**cd k8s-master-app/:**

This command changes the current directory to k8s-master-app/. This is likely a directory where you are working on a Kubernetes master node application or setup.

**sudo usermod -aG docker $USER:**

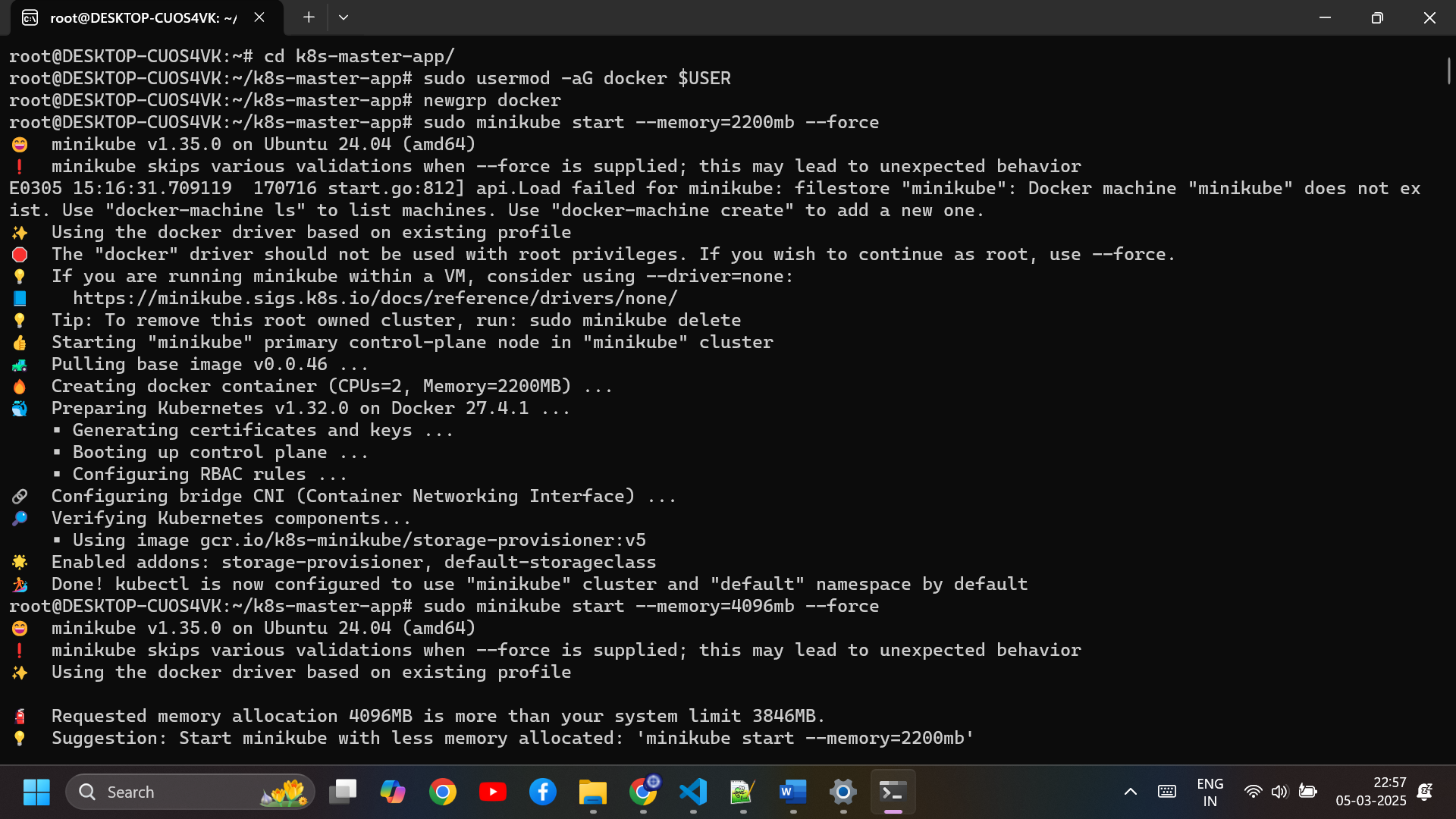
* sudo: This runs the command as a superuser (administrator) to gain necessary permissions for modifying user groups.
* usermod: This is a command used to modify a user's account settings.
* -aG docker: The -a flag stands for "append," meaning you are adding the user to an additional group (in this case, the docker group). The G flag specifies the group(s) to which the user will be added.
* $USER: This variable refers to the currently logged-in user.

In essence, this command adds the current user to the Docker group, giving the user permissions to run Docker commands without needing sudo.

**newgrp docker:**

This command changes the user's current group ID (GID) to the Docker group for the current session.

* **Why use newgrp docker**? When you add a user to a group (like Docker), the group membership will be effective only after a new session. Without running newgrp docker, you might need to log out and log back in for the changes to take effect. By using newgrp docker, you immediately apply the new group membership for the current terminal session, so you can run Docker commands without having to log out and back in.



**sudo minikube start --memory=2200mb –force**

**docker container prune -f**

* This command removes **all stopped containers**. The -f flag forces the action without asking for confirmation.

**docker image prune -a -f**

* Removes **unused Docker images** from your system.
* **-a**: This flag tells Docker to remove all unused images, not just dangling ones (those that are not tagged and are not being used by any container).
* **-f**: Again, forces the action without asking for confirmation.

**docker volume prune -f**

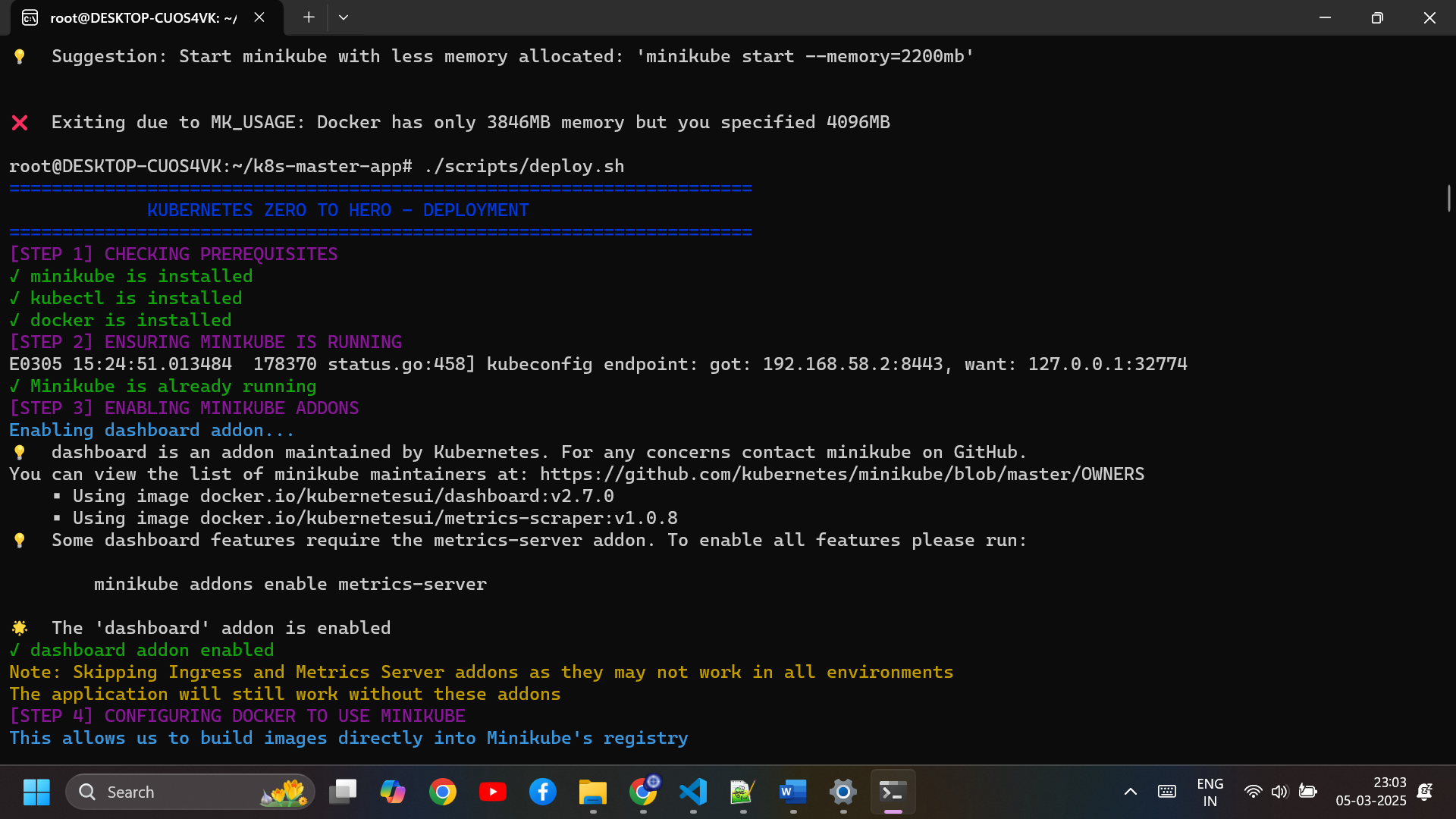
* Removes **unused Docker volumes**. Volumes are used to persist data in Docker containers.
* **-f**: Forces the pruning without confirmation.

**docker network prune -f**

* Removes **unused Docker networks**. These are the networks that aren't being used by any containers.
* **-f**: Again, forces the pruning without confirmation.

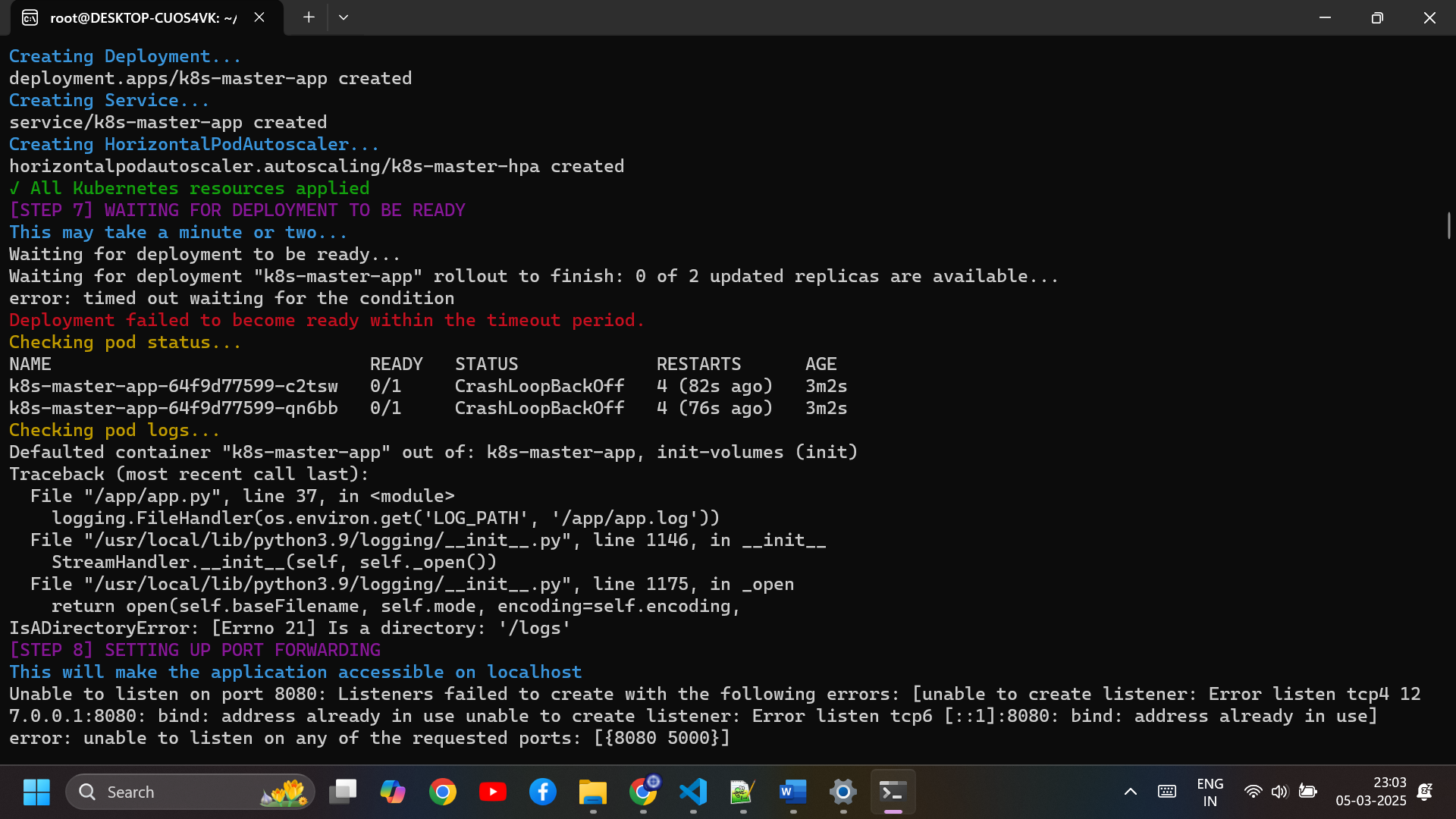
**docker system prune -a -f**

* A more aggressive cleanup command that removes **unused containers, networks, images, and volumes**.
* **-a**: Removes all unused images, not just the dangling ones.
* **-f**: Forces the action without confirmation.



**free -h**

**./scripts/deploy.sh**



**Steps to Fix Your Deployment**

Follow these steps to resolve your issues:

**1️.Update app.py to Fix Logging Issue**

Run the following command to edit the file:

nano ~/k8s-master-app/app/app.py

This is the existing part we need to change:  
# Set up logging to print to console and file

logging.basicConfig(

    level=logging.INFO,

    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',

    handlers=[

        logging.StreamHandler(sys.stdout),

        logging.FileHandler(os.environ.get('LOG\_PATH', '/app/app.log'))

    ]

)

Then **replace** the existing logging setup with this:

# Set up logging to print to console and file

log\_file = os.path.join(os.environ.get('LOG\_PATH', '/logs'), 'app.log')

logging.basicConfig(

level=logging.INFO,

format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',

handlers=[

logging.StreamHandler(),

logging.FileHandler(log\_file)

]

)

* **Ensure proper indentation**.
* **Save the file** (CTRL + X → Y → ENTER).

**2️. Change Directory to the App Folder**

cd ~/k8s-master-app/app

**3️. Rebuild the Docker Image**

docker build -t k8s-master-app:latest .

This ensures the latest app.py changes are included in the container.

**4️. Delete Existing Pods**

kubectl delete pod -l app=k8s-master -n k8s-demo

This will **force Kubernetes to pull the updated container**.

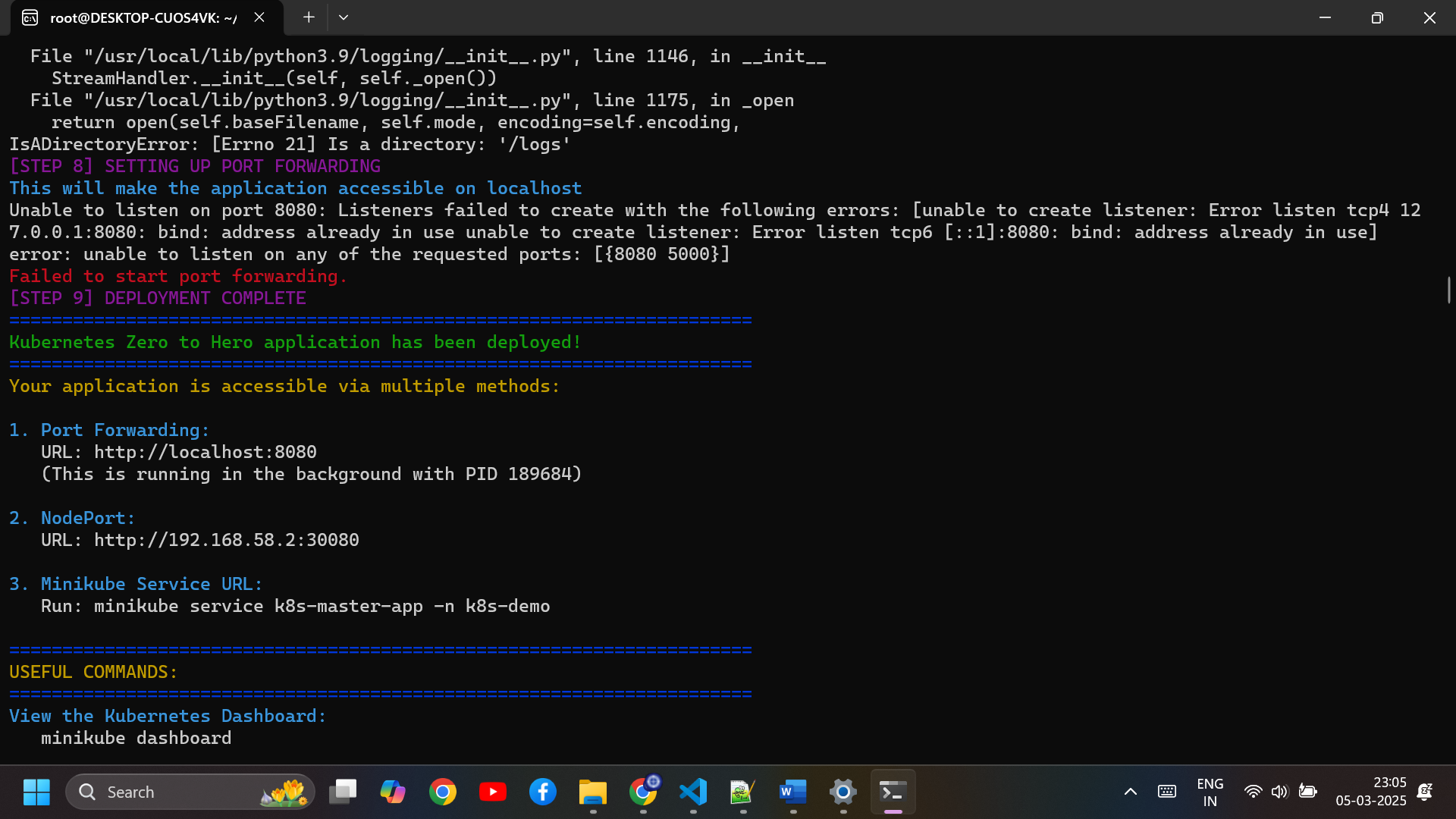
**5️. Move Back to the Project Root**

cd ..

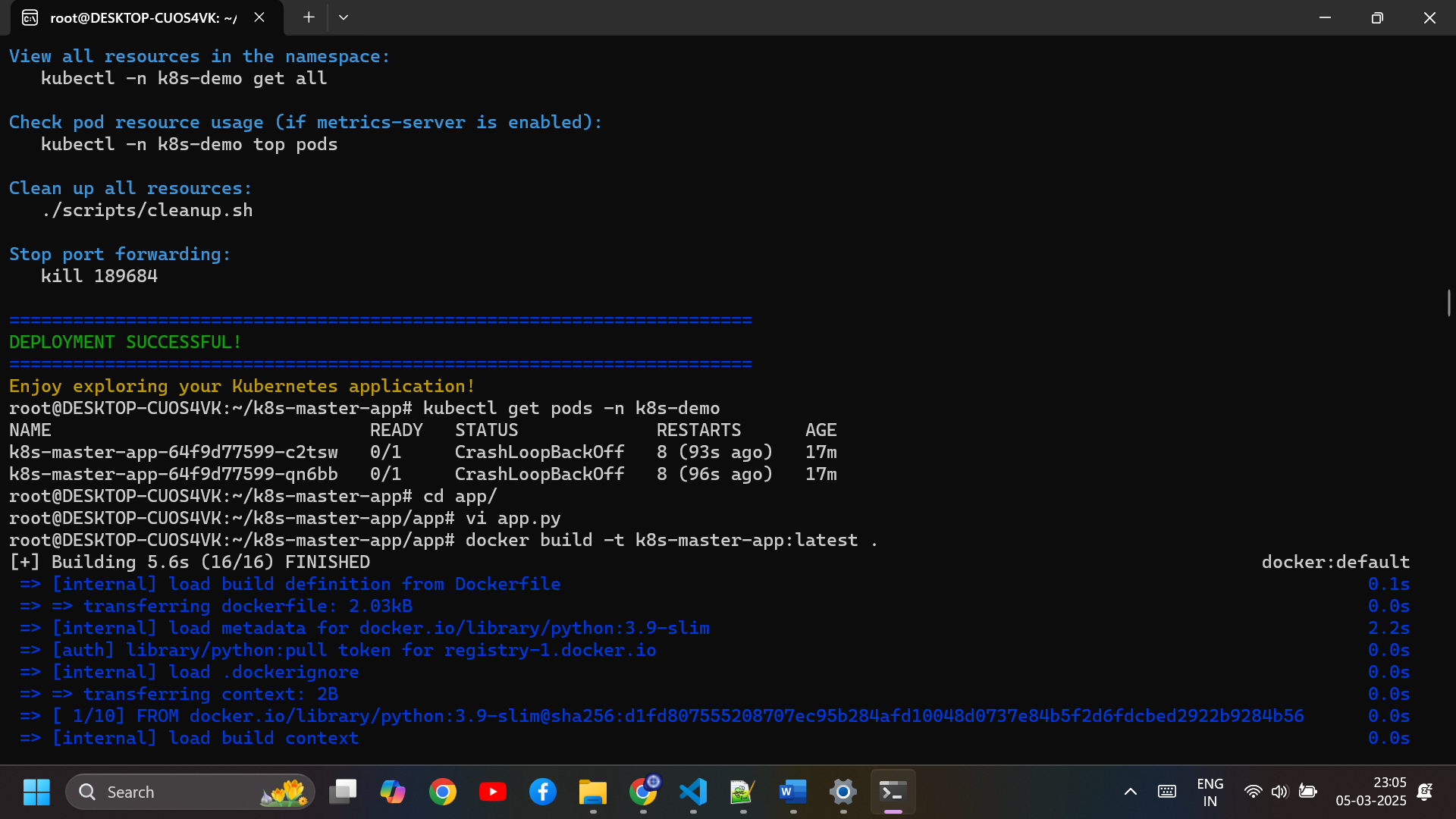
**6️. Redeploy the Application**

./scripts/deploy.sh

This will redeploy the updated image with the fixed logging setup.

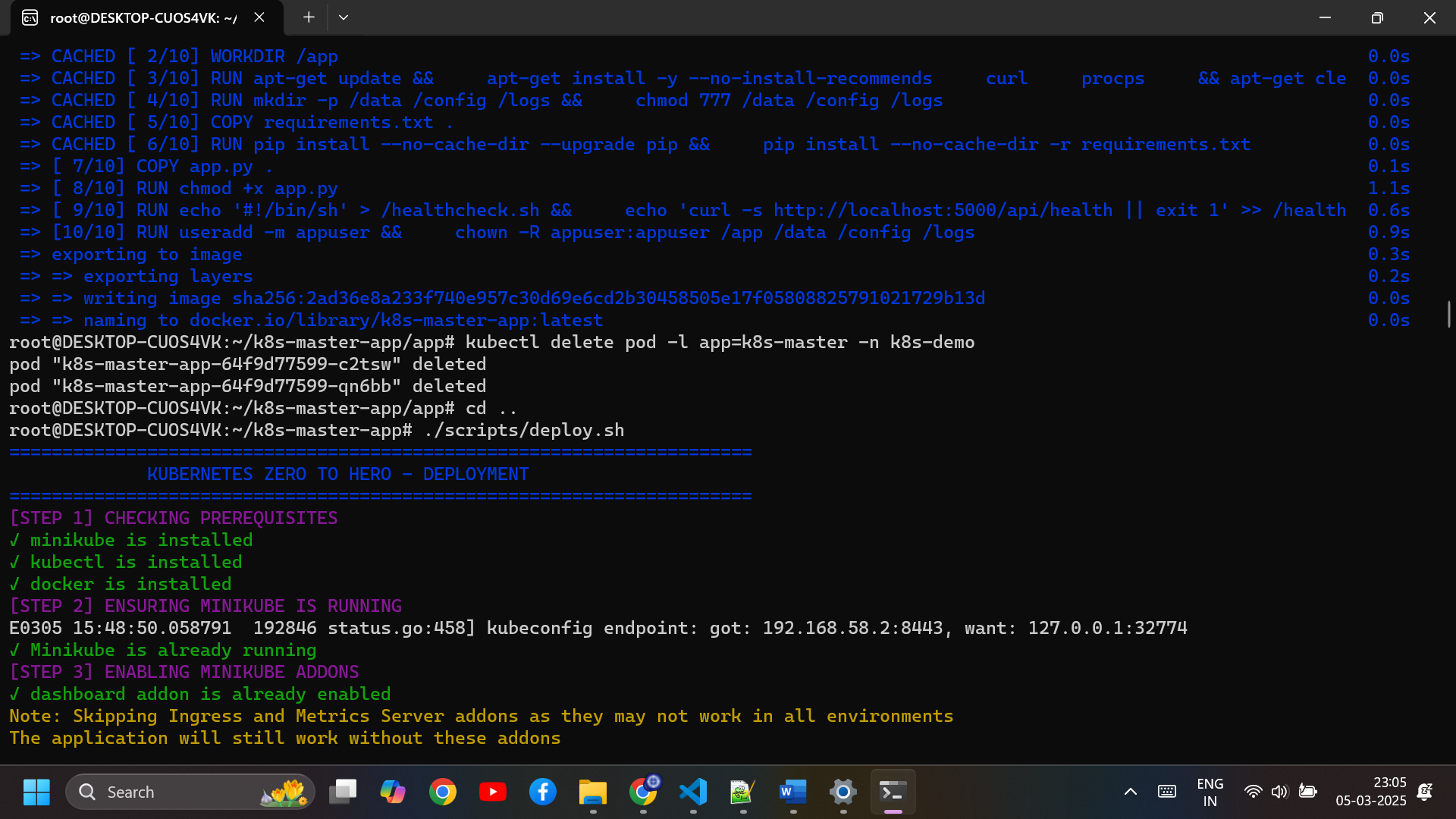


**kubectl get pods -n k8s-demo:**



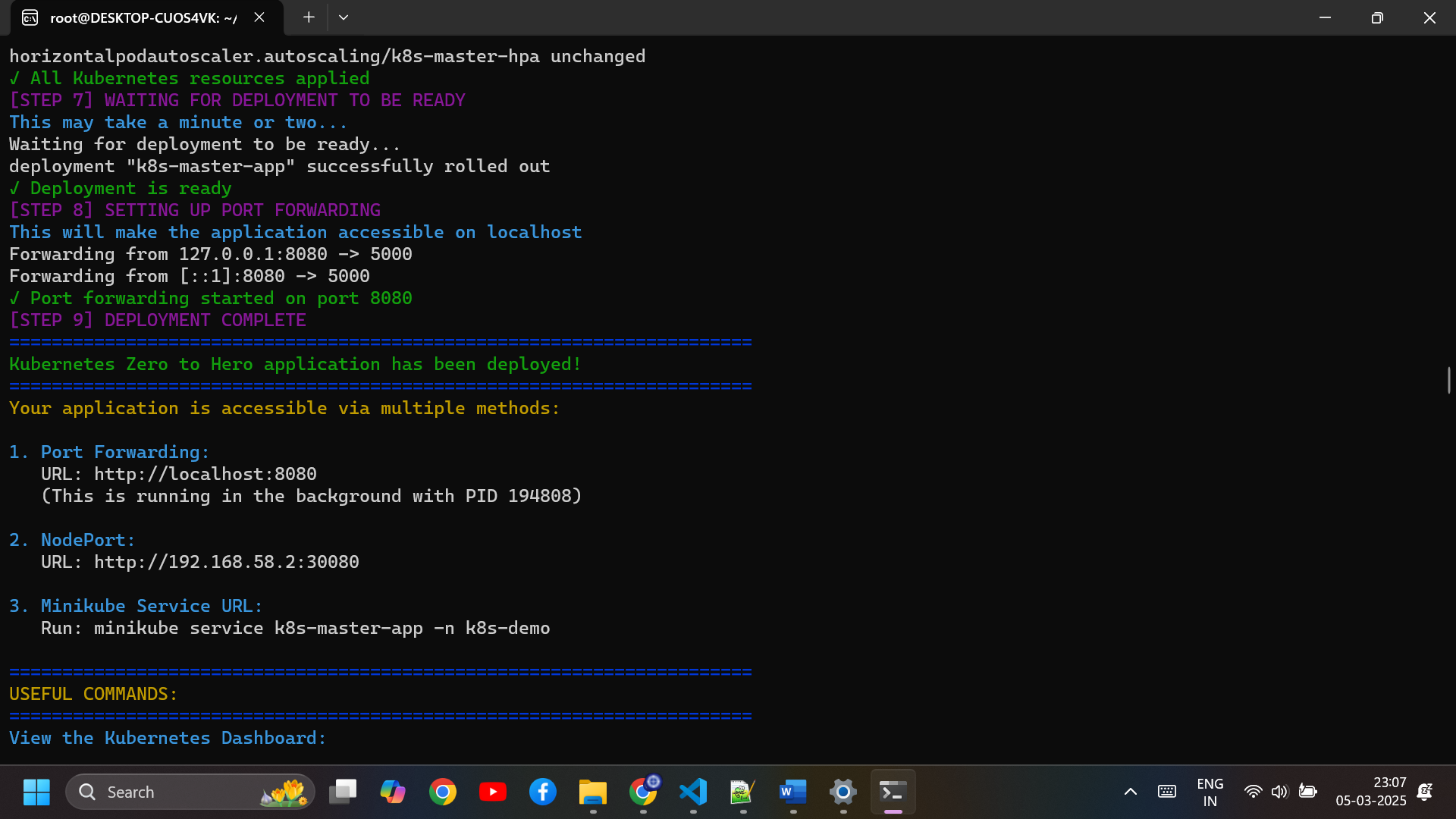
**kubectl delete pod -l app=k8s-master -n k8s-demo**

**sudo systemctl stop Jenkins**

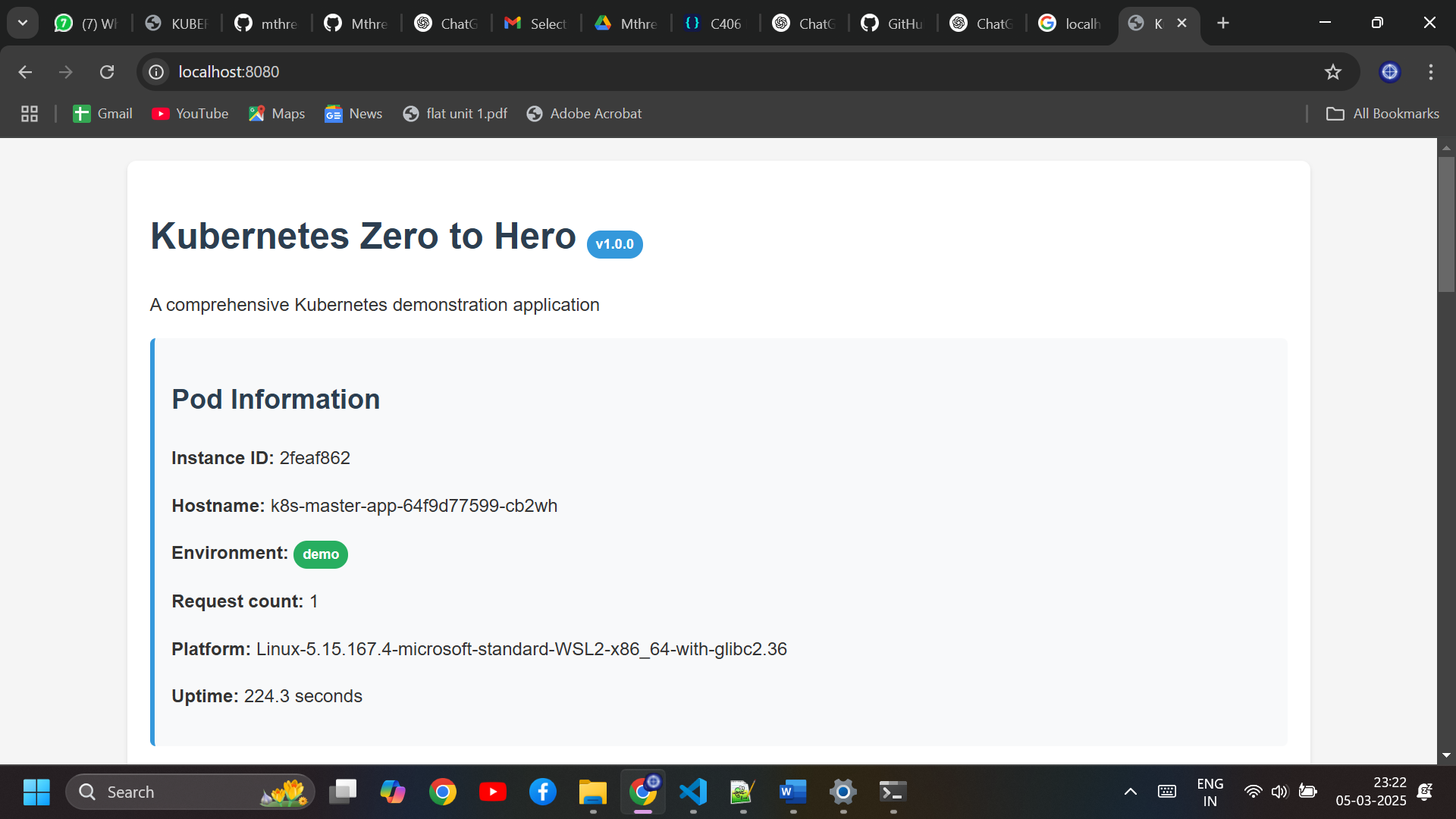


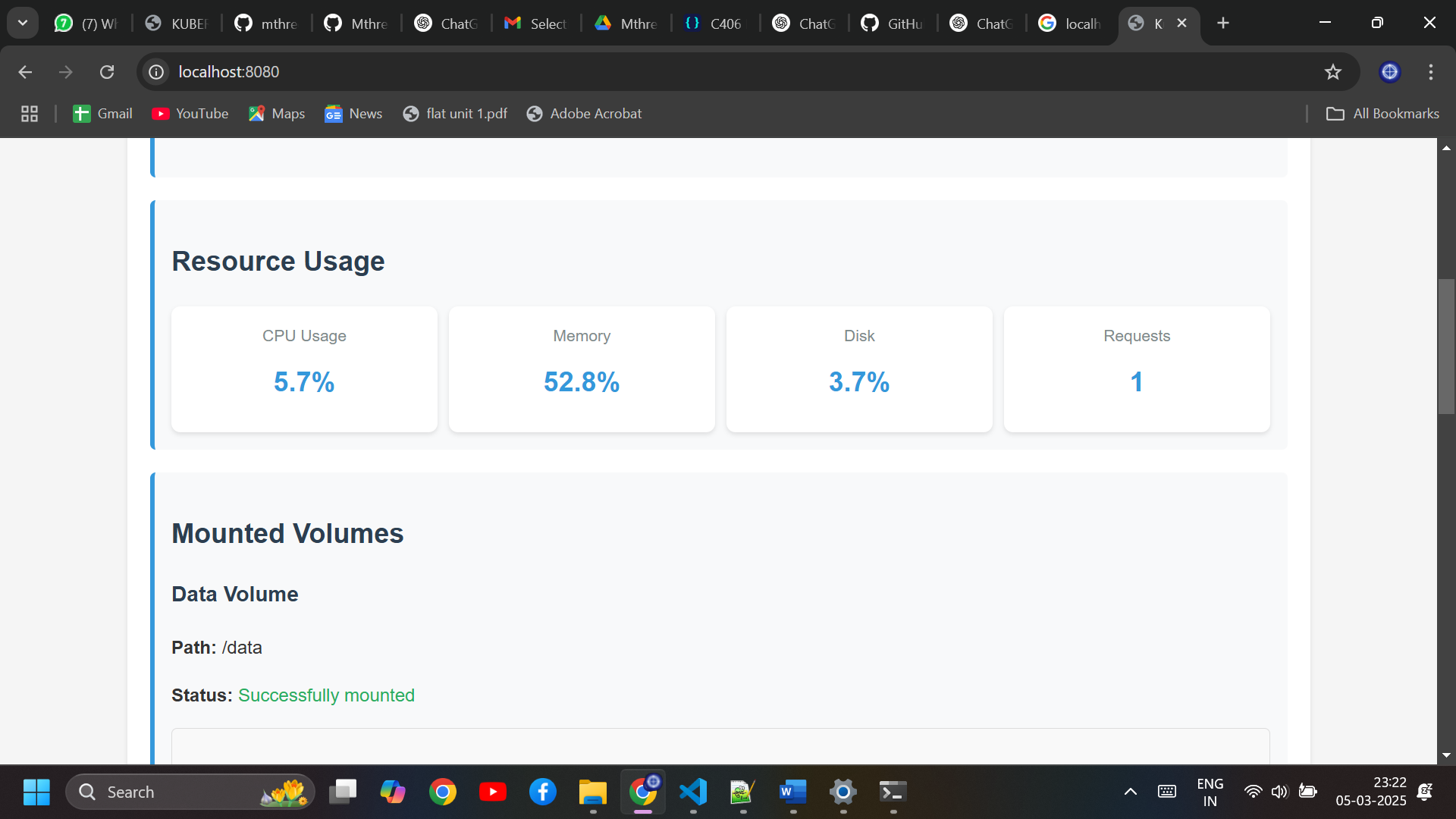
**cd k8s-master-app/**

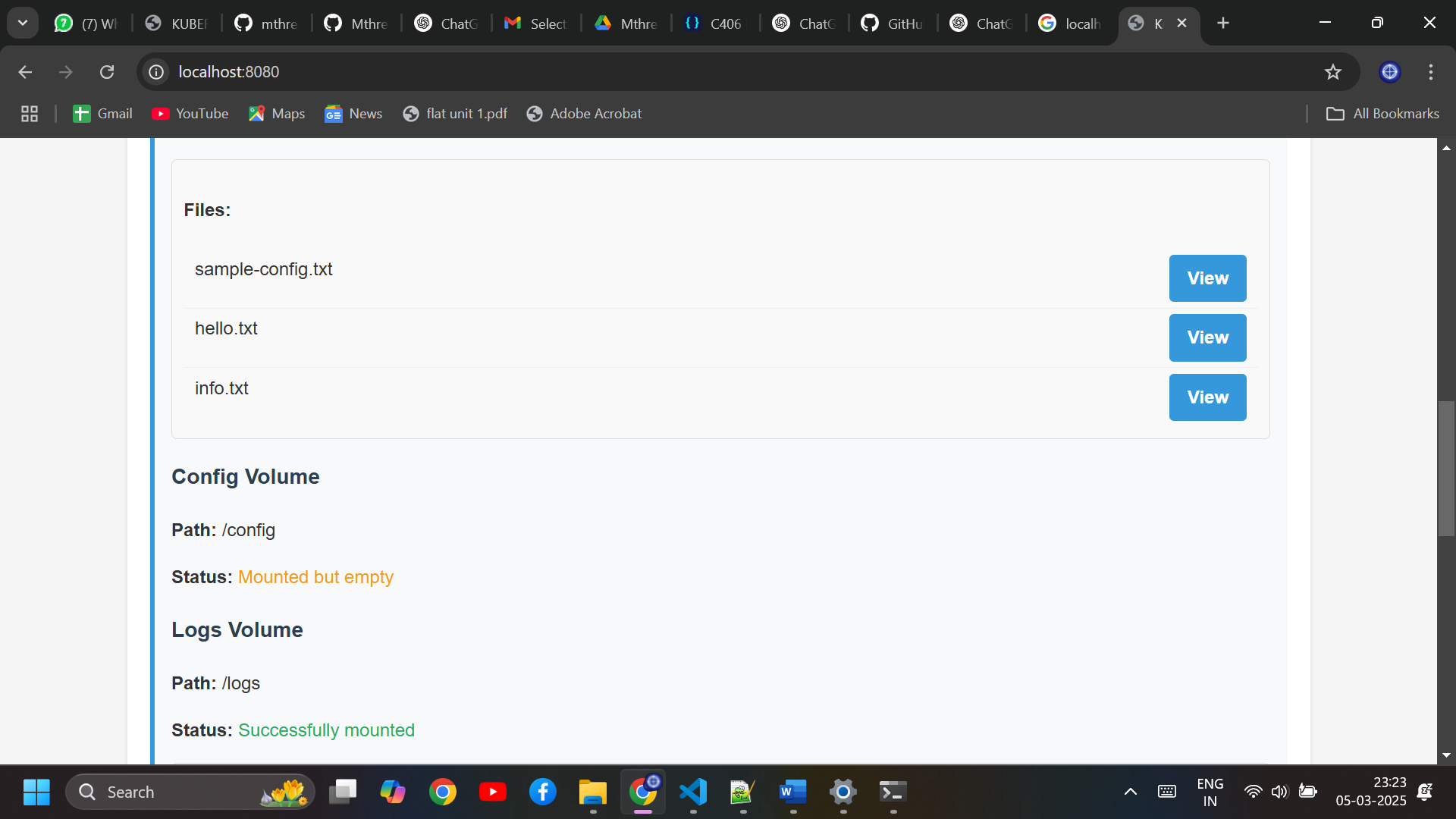
**./scripts/deploy.sh**

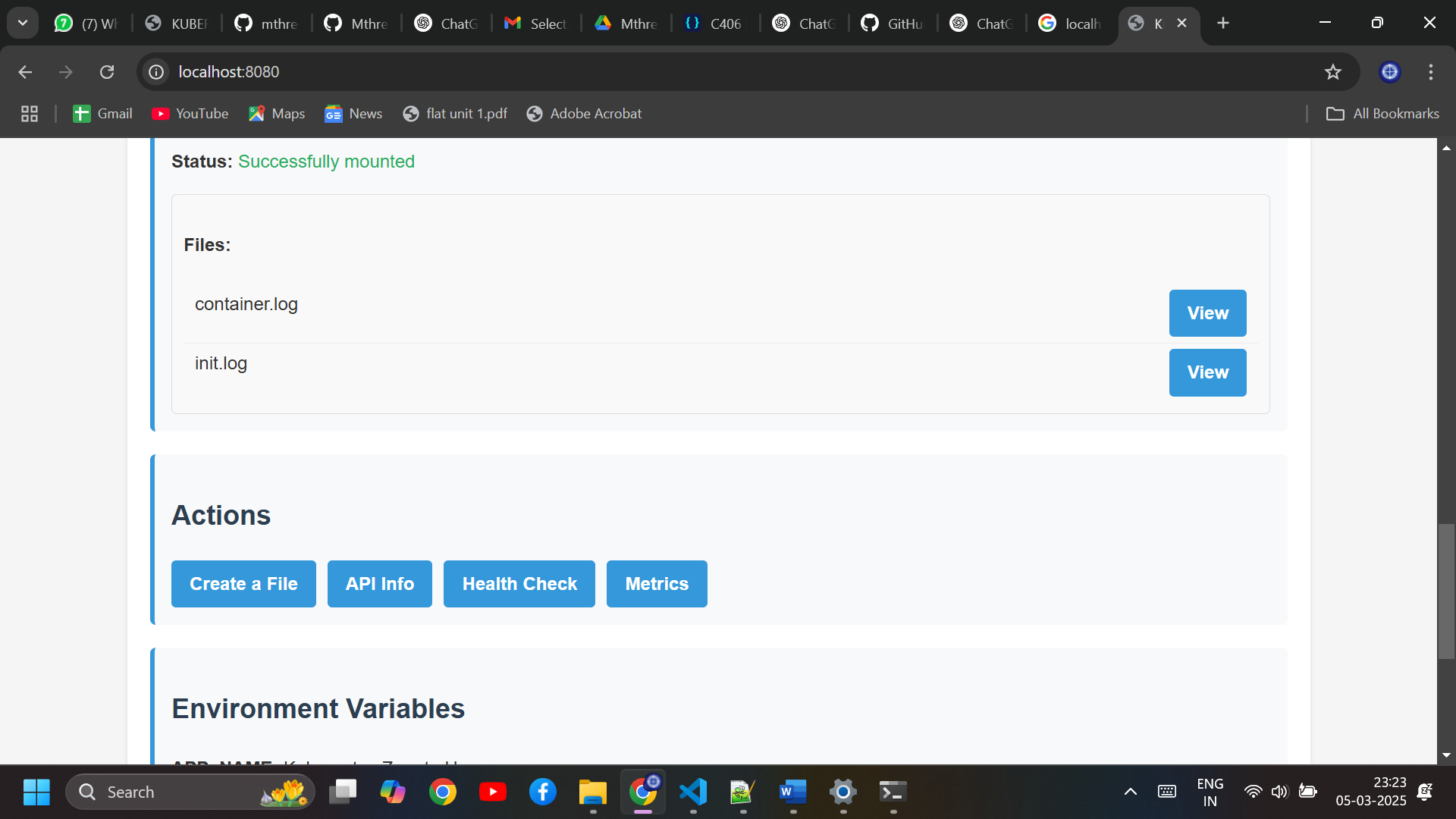


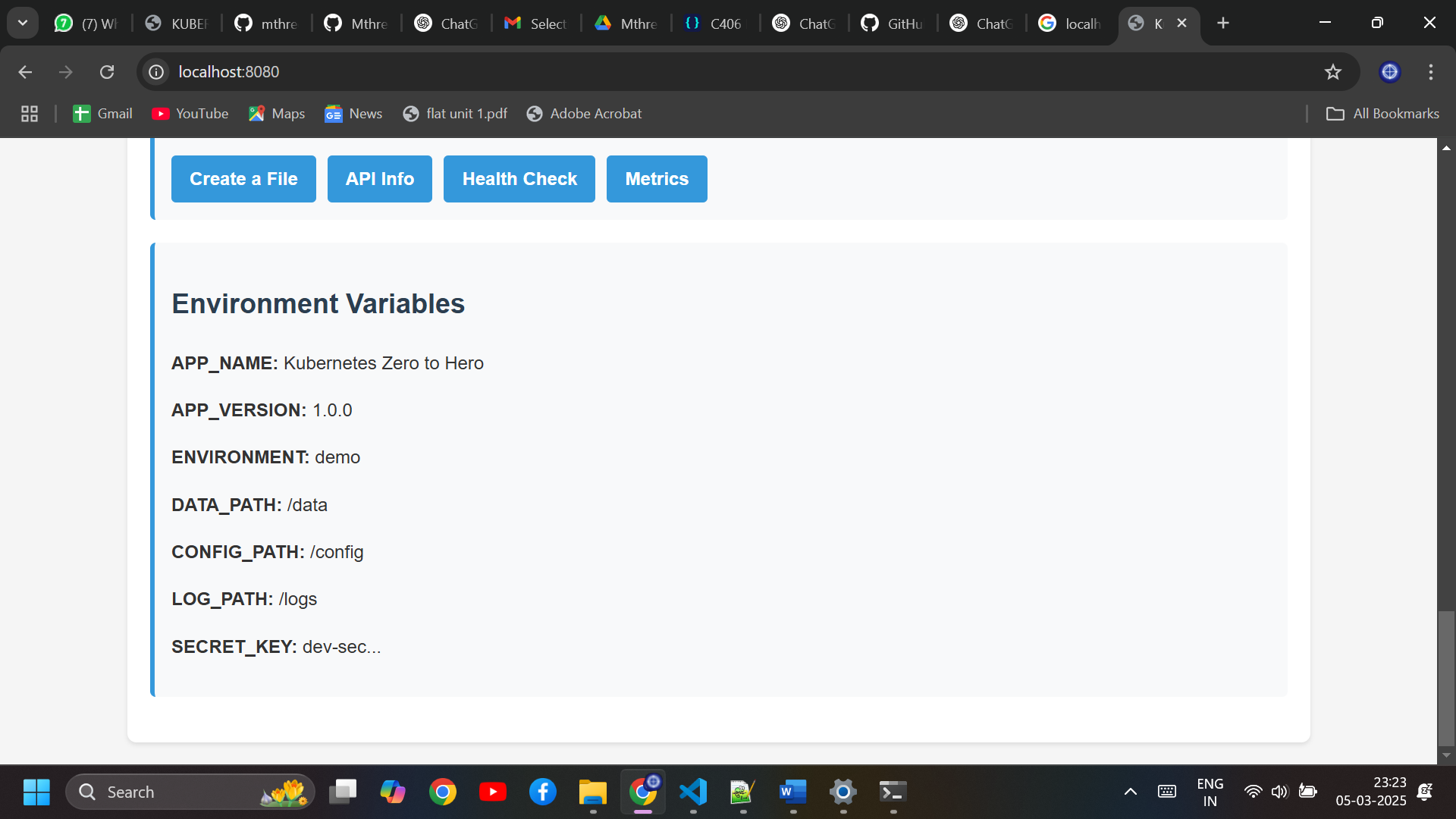
The Output in the port is:











**KUBERNETES COMMANDS:**

**1️.Namespace Management**

**View Current Namespace**

kubectl config view --minify | grep namespace

* Displays the **active namespace** in the current Kubernetes context.

**Create a Namespace**

kubectl apply -f k8s/base/namespace.yaml

* Creates a namespace **k8s-demo** from a YAML file.

**List All Namespaces**

kubectl get namespaces

* Displays all namespaces in the Kubernetes cluster.

**Delete a Namespace**

kubectl delete namespace k8s-demo

* Deletes the **k8s-demo** namespace and all resources within it.

**2️.Deployment Commands**

**Apply a Deployment**

kubectl apply -f k8s/base/deployment.yaml

* Deploys the application using the **Deployment YAML**.
* Ensures the desired number of **pod replicas** are running.

**Check Deployment Status**

kubectl -n k8s-demo rollout status deployment/k8s-master-app --timeout=180s

* Waits for the deployment to **fully roll out** (up to 180s timeout).

**List All Deployments**

kubectl get deployments -n k8s-demo

* Displays all **Deployments** running in the namespace.

**Describe a Deployment**

kubectl describe deployment k8s-master-app -n k8s-demo

* Shows detailed **configuration and status** of the deployment.

**Delete a Deployment**

kubectl delete -f k8s/base/deployment.yaml

* Deletes the deployment while **keeping other resources intact**.

**3️.Service Management**

**Apply a Service (NodePort)**

kubectl apply -f k8s/networking/service.yaml

* Creates a **Service** to expose the application.
* Uses **NodePort** to make it accessible externally.

**List Services**

kubectl get services -n k8s-demo

* Displays all **Services** running in the namespace.

**Describe a Service**

kubectl describe service k8s-master-app -n k8s-demo

* Provides details about the service, including **ports and endpoints**.

**Delete a Service**

kubectl delete -f k8s/networking/service.yaml

* Removes the **Service** but keeps the running pods.

**4️.ConfigMaps & Secrets**

**Apply a ConfigMap**

kubectl apply -f k8s/config/configmap.yaml

* Stores **non-sensitive** environment variables in Kubernetes.

**View ConfigMaps**

kubectl get configmaps -n k8s-demo

* Lists all ConfigMaps in the namespace.

**Apply a Secret**

kubectl apply -f k8s/config/secret.yaml

* Stores **sensitive** data (e.g., passwords, API keys) in Kubernetes.

**View Secrets**

kubectl get secrets -n k8s-demo

* Lists all Secrets in the namespace.

**5️.Pod Management**

**View Running Pods**

kubectl get pods -n k8s-demo

* Displays all **pods** running in the namespace.

**Describe a Pod**

kubectl describe pod <pod-name> -n k8s-demo

* Shows detailed information about a specific **pod**.

**View Logs from a Pod**

kubectl logs -n k8s-demo -l app=k8s-master

* Retrieves logs for **all pods** labeled app=k8s-master.

**Access a Running Pod (Bash Shell)**

kubectl exec -it <pod-name> -n k8s-demo -- /bin/bash

* Opens a **shell session** inside a running pod.

**Delete a Pod**

kubectl delete pod <pod-name> -n k8s-demo

* Deletes a specific pod (it will restart if managed by a Deployment).

**6️.Scaling & Auto-Scaling**

**Apply Horizontal Pod Autoscaler (HPA)**

kubectl apply -f k8s/monitoring/hpa.yaml

* Enables **auto-scaling** based on CPU and memory usage.

**View HPA Status**

kubectl get hpa -n k8s-demo

* Displays **scaling rules and current pod count**.

**Manually Scale the Deployment**

kubectl scale deployment k8s-master-app --replicas=3 -n k8s-demo

* Sets the number of running pods to **3**.

**7️.Port Forwarding & Exposing Services**

**Port Forwarding to Localhost**

kubectl -n k8s-demo port-forward svc/k8s-master-app 8080:80

* Maps **localhost:8080** to the service’s port **80**.

**Find Minikube IP**

minikube ip

* Retrieves the IP of the Minikube cluster.

**Access App via NodePort**

http://<minikube-ip>:30080

* Open the app in a **browser** using Minikube’s IP and **NodePort 30080**.

**Open the App via Minikube Service**

minikube service k8s-master-app -n k8s-demo

* Opens the **Kubernetes service** in the browser.

**8️. Cleaning Up Resources**

**Delete All Resources in Namespace**

kubectl delete namespace k8s-demo

* Completely removes the namespace and **all deployed resources**.

**Manually Delete Resources**

kubectl delete -f k8s/monitoring/hpa.yaml

kubectl delete -f k8s/networking/service.yaml

kubectl delete -f k8s/base/deployment.yaml

kubectl delete -f k8s/config/secret.yaml

kubectl delete -f k8s/config/configmap.yaml

kubectl delete -f k8s/base/namespace.yaml

* Deletes **each resource separately**.

**Remove Docker Image from Minikube**

docker rmi k8s-master-app:latest

* Deletes the **Docker image** to free up space.

**WHAT ACTUALLY WE HAVE DONE** :

**1. Environment Check (Prerequisites):**

* **Minikube** is installed and running.
* **kubectl** (the command-line tool for interacting with Kubernetes clusters) is installed.
* **Docker** is installed, which is necessary for building and running containerized applications.

**2. Ensuring Minikube is Running:**

* The script checks if Minikube is running properly. If it wasn't running, it would start it. It confirmed that Minikube is already running and accessible.

**3. Enabling Minikube Addons:**

* **Dashboard**: Minikube's Kubernetes dashboard addon is already enabled. The dashboard provides a web-based user interface to manage Kubernetes resources.
* **Ingress & Metrics Server**: These are optional addons that are skipped since they may not work in all environments, but the application will still function without them.

**4. Configuring Docker to Use Minikube:**

* **Docker is configured to use Minikube's registry**. This is important because it ensures that any Docker images built on your local machine are directly available in Minikube’s Docker environment (instead of needing to push them to an external Docker registry like Docker Hub).

**5. Building Docker Image:**

* The **Docker image** for the application (k8s-master-app:latest) was successfully built. This includes:
  + Pulling the base image python:3.9-slim.
  + Installing dependencies and copying application files.
  + Creating a custom Docker image (k8s-master-app) that contains your application.

**6. Deploying to Kubernetes:**

* **Namespace**: A Kubernetes namespace (k8s-demo) is created or checked if it already exists. Namespaces are used to logically separate resources in Kubernetes.
* **ConfigMaps & Secrets**:
  + **ConfigMap**: A ConfigMap is used to store configuration data (e.g., application settings, files) in Kubernetes.
  + **Secret**: Sensitive data (e.g., passwords, tokens) is stored securely in Secrets.
* **Deployment**: A **Kubernetes Deployment** is created to manage the lifecycle of your application pods. It ensures the application is running and can scale up or down as needed.
* **Service**: A **Service** is created to expose the application to other pods or external traffic, and ensures that network traffic is properly routed to the application.
* **HorizontalPodAutoscaler**: An **autoscaler** is set up to adjust the number of pods based on resource usage (like CPU or memory) to scale the application as needed.

**7. Waiting for Deployment to Be Ready:**

* The script waits for the **Deployment** to be fully rolled out and confirms that the deployment is ready.

**8. Setting Up Port Forwarding:**

* Port forwarding is set up so that your application becomes accessible locally via **localhost:8080**. This forwards traffic from your local machine’s port 8080 to port 5000 inside the Minikube VM, where the application is running.

**9. Deployment Complete:**

* The deployment is complete, and the application is now accessible in multiple ways:
  + **Port Forwarding**: Access the app via http://localhost:8080.
  + **NodePort**: The application can also be accessed externally on http://192.168.58.2:30080.
  + **Minikube Service URL**: You can use minikube service k8s-master-app -n k8s-demo to view the service from Minikube's context.

**Useful Commands:**

* View the **Kubernetes Dashboard**: minikube dashboard.
* View application **logs**: kubectl -n k8s-demo logs -l app=k8s-master.
* Access a **pod shell**: kubectl -n k8s-demo exec -it pod/k8s-master-app-64f9d77599-cb2wh -- /bin/bash.
* View all Kubernetes resources in the namespace: kubectl -n k8s-demo get all.
* Clean up all resources: ./scripts/cleanup.sh.
* Stop **port forwarding**: kill 194808 (stopping the process).

**Summary of What Was Done:**

* You have successfully deployed a **Kubernetes application** (k8s-master-app) onto a local Minikube cluster.
* The deployment involved building a **Docker image**, configuring Kubernetes resources (such as Deployment, Service, ConfigMaps, Secrets, and Autoscaling), and making the application accessible via different methods (port forwarding, NodePort, and Minikube service).
* The setup ensures that your application is ready for exploration and testing locally using **Kubernetes** with Minikube.

**Running the Script on AWS EC2**

I attempted to run the **fixed-k8s.sh** script on an AWS EC2 instance, but encountered several challenges:

**1. Minikube Installation Issues**

* Since **AWS EC2 is a remote environment**, Minikube was not the ideal choice for Kubernetes orchestration.
* Minikube is best suited for local testing, while AWS uses **Amazon EKS (Elastic Kubernetes Service)** for production Kubernetes clusters.
* Running Minikube required enabling the **Docker driver**, but EC2 instances had limited CPU cores (less than 2), leading to failure.

**Error Message:**

arduino

X Exiting due to RSRC\_INSUFFICIENT\_CORES: Docker has less than 2 CPUs available, but Kubernetes requires at least 2 to be available.

**Solution:**

* Instead of using Minikube, I needed to deploy the application to an AWS EKS cluster.

**2. Disk Space Limitations**

* AWS EC2 instances typically have a **default disk size of 8GB**, which quickly filled up when deploying Kubernetes.
* Minikube requires more storage for **container images, logs, and persistent volumes**.
* Expanding storage in AWS required:
  + Stopping the instance.
  + Modifying the root volume size in AWS Console.
  + Running the following command to resize the file system:

sudo growpart /dev/xvda 1 sudo resize2fs /dev/xvda1

**3. Kubernetes Cluster Connection Issues**

* The script relied on Minikube’s built-in Kubernetes API (192.168.49.2:8443).
* In AWS, we needed to use **Amazon EKS**, which required updating the kubectl context with:

aws eks update-kubeconfig --name my-cluster --region us-east-1

* The connection was failing due to **incorrect cluster configuration**.

**Error Message:**

pgsql

The connection to the server 192.168.49.2:8443 was refused - did you specify the right host or port?

**Solution:**

* Instead of using minikube, deploy the application to an existing EKS cluster.

**4. Port Forwarding Issues**

* The script tried to expose the application using **port forwarding (kubectl port-forward)**, which was not working on AWS EC2.
* Instead, I needed to expose the application via **LoadBalancer** or **Ingress Controller**.

**Alternative Approach:**

* Use a **LoadBalancer Service**:

apiVersion: v1 kind: Service metadata: name: k8s-master-app namespace: k8s-demo spec: type: LoadBalancer selector: app: k8s-master ports: - protocol: TCP port: 80 targetPort: 5000

**Lessons Learned**

* **Minikube is not suited for AWS EC2**; instead, we should use **EKS (Elastic Kubernetes Service)**.
* **Disk space issues on EC2 instances** can be resolved by expanding the root volume.
* **Kubernetes API connection issues** arise due to incorrect cluster contexts (kubectl should be configured for EKS).
* **Use LoadBalancer services** instead of port forwarding for better application accessibility in AWS.

**Next Steps**

* Modify the script to **deploy directly to an AWS EKS cluster** instead of Minikube.
* Update **storage management** to avoid No Space Left errors in EC2.
* Use **AWS Load Balancer** instead of Minikube NodePort services.