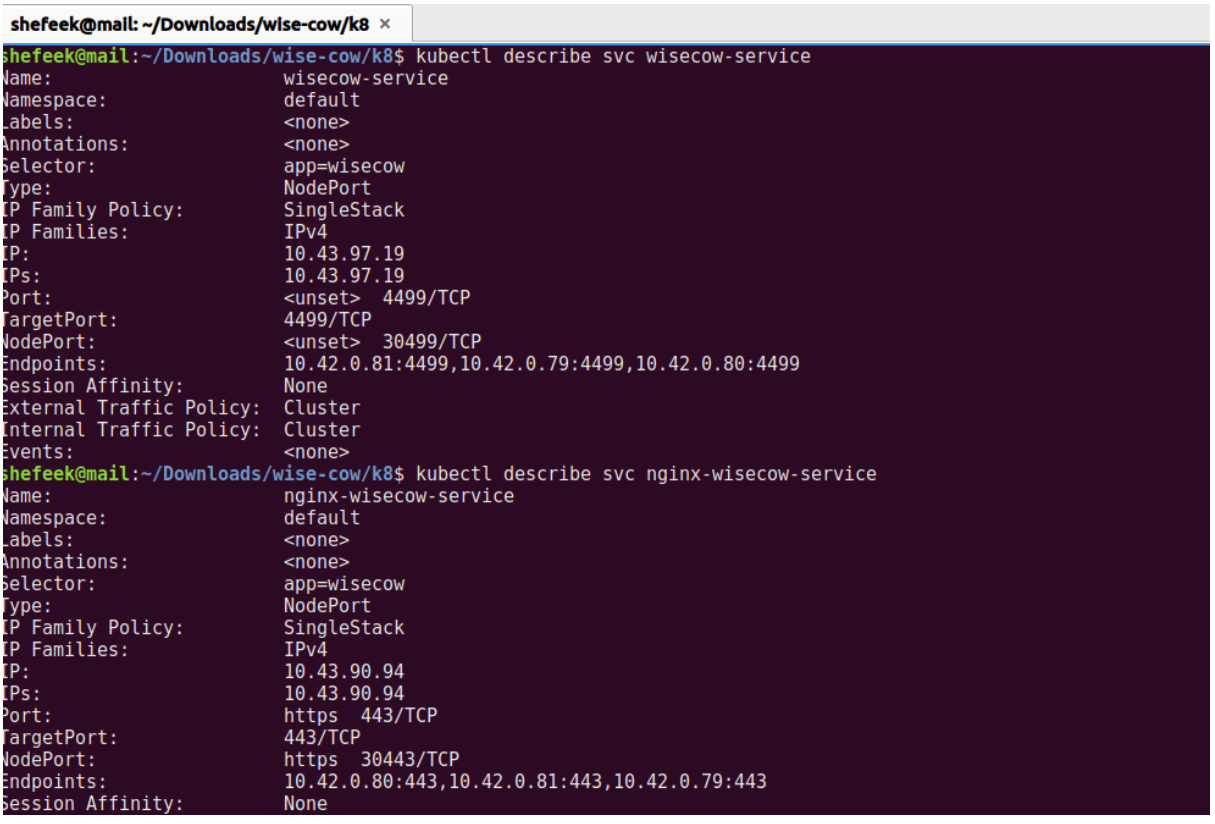
updated the service.yml

requests coming to NodeIP on port 30443. Kubernetes takes them and sends them straight to NGINX on port 443. NGINX decrypts the SSL, then passes the requests to the Wisecow app on port 4499. The app handles the request and the response travels back the same way to the client.
the following file is used to send traffic hit on the node port to listen the https trafic .
the service name is ***“nginx-wisecow-service”.***

```
apiVersion: v1
kind: Service
metadata:
  name: wisecow-service
spec:
  selector:
    app: wisecow
  type: NodePort
  ports:
    - protocol: TCP
      port: 4499
      targetPort: 4499
      nodePort: 30499
---
apiVersion: v1
kind: Service
metadata:
  name: nginx-wisecow-service
spec:
```

```
  selector:
    app: wisecow
  type: NodePort
  ports:
    - name: https
      port: 443
      targetPort: 443
      nodePort: 30443
      protocol: TCP
```

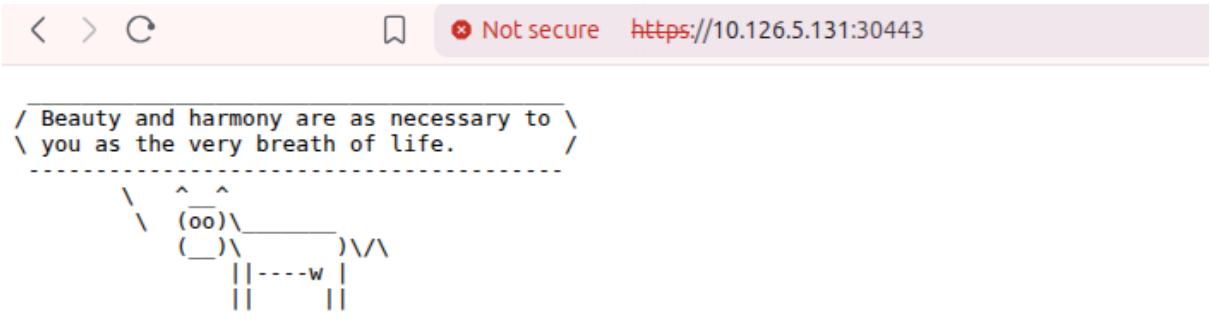
kubectl apply -f service.yml



checking ssl termination is working inside the pod

```
root@wisecow-deployment-dbd59c4dd-g7x2n:/app# curl http://10.42.0.79:4499

<pre> ______________________________________
/ If you sow your wild oats, hope for a \
\ crop failure.                         /
 --------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||</pre>
root@wisecow-deployment-dbd59c4dd-g7x2n:/app# curl https://127.0.0.1:443 --insecure

<pre> _________________________________
/ Questionable day. Ask somebody \
\ something.                     /
 ---------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||</pre>
root@wisecow-deployment-dbd59c4dd-g7x2n:/app#
```

successfully getting https://10.126.5.131:30443/

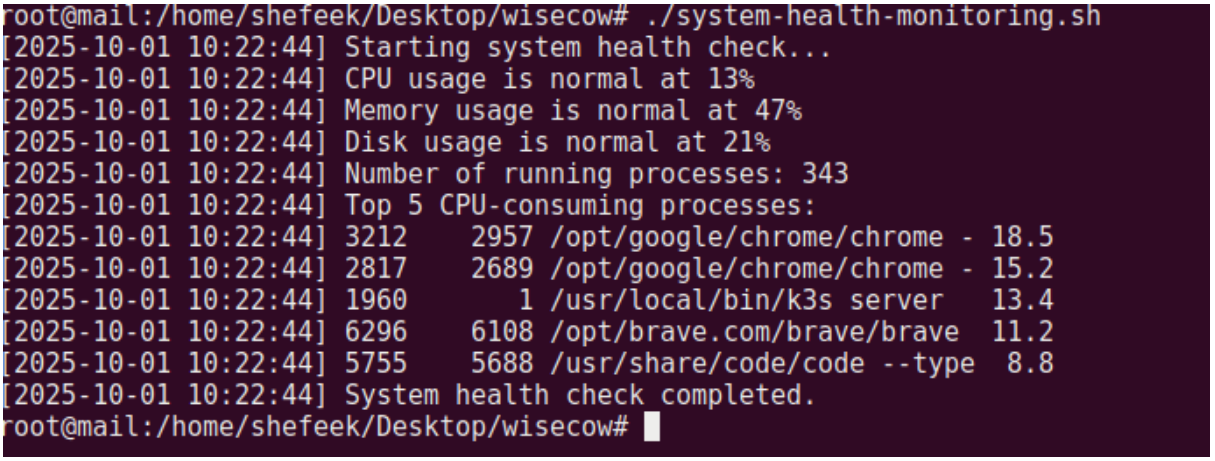


PART 2

1. System Health Monitoring Script:

```
#!/bin/bash

# Thresholds
CPU_THRESHOLD=80
MEM_THRESHOLD=80
DISK_THRESHOLD=80

# Log file
LOG_FILE="/var/log/system_health.log"

# Function to log messages
log_message() {
    local timestamp=$(date '+%Y-%m-%d %H:%M:%S')
    echo "[$timestamp] $1" | tee -a "$LOG_FILE"
}

# Function to check CPU usage
check_cpu() {
    local cpu_usage=$(top -bn1 | grep "Cpu(s)" | awk '{print $2 + $4}' | cut -
d. -f1)
    if [ "$cpu_usage" -gt "$CPU_THRESHOLD" ]; then
        log_message "ALERT: CPU usage is at ${cpu_usage}% (Threshold:
${CPU_THRESHOLD}%)"
    else
        log_message "CPU usage is normal at ${cpu_usage}%"
    fi
}

# Function to check memory usage
check_memory() {
    local mem_total=$(free -m | awk '/Mem:/ {print $2}')
    local mem_used=$(free -m | awk '/Mem:/ {print $3}')
    local mem_percent=$((100 * mem_used / mem_total))
    if [ "$mem_percent" -gt "$MEM_THRESHOLD" ]; then
        log_message "ALERT: Memory usage is at ${mem_percent}% (Thresh
old: ${MEM_THRESHOLD}%)"
    else
        log_message "Memory usage is normal at ${mem_percent}%"
```

```
    fi
}

# Function to check disk space
check_disk() {
    local disk_usage=$(df -h / | tail -1 | awk '{print $5}' | cut -d% -f1)
    if [ "$disk_usage" -gt "$DISK_THRESHOLD" ]; then
        log_message "ALERT: Disk usage is at ${disk_usage}% (Threshold:
${DISK_THRESHOLD}%)"
    else
        log_message "Disk usage is normal at ${disk_usage}%"
    fi
}

# Function to check running processes
check_processes() {
    local process_count=$(ps -e | wc -l)
    log_message "Number of running processes: ${process_count}"
    # List top 5 processes by CPU usage
    log_message "Top 5 CPU-consuming processes:"
    ps -eo pid,ppid,cmd,%cpu --sort=-%cpu | head -n 6 | tail -n 5 | while re
ad -r line; do
        log_message "$line"
    done
}

# Main execution
log_message "Starting system health check..."

check_cpu
check_memory
check_disk
check_processes

log_message "System health check completed."
```



Automated Backup Solution

```
#!/bin/bash

# Configuration
SOURCE_DIR="/path/to/source/directory"  # Directory to back up
REMOTE_SERVER="user@remote_server.com"  # Remote server address
REMOTE_DIR="/path/to/remote/backup/directory"  # Remote backup desti
nation
SSH_KEY="/path/to/ssh/key"  # Path to SSH private key
BACKUP_LOG="backup_log.txt"  # Log file for backup reports
TIMESTAMP=$(date +%Y%m%d_%H%M%S)
ARCHIVE_NAME="backup_${TIMESTAMP}.tar.gz"

# Function to log messages
log_message() {
    echo "$(date '+%Y-%m-%d %H:%M:%S') - $1" >> "$BACKUP_LOG"
}

# Check if source directory exists
if [ ! -d "$SOURCE_DIR" ]; then
    log_message "ERROR: Source directory $SOURCE_DIR does not exist"
    exit 1
fi

# Create backup archive
log_message "Starting backup process"
if tar -czf "$ARCHIVE_NAME" -C "$SOURCE_DIR" . 2>> "$BACKUP_LOG";
```

```
then
    log_message "Backup archive created: $ARCHIVE_NAME"
else
    log_message "ERROR: Failed to create backup archive"
    exit 1
fi

# Transfer backup to remote server
if scp -i "$SSH_KEY" "$ARCHIVE_NAME" "${REMOTE_SERVER}:${REMOTE
_DIR}/" 2>> "$BACKUP_LOG"; then
    log_message "Successfully transferred $ARCHIVE_NAME to $REMOTE_
SERVER:$REMOTE_DIR"
else
    log_message "ERROR: Failed to transfer backup"
    rm -f "$ARCHIVE_NAME"
    exit 1
fi

# Clean up local archive
if rm -f "$ARCHIVE_NAME"; then
    log_message "Cleaned up local archive: $ARCHIVE_NAME"
else
    log_message "ERROR: Failed to clean up local archive"
    exit 1
fi

log_message "Backup process completed successfully"
```