INDEX

S. No.	Experiment	Date	Sign
1	Introduction to DevOps and MLOps: Understand the basic principles, differences, and lifecycle of both		
2	Setting Up Git: Install Git, create repositories, and practice version control with basic commands (commit, push, pull)		
3	Branching and Merging in Git: Create and manage multiple branches, merge changes, and resolve conflicts		
4	Infrastructure as Code with Ansible: Write a simple Ansible playbook to install software (e.g., Apache) on a local/virtual machine		
5	Continuous Integration with GitHub Actions: Set up a simple CI pipeline that runs tests automatically on push		
6	Containerization with Docker: Build and run a Docker image for a sample application, exploring Dockerfile creation, container management, and pushing images to a registry		
7	Orchestration with Kubernetes: Deploy containerized workloads on a Kubernetes cluster, covering Pods, Deployments, Services, and scaling via rolling updates		
8	ML Experiment Tracking with MLflow: Log experiments, hyperparameters, metrics, and artifacts using MLflow to enable reproducibility and model comparison		
9	Model Deployment with Kubeflow: Automate end-to-end ML workflows (training, validation, serving) within a Kubernetes environment using Kubeflow Pipelines		
10	Observability and Logging in MLOps: Implement monitoring and logging (e.g., Prometheus, Grafana, ELK) to track performance, resource utilization, and application logs in ML pipelines		

Experiment 1

Aim

Introduction to DevOps and MLOps: Understand the basic principles, differences, and lifecycle of both

Theory

Today's software development and machine learning practices demand efficient processes, inter-team collaboration, and automation to manage complex systems effectively. Two practices that have emerged to address these demands are DevOps and MLOps.

DevOps

DevOps is a technical and cultural movement that closes the gap between software development and IT operations. Historically, both teams operated in silos, which resulted in slow delivery, unreliable environments, and deployment failures. DevOps strives to dismantle these silos by encouraging cooperation, continuous feedback, and end-to-end automation.

Major Objectives of DevOps:

- **Faster Deployment**: Build, testing, and release automation to ship features quickly.
- **Better Quality**: Regular testing and monitoring to identify problems early.
- **Scalability**: Infrastructure as Code (IaC) and containerization enable scaling applications with ease.
- **Reliability**: Continuous monitoring keeps systems stable and responsive.

DevOps Lifecycle:

- **Plan**: Determine features, objectives, and project scope.
- **Develop**: Writing and managing application code.
- **Build**: Compile code and create executable artifacts.
- **Test**: Automated and manual testing for assurance of quality.
- **Release**: Deployment to production environments.
- **Operate**: Monitoring and upkeep of systems in real-time.
- **Monitor**: Gathering data, logs, and performance metrics.

These tools such as *Git*, *Jenkins*, *Docker*, *Kubernetes*, and *Ansible* are commonly utilized in DevOps pipelines.

MLOps

MLOps refers to the application of DevOps practices to machine learning systems. Although sharing similar philosophy, MLOps mitigates special issues that emerge owing to the data-driven and experimentation-based nature of ML development.
Machine learning models are not simply code—they depend significantly on data, model training, and performance testing. In contrast to standard software, ML models can get worse

over time due to changing data distributions (referred to as model drift). MLOps practices seek to ensure models are versioned, tested, deployed, and monitored regularly.

MLOps Lifecycle:

- **Data Collection & Preparation**: Collecting and cleaning data sets.
- **Model Development**: Developing and testing ML models.
- **Model Training**: Training the model with data.
- **Model Evaluation**: Performance validation through metrics (accuracy, precision, etc.).
- **Model Deployment**: Deploying the model through APIs or integrating it into apps.
- **Model Monitoring**: Monitoring predictions, performance, and drift.
- **Model Retraining**: Periodically updating the model with new data.

Some of the popular MLOps tools are *MLflow*, *Kubeflow*, *TensorFlow Extended (TFX)*, *DVC (Data Version Control)*, and *Seldon Core*.

Differences

Category	**DevOps**	**MLOps**
Main Focus	Application development and delivery	Managing the ML lifecycle and model operations
Artifacts	Application code	Code, datasets, trained models, and metrics
Automation	Build, test, deploy	Data pipeline automation, training, and model deployment
Testing	Unit & integration tests	Data validation, model validation, performance tests
Monitoring	System performance, errors	Model accuracy, prediction quality, drift detection
Reusability	High (same code runs consistently)	Harder (data changes may require retraining)

MLOps can be considered a superset of DevOps with additional complexity due to the involvement of large datasets, model behavior, and continuous evaluation needs.

Conclusion

In this experiment, we learned the basic principles of DevOps and MLOps. We discovered that both have common such as automation, CI/CD, and monitoring but with the introduction of data and model management, MLOps has other challenges. Having an understanding of their lifecycle differences aids in creating more efficient workflow for application and machine learning development. This forms the basis of learning more about real-world DevOps and MLOps implementations in subsequent experiments.

Experiment 2

Aim

Setting Up Git: Install Git, create repositories, and practice version control with basic commands (commit, push, pull)

Theory

Version control is a cornerstone of modern software development. Whether you're working solo or with a team, being able to track changes, collaborate effectively, and roll back when something breaks is crucial. That's where Git comes in.

Git is a distributed version control system developed by Linus Torvalds (yes, the creator of Linux!) in 2005. Unlike older systems, Git gives every developer a full copy of the repository — history and all — making it incredibly fast, reliable, and flexible.

Git is used for:

- Track every change made to your code
- Work in parallel with teammates without stepping on each other's toes
- Easily roll back to a previous version if something goes wrong
- Integrates with platforms like GitHub, GitLab, and Bitbucket for remote collaboration

Experiment

Installing git

For Windows:

- Installer can be found in official Git website

For Linux:

```
sudo apt update
sudo apt install git
```

Creating a Local Repository

```
mkdir my-repo
cd my-repo
git init
```

Making a commit

```
git add hello.txt
git commit -m "Initial commit with hello.txt"
```

Pushing to GitHub

git remote add origin https://github.com/kinjalrk2k/my-repo.git
git branch -M main
git push -u origin main

Pulling changes

git pull origin main

Outputs

```
● ➜  Experiment-2 git:(main) ✗ mkdir my-repo
● ➜  Experiment-2 git:(main) ✗ cd my-repo
● ➜  my-repo git:(main) ✗ git init
  Initialized empty Git repository in /Users/kinjal/Desktop/DRIVE/STUDY/MTech/Semester 2/GitHub-Labs/DevOps-and-MLOps-Lab/Tuesday-Lab-Class/Experiment-2/my
  -repo/.git/
```

Creating a Local Repository

```
● ➜  my-repo git:(main) ✗ git add hello.txt
● ➜  my-repo git:(main) ✗ git commit -m "Initial commit with hello.txt"
  [main (root-commit) 1b8ef3d] Initial commit with hello.txt
   1 file changed, 1 insertion(+)
   create mode 100644 hello.txt
```

Making a commit

```
● ➜  my-repo git:(main) git remote add origin https://github.com/kinjalrk2k/my-repo.git
● ➜  my-repo git:(main) git branch -M main
● ➜  my-repo git:(main) git push -u origin main
  Enumerating objects: 3, done.
  Counting objects: 100% (3/3), done.
  Writing objects: 100% (3/3), 244 bytes | 244.00 KiB/s, done.
  Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
  remote: This repository moved. Please use the new location:
  remote:   https://github.com/Kinjalrk2k/my-repo.git
  To https://github.com/kinjalrk2k/my-repo.git
   * [new branch]      main -> main
  branch 'main' set up to track 'origin/main'.
```

Pushing to GitHub

```
● ➜  my-repo git:(main) git pull origin main
  From https://github.com/kinjalrk2k/my-repo
   * branch            main       -> FETCH_HEAD
  Already up to date.
```

Pulling changes

Conclusion

In this lab, we successfully set up Git on our local machine, created a repository, and practiced essential version control commands like commit, push, and pull. These may seem like small steps, but they form the foundation of professional software development.

Experiment 3

Aim

Branching and Merging in Git: Create and manage multiple branches, merge changes, and resolve conflicts

Theory

Working with branches in Git is one of the most powerful ways to manage changes in your project — especially when working in teams. Imagine being able to experiment freely without breaking the main codebase. That’s the magic of branching.

A branch in Git is like a parallel universe of your code. You can create a new branch, make changes, test things out, and when you're happy with it, you can merge it back into the main project. All without disrupting the original version.

By default, Git starts you off on the main (or master) branch. But as your project grows, you'll often find yourself creating branches for:

- New features (feature/login-system)
- Bug fixes (fix/button-alignment)
- Experiments (test/new-ui)

Once you've made changes in a branch and want to incorporate them into another branch (usually main), you perform a merge. Git tries to automatically combine the changes. But sometimes, it might need help — that’s when you’ll run into merge conflicts, which you’ll resolve manually.

Experiment

Creating new branch

```
git branch feature-1
```

Switch to new branch

```
git checkout feature-1
```

Merge back to main

```
git checkout main
git merge feature-1
```

Merge conflict

```
git merge feature-update
# Merge conflicts are manually fixed in the editor
git add welcome.txt
git commit -m "Resolved merge conflict between main and feature-update"
```

Outputs

```
● ➜  my-repo git:(main) git branch feature-1
● ➜  my-repo git:(main) git checkout feature-1
  Switched to branch 'feature-1'
```

Creating and switching branch

```
● ➜  my-repo git:(feature-1) git checkout main
  Switched to branch 'main'
  Your branch is up to date with 'origin/main'.
● ➜  my-repo git:(main) git merge feature-1
  Updating 1b8ef3d..b70e03f
  Fast-forward
   readme.md | 1 +
   1 file changed, 1 insertion(+)
   create mode 100644 readme.md
```

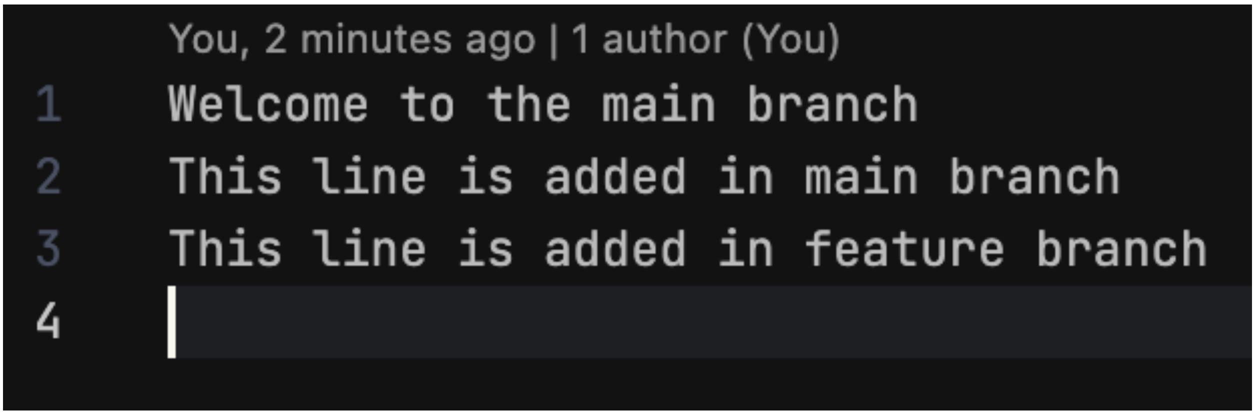
Merging back to main

```
⊗ ➜  my-repo git:(main) git merge feature-update
  Auto-merging welcome.txt
  CONFLICT (content): Merge conflict in welcome.txt
  Automatic merge failed; fix conflicts and then commit the result.
```

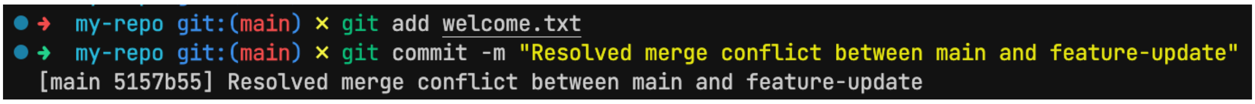
Merge conflicts

```
You, 17 seconds ago | 1 author (You)
Welcome to the main branch
Accept Current Change | Accept Incoming Change | Accept Both Changes | Compare Changes
<<<<<<< HEAD (Current Change)
This line is added in main branch
=======
This line is added in feature branch
>>>>>>> feature-update (Incoming Change)
```

Inspecting merge conflicts in editor



Fixing merge conflicts in editor



Resolved merge conflict

Conclusion

In this lab, we explored the core concepts of branching and merging in Git, which are foundational to collaborative and safe software development. We learned how to create branches for isolated development, merge changes back into the main branch, and resolve conflicts when changes overlap.

Experiment 4

Aim

Infrastructure as Code with Ansible: Write a simple Ansible playbook to install software (e.g., Apache) on a local/virtual machine

Theory

Managing infrastructure manually can quickly become inefficient and error-prone, especially as systems scale. That's where Infrastructure as Code (IaC) comes in. With IaC, we define and manage our server infrastructure using machine-readable configuration files rather than manual processes. One of the most popular tools for IaC is Ansible.

Ansible is an open-source automation tool used for configuration management, application deployment, and task automation. It's agentless, meaning it doesn't require any special software to be installed on the target systems — just SSH and Python.

Ansible uses YAML-based files called playbooks to describe the desired state of a system. These playbooks are easy to read and write, making Ansible very approachable even for beginners.

Common Ansible Use Cases:

- Installing and configuring software
- Managing users and permissions
- Automating routine system tasks
- Deploying applications to multiple servers

In this lab, we'll use Ansible to install Apache HTTP Server on a local or virtual machine using a simple playbook.

Experiment

hosts.ini

```
[local]
localhost ansible_connection=local
```

apache-install.yml

```
---
- name: Install Apache Web Server
  hosts: local
  become: yes

  tasks:
    - name: Ensure Apache is installed
      apt:
```

```
      name: apache2
      state: present
      update_cache: yes

  - name: Ensure Apache is running
    service:
      name: apache2
      state: started
      enabled: yes
```

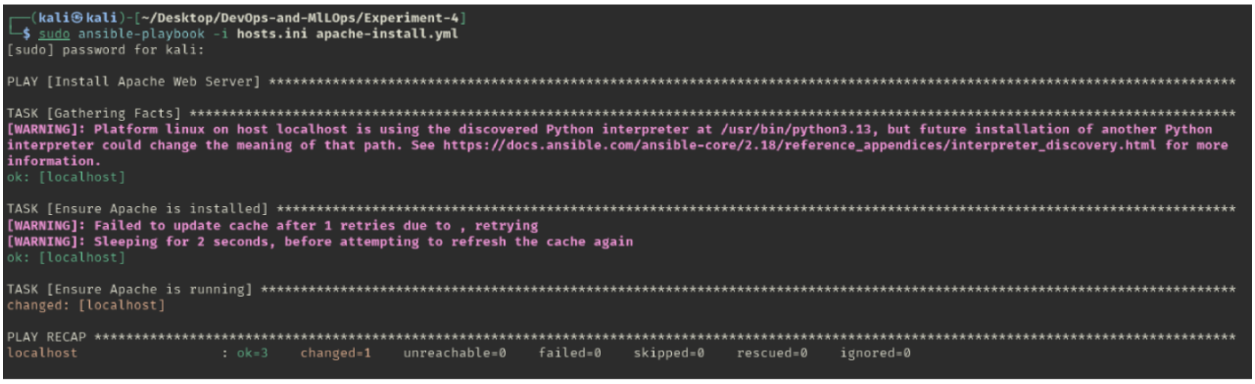
Running

ansible-playbook -i hosts.ini apache-install.yml

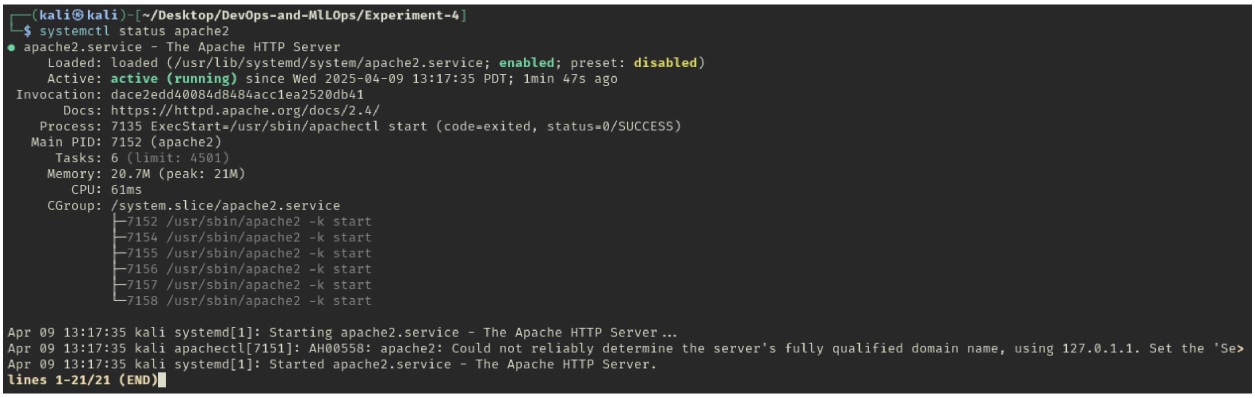
Verify

systemctl status apache2

Outputs



Running Ansible playbook



Verifying installation of Apache

Conclusion

In this lab, we took a hands-on approach to understanding Infrastructure as Code using Ansible. By writing a simple playbook to install Apache, we saw how system configuration tasks that normally involve multiple steps can be automated and standardized using just a few lines of YAML.

Experiment 5

Aim

Continuous Integration with GitHub Actions: Set up a simple CI pipeline that runs tests automatically on push

Theory

Modern software development isn't just about writing code — it's also about ensuring that code works consistently, reliably, and automatically. That's where Continuous Integration (CI) comes into play.

Continuous Integration is a development practice where developers integrate code into a shared repository frequently — sometimes several times a day. Each integration is then verified by an automated build and test process, allowing teams to detect problems early.

CI helps developers catch issues before they reach production, streamlining collaboration and maintaining code quality.

GitHub Actions is a feature built into GitHub that allows you to automate workflows — right from your repository. It lets you set up CI/CD pipelines that can automatically:

- Run tests when you push code
- Build your application
- Deploy to production
- Send alerts or messages
- And much more

GitHub Actions uses simple YAML-based workflows, making it accessible even for beginners. It also integrates tightly with your repository, so everything stays in one place — no external tools needed.

Experiment

main.py

```
def add(a, b):
    return a + b
```

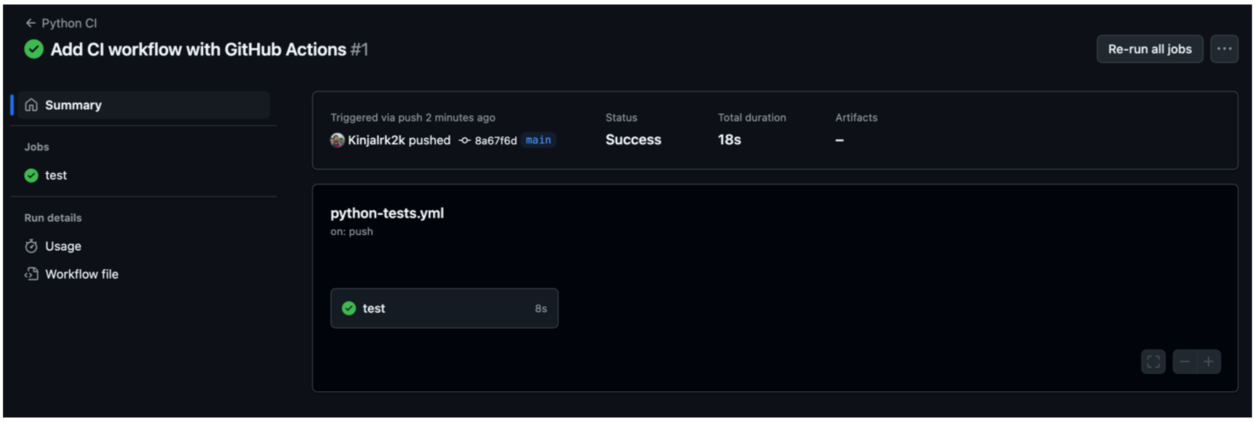
test_main.py

```
from main import add

def test_add():
    assert add(2, 3) == 5
```

Workflow file - .github/workflows/python-tests.yml

```
name: Python CI

on: [push]

jobs:
  test:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Set up Python
        uses: actions/setup-python@v4
        with:
          python-version: "3.10"

      - name: Install dependencies
        run: |
          python -m pip install --upgrade pip
          pip install pytest

      - name: Run tests
        run: |
          pytest
```

Outputs



CI tests running on GitHub Actions

Conclusion

In this lab, we successfully set up a Continuous Integration pipeline using GitHub Actions, where every code push automatically triggered a test run. We wrote a basic test, defined a CI workflow, and verified it on GitHub — all within a few minutes.

Experiment 6

Aim

Containerization with Docker: Build and run a Docker image for a sample application, exploring Dockerfile creation, container management, and pushing images to a registry

Theory

Imagine you're working on an application that runs perfectly on your machine, but when your teammate tries it — it breaks. Different OS, different dependencies, or even slightly different versions can all lead to frustrating bugs. This is exactly the problem that Docker was designed to solve.

Docker is a platform that enables you to package applications — along with all their dependencies — into standardized containers. These containers can run anywhere: your laptop, a cloud server, or even inside a CI/CD pipeline.

A Docker container is like a lightweight, standalone box that contains everything an application needs to run: code, runtime, system tools, libraries, and settings.

Reasons to use Containers

- Consistency across environments — no more "it works on my machine"
- Isolation — containers run separately from each other and the host system
- Portability — you can move containers across environments effortlessly
- Efficiency — they're faster and lighter than full virtual machines

Experiment

Building a sample application

app.py

```
from flask import Flask, request, jsonify
app = Flask(__name__)

@app.route("/", methods=["GET"])
def echo():
    return jsonify({"headers": dict(request.headers)})

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=9000)
```

requirements.txt

```
flask
```

Dockerfile

FROM python:3.10-slim

WORKDIR /app

COPY . .

RUN pip install -r requirements.txt

EXPOSE 9000

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

Building Docker Image

docker build -t flask-echo-headers-server .

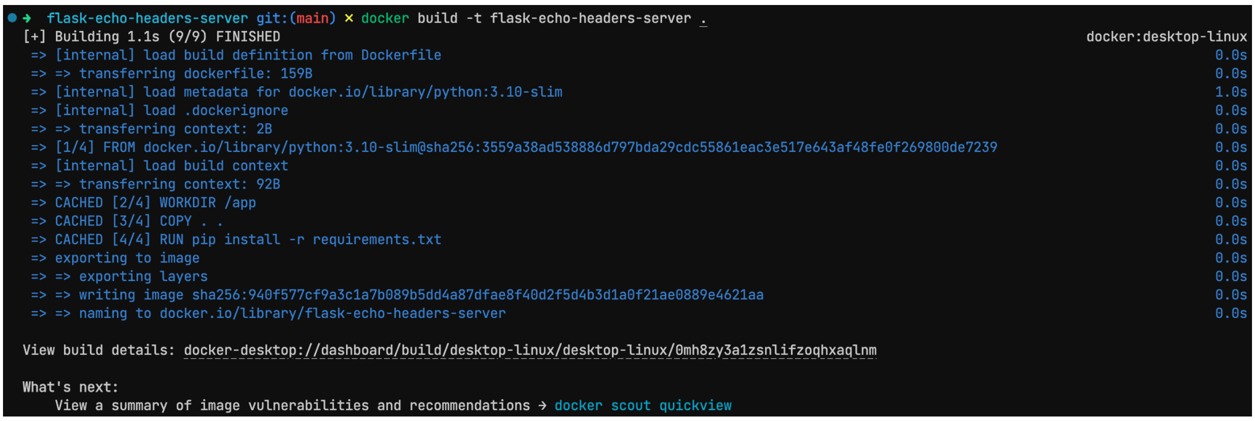
Running Docker Container

docker run -p 9000:9000 flask-echo-headers-server

Pushing to Registry

docker login
docker tag flask-echo-headers-server kinjal2209/flask-echo-headers-server
docker push kinjal2209/flask-echo-headers-server

Outputs



```
● ➜  flask-echo-headers-server git:(main) ✗ docker build -t flask-echo-headers-server .
 [+] Building 1.1s (9/9) FINISHED                                                                          docker:desktop-linux
  => [internal] load build definition from Dockerfile                                                                      0.0s
  => => transferring dockerfile: 159B                                                                                      0.0s
  => [internal] load metadata for docker.io/library/python:3.10-slim                                                       1.0s
  => [internal] load .dockerignore                                                                                         0.0s
  => => transferring context: 2B                                                                                           0.0s
  => [1/4] FROM docker.io/library/python:3.10-slim@sha256:3559a38ad538886d797bda29cdc55861eac3e517e643af48fe0f269800de7239  0.0s
  => [internal] load build context                                                                                         0.0s
  => => transferring context: 92B                                                                                          0.0s
  => CACHED [2/4] WORKDIR /app                                                                                             0.0s
  => CACHED [3/4] COPY . .                                                                                                 0.0s
  => CACHED [4/4] RUN pip install -r requirements.txt                                                                      0.0s
  => exporting to image                                                                                                    0.0s
  => => exporting layers                                                                                                   0.0s
  => => writing image sha256:940f577cf9a3c1a7b089b5dd4a87dfae8f40d2f5d4b3d1a0f21ae0889e4621aa                              0.0s
  => => naming to docker.io/library/flask-echo-headers-server                                                              0.0s

 View build details: docker-desktop://dashboard/build/desktop-linux/desktop-linux/0mh8zy3a1zsnlifzoqhxaqlnm

 What's next:
     View a summary of image vulnerabilities and recommendations → docker scout quickview
```

Building Docker Image

```
○ ➜  flask-echo-headers-server git:(main) ✗ docker run -p 6000:6000 flask-echo-headers-server
 * Serving Flask app 'app'
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:6000
 * Running on http://172.17.0.2:6000
Press CTRL+C to quit
```

Running Docker Container

```
● ➜  flask-echo-headers-server git:(main) ✗ docker login
  Authenticating with existing credentials...
  Login Succeeded
● ➜  flask-echo-headers-server git:(main) ✗ docker tag flask-echo-headers-server kinjal2209/flask-echo-headers-server
● ➜  flask-echo-headers-server git:(main) ✗ docker push kinjal2209/flask-echo-headers-server
  Using default tag: latest
  The push refers to repository [docker.io/kinjal2209/flask-echo-headers-server]
  3a118dba24e9: Pushed
  a54f78598602: Pushed
  ccf837cb9481: Pushed
  e56d47134802: Mounted from library/python
  2464acbfd551: Mounted from library/python
  72bbb1662472: Mounted from library/python
  c9b18059ed42: Mounted from library/python
  latest: digest: sha256:bd2a5b24d609302aa4a5abf715ec89160e3b1d49e884fe9dbfd8d3269a83582e size: 1783
```

Pushing to Registry

Conclusion

In this lab, we explored containerization using Docker — from writing a Dockerfile to running and pushing an image. We saw how Docker packages applications into isolated containers that can run consistently across different environments.

Experiment 7

Aim

Orchestration with Kubernetes: Deploy containerized workloads on a Kubernetes cluster, covering Pods, Deployments, Services, and scaling via rolling updates

Theory

As applications become more complex — with multiple containers, services, and environments — managing them manually can become nearly impossible. That's where Kubernetes comes in. Often referred to as "K8s" Kubernetes is the industry-standard platform for orchestrating containerized applications at scale.

Kubernetes is an open-source container orchestration system developed by Google and now maintained by the Cloud Native Computing Foundation (CNCF). Its primary role is to automate deployment, scaling, and management of containerized applications.

It doesn't just run containers — it handles the entire lifecycle of an application, ensuring availability, stability, and scalability in a production environment.

Experiment

Startup Minikube for local Kubernetes cluster

minikube start

Kubernetes setup

deplyment.yaml

```
apiVersion: apps/v1

kind: Deployment

metadata:
  name: flask-echo-headers-server

spec:
  replicas: 2
  selector:
    matchLabels:
      app: flask-echo-headers-server
  template:
    metadata:
      labels:
        app: flask-echo-headers-server
    spec:
```

```yaml
      containers:
        - name: web
          image: flask-echo-headers-server
          imagePullPolicy: Never
          ports:
            - containerPort: 9000
```

service.yaml

```yaml
apiVersion: v1

kind: Service

metadata:
  name: flask-echo-headers-service

spec:
  selector:
    app: flask-echo-headers-server
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9000
  type: NodePort
```

Build docker container within minikube

```
eval $(minikube docker-env)
docker build -t my-k8s-app .
```

Deploy service

```
kubectl apply -f deployment.yaml
kubectl apply -f service.yaml
kubectl port-forward service/flask-echo-headers-service 8080:80
```

Scaling

```
kubectl scale deployment flask-echo-headers-server --replicas=5
```

Rolling deployments

```
kubectl apply -f deployment.yaml
kubectl rollout status deployment/flask-echo-headers-server
```

Outputs

```
→ Tuesday-Lab-Class git:(main) ✗ minikube start
😄  minikube v1.25.2 on Darwin 15.3.2 (arm64)
🎉  minikube 1.35.0 is available! Download it: https://github.com/kubernetes/minikube/releases/tag/v1.35.0
💡  To disable this notice, run: 'minikube config set WantUpdateNotification false'

✨  Automatically selected the docker driver
👍  Starting control plane node minikube in cluster minikube
🚜  Pulling base image ...
💾  Downloading Kubernetes v1.23.3 preload ...
    > preloaded-images-k8s-v17-v1...: 419.07 MiB / 419.07 MiB  100.00% 15.11 Mi
    > gcr.io/k8s-minikube/kicbase: 343.12 MiB / 343.12 MiB  100.00% 4.78 MiB p/
    > gcr.io/k8s-minikube/kicbase: 0 B [_________________________] ?% ? p/s 0s
    > index.docker.io/kicbase/sta...: 343.12 MiB / 343.12 MiB  100.00% 9.51 MiB
    > index.docker.io/kicbase/sta...: 0 B [_______________________] ?% ? p/s 0s
    > gcr.io/k8s-minikube/kicbase...: 343.12 MiB / 343.12 MiB  100.00% 6.56 MiB
    > gcr.io/k8s-minikube/kicbase...: 0 B [_______________________] ?% ? p/s 0s
    > index.docker.io/kicbase/sta...: 343.12 MiB / 343.12 MiB  100.00% 16.32 Mi
    > index.docker.io/kicbase/sta...: 0 B [_______________________] ?% ? p/s 0s
❗  minikube was unable to download gcr.io/k8s-minikube/kicbase:v0.0.30, but successfully downloaded docker.io/kicbase/stable:v0.0.30 as a fallback image
E0410 12:35:22.231716   65613 cache.go:203] Error downloading kic artifacts:  failed to download kic base image or any fallback image
🔥  Creating docker container (CPUs=2, Memory=2200MB) ...
🐳  Preparing Kubernetes v1.23.3 on Docker 20.10.12 ...
    ▪ kubelet.housekeeping-interval=5m
    ▪ Generating certificates and keys ...
    ▪ Booting up control plane ...
    ▪ Configuring RBAC rules ...
🔎  Verifying Kubernetes components...
    ▪ Using image gcr.io/k8s-minikube/storage-provisioner:v5
🌟  Enabled addons: storage-provisioner
🏄  Done! kubectl is now configured to use "minikube" cluster and "default" namespace by default
```

minikube starting

```
→ kube-deployment git:(main) ✗ kubectl apply -f deployment.yaml
deployment.apps/flask-echo-headers-server created
→ kube-deployment git:(main) ✗ kubectl apply -f service.yaml
service/flask-echo-headers-service created
→ kube-deployment git:(main) ✗ kubectl port-forward service/flask-echo-headers-service 8080:80
Forwarding from 127.0.0.1:8080 -> 9000
Forwarding from [::1]:8080 -> 9000
Handling connection for 8080
```

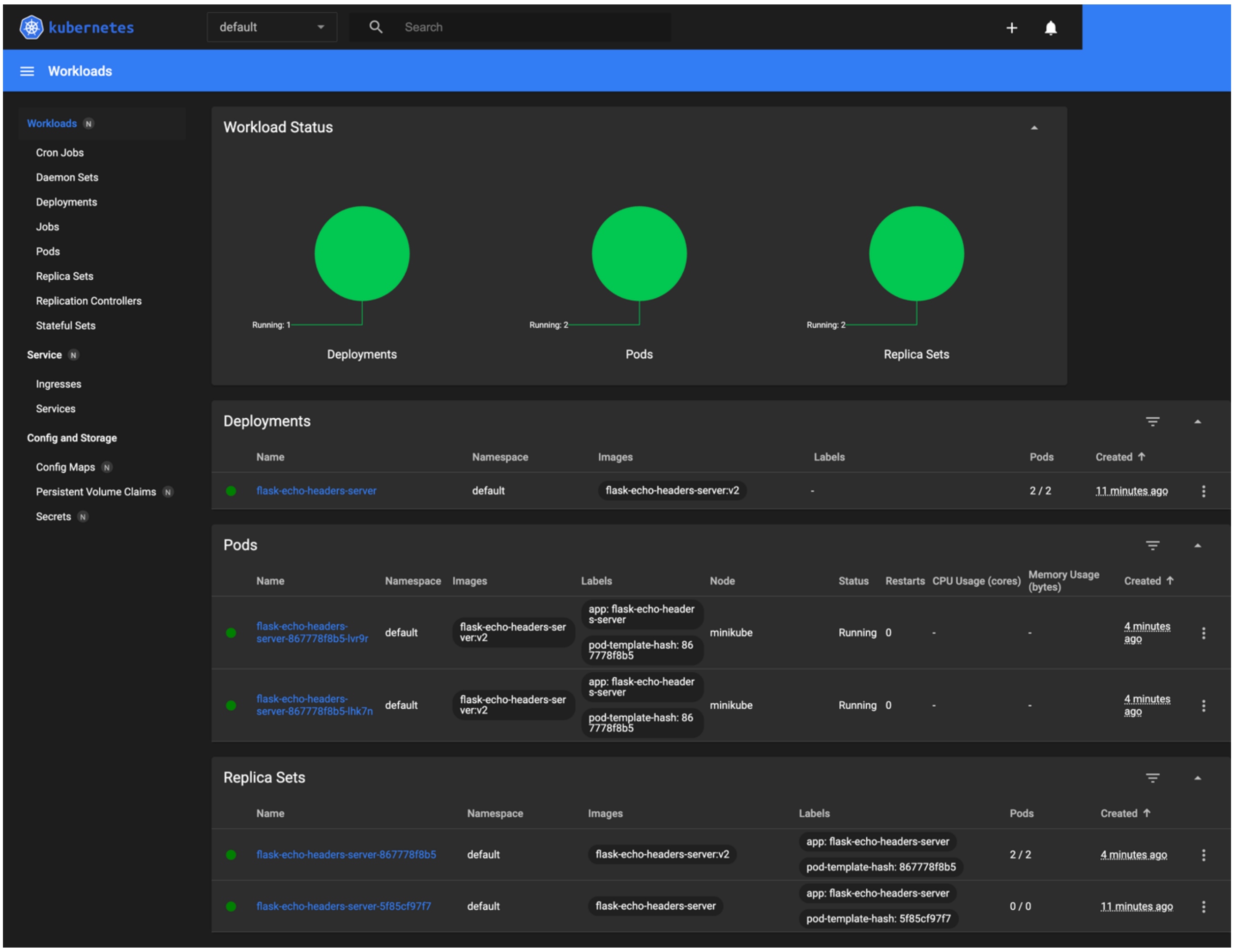
Deploying service

```
→ kube-deployment git:(main) ✗ kubectl scale deployment flask-echo-headers-server --replicas=5
deployment.apps/flask-echo-headers-server scaled
```

Scaling

```
→ kube-deployment git:(main) ✗ kubectl apply -f deployment.yaml
deployment.apps/flask-echo-headers-server configured
→ kube-deployment git:(main) ✗ kubectl rollout status deployment/flask-echo-headers-server
deployment "flask-echo-headers-server" successfully rolled out
```

Rolling deployments



Minikube dashboard

Conclusion

In this lab, we stepped into the world of container orchestration with Kubernetes. We learned how to:

- Create and manage pods using deployments
- Expose our app using services
- Perform rolling updates for smooth, zero-downtime deployments
- Scale applications with just one command

Experiment 8

Aim

ML Experiment Tracking with MLflow: Log experiments, hyperparameters, metrics, and artifacts using MLflow to enable reproducibility and model comparison

Theory

In machine learning, especially in real-world projects, we rarely build just one model. We often try out different algorithms, tweak hyperparameters, and change training data to improve performance. Over time, this leads to multiple versions of models — and without a structured way to keep track, it's nearly impossible to reproduce results or understand which version worked best.

That’s where MLflow becomes essential.

MLflow is an open-source platform that helps manage the end-to-end machine learning lifecycle. One of its core modules, MLflow Tracking, is specifically built for logging and organizing machine learning experiments. With MLflow, you can log key aspects like:

- Hyperparameters used in training (e.g., learning rate, batch size)
- Metrics like accuracy, RMSE, or loss
- Artifacts such as trained models or output plots
- Code versions and environments

This creates a reliable history of experiments, making it easy to reproduce past results and compare models side by side.

Experiment

Setup MLflow

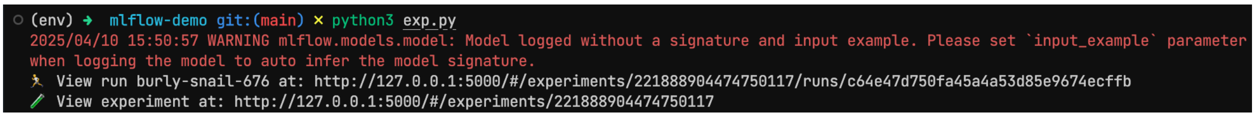
```
pip install mlflow
mlflow ui
```

ML experiment

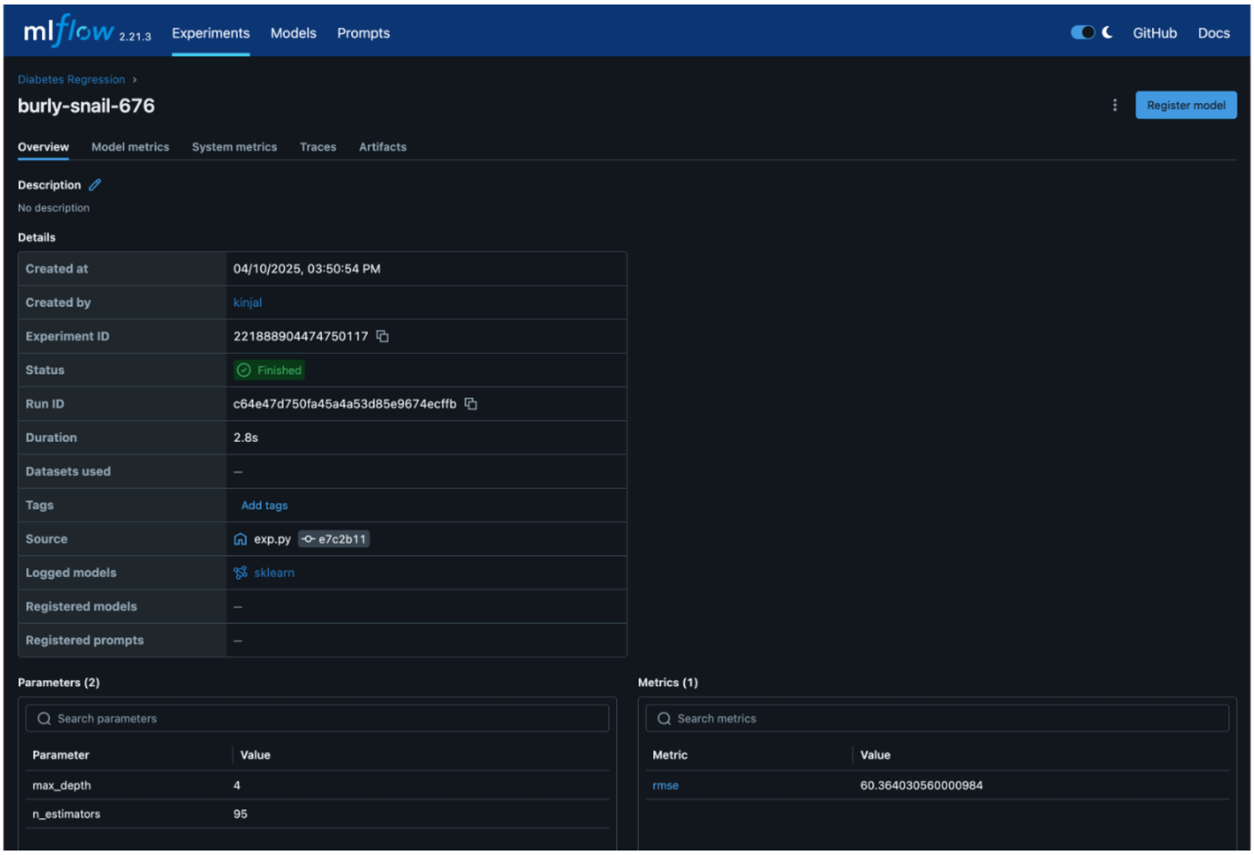
```
import mlflow
import mlflow.sklearn
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error
from sklearn.datasets import load_diabetes
from sklearn.model_selection import train_test_split

# Load dataset
X, y = load_diabetes(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
```

```
# Set experiment
mlflow.set_experiment("Diabetes Regression")

# Start a new MLflow run
with mlflow.start_run():
    # Set hyperparameters
    n_estimators = 150
    max_depth = 6

    # Train model
    model = RandomForestRegressor(n_estimators=n_estimators, max_depth=max_depth)
    model.fit(X_train, y_train)

    # Predict and evaluate
    preds = model.predict(X_test)
    rmse = mean_squared_error(y_test, preds, squared=False)

    # Log everything
    mlflow.log_param("n_estimators", n_estimators)
    mlflow.log_param("max_depth", max_depth)
    mlflow.log_metric("rmse", rmse)
    mlflow.sklearn.log_model(model, "model")
```

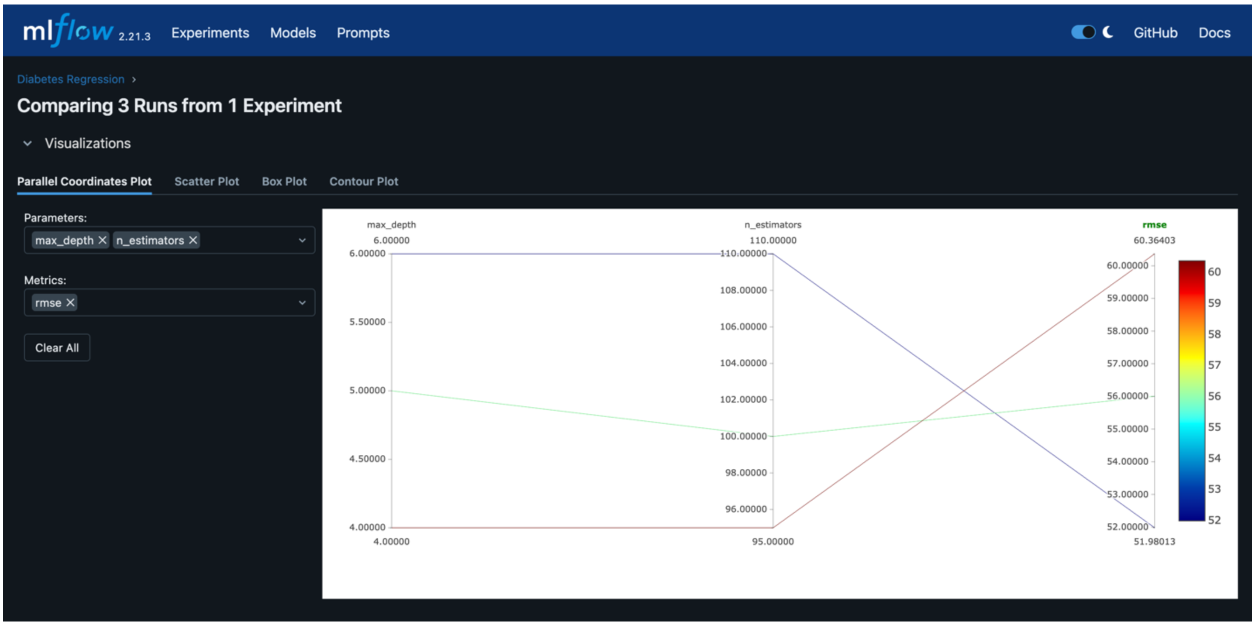
Outputs



Running ML experiment



Logging hyperparameters, metrics and artifacts



Model comparison

Conclusion

In this lab, we learned how to use MLflow to bring structure, traceability, and transparency into the machine learning workflow. By logging experiments with hyperparameters, evaluation metrics, and trained models, MLflow makes it incredibly easy to:

- Reproduce previous model results with confidence
- Compare different model runs quickly and visually
- Collaborate with teammates on model development
- Scale from local experiments to team-wide MLOps pipelines

Experiment 9

Aim

Model Deployment with Kubeflow: Automate end-to-end ML workflows (training, validation, serving) within a Kubernetes environment using Kubeflow Pipelines

Theory

Machine learning models go through many stages — from data preprocessing and model training to validation and finally deployment. Managing all these steps manually becomes increasingly complex and error-prone, especially when teams are working on multiple models or deploying frequently. That's where Kubeflow Pipelines come into play.

Kubeflow is an open-source MLOps platform built on Kubernetes. It provides a set of tools that make it easier to develop, orchestrate, deploy, and manage ML workflows in a scalable and reproducible way — all while leveraging Kubernetes' strengths like scalability and container orchestration.

Kubeflow Pipelines is a core component of Kubeflow that helps automate ML workflows as a series of steps. Each step can be a containerized operation — like loading data, training a model, validating it, or deploying it. These steps are defined as DAGs (Directed Acyclic Graphs) and run seamlessly in Kubernetes.

Experiment

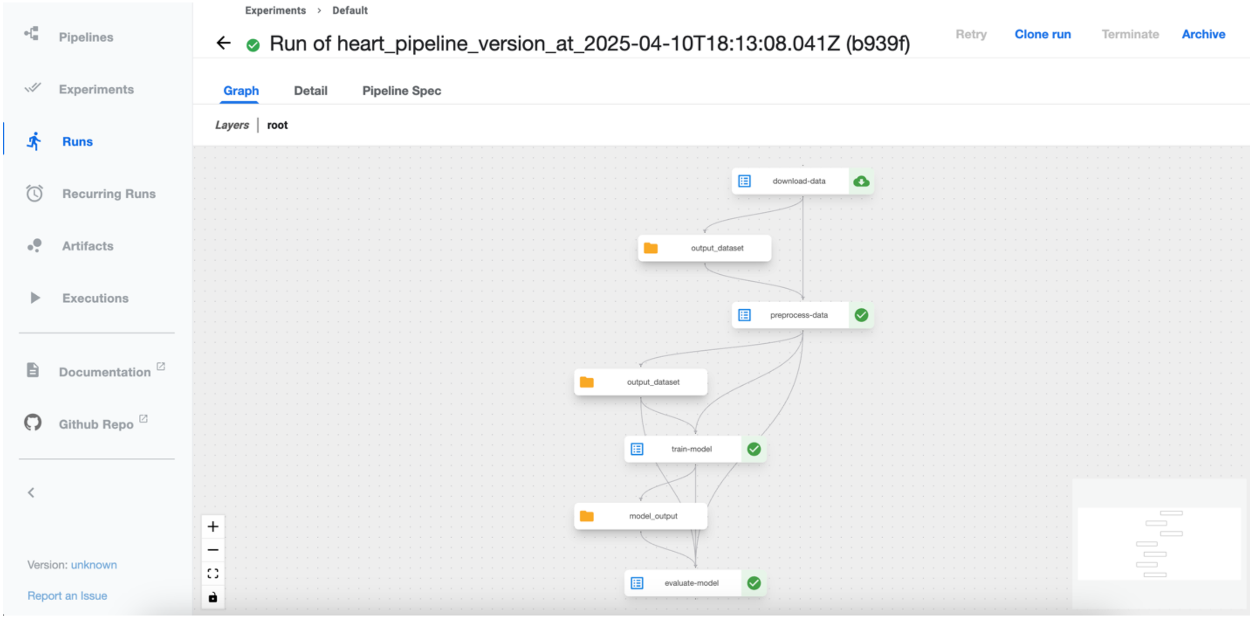
Pipeline

```
import kfp
from kfp import dsl
from kfp.dsl import component, Input, Output, Dataset, Model

@component(
    packages_to_install=["pandas"]
)
def download_data(output_dataset: Output[Dataset]):
    import os
    import pandas as pd
    url = "https://raw.githubusercontent.com/sharmaroshan/Heart-UCI-
Dataset/refs/heads/master/heart.csv"
    df = pd.read_csv(url)

    os.makedirs(output_dataset.path, exist_ok=True)
    df.to_csv(os.path.join(output_dataset.path, "heart.csv"), index=False)

@component(
    packages_to_install=["pandas", "scikit-learn"]
```

```
)
def preprocess_data(input_dataset: Input[Dataset], output_dataset: Output[Dataset]):
    import pandas as pd
    from sklearn.model_selection import train_test_split
    import os

    df = pd.read_csv(input_dataset.path + "/heart.csv")
    X = df.drop("target", axis=1)
    y = df["target"]
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

    train_df = X_train.copy()
    train_df["target"] = y_train
    test_df = X_test.copy()
    test_df["target"] = y_test

    os.makedirs(output_dataset.path, exist_ok=True)

    train_df.to_csv(output_dataset.path + "/train.csv", index=False)
    test_df.to_csv(output_dataset.path + "/test.csv", index=False)

@component(
    packages_to_install=["pandas", "scikit-learn", "joblib"]
)
def train_model(preprocessed_dataset: Input[Dataset], model_output: Output[Model]):
    import pandas as pd
    from sklearn.linear_model import LogisticRegression
    import joblib
    import os

    df = pd.read_csv(preprocessed_dataset.path + "/train.csv")
    X_train = df.drop("target", axis=1)
    y_train = df["target"]

    model = LogisticRegression(max_iter=1000)
    model.fit(X_train, y_train)

    os.makedirs(model_output.path, exist_ok=True)

    joblib.dump(model, model_output.path + "/model.joblib")

@component(
    packages_to_install=["pandas", "scikit-learn", "joblib"]
)
def evaluate_model(preprocessed_dataset: Input[Dataset], model_input: Input[Model]):
    import pandas as pd
```

```python
    import joblib
    from sklearn.metrics import classification_report

    df = pd.read_csv(preprocessed_dataset.path + "/test.csv")
    X_test = df.drop("target", axis=1)
    y_test = df["target"]

    model = joblib.load(model_input.path + "/model.joblib")
    y_pred = model.predict(X_test)

    report = classification_report(y_test, y_pred)
    print("Classification Report:\n", report)

@dsl.pipeline(
    name="heart-disease-pipeline",
    description="Heart Disease Prediction Pipeline"
)
def heart_disease_pipeline():
    raw_data = download_data()

    preprocessed = preprocess_data(
        input_dataset=raw_data.outputs["output_dataset"]
    )

    trained_model = train_model(
        preprocessed_dataset=preprocessed.outputs["output_dataset"]
    )

    evaluate_model(
        preprocessed_dataset=preprocessed.outputs["output_dataset"],
        model_input=trained_model.outputs["model_output"]
    )

from kfp.v2 import compiler

compiler.Compiler().compile(
    pipeline_func=heart_disease_pipeline,
    package_path="heart_pipeline.yaml",
)
```

Output



Running Experiment on Kubeflow

Conclusion

In this lab, we successfully built and ran a simple ML pipeline using Kubeflow Pipelines on Kubernetes. By breaking the ML workflow into discrete, containerized steps, Kubeflow enabled:

- Automation of repetitive and error-prone tasks
- Reproducibility of results across environments and team members
- Scalability using Kubernetes' native orchestration
- Model comparison and tracking through UI and logs

This approach not only simplifies deployment but also ensures that models are built and delivered in a production-ready, version-controlled, and collaborative manner — making it ideal for modern MLOps workflows.

Experiment 10

Aim

Observability and Logging in MLOps: Implement monitoring and logging (e.g., Prometheus, Grafana, ELK) to track performance, resource utilization, and application logs in ML pipelines)

Thoery

In MLOps, deploying a model is just the beginning. Ensuring its reliable operation in production is where the real challenge lies. For that, observability becomes a critical capability — it helps teams understand what's happening within their ML systems and pipelines at any given time.

Where monitoring tracks known metrics (like CPU, memory, accuracy), observability provides the ability to explore unknowns — such as unexpected behavior or drift in the pipeline — by analyzing logs, metrics, and traces.

- Track resource usage: Helps identify bottlenecks in ETL steps, model training, and serving.
- Detect failures early: Alerts when a DAG step or batch job fails.
- Understand system health: Monitor data ingestion rates, latency, and throughput.
- Improve collaboration: Easier debugging across teams (data engineers, ML engineers, ops).

Experiment

<u>Docker setup for the entire stack</u>

```
version: "3.8"

services:
  airflow:
    build: ./airflow
    container_name: airflow
    environment:
      - AIRFLOW__CORE__EXECUTOR=SequentialExecutor
      - AIRFLOW__CORE__LOAD_EXAMPLES=False
      - AIRFLOW__WEBSERVER__RBAC=True
    volumes:
      - ./airflow/dags:/opt/airflow/dags
      - ./airflow/logs:/opt/airflow/logs
    ports:
      - "8080:8080"
      - "8793:8793" # Prometheus exporter
    command: >
      bash -c "airflow db init &&
               airflow users create --username admin --password admin --firstname Admin --lastname
User --role Admin --email admin@example.com &&
               airflow webserver & airflow scheduler"
```

```yaml
  prometheus:
    image: prom/prometheus:latest
    volumes:
      - ./prometheus/prometheus.yml:/etc/prometheus/prometheus.yml
    ports:
      - "9090:9090"

  grafana:
    image: grafana/grafana:latest
    ports:
      - "3000:3000"
    depends_on:
      - prometheus
    environment:
      - GF_SECURITY_ADMIN_USER=admin
      - GF_SECURITY_ADMIN_PASSWORD=admin

  elasticsearch:
    image: docker.elastic.co/elasticsearch/elasticsearch:7.17.10
    environment:
      - discovery.type=single-node
    ports:
      - "9200:9200"

  kibana:
    image: docker.elastic.co/kibana/kibana:7.17.10
    ports:
      - "5601:5601"
    depends_on:
      - elasticsearch

  filebeat:
    image: docker.elastic.co/beats/filebeat:7.17.10
    volumes:
      - ./filebeat/filebeat.yml:/usr/share/filebeat/filebeat.yml
      - ./airflow/logs:/usr/share/airflow/logs
    depends_on:
      - elasticsearch
      - kibana
```

Prometheus setup

```yaml
global:
  scrape_interval: 5s

scrape_configs:
  - job_name: "airflow"
```

```
  static_configs:
    - targets: ["localhost:9090"]
```

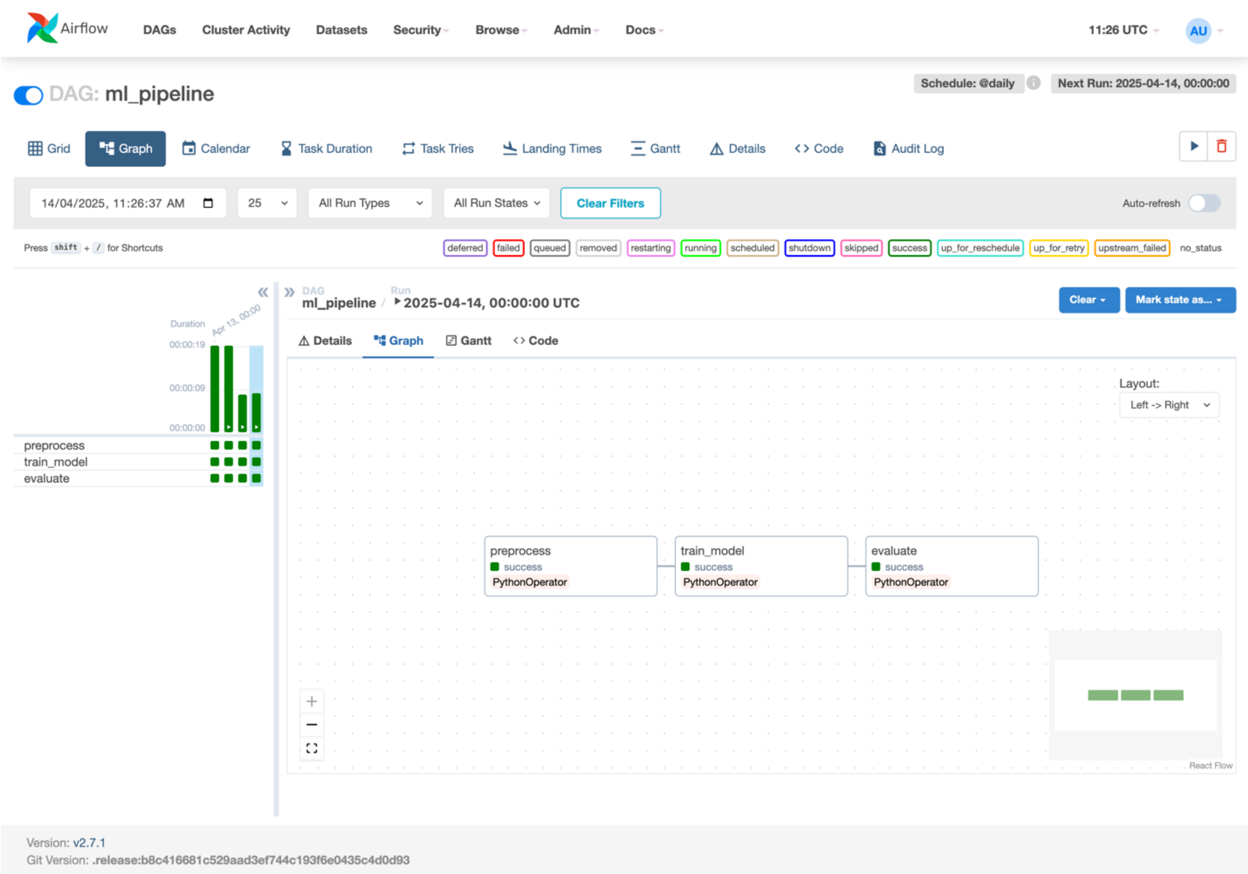
Filebeat setup

```
filebeat.inputs:
  - type: log
    enabled: true
    paths:
      - /usr/share/airflow/logs/**/*.log

output.elasticsearch:
  hosts: ["http://elasticsearch:9200"]

setup.kibana:
  host: "kibana:5601"
```

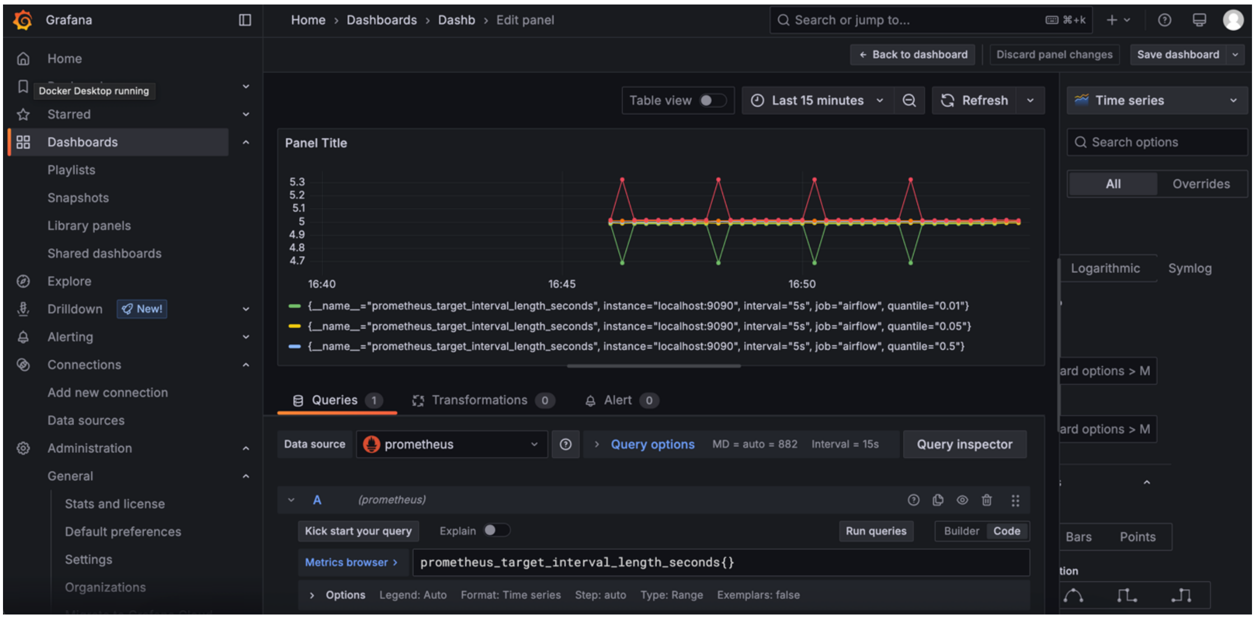
Running

```
docker-compose up -d
```

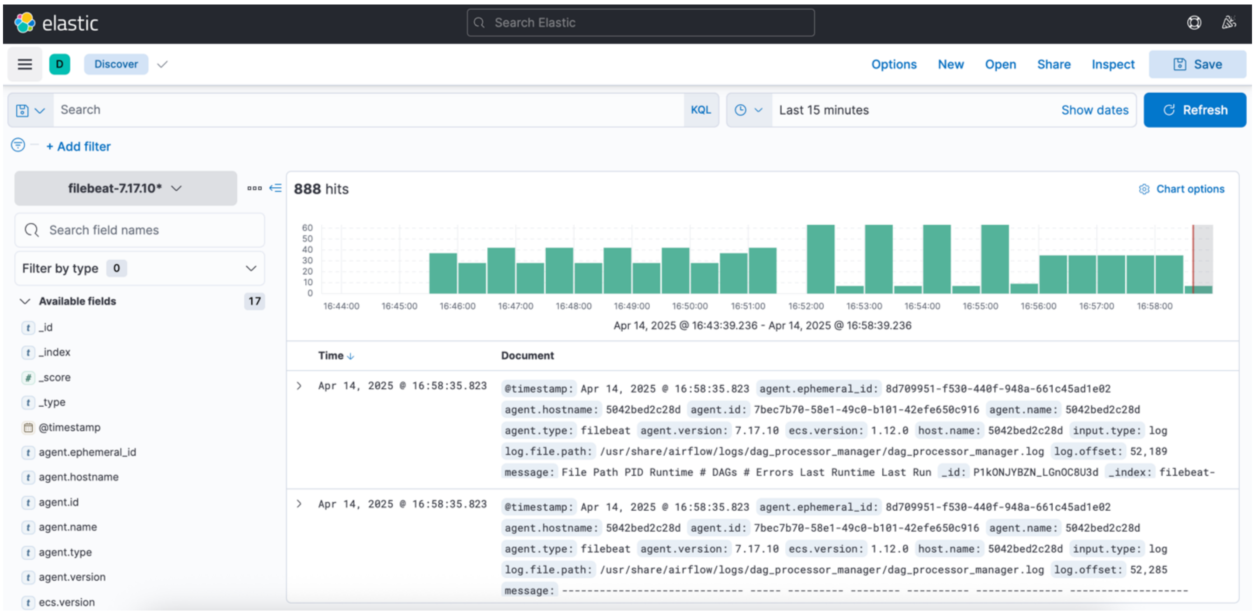
Outputs



Airflow DAG



Grafana dashboard



Kibana Dashboard

Conclusion

This experiment brings practical observability into MLOps workflows. Rather than treating pipelines as black boxes, tools like Prometheus and ELK open them up to inspection — from performance bottlenecks to silent failures.

- By combining metrics, logging, and visual dashboards, teams can:
- Debug faster
- Prevent issues proactively
- Maintain healthy ML pipelines in production

Observability isn’t just for ops anymore — it's a foundational skill for ML engineers who want to own their models beyond deployment.