INDEX

S. No.	Experiment	Date	Sign
1	Configuration Management with Terraform: Write Terraform code to provision infrastructure (e.g., AWS EC2 instance or Azure VM) and understand providers, variables, and state management.		
2	Secrets Management with HashiCorp Vault: Securely store and access secrets like API keys and credentials using Vault and integrate it into your deployment pipeline.		
3	Model Serving with FastAPI — Deploy a trained ML model using FastAPI to expose endpoints like /predict and test with real input data.		
4	Automated ML Pipelines with Apache Airflow: Build a Directed Acyclic Graph (DAG) in Airflow to automate tasks such as data loading, preprocessing, training, and evaluation.		
5	Hyperparameter Tuning with Optuna – Implement Optuna for automated hyperparameter optimization and integrate it with MLflow for experiment logging.		
6	Data Versioning with DVC: Use DVC to track versions of datasets and models, and connect it with Git for full ML lifecycle version control.		
7	Basic Monitoring with Prometheus and Grafana: Set up Prometheus to monitor system resources and visualize the metrics using Grafana dashboards.		
8	Deploy ML Model using Jenkins Pipeline: Create a Jenkins pipeline that trains, tests, and deploys a machine learning model.		
9	A/B Testing of ML Models: Deploy two versions of a model and perform A/B testing using load balancers or routing in Kubernetes.		
10	Model Drift Detection: Use tools like Evidently or custom scripts to monitor and detect data/model drift over time.		

Experiment 1

Aim

Configuration Management with Terraform: Write Terraform code to provision infrastructure (e.g., AWS EC2 instance or Azure VM) and understand providers, variables, and state management.

Theory

In modern DevOps and MLOps practices, managing infrastructure through code — known as Infrastructure as Code (IaC) — is essential for consistency, scalability, and automation. One of the most widely adopted tools for this purpose is Terraform, an open-source IaC tool developed by HashiCorp.

Terraform allows users to define and provision cloud infrastructure using a declarative configuration language called HCL (HashiCorp Configuration Language). Instead of manually clicking through a cloud console, infrastructure is defined in .tf files that can be version-controlled and reused across environments.

Providers: These are plugins that allow Terraform to interact with cloud platforms (e.g., AWS, Azure, GCP). Each provider exposes resources and data sources.

Resources: The actual infrastructure elements you want to create, such as EC2 instances, S3 buckets, or Azure VMs.

Variables: Used for dynamic configuration. Helps avoid hardcoding values and enables reusability.

State Management: Terraform keeps track of what infrastructure it manages via a state file. This file is crucial for determining changes and performing updates.

Plan, Apply, Destroy: The main Terraform workflow involves terraform plan, terraform apply, and optionally terraform destroy, allowing safe preview and execution of infrastructure changes.

Experiment

main.tf

```
provider "aws" {
  region = var.aws_region
}

resource "aws_instance" "mlops_vm" {
  ami           = var.ami_id
  instance_type = var.instance_type
  key_name      = var.key_name
```

```
  tags = {
    Name = "MLOps-Lab-Instance"
  }
}
```

terraform.tfvars

```
ami_id       = "ami-0c02fb55956c7d316" # Amazon Linux 2 AMI
key_name     = "your-ssh-key-name"
```

variables.tf

```
variable "aws_region" {
  type    = string
  default = "us-east-1"
}

variable "ami_id" {
  description = "AMI ID for the EC2 instance"
  type        = string
}

variable "instance_type" {
  default = "t2.micro"
  type    = string
}

variable "key_name" {
  description = "SSH key name in AWS"
  type        = string
}
```

outputs.tf

```
output "instance_id" {
  value = aws_instance.mlops_vm.id
}

output "public_ip" {
  value = aws_instance.mlops_vm.public_ip
}
```

Commands

```
terraform init
terraform plan
```

terraform apply

Outputs

```
➜  terraform-demo git:(main) ✗ terraform init
  Initializing the backend...
  Initializing provider plugins...
  - Finding latest version of hashicorp/aws...
  - Installing hashicorp/aws v5.94.1...
  - Installed hashicorp/aws v5.94.1 (signed by HashiCorp)
  Terraform has created a lock file .terraform.lock.hcl to record the provider
  selections it made above. Include this file in your version control repository
  so that Terraform can guarantee to make the same selections by default when
  you run "terraform init" in the future.

  Terraform has been successfully initialized!

  You may now begin working with Terraform. Try running "terraform plan" to see
  any changes that are required for your infrastructure. All Terraform commands
  should now work.

  If you ever set or change modules or backend configuration for Terraform,
  rerun this command to reinitialize your working directory. If you forget, other
  commands will detect it and remind you to do so if necessary.
```

terraform init

```
➜  terraform-demo git:(main) ✗ terraform plan

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # aws_instance.mlops_vm will be created
  + resource "aws_instance" "mlops_vm" {
      + ami                                  = "ami-0c02fb55956c7d316"
      + arn                                  = (known after apply)
      + associate_public_ip_address          = (known after apply)
      + availability_zone                    = (known after apply)
      + cpu_core_count                       = (known after apply)
      + cpu_threads_per_core                 = (known after apply)
      + disable_api_stop                     = (known after apply)
      + disable_api_termination              = (known after apply)
      + ebs_optimized                        = (known after apply)
      + enable_primary_ipv6                  = (known after apply)
      + get_password_data                    = false
      + host_id                              = (known after apply)
      + host_resource_group_arn              = (known after apply)
      + iam_instance_profile                 = (known after apply)
      + id                                   = (known after apply)
      + instance_initiated_shutdown_behavior = (known after apply)
      + instance_lifecycle                   = (known after apply)
      + instance_state                       = (known after apply)
      + instance_type                        = "t2.micro"
      + ipv6_address_count                   = (known after apply)
      + ipv6_addresses                       = (known after apply)
      + key_name                             = "your-ssh-key-name"
      + monitoring                           = (known after apply)
      + outpost_arn                          = (known after apply)
      + password_data                        = (known after apply)
      + placement_group                      = (known after apply)
      + placement_partition_number           = (known after apply)
      + primary_network_interface_id         = (known after apply)
      + private_dns                          = (known after apply)
      + private_ip                           = (known after apply)
      + public_dns                           = (known after apply)
      + public_ip                            = (known after apply)
      + secondary_private_ips                = (known after apply)
```

```
      + security_groups                      = (known after apply)
      + source_dest_check                    = true
      + spot_instance_request_id             = (known after apply)
      + subnet_id                            = (known after apply)
      + tags                                 = {
          + "Name" = "MLOps-Lab-Instance"
        }
      + tags_all                             = {
          + "Name" = "MLOps-Lab-Instance"
        }
      + tenancy                              = (known after apply)
      + user_data                            = (known after apply)
      + user_data_base64                     = (known after apply)
      + user_data_replace_on_change          = false
      + vpc_security_group_ids               = (known after apply)

      + capacity_reservation_specification (known after apply)

      + cpu_options (known after apply)

      + ebs_block_device (known after apply)

      + enclave_options (known after apply)

      + ephemeral_block_device (known after apply)

      + instance_market_options (known after apply)

      + maintenance_options (known after apply)

      + metadata_options (known after apply)

      + network_interface (known after apply)

      + private_dns_name_options (known after apply)

      + root_block_device (known after apply)
    }

Plan: 1 to add, 0 to change, 0 to destroy.

Changes to Outputs:
  + instance_id = (known after apply)
  + public_ip   = (known after apply)

Note: You didn't use the -out option to save this plan, so Terraform can't guarantee to take exactly these actions if you run "terraform apply" now.
```

terraform plan

Conclusion

This lab demonstrated the power and simplicity of using Terraform for infrastructure provisioning. Instead of manually clicking through AWS's UI, the infrastructure was defined as code — making it reproducible, auditable, and scalable.

We explored essential components like providers, variables, and Terraform state, which are critical for real-world DevOps and MLOps automation workflows. Understanding and applying these principles lays the groundwork for managing complex ML systems across environments with reliability and control.

Experiment 2

Aim

Secrets Management with HashiCorp Vault: Securely store and access secrets like API keys and credentials using Vault and integrate it into your deployment pipeline.

Theory

Secrets management refers to the process of securely storing, accessing, and controlling the distribution of sensitive data (e.g., passwords, API keys, certificates). It's a crucial aspect of maintaining security and ensuring that only authorized users or systems have access to this data. Secrets management is especially important in modern application deployments, where automated CI/CD pipelines, containerization, and cloud services are commonplace.

With the rise of microservices, distributed systems, and DevOps practices, traditional methods of managing secrets (e.g., hardcoding credentials in code or using environment variables) are no longer secure or scalable. In response, solutions like HashiCorp Vault have emerged to provide a centralized, secure way to manage secrets.

HashiCorp Vault is a powerful, open-source tool for managing secrets and sensitive data. It provides:

- **Centralized secrets management:** Store API keys, database credentials, SSH keys, etc.
- **Dynamic secrets:** Generate secrets dynamically, such as AWS credentials or database passwords, with a set TTL (Time-to-Live).
- **Access control:** Fine-grained access control using policies, ensuring only authorized users or applications can access specific secrets.
- **Audit logs:** Track access to secrets with detailed logs for compliance.

Vault integrates seamlessly into DevOps pipelines, enabling secure storage and retrieval of secrets during build, test, and deployment processes.

Experiment

<u>Start Vault Server</u>

vault server -dev

<u>Setup CLI</u>

export VAULT_ADDR='http://127.0.0.1:8200'
export VAULT_TOKEN='hvs.s2DHK9sFNN4Wbw0jr7snfNpK'

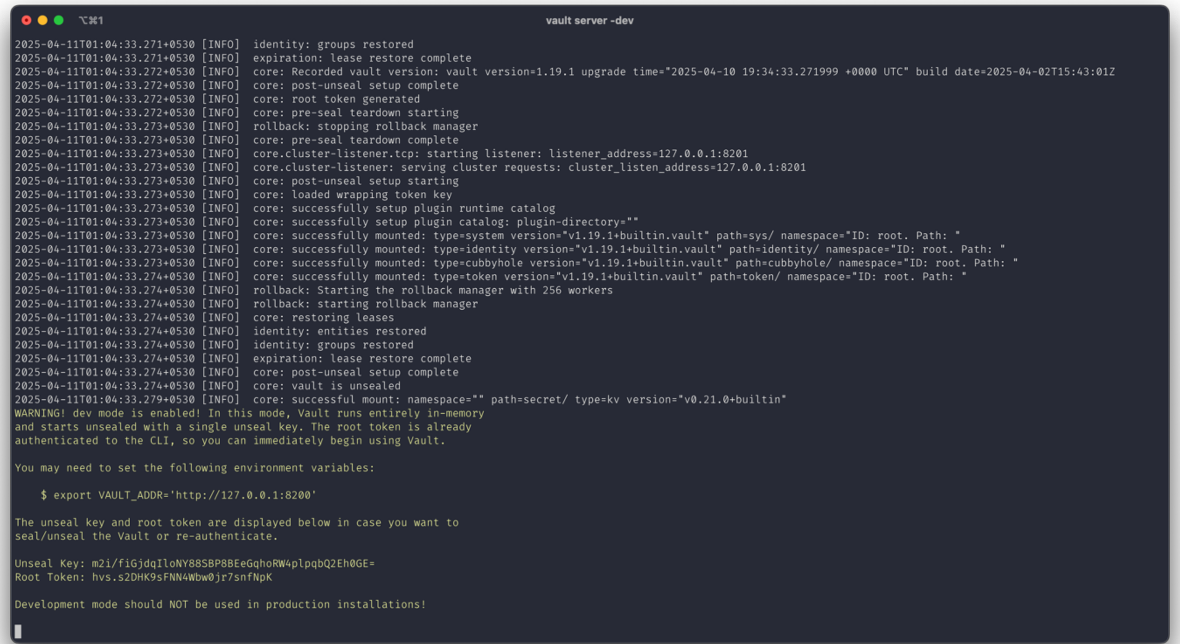
<u>Set/Get Secret</u>

vault kv put secret/myapp/api_key value="12345-abcde-67890-fghij"
vault kv get secret/myapp/api_key

Access from application

```
import os
from hvac import Client
from dotenv import load_dotenv

load_dotenv()

# Initialize the Vault client
vault_client = Client(url=os.getenv("VAULT_ADDR"), token=os.getenv("VAULT_TOKEN"))

# Fetch the secret from Vault
secret = vault_client.secrets.kv.v2.read_secret(path="/myapp/api_key")

# Extract the API key from the response
api_key = secret["data"]["data"]["value"]

# Use the API key in the application (for demonstration purposes, we'll print it)
print(f"Retrieved API Key: {api_key}")
```

Outputs



Starting vault server

```
➜  ~ vault kv put secret/myapp/api_key value="12345-abcde-67890-fghij"
===== Secret Path =====
secret/data/myapp/api_key

======= Metadata =======
Key                Value
---                -----
created_time       2025-04-10T19:37:22.106525Z
custom_metadata    <nil>
deletion_time      n/a
destroyed          false
version            1
```

Setting secret

```
➜  ~ vault kv get secret/myapp/api_key
===== Secret Path =====
secret/data/myapp/api_key

======= Metadata =======
Key                Value
---                -----
created_time       2025-04-10T19:37:22.106525Z
custom_metadata    <nil>
deletion_time      n/a
destroyed          false
version            1

==== Data ====
Key        Value
---        -----
value      12345-abcde-67890-fghij
```

Getting secret

```
(env) ➜  vault-demo git:(main) ✗ python3 app.py
Retrieved API Key: 12345-abcde-67890-fghij
```

Accessing secret from application

Conclusion

HashiCorp Vault provides a secure, scalable, and highly flexible solution for managing secrets and sensitive data. In this lab, we demonstrated how to:

- Store secrets like API keys securely in Vault.
- Integrate Vault into a deployment pipeline, ensuring secrets are accessed dynamically and securely during runtime.
- Use Vault's secrets management features, such as fine-grained access control, audit logging, and the ability to store dynamic secrets.

Experiment 3

Aim

Model Serving with FastAPI — Deploy a trained ML model using FastAPI to expose endpoints like /predict and test with real input data.

Theory

Model serving refers to the process of deploying a machine learning model into a production environment where it can receive real-time input data, process it, and return predictions or other results. After training a model, the next step is to make it accessible for use in applications, APIs, or other services.

To achieve this, you need to create a RESTful API that can accept input, pass the data to the trained model, and then return the prediction. Model serving frameworks like FastAPI are highly suited for this task because they provide easy, fast, and scalable ways to build APIs.

FastAPI is a modern, fast (high-performance), web framework for building APIs with Python. It's built on top of Starlette for the web parts and Pydantic for data validation. What makes FastAPI particularly attractive for model serving is its:

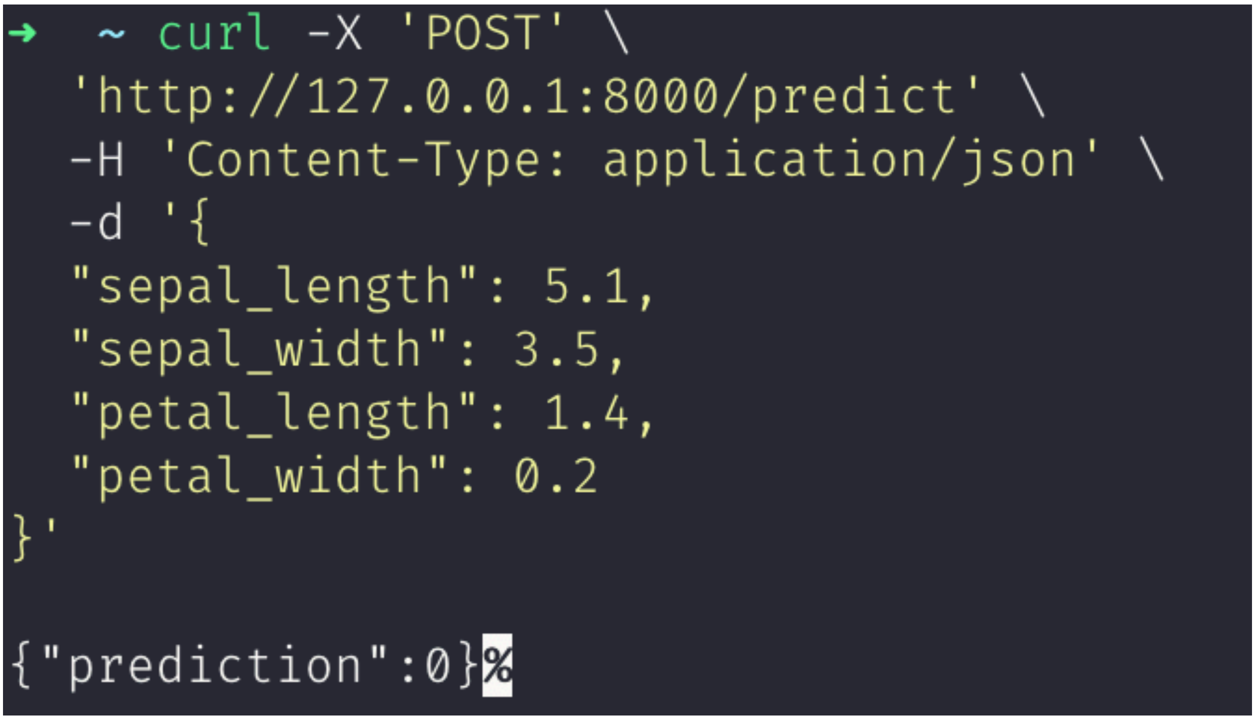
- High performance: Due to its asynchronous capabilities and reliance on Python type hints, FastAPI is one of the fastest frameworks available for building APIs.
- Easy integration: FastAPI integrates easily with machine learning frameworks (e.g., TensorFlow, PyTorch, Scikit-Learn) and tools (e.g., Docker) for deploying models.
- Automatic validation: FastAPI automatically validates incoming data using Pydantic models, which ensures that the data adheres to the expected format and type.

Experiment

```
from fastapi import FastAPI
from pydantic import BaseModel
import joblib
import numpy as np

# Load the trained model
model = joblib.load("model.pkl")

# Define FastAPI app
app = FastAPI()

# Define a Pydantic model for input validation
class PredictionInput(BaseModel):
    sepal_length: float
    sepal_width: float
    petal_length: float
    petal_width: float
```

```
# Define a prediction endpoint
@app.post("/predict")
def predict(input_data: PredictionInput):
    # Convert the input data to a numpy array (model expects numpy arrays)
    input_features = np.array([[input_data.sepal_length,
                               input_data.sepal_width,
                               input_data.petal_length,
                               input_data.petal_width]])

    # Get the model's prediction
    prediction = model.predict(input_features)

    # Return the prediction result
    return {"prediction": int(prediction[0])}
```

Outputs



Testing /predict endpoint

Conclusion

In this lab, we demonstrated how to deploy a machine learning model using FastAPI. We walked through the following steps:

- Built a FastAPI app that exposes a /predict endpoint to make predictions using the trained model.
- Tested the deployment by sending a POST request to the FastAPI server and receiving predictions.

Experiment 4

Aim

Automated ML Pipelines with Apache Airflow: Build a Directed Acyclic Graph (DAG) in Airflow to automate tasks such as data loading, preprocessing, training, and evaluation.

Theory

Machine learning projects often involve repetitive tasks—loading data, preprocessing, model training, evaluation, and storing results. Manually executing each of these steps introduces risks: inconsistencies, human errors, and inefficiency. This is where workflow orchestration tools come in.

Apache Airflow is an open-source platform to programmatically author, schedule, and monitor workflows. It uses DAGs (Directed Acyclic Graphs) to represent workflows. Each DAG consists of a set of tasks, and each task performs a discrete piece of work. These tasks are connected based on dependencies.

For ML workflows, Airflow is a game-changer. It allows:

- Modular, reusable task definitions
- Scheduled or event-triggered runs
- Monitoring and logging
- Integration with ML tools, cloud providers, and databases

Airflow also supports task retries, alerting, and dynamic pipelines—ideal for real-world ML systems.

Experiment

<u>Setup Airflow</u>

```
python -m venv airflow-venv
source airflow-venv/bin/activate
pip install apache-airflow

# Initialize Airflow
airflow db init

# Create a user
airflow users create \
  --username admin \
  --password admin \
  --firstname Admin \
  --lastname User \
  --role Admin \
  --email admin@example.com
```

```
# Start Airflow webserver and scheduler
airflow webserver -p 8080  # In one terminal
airflow scheduler          # In another terminal
```

Create Pipeline

steps.py

```python
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
import joblib

def save_iris_data():
    iris = load_iris()
    df = pd.DataFrame(iris.data, columns=iris.feature_names)
    df['target'] = iris.target
    df.to_csv('data/iris.csv', index=False)

def preprocess_data():
    df = pd.read_csv('data/iris.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)
    X_train.to_csv('data/X_train.csv', index=False)
    X_test.to_csv('data/X_test.csv', index=False)
    y_train.to_csv('data/y_train.csv', index=False)
    y_test.to_csv('data/y_test.csv', index=False)

def train_model():
    X_train = pd.read_csv('data/X_train.csv')
    y_train = pd.read_csv('data/y_train.csv').values.ravel()
    model = RandomForestClassifier(n_estimators=100)
    model.fit(X_train, y_train)
    joblib.dump(model, 'data/model.pkl')

def evaluate_model():
    X_test = pd.read_csv('data/X_test.csv')
    y_test = pd.read_csv('data/y_test.csv').values.ravel()
    model = joblib.load('data/model.pkl')
    preds = model.predict(X_test)
    acc = accuracy_score(y_test, preds)
    with open("data/accuracy.txt", "w") as f:
        f.write(f"Accuracy: {acc}")
```

pipeline.py

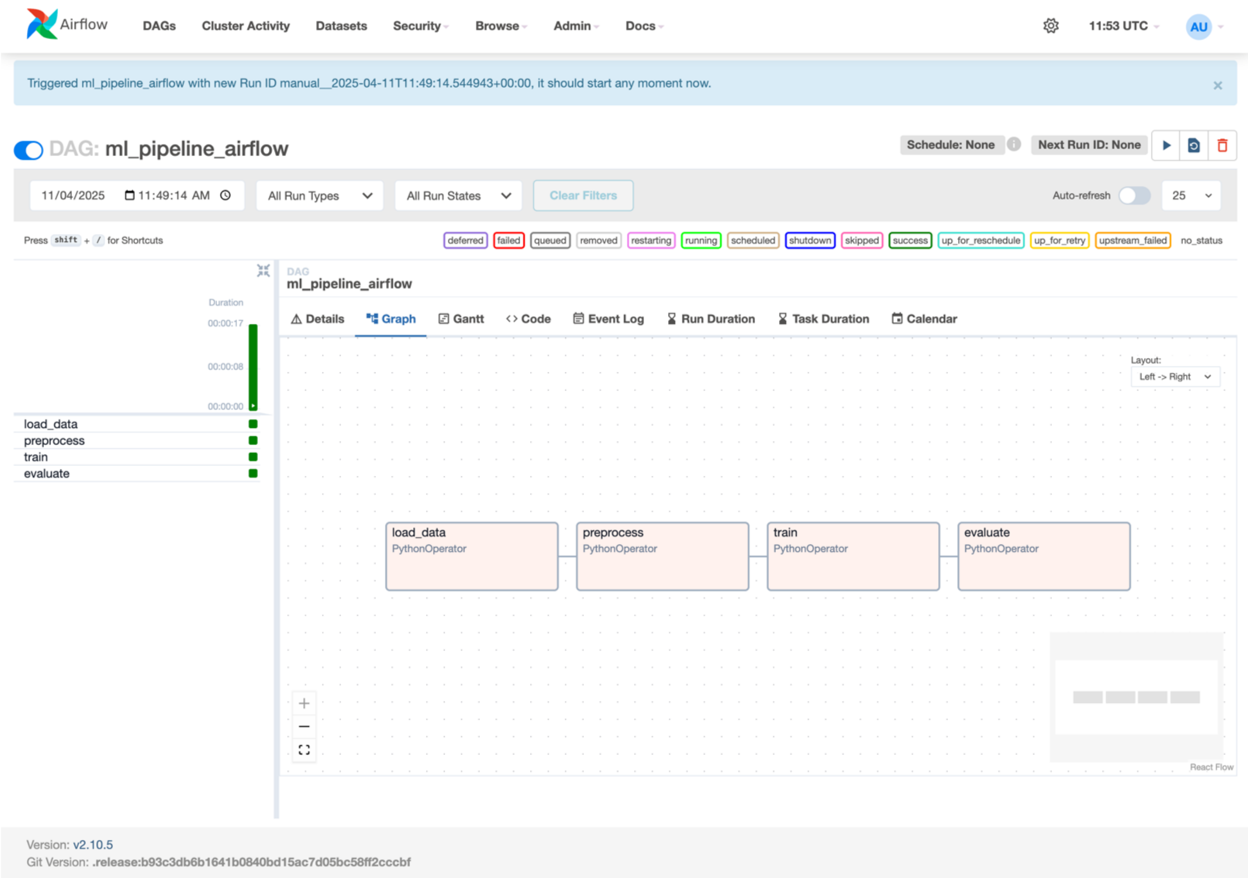
```
from airflow import DAG
from airflow.operators.python import PythonOperator
from datetime import datetime
import steps

default_args = {
    'owner': 'airflow',
    'start_date': datetime(2024, 1, 1),
    'retries': 1,
}

with DAG(
    dag_id='ml_pipeline_airflow',
    default_args=default_args,
    schedule_interval=None,
    catchup=False,
    tags=["mlops", "demo"]
) as dag:

    load_data = PythonOperator(
        task_id='load_data',
        python_callable=steps.save_iris_data,
    )

    preprocess = PythonOperator(
        task_id='preprocess',
        python_callable=steps.preprocess_data,
    )

    train = PythonOperator(
        task_id='train',
        python_callable=steps.train_model,
    )

    evaluate = PythonOperator(
        task_id='evaluate',
        python_callable=steps.evaluate_model,
    )

    load_data >> preprocess >> train >> evaluate
```

Copy Pipeline

```
mkdir -p ~/airflow/dags
cp steps.py ~/airflow/dags/
```

cp pipeline.py ~/airflow/dags/

Outputs



Conclusion

This experiment demonstrates the power of Apache Airflow in orchestrating ML pipelines. By modularizing tasks like data loading, preprocessing, training, and evaluation, we gained:

- Reusability: Each task is reusable and independently manageable.
- Traceability: Logs and execution states are easily monitored.
- Scalability: Future tasks (e.g., model registration, deployment) can be seamlessly added.
- Automation: Pipelines can run on a schedule or based on events, eliminating manual triggers.

For ML teams, especially in production environments, using a tool like Airflow ensures that workflows are reliable, reproducible, and maintainable. It brings engineering discipline to data science.

Experiment 5

Aim

Hyperparameter Tuning with Optuna – Implement Optuna for automated hyperparameter optimization and integrate it with MLflow for experiment logging.

Theory

Hyperparameter tuning is the process of finding the best combination of hyperparameters (such as learning rate, batch size, number of layers, etc.) for a machine learning model. The goal is to improve model performance by identifying optimal values for these hyperparameters.

Hyperparameters are parameters that are not learned from the data but are set before training, such as the learning rate in a neural network or the number of trees in a random forest. Selecting the right values for these hyperparameters can significantly affect a model's accuracy, speed, and overall performance.

Optuna is an open-source hyperparameter optimization framework designed to automate the process of finding optimal hyperparameters for machine learning models. It supports different optimization algorithms, including random search, grid search, and a more sophisticated approach called Tree-structured Parzen Estimators (TPE), which helps find better results in fewer trials.

Optuna is particularly efficient for optimizing complex models or models that require expensive computational resources to train, such as deep neural networks.

MLflow is an open-source platform designed to manage the entire machine learning lifecycle, including experimentation, reproducibility, and deployment. It provides tools for:

- Tracking experiments: Log parameters, metrics, and outputs of model training.
- Managing models: Store and serve models.
- Reproducibility: Ensures that experiments can be reproduced and results can be traced.

When combined with Optuna, MLflow can log the hyperparameter configurations and the results of each optimization trial, making it easier to track the effectiveness of different configurations.

Experiment

Setup MLflow

```
pip install mlflow
mlflow ui
```

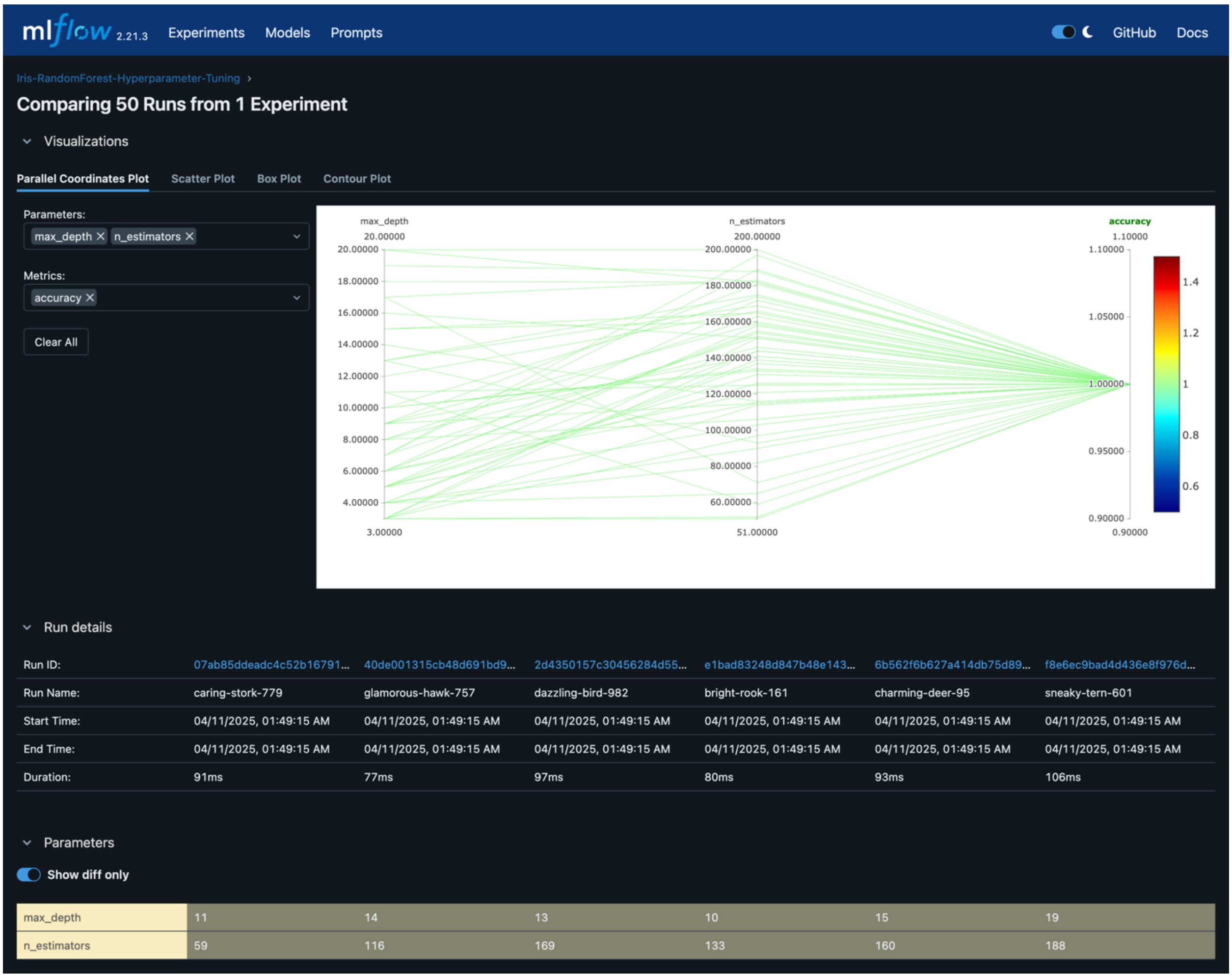
ML Experiment

```
from sklearn.datasets import load_iris
import pandas as pd
```

```
from sklearn.model_selection import train_test_split

import optuna
import mlflow
import mlflow.sklearn
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score

# Load the Iris dataset
iris = load_iris()
X = pd.DataFrame(iris.data, columns=iris.feature_names)
y = iris.target

# Split data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Define the objective function for hyperparameter optimization
def objective(trial):
    # Start a new MLflow run for each trial
    with mlflow.start_run():
        # Hyperparameter search space
        n_estimators = trial.suggest_int("n_estimators", 50, 200)  # Number of trees
        max_depth = trial.suggest_int("max_depth", 3, 20)  # Maximum depth of trees

        # Initialize the RandomForest model with selected hyperparameters
        model = RandomForestClassifier(n_estimators=n_estimators, max_depth=max_depth)

        # Train the model
        model.fit(X_train, y_train)

        # Predict and calculate accuracy
        y_pred = model.predict(X_test)
        accuracy = accuracy_score(y_test, y_pred)

        # Log the hyperparameters and the accuracy with MLflow
        mlflow.log_param("n_estimators", n_estimators)
        mlflow.log_param("max_depth", max_depth)
        mlflow.log_metric("accuracy", accuracy)

        return accuracy

if __name__ == "__main__":
    # Start MLflow experiment
    mlflow.set_tracking_uri(uri="http://127.0.0.1:5000/")
    mlflow.set_experiment("Iris-RandomForest-Hyperparameter-Tuning")  # Set an experiment
name if necessary
```

```
# Create an Optuna study to optimize the objective function
study = optuna.create_study(direction="maximize")  # We want to maximize accuracy
study.optimize(objective, n_trials=50)  # Run 50 trials

# Best trial information
print(f"Best Trial: {study.best_trial.params}")
print(f"Best Accuracy: {study.best_value}")
```

Outputs



Hyperparameter tuning visualization in Mlflow

Conclusion

In this experiment, we demonstrated how to:

- Use Optuna to automate the hyperparameter optimization process for a machine learning model.
- Integrate Optuna with MLflow to track experiments, including hyperparameter values and performance metrics, ensuring reproducibility and ease of comparison between different trials.
- Leverage the power of automated hyperparameter search to find the best configuration for a Random Forest classifier using the Iris dataset.

By combining Optuna and MLflow, we gain significant benefits in terms of both optimization and experiment tracking. Optuna helps automate the search for optimal hyperparameters, making the process faster and more efficient. Meanwhile, MLflow provides a centralized platform for managing and tracking experiments, which is invaluable when working on more complex ML projects.

Experiment 6

Aim

Data Versioning with DVC: Use DVC to track versions of datasets and models, and connect it with Git for full ML lifecycle version control.

Theory

DVC (Data Version Control) is an open-source tool designed to bring the benefits of version control (like Git) to data science and machine learning workflows. While Git is great for tracking code, it isn't optimized for large files like datasets, model binaries, or artifacts. This is where DVC steps in.

With DVC, you can:

- Track large files like datasets and models.
- Reproduce entire ML pipelines.
- Collaborate easily across teams.
- Integrate with Git for full lifecycle control (code + data + experiments).

It uses .dvc files and a remote storage system (like a local folder, S3, GCS, etc.) to manage actual data, while keeping metadata in Git.

In a typical ML project, data evolves just as much as code. You might start with a sample dataset, then receive an updated one, or clean your data differently across iterations. Similarly, model files change based on training data, hyperparameters, or model architecture. If you can't track and reproduce these changes, you're left with guesswork when bugs or inconsistencies arise.

That's why DVC is crucial: it helps ensure that your code, data, and models are always in sync—and reproducible.

Experiment

Initialize DVC

```
git init
dvc init
```

Adding dataset to DVC

```
dvc add data/iris.csv
git add data.dvc .gitignore
git commit -m "Add Iris dataset with DVC"
```

DVC Pipeline

```
dvc stage add -n train_model \
  -d train.py -d data/iris.csv -d params.yaml \
```

-o models/model.pkl \
python train.py

git add dvc.yaml dvc.lock train.py params.yaml
git commit -m "Add training pipeline"

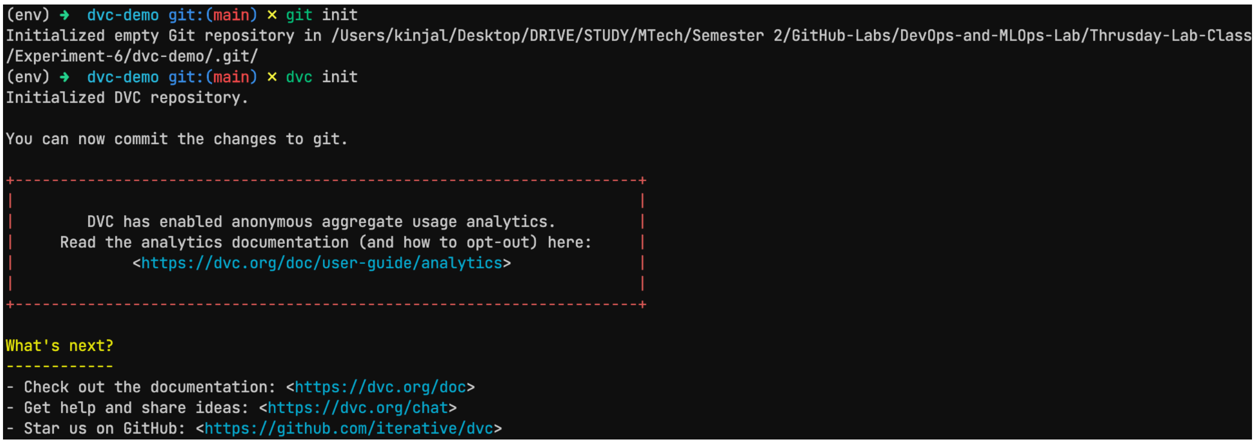
Running DVC Pipeline by changing hyperparameters

dvc repro

change hyperparmeters in params.yaml

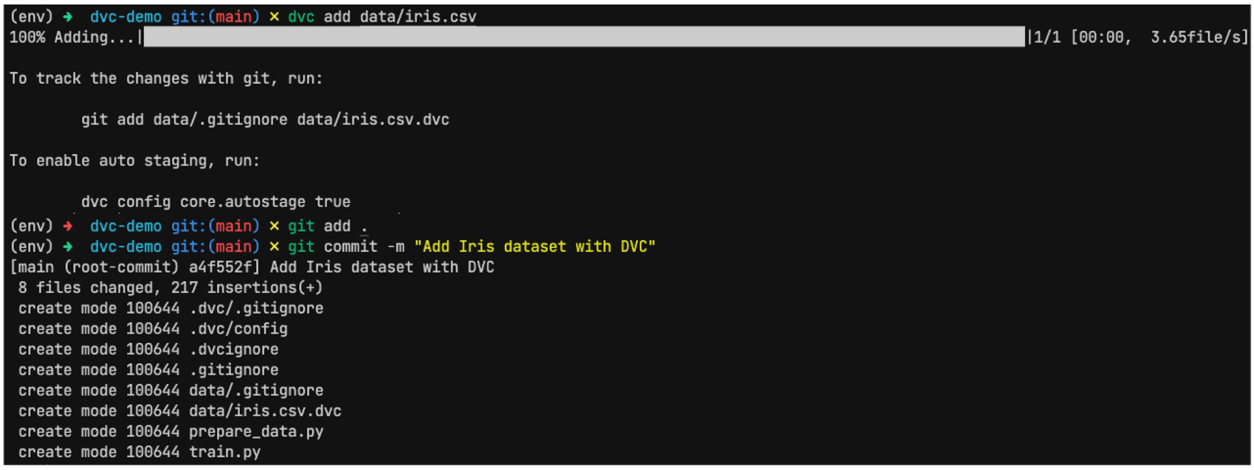
dvc repro
git add .
git commit -m "Increase number of trees to 200"

Outputs



```
(env) ➜  dvc-demo git:(main) ✗ git init
Initialized empty Git repository in /Users/kinjal/Desktop/DRIVE/STUDY/MTech/Semester 2/GitHub-Labs/DevOps-and-MLOps-Lab/Thrusday-Lab-Class
/Experiment-6/dvc-demo/.git/
(env) ➜  dvc-demo git:(main) ✗ dvc init
Initialized DVC repository.

You can now commit the changes to git.

+---------------------------------------------------------------------+
|                                                                     |
|        DVC has enabled anonymous aggregate usage analytics.         |
|     Read the analytics documentation (and how to opt-out) here:     |
|             <https://dvc.org/doc/user-guide/analytics>              |
|                                                                     |
+---------------------------------------------------------------------+

What's next?
------------
- Check out the documentation: <https://dvc.org/doc>
- Get help and share ideas: <https://dvc.org/chat>
- Star us on GitHub: <https://github.com/iterative/dvc>
```

Initialize DVC



```
(env) ➜  dvc-demo git:(main) ✗ dvc add data/iris.csv
100% Adding...|████████████████████████████████████████|1/1 [00:00,  3.65file/s]

To track the changes with git, run:

        git add data/.gitignore data/iris.csv.dvc

To enable auto staging, run:

        dvc config core.autostage true
(env) ➜  dvc-demo git:(main) ✗ git add .
(env) ➜  dvc-demo git:(main) ✗ git commit -m "Add Iris dataset with DVC"
[main (root-commit) a4f552f] Add Iris dataset with DVC
 8 files changed, 217 insertions(+)
 create mode 100644 .dvc/.gitignore
 create mode 100644 .dvc/config
 create mode 100644 .dvcignore
 create mode 100644 .gitignore
 create mode 100644 data/.gitignore
 create mode 100644 data/iris.csv.dvc
 create mode 100644 prepare_data.py
 create mode 100644 train.py
```

Adding dataset to DVC

```
(env) ➜  dvc-demo git:(main) ✗ dvc stage add -n train_model \
  -d train.py -d data/iris.csv -d params.yaml \
  -o models/model.pkl \
  python train.py
Added stage 'train_model' in 'dvc.yaml'

To track the changes with git, run:

        git add dvc.yaml models/.gitignore

To enable auto staging, run:

        dvc config core.autostage true
(env) ➜  dvc-demo git:(main) ✗
(env) ➜  dvc-demo git:(main) ✗ git add dvc.yaml models/.gitignore
(env) ➜  dvc-demo git:(main) ✗ git commit -m "Add training pipeline"
[main 400f9e9] Add training pipeline
 2 files changed, 10 insertions(+)
 create mode 100644 dvc.yaml
 create mode 100644 models/.gitignore
```

DVC Pipeline

```
(env) ➜  dvc-demo git:(main) ✗ dvc repro
'data/iris.csv.dvc' didn't change, skipping
Running stage 'train_model':
> python train.py
Generating lock file 'dvc.lock'
Updating lock file 'dvc.lock'

To track the changes with git, run:

        git add dvc.lock

To enable auto staging, run:

        dvc config core.autostage true
Use `dvc push` to send your updates to remote storage.
(env) ➜  dvc-demo git:(main) ✗
(env) ➜  dvc-demo git:(main) ✗ dvc repro
```

```
'data/iris.csv.dvc' didn't change, skipping
Running stage 'train_model':
> python train.py
Updating lock file 'dvc.lock'

To track the changes with git, run:

        git add dvc.lock

To enable auto staging, run:

        dvc config core.autostage true
Use `dvc push` to send your updates to remote storage.
(env) ➜  dvc-demo git:(main) ✗
(env) ➜  dvc-demo git:(main) ✗ git add .
(env) ➜  dvc-demo git:(main) ✗ git commit -m "Increase number of trees to 200"
[main e43a8e6] Increase number of trees to 200
 2 files changed, 24 insertions(+)
 create mode 100644 dvc.lock
 create mode 100644 params.yaml
```

Running DVC Pipeline by changing hyperparameters

Conclusion

In this experiment, we explored how DVC integrates with Git to provide full lifecycle management of ML projects. By versioning the dataset and model artifacts, and defining reproducible pipelines, DVC makes your ML workflow:

- Reproducible: Any experiment can be reproduced exactly.
- Trackable: Data and model changes are logged alongside code.
- Collaborative: Team members can pull the latest data and run the pipeline with confidence.
- Maintainable: Pipelines are modular, auditable, and easy to extend.

As machine learning systems grow more complex, tools like DVC become essential for teams to stay productive, agile, and aligned.

Experiment 7

Aim

Basic Monitoring with Prometheus and Grafana: Set up Prometheus to monitor system resources and visualize the metrics using Grafana dashboards.

Theory

In both DevOps and MLOps, observability plays a crucial role in ensuring system reliability, performance, and transparency. As systems become increasingly complex—spanning microservices, containers, and ML pipelines—monitoring becomes the foundation for proactive debugging, incident response, and performance tuning.

Prometheus is an open-source monitoring system built for reliability and scalability. Originally developed at SoundCloud, it is now a part of the Cloud Native Computing Foundation (CNCF). Prometheus excels at time-series data collection, meaning it scrapes and stores metrics over time, allowing us to observe trends, set alerts, and investigate issues.

Key features:

- Pull-based model via HTTP (scrapes targets)
- Powerful PromQL (Prometheus Query Language)
- Lightweight and simple to deploy
- Supports exporters for different services (node, Docker, Kubernetes, etc.)

Grafana is a popular open-source visualization tool that connects to Prometheus (and other data sources) to provide beautiful and customizable dashboards. It brings data to life, making it easier to detect patterns and understand performance metrics at a glance.

Prometheus and Grafana together form a robust observability stack.

Experiment

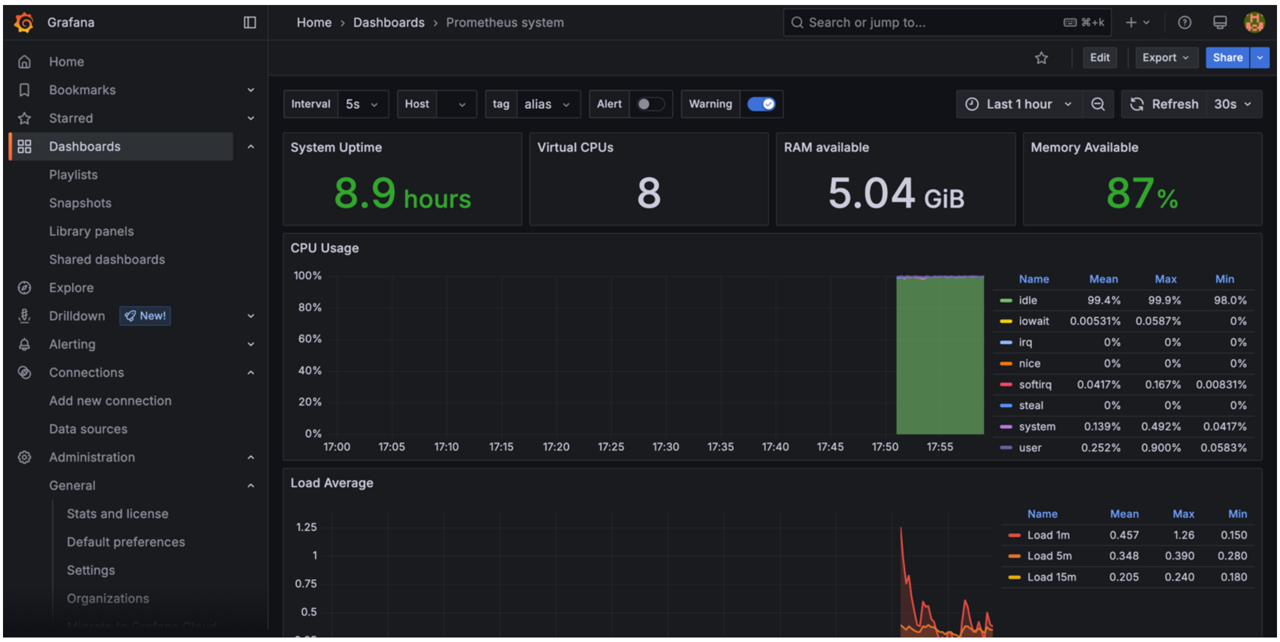
prometheus.yml

```
# monitoring-stack/prometheus/prometheus.yml
global:
  scrape_interval: 15s

scrape_configs:
  - job_name: "prometheus"
    static_configs:
      - targets: ["localhost:9090"]

  - job_name: "node_exporter"
    static_configs:
      - targets: ["node_exporter:9100"]
```

docker-compose.yaml

```
# monitoring-stack/docker-compose.yml
version: "3.8"

services:
  prometheus:
    image: prom/prometheus
    container_name: prometheus
    volumes:
      - ./prometheus/prometheus.yml:/etc/prometheus/prometheus.yml
    ports:
      - "9090:9090"

  node_exporter:
    image: prom/node-exporter
    container_name: node_exporter
    ports:
      - "9100:9100"

  grafana:
    image: grafana/grafana
    container_name: grafana
    ports:
      - "3000:3000"
```

Outputs



Grafana dashboard

Conclusion

This experiment demonstrated how to set up basic monitoring using Prometheus and Grafana, two essential tools in the DevOps and MLOps ecosystem. With this setup:

- You gain real-time visibility into system performance.
- Potential issues can be detected before they become critical.
- Trends over time help inform scaling and optimization decisions.

By integrating such a stack early in your infrastructure or ML pipeline, you ensure better reliability, transparency, and peace of mind for developers and operators alike.

Experiment 8

Aim

Deploy ML Model using Jenkins Pipeline: Create a Jenkins pipeline that trains, tests, and deploys a machine learning model.

Theory

Jenkins is a well-established open-source automation server widely used for continuous integration and delivery (CI/CD). While it's traditionally used in software development workflows, Jenkins is equally powerful in MLOps pipelines, where machine learning models need to be trained, tested, and deployed in an automated and repeatable manner.

- Integrating ML tasks into a Jenkins pipeline ensures that:
- Models are always trained and tested in a clean, repeatable environment.
- Deployment is automated, reducing manual errors.
- Teams can track changes and experiment outcomes via version control and logs.

A Jenkins pipeline is a script (usually written in Groovy) that defines the stages of your process, such as:

- Build: Clone the code, install dependencies
- Train: Run the training script
- Test: Evaluate the model's performance
- Deploy: Push model to production (or simulate it locally)

Experiment

Training

```
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
import joblib
from sklearn.datasets import load_iris
import os

iris = load_iris()
X, y = iris.data, iris.target
model = RandomForestClassifier(n_estimators=100)
model.fit(X, y)
os.makedirs('model', exist_ok=True)
joblib.dump(model, 'model/model.pkl')
print("Model trained and saved.")
```

Testing

```
import joblib
from sklearn.datasets import load_iris
```

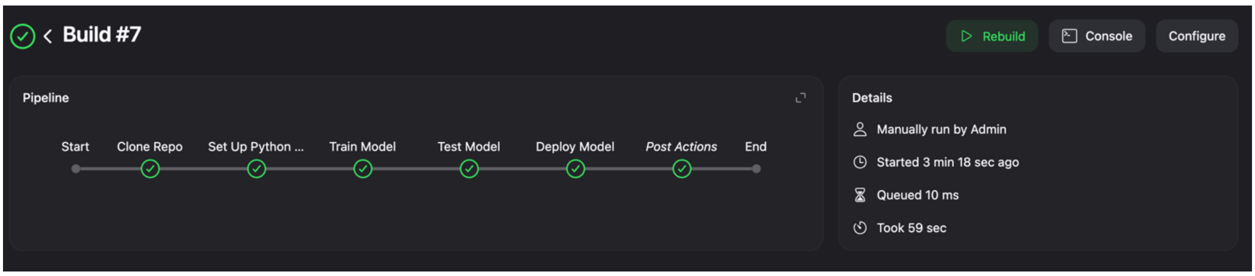
```
from sklearn.metrics import accuracy_score

model = joblib.load("model/model.pkl")
iris = load_iris()
X, y = iris.data, iris.target
preds = model.predict(X)
print(f"Accuracy: {accuracy_score(y, preds):.2f}")
```

Pipeline

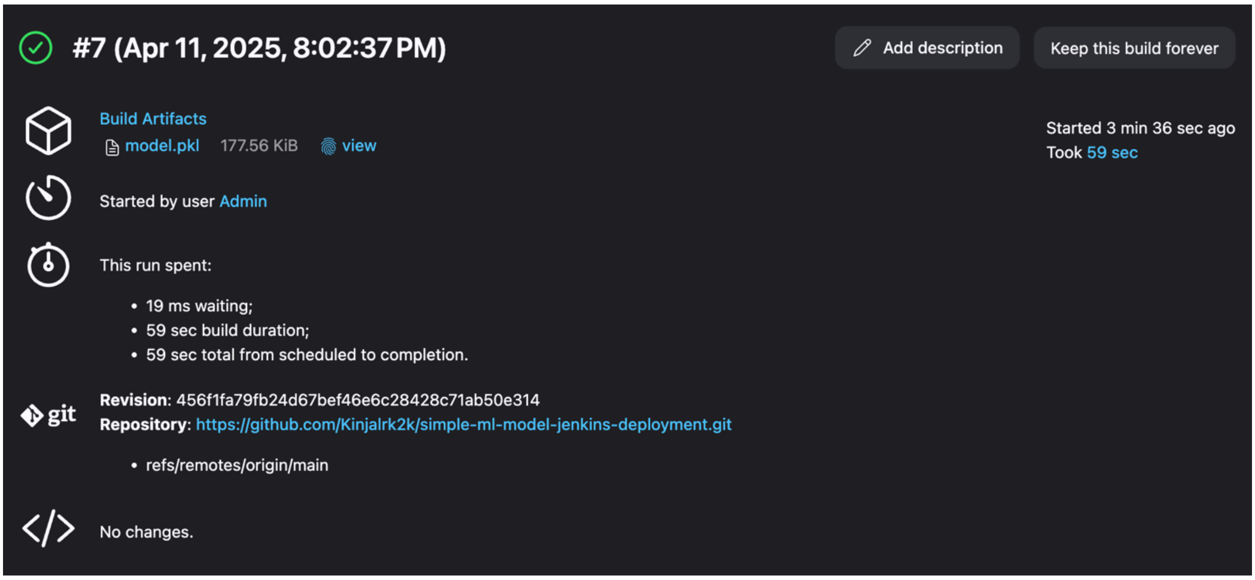
```
pipeline {
    agent any

    environment {
        VENV = 'venv'
        PYTHON = 'python3'
    }

    stages {
        stage('Clone Repo') {
            steps {
                git branch: 'main', url: 'https://github.com/Kinjalrk2k/simple-ml-model-jenkins-
deployment.git'
            }
        }

        stage('Set Up Python Env') {
            steps {
                sh '''
                    $PYTHON -m venv $VENV
                    . $VENV/bin/activate
                    pip install --upgrade pip
                    pip install -r requirements.txt
                '''
            }
        }

        stage('Train Model') {
            steps {
                sh '''
                    . $VENV/bin/activate
                    python train.py
                '''
            }
        }

        stage('Test Model') {
```

```
            steps {
                sh '''
                    . $VENV/bin/activate
                    python test.py
                '''
            }
        }

        stage('Deploy Model') {
             steps {
                sh '''
                    echo "Model is already saved in model/model.pkl"
                '''
                archiveArtifacts artifacts: 'model/model.pkl', fingerprint: true
            }
        }
    }

    post {
        always {
            echo '🧹 Cleaning up workspace...'
            cleanWs()
        }
    }
}
```

Outputs



Pipeline Overview



Post build artifacts

Conclusion

In this lab, we automated a simple end-to-end ML workflow using Jenkins Pipelines. This included cloning the project, training a model, testing it, and deploying the final artifact—all from a single script.

- By introducing Jenkins to your ML stack, you:
- Eliminate manual and error-prone steps
- Ensure reproducibility and consistency
- Create a foundation for scalable and collaborative model delivery

Experiment 9

Aim

A/B Testing of ML Models: Deploy two versions of a model and perform A/B testing using load balancers or routing in Kubernetes.

Theory

A/B testing is a statistical approach used to compare two variants (A and B) of a product or service to determine which performs better. In the context of ML models, A/B testing helps validate whether a newer version of a model (Model B) performs better than the existing one (Model A) in real-world scenarios.

This technique is especially valuable in production environments to:

- Evaluate new models on live traffic without fully replacing the old one.
- Collect real user feedback on performance.
- Make data-driven decisions on promoting models.

In an MLOps pipeline, A/B testing typically involves:

- Deploying two model versions simultaneously.
- Splitting traffic between them using routing rules (e.g., 80/20 or 50/50).
- Monitoring performance metrics and analyzing results.

Kubernetes makes A/B testing easier through:

- Multiple Deployments for different model versions.
- Services for abstracting access to Pods.
- Ingress controllers (like NGINX) or Service Meshes (like Istio) to split traffic intelligently.

Experiment

Deployment A

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: iris-model-a
spec:
  replicas: 1
  selector:
    matchLabels:
      app: iris
      version: a
  template:
    metadata:
```

```
      labels:
        app: iris
        version: a
    spec:
      containers:
        - name: iris
          image: model-a:latest
          imagePullPolicy: Never
          ports:
            - containerPort: 9000
```

Service A

```
apiVersion: v1
kind: Service
metadata:
  name: iris-a-service
spec:
  selector:
    app: iris
    version: a
  ports:
    - port: 80
      targetPort: 9000
```

Deployment B

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: iris-model-b
spec:
  replicas: 1
  selector:
    matchLabels:
      app: iris
      version: b
  template:
    metadata:
      labels:
        app: iris
        version: b
    spec:
      containers:
        - name: iris
          image: model-b:latest
          imagePullPolicy: Never
```

```
        ports:
          - containerPort: 9000
```

Service B

```
apiVersion: v1
kind: Service
metadata:
  name: iris-b-service
spec:
  selector:
    app: iris
    version: b
  ports:
    - port: 80
      targetPort: 9000
```

Main Ingress

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: iris-ingress
  annotations:
    # nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
    - host: iris.local
      http:
        paths:
          - path: /predict
            pathType: Prefix
            backend:
              service:
                name: iris-a-service
                port:
                  number: 80
```

Canary Ingress

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: iris-canary
  annotations:
    nginx.ingress.kubernetes.io/canary: "true"
    nginx.ingress.kubernetes.io/canary-weight: "50" # split traffic
```

```yaml
spec:
  rules:
    - host: iris.local
      http:
        paths:
          - path: /predict
            pathType: Prefix
            backend:
              service:
                name: iris-b-service
                port:
                  number: 80
```

Outputs

```
➜  ~ curl -X POST http://iris.local:8080/predict \
  -H "Content-Type: application/json" \
  -d '{"features": [5.1, 3.5, 1.4, 0.2]}'

{"prediction":0,"version":1}
➜  ~ curl -X POST http://iris.local:8080/predict \
  -H "Content-Type: application/json" \
  -d '{"features": [5.1, 3.5, 1.4, 0.2]}'

{"prediction":0,"version":2}
➜  ~ curl -X POST http://iris.local:8080/predict \
  -H "Content-Type: application/json" \
  -d '{"features": [5.1, 3.5, 1.4, 0.2]}'

{"prediction":0,"version":1}
➜  ~ curl -X POST http://iris.local:8080/predict \
  -H "Content-Type: application/json" \
  -d '{"features": [5.1, 3.5, 1.4, 0.2]}'
```

A/B Testing through API

Conclusion

In this lab, we successfully demonstrated how to deploy two versions of an ML model and conduct A/B testing using Kubernetes and traffic splitting. This is a powerful pattern for:

- Validating new models in production
- Reducing risk in rollouts
- Gaining user-driven feedback before fully replacing older versions

Experiment 10

Aim

Model Drift Detection: Use tools like Evidently or custom scripts to monitor and detect data/model drift over time.

Theory

In the real world, machine learning models are not "train once, run forever." They depend heavily on the data distribution they were trained on. Over time, changes in data patterns can degrade model performance—a phenomenon known as model drift or data drift.

Data Drift: The statistical distribution of input data changes over time, even if the output (label) remains the same. Example: A spam detection model starts receiving emails with different vocabulary or structure compared to training data.

Concept Drift: The relationship between inputs and outputs changes over time. Example: Customer behavior shifts due to market changes—features that once predicted churn may no longer work.

If drift goes unnoticed:

- Model predictions can become inaccurate or even harmful.
- Business decisions based on the model become unreliable.
- Trust in AI/ML systems erodes.

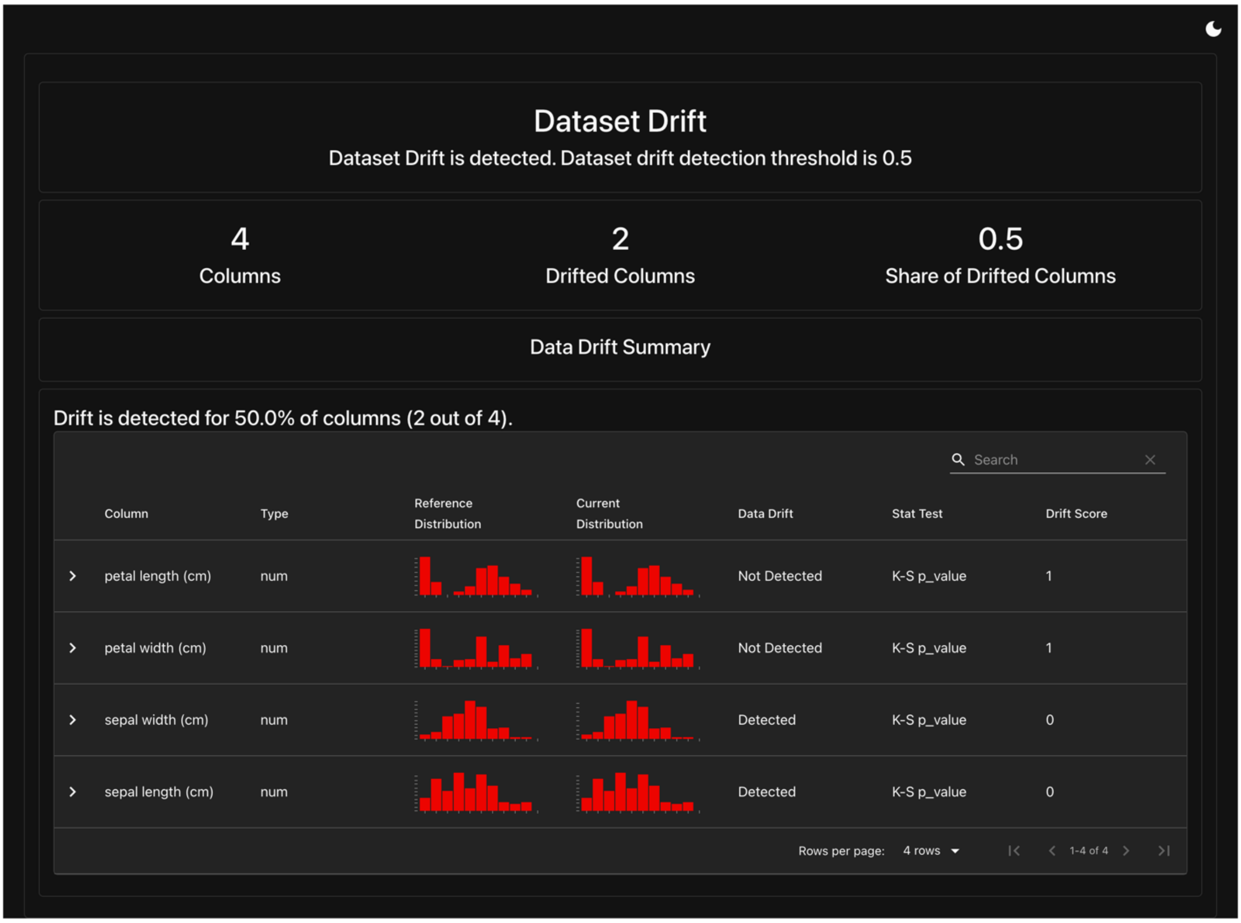
Monitoring drift is essential for ML model reliability and accountability.

Experiment

```
import pandas as pd
from sklearn.datasets import load_iris
from evidently import Report
from evidently.presets import DataDriftPreset
import matplotlib.pyplot as plt

# Load base dataset
iris = load_iris()
df_ref = pd.DataFrame(iris.data, columns=iris.feature_names)

# Simulate drift by changing feature distributions
df_drift = df_ref.copy()
df_drift["sepal length (cm)"] += 1.5  # Simulate shift
df_drift["sepal width (cm)"] *= 1.2  # Simulate scale change

# Create Evidently report
report = Report(metrics=[DataDriftPreset()])
eval = report.run(reference_data=df_ref, current_data=df_drift)
```

```
# Save as HTML report
eval.save_html("iris_drift_report.html")
print("Drift report saved as iris_drift_report.html")
```

Output



Conclusion

In this lab, we explored model drift detection using Evidently, a powerful tool for ML monitoring. We simulated data drift by altering distributions in a well-known dataset, and generated a visual, data-driven report to highlight the impact.

Detecting drift is not just a technical best practice—it's a critical component of maintaining trustworthy and effective machine learning systems. With regular monitoring in place, teams can proactively retrain or recalibrate models, ensuring sustained performance even as the world around them changes.