6. To design and deploy a multi-container application using Docker Compose by creating an application service and a dependent database/Redis service, configuring service dependencies, networks, and volumes, and verifying inter-container communication.

Prerequisites (Ubuntu)

Make sure Docker and Docker Compose are installed.

```bash
docker --version
docker compose version
```

If not installed (usually not needed in exam, but for completeness):

```bash
sudo apt update
sudo apt install docker.io docker-compose-plugin -y
sudo systemctl start docker
sudo systemctl enable docker
```

Step 1: Create a project directory

```bash
mkdir multi_container_app
cd multi_container_app
```

Step 2: Create application code (simple Python app)

This app connects to **Redis** and stores a key.

Create a file `app.py`:

```bash
nano app.py
```

Paste this code:

```python
import redis
import time

time.sleep(5)  # wait for redis to start

r = redis.Redis(host='redis', port=6379, decode_responses=True)
r.set('message', 'Hello from App Container')

print("Stored message in Redis:", r.get('message'))
```

Step 3: Create requirements file

```bash
nano requirements.txt
```

```text
redis
```

Step 4: Create Dockerfile for application service

bash

```
nano Dockerfile
```

dockerfile

```
FROM python:3.10-slim

WORKDIR /app

COPY requirements.txt .
RUN pip install -r requirements.txt

COPY app.py .

CMD ["python", "app.py"]
```

Step 5: Create Docker Compose file

This is the **core part** of the lab answer.

bash

```
nano docker-compose.yml
```

yaml

```yaml
version: "3.8"

services:
  app:
    build: .
    depends_on:
      - redis
    networks:
      - app_network

  redis:
    image: redis:latest
    volumes:
      - redis_data:/data
    networks:
      - app_network

networks:
  app_network:

volumes:
  redis_data:
```

Step 6: Run the multi-container application

```bash
docker compose up --build
```

Step 7: Verify inter-container communication

Expected output:

```text
Stored message in Redis: Hello from App Container
```

Step 8: Verify running containers (optional but good)

```bash
docker ps
```

Step 9: Verify volume creation

```bash
docker volume ls
```

You should see:

```text
redis_data
```

Step 10: Stop containers

```bash
docker compose down
```

7. To implement data ingestion, cleaning, and versioning using Data Version Control (DVC) by tracking raw and processed datasets, creating a reproducible pipeline with dvc.yaml, and validating reproducibility via dvc repro.

Step 1: Install prerequisites (Ubuntu)

```bash
sudo apt update
sudo apt install git python3 python3-pip -y
pip3 install dvc
```

Verify:

```bash
dvc --version
```

Step 2: Create project directory

```bash
mkdir dvc_lab
cd dvc_lab
```

Step 3: Initialize Git and DVC

```bash
git init
dvc init
```

Step 4: Create data directories

```bash
mkdir data
mkdir data/raw data/processed
```

Step 5: Create raw dataset (Data Ingestion)

Create a simple CSV file.

```bash
nano data/raw/data.csv
```

```
id,name,age
1,Alice,23
2,Bob,
3,Charlie,25
```

Step 6: Track raw data using DVC

```bash
dvc add data/raw/data.csv
git add data/raw/data.csv.dvc .gitignore
git commit -m "Add raw dataset with DVC tracking"
```

Step 7: Create data cleaning script

```bash
nano clean_data.py
```

```python
import pandas as pd

df = pd.read_csv("data/raw/data.csv")

# Remove rows with missing values
df = df.dropna()

df.to_csv("data/processed/cleaned_data.csv", index=False)
```

Install dependency:

```bash
pip3 install pandas
```

Step 8: Create DVC pipeline (dvc.yaml)

This is the **core requirement of the question.**

```bash
dvc run -n clean_data \
-d clean_data.py \
-d data/raw/data.csv \
-o data/processed/cleaned_data.csv \
python3 clean_data.py
```

Step 9: Track pipeline files

```bash
git add dvc.yaml dvc.lock clean_data.py
git commit -m "Add data cleaning pipeline"
```

Step 10: Track processed data

```bash
dvc add data/processed/cleaned_data.csv
git add data/processed/cleaned_data.csv.dvc
git commit -m "Track processed dataset"
```

Step 11: Verify reproducibility using `dvc repro`

```bash
dvc repro
```

Expected output:

```text
Stage 'clean_data' didn't change, skipping
```

Step 12: Validate versioning (optional but good)

Modify raw data:

```bash
nano data/raw/data.csv
```

Add a new row:

```
c

4,David,28
```

Run:

```bash
dvc repro
```

8. To perform experiment tracking using MLflow by training a machine learning model, logging metrics, parameters, and artifacts, and comparing multiple runs to identify the best-performing model.

Step 1: Install required packages (Ubuntu)

```bash
sudo apt update
sudo apt install python3 python3-pip -y
pip3 install mlflow scikit-learn pandas
```

Verify:

```bash
mlflow --version
```

Step 2: Create project directory

```bash
mkdir mlflow_lab
cd mlflow_lab
```

Step 3: Create training script

Create `train.py`:

```bash
nano train.py
```

python

Copy c

```python
import mlflow
import mlflow.sklearn
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

# Load dataset
X, y = load_iris(return_X_y=True)

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# Different hyperparameter values
C_value = 1.0

with mlflow.start_run():
    model = LogisticRegression(C=C_value, max_iter=200)
    model.fit(X_train, y_train)

    y_pred = model.predict(X_test)
    accuracy = accuracy_score(y_test, y_pred)

    # Log parameters
    mlflow.log_param("C", C_value)
    mlflow.log_param("model", "LogisticRegression")

    # Log metrics
    mlflow.log_metric("accuracy", accuracy)

    # Log model as artifact
    mlflow.sklearn.log_model(model, "model")

    print("Accuracy:", accuracy)
```

Step 4: Run first experiment

```bash
python3 train.py
```

Step 5: Run second experiment with different parameter

Edit `train.py`:

```bash
nano train.py
```

Change:

```python
C_value = 0.1
```

Run again:

```bash
python3 train.py
```

Now you have **multiple runs** logged in MLflow.

Step 6: Start MLflow Tracking UI

```bash
mlflow ui
```

Open browser:

```text
http://127.0.0.1:5000
```

9. To optimize and standardize model inference using ONNX by exporting a trained ML model toONNX format, running inference with ONNX Runtime, and benchmarking performance against the native scikit-learn model.

Step 1: Install required packages (Ubuntu)

```bash
sudo apt update
sudo apt install python3 python3-pip -y
pip3 install scikit-learn onnx onnxruntime skl2onnx numpy
```

Verify:

```bash
python3 -c "import onnx, onnxruntime; print('ONNX setup OK')"
```

Step 2: Create project directory

```bash
mkdir onnx_lab
cd onnx_lab
```

Step 3: Create Python script

```bash
nano onnx_inference.py
```

```python
import time
import numpy as np
import onnxruntime as ort

from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from skl2onnx import convert_sklearn
from skl2onnx.common.data_types import FloatTensorType

# Load dataset
X, y = load_iris(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# Train scikit-learn model
model = LogisticRegression(max_iter=200)
model.fit(X_train, y_train)

# ----------------------------
# Native scikit-learn inference
# ----------------------------
start = time.time()
sk_preds = model.predict(X_test)
sk_time = time.time() - start

print("Scikit-learn inference time:", sk_time)

# ----------------------------
# Export model to ONNX
# ----------------------------
initial_type = [("input", FloatTensorType([None, X.shape[1]]))]
onnx_model = convert_sklearn(model, initial_types=initial_type)

with open("model.onnx", "wb") as f:
    f.write(onnx_model.SerializeToString())

print("Model exported to ONNX format")

# ----------------------------
# ONNX Runtime inference
# ----------------------------
session = ort.InferenceSession("model.onnx", providers=["CPUExecutionProvider"])
input_name = session.get_inputs()[0].name

start = time.time()
onnx_preds = session.run(
    None, {input_name: X_test.astype(np.float32)}
)
onnx_time = time.time() - start

print("ONNX Runtime inference time:", onnx_time)
```

Step 4: Run the program

```bash
python3 onnx_inference.py
```

Step 5: Expected output (example)

```text
Scikit-learn inference time: 0.0012
Model exported to ONNX format
ONNX Runtime inference time: 0.0004
```

(Exact timings may vary, but ONNX is usually faster.)

Step 6: Verify ONNX model file

```bash
ls
```

You should see:

```text
model.onnx   onnx_inference.py
```

10. To serve a machine learning model using FastAPI by developing a /predict REST endpoint with input validation, writing test cases, containerizing the service, and verifying predictions through Postman or curl.

Step 1: Install required packages (Ubuntu)

```bash
sudo apt update
sudo apt install python3 python3-pip docker.io -y
pip3 install fastapi uvicorn scikit-learn pydantic pytest
```

Start Docker:

```bash
sudo systemctl start docker
sudo systemctl enable docker
```

Step 2: Create project directory

```bash
mkdir fastapi_ml
cd fastapi_ml
```

Step 3: Create ML model and API

Create `main.py`:

```bash
nano main.py
```

```python
from fastapi import FastAPI
from pydantic import BaseModel
from sklearn.datasets import load_iris
from sklearn.linear_model import LogisticRegression

# Load and train model
X, y = load_iris(return_X_y=True)
model = LogisticRegression(max_iter=200)
model.fit(X, y)

app = FastAPI()

# Input validation
class IrisInput(BaseModel):
    sepal_length: float
    sepal_width: float
    petal_length: float
    petal_width: float

@app.post("/predict")
def predict(data: IrisInput):
    features = [[
        data.sepal_length,
        data.sepal_width,
        data.petal_length,
        data.petal_width
    ]]
    prediction = model.predict(features)
    return {"prediction": int(prediction[0])}
```

Step 4: Run FastAPI locally

```bash
uvicorn main:app --reload
```

Test in browser:

```text
http://127.0.0.1:8000/docs
```

Step 6: Write test case

Create `test_api.py`:

```bash
nano test_api.py
```

```python
from fastapi.testclient import TestClient
from main import app

client = TestClient(app)

def test_predict():
    response = client.post("/predict", json={
        "sepal_length": 5.1,
        "sepal_width": 3.5,
        "petal_length": 1.4,
        "petal_width": 0.2
    })
    assert response.status_code == 200
    assert "prediction" in response.json()
```

Run test:

```bash
pytest
```

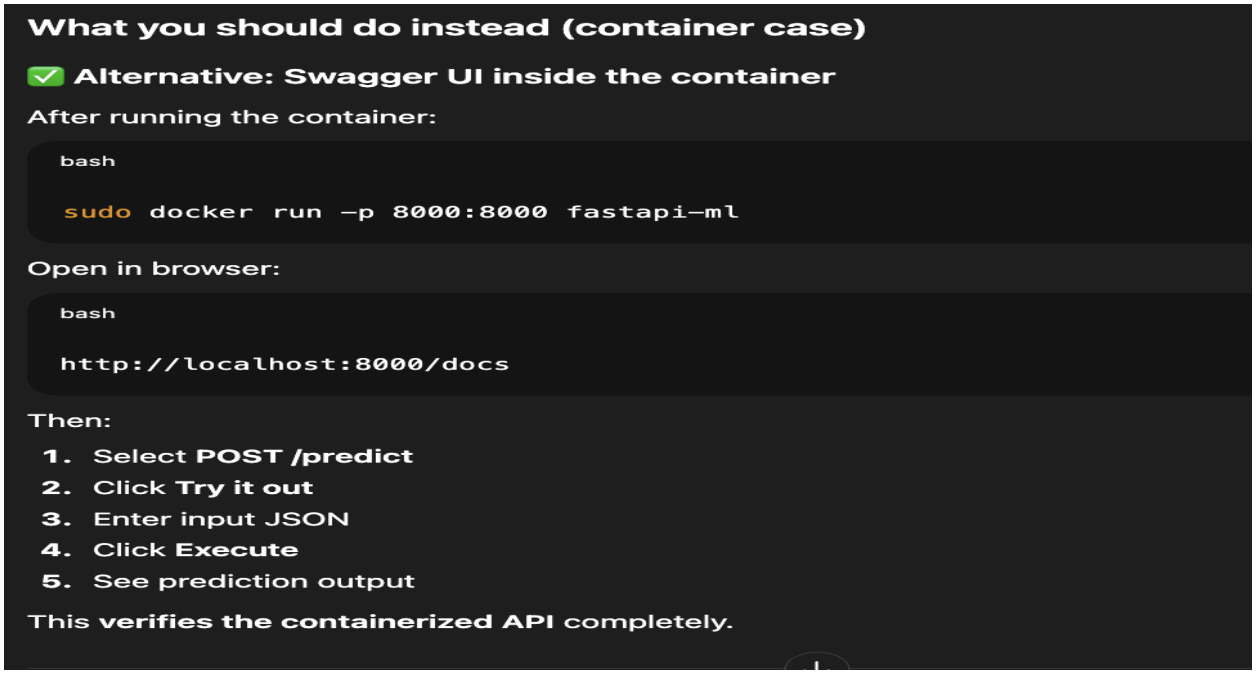
Step 7: Create Dockerfile (containerization)

```bash
nano Dockerfile
```

```dockerfile
FROM python:3.10-slim

WORKDIR /app

COPY main.py .
RUN pip install fastapi uvicorn scikit-learn pydantic

CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]
```

Step 8: Build and run Docker container

```bash
sudo docker build -t fastapi-ml .
sudo docker run -p 8000:8000 fastapi-ml
```



1. **Create a simple User Registration Form for an event with fields (Name, Email, Phone). Initialize a Git repository, commit the project files, and push them to GitHub using appropriate Git commands. Create a new branch named update-form, modify the registration form by adding a new field (Department), merge the branch into main, and push the changes to GitHub. Show the use of: git branch, git checkout, git merge, git push.**

mkdir student-portal
cd student-portal
nano index.html

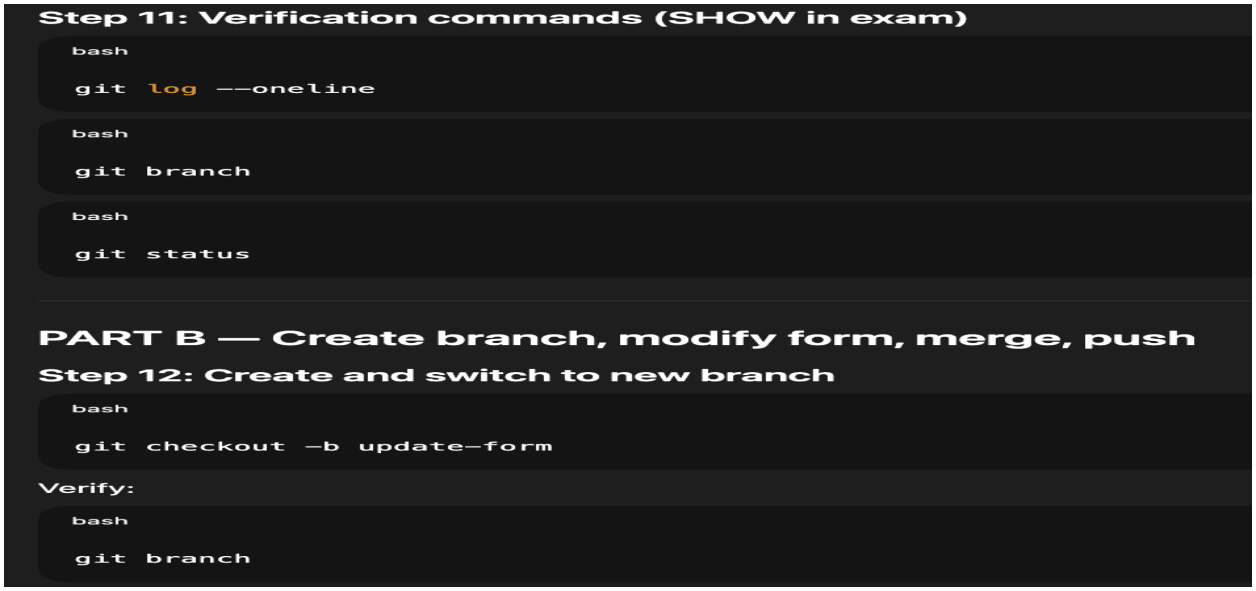
<!doctype html>
<html lang="en">
<head>
<meta charset="utf-8"/>
<meta name="viewport" content="width=device-width,initial-scale=1"/>
<title>Event Registration</title>
</head>
<body>
<h1>Event Registration</h1>
<form id="regForm" action="#" method="post">
<label>Name: <input type="text" name="name" required></label>
<label>Email: <input type="email" name="email" required></label>
<label>Phone: <input type="tel" name="phone" required></label>
<button type="submit">Register</button>
</form>
</body>

</html>

git init
git config --global user.name "Your Name"
git config --global user.email "you@example.com"

git add index.html
git commit -m "Initial commit: add registration form"

git branch -M main
git push -u origin main



Step 13: Modify `index.html` (Add Department field)

```bash
nano index.html
```

Add this line AFTER Phone:

```html
<label>Department: <input type="text" name="department" required></la
```

Save and exit.

Step 14: Stage and commit changes

```bash
git add index.html
git commit -m "Add Department field to registration form"
```

Step 15: Switch back to main branch

```bash
git checkout main
```

git merge update-form
git push origin main

git branch -d update-form
git show HEAD:index.html

2. **Jenkins**

A. Repo (on your machine)

If you don't already have the repo locally, clone it:

```bash
git clone https://github.com/YOUR_USERNAME/student-portal.git
cd student-portal
```

B. Create the Jenkins Freestyle job (UI steps — exact)

1. Open `http://<JENKINS_HOST>/` → **New Item**.
2. Enter name: `student-portal-site` → select **Freestyle project** → **OK**.
3. **Source Code Management** → choose **Git** → Repository URL:
   `https://github.com/YOUR_USERNAME/student-portal.git`
   - If private repo: add credentials (username/password or SSH key) using **Add**.
4. **Build Triggers** → check **GitHub hook trigger for GITScm polling** (or "Build when a change is pushed to GitHub").
5. **Build** → **Add build step** → **Execute shell**. Put these exact lines:

```bash
echo "Building student-portal"
ls -la
echo "Contents of index.html:"
sed -n '1,80p' index.html
```

6. Save the job.

C. Configure GitHub webhook (UI steps — exact)

1. On GitHub repo → **Settings → Webhooks → Add webhook**.
2. Payload URL:
   `http://<JENKINS_HOST>/github-webhook/`
3. Content type: `application/json`
4. Which events: **Just the push event → Add webhook**.

(If Jenkins is not publicly reachable, use `ngrok` or use GitHub Enterprise inside lab — your instructor will usually provide connectivity.)

D. Make a tiny change and push (exact commands)

Run these inside the cloned repo:

bash Copy code

```
# create small branch, change index.html heading, and push
git checkout -b tiny-update
sed -i 's/Event Registration/Event Registration - Updated/' index.html
git add index.html
git commit -m "Tiny update: change heading for CI test"
git push -u origin tiny-update

# merge to main (or create PR and merge); here we merge locally and push
git checkout main
git merge tiny-update
git push origin main
```

Result: GitHub receives `push` event → webhook posts to Jenkins → Jenkins starts `student-portal-site` job.

E. Verify Jenkins build triggered (commands you can run)

1. **(Quick) check job console via CLI** — if Jenkins has no auth or you have a token:

bash Copy code

```
# Get console text of last build
curl "http://<JENKINS_HOST>/job/student-portal-site/lastBuild/consoleText"
```

You should see:

php-template Copy code

```
Building student-portal
total 8
-rw-r--r-- ... index.html
Contents of index.html:
<!doctype html>
...
<h1>Event Registration - Updated</h1>
...
Finished: SUCCESS
```

3. **Create a simple web application (HTML or Python Flask). Write a Dockerfile, build a Docker image, and run the container so the application is accessible on a browser using port mapping. Using Docker commands, perform the following operations: List images and containers Stop a running container Remove a container and image Demonstrate commands like: docker ps, docker stop, docker rm, docker rmi.**

```bash
mkdir student-portal-docker
cd student-portal-docker
```

Step 2: Create `app.py`

```bash
nano app.py
```

Paste **EXACTLY THIS**:

```python
from flask import Flask, render_template_string

app = Flask(__name__)

HTML = """
<!doctype html>
<html><body>
<h1>Student Portal</h1>
<p>Registration form:</p>
<form>
Name: <input name="name"><br>
Email: <input name="email"><br>
Phone: <input name="phone"><br>
Department: <input name="department"><br>
</form>
</body></html>
"""

@app.route("/")
def home():
    return render_template_string(HTML)

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5000)
```

Step 3: Create `requirements.txt`

```bash
nano requirements.txt
```

```text
Flask==2.3.2
```

Step 4: Create `Dockerfile`

```bash
nano Dockerfile
```

Paste **EXACTLY THIS**:

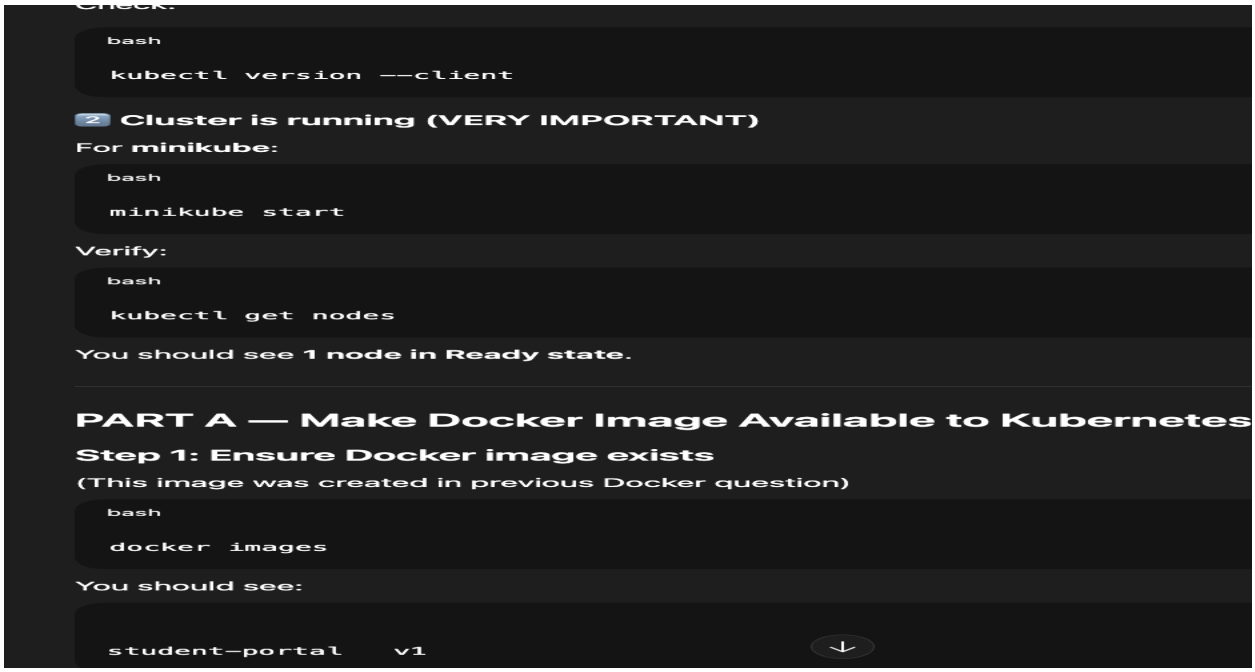
```dockerfile
FROM python:3.11-slim

WORKDIR /app

COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY app.py .

EXPOSE 5000

CMD ["python", "app.py"]
```

Build docker image
docker build -t student-portal:v1 .
docker run -d --name student-portal -p 5000:5000 student-portal:v1

Open http://localhost:5000

Listing running containers
docker ps
List all containers
docker ps -a
List docker images
docker images
Stop running container
docker stop student-portal
Remove container
docker rm student-portal
Remove docker image
docker rmi student-portal:v1

4. **Deploy the previously created Docker image on Kubernetes using a Deployment YAML file. Verify that the pod is running using kubectl commands. Expose the application using a NodePort Service, access it using the node's IP and port, and scale the deployment to 3 replicas using kubectl scale.**



Step 2: Load image into minikube
minikube image load student-portal:v1

nano deployment.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: student-portal-deploy
spec:
  replicas: 1
  selector:
    matchLabels:
      app: student-portal
  template:
    metadata:
      labels:
        app: student-portal
    spec:
      containers:
      - name: student-portal
        image: student-portal:v1
        ports:
        - containerPort: 5000
```

Step 4: Apply deployment

```bash
kubectl apply -f deployment.yaml
```

Expected:

```bash
deployment.apps/student-portal-deploy created
```

Step 5: Verify deployment

```bash
kubectl get deployments
```

Expected:

```
student-portal-deploy   1/1   1   1   20s
```

Step 6: Verify pod is running

```bash
kubectl get pods -l app=student-portal
```

Expected:

```sql
student-portal-deploy-xxxxxx   1/1   Running
```

Step 7: View pod logs (proof of running app)

```bash
kubectl logs <pod-name>
```

You should see Flask startup logs.

PART C — Expose Application Using NodePort

Step 8: Create NodePort Service YAML

```bash
nano service-nodeport.yaml
```

Paste **EXACTLY THIS**:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: student-portal-nodeport
spec:
  type: NodePort
  selector:
    app: student-portal
  ports:
  - protocol: TCP
    port: 5000
    targetPort: 5000
    nodePort: 30080
```

Save and exit.

Step 9: Apply service

```bash
kubectl apply -f service-nodeport.yaml
```

Step 10: Verify service

```bash
kubectl get svc student-portal-nodeport
```

Expected:

```css
student-portal-nodeport   NodePort   10.96.x.x   <none>   5000:30080/TCP
```

PART D — Access Application

Step 11: Get Node IP

If using minikube

```bash
minikube ip
```

Example:

```
192.168.49.2
```

Step 12: Access application

Open browser OR use curl: `curl http://<NODE_IP>:30080`

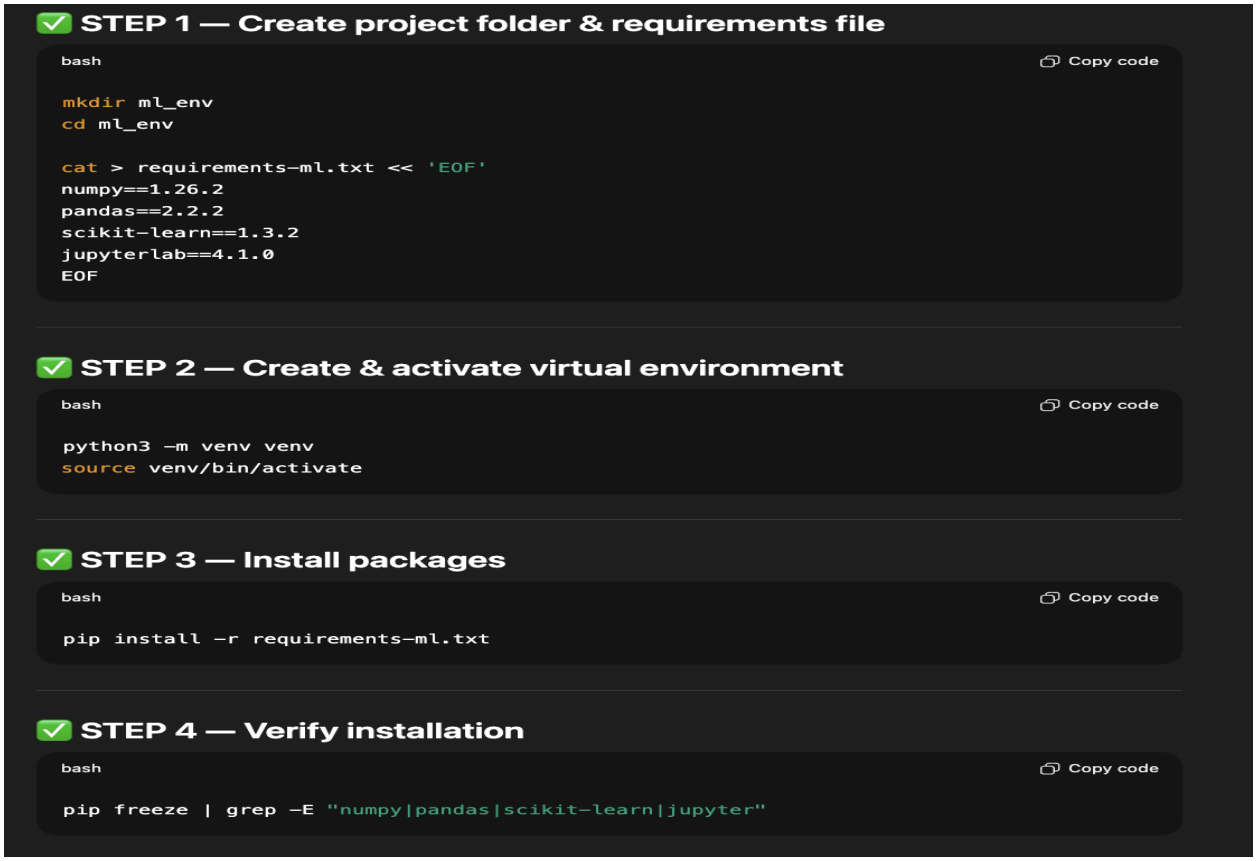
Scale deployment

```
kubectl scale deployment student-portal-deploy --replicas=3
```

Verify scaled pods

```
kubectl get pods -l app=student-portal
```

5. **Set up a simple ML project environment by creating a requirements.txt file, installing packages, and verifying environment setup. Document the steps in a Jupyter Notebook and commit it to the Git repository.**



✅ STEP 5 — Open Jupyter Notebook

```bash
jupyter lab
```

This will open a browser window.

Inside Jupyter Lab UI:

1. Click **File → New → Notebook**
2. Select **Python 3 (ipykernel)**
3. Create the two cells exactly as given:
   - Markdown cell
   - Code cell
4. Save as **ml_setup.ipynb**

Then **run** the code cell → output should match requirements.

✅ STEP 6 — Initialize Git

(inside the same `ml_env` folder)

```bash
git init
git add requirements-ml.txt ml_setup.ipynb
git commit -m "Add ML environment requirements and verification notebook"
```

✅ STEP 7 — Connect to GitHub & push

(Replace `<your_repo_url>` with your actual GitHub repo link)

```bash
git remote add origin <your_repo_url>
git branch -M main
git push -u origin main
```