AI & MLOPS Projects Part-2

11. AI for Infrastructure & Network Monitoring

Project 5. AI-Powered Automated Remediation of Network Failures: AI-driven automation of recovery actions when network issues are detected, reducing downtime.

---

12. AI for ChatOps & Automated Support

Project 1. AI-Driven Automated Ticketing System: Integrating AI with ChatOps tools (like Slack/MS Teams) to automatically create and categorize tickets based on incident discussions.

Project 2. Smart ChatOps Assistance for Infrastructure Troubleshooting: AI-powered assistant in chat systems like Slack that suggests infrastructure fixes in real time based on the conversation and context.

Project 3. Chatbot for Continuous Deployment Assistance: A ChatOps bot powered by AI that helps DevOps teams by automatically updating the status of deployments and helping with rollback decisions.

Project 4. AI Chatbots for Security Incident Response: AI-powered bots integrated into ChatOps to respond to security incidents and suggest next steps or remediation actions.

Project 5. Automated Root Cause Analysis via ChatOps: Using AI to correlate logs and metrics automatically and suggest a root cause in a ChatOps platform when an issue is reported.

---

13. AI for Patch Management & Security

Project 1. AI-Powered Vulnerability Risk Scoring: Implementing AI to assess the risk of security vulnerabilities in systems based on potential impact, and prioritizing patches accordingly.

Project 2. Predictive Patch Testing with AI: AI-driven system to predict the most effective patching sequence for systems to minimize downtime and risks.

Project 3. AI-Based Security Policy Violations Detection: Using machine learning to continuously monitor and detect policy violations in infrastructure, including IAM roles and network security configurations.

Project 4. Automated Patch Scheduling with AI: AI model to suggest and automatically schedule patch deployment windows based on historical data and risk levels.

Project 5. Security Misconfiguration Detection in Infrastructure-as-Code: AI-based system that scans Terraform, Ansible, or other IAC configurations for security misconfigurations before deployment.

MLOps Projects

1. Model Deployment and Management

Project 1. End-to-End ML Model Deployment on Kubernetes: Automate ML model training, testing, and deployment using Kubeflow.

Project 2. CI/CD for ML Models with GitHub Actions & Docker: Build a pipeline that automates model versioning, testing, and deployment.

Project 3. Serverless ML Model Deployment with AWS Lambda & S3: Automate model deployment using a serverless framework.

Project 4. Multi-Cloud Model Deployment & Monitoring: Deploy models across AWS, GCP, and Azure with centralized monitoring.

Project 5. ML Model Canary Deployment with Kubernetes & Istio: Deploy new ML models in production using progressive rollout strategies.

Project 6. Automated ML Model Deployment on Multi-Cloud: Implement a system that deploys models across AWS, GCP, and Azure dynamically.

Project 7. CI/CD Pipeline for ML with Feature Drift Detection: Implement an automated system that detects feature drift and retrains models accordingly.

2. Model Monitoring and Optimization

Project 1. Drift Detection in ML Models: Implement a system that monitors model performance and triggers retraining if accuracy drops.

Project 2. Automated Model Retraining in Production: Use Apache Airflow to schedule model retraining based on new data.

Project 3. AI-Based Model Staleness Detection: Monitor ML models for concept drift and trigger automatic retraining.

Project 4. AutoML Pipeline for Hyperparameter Tuning: Build an automated training pipeline that finds the best ML model configuration.

Project 5. Smart Hyperparameter Tuning with Reinforcement Learning: Use RL to optimize ML model parameters dynamically.

Project 6. AutoML Pipeline for Continuous Model Optimization: Automate the process of selecting the best ML models.

---

3. Model Explainability and Fairness

Project 1. Explainable AI (XAI) in MLOps: Develop a framework that provides explainability for ML models deployed in production.

Project 2. Automated Bias Detection in ML Models: Implement fairness testing in MLOps pipelines to detect biased predictions.

Project 3. AI-Powered Model Explainability Dashboard: Build an interactive dashboard using SHAP or LIME for model explainability.

Project 4. AI-Based Model Interpretability & Bias Detection: Implement SHAP or LIME to analyze model decisions and detect bias.

---

10. Data Versioning & Management

Project 1. Automated Data Versioning with DVC: Using Data Version Control (DVC) to track and manage datasets that are used to train models, ensuring reproducibility and version control of data.

Project 2. Data Quality Automation: Creating pipelines to automatically clean and validate datasets used for training, removing anomalies, duplicates, and correcting labels.

Project 3. Data Drift Detection: Implementing systems to detect when the statistical properties of data change over time, which could impact model performance.

Project 4. Continuous Data Validation: Developing automated tools to validate incoming data in real-time and ensuring it adheres to the expected schema and quality standards before being fed into models.

Project 5. Data Pipeline Monitoring: Implementing monitoring for data pipelines to ensure they run efficiently and consistently without failures, catching errors early.

---

11. Model Monitoring Dashboards and Alerts

Project 1. End-to-End ML Monitoring Dashboard: Build a dashboard that tracks model performance, data drift, and system metrics.

Project 2. Data Drift Monitoring & Alerting System: Continuously track dataset changes and trigger model retraining if necessary.

---

12. Model Rollback and Failure Management

Project 1. Intelligent Model Rollback System: Automatically roll back ML models if performance degrades in production.

Project 2. AI-Enhanced Model Rollback Strategy: Automatically roll back to the best-performing model based on real-time inference results.

Project 3. Self-Healing ML Pipelines: Detect and resolve ML training failures automatically.

---

13. Automated Model Training & Tuning

Project 1. Automated Hyperparameter Optimization: Implementing systems that automatically tune the hyperparameters of machine learning models based on past training runs, improving accuracy and reducing human effort.

Project 2. CI/CD for Model Training Pipelines: Automating the entire process of model training, from data preprocessing to model evaluation, using CI/CD pipelines.

Project 3. Automated Model Versioning: Creating an automated system that manages different versions of models, ensuring that only validated models are deployed into production.

---

11. AI for Infrastructure & Network Monitoring

Project 5. AI-Powered Automated Remediation of Network Failures:
AI-driven automation of recovery actions when network issues are detected, reducing downtime.

Network failures can lead to downtime, affecting business operations. This project uses AI to detect and analyze network issues in real time, then automatically executes remediation actions. The goal is to minimize downtime by identifying patterns and taking corrective measures without human intervention.

Technologies Used

- **Python**: For scripting and automation
- **Machine Learning**: To predict network failures
- **Prometheus & Grafana**: For monitoring network health
- **ELK Stack**: For logging and analytics
- **Ansible**: For automation of remediation
- **Docker & Kubernetes**: For containerized deployment

Step-by-Step Implementation

Step 1: Setup Network Monitoring

Use **Prometheus** and **Grafana** to collect and visualize network metrics.

Install Prometheus & Grafana

```
sudo apt update
sudo apt install prometheus grafana -y
```

Start Prometheus & Grafana

```
sudo systemctl start prometheus
```

sudo systemctl enable prometheus
sudo systemctl start grafana-server
sudo systemctl enable grafana-server

Configure Prometheus to Monitor Network

Edit Prometheus config file:

sudo nano /etc/prometheus/prometheus.yml

Add:

yaml

```
scrape_configs:
  - job_name: 'network'
    static_configs:
      - targets: ['localhost:9090']
```

Restart Prometheus:

sudo systemctl restart prometheus

Step 2: Collect and Analyze Network Logs

Use the **ELK Stack (Elasticsearch, Logstash, Kibana)** to analyze network logs.

Install ELK

sudo apt update
sudo apt install elasticsearch logstash kibana -y

Start Services

```
sudo systemctl start elasticsearch
sudo systemctl enable elasticsearch
sudo systemctl start logstash
sudo systemctl enable logstash
sudo systemctl start kibana
sudo systemctl enable kibana
```

Configure Logstash

Edit Logstash pipeline:

```
sudo nano /etc/logstash/conf.d/network.conf
```

Example configuration:

yaml

```
input {
  file {
    path => "/var/log/syslog"
    start_position => "beginning"
  }
}
filter {
  grok {
    match => { "message" => "%{SYSLOGBASE}" }
  }
}
output {
  elasticsearch {
    hosts => ["localhost:9200"]
  }
}
```

Restart Logstash:

```
sudo systemctl restart logstash
```

Step 3: Implement AI for Anomaly Detection

Use a Python script to detect anomalies in network logs.

Install Required Libraries

```
pip install pandas scikit-learn tensorflow
```

Python Script for Anomaly Detection

```python
python

import pandas as pd
from sklearn.ensemble import IsolationForest

# Load network logs
data = pd.read_csv("network_logs.csv")

# Train anomaly detection model
model = IsolationForest(contamination=0.05)
model.fit(data)

# Detect anomalies
data['anomaly'] = model.predict(data)
anomalies = data[data['anomaly'] == -1]
print("Detected anomalies:\n", anomalies)
```

Step 4: Automate Remediation with Ansible

When an issue is detected, trigger automated remediation.

Install Ansible

```
sudo apt update
sudo apt install ansible -y
```

Create Ansible Playbook

```
nano fix_network_issue.yml
```

Example playbook:

yaml

```yaml
- name: Restart network service
  hosts: all
  tasks:
    - name: Restart network
      service:
        name: network-manager
        state: restarted
```

Run Ansible Playbook

```
ansible-playbook -i inventory fix_network_issue.yml
```

Step 5: Deploy Everything with Docker

Create a Dockerfile:

dockerfile

```dockerfile
FROM python:3.9
WORKDIR /app
COPY . .
RUN pip install -r requirements.txt
```

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

Build and run:

```
docker build -t ai-network-monitor .
docker run -d -p 5000:5000 ai-network-monitor
```

- **Prometheus & Grafana** collect and display network data.
- **ELK Stack** processes logs for network insights.
- **Python (ML Model)** detects network anomalies.
- **Ansible** executes remediation actions.
- **Docker** packages everything for easy deployment.

---

12. AI for ChatOps & Automated Support

Project 1. AI-Driven Automated Ticketing System: Integrating AI with ChatOps tools (like Slack/MS Teams) to automatically create and categorize tickets based on incident discussions.

In modern IT operations, **ChatOps** enables teams to collaborate efficiently using messaging platforms like **Slack** or **MS Teams**. By integrating AI, we can automate **ticket creation and categorization** based on incident discussions in chat channels.

This project will use:

- **Natural Language Processing (NLP)** for intent recognition
- **Slack/MS Teams API** for chat integration
- **JIRA/ServiceNow API** for ticketing
- **Python (Flask/FastAPI)** for the backend

Project Steps

1. Setup Environment

Install dependencies:

```
pip install flask fastapi uvicorn slack_sdk requests transformers
```

---

2. Configure Slack API

- Go to **Slack API** (https://api.slack.com/)
- Create a new **Slack App**
- Enable **Event Subscriptions**
- Subscribe to message.channels events
- Generate a **Bot Token**

Save the token:

```
export SLACK_BOT_TOKEN="xoxb-your-bot-token"
export SLACK_SIGNING_SECRET="your-signing-secret"
```

---

3. Build AI Model for Ticket Categorization

Using a pre-trained NLP model (BERT) to classify messages into categories:

python

```
from transformers import pipeline

classifier = pipeline("zero-shot-classification", model="facebook/bart-large-mnli")

def categorize_ticket(text):
    labels = ["Network Issue", "Software Bug", "Access Request", "Hardware
Failure"]
```

```
    result = classifier(text, candidate_labels=labels)
    return result["labels"][0]  # Highest probability category
```

---

4. Create Flask/FastAPI Backend

Set up a webhook for Slack messages:

python

```python
from flask import Flask, request
import requests
import os

app = Flask(__name__)
SLACK_BOT_TOKEN = os.getenv("SLACK_BOT_TOKEN")

@app.route("/slack/events", methods=["POST"])
def slack_events():
    data = request.json
    if "event" in data:
        text = data["event"]["text"]
        category = categorize_ticket(text)
        create_ticket(text, category)
        return "OK"
    return "No event", 400

def create_ticket(description, category):
    ticket_data = {"description": description, "category": category}
    requests.post("https://your-ticketing-system/api/tickets", json=ticket_data)

if __name__ == "__main__":
    app.run(port=3000)
```

Run the Flask app:

```
python app.py
```

---

5. Connect Ticketing System (JIRA/ServiceNow API)

Send ticket data to **JIRA API**:

python

```python
JIRA_API_URL = "https://your-jira-instance.atlassian.net/rest/api/3/issue"
JIRA_AUTH = ("your-email", "your-api-token")

def create_ticket(description, category):
    ticket_data = {
        "fields": {
            "project": {"key": "ITOPS"},
            "summary": description,
            "description": category,
            "issuetype": {"name": "Task"}
        }
    }
    requests.post(JIRA_API_URL, json=ticket_data, auth=JIRA_AUTH)
```

---

6. Deploy the Project

Use **ngrok** to expose your local server:

```
ngrok http 3000
```

Update the **Slack Event Subscription** URL with ngrok URL.

Conclusion

- **Slack/MS Teams** captures incident discussions
- **AI (NLP)** categorizes the issue
- **JIRA/ServiceNow API** creates a ticket automatically

This **AI-driven ChatOps integration** helps teams automate ticketing, improving response times and efficiency.

Project 2. Smart ChatOps Assistance for Infrastructure Troubleshooting: AI-powered assistant in chat systems like Slack that suggests infrastructure fixes in real time based on the conversation and context.

Introduction

ChatOps is a collaborative approach that integrates DevOps tools directly into chat applications like Slack, Teams, or Discord. This project builds an **AI-powered ChatOps assistant** that listens to conversations, understands context, and suggests solutions for **infrastructure issues** in real time.

It will use:

- **Natural Language Processing (NLP)** for understanding messages
- **Predefined Infrastructure Troubleshooting Playbooks**
- **Slack API** for integration
- **Python and OpenAI API** for AI-based suggestions
- **Docker and Kubernetes** for deployment

Step 1: Setting Up the Environment

1.1 Install Required Packages

Run the following command to install dependencies:

```
pip install flask slack_sdk openai python-dotenv requests
```

1.2 Set Up a Slack App

1. Go to Slack API and create a new app.
2. Enable **Bot Token Scopes**: chat:write, channels:history, app_mentions:read
3. Install the bot in your workspace and copy the **Bot User OAuth Token**.

1.3 Create a .env File

Store your Slack and OpenAI API keys securely in a .env file.

env

```
SLACK_BOT_TOKEN="xoxb-your-slack-token"
SLACK_SIGNING_SECRET="your-slack-signing-secret"
OPENAI_API_KEY="your-openai-key"
```

Step 2: Writing the Smart ChatOps Bot

Create a new Python file, **chatops_bot.py**.

2.1 Import Required Libraries

python

```
import os
import openai
import json
import logging
from slack_sdk import WebClient
from slack_sdk.errors import SlackApiError
from slack_sdk.web import SlackResponse
```

```
from flask import Flask, request, jsonify
from dotenv import load_dotenv

load_dotenv()

# Load API keys
SLACK_BOT_TOKEN = os.getenv("SLACK_BOT_TOKEN")
OPENAI_API_KEY = os.getenv("OPENAI_API_KEY")

slack_client = WebClient(token=SLACK_BOT_TOKEN)
openai.api_key = OPENAI_API_KEY

app = Flask(__name__)
logging.basicConfig(level=logging.INFO)
```

2.2 Function to Handle Incoming Messages

python

```python
def get_ai_suggestion(message):
    """ Uses OpenAI to suggest a fix based on the user's message. """
    response = openai.ChatCompletion.create(
        model="gpt-4",
        messages=[
            {"role": "system", "content": "You are an expert DevOps assistant."},
            {"role": "user", "content": message},
        ],
    )
    return response["choices"][0]["message"]["content"].strip()
```

2.3 Slack Event Listener

python

```
@app.route("/slack/events", methods=["POST"])
def slack_events():
    """ Listens for messages and responds with AI-based troubleshooting
suggestions. """
    data = request.json

    if "event" in data:
        event = data["event"]

        if event.get("type") == "app_mention" or event.get("type") == "message":
            user_message = event.get("text", "")
            channel_id = event["channel"]

            response_text = get_ai_suggestion(user_message)

            try:
                slack_client.chat_postMessage(channel=channel_id, text=response_text)
            except SlackApiError as e:
                logging.error(f"Slack API Error: {e.response['error']}")

    return jsonify({"status": "ok"})
```

Step 3: Running the Bot

3.1 Start Flask Server

Run the following command to start the bot:

```
python chatops_bot.py
```

Step 4: Deploying with Docker and Kubernetes

4.1 Create a Dockerfile

dockerfile

```
FROM python:3.9
WORKDIR /app
COPY . .
RUN pip install -r requirements.txt
CMD ["python", "chatops_bot.py"]
```

4.2 Build and Run the Docker Container

```
docker build -t chatops-bot .
docker run -d -p 5000:5000 --env-file .env chatops-bot
```

4.3 Kubernetes Deployment

Create a deployment.yaml file:

yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: chatops-bot
spec:
  replicas: 1
  selector:
    matchLabels:
      app: chatops-bot
  template:
    metadata:
      labels:
        app: chatops-bot
```

```yaml
    spec:
      containers:
      - name: chatops-bot
        image: chatops-bot:latest
        envFrom:
        - secretRef:
            name: chatops-secrets
---
apiVersion: v1
kind: Service
metadata:
  name: chatops-service
spec:
  type: LoadBalancer
  selector:
    app: chatops-bot
  ports:
  - protocol: TCP
    port: 80
    targetPort: 5000
```

Apply the deployment:

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

How the Code Works

1. The Flask server listens for **Slack events** (/slack/events endpoint).
2. When the bot is mentioned, it extracts the **message text**.
3. It **sends the message to OpenAI** and gets a suggested fix.
4. It **posts the response back to Slack**.
5. The bot runs inside a **Docker container**, and we use **Kubernetes** to scale and deploy it.

Example Output

Slack User:

> "Hey @ChatOpsBot, my Kubernetes pod is in CrashLoopBackOff.
> What should I do?"

ChatOpsBot Response:

> "Your pod is in **CrashLoopBackOff** likely due to:
>
> - A failing startup command
> - Insufficient resources
> - A missing config file
>
> Try running kubectl logs <pod-name> to debug."

Conclusion

- This project helps **DevOps teams troubleshoot issues faster** within Slack.
- Uses **AI (OpenAI API) to suggest fixes** dynamically.
- Easily **scalable with Docker and Kubernetes**.

Project 3. Chatbot for Continuous Deployment Assistance: A ChatOps bot powered by AI that helps DevOps teams by automatically updating the status of deployments and helping with rollback decisions.

A **ChatOps bot** is an AI-powered assistant that integrates with messaging platforms like Slack, Microsoft Teams, or Discord. This chatbot helps DevOps teams by providing real-time updates on deployment status, detecting failures, and assisting in rollback decisions.

In this project, we will:

- Develop a chatbot using **Python (Flask/FastAPI)**
- Use **OpenAI's GPT API** for AI-powered responses
- Integrate with **Slack (or Discord)** for communication
- Fetch deployment status from **Jenkins/GitHub Actions/Kubernetes**
- Provide rollback options using **Kubernetes commands**

Project Steps with Commands

1. Set Up Your Environment

Ensure you have Python installed:

python3 --version

Install required dependencies:

pip install flask slack_sdk openai requests kubernetes

---

2. Create a Slack App

- Go to Slack API
- Create a new app → From Scratch
- Enable **Socket Mode** and **Event Subscriptions**
- Get a **Bot Token** and add required permissions

---

3. Write the Chatbot Code

Create a file bot.py and add the following:

python

import os

```
import openai
import requests
from flask import Flask, request, jsonify
from slack_sdk import WebClient
from slack_sdk.errors import SlackApiError
from kubernetes import client, config

app = Flask(__name__)

# Load API keys
SLACK_BOT_TOKEN = "xoxb-XXXXXXX"  # Replace with your bot token
SLACK_SIGNING_SECRET = "XXXXXXX"
OPENAI_API_KEY = "sk-XXXXXXX"  # Replace with your OpenAI key

client = WebClient(token=SLACK_BOT_TOKEN)
openai.api_key = OPENAI_API_KEY

# Load Kubernetes config
config.load_kube_config()

# Function to check deployment status
def get_deployment_status(namespace="default", deployment_name="my-app"):
    v1 = client.AppsV1Api()
    try:
        deployment = v1.read_namespaced_deployment(deployment_name,
namespace)
        return f"Deployment {deployment_name} status:
{deployment.status.conditions[-1].type} -
{deployment.status.conditions[-1].status}"
    except Exception as e:
        return f"Error fetching deployment status: {str(e)}"

# Function to trigger rollback
def rollback_deployment(namespace="default", deployment_name="my-app"):
    v1 = client.AppsV1Api()
```

```
    try:
        deployment = v1.read_namespaced_deployment(deployment_name,
namespace)
        deployment.spec.revisionHistoryLimit = 1  # Rollback to previous version
        v1.patch_namespaced_deployment(deployment_name, namespace,
deployment)
        return f"Rollback initiated for {deployment_name}."
    except Exception as e:
        return f"Rollback failed: {str(e)}"

# Slack message handler
@app.route("/slack/events", methods=["POST"])
def slack_events():
    data = request.json
    if "event" in data:
        event = data["event"]
        if "text" in event:
            user_input = event["text"].lower()
            response_text = ""

            if "deployment status" in user_input:
                response_text = get_deployment_status()
            elif "rollback" in user_input:
                response_text = rollback_deployment()
            else:
                response_text = openai.ChatCompletion.create(
                    model="gpt-4",
                    messages=[{"role": "user", "content": user_input}],
                )["choices"][0]["message"]["content"]

            try:
                client.chat_postMessage(channel=event["channel"], text=response_text)
            except SlackApiError as e:
                print(f"Error sending message: {e.response['error']}")
```

```
    return jsonify({"status": "ok"})

if __name__ == "__main__":
    app.run(port=3000)
```

---

4. Run the Chatbot Locally

```
python bot.py
```

If you are using ngrok to expose Flask:

```
ngrok http 3000
```

Update Slack's **Request URL** with the ngrok URL.

---

5. Deploy on Kubernetes

Create a **Dockerfile**:

dockerfile

```
FROM python:3.9
WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY bot.py .
CMD ["python", "bot.py"]
```

Build & Push to Docker Hub:

```
docker build -t your-docker-username/chatbot .
docker push your-docker-username/chatbot
```

Deploy to Kubernetes:

yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: chatbot
  labels:
    app: chatbot
spec:
  replicas: 1
  selector:
    matchLabels:
      app: chatbot
  template:
    metadata:
      labels:
        app: chatbot
    spec:
      containers:
      - name: chatbot
        image: your-docker-username/chatbot
        ports:
        - containerPort: 3000
---
apiVersion: v1
kind: Service
metadata:
  name: chatbot-service
spec:
  type: LoadBalancer
  selector:
```

```
    app: chatbot
  ports:
  - protocol: TCP
    port: 80
    targetPort: 3000
```

Apply:

```
kubectl apply -f chatbot-deployment.yaml
kubectl get pods
```

- **Slack Integration:** Listens for Slack messages & replies automatically.
- **GPT AI:** Uses OpenAI's GPT model for intelligent responses.
- **Kubernetes Status Check:** Fetches deployment status.
- **Rollback Feature:** Rolls back to the previous deployment if requested.
- **Docker & Kubernetes Deployment:** Runs as a microservice.

Conclusion

This project helps DevOps teams **automate deployments, monitor applications, and make rollback decisions** via Slack. You can extend this by:

- **Adding CI/CD Integration** with GitHub Actions or Jenkins
- **Supporting Multiple Cloud Providers** (AWS, GCP, Azure)
- **Enhancing Security** with authentication mechanisms

Project 4. AI Chatbots for Security Incident Response: AI-powered bots integrated into ChatOps to respond to security incidents and suggest next steps or remediation actions.

In this project, we will build an **AI-powered chatbot** that integrates into **ChatOps** (such as Slack or Microsoft Teams) to help in **security incident response**. The bot will analyze security logs, detect incidents, and suggest remediation actions.

Key Features

- Real-time monitoring of security logs
- AI-based response generation using OpenAI's GPT
- Integration with ChatOps (Slack)
- Automated remediation recommendations

Technology Stack

- Python (Flask for API)
- OpenAI GPT (for AI responses)
- Slack API (for ChatOps integration)
- MongoDB (for logging incidents)
- Docker (for containerization)
- Jenkins (for CI/CD)
- Kubernetes (for deployment)
- Security Tools (Trivy, SonarQube)

Step-by-Step Implementation

Step 1: Set Up the Project

```
mkdir security-chatbot
cd security-chatbot
python3 -m venv venv
source venv/bin/activate
pip install flask openai slack_sdk pymongo requests
```

Step 2: Create config.py for API Keys

python

```
OPENAI_API_KEY = "your_openai_api_key"
SLACK_BOT_TOKEN = "your_slack_bot_token"
MONGO_URI = "mongodb://localhost:27017/security_logs"
```

Step 3: Implement Security Incident Chatbot (chatbot.py)

python

```
import os
import openai
import json
from slack_sdk import WebClient
from flask import Flask, request
from pymongo import MongoClient

# Load Config
from config import OPENAI_API_KEY, SLACK_BOT_TOKEN, MONGO_URI

# Initialize Services
app = Flask(__name__)
openai.api_key = OPENAI_API_KEY
slack_client = WebClient(token=SLACK_BOT_TOKEN)
mongo_client = MongoClient(MONGO_URI)
db = mongo_client.security_logs

# Function to analyze incidents
def analyze_incident(log):
    response = openai.ChatCompletion.create(
        model="gpt-4",
        messages=[{"role": "system", "content": "Analyze this security log and suggest remediation actions."},
                  {"role": "user", "content": log}]
```

```python
    )
    return response['choices'][0]['message']['content']

# API Endpoint for Slack Bot
@app.route("/slack/events", methods=["POST"])
def slack_events():
    data = request.json
    if "challenge" in data:
        return data["challenge"]

    event = data.get("event", {})
    if event.get("type") == "message" and "text" in event:
        log_text = event["text"]
        db.logs.insert_one({"log": log_text})  # Save in DB
        response_text = analyze_incident(log_text)
        slack_client.chat_postMessage(channel=event["channel"],
text=response_text)

    return "", 200

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

Step 4: Run the Application

```
export FLASK_APP=chatbot.py
flask run
```

Step 5: Dockerize the Application (Dockerfile)

Dockerfile

```
FROM python:3.9
```

```
WORKDIR /app
COPY . .
RUN pip install -r requirements.txt
CMD ["flask", "run", "--host=0.0.0.0"]
```

Build and run:

```
docker build -t security-chatbot .
docker run -p 5000:5000 security-chatbot
```

Step 6: Deploy to Kubernetes

Create deployment.yaml

yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: security-chatbot
spec:
  replicas: 1
  selector:
    matchLabels:
      app: security-chatbot
  template:
    metadata:
      labels:
        app: security-chatbot
    spec:
      containers:
      - name: security-chatbot
        image: your-dockerhub-username/security-chatbot
        ports:
```

```
      - containerPort: 5000
```

Deploy:

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

- **analyze_incident(log)**: Uses OpenAI GPT-4 to analyze security logs and suggest actions.
- **Slack API Integration**: Listens for messages in Slack and processes them.
- **MongoDB Logging**: Saves security logs in MongoDB for future analysis.
- **Docker & Kubernetes**: Ensures portability and scalability.

Project 5. Automated Root Cause Analysis via ChatOps: Using AI to correlate logs and metrics automatically and suggest a root cause in a ChatOps platform when an issue is reported.

Automated Root Cause Analysis (RCA) via ChatOps integrates AI-driven log and metrics correlation with a ChatOps platform (e.g., Slack, Microsoft Teams). When an issue is reported, AI automatically analyzes logs and metrics, identifies patterns, and suggests a possible root cause.

Why is this useful?

- Reduces manual troubleshooting effort
- Speeds up issue resolution
- Enhances collaboration through ChatOps

Tech Stack

- **Python** (for AI-based log correlation)
- **Elasticsearch + Kibana** (for centralized logging and visualization)
- **Prometheus + Grafana** (for metrics monitoring)
- **Slack API** (for ChatOps integration)

- **Docker + Kubernetes** (for containerization and orchestration)

Step-by-Step Implementation

Step 1: Setup Log and Metrics Collection

Install and configure **Elasticsearch, Kibana, Prometheus, and Grafana** to collect logs and metrics.

Install Elasticsearch & Kibana

```
docker network create monitoring
docker run -d --name elasticsearch --net monitoring -p 9200:9200 -e
"discovery.type=single-node" docker.elastic.co/elasticsearch/elasticsearch:7.17.0
docker run -d --name kibana --net monitoring -p 5601:5601
docker.elastic.co/kibana/kibana:7.17.0
```

Install Prometheus

```
mkdir -p /etc/prometheus
cat <<EOF > /etc/prometheus/prometheus.yml
global:
  scrape_interval: 15s
scrape_configs:
  - job_name: 'prometheus'
    static_configs:
      - targets: ['localhost:9090']
EOF

docker run -d --name prometheus --net monitoring -p 9090:9090 -v
/etc/prometheus/prometheus.yml:/etc/prometheus/prometheus.yml
prom/prometheus
```

Install Grafana

docker run -d --name grafana --net monitoring -p 3000:3000 grafana/grafana

---

Step 2: Deploy AI-based Log Analysis with Python

Install Dependencies:

pip install elasticsearch pandas scikit-learn slack_sdk

Python Script for Log Correlation

python

```
from elasticsearch import Elasticsearch
import pandas as pd
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.cluster import KMeans
from slack_sdk import WebClient
import os

# Connect to Elasticsearch
es = Elasticsearch(["http://localhost:9200"])

# Fetch logs from Elasticsearch
def fetch_logs():
    response = es.search(index="logs", body={"query": {"match_all": {}}},
size=1000)
    logs = [hit["_source"]["message"] for hit in response["hits"]["hits"]]
    return logs

# Process logs using AI clustering
def analyze_logs(logs):
    vectorizer = TfidfVectorizer(stop_words="english")
    X = vectorizer.fit_transform(logs)
    kmeans = KMeans(n_clusters=3, random_state=42).fit(X)
```

```
    return kmeans.labels_

# Send root cause to Slack
def notify_slack(message):
    client = WebClient(token=os.getenv("SLACK_BOT_TOKEN"))
    client.chat_postMessage(channel="#alerts", text=message)

# Main execution
logs = fetch_logs()
clusters = analyze_logs(logs)
notify_slack(f"Possible root causes detected: {set(clusters)}")
```

🔹 **Explanation:**

- Connects to **Elasticsearch** and fetches logs
- Uses **TF-IDF Vectorization** and **KMeans clustering** to group similar issues
- Sends detected root causes to **Slack**

Step 3: Integrate ChatOps with Slack

Create a Slack Bot

1. Go to Slack API
2. Create a new Slack App → Enable **Bots**
3. Get **OAuth Token** and set it as an environment variable

```
export SLACK_BOT_TOKEN="your-slack-bot-token"
```

Run the Python Log Analysis Script

```
python log_analysis.py
```

Once an issue is detected, a **Slack alert** is sent with potential root causes.

Step 4: Deploy the System Using Docker & Kubernetes

Create a Dockerfile
dockerfile

```
FROM python:3.9
WORKDIR /app
COPY . /app
RUN pip install -r requirements.txt
CMD ["python", "log_analysis.py"]
```

Build and Push the Docker Image

```
docker build -t your-dockerhub-user/log-analysis .
docker push your-dockerhub-user/log-analysis
```

Deploy to Kubernetes

Create a Kubernetes Deployment:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: log-analysis
spec:
  replicas: 1
  selector:
    matchLabels:
      app: log-analysis
  template:
    metadata:
      labels:
```

```yaml
      app: log-analysis
  spec:
    containers:
    - name: log-analysis
      image: your-dockerhub-user/log-analysis
      env:
      - name: SLACK_BOT_TOKEN
        valueFrom:
          secretKeyRef:
            name: slack-secret
            key: token
```

Apply the Deployment:

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

Summary

- Configured **Elasticsearch, Kibana, Prometheus, and Grafana** for logs and metrics
- Built an **AI-based Python script** to analyze logs and detect root causes
- Integrated **Slack ChatOps** to send alerts
- Deployed everything with **Docker & Kubernetes**

---

13. AI for Patch Management & Security

Project 1. AI-Powered Vulnerability Risk Scoring: Implementing AI to assess the risk of security vulnerabilities in systems based on potential impact, and prioritizing patches accordingly.

Security vulnerabilities can pose serious threats to systems. Traditional vulnerability management relies on manual triaging, which is slow and inefficient. This project leverages **AI/ML** to **automate vulnerability risk assessment** by analyzing **CVEs (Common Vulnerabilities and Exposures)**, their potential impact, and system configurations to prioritize patching effectively.

Project Overview

We will develop a system that:

- Collects vulnerability data from public sources like **NVD (National Vulnerability Database)**
- Uses **Machine Learning (ML)** to predict the severity of vulnerabilities
- Prioritizes patches based on the risk score
- Provides an API or CLI for fetching risk assessments

Tech Stack

- **Python** (Main Programming Language)
- **Flask** (For API Development)
- **Scikit-learn** (For ML Model)
- **Pandas, NumPy** (For Data Processing)
- **BeautifulSoup, Requests** (For Web Scraping CVE Data)
- **MongoDB/PostgreSQL** (Database for storing vulnerabilities)
- **Docker** (For containerization)
- **Jupyter Notebook** (For ML model training)

Step 1: Set Up Your Environment

Install Dependencies

 # Update system

```
sudo apt update && sudo apt upgrade -y

# Install Python & Pip
sudo apt install python3 python3-pip -y

# Install Virtual Environment
pip3 install virtualenv

# Create and activate a virtual environment
virtualenv venv
source venv/bin/activate

# Install required Python packages
pip install flask pandas numpy requests beautifulsoup4 scikit-learn joblib pymongo
```

Step 2: Scraping CVE Data

We will scrape the **National Vulnerability Database (NVD)** to fetch CVE details.

Create a Python Script: scraper.py

python

```python
import requests
from bs4 import BeautifulSoup
import json

# Function to fetch CVEs from NVD
def fetch_cves():
    url = "https://nvd.nist.gov/vuln/full-listing"
    response = requests.get(url)
    soup = BeautifulSoup(response.text, "html.parser")

    cve_data = []
```

```
    # Extract CVE details
    for link in soup.find_all("a", href=True):
        if "CVE-" in link.text:
            cve_id = link.text
            cve_url = "https://nvd.nist.gov" + link['href']
            cve_data.append({"id": cve_id, "url": cve_url})

    # Save data
    with open("cve_data.json", "w") as file:
        json.dump(cve_data, file, indent=4)

    print(f"Scraped {len(cve_data)} CVEs successfully!")

# Run the scraper
fetch_cves()
```

Step 3: Train the AI Model for Risk Scoring

We will use **Scikit-learn** to train a simple risk prediction model.

Create train_model.py

python

```
import pandas as pd
import numpy as np
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
import joblib

# Load sample vulnerability dataset (CSV format)
df = pd.read_csv("vulnerability_data.csv")
```

 # Assume dataset contains 'severity' (0: Low, 1: Medium, 2: High)
X = df[['cvss_score', 'exploitability', 'impact_score']]
y = df['severity']

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

 # Train ML model
model = RandomForestClassifier(n_estimators=100, random_state=42)
model.fit(X_train, y_train)

 # Evaluate model
y_pred = model.predict(X_test)
print(f"Model Accuracy: {accuracy_score(y_test, y_pred) * 100:.2f}%")

 # Save model
joblib.dump(model, "risk_model.pkl")

Step 4: Build an API to Predict Vulnerability Risk

We will create a **Flask API** to accept CVE data and return a risk score.

Create app.py

python

from flask import Flask, request, jsonify
import joblib
import numpy as np

app = Flask(__name__)

```python
# Load trained model
model = joblib.load("risk_model.pkl")

@app.route('/predict', methods=['POST'])
def predict_risk():
    data = request.get_json()
    features = np.array([[data['cvss_score'], data['exploitability'],
data['impact_score']]])

    risk_level = model.predict(features)[0]
    risk_labels = {0: "Low", 1: "Medium", 2: "High"}

    return jsonify({"risk_level": risk_labels[risk_level]})

if __name__ == '__main__':
    app.run(debug=True)
```

Step 5: Running the API

Start the Flask App

```
export FLASK_APP=app.py
flask run --host=0.0.0.0 --port=5000
```

Test the API

```
curl -X POST http://127.0.0.1:5000/predict -H "Content-Type: application/json" -d
'{
  "cvss_score": 7.5,
  "exploitability": 0.8,
  "impact_score": 0.9
}'
```

Expected Output:

json

```json
{
  "risk_level": "High"
}
```

Step 6: Store Results in MongoDB

MongoDB Setup

```
sudo apt install mongodb -y
sudo systemctl start mongodb
```

Modify app.py to Store Predictions

python

```python
from pymongo import MongoClient

# Connect to MongoDB
client = MongoClient("mongodb://localhost:27017/")
db = client["vulnerability_db"]
collection = db["risks"]

@app.route('/predict', methods=['POST'])
def predict_risk():
    data = request.get_json()
    features = np.array([[data['cvss_score'], data['exploitability'],
data['impact_score']]])

    risk_level = model.predict(features)[0]
    risk_labels = {0: "Low", 1: "Medium", 2: "High"}
```

```python
    result = {
        "cvss_score": data['cvss_score'],
        "exploitability": data['exploitability'],
        "impact_score": data['impact_score'],
        "risk_level": risk_labels[risk_level]
    }

    collection.insert_one(result)

    return jsonify(result)
```

Step 7: Containerizing the Application with Docker

Create Dockerfile

dockerfile

```dockerfile
FROM python:3.9

WORKDIR /app

COPY . .

RUN pip install -r requirements.txt

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

Build & Run the Docker Container

```
docker build -t ai-risk-scoring .
docker run -p 5000:5000 ai-risk-scoring
```

Conclusion

This project automates **vulnerability risk assessment** using AI. It fetches CVE data, predicts severity, prioritizes patches, and provides an API to fetch results. You can expand it further by integrating **real-time threat feeds** or **advanced AI models**.

Project 2. Predictive Patch Testing with AI: AI-driven system to predict the most effective patching sequence for systems to minimize downtime and risks.

Predictive Patch Testing with AI is a system that uses machine learning to analyze historical patch data, predict the most effective patching sequence, and reduce system downtime and security risks. This project helps DevOps teams optimize patch deployment, ensuring minimal disruption and maximum system stability.

Project Setup & Implementation

Step 1: Install Dependencies

Ensure you have Python and required libraries installed. Use the following command:

pip install pandas numpy scikit-learn flask

Step 2: Prepare Data

We need patch history data in a CSV file (patch_data.csv) with columns like:

- patch_id - Unique ID for each patch
- severity - Severity of the patch (High, Medium, Low)
- downtime - Downtime caused by the patch
- success_rate - Success rate of applying the patch

- dependency - Dependency level (if a patch depends on another patch)

Example Data (patch_data.csv):

csv

```
patch_id,severity,downtime,success_rate,dependency
P001,High,2,95,None
P002,Medium,1,98,P001
P003,Low,0.5,99,P002
```

Step 3: Load and Preprocess Data

Create a Python script (data_preprocessing.py) to load and clean the data.

python

```python
import pandas as pd

# Load dataset
df = pd.read_csv("patch_data.csv")

# Convert severity to numerical values (High: 3, Medium: 2, Low: 1)
severity_map = {"High": 3, "Medium": 2, "Low": 1}
df["severity"] = df["severity"].map(severity_map)

print("Preprocessed Data:")
print(df.head())
```

Run:

```
python data_preprocessing.py
```

Step 4: Train Machine Learning Model

Create predict_patch_sequence.py to train a predictive model.

python

```
import pandas as pd
import numpy as np
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split

# Load dataset
df = pd.read_csv("patch_data.csv")

# Convert severity to numerical values
severity_map = {"High": 3, "Medium": 2, "Low": 1}
df["severity"] = df["severity"].map(severity_map)

# Feature selection
X = df[["severity", "downtime", "success_rate"]]
y = df.index  # Assuming patch order is an important feature

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

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

# Save model
import joblib
joblib.dump(model, "patch_model.pkl")

print("Model trained and saved successfully.")
```

Run:

python predict_patch_sequence.py

Step 5: Build a REST API with Flask

Create app.py to expose an API for patch prediction.

python

```
from flask import Flask, request, jsonify
import joblib
import pandas as pd

app = Flask(__name__)

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

@app.route('/predict', methods=['POST'])
def predict():
    data = request.json
    df = pd.DataFrame([data])
    prediction = model.predict(df)
    return jsonify({"recommended_patch_sequence": prediction.tolist()})

if __name__ == '__main__':
    app.run(debug=True)
```

Run:

python app.py

Step 6: Test the API

Use curl or Postman to send a request.

```
curl -X POST "http://127.0.0.1:5000/predict" -H "Content-Type: application/json"
-d '{"severity":3, "downtime":1, "success_rate":95}'
```

Explanation of Code

1. Data Processing (data_preprocessing.py)

- Loads CSV data into a Pandas DataFrame.
- Converts text-based severity levels into numerical values.
- Displays the processed data.

2. Machine Learning Model (predict_patch_sequence.py)

- Uses RandomForestRegressor to predict patch application order.
- Splits data into training and test sets.
- Trains and saves the model for future use.

3. Flask API (app.py)

- Loads the trained model.
- Exposes a /predict endpoint where users can send patch data.
- Returns a recommended patch sequence based on ML predictions.

Conclusion

This project helps automate patching decisions by leveraging AI to minimize downtime and risks. It is useful for system administrators and DevOps engineers managing security updates.

Project 3. AI-Based Security Policy Violations Detection: Using machine learning to continuously monitor and detect policy violations in infrastructure, including IAM roles and network security configurations.

Security misconfigurations in cloud infrastructure can expose systems to unauthorized access, data breaches, and compliance failures. This project builds a **Machine Learning (ML)-based security monitoring system** that continuously scans IAM roles, firewall rules, and other network configurations to detect policy violations.

Using Python, AWS, and machine learning libraries like **scikit-learn**, we'll automate security checks and alert on policy violations.

Project Overview

- **Step 1:** Set up a Python environment
- **Step 2:** Collect and preprocess cloud security data
- **Step 3:** Train an ML model to detect security policy violations
- **Step 4:** Deploy the model to monitor IAM and network security policies
- **Step 5:** Automate alerts for violations using AWS Lambda and SNS

Step-by-Step Implementation

Step 1: Set Up Python Environment

Install necessary dependencies:

```
sudo apt update && sudo apt install python3-pip -y
pip3 install boto3 pandas numpy scikit-learn
```

Step 2: Collect IAM & Network Security Data

Use **AWS Boto3 SDK** to fetch IAM roles, security groups, and network configurations.

Fetch IAM Policies

python

```
import boto3

def get_iam_policies():
    iam = boto3.client('iam')
    policies = iam.list_policies(Scope='Local')['Policies']

    policy_data = []
    for policy in policies:
        policy_data.append({
            "PolicyName": policy['PolicyName'],
            "PolicyId": policy['PolicyId'],
            "Arn": policy['Arn'],
            "AttachmentCount": policy.get('AttachmentCount', 0),
            "CreateDate": policy['CreateDate'].isoformat()
        })

    return policy_data

print(get_iam_policies())
```

Fetch Security Group Rules

python

```
def get_security_groups():
    ec2 = boto3.client('ec2')
    security_groups = ec2.describe_security_groups()['SecurityGroups']
```

```
    sg_data = []
    for sg in security_groups:
        for rule in sg.get('IpPermissions', []):
            sg_data.append({
                "GroupName": sg['GroupName'],
                "FromPort": rule.get('FromPort', 'Any'),
                "ToPort": rule.get('ToPort', 'Any'),
                "IpProtocol": rule.get('IpProtocol', 'Any'),
                "CidrIp": rule.get('IpRanges', [])
            })

    return sg_data

print(get_security_groups())
```

Step 3: Train ML Model for Policy Violations

We use a **Random Forest Classifier** to train the model on labeled security configurations.

python

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score

# Sample dataset: ['AttachmentCount', 'OpenPort', 'Protocol', 'Violation']
data = pd.DataFrame([
    [0, 22, 'TCP', 1],  # Violation: SSH open
    [0, 80, 'TCP', 0],  # No violation: HTTP open
    [1, 3389, 'TCP', 1], # Violation: RDP open
```

```
    [1, 443, 'TCP', 0]   # No violation: HTTPS open
], columns=['AttachmentCount', 'OpenPort', 'Protocol', 'Violation'])

X = data[['AttachmentCount', 'OpenPort']]
y = data['Violation']

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

# Train model
model = RandomForestClassifier(n_estimators=100, random_state=42)
model.fit(X_train, y_train)

# Test accuracy
y_pred = model.predict(X_test)
print("Model Accuracy:", accuracy_score(y_test, y_pred))
```

Step 4: Deploy ML Model to Monitor Security

Save the trained model and deploy it to AWS Lambda.

python

```
import joblib

joblib.dump(model, "security_violation_model.pkl")
print("Model saved successfully!")
```

Upload the model to an **S3 bucket** and deploy it with **AWS Lambda**.

Step 5: Automate Alerts with AWS SNS

If the model detects a policy violation, send an alert using **AWS SNS**.

python

```
def send_alert(message):
    sns = boto3.client('sns')
    response = sns.publish(
        TopicArn="arn:aws:sns:us-east-1:123456789012:SecurityAlerts",
        Message=message,
        Subject="Security Policy Violation Detected"
    )
    print("Alert sent!", response)

# Example alert
send_alert("Unauthorized security group rule detected!")
```

Final Output

- Monitors IAM policies and security group rules
- Uses **ML model** to detect violations
- Deploys as **AWS Lambda function**
- Sends **alerts via AWS SNS**

Project 4. Automated Patch Scheduling with AI: AI model to suggest and automatically schedule patch deployment windows based on historical data and risk levels.

Automated Patch Scheduling with AI is a DevSecOps project that leverages **Machine Learning** to analyze historical patch data and risk levels, then automatically schedules optimal patch deployment windows. This helps organizations reduce downtime, enhance security, and streamline patch management.

The project consists of:

- **Data collection:** Extracting patch history and risk levels
- **AI model:** Using Machine Learning to predict the best patching windows
- **Scheduler:** Automating patch deployment using a CI/CD pipeline
- **Monitoring:** Logging and notifications for visibility

Tech Stack

- **Python (Flask)** – Backend API for scheduling
- **Scikit-learn (ML)** – AI model for patch prediction
- **PostgreSQL** – Database for storing patch data
- **Jenkins / GitHub Actions** – CI/CD pipeline for automation
- **Docker & Kubernetes** – Containerized deployment
- **Grafana & Prometheus** – Monitoring

Step-by-Step Implementation

Step 1: Setting Up the Environment

Install Dependencies

```
sudo apt update && sudo apt install -y python3 python3-pip postgresql docker docker-compose
pip3 install flask scikit-learn pandas psycopg2-binary
```

Step 2: Creating the Database

Login to PostgreSQL and create the patch history table:

```
sudo -u postgres psql
```

sql

```
CREATE DATABASE patch_db;
\c patch_db

CREATE TABLE patch_history (
    id SERIAL PRIMARY KEY,
    patch_name VARCHAR(100),
    applied_date TIMESTAMP,
    risk_level INTEGER,
    success BOOLEAN
);
```

Exit PostgreSQL:

```
\q
```

Step 3: Generating Patch Data (For AI Training)

Save the following as **generate_data.py**

python

```
import pandas as pd
import random
from datetime import datetime, timedelta

# Generate sample patching data
def generate_patch_data():
    data = []
    for i in range(100):
```

```
        patch_name = f"Patch-{random.randint(1000, 9999)}"
        applied_date = datetime.now() - timedelta(days=random.randint(1, 365))
        risk_level = random.randint(1, 10)
        success = random.choice([True, False])
        data.append([patch_name, applied_date, risk_level, success])

    df = pd.DataFrame(data, columns=["patch_name", "applied_date", "risk_level",
"success"])
    df.to_csv("patch_history.csv", index=False)
    print("Sample data generated: patch_history.csv")

generate_patch_data()
```

Run it:

```
python3 generate_data.py
```

Step 4: AI Model for Patch Scheduling

Save this as **train_model.py**

python

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
import pickle

# Load the dataset
df = pd.read_csv("patch_history.csv")

# Convert applied_date to numerical
df["applied_date"] = pd.to_datetime(df["applied_date"])
```

```
df["applied_date"] = df["applied_date"].astype(int) / 10**9

# Features and Labels
X = df[["applied_date", "risk_level"]]
y = df["success"]

# Train/Test Split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Train Model
model = RandomForestClassifier(n_estimators=100, random_state=42)
model.fit(X_train, y_train)

# Save Model
with open("patch_scheduler.pkl", "wb") as f:
    pickle.dump(model, f)

print("Model trained and saved as patch_scheduler.pkl")
```

Run it:

```
python3 train_model.py
```

Step 5: Flask API for Automated Patch Scheduling

Save this as **app.py**

python

```
from flask import Flask, request, jsonify
import pickle
import pandas as pd
```

```python
import datetime

app = Flask(__name__)

# Load trained AI model
with open("patch_scheduler.pkl", "rb") as f:
    model = pickle.load(f)

@app.route("/schedule", methods=["POST"])
def schedule_patch():
    data = request.json
    patch_name = data.get("patch_name")
    risk_level = data.get("risk_level")

    # Predict best deployment time
    now = datetime.datetime.now().timestamp()
    prediction = model.predict([[now, risk_level]])

    if prediction[0]:
        return jsonify({"message": f"Patch {patch_name} is scheduled now!"})
    else:
        return jsonify({"message": f"Patch {patch_name} should be postponed!"})

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

Run the API:

```
python3 app.py
```

Test it with Postman or CURL:

```
curl -X POST "http://localhost:5000/schedule" -H "Content-Type: application/json"
-d '{"patch_name":"Security Patch","risk_level":7}'
```

Step 6: Deploying with Docker

Create a **Dockerfile**

dockerfile

```
FROM python:3.9
WORKDIR /app
COPY . .
RUN pip install flask scikit-learn pandas
CMD ["python", "app.py"]
```

Build and Run:

docker build -t patch-scheduler .

docker run -p 5000:5000 patch-scheduler

Step 7: Automating with Jenkins

Create a **Jenkinsfile**

groovy

```
pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                sh 'docker build -t patch-scheduler .'
            }
        }
```

```
        stage('Deploy') {
            steps {
                sh 'docker run -d -p 5000:5000 patch-scheduler'
            }
        }
    }
}
```

Step 8: Monitoring with Grafana & Prometheus

Install Prometheus and Node Exporter:

```
sudo apt install prometheus node-exporter
```

Configure **prometheus.yml**

yaml

```
global:
  scrape_interval: 15s

scrape_configs:
  - job_name: 'patch-scheduler'
    static_configs:
      - targets: ['localhost:5000']
```

Restart Prometheus:

```
sudo systemctl restart prometheus
```

Set up **Grafana**, add Prometheus as a data source, and visualize API request logs.

Project Summary

This project automates patch scheduling using AI by analyzing historical data and risk levels. The Flask API predicts the best deployment windows, and CI/CD pipelines automate deployment. Monitoring is enabled via Prometheus & Grafana.

Project 5. Security Misconfiguration Detection in Infrastructure-as-Code: AI-based system that scans Terraform, Ansible, or other IAC configurations for security misconfigurations before deployment.

Security misconfigurations in Infrastructure-as-Code (IaC) can expose cloud environments to attacks. This project uses AI-based static analysis to detect misconfigurations in Terraform, Ansible, or Kubernetes manifests **before deployment**. It leverages tools like **Checkov, tfsec, Ansible-lint**, and **OpenAI LLM** to analyze security risks in IaC files.

Tech Stack

- **Terraform / Ansible / Kubernetes YAML** (IaC configurations)
- **Python** (for AI-based analysis)
- **Checkov / tfsec / Ansible-lint** (Static code analysis tools)
- **OpenAI API / Hugging Face Transformers** (For AI-based analysis)
- **Docker & GitHub Actions** (For CI/CD integration)

Step-by-Step Guide

Step 1: Set Up the Project

```
mkdir iac-security-scanner
cd iac-security-scanner
python3 -m venv venv
source venv/bin/activate  # On Windows: venv\Scripts\activate
pip install -r requirements.txt
```

Create requirements.txt:

txt

```
checkov
tfsec
ansible-lint
openai
transformers
```

Step 2: Create Terraform Misconfiguration for Testing

Create example.tf with security flaws:

hcl

```
resource "aws_security_group" "bad_sg" {
  name        = "bad_sg"
  description = "Open to world"

  ingress {
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]  # Security Risk!
  }
}
```

Step 3: Implement Static Code Analysis

Create scanner.py:

python

```python
import os
import subprocess

def run_checkov():
    print("Running Checkov...")
    subprocess.run(["checkov", "-d", "."])

def run_tfsec():
    print("Running tfsec...")
    subprocess.run(["tfsec", "."])

def run_ansible_lint():
    print("Running Ansible Lint...")
    subprocess.run(["ansible-lint", "."])

if __name__ == "__main__":
    run_checkov()
    run_tfsec()
    run_ansible_lint()
```

Run the script:

```
python scanner.py
```

Step 4: Add AI-Based Misconfiguration Detection

Modify scanner.py:

python

```python
import openai
```

```python
openai.api_key = "your-api-key"

def ai_analysis(iac_code):
    response = openai.ChatCompletion.create(
        model="gpt-4",
        messages=[{"role": "system", "content": "Analyze the security risks in this IaC code."},
                  {"role": "user", "content": iac_code}]
    )
    return response["choices"][0]["message"]["content"]

with open("example.tf", "r") as file:
    iac_code = file.read()

ai_result = ai_analysis(iac_code)
print("\nAI Security Analysis:\n", ai_result)
```

Step 5: Automate with GitHub Actions

Create .github/workflows/security-scan.yml:

yaml

```yaml
name: IaC Security Scan

on: [push, pull_request]

jobs:
  security-scan:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout Repository
        uses: actions/checkout@v3
```

```yaml
    - name: Set up Python
      uses: actions/setup-python@v4
      with:
        python-version: '3.8'

    - name: Install Dependencies
      run: |
        pip install checkov tfsec ansible-lint openai

    - name: Run Security Scan
      run: python scanner.py
```

Step 6: Dockerize the Application

Create Dockerfile:

dockerfile

```dockerfile
FROM python:3.8

WORKDIR /app
COPY . .

RUN pip install -r requirements.txt

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

Build and run:

```
docker build -t iac-security-scanner .
docker run iac-security-scanner
```

Explanation

1. **Checkov & tfsec** - Scans Terraform files for security issues.
2. **Ansible-lint** - Checks Ansible playbooks for best practices.
3. **OpenAI API** - AI-based analysis of misconfigurations.
4. **GitHub Actions** - Automates security scans on every code push.
5. **Docker** - Packages the tool for easy deployment.

MLOps Projects

1. Model Deployment and Management

Project 1. End-to-End ML Model Deployment on Kubernetes: Automate ML model training, testing, and deployment using Kubeflow.

Machine Learning (ML) models need a reliable, scalable, and automated way to be deployed and managed in production. **Kubeflow** is an open-source ML platform that runs on **Kubernetes**, providing tools for model training, tuning, deployment, and monitoring.

In this project, we will:

- Train an ML model
- Deploy it on Kubernetes using **Kubeflow**
- Automate the process with pipelines

This guide is beginner-friendly, covering every step with explanations.

Step-by-Step Project Guide

1. Set Up the Environment

We will use **Kind** (Kubernetes in Docker) to create a Kubernetes cluster, then install Kubeflow.

Install Required Tools

 # Install Kind

curl -Lo ./kind https://kind.sigs.k8s.io/dl/v0.20.0/kind-linux-amd64

chmod +x ./kind

sudo mv ./kind /usr/local/bin/kind

 # Install kubectl (Kubernetes CLI)

curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"

chmod +x kubectl

sudo mv kubectl /usr/local/bin/

 # Install Kustomize (for Kubeflow manifests)

curl -s "https://raw.githubusercontent.com/kubernetes-sigs/kustomize/master/hack/install_kustomize.sh" |

chmod +x kustomize

sudo mv kustomize /usr/local/bin/

 # Install Minikube (optional alternative to Kind)

curl -LO
https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64

chmod +x minikube-linux-amd64

sudo mv minikube-linux-amd64 /usr/local/bin/minikube

Create a Kubernetes Cluster

kind create cluster --name kubeflow-cluster

kubectl cluster-info --context kind-kubeflow-cluster

Install Kubeflow

export KF_NAME=my-kubeflow

export KF_DIR=$HOME/${KF_NAME}

export
KF_CONFIG=https://raw.githubusercontent.com/kubeflow/manifests/master/kusto
mize

mkdir -p ${KF_DIR} && cd ${KF_DIR}

 # Download and apply Kubeflow manifests

git clone https://github.com/kubeflow/manifests.git

cd manifests

while ! kustomize build example | kubectl apply -f -; do echo "Retrying to apply Kubeflow manifests..."; sleep 10; done

 # Check if all pods are running

kubectl get pods -n kubeflow

---

2. Train an ML Model

We will train a **Logistic Regression model** using **scikit-learn** and save it as a **pickle file**.

Create Python Training Script (train.py)

python

import pickle

import numpy as np

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

iris = load_iris()

X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.target, test_size=0.2, random_state=42)

 # Train model

```
model = LogisticRegression(max_iter=200)
model.fit(X_train, y_train)

# Evaluate model
y_pred = model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print(f"Model Accuracy: {accuracy:.2f}")

# Save model
with open("model.pkl", "wb") as f:
    pickle.dump(model, f)

print("Model saved as model.pkl")
```

Run the Training Script

```
python3 train.py
```

This will generate a model.pkl file, which we will deploy.

---

3. Create a Flask API to Serve the Model

We need an API to serve the trained model so other applications can use it.

Install Flask & Dependencies

pip install flask gunicorn scikit-learn numpy pandas

Create app.py (Flask API)

python

```
import pickle
import numpy as np
from flask import Flask, request, jsonify

app = Flask(__name__)

# Load trained model
with open("model.pkl", "rb") as f:
    model = pickle.load(f)

@app.route("/predict", methods=["POST"])
def predict():
    data = request.get_json()
    features = np.array(data["features"]).reshape(1, -1)
    prediction = model.predict(features)
    return jsonify({"prediction": int(prediction[0])})
```

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

---

4. Containerize the Model API

To deploy on Kubernetes, we package our API in a **Docker container**.

Create Dockerfile

dockerfile

```dockerfile
FROM python:3.8

WORKDIR /app

COPY requirements.txt requirements.txt

RUN pip install -r requirements.txt

COPY . .

CMD ["gunicorn", "--bind", "0.0.0.0:5000", "app:app"]
```

Create requirements.txt

txt

```
flask

gunicorn

scikit-learn
```

numpy

pandas

Build and Push the Docker Image

 # Build the image

docker build -t username/ml-api .

 # Tag and push to Docker Hub

docker tag username/ml-api username/ml-api:v1

docker push username/ml-api:v1

---

5. Deploy to Kubernetes Using Kubeflow

Create a Kubernetes Deployment

yaml

apiVersion: apps/v1

kind: Deployment

metadata:

  name: ml-api

spec:

  replicas: 1

```yaml
  selector:
    matchLabels:
      app: ml-api
  template:
    metadata:
      labels:
        app: ml-api
    spec:
      containers:
      - name: ml-api
        image: username/ml-api:v1
        ports:
        - containerPort: 5000
```

Create a Service for External Access

yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: ml-api-service
spec:
```

```yaml
  type: LoadBalancer
  selector:
    app: ml-api
  ports:
    - protocol: TCP
      port: 80
      targetPort: 5000
```

Apply Deployments

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

 # Check if pods are running

```
kubectl get pods
```

 # Get the service URL

```
kubectl get service ml-api-service
```

---

6. Test the Model API

Once deployed, we can send a request to the API.

Make a Prediction Request

```
curl -X POST "http://<EXTERNAL-IP>/predict" -H "Content-Type: application/json" -d '{"features": [5.1, 3.5, 1.4, 0.2]}'
```

Expected Response:

json

```
{"prediction": 0}
```

Summary

1. Set up Kubernetes and Kubeflow
2. Train and save an ML model
3. Create a Flask API to serve predictions
4. Containerize the API using Docker
5. Deploy it on Kubernetes using Kubeflow
6. Test the API

Project 2. CI/CD for ML Models with GitHub Actions & Docker: Build a pipeline that automates model versioning, testing, and deployment.

Project Overview

Objective

In this project, we will set up a **CI/CD pipeline for machine learning models** using **GitHub Actions & Docker**. The pipeline will:

- Automate model versioning
- Run unit tests
- Containerize the model using Docker
- Deploy the model

Step 1: Set Up the Project

Create a new project folder:

mkdir ml-cicd-project && cd ml-cicd-project

Initialize a Git repository:

git init

Create a virtual environment (optional but recommended):

python3 -m venv venv

source venv/bin/activate # On Windows: venv\Scripts\activate

Step 2: Create a Simple ML Model

Create a directory for the model:

mkdir model && cd model

Create a Python script (train.py) for training:

python

```
import pickle
from sklearn.linear_model import LogisticRegression
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split

# Load dataset
iris = load_iris()
X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.target, test_size=0.2, random_state=42)

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

# Save the model
with open("model.pkl", "wb") as f:
    pickle.dump(model, f)

print("Model trained and saved!")
```

Run the script:

python train.py

This will generate model.pkl, which stores our trained model.

Step 3: Create API to Serve the Model

Go back to the root directory:

cd ..

Create a new file app.py:

python

import pickle

from flask import Flask, request, jsonify

 # Load the model

with open("model/model.pkl", "rb") as f:

    model = pickle.load(f)

app = Flask(__name__)

```python
@app.route('/predict', methods=['POST'])
def predict():
    data = request.get_json()
    prediction = model.predict([data["features"]])
    return jsonify({"prediction": int(prediction[0])})

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

Install dependencies:

```
pip install flask scikit-learn
```

Run the API locally:

```
python app.py
```

Test using curl:

```
curl -X POST http://127.0.0.1:5000/predict -H "Content-Type: application/json" -d
'{"features": [5.1, 3.5, 1.4, 0.2]}'
```

Step 4: Create Unit Tests

Create a tests/ directory:

```
mkdir tests && cd tests
```

Create test_app.py:

```python
python

import json

import pytest

from app import app


@pytest.fixture

def client():

    return app.test_client()


def test_prediction(client):

    response = client.post("/predict", json={"features": [5.1, 3.5, 1.4, 0.2]})

    assert response.status_code == 200

    assert "prediction" in json.loads(response.data)
```

Run tests using pytest:

```
pip install pytest

pytest tests/
```

Step 5: Dockerize the Application

Create a Dockerfile:

dockerfile

 # Use Python base image

FROM python:3.9

 # Set working directory

WORKDIR /app

 # Copy files

COPY . .

 # Install dependencies

RUN pip install flask scikit-learn pytest

 # Run the app

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

Build and run the container:

```
docker build -t ml-api .
docker run -p 5000:5000 ml-api
```

Test the API again using curl (as shown in Step 3).

Step 6: Automate with GitHub Actions

Create a .github/workflows/ci-cd.yml file:

```yaml
name: ML CI/CD Pipeline

on:
  push:
    branches:
      - main
  pull_request:
    branches:
      - main

jobs:
  test:
    runs-on: ubuntu-latest
```

```yaml
  steps:
    - name: Checkout Repository
      uses: actions/checkout@v3

    - name: Set up Python
      uses: actions/setup-python@v4
      with:
        python-version: '3.9'

    - name: Install dependencies
      run: pip install -r requirements.txt

    - name: Run Tests
      run: pytest tests/

docker:
  runs-on: ubuntu-latest
  needs: test
  steps:
    - name: Checkout Repository
      uses: actions/checkout@v3
```

- name: Build Docker Image

  run: docker build -t ml-api .

- name: Log in to DockerHub

  run: echo "${{ secrets.DOCKER_PASSWORD }}" | docker login -u "${{ secrets.DOCKER_USERNAME }}" --password-stdin

- name: Push Docker Image

  run: docker tag ml-api your-dockerhub-username/ml-api:latest && docker push your-dockerhub-username/ml-api:latest

- **Tests the ML model** when code is pushed to GitHub
- **Builds a Docker image** only if tests pass
- **Pushes the image to Docker Hub**

Step 7: Deploy the Model using Docker

On a cloud server, pull and run the container:

docker pull your-dockerhub-username/ml-api:latest

docker run -d -p 5000:5000 your-dockerhub-username/ml-api

This project automated ML model deployment using:
✔ **GitHub Actions** for CI/CD
✔ **Docker** for containerization
✔ **Flask API** to serve predictions
✔ **Unit tests** for reliability

Project 3. Serverless ML Model Deployment with AWS Lambda & S3:
Automate model deployment using a serverless framework.

Machine learning models are often deployed as APIs for real-world applications. **AWS Lambda** allows serverless deployment, meaning we don't need to manage infrastructure. **Amazon S3** is used to store the trained model. This project will guide you in:

- Training an ML model
- Storing the model in S3
- Creating an AWS Lambda function to load and serve predictions
- Deploying the function with API Gateway

Step-by-Step Implementation

Step 1: Train an ML Model & Save It

We will train a simple **scikit-learn model**, serialize it, and store it in S3.

Install dependencies

pip install numpy pandas scikit-learn boto3 joblib

Train & Save Model

python

```
import numpy as np
import pandas as pd
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_iris
import joblib
import boto3

# Load dataset
data = load_iris()
X = data.data
y = data.target

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

# Train model
model = LogisticRegression(max_iter=200)
model.fit(X_train, y_train)
```

 # Save model

joblib.dump(model, "iris_model.pkl")

print("Model saved as iris_model.pkl")

 # Upload model to S3

s3 = boto3.client("s3")

bucket_name = "your-s3-bucket-name"

s3.upload_file("iris_model.pkl", bucket_name, "iris_model.pkl")

print(f"Model uploaded to S3 bucket {bucket_name}")

Explanation:

- Loads the Iris dataset
- Splits it into training and test sets
- Trains a logistic regression model
- Saves the model using joblib
- Uploads it to an **S3 bucket** using **boto3**

Step 2: Create AWS Lambda Function

AWS Lambda will load the model from S3 and serve predictions.

Create a new Python file (lambda_function.py)

python

import json

```
import boto3
import joblib
import numpy as np
import os
from io import BytesIO

# Load model from S3
s3 = boto3.client("s3")
bucket_name = "your-s3-bucket-name"
model_file = "iris_model.pkl"

def load_model():
    response = s3.get_object(Bucket=bucket_name, Key=model_file)
    model = joblib.load(BytesIO(response["Body"].read()))
    return model

model = load_model()

def lambda_handler(event, context):
    try:
        body = json.loads(event["body"])
```

```
    features = np.array(body["features"]).reshape(1, -1)

    # Make prediction
    prediction = model.predict(features).tolist()

    return {
        "statusCode": 200,
        "body": json.dumps({"prediction": prediction})
    }

  except Exception as e:
    return {
        "statusCode": 400,
        "body": json.dumps({"error": str(e)})
    }
```

- Loads the model from **S3** dynamically when Lambda starts
- Receives **features** from API request
- Converts them into the correct format
- Predicts using the ML model
- Returns the prediction in **JSON format**

Step 3: Deploy Lambda using AWS CLI

1. Package Dependencies

AWS Lambda needs dependencies packaged with the code. Run:

```
mkdir package

pip install joblib scikit-learn numpy boto3 -t package/

cd package

zip -r ../lambda_function.zip .

cd ..

zip -g lambda_function.zip lambda_function.py
```

This creates lambda_function.zip with all necessary files.

2. Upload Lambda Function

```
aws lambda create-function \

    --function-name iris-predictor \

    --runtime python3.8 \

    --role arn:aws:iam::your-account-id:role/your-lambda-role \

    --handler lambda_function.lambda_handler \

    --timeout 10 \

    --memory-size 256 \

    --zip-file fileb://lambda_function.zip
```

- Creates a Lambda function named iris-predictor
- Uses Python 3.8
- Assigns an **IAM role** with S3 access
- Allocates **256MB memory** and **10-second timeout**

Step 4: Expose as API with API Gateway

To make the function accessible via HTTP:

1. Create API Gateway

```
aws apigateway create-rest-api --name "IrisPredictorAPI"
```

2. Deploy API

```
aws apigateway create-deployment \
  --rest-api-id your-api-id \
  --stage-name prod
```

Now, your model is live at

arduino

https://your-api-id.execute-api.region.amazonaws.com/prod

Use **Postman** or curl to test:

```
curl -X POST "https://your-api-id.execute-api.region.amazonaws.com/prod" \
  -H "Content-Type: application/json" \
```

```
-d '{"features": [5.1, 3.5, 1.4, 0.2]}'
```

Conclusion

You have successfully deployed a **serverless ML model** using AWS Lambda & S3. This approach eliminates infrastructure management and scales automatically.

Project 4. Multi-Cloud Model Deployment & Monitoring: Deploy models across AWS, GCP, and Azure with centralized monitoring.

In this project, we will deploy a machine learning model across **AWS, GCP, and Azure** while ensuring centralized monitoring. This **multi-cloud strategy** enhances reliability, scalability, and cost-efficiency. We will use **Docker, Kubernetes, Terraform, and Prometheus/Grafana** for deployment and monitoring.

Project Steps

1. **Set Up Cloud Accounts** (AWS, GCP, Azure)
2. **Train & Save the Machine Learning Model**
3. **Containerize the Model using Docker**
4. **Deploy Model on Kubernetes Clusters (EKS, GKE, AKS)**
5. **Set Up Centralized Monitoring with Prometheus & Grafana**
6. **Test & Validate the Multi-Cloud Deployment**

Step 1: Set Up Cloud Accounts & CLI

1.1 Install Cloud CLIs

AWS CLI

curl "https://awscli.amazonaws.com/AWSCLIV2.pkg" -o "AWSCLIV2.pkg" && sudo installer -pkg AWSCLIV2.pkg -target /

GCP CLI

curl https://sdk.cloud.google.com | && exec -l $SHELL && gcloud init

Azure CLI

curl -sL https://aka.ms/InstallAzureCLIDeb | sudo

1.2 Authenticate & Configure CLI

aws configure

gcloud auth login

az login

Step 2: Train & Save the Model

We will use a **simple Flask API** to serve a trained machine learning model.

2.1 Install Dependencies

pip install flask scikit-learn joblib pandas

2.2 Train and Save Model (train.py)

python

import joblib

import pandas as pd

from sklearn.model_selection import train_test_split

from sklearn.ensemble import RandomForestClassifier

from sklearn.metrics import accuracy_score

 # Load dataset

data =
pd.read_csv("https://raw.githubusercontent.com/jbrownlee/Datasets/master/pima-indians-diabetes.data.csv")

X = data.iloc[:, :-1]

y = data.iloc[:, -1]

 # Split dataset

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

 # Train model

model = RandomForestClassifier(n_estimators=100, random_state=42)

model.fit(X_train, y_train)

```
# Evaluate
y_pred = model.predict(X_test)
print(f"Accuracy: {accuracy_score(y_test, y_pred)}")

# Save model
joblib.dump(model, "model.pkl")
```

Step 3: Create Flask API & Dockerize

3.1 Build Flask API (app.py)

python

```
from flask import Flask, request, jsonify
import joblib
import numpy as np

app = Flask(__name__)

# Load model
model = joblib.load("model.pkl")
```

```python
@app.route('/predict', methods=['POST'])
def predict():
    data = request.get_json()
    prediction = model.predict([np.array(data["features"])])
    return jsonify({"prediction": int(prediction[0])})

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

3.2 Create Dockerfile

dockerfile

```dockerfile
FROM python:3.9
WORKDIR /app
COPY . /app
RUN pip install flask joblib scikit-learn numpy
CMD ["python", "app.py"]
```

3.3 Build and Push Docker Image

```
docker build -t ml-model:latest .
docker tag ml-model gcr.io/YOUR_PROJECT_ID/ml-model:v1
docker push gcr.io/YOUR_PROJECT_ID/ml-model:v1
```

Step 4: Deploy on Kubernetes (EKS, GKE, AKS)

4.1 Create Kubernetes Deployment File (deployment.yaml)

yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ml-model
spec:
  replicas: 2
  selector:
    matchLabels:
      app: ml-model
  template:
    metadata:
      labels:
        app: ml-model
    spec:
      containers:
        - name: ml-model
```

```yaml
        image: gcr.io/YOUR_PROJECT_ID/ml-model:v1
        ports:
          - containerPort: 5000
---
apiVersion: v1
kind: Service
metadata:
  name: ml-service
spec:
  selector:
    app: ml-model
  ports:
    - protocol: TCP
      port: 80
      targetPort: 5000
  type: LoadBalancer
```

4.2 Deploy on AWS EKS

```
eksctl create cluster --name ml-cluster --region us-east-1
kubectl apply -f deployment.yaml
```

4.3 Deploy on GCP GKE

```
gcloud container clusters create ml-cluster --zone us-central1-a
kubectl apply -f deployment.yaml
```

4.4 Deploy on Azure AKS

```
az aks create --resource-group myResourceGroup --name ml-cluster --node-count 1
kubectl apply -f deployment.yaml
```

Step 5: Set Up Centralized Monitoring

5.1 Install Prometheus

```
kubectl apply -f
https://raw.githubusercontent.com/prometheus-operator/prometheus-operator/main/
bundle.yaml
```

5.2 Deploy Grafana

```
kubectl apply -f
https://raw.githubusercontent.com/grafana/grafana/main/deploy/kubernetes/grafana
.yaml
```

5.3 Access Grafana

```
kubectl port-forward svc/grafana 3000:3000
```

- Open **http://localhost:3000** in your browser
- Default login: **admin/admin**

Step 6: Test the Deployment

6.1 Get Load Balancer URL

kubectl get svc ml-service

6.2 Send a Prediction Request

curl -X POST "http://<LOAD_BALANCER_IP>/predict" -H "Content-Type: application/json" -d '{"features": [5, 166, 72, 19, 175, 25.8, 0.587, 51]}'

Final Thoughts

- We **trained and containerized** a model
- We **deployed it across AWS, GCP, and Azure** using Kubernetes
- We **set up centralized monitoring** with Prometheus and Grafana
- We **tested and validated the deployment**

This project follows **MLOps best practices**, ensuring scalability, reliability, and monitoring in a **multi-cloud environment**

Project 5. ML Model Canary Deployment with Kubernetes & Istio: Deploy new ML models in production using progressive rollout strategies.

In production environments, deploying machine learning (ML) models can be risky due to unpredictable behavior. **Canary deployment** helps mitigate this risk by gradually rolling out a new version of the model while monitoring its performance. If successful, traffic shifts entirely to the new model; otherwise, the rollout is reverted.

This project involves:

- Containerizing an ML model
- Deploying it on Kubernetes
- Implementing Istio for canary traffic shifting
- Monitoring with Prometheus & Grafana

Project Setup and Step-by-Step Execution

Step 1: Prerequisites

Ensure you have the following installed:

- **Kubernetes cluster** (minikube/kind or cloud-based)
- **Istio** (service mesh)
- **Docker** (for containerization)
- **Kubectl** (to manage Kubernetes)
- **Helm** (for package management)
- **Prometheus & Grafana** (for monitoring)

Step 2: Set Up Kubernetes Cluster

If using kind, create a cluster:

kind create cluster --name ml-deploy

kubectl get nodes

For minikube:

minikube start

Step 3: Install Istio

Download and install Istio:

curl -L https://istio.io/downloadIstio | sh -

cd istio-*

export PATH=$PWD/bin:$PATH

istioctl install --set profile=demo -y

Enable Istio injection:

kubectl label namespace default istio-injection=enabled

Step 4: Build & Containerize the ML Model

Create a simple **Flask-based ML model API**.

app.py (Flask App for ML Model)

python

from flask import Flask, request, jsonify

import numpy as np

```
import joblib

app = Flask(__name__)

# Load ML model
model = joblib.load("model.pkl")

@app.route('/predict', methods=['POST'])
def predict():
    data = request.json
    prediction = model.predict(np.array(data['features']).reshape(1, -1))
    return jsonify({'prediction': prediction.tolist()})

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

Save this model as model.pkl:

python

```
from sklearn.ensemble import RandomForestClassifier
import joblib
import numpy as np
```

```
X_train = np.random.rand(100, 5)

y_train = np.random.randint(0, 2, 100)


model = RandomForestClassifier()

model.fit(X_train, y_train)


joblib.dump(model, "model.pkl")
```

Step 5: Dockerize the ML API

Create a Dockerfile:

Dockerfile

```
FROM python:3.9

WORKDIR /app

COPY . /app

RUN pip install flask numpy joblib scikit-learn

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

Build and push the image:

```
docker build -t your-dockerhub/ml-model:v1 .
```

```
docker push your-dockerhub/ml-model:v1
```

Step 6: Deploy the ML Model in Kubernetes

Create a deployment YAML ml-deployment.yaml:

yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ml-model-v1
  labels:
    app: ml-model
    version: v1
spec:
  replicas: 3
  selector:
    matchLabels:
      app: ml-model
      version: v1
  template:
    metadata:
```

```
      labels:
        app: ml-model
        version: v1
    spec:
      containers:
      - name: ml-model
        image: your-dockerhub/ml-model:v1
        ports:
        - containerPort: 5000
```

Apply the deployment:

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

Step 7: Expose the Service

Create a Kubernetes service ml-service.yaml:

yaml

```
apiVersion: v1
kind: Service
metadata:
  name: ml-service
```

```yaml
spec:
  selector:
    app: ml-model
  ports:
    - protocol: TCP
      port: 80
      targetPort: 5000
  type: ClusterIP
```

Apply it:

```
kubectl apply -f ml-service.yaml
```

Step 8: Deploy Istio Gateway & VirtualService

Create an istio-gateway.yaml file:

yaml

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: ml-gateway
spec:
```

```
  selector:
    istio: ingressgateway
  servers:
  - port:
      number: 80
      name: http
      protocol: HTTP
    hosts:
    - "*"
```

Create a virtual-service.yaml:

yaml

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: ml-virtual-service
spec:
  hosts:
  - "*"
  gateways:
```

```yaml
  - ml-gateway
  http:
  - route:
    - destination:
        host: ml-service
        subset: v1
      weight: 90
    - destination:
        host: ml-service
        subset: v2
      weight: 10
```

Apply the configurations:

```
kubectl apply -f istio-gateway.yaml
kubectl apply -f virtual-service.yaml
```

Step 9: Deploy ML Model v2 (Canary Release)

Modify ml-deployment.yaml for v2:

yaml

```yaml
apiVersion: apps/v1
```

```yaml
kind: Deployment
metadata:
  name: ml-model-v2
  labels:
    app: ml-model
    version: v2
spec:
  replicas: 1
  selector:
    matchLabels:
      app: ml-model
      version: v2
  template:
    metadata:
      labels:
        app: ml-model
        version: v2
    spec:
      containers:
      - name: ml-model
        image: your-dockerhub/ml-model:v2
```

```yaml
        ports:
        - containerPort: 5000
```

Apply:

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

Step 10: Gradually Increase Traffic to v2

Modify virtual-service.yaml:

yaml

```yaml
http:
  - route:
    - destination:
        host: ml-service
        subset: v1
      weight: 50
    - destination:
        host: ml-service
        subset: v2
      weight: 50
```

Apply:

kubectl apply -f virtual-service.yaml

Step 11: Monitoring with Prometheus & Grafana

Install using Helm:

helm repo add prometheus-community https://prometheus-community.github.io/helm-charts

helm install prometheus prometheus-community/kube-prometheus-stack

Access Grafana:

kubectl port-forward svc/prometheus-grafana 3000:80

Login at http://localhost:3000 (default user: admin, password: prom-operator).

Conclusion

- The **ML model was deployed in Kubernetes**.
- **Istio was used to route traffic** between different versions.
- **Canary deployment** allowed progressive rollout of v2.
- **Monitoring tools** (Prometheus & Grafana) helped track performance.

Project 6. Automated ML Model Deployment on Multi-Cloud: Implement a system that deploys models across AWS, GCP, and Azure dynamically.

In this project, we will automate the deployment of a machine learning model across AWS, GCP, and Azure dynamically. The objective is to create a CI/CD pipeline that:

- Trains and packages an ML model
- Deploys it to AWS, GCP, and Azure using Terraform and Kubernetes
- Automates the entire workflow with GitHub Actions or Jenkins

Tech Stack Used

- **Machine Learning**: Scikit-learn, Flask (for API)
- **Cloud Platforms**: AWS, GCP, Azure
- **Infrastructure as Code (IaC)**: Terraform
- **Containerization**: Docker
- **Orchestration**: Kubernetes
- **CI/CD**: GitHub Actions / Jenkins

Step-by-Step Implementation

Step 1: Setup ML Model and API

We create a basic Flask API that serves a trained ML model.

Install Required Libraries

pip install flask scikit-learn joblib

Code: ml_model.py (Train and Save Model)

python

import joblib

```
import numpy as np

from sklearn.ensemble import RandomForestClassifier

from sklearn.datasets import load_iris

from sklearn.model_selection import train_test_split


# Load dataset

iris = load_iris()

X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.target, test_size=0.2, random_state=42)


# Train model

model = RandomForestClassifier(n_estimators=100, random_state=42)

model.fit(X_train, y_train)


# Save model

joblib.dump(model, 'iris_model.pkl')

print("Model saved as iris_model.pkl")
```

Code: app.py (Flask API for Inference)

python

```
from flask import Flask, request, jsonify

import joblib
```

```
import numpy as np

app = Flask(__name__)

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

@app.route('/predict', methods=['POST'])
def predict():
    data = request.json['features']
    prediction = model.predict([np.array(data)])
    return jsonify({'prediction': int(prediction[0])})

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

Step 2: Containerize the Application

We will create a Docker container for our Flask API.

Dockerfile

```
FROM python:3.9
```

WORKDIR /app

COPY requirements.txt .

RUN pip install -r requirements.txt

COPY . .

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

requirements.txt

nginx

flask

scikit-learn

joblib

numpy

Build and Run Docker Container

docker build -t ml-api .

docker run -p 5000:5000 ml-api

Step 3: Push Image to Docker Hub

docker tag ml-api <your-dockerhub-username>/ml-api:v1

docker login

```
docker push <your-dockerhub-username>/ml-api:v1
```

Step 4: Deploy Using Terraform on Multi-Cloud

We will use Terraform to provision cloud infrastructure on AWS, GCP, and Azure.

Terraform Files

- main.tf → Defines multi-cloud infrastructure
- aws.tf → AWS resources
- gcp.tf → GCP resources
- azure.tf → Azure resources

Example: AWS Terraform Deployment (aws.tf)

hcl

```
provider "aws" {

  region = "us-east-1"

}


resource "aws_instance" "ml_server" {

  ami           = "ami-0abcdef1234567890"

  instance_type = "t2.micro"


  user_data = <<-EOF

              #!/bin/
```

```
        docker run -d -p 5000:5000 <your-dockerhub-username>/ml-api:v1
        EOF
}
```

Deploy on AWS

```
terraform init
terraform apply -auto-approve
```

Step 5: Automate Deployment Using GitHub Actions

Create a .github/workflows/deploy.yml file.

yaml

```yaml
name: Deploy ML Model to Multi-Cloud

on:
  push:
    branches:
      - main

jobs:
```

```
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v2

      - name: Build Docker image
        run: |
          docker build -t <your-dockerhub-username>/ml-api:v1 .
          docker login -u ${{ secrets.DOCKER_USERNAME }} -p ${{
secrets.DOCKER_PASSWORD }}
          docker push <your-dockerhub-username>/ml-api:v1

  deploy:
    needs: build
    runs-on: ubuntu-latest
    steps:
      - name: Deploy to AWS
        run: terraform -chdir=terraform/aws apply -auto-approve

      - name: Deploy to GCP
        run: terraform -chdir=terraform/gcp apply -auto-approve
```

```
- name: Deploy to Azure
  run: terraform -chdir=terraform/azure apply -auto-approve
```

Step 6: Access the API

Find the public IP of the deployed instance and test using curl:

```
curl -X POST http://<public-ip>:5000/predict -H "Content-Type: application/json" -d '{"features": [5.1, 3.5, 1.4, 0.2]}'
```

---

1. **Machine Learning Model (ml_model.py)**
   - Loads dataset, trains a RandomForest model, and saves it as a .pkl file.
2. **Flask API (app.py)**
   - Loads the model and serves predictions via HTTP POST requests.
3. **Dockerfile**
   - Creates a lightweight containerized version of the ML API.
4. **Terraform Scripts (aws.tf, gcp.tf, azure.tf)**
   - Automates deployment across multiple clouds.
5. **GitHub Actions (deploy.yml)**
   - Automates the CI/CD pipeline for multi-cloud deployment.

Conclusion

This project demonstrates how to automate ML model deployment across AWS, GCP, and Azure using Docker, Terraform, and CI/CD. It ensures efficient, scalable, and reproducible deployment across multiple cloud providers.

Project 7. CI/CD Pipeline for ML with Feature Drift Detection: Implement an automated system that detects feature drift and retrains models accordingly.

In Machine Learning, feature drift occurs when the statistical properties of input data change over time, making the model less effective. A **CI/CD pipeline for ML with feature drift detection** automates the process of detecting drift and retraining the model to maintain its performance.

This project will cover:

- Data preprocessing and training an initial model
- Feature drift detection using statistical methods
- Automated retraining and deployment using CI/CD tools (GitHub Actions/Jenkins)
- Model monitoring with Prometheus and Grafana

Step 1: Setup Your Environment

Ensure you have the required dependencies installed.

 # Create a virtual environment

python3 -m venv ml-cicd-env

source ml-cicd-env/bin/activate # On Linux/macOS

ml-cicd-env\Scripts\activate # On Windows

 # Install dependencies

pip install pandas numpy scikit-learn evidently flask fastapi uvicorn requests mlflow prometheus_client

---

Step 2: Create Dataset and Initial Model

We'll create a sample dataset and train an initial model.

train_model.py

python

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
import joblib

# Generate sample dataset
np.random.seed(42)
data = pd.DataFrame({
    'feature1': np.random.normal(0, 1, 1000),
    'feature2': np.random.normal(5, 2, 1000),
    'feature3': np.random.choice([0, 1], size=1000),
    'target': np.random.choice([0, 1], size=1000)
})
```

X = data.drop(columns=['target'])

y = data['target']

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

 # Train a RandomForest model

model = RandomForestClassifier(n_estimators=100, random_state=42)

model.fit(X_train, y_train)

 # Save the model

joblib.dump(model, 'model.pkl')

print("Model trained and saved as model.pkl")

Run the script:

python train_model.py

Step 3: Implement Feature Drift Detection

We use the **Evidently AI** library for feature drift detection.

drift_detection.py

python

```python
import pandas as pd
import numpy as np
import joblib
from evidently.report import Report
from evidently.metric_preset import DataDriftPreset
import json

# Load model and generate new (possibly drifted) data
model = joblib.load('model.pkl')

new_data = pd.DataFrame({
    'feature1': np.random.normal(0.2, 1, 200),  # Small shift in mean
    'feature2': np.random.normal(5.5, 2, 200),
    'feature3': np.random.choice([0, 1], size=200)
})

# Compare new data with training data
report = Report(metrics=[DataDriftPreset()])
```

report.run(reference_data=pd.read_csv("train_data.csv"), current_data=new_data)

 # Save drift report

with open("drift_report.json", "w") as f:

  json.dump(report.as_dict(), f)

print("Drift detection completed. Check drift_report.json")

Run the script:

python drift_detection.py

If drift is detected, retrain the model.

Step 4: Automate CI/CD with GitHub Actions

Create .github/workflows/ml_pipeline.yml:

yaml

name: ML CI/CD Pipeline

on:

  push:

```yaml
    branches:
      - main
  schedule:
    - cron: '0 0 * * *' # Run daily

jobs:
  detect-drift:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout Repository
        uses: actions/checkout@v3

      - name: Setup Python
        uses: actions/setup-python@v4
        with:
          python-version: 3.8

      - name: Install Dependencies
        run: |
          python -m venv env
          source env/bin/activate
```

```
        pip install -r requirements.txt

    - name: Run Drift Detection
      run: |
        python drift_detection.py

    - name: Check Drift Report
      id: check_drift
      run: |
        if grep -q "data_drift" drift_report.json; then
          echo "Drift detected"
          echo "drift=true" >> $GITHUB_ENV
        else
          echo "No drift detected"
        fi

retrain:
  needs: detect-drift
  if: env.drift == 'true'
  runs-on: ubuntu-latest
  steps:
```

```yaml
- name: Checkout Repository
  uses: actions/checkout@v3

- name: Setup Python
  uses: actions/setup-python@v4
  with:
    python-version: 3.8

- name: Install Dependencies
  run: |
    python -m venv env
    source env/bin/activate
    pip install -r requirements.txt

- name: Retrain Model
  run: python train_model.py

- name: Commit & Push Model
  run: |
    git config --global user.name "GitHub Actions"
    git config --global user.email "actions@github.com"
```

```
git add model.pkl
git commit -m "Updated model due to drift"
git push
```

Step 5: Deploy Model as an API

We use **FastAPI** to serve the model.

api.py

python

```python
from fastapi import FastAPI
import joblib
import numpy as np

app = FastAPI()

model = joblib.load('model.pkl')

@app.post("/predict")
def predict(data: dict):
    features = np.array(data["features"]).reshape(1, -1)
```

```python
    prediction = model.predict(features).tolist()
    return {"prediction": prediction}
```

Run the API:

```
uvicorn api:app --host 0.0.0.0 --port 8000
```

Test using:

```
curl -X 'POST' 'http://127.0.0.1:8000/predict' -H 'Content-Type: application/json' -d '{"features": [0.1, 5.2, 1]}'
```

Step 6: Monitor with Prometheus and Grafana

monitor.py

python

```python
from prometheus_client import start_http_server, Gauge
import random
import time

drift_metric = Gauge('feature_drift', 'Feature drift detected')
```

```python
def monitor_drift():
    while True:
        drift_metric.set(random.choice([0, 1]))  # Simulated drift
        time.sleep(10)

if __name__ == "__main__":
    start_http_server(9090)
    monitor_drift()
```

Run:

```
python monitor.py
```

Add Prometheus configuration:

yaml

```yaml
scrape_configs:
  - job_name: 'ml-drift-monitor'
    static_configs:
      - targets: ['localhost:9090']
```

Run Prometheus:

prometheus --config.file=prometheus.yml

Conclusion

This **CI/CD pipeline for ML with feature drift detection** automates the entire process:

1. Detects feature drift using **Evidently AI**.
2. Retrains the model when drift is detected.
3. Deploys the updated model via **FastAPI**.
4. Uses **GitHub Actions** for automation.
5. Monitors drift with **Prometheus & Grafana**.

---

2. Model Monitoring and Optimization

Project 1. Drift Detection in ML Models: Implement a system that monitors model performance and triggers retraining if accuracy drops.

In machine learning, **drift** refers to changes in the data distribution over time, causing a model's performance to decline. Drift can be of three types:

- **Concept Drift**: The relationship between input features and target variable changes.
- **Data Drift**: The distribution of input features changes.
- **Model Drift**: A combination of both.

To handle drift, we monitor model accuracy and trigger retraining if performance drops below a threshold.

Project Setup

We'll use **Python, Scikit-learn, Pandas, NumPy, and Evidently AI** for monitoring.

Step 1: Install Required Libraries

Run the following command in the terminal:

pip install numpy pandas scikit-learn evidently matplotlib

Step 2: Load and Train an Initial Model

We use a synthetic dataset to train an initial model.

python

```
import numpy as np

import pandas as pd

from sklearn.model_selection import train_test_split

from sklearn.ensemble import RandomForestClassifier

from sklearn.metrics import accuracy_score


# Generate synthetic data

np.random.seed(42)

X = np.random.rand(1000, 5)

y = (X[:, 0] + X[:, 1] > 1).astype(int)  # Simple decision boundary
```

Split data

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

Train initial model

model = RandomForestClassifier(n_estimators=100, random_state=42)

model.fit(X_train, y_train)

Evaluate accuracy

y_pred = model.predict(X_test)

initial_accuracy = accuracy_score(y_test, y_pred)

print(f"Initial Model Accuracy: {initial_accuracy:.2f}")

🔹 **Explanation:**

- Creates synthetic data
- Splits data into training and testing
- Trains a RandomForest model
- Evaluates initial accuracy

Step 3: Simulating Drift

We simulate drift by modifying the test dataset.

python

 # Simulate data drift by changing feature distribution

X_drifted = X_test.copy()

X_drifted[:, 0] += 0.5 # Shift first feature

 # Evaluate accuracy after drift

y_pred_drifted = model.predict(X_drifted)

new_accuracy = accuracy_score(y_test, y_pred_drifted)

print(f"New Accuracy After Drift: {new_accuracy:.2f}")

🔹 **Explanation:**

- Alters feature distribution (simulating real-world drift)
- Checks how accuracy is affected

Step 4: Detecting Drift with Evidently AI

We use **Evidently AI** to visualize and monitor drift.

python

from evidently.report import Report

from evidently.metrics import DatasetDriftMetric, DataDriftTable

 # Create a report

report = Report(metrics=[DatasetDriftMetric(), DataDriftTable()])

report.run(reference_data=pd.DataFrame(X_test), current_data=pd.DataFrame(X_drifted))

 # Display the drift report

report.show(mode="inline")

🔹 **Explanation:**

- Uses Evidently AI to check for **dataset drift**
- Compares original vs. drifted dataset
- Displays drift report

Step 5: Automating Retraining

If accuracy drops below **90%**, we retrain the model.

python

```
if new_accuracy < 0.90:

    print("Drift detected! Retraining model...")

    model.fit(X_train, y_train)

    y_pred_retrained = model.predict(X_test)

    retrained_accuracy = accuracy_score(y_test, y_pred_retrained)

    print(f"Retrained Model Accuracy: {retrained_accuracy:.2f}")
```

else:

    print("No drift detected. Model remains the same.")

🔹 **Explanation:**

- If drift occurs, retrains the model
- Checks **new accuracy**

Step 6: Automating with a Pipeline

We can schedule this script with **cron jobs or Airflow** to run periodically.

python drift_detection.py

To run it every hour using **cron**:

crontab -e

Add:

0 * * * * /usr/bin/python3 /path/to/drift_detection.py

Final Thoughts

✅ We built a **drift detection system**
✅ Monitored **model performance**
✅ Triggered **retraining** if drift was detected

Project 2. Automated Model Retraining in Production: Use Apache Airflow to schedule model retraining based on new data.

In machine learning, models degrade over time as new data becomes available. Automated model retraining ensures that models remain accurate without manual intervention. **Apache Airflow**, an open-source workflow orchestration tool, helps in scheduling and managing this retraining process efficiently.

In this project, we will:

- Use **Apache Airflow** to schedule and automate model retraining.
- Load new data, preprocess it, retrain the model, evaluate performance, and update it.
- Store the model in a version-controlled system.

Step-by-Step Implementation

Step 1: Set Up Environment

Install dependencies

pip install apache-airflow pandas scikit-learn joblib

Step 2: Initialize Apache Airflow

Set up Airflow's working directory:

export AIRFLOW_HOME=~/airflow

airflow db init

Create an admin user:

airflow users create --username admin --firstname Admin --lastname User --role Admin --email admin@example.com

Start the webserver and scheduler:

airflow webserver --port 8080 &

airflow scheduler &

Step 3: Create the Airflow DAG (Directed Acyclic Graph)

A DAG defines the retraining pipeline steps.

Code: ml_retraining_dag.py

python

```
from airflow import DAG

from airflow.operators.python import PythonOperator

from datetime import datetime, timedelta

import pandas as pd

import joblib

from sklearn.model_selection import train_test_split

from sklearn.ensemble import RandomForestClassifier

from sklearn.metrics import accuracy_score
```

```python
# Define default arguments
default_args = {
    "owner": "airflow",
    "depends_on_past": False,
    "start_date": datetime(2024, 2, 10),
    "retries": 1,
    "retry_delay": timedelta(minutes=5),
}

# Define DAG
dag = DAG(
    "ml_model_retraining",
    default_args=default_args,
    schedule_interval="@daily",  # Runs every day
    catchup=False,
)

# Load data function
def load_data():
    df = pd.read_csv("data.csv")  # Load dataset
    df.to_csv("/tmp/data.csv", index=False)
```

 # Preprocess data function

```
def preprocess_data():
    df = pd.read_csv("/tmp/data.csv")
    df = df.dropna()  # Handle missing values
    df.to_csv("/tmp/preprocessed_data.csv", index=False)
```

 # Train model function

```
def train_model():
    df = pd.read_csv("/tmp/preprocessed_data.csv")
    X = df.drop(columns=["target"])  # Features
    y = df["target"]  # Labels
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

    model = RandomForestClassifier(n_estimators=100, random_state=42)
    model.fit(X_train, y_train)

    y_pred = model.predict(X_test)
    accuracy = accuracy_score(y_test, y_pred)
    print(f"Model Accuracy: {accuracy:.2f}")
```

joblib.dump(model, "/tmp/model.pkl") # Save the model

 # Define tasks

load_data_task = PythonOperator(task_id="load_data", python_callable=load_data, dag=dag)

preprocess_data_task = PythonOperator(task_id="preprocess_data", python_callable=preprocess_data, dag=dag)

train_model_task = PythonOperator(task_id="train_model", python_callable=train_model, dag=dag)

 # Define task dependencies

load_data_task >> preprocess_data_task >> train_model_task

Step 4: Deploy the DAG

Save the DAG in the Airflow DAGs folder:

mkdir -p ~/airflow/dags

mv ml_retraining_dag.py ~/airflow/dags/

Restart Airflow scheduler to detect the new DAG:

airflow scheduler restart

Go to **http://localhost:8080** in your browser, enable the DAG, and trigger it manually.

Code Explanation

1. Apache Airflow DAG Structure

- **DAG**: Defines the pipeline, specifying tasks and scheduling.
- **Operators**: Tasks in the workflow (PythonOperator runs Python functions).
- **Dependencies (>>)**: Defines execution order.

2. ML Retraining Steps

- **Load Data (load_data)**: Reads and stores new data.
- **Preprocess Data (preprocess_data)**: Cleans the dataset.
- **Train Model (train_model)**: Splits data, trains a model, evaluates accuracy, and saves it.

Conclusion

This project automates machine learning model retraining using Apache Airflow. The scheduled DAG ensures models remain updated with new data. This setup can be extended with **feature engineering**, **hyperparameter tuning**, or **model deployment** using services like **AWS S3, MLflow, or Docker**.

Project 3. AI-Based Model Staleness Detection: Monitor ML models for concept drift and trigger automatic retraining.

Machine learning models may become **stale** over time due to **concept drift**, which occurs when the statistical properties of incoming data change. This can lead to poor model performance.

To solve this, we'll build a **Model Staleness Detection System** that:

- **Monitors incoming data** for drift.
- **Detects performance degradation** using metrics.
- **Triggers retraining** when needed.

This system is useful in **real-time applications** like fraud detection, recommendation systems, and stock market predictions.

📌 Project Workflow

1. Train a baseline ML model.
2. Monitor model performance on incoming data.
3. Detect drift using statistical tests.
4. Retrain the model if drift is detected.
5. Deploy the updated model.

📌 Tech Stack

- Python
- Scikit-learn (ML Model)
- NumPy & Pandas (Data Handling)
- SciPy (Statistical Tests)
- Flask (API for Model Monitoring)
- Docker (Containerization)
- Jenkins (CI/CD for Auto Retraining)

📌 Step-by-Step Guide

1️⃣ Set Up the Environment

Install the required libraries:

pip install numpy pandas scikit-learn scipy flask joblib requests

---

2 Train an Initial Model

Create a file train_model.py to train and save a simple **Random Forest Classifier**.

python

```
import numpy as np

import pandas as pd

import joblib

from sklearn.ensemble import RandomForestClassifier

from sklearn.model_selection import train_test_split

from sklearn.metrics import accuracy_score


# Load dataset (Iris dataset as an example)

from sklearn.datasets import load_iris

data = load_iris()

X, y = data.data, data.target


# Split data

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

Train model

model = RandomForestClassifier(n_estimators=100, random_state=42)

model.fit(X_train, y_train)

Save model

joblib.dump(model, "model.pkl")

Evaluate

y_pred = model.predict(X_test)

print(f"Initial Model Accuracy: {accuracy_score(y_test, y_pred):.2f}")

✔️ This code trains a model and saves it as model.pkl.

3️⃣ Create a Flask API for Model Predictions

Create a app.py file to serve predictions.

python

from flask import Flask, request, jsonify

import joblib

import numpy as np

```python
app = Flask(__name__)


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


@app.route("/predict", methods=["POST"])
def predict():
    data = request.json
    X = np.array(data["features"]).reshape(1, -1)
    prediction = model.predict(X).tolist()
    return jsonify({"prediction": prediction})


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

Run the API:

```
python app.py
```

Test it with **Postman** or:

```
curl -X POST "http://localhost:5000/predict" -H "Content-Type: application/json"
-d '{"features": [5.1, 3.5, 1.4, 0.2]}'
```

✔️ This exposes an API to make predictions.

4️⃣ Detect Concept Drift

Create a detect_drift.py file to **compare old vs. new data distributions** using the **Kolmogorov-Smirnov (KS) test**.

python

```python
import numpy as np

import pandas as pd

from scipy.stats import ks_2samp


# Load the reference dataset

from sklearn.datasets import load_iris

data = load_iris()

X_ref = data.data  # Reference data


def detect_drift(new_data):

    drift_detected = False

    new_data = np.array(new_data)


    for i in range(X_ref.shape[1]):

        stat, p_value = ks_2samp(X_ref[:, i], new_data[:, i])
```

```python
        if p_value < 0.05:  # If p-value is low, data distribution has changed
            drift_detected = True
            break
    return drift_detected
```

 # Example usage

```python
new_data = np.random.rand(10, 4) * 10  # Simulated new data
print("Drift Detected:", detect_drift(new_data))
```

✔️ This script detects drift by **comparing distributions**.

5️⃣ Automate Model Retraining

Modify train_model.py to retrain the model if drift is detected.

python

```python
def retrain_model():
    print("Retraining model...")
    model.fit(X, y)
    joblib.dump(model, "model.pkl")
    print("Model retrained and saved!")
```

Example

if detect_drift(new_data):

    retrain_model()

✔️ If drift is detected, the model **automatically retrains**.

6️⃣ Automate with Jenkins

Create a Jenkinsfile for **CI/CD retraining**:

groovy

```
pipeline {
    agent any
    stages {
        stage('Check for Drift') {
            steps {
                sh 'python detect_drift.py'
            }
        }
        stage('Retrain Model') {
            when {
                expression { return sh(script: 'python detect_drift.py', returnStdout: true).contains("Drift Detected: True") }
```

```
            }
            steps {
                sh 'python train_model.py'
            }
        }
        stage('Deploy API') {
            steps {
                sh 'docker build -t model-api .'
                sh 'docker run -d -p 5000:5000 model-api'
            }
        }
    }
}
```

✔️ **Automatically detects drift, retrains, and redeploys** the model.

7️⃣ Deploy with Docker

Create a Dockerfile:

dockerfile

```
FROM python:3.9
```

```
WORKDIR /app

COPY . /app

RUN pip install -r requirements.txt

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

EXPOSE 5000
```

Build and run the container:

```
docker build -t model-api .

docker run -p 5000:5000 model-api
```

✔️ **Deploys the model as a containerized API**.

📌 Summary

✅ Trains an initial **ML model**.
✅ Creates a **Flask API** for predictions.
✅ Implements **concept drift detection**.
✅ Triggers **automatic retraining**.
✅ Uses **Jenkins for CI/CD**.
✅ **Deploys with Docker**.

Project 4. AutoML Pipeline for Hyperparameter Tuning: Build an automated training pipeline that finds the best ML model configuration.

Project Overview

AutoML Pipeline for Hyperparameter Tuning
This project automates the process of selecting the best machine learning model and optimizing its hyperparameters. It uses Optuna for hyperparameter tuning and scikit-learn for model training and evaluation.

Key Features

- Automatically selects the best ML model
- Performs hyperparameter tuning using Optuna
- Uses cross-validation to prevent overfitting
- Supports multiple ML algorithms

Step 1: Install Required Libraries

Run the following command to install the necessary Python libraries:

```
pip install numpy pandas scikit-learn optuna
```

Step 2: Import Libraries

python

```
import numpy as np

import pandas as pd

import optuna

from sklearn.model_selection import train_test_split, cross_val_score
```

from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier

from sklearn.svm import SVC

from sklearn.metrics import accuracy_score

from sklearn.datasets import load_iris

Step 3: Load Dataset

We will use the Iris dataset for this project.

python

 # Load dataset

data = load_iris()

X = pd.DataFrame(data.data, columns=data.feature_names)

y = data.target

 # Split into training and testing sets

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

Step 4: Define the Objective Function

Optuna optimizes this function to find the best model and hyperparameters.

python

```
def objective(trial):
    model_type = trial.suggest_categorical("model", ["RandomForest",
"GradientBoosting", "SVC"])

    if model_type == "RandomForest":
        n_estimators = trial.suggest_int("n_estimators", 50, 200)
        max_depth = trial.suggest_int("max_depth", 3, 20)
        model = RandomForestClassifier(n_estimators=n_estimators,
max_depth=max_depth, random_state=42)

    elif model_type == "GradientBoosting":
        n_estimators = trial.suggest_int("n_estimators", 50, 200)
        learning_rate = trial.suggest_loguniform("learning_rate", 0.01, 0.3)
        model = GradientBoostingClassifier(n_estimators=n_estimators,
learning_rate=learning_rate, random_state=42)

    elif model_type == "SVC":
        C = trial.suggest_loguniform("C", 0.1, 10)
        kernel = trial.suggest_categorical("kernel", ["linear", "rbf", "poly"])
```

    model = SVC(C=C, kernel=kernel, random_state=42)

  # Perform cross-validation

  score = cross_val_score(model, X_train, y_train, cv=5, scoring="accuracy").mean()

  return score

Step 5: Run the Optuna Hyperparameter Optimization

python

 # Create study and optimize

study = optuna.create_study(direction="maximize")

study.optimize(objective, n_trials=20)

 # Print the best parameters

print("Best model and parameters:", study.best_params)

Step 6: Train the Best Model on Full Training Data

python

best_params = study.best_params

best_model = None

if best_params["model"] == "RandomForest":

    best_model = RandomForestClassifier(n_estimators=best_params["n_estimators"], max_depth=best_params["max_depth"], random_state=42)

elif best_params["model"] == "GradientBoosting":

    best_model = GradientBoostingClassifier(n_estimators=best_params["n_estimators"], learning_rate=best_params["learning_rate"], random_state=42)

elif best_params["model"] == "SVC":

    best_model = SVC(C=best_params["C"], kernel=best_params["kernel"], random_state=42)

best_model.fit(X_train, y_train)

Step 7: Evaluate the Model

python

y_pred = best_model.predict(X_test)

accuracy = accuracy_score(y_test, y_pred)

print(f"Test Accuracy: {accuracy:.4f}")

Step 8: Save and Load the Model

python

import joblib

 # Save the model

joblib.dump(best_model, "best_model.pkl")

 # Load the model

loaded_model = joblib.load("best_model.pkl")

Explanation of the Code

1. **Load dataset** → We use load_iris() to get sample data.
2. **Split into train/test sets** → train_test_split() ensures our model is trained and tested separately.
3. **Define an objective function** → This function is optimized by Optuna to find the best model and parameters.
4. **Run Optuna optimization** → It tests different configurations to find the best model.

5. **Train the best model** → The best configuration is used to train the final model.
6. **Evaluate the model** → We check its accuracy using accuracy_score().
7. **Save and load the model** → We use joblib.dump() and joblib.load() to store the trained model.

Summary

- This pipeline selects the best model and hyperparameters using **Optuna**.
- Supports **RandomForest, GradientBoosting, and SVC** models.
- Uses **cross-validation** to prevent overfitting.
- The trained model is **saved for future use**.

Project 5. Smart Hyperparameter Tuning with Reinforcement Learning: Use RL to optimize ML model parameters dynamically.

Hyperparameter tuning is crucial for optimizing machine learning (ML) models. Traditional methods like grid search and random search can be inefficient. In this project, we will use **Reinforcement Learning (RL)** to dynamically optimize ML model parameters.

Instead of manually testing different hyperparameters, we train an RL agent to find the best combination of parameters based on model performance. This project is useful for **automated ML workflows** and improves accuracy while reducing computational costs.

We will use:

- **OpenAI Gym**: RL environment
- **Stable-Baselines3**: RL training framework
- **Scikit-Learn**: ML model
- **Optuna**: Hyperparameter optimization

Step 1: Set Up the Environment

Install the required dependencies:

pip install numpy pandas scikit-learn stable-baselines3 gym optuna matplotlib

---

Step 2: Import Libraries

python

```
import gym
import numpy as np
import optuna
import pandas as pd
import matplotlib.pyplot as plt
from stable_baselines3 import PPO
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score
```

Step 3: Load and Prepare Dataset

We use the **Iris dataset** for simplicity.

python

```
from sklearn.datasets import load_iris

# Load dataset
iris = load_iris()
X, y = iris.data, iris.target

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

Step 4: Define RL Environment

We create a **custom Gym environment** where RL will tune hyperparameters.

python

```
class HyperparameterTuningEnv(gym.Env):
    def __init__(self):
        super(HyperparameterTuningEnv, self).__init__()
```

```
        # Action space: Number of estimators (10 to 200) and max depth (1 to 20)
        self.action_space = gym.spaces.Box(low=np.array([10, 1]),
high=np.array([200, 20]), dtype=np.float32)

        # State space (Not used here but required)
        self.observation_space = gym.spaces.Discrete(1)

    def step(self, action):
        n_estimators = int(action[0])
        max_depth = int(action[1])

        # Train model with selected hyperparameters
        model = RandomForestClassifier(n_estimators=n_estimators,
max_depth=max_depth, random_state=42)
        model.fit(X_train, y_train)

        # Evaluate accuracy
        predictions = model.predict(X_test)
        accuracy = accuracy_score(y_test, predictions)

        # Reward function: Accuracy of model
        reward = accuracy * 100  # Scale reward
```

```
        return np.array([0]), reward, True, {}

    def reset(self):
        return np.array([0])
```

Step 5: Train RL Model to Optimize Hyperparameters

python

```
env = HyperparameterTuningEnv()
model = PPO("MlpPolicy", env, verbose=1)

# Train RL model
model.learn(total_timesteps=10000)
```

Step 6: Test the Trained RL Model

python

```
obs = env.reset()
```

action, _ = model.predict(obs)

print(f"Optimized Hyperparameters: n_estimators={int(action[0])}, max_depth={int(action[1])}")

Step 7: Evaluate Performance with Best Hyperparameters

python

best_n_estimators = int(action[0])

best_max_depth = int(action[1])

 # Train final model

final_model = RandomForestClassifier(n_estimators=best_n_estimators, max_depth=best_max_depth, random_state=42)

final_model.fit(X_train, y_train)

 # Test and evaluate

final_predictions = final_model.predict(X_test)

final_accuracy = accuracy_score(y_test, final_predictions)

print(f"Final Model Accuracy: {final_accuracy * 100:.2f}%")

Explanation of Code

1. **Dataset Loading**: We use the **Iris dataset** and split it into training and test sets.
2. **Custom Gym Environment**:
    - RL learns to choose n_estimators (number of trees) and max_depth (tree depth).
    - It trains a RandomForest model with selected parameters.
    - The accuracy score is used as a **reward**.
3. **Training RL Model**:
    - We use **Proximal Policy Optimization (PPO)** from Stable-Baselines3.
    - The RL agent explores different hyperparameters and learns the best values.
4. **Testing RL Model**:
    - The trained model suggests the best n_estimators and max_depth.
    - We use these hyperparameters to train the final **RandomForestClassifier** and check accuracy.

Conclusion

This project shows how **Reinforcement Learning (RL) can automate hyperparameter tuning** for ML models. Instead of manually trying different values, RL finds the best configuration, leading to **better accuracy with minimal effort**.

Project 6. AutoML Pipeline for Continuous Model Optimization: Automate the process of selecting the best ML models.

Machine Learning (ML) models need continuous tuning and selection of the best performing model. **AutoML (Automated Machine Learning)** simplifies this by automatically selecting the best model, optimizing hyperparameters, and retraining models with new data.

In this project, we will build an **AutoML Pipeline** that:

- Loads and preprocesses data
- Selects the best ML model using **Auto-Sklearn**
- Optimizes hyperparameters
- Continuously updates the model when new data is available

We will use **Python, Scikit-learn, Auto-Sklearn, and Flask** for deployment.

Step 1: Install Dependencies

pip install numpy pandas scikit-learn auto-sklearn flask

Step 2: Load and Preprocess Data

We will use the **Iris dataset** for simplicity.

Create automl_pipeline.py

python

```
import numpy as np

import pandas as pd

from sklearn.model_selection import train_test_split

from sklearn.datasets import load_iris

import autosklearn.classification
```

```python
# Load dataset

iris = load_iris()

X, y = iris.data, iris.target


# Split dataset

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


# Initialize AutoML

automl =
autosklearn.classification.AutoSklearnClassifier(time_left_for_this_task=120,
per_run_time_limit=30)


# Train model

automl.fit(X_train, y_train)


# Evaluate model

accuracy = automl.score(X_test, y_test)

print(f"Best Model Accuracy: {accuracy:.2f}")


# Save best model

import joblib
```

joblib.dump(automl, "best_automl_model.pkl")

Explanation

- Loads **Iris dataset** from Scikit-learn
- Splits data into training (80%) and testing (20%)
- Uses **Auto-Sklearn** to find the best ML model
- Evaluates the model on test data
- Saves the best model for future use

Step 3: Deploy as an API

Create app.py

python

from flask import Flask, request, jsonify

import joblib

import numpy as np

app = Flask(__name__)

 # Load trained model

model = joblib.load("best_automl_model.pkl")

```python
@app.route("/predict", methods=["POST"])
def predict():
    data = request.get_json()
    features = np.array(data["features"]).reshape(1, -1)
    prediction = model.predict(features)
    return jsonify({"prediction": int(prediction[0])})


if __name__ == "__main__":
    app.run(debug=True)
```

Explanation

- Uses **Flask** to create an API
- Loads the **best AutoML model**
- Accepts new data via a **POST request** and predicts the class

---

Step 4: Run the API

```
python app.py
```

Now, send a test request using Postman or curl:

```
curl -X POST http://127.0.0.1:5000/predict -H "Content-Type: application/json" -d '{"features": [5.1, 3.5, 1.4, 0.2]}'
```

Step 5: Automate Continuous Training

When new data is available, the pipeline should **retrain the model**.

Create update_model.py

python

```
import joblib

import numpy as np

import pandas as pd

from sklearn.model_selection import train_test_split

from sklearn.datasets import load_iris

import autosklearn.classification


# Load new dataset (simulate new data)

iris = load_iris()

X, y = iris.data, iris.target


# Split dataset

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

 # Load existing model

automl = joblib.load("best_automl_model.pkl")

 # Retrain model with new data

automl.fit(X_train, y_train)

 # Save updated model

joblib.dump(automl, "best_automl_model.pkl")

print("Model updated successfully.")

Run Continuous Model Update

python update_model.py

Step 6: Automate with Cron Job (Linux)

crontab -e

Add this line to retrain every day at midnight:

ruby

0 0 * * * /usr/bin/python3 /path/to/update_model.py

Final Summary

1. **Automates ML model selection** using AutoML.
2. **Deploys an API** for real-time predictions.
3. **Continuously updates the model** when new data is available.
4. **Automates retraining with a cron job**.

---

3. Model Explainability and Fairness

Project 1. Explainable AI (XAI) in MLOps: Develop a framework that provides explainability for ML models deployed in production.

Machine Learning (ML) models are often seen as black boxes, making it difficult to understand their predictions. Explainable AI (XAI) aims to provide transparency, interpretability, and trust in ML models.

In this project, we will integrate XAI into an MLOps pipeline, ensuring that deployed models provide explanations for their predictions. We will use tools like **SHAP (SHapley Additive exPlanations)** and **LIME (Local Interpretable Model-agnostic Explanations)** to explain model predictions.

The project follows an MLOps approach by incorporating **CI/CD, model training, monitoring, and explainability** within a containerized environment.

Step-by-Step Guide

1. Prerequisites

Ensure you have the following installed:

- Python 3.8+
- Docker & Docker Compose

- Git
- Kubernetes (kind or Minikube)
- Jenkins / GitHub Actions for CI/CD

Install necessary Python libraries:

pip install pandas numpy scikit-learn shap lime flask gunicorn

---

2. Project Structure

```
xai-mlops-project/
│── data/               # Dataset
│── models/             # Trained models
│── src/
│   ├── train.py        # Model training
│   ├── explain.py      # Explainability
│   ├── app.py          # API to serve predictions
│── Dockerfile          # Containerization
│── docker-compose.yml  # Running in a container
│── requirements.txt    # Python dependencies
│── Jenkinsfile         # CI/CD Pipeline
```

---

3. Load Dataset & Train Model

Create train.py to train a simple ML model.

python

```
import pandas as pd

import numpy as np

from sklearn.model_selection import train_test_split

from sklearn.ensemble import RandomForestClassifier

from sklearn.metrics import accuracy_score

import joblib


# Load dataset (Example: Titanic dataset)

data = pd.read_csv("data/titanic.csv")


# Preprocessing

data = data.dropna()  # Drop missing values

X = data[['Pclass', 'Age', 'Fare']]  # Features

y = data['Survived']  # Target


# Split data

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

 # Train model

model = RandomForestClassifier(n_estimators=100, random_state=42)

model.fit(X_train, y_train)

 # Evaluate

predictions = model.predict(X_test)

accuracy = accuracy_score(y_test, predictions)

print(f"Model Accuracy: {accuracy}")

 # Save model

joblib.dump(model, "models/model.pkl")

Run the training script:

python src/train.py

---

4. Explain Model Predictions

Create explain.py to provide explanations using SHAP.

python

import shap

import joblib

import pandas as pd

 # Load model & data

model = joblib.load("models/model.pkl")

data = pd.read_csv("data/titanic.csv")

data = data.dropna()[['Pclass', 'Age', 'Fare']]

 # Use SHAP for Explainability

explainer = shap.Explainer(model)

shap_values = explainer(data)

 # Visualize feature importance

shap.summary_plot(shap_values, data)

Run the script:

python src/explain.py

---

5. Deploy Model with API

Create app.py to serve predictions via a REST API.

python

```python
from flask import Flask, request, jsonify
import joblib
import pandas as pd

app = Flask(__name__)
model = joblib.load("models/model.pkl")

@app.route('/predict', methods=['POST'])
def predict():
    data = request.get_json()
    df = pd.DataFrame(data)
    predictions = model.predict(df)
    return jsonify({"predictions": predictions.tolist()})

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

Run the API:

```
python src/app.py
```

---

6. Dockerize the Application

Create a Dockerfile:

dockerfile

```
FROM python:3.8

WORKDIR /app

COPY requirements.txt .

RUN pip install -r requirements.txt

COPY src/ .

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

Build and run the Docker container:

```
docker build -t xai-mlops .

docker run -p 5000:5000 xai-mlops
```

---

7. Set Up CI/CD with Jenkins

Create Jenkinsfile for automation:

groovy

```
pipeline {
```

```
    agent any
    stages {
        stage('Build') {
            steps {
                sh 'docker build -t xai-mlops .'
            }
        }
        stage('Test') {
            steps {
                sh 'pytest tests/'
            }
        }
        stage('Deploy') {
            steps {
                sh 'docker run -d -p 5000:5000 xai-mlops'
            }
        }
    }
}
```

---

8. Deploy on Kubernetes

Create a deployment YAML (deployment.yaml):

yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: xai-mlops
spec:
  replicas: 2
  selector:
    matchLabels:
      app: xai-mlops
  template:
    metadata:
      labels:
        app: xai-mlops
    spec:
      containers:
      - name: xai-mlops
        image: xai-mlops:latest
```

ports:

- containerPort: 5000

Deploy on Kubernetes:

kubectl apply -f deployment.yaml

Code Explanation

- train.py → Loads dataset, preprocesses data, trains a **RandomForestClassifier**, and saves the model.
- explain.py → Uses **SHAP** to explain feature importance in model predictions.
- app.py → A **Flask API** to serve predictions via HTTP requests.
- Dockerfile → Packages the application into a **Docker container**.
- Jenkinsfile → Automates build, test, and deployment using **Jenkins**.
- deployment.yaml → Defines Kubernetes deployment.

Conclusion

This project integrates **MLOps with Explainable AI**, ensuring model predictions are transparent. It follows best practices by:

- Using **SHAP** for interpretability
- Deploying the model via **Flask API**
- Containerizing with **Docker**
- Automating deployment using **CI/CD**
- Running in **Kubernetes**

Project 2. Automated Bias Detection in ML Models: Implement fairness testing in MLOps pipelines to detect biased predictions.

Objective

Bias in Machine Learning models can lead to unfair and discriminatory predictions, affecting real-world applications like hiring, lending, and law enforcement. In this project, we will integrate **fairness testing in an MLOps pipeline** to **detect biased predictions** and ensure model fairness before deployment.

Step 1: Setup the Environment

1. Install Required Libraries

Ensure you have Python installed. Then, install the necessary libraries:

pip install pandas numpy scikit-learn aif360 mlflow flask

2. Create Project Structure

mkdir ml_bias_detection && cd ml_bias_detection

mkdir data src models

touch src/train.py src/bias_check.py src/app.py

Step 2: Data Collection

We will use the **Adult Income Dataset** from UCI, a common dataset for fairness testing.

python

```
import pandas as pd
from sklearn.model_selection import train_test_split

# Load dataset
url = "https://archive.ics.uci.edu/ml/machine-learning-databases/adult/adult.data"
columns = ["age", "workclass", "fnlwgt", "education", "education-num", "marital-status",
           "occupation", "relationship", "race", "sex", "capital-gain", "capital-loss",
           "hours-per-week", "native-country", "income"]

df = pd.read_csv(url, names=columns, na_values="?")
df.dropna(inplace=True)  # Remove missing values

# Convert categorical to numerical
df['income'] = df['income'].apply(lambda x: 1 if x == " >50K" else 0)

# Split data
X = df.drop(["income"], axis=1)
y = df["income"]
```

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

 # Save data

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)

print("Data Preprocessing Completed.")

Explanation:

- The dataset is cleaned, missing values removed, categorical data converted, and split into training/testing sets.

Step 3: Train a Machine Learning Model

python

from sklearn.ensemble import RandomForestClassifier

from sklearn.metrics import accuracy_score

import joblib

```
# Load data

X_train = pd.read_csv("data/X_train.csv")

X_test = pd.read_csv("data/X_test.csv")

y_train = pd.read_csv("data/y_train.csv").values.ravel()

y_test = pd.read_csv("data/y_test.csv").values.ravel()


# Train model

clf = RandomForestClassifier(n_estimators=100, random_state=42)

clf.fit(X_train, y_train)


# Save model

joblib.dump(clf, "models/model.pkl")


# Evaluate

y_pred = clf.predict(X_test)

acc = accuracy_score(y_test, y_pred)

print(f"Model Accuracy: {acc:.2f}")
```

Explanation:

- A **Random Forest classifier** is trained and saved as model.pkl.
- Accuracy is printed after testing.

Step 4: Detect Bias Using AI Fairness 360

We use **AIF360** to analyze bias in predictions based on **gender**.

python

```python
from aif360.datasets import BinaryLabelDataset
from aif360.metrics import ClassificationMetric
import numpy as np

# Load Data
X_test = pd.read_csv("data/X_test.csv")
y_test = pd.read_csv("data/y_test.csv").values.ravel()
clf = joblib.load("models/model.pkl")

# Make Predictions
y_pred = clf.predict(X_test)

# Convert data into AIF360 format
test_dataset = BinaryLabelDataset(favorable_label=1,
                                  unfavorable_label=0,
                                  df=pd.concat([X_test, pd.Series(y_test, name="income")], axis=1),
                                  label_names=["income"],
```

```
                        protected_attribute_names=["sex"])


pred_dataset = test_dataset.copy()

pred_dataset.labels = y_pred.reshape(-1, 1)


# Bias Metrics

metric = ClassificationMetric(test_dataset, pred_dataset,
privileged_groups=[{"sex": 1}], unprivileged_groups=[{"sex": 0}])

disparate_impact = metric.disparate_impact()

print(f"Disparate Impact Ratio: {disparate_impact:.2f}")


if disparate_impact < 0.8 or disparate_impact > 1.2:

    print("Bias Detected! Consider Mitigating it.")

else:

    print("No significant bias detected.")
```

Explanation:

- The model's predictions are analyzed using **Disparate Impact** (a fairness metric).
- If the ratio is below **0.8** or above **1.2**, bias is detected.

Step 5: Deploy Bias Detection as an API

python

```
from flask import Flask, request, jsonify
import joblib

app = Flask(__name__)
model = joblib.load("models/model.pkl")

@app.route("/predict", methods=["POST"])
def predict():
    data = request.json
    df = pd.DataFrame(data)
    pred = model.predict(df)
    return jsonify({"prediction": pred.tolist()})

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

Explanation:

- This Flask API accepts JSON input and returns predictions.

Run the API

python src/app.py

Test API with Curl

curl -X POST http://127.0.0.1:5000/predict -H "Content-Type: application/json" -d '{"age":[35],"sex":[1],"education-num":[13]}'

Step 6: Automate with MLflow & MLOps Pipeline

We can track the model training and bias detection using **MLflow**.

python

```
import mlflow

mlflow.set_experiment("Bias_Detection")
with mlflow.start_run():
    mlflow.log_metric("accuracy", acc)
    mlflow.log_metric("disparate_impact", disparate_impact)
    mlflow.sklearn.log_model(clf, "model")
    print("Logged in MLflow.")
```

Run MLflow UI

```
mlflow ui --host 0.0.0.0 --port 5001
```

Explanation:

- Tracks model accuracy and fairness metrics in **MLflow**.

Conclusion

✅ Collected and Preprocessed Data
✅ Trained ML Model
✅ Checked for Bias using AIF360
✅ Deployed Bias Detection API
✅ Integrated with MLflow for tracking

Project 3. AI-Powered Model Explainability Dashboard: Build an interactive dashboard using SHAP or LIME for model explainability.

Machine learning models are often treated as "black boxes," making it difficult to understand their decision-making process. **SHAP (SHapley Additive exPlanations)** and **LIME (Local Interpretable Model-agnostic Explanations)** are powerful tools that help explain model predictions. In this project, we will create an **interactive dashboard** to visualize model explanations using **Flask and Streamlit**.

Project Steps

1. Install Dependencies

Before starting, install the necessary Python libraries:

pip install pandas numpy scikit-learn shap lime matplotlib seaborn flask streamlit

2. Load Dataset and Train a Model

We will use the **Iris dataset** and train a **Random Forest classifier**.

python

```
import pandas as pd

import numpy as np

import shap

import lime.lime_tabular

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.model_selection import train_test_split

from sklearn.ensemble import RandomForestClassifier

from sklearn.preprocessing import LabelEncoder

from sklearn.datasets import load_iris


# Load dataset

iris = load_iris()

X = pd.DataFrame(iris.data, columns=iris.feature_names)

y = iris.target
```

Train-test split

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

Train model

model = RandomForestClassifier(n_estimators=100, random_state=42)

model.fit(X_train, y_train)

print("Model trained successfully!")

3. SHAP Explanation

SHAP helps in **global and local interpretability** of models.

python

explainer = shap.Explainer(model, X_train)

shap_values = explainer(X_test)

Plot summary

shap.summary_plot(shap_values, X_test)

4. LIME Explanation

LIME provides local explanations for individual predictions.

python

```
explainer = lime.lime_tabular.LimeTabularExplainer(
    X_train.values, feature_names=X_train.columns,
class_names=iris.target_names, mode="classification"
)
```

 # Explain a single instance

```
i = 5  # Index of the test sample
exp = explainer.explain_instance(X_test.iloc[i].values, model.predict_proba,
num_features=2)
```

 # Show explanation

```
exp.show_in_notebook()
```

5. Flask API for Model Predictions

We will create a **Flask API** that serves the model and explainability results.

python

```
from flask import Flask, request, jsonify
```

```
app = Flask(__name__)


@app.route('/predict', methods=['POST'])
def predict():
    data = request.json
    input_data = pd.DataFrame([data])
    prediction = model.predict(input_data)[0]

    # SHAP explanation
    shap_values = explainer(input_data)
    shap_contributions = {col: val for col, val in zip(X.columns, shap_values.values[0])}

    return jsonify({'prediction': int(prediction), 'shap_explanation': shap_contributions})


if __name__ == '__main__':
    app.run(debug=True)
```

6. Streamlit Dashboard

Now, let's create an **interactive dashboard** using **Streamlit**.

python

```python
import streamlit as st

st.title("AI Model Explainability Dashboard")

# User input form
st.sidebar.header("User Input Features")
sepal_length = st.sidebar.slider("Sepal Length", float(X_train["sepal length (cm)"].min()), float(X_train["sepal length (cm)"].max()))
sepal_width = st.sidebar.slider("Sepal Width", float(X_train["sepal width (cm)"].min()), float(X_train["sepal width (cm)"].max()))
petal_length = st.sidebar.slider("Petal Length", float(X_train["petal length (cm)"].min()), float(X_train["petal length (cm)"].max()))
petal_width = st.sidebar.slider("Petal Width", float(X_train["petal width (cm)"].min()), float(X_train["petal width (cm)"].max()))

input_features = [[sepal_length, sepal_width, petal_length, petal_width]]

# Prediction
prediction = model.predict(input_features)[0]
st.write(f"**Prediction:** {iris.target_names[prediction]}")
```

SHAP Explanation

shap_values = explainer(pd.DataFrame(input_features, columns=X.columns))

st.set_option('deprecation.showPyplotGlobalUse', False)

shap.summary_plot(shap_values, X.columns)

st.pyplot()

How to Run the Project?

Run the Flask API
python flask_app.py

Run the Streamlit Dashboard
streamlit run streamlit_dashboard.py

Code Explanation

- **Data Preparation**: We load the Iris dataset and train a **RandomForestClassifier**.
- **SHAP & LIME**:
  - SHAP is used for a **global** view of feature importance.
  - LIME provides **local** explanations for individual predictions.
- **Flask API**: Accepts input data, predicts the class, and returns SHAP values.
- **Streamlit Dashboard**: Creates an **interactive UI** where users can input values and see model explanations.

Project 4. AI-Based Model Interpretability & Bias Detection: Implement SHAP or LIME to analyze model decisions and detect bias.

Machine learning models often act as "black boxes," making it difficult to understand how they arrive at their decisions. **Model interpretability** helps uncover these decision-making processes, which is crucial for **trust, fairness, and debugging biases** in AI models.

Two popular techniques for interpretability are:

- **SHAP (SHapley Additive Explanations):** Based on cooperative game theory, it assigns importance scores to features.
- **LIME (Local Interpretable Model-agnostic Explanations):** Approximates black-box model behavior using interpretable models (e.g., linear regression) for local explanations.

This project will use **SHAP** and **LIME** to analyze an AI model, detect biases, and explain predictions.

Project Steps

We'll implement SHAP and LIME for a **classification model** trained on the famous **Titanic dataset**.

Step 1: Install Required Libraries

pip install pandas numpy matplotlib seaborn shap lime scikit-learn

---

Step 2: Load and Preprocess the Data

python

import pandas as pd

import numpy as np

```
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.ensemble import RandomForestClassifier

# Load Titanic dataset
url = "https://raw.githubusercontent.com/datasciencedojo/datasets/master/titanic.csv"
df = pd.read_csv(url)

# Select relevant features
df = df[['Survived', 'Pclass', 'Sex', 'Age', 'SibSp', 'Parch', 'Fare', 'Embarked']]
df.dropna(inplace=True)  # Remove missing values

# Encode categorical variables
df['Sex'] = LabelEncoder().fit_transform(df['Sex'])
df['Embarked'] = LabelEncoder().fit_transform(df['Embarked'])

# Split dataset
X = df.drop('Survived', axis=1)
y = df['Survived']
```

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

 # Scale features

scaler = StandardScaler()

X_train = scaler.fit_transform(X_train)

X_test = scaler.transform(X_test)

 # Train model

model = RandomForestClassifier(n_estimators=100, random_state=42)

model.fit(X_train, y_train)

print("Model trained successfully!")

Step 3: Apply SHAP for Model Interpretability

python

import shap

 # Create SHAP explainer

explainer = shap.TreeExplainer(model)

shap_values = explainer.shap_values(X_test)

Plot feature importance

shap.summary_plot(shap_values[1], X_test, feature_names=X.columns)

Explanation

- The **summary plot** visualizes the impact of each feature on the model's predictions.
- **Positive values** indicate an increased likelihood of survival.
- **Negative values** indicate a decreased likelihood of survival.

Step 4: Apply LIME for Local Explanations

python

import lime

import lime.lime_tabular

 # Create LIME explainer

explainer = lime.lime_tabular.LimeTabularExplainer(X_train, feature_names=X.columns, class_names=['Died', 'Survived'], discretize_continuous=True)

 # Pick a sample for explanation

idx = 10

exp = explainer.explain_instance(X_test[idx], model.predict_proba)

 # Show explanation

exp.show_in_notebook()

Explanation

- LIME creates a **local interpretable model** (e.g., linear regression) to approximate the **black-box model** decision.
- It helps us understand which features contributed to a **specific prediction**.

Step 5: Detect Bias in Model Predictions

python

 # Check bias based on gender

X_test_gender = pd.DataFrame(X_test, columns=X.columns)

X_test_gender['Sex'] = 1 # Assume all passengers are female

pred_female = model.predict(X_test_gender)

X_test_gender['Sex'] = 0 # Assume all passengers are male

pred_male = model.predict(X_test_gender)

 # Compare survival rates

```
print(f"Predicted survival rate for females: {pred_female.mean():.2f}")

print(f"Predicted survival rate for males: {pred_male.mean():.2f}")
```

Explanation

- This test **modifies the "Sex" feature** and re-evaluates predictions.
- If the model predicts significantly **higher survival for females or males**, it may indicate **bias** in the dataset or model.

Conclusion

- **SHAP** provides **global explanations** (which features matter the most across all predictions).
- **LIME** gives **local explanations** (how individual predictions are made).
- **Bias detection** allows us to investigate fairness in AI models.

---

10. Data Versioning & Management

Project 1. Automated Data Versioning with DVC: Using Data Version Control (DVC) to track and manage datasets that are used to train models, ensuring reproducibility and version control of data.

Introduction

Data Version Control (DVC) is a tool that enables versioning for datasets and machine learning models, just like Git does for code. This project will demonstrate how to use DVC to track and manage datasets used in machine learning training, ensuring reproducibility and version control of data.

Project: Automated Data Versioning with DVC

Step 1: Install Required Tools

Ensure you have **Git, Python, and DVC** installed.

Install Git and Python (if not installed)

```
sudo apt update
sudo apt install git python3 python3-pip -y
```

Install DVC

```
pip install dvc
```

Step 2: Initialize a Git Repository

Create a new project folder and initialize Git.

```
mkdir dvc-project && cd dvc-project
git init
```

Step 3: Initialize DVC

```
dvc init
git commit -m "Initialize DVC in the project"
```

Step 4: Add a Dataset

Download or create a sample dataset. Here, we create a dummy CSV file.

```
mkdir data

echo "name,age,salary" > data/employees.csv

echo "Alice,25,50000" >> data/employees.csv

echo "Bob,30,60000" >> data/employees.csv

echo "Charlie,35,70000" >> data/employees.csv
```

Now, let's **track the dataset with DVC**:

```
dvc add data/employees.csv
```

DVC will create a .dvc file (data/employees.csv.dvc) to track dataset changes.

Commit changes to Git:

```
git add data/employees.csv.dvc .gitignore

git commit -m "Add dataset with DVC"
```

Step 5: Configure Remote Storage for DVC

DVC supports remote storage options like **Google Drive, AWS S3, or local storage**.

Using Local Storage (for simplicity)

```
mkdir ~/dvc-storage
dvc remote add myremote ~/dvc-storage
dvc remote default myremote
```

Push Dataset to Remote Storage

```
dvc push
```

Now, you can **remove the local dataset** and retrieve it whenever needed.

```
rm data/employees.csv  # Simulating dataset deletion
dvc pull  # Retrieves the dataset from storage
```

Step 6: Automate Dataset Versioning

Modify the dataset to track changes:

```
echo "David,40,80000" >> data/employees.csv
```

Track changes with DVC:

```
dvc add data/employees.csv
git commit -am "Updated dataset"
dvc push  # Save new version remotely
```

If you ever need an **older version**, use:

```
git checkout <commit-hash>
dvc checkout
```

Step 7: Use the Dataset in a Python Script

Create train.py to read the dataset:

python

```python
import pandas as pd

# Load dataset
data = pd.read_csv("data/employees.csv")

# Print dataset
print("Dataset Preview:")
print(data)

# Save processed data
processed_file = "data/processed.csv"
data.to_csv(processed_file, index=False)
```

print(f"Processed data saved at {processed_file}")

Run the script:

python3 train.py

Track processed data with DVC:

dvc add data/processed.csv

git commit -am "Add processed data"

dvc push

Step 8: Clone & Restore Data on a New System

On another machine, restore the project:

git clone <repo-url>

cd dvc-project

dvc pull # Retrieve datasets

python3 train.py # Process and verify

Explanation of Code

1. **dvc init** - Initializes DVC in the project.

2. **dvc add data/employees.csv** - Tracks the dataset.
3. **dvc remote add myremote ~/dvc-storage** - Sets up remote storage.
4. **dvc push & dvc pull** - Uploads/downloads datasets.
5. **Python Script (train.py)** - Reads and processes the dataset.

Conclusion

By using DVC, we:
✅ Track dataset versions automatically.
✅ Store datasets efficiently.
✅ Ensure reproducibility in ML projects.

Project 2. Data Quality Automation: Creating pipelines to automatically clean and validate datasets used for training, removing anomalies, duplicates, and correcting labels.

In machine learning and data analytics, poor-quality data leads to inaccurate models and unreliable insights. This project automates data cleaning and validation by building pipelines that:

- Remove anomalies and outliers
- Identify and remove duplicate records
- Correct mislabeled data
- Standardize formats for consistency

Using **Python, Pandas, NumPy, Scikit-learn, and Great Expectations**, we will automate these tasks and integrate them into a reproducible pipeline.

Project Setup

Step 1: Install Required Libraries

Run the following command to install the necessary libraries:

pip install pandas numpy scikit-learn great-expectations

Step 2: Load Sample Dataset

Create a file data.csv with some sample records:

cs

id,name,age,income,label

1,Alice,25,50000,verified

2,Bob,200,60000,fraud

3,Charlie,30,55000,verified

4,Alice,25,50000,verified

5,David,40,-2000,verified

6,Eve,35,70000,invalid

Python Code to Read Data

python

import pandas as pd

 # Load dataset

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

```
# Display first few rows
print(df.head())
```

Here, we load the dataset and display the first few rows to check its structure.

Step 3: Handle Anomalies and Outliers

Outliers can be detected using statistical methods. We remove unrealistic values for **age** and **income**.

python

```
# Remove unrealistic age values
df = df[(df['age'] > 0) & (df['age'] < 120)]

# Remove negative income
df = df[df['income'] > 0]

print(df)
```

This ensures **age** is within a valid range (0–120) and **income** is positive.

Step 4: Remove Duplicates

Duplicates can be removed using drop_duplicates().

python

 # Remove duplicate rows

df = df.drop_duplicates()

print(df)

Step 5: Correct Mislabeled Data

If labels contain typos or inconsistent values, we standardize them.

python

 # Correct labels

df['label'] = df['label'].replace({'fraud': 'unverified', 'invalid': 'unverified'})

print(df)

Here, we replace incorrect labels **"fraud"** and **"invalid"** with **"unverified"**.

Step 6: Validate Data Using Great Expectations

We use **Great Expectations** to define validation rules.

python

from great_expectations.dataset import PandasDataset

 # Convert DataFrame into a Great Expectations dataset

df_ge = PandasDataset(df)

 # Define validation rules

df_ge.expect_column_values_to_be_between("age", 0, 120)

df_ge.expect_column_values_to_not_be_null(["name", "income", "label"])

 # Run validation

results = df_ge.validate()

print(results)

This checks if **age** is within range, and ensures **name, income, and label** are not null.

Step 7: Automate with a Pipeline

We wrap everything into a function for automation.

python

def clean_data(file_path):

    df = pd.read_csv(file_path)

```
    # Remove anomalies
    df = df[(df['age'] > 0) & (df['age'] < 120)]
    df = df[df['income'] > 0]

    # Remove duplicates
    df = df.drop_duplicates()

    # Standardize labels
    df['label'] = df['label'].replace({'fraud': 'unverified', 'invalid': 'unverified'})

    return df

# Run the pipeline
cleaned_df = clean_data("data.csv")
print(cleaned_df)
```

This function loads data, cleans it, and returns a **validated** dataset.

Final Summary

- **Step 1:** Installed libraries

- **Step 2:** Loaded dataset
- **Step 3:** Removed anomalies
- **Step 4:** Removed duplicates
- **Step 5:** Corrected mislabeled data
- **Step 6:** Validated using **Great Expectations**
- **Step 7:** Automated the process with a function

Project 3. Data Drift Detection: Implementing systems to detect when the statistical properties of data change over time, which could impact model performance.

In machine learning, **data drift** occurs when the statistical properties of incoming data change over time compared to the data on which the model was trained. This can lead to performance degradation. Detecting data drift is essential to maintaining model accuracy and reliability.

Common causes of data drift include:

- Changes in user behavior
- Seasonal effects
- External events affecting data distributions

To monitor and detect data drift, we can use statistical tests and distance metrics like:

- **Kolmogorov-Smirnov Test** (KS Test)
- **Population Stability Index (PSI)**
- **Kullback-Leibler Divergence (KL Divergence)**

Project: Data Drift Detection

Step 1: Set Up the Environment

Install the required libraries:

pip install pandas numpy scikit-learn scipy matplotlib seaborn

---

Step 2: Prepare the Dataset

We'll simulate data drift by generating synthetic data with changing distributions.

Code: Generating Sample Data

python

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from scipy.stats import ks_2samp

 # Generate baseline (original) data

np.random.seed(42)

baseline_data = np.random.normal(loc=50, scale=10, size=1000) # Mean = 50, Std = 10

 # Generate new incoming data (simulating drift)

drifted_data = np.random.normal(loc=55, scale=12, size=1000) # Mean = 55, Std = 12

 # Convert to DataFrame

df_baseline = pd.DataFrame(baseline_data, columns=['value'])

df_drifted = pd.DataFrame(drifted_data, columns=['value'])

 # Plot distributions

plt.figure(figsize=(8, 5))

plt.hist(df_baseline['value'], bins=30, alpha=0.5, label='Baseline Data')

plt.hist(df_drifted['value'], bins=30, alpha=0.5, label='New Data (Potential Drift)')

plt.legend()

plt.title("Data Distribution Comparison")

plt.show()

🔹 **Explanation**

- We generate **baseline data** that represents the original training data distribution.
- We create **drifted data** with a slightly different mean and standard deviation to simulate real-world data drift.
- A histogram is used to visualize the distributions.

Step 3: Detect Data Drift Using KS Test

The **Kolmogorov-Smirnov (KS) test** compares the distributions of two datasets.

Code: KS Test Implementation

python

```python
# Perform KS Test
ks_stat, p_value = ks_2samp(df_baseline['value'], df_drifted['value'])

# Interpret the result
alpha = 0.05  # Significance level
if p_value < alpha:
    print(f"Data drift detected! KS Statistic: {ks_stat:.4f}, p-value: {p_value:.4f}")
else:
    print(f"No significant data drift. KS Statistic: {ks_stat:.4f}, p-value: {p_value:.4f}")
```

🔹 **Explanation**

- **KS Test** measures the difference between the cumulative distributions of two datasets.
- If the **p-value is below 0.05**, it means data drift is statistically significant.

Step 4: Implement Population Stability Index (PSI)

The **Population Stability Index (PSI)** helps detect data shifts by measuring differences in bin distributions.

Code: PSI Implementation

python

```
def calculate_psi(expected, actual, bins=10):
    # Create bin edges
    min_val = min(expected.min(), actual.min())
    max_val = max(expected.max(), actual.max())
    bin_edges = np.linspace(min_val, max_val, bins+1)

    # Calculate percentages per bin
    expected_counts, _ = np.histogram(expected, bins=bin_edges)
    actual_counts, _ = np.histogram(actual, bins=bin_edges)

    expected_perc = expected_counts / len(expected)
    actual_perc = actual_counts / len(actual)

    # Avoid division by zero
    expected_perc = np.where(expected_perc == 0, 0.0001, expected_perc)
    actual_perc = np.where(actual_perc == 0, 0.0001, actual_perc)

    # Calculate PSI
    psi_values = (expected_perc - actual_perc) * np.log(expected_perc /
actual_perc)
    psi_score = np.sum(psi_values)
    return psi_score
```

```
# Compute PSI
psi_score = calculate_psi(df_baseline['value'], df_drifted['value'])
print(f"PSI Score: {psi_score:.4f}")

# Interpretation
if psi_score > 0.25:
    print("Significant data drift detected.")
elif psi_score > 0.1:
    print("Moderate data drift detected.")
else:
    print("No significant data drift detected.")
```

🔹 **Explanation**

- **PSI** measures the divergence between expected and actual distributions.
- If **PSI > 0.25**, there is **significant data drift**.

Step 5: Automate Data Drift Monitoring

We can set up a scheduled job to continuously monitor drift in real-world scenarios.

Code: Automating Drift Detection

python

```
import time


def monitor_data_drift(baseline_data, new_data):
    while True:
        # Compute KS test
        ks_stat, p_value = ks_2samp(baseline_data, new_data)
        psi_score = calculate_psi(baseline_data, new_data)

        print(f"KS p-value: {p_value:.4f}, PSI: {psi_score:.4f}")

        if p_value < 0.05 or psi_score > 0.25:
            print("Alert: Significant data drift detected!")

        time.sleep(10)  # Check every 10 seconds
```

 # Simulate continuous monitoring

```
# monitor_data_drift(df_baseline['value'], df_drifted['value'])
```

🔹 **Explanation**

- The script continuously monitors for drift by running the **KS test** and **PSI** at regular intervals.
- If drift is detected, an **alert** is triggered.

Summary

✅ **Step 1**: Install required packages.
✅ **Step 2**: Generate synthetic baseline and drifted data.
✅ **Step 3**: Detect drift using **KS test**.
✅ **Step 4**: Use **PSI** to quantify drift.
✅ **Step 5**: Automate monitoring for real-world scenarios.

Project 4. Continuous Data Validation: Developing automated tools to validate incoming data in real-time and ensuring it adheres to the expected schema and quality standards before being fed into models.

Project Overview

Data validation is crucial in any data pipeline. If raw data contains incorrect or inconsistent values, it can lead to inaccurate results in machine learning models and decision-making. This project sets up an automated validation system using **Python, Pandera, FastAPI, and Kafka**, ensuring that incoming data follows predefined schema rules.

Technologies Used:

- **Python**: Core programming language
- **Pandera**: Data validation library
- **FastAPI**: API framework for real-time validation
- **Apache Kafka**: Message queue for streaming data
- **Docker**: Containerization for deployment
- **PostgreSQL**: Storing validated data

Step-by-Step Implementation

1. Setup the Environment

Install the necessary dependencies:

pip install fastapi uvicorn pandas pandera kafka-python psycopg2

---

2. Define the Data Schema with Pandera

Create a validation schema to check incoming data before processing.

python

```
import pandera as pa
from pandera.typing import Series


class DataSchema(pa.DataFrameModel):
    id: Series[int] = pa.Field(ge=1)  # ID should be >= 1
    name: Series[str] = pa.Field(str_length={"min_value": 1})  # Non-empty string
    age: Series[int] = pa.Field(ge=18, le=100)  # Age should be between 18-100
    email: Series[str] = pa.Field(str_matches=r"[^@]+@[^@]+\.[^@]+")  # Valid email format
```

---

3. Create a FastAPI Endpoint for Real-Time Validation

python

```
from fastapi import FastAPI, HTTPException
import pandas as pd
from schema import DataSchema

app = FastAPI()

@app.post("/validate/")
async def validate_data(data: list[dict]):
    df = pd.DataFrame(data)
    try:
        validated_df = DataSchema.validate(df)
        return {"message": "Data is valid", "validated_data":
validated_df.to_dict(orient="records")}
    except pa.errors.SchemaError as e:
        raise HTTPException(status_code=400, detail=str(e))
```

Run the API:

uvicorn main:app --reload

Test with:

```
curl -X 'POST' 'http://127.0.0.1:8000/validate/' -H 'Content-Type: application/json'
-d '[{"id":1,"name":"Alice","age":25,"email":"alice@example.com"}]'
```

---

4. Kafka for Real-Time Data Processing

Start Kafka (Docker Setup)

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

Kafka Producer (Sending Data for Validation)

python

```python
from kafka import KafkaProducer

import json


producer = KafkaProducer(bootstrap_servers='localhost:9092',
value_serializer=lambda v: json.dumps(v).encode('utf-8'))


data = {"id": 1, "name": "Alice", "age": 25, "email": "alice@example.com"}

producer.send("data_topic", value=data)
```

Kafka Consumer (Validating Data Before Storage)

python

```python
from kafka import KafkaConsumer

import json

import pandas as pd

from schema import DataSchema


consumer = KafkaConsumer("data_topic", bootstrap_servers='localhost:9092',
value_deserializer=lambda v: json.loads(v.decode('utf-8')))


for message in consumer:

    df = pd.DataFrame([message.value])

    try:

        validated_df = DataSchema.validate(df)

        print("Validated Data:", validated_df.to_dict(orient="records"))

    except Exception as e:

        print("Validation Failed:", str(e))
```

---

5. Store Validated Data in PostgreSQL

Install PostgreSQL and Create a Database

```
sudo apt update && sudo apt install postgresql postgresql-contrib

sudo -u postgres psql
```

Inside PostgreSQL:

CREATE DATABASE datavalidation;

CREATE TABLE validated_data (id SERIAL PRIMARY KEY, name TEXT, age INT, email TEXT);

Python Code to Store Validated Data

```
import psycopg2


def save_to_db(data):
    conn = psycopg2.connect("dbname=datavalidation user=postgres password=yourpassword")
    cur = conn.cursor()
    cur.execute("INSERT INTO validated_data (name, age, email) VALUES (%s, %s, %s)", (data["name"], data["age"], data["email"]))
    conn.commit()
    cur.close()
    conn.close()
```

Final Steps & Execution

Start the API:
uvicorn main:app --reload

- Start Kafka Producer and Consumer

- Ensure PostgreSQL is running and data is stored correctly

Project Summary

- **FastAPI** handles real-time validation
- **Pandera** ensures schema enforcement
- **Kafka** streams data in real-time
- **PostgreSQL** stores validated data

Project 5. Data Pipeline Monitoring: Implementing monitoring for data pipelines to ensure they run efficiently and consistently without failures, catching errors early.

1. Introduction

In any data-driven system, data pipelines play a crucial role in extracting, transforming, and loading (ETL) data. However, failures can occur due to issues like data inconsistencies, missing files, or infrastructure problems. Monitoring these pipelines helps detect failures early, optimize performance, and ensure data integrity.

In this project, we will set up monitoring for a data pipeline using **Apache Airflow**, **Prometheus**, and **Grafana**. We will:

- Build a simple ETL pipeline with Apache Airflow
- Monitor pipeline execution with logs and alerts
- Use Prometheus for collecting metrics
- Visualize metrics in Grafana

---

2. Project Setup & Prerequisites

Technologies Used

- **Apache Airflow** (Pipeline Orchestration)
- **Docker** (Containerization)
- **PostgreSQL** (Airflow Metadata Database)
- **Prometheus** (Metrics Collection)
- **Grafana** (Visualization & Alerting)

Prerequisites

- Install **Docker & Docker Compose**
- Basic understanding of Python

Install docker-compose if not available:
sudo apt update && sudo apt install docker-compose -y

---

3. Step-by-Step Implementation

Step 1: Set Up Apache Airflow using Docker

Create a project directory:

mkdir data-pipeline-monitoring && cd data-pipeline-monitoring

Create a docker-compose.yml file:

yaml

version: '3'

services:

```yaml
  postgres:
    image: postgres:13
    container_name: postgres_airflow
    environment:
      POSTGRES_USER: airflow
      POSTGRES_PASSWORD: airflow
      POSTGRES_DB: airflow
    ports:
      - "5432:5432"

  airflow-webserver:
    image: apache/airflow:2.6.3
    container_name: airflow_webserver
    depends_on:
      - postgres
    environment:
      AIRFLOW__CORE__EXECUTOR: LocalExecutor
      AIRFLOW__CORE__SQL_ALCHEMY_CONN:
postgresql+psycopg2://airflow:airflow@postgres/airflow
    ports:
      - "8080:8080"
    command: webserver
```

```yaml
  airflow-scheduler:
    image: apache/airflow:2.6.3
    container_name: airflow_scheduler
    depends_on:
      - airflow-webserver
    command: scheduler
```

Start the services:

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

Check Airflow UI at http://localhost:8080/ (default user: admin, pass: admin).

Step 2: Create a Simple ETL DAG in Airflow

Create a DAG file in dags/etl_pipeline.py:

python

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

```
def extract():
    print("Extracting data...")
    return {"data": random.randint(1, 100)}


def transform(ti):
    data = ti.xcom_pull(task_ids='extract')
    transformed_data = data["data"] * 10
    print(f"Transformed Data: {transformed_data}")
    return transformed_data


def load(ti):
    final_data = ti.xcom_pull(task_ids='transform')
    print(f"Loading Data: {final_data}")


default_args = {
    'owner': 'airflow',
    'start_date': datetime(2024, 2, 9),
    'retries': 1
}
```

```python
dag = DAG(
    'etl_pipeline',
    default_args=default_args,
    schedule_interval='@daily',
    catchup=False
)

extract_task = PythonOperator(task_id='extract', python_callable=extract, dag=dag)
transform_task = PythonOperator(task_id='transform', python_callable=transform, dag=dag)
load_task = PythonOperator(task_id='load', python_callable=load, dag=dag)

extract_task >> transform_task >> load_task
```

Restart Airflow:

```
docker-compose restart
```

Check DAG execution in Airflow UI.

Step 3: Set Up Monitoring with Prometheus & Grafana

Create a prometheus.yml file:

yaml

```yaml
global:
  scrape_interval: 15s

scrape_configs:
  - job_name: 'airflow'
    static_configs:
      - targets: ['airflow_webserver:8080']
```

Modify docker-compose.yml to add Prometheus & Grafana:

yaml

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

```yaml
grafana:
  image: grafana/grafana
  container_name: grafana
  ports:
    - "3000:3000"
  environment:
    - GF_SECURITY_ADMIN_PASSWORD=admin
```

Start Prometheus & Grafana:

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

- Access Prometheus at http://localhost:9090
- Access Grafana at http://localhost:3000 (user: admin, pass: admin)

---

4. Code Explanation

1. **Airflow DAG (etl_pipeline.py)**
   - The DAG defines **three tasks**: extract, transform, and load.
   - The xcom_pull function is used to pass data between tasks.
   - Tasks are executed sequentially (extract >> transform >> load).
2. **Docker Setup**
   - docker-compose.yml defines services for Airflow, PostgreSQL, Prometheus, and Grafana.
   - Each service runs in an isolated container but communicates via networking.
3. **Monitoring**
   - Prometheus scrapes metrics from Airflow.
   - Grafana visualizes these metrics.

- Alerts can be set in Grafana to notify failures.

---

5. Testing & Debugging

To check logs:

docker logs airflow_webserver -f

docker logs airflow_scheduler -f

To check active DAGs:

docker exec -it airflow_webserver airflow dags list

To manually trigger DAG:

docker exec -it airflow_webserver airflow dags trigger etl_pipeline

6. Conclusion

By the end of this project, we have:

- Built a basic ETL pipeline using Apache Airflow
- Integrated Prometheus to collect execution metrics
- Used Grafana to visualize and monitor pipeline performance
- Ensured real-time monitoring and alerting for failures

---

11. Model Monitoring Dashboards and Alerts

Project 1. End-to-End ML Monitoring Dashboard: Build a dashboard that tracks model performance, data drift, and system metrics.

Machine learning (ML) models in production can degrade over time due to data drift, concept drift, or system issues. A **Monitoring Dashboard** helps track **model performance, data drift, and system metrics** to maintain reliability.

In this project, we will:

- Train a sample ML model
- Deploy it using **Flask**
- Collect real-time predictions
- Use **Prometheus & Grafana** for monitoring
- Implement **Evidently AI** for data drift detection

Project Setup and Steps

1. Install Dependencies

 # Create and activate a virtual environment

python3 -m venv ml-monitoring-env

source ml-monitoring-env/bin/activate # On Windows: ml-monitoring-env\Scripts\activate

 # Install required libraries

pip install flask scikit-learn pandas numpy prometheus_client evidently requests matplotlib seaborn

---

2. Train and Save the ML Model

We'll train a simple **Logistic Regression** model on the **Iris dataset**.

Code: train_model.py

python

```
import pickle

import pandas as pd

import numpy as np

from sklearn.model_selection import train_test_split

from sklearn.linear_model import LogisticRegression

from sklearn.metrics import accuracy_score

from sklearn.datasets import load_iris


# Load dataset

iris = load_iris()

X = iris.data

y = iris.target


# Split data

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

 # Train model

model = LogisticRegression(max_iter=200)

model.fit(X_train, y_train)

 # Evaluate accuracy

y_pred = model.predict(X_test)

accuracy = accuracy_score(y_test, y_pred)

print(f"Model Accuracy: {accuracy:.2f}")

 # Save model

with open("iris_model.pkl", "wb") as f:

    pickle.dump(model, f)

Run the script:

python train_model.py

It will output accuracy and save the model as iris_model.pkl.

---

3. Build a Flask API to Serve the Model

We will create a Flask API that takes input data and returns predictions.

Code: app.py

python

```
import pickle
import numpy as np
from flask import Flask, request, jsonify
from prometheus_client import Counter, generate_latest, REGISTRY

# Load model
with open("iris_model.pkl", "rb") as f:
    model = pickle.load(f)

app = Flask(__name__)

# Define Prometheus metrics
prediction_counter = Counter("predictions_total", "Total number of predictions")

@app.route("/predict", methods=["POST"])
def predict():
    data = request.json["features"]
    prediction = model.predict([data]).tolist()

    # Increment metric counter
```

```
    prediction_counter.inc()


    return jsonify({"prediction": prediction})


@app.route("/metrics")
def metrics():
    return generate_latest(REGISTRY)


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

Run the API:

```
python app.py
```

- The API listens on port 5000.
- Send a test prediction request:

```
curl -X POST "http://localhost:5000/predict" -H "Content-Type: application/json" -d '{"features": [5.1, 3.5, 1.4, 0.2]}'
```

- Check Prometheus metrics:

```
curl "http://localhost:5000/metrics"
```

---

4. Set Up Prometheus for Monitoring

Create a Prometheus Config File (prometheus.yml)

yaml

global:

  scrape_interval: 5s

scrape_configs:

  - job_name: "flask_app"

    static_configs:

      - targets: ["localhost:5000"]

Run Prometheus

docker run -d --name=prometheus -p 9090:9090 -v $(pwd)/prometheus.yml:/etc/prometheus/prometheus.yml prom/prometheus

- Access Prometheus UI at **http://localhost:9090**
- Query predictions_total to see prediction counts.

---

5. Visualize Metrics with Grafana

Run Grafana

docker run -d --name=grafana -p 3000:3000 grafana/grafana

- Open **http://localhost:3000**

- Add Prometheus as a data source (URL: http://localhost:9090).
- Create a dashboard to visualize predictions_total.

---

6. Implement Data Drift Detection with Evidently AI

We'll compare incoming data with training data to detect drift.

Code: drift_monitor.py

python

```python
import pandas as pd

import requests

import numpy as np

from evidently import ColumnMapping

from evidently.report import Report

from evidently.metrics import DataDriftTable


# Load training data

iris_df = pd.DataFrame(load_iris().data, columns=load_iris().feature_names)


# Simulate new incoming data

new_data = iris_df.sample(50, random_state=42)


# Generate drift report

report = Report(metrics=[DataDriftTable()])
```

report.run(reference_data=iris_df, current_data=new_data)

report.save_html("drift_report.html")

print("Drift report saved.")

Run Drift Detection:

python drift_monitor.py

It will generate a drift_report.html file.

Summary of Commands

 # 1. Create virtual environment and install dependencies

python3 -m venv ml-monitoring-env

source ml-monitoring-env/bin/activate

pip install flask scikit-learn pandas numpy prometheus_client evidently requests matplotlib seaborn

 # 2. Train and save the model

python train_model.py

 # 3. Run the Flask API

python app.py

4. Run Prometheus

docker run -d --name=prometheus -p 9090:9090 -v $(pwd)/prometheus.yml:/etc/prometheus/prometheus.yml prom/prometheus

5. Run Grafana

docker run -d --name=grafana -p 3000:3000 grafana/grafana

6. Check prediction API

curl -X POST "http://localhost:5000/predict" -H "Content-Type: application/json" -d '{"features": [5.1, 3.5, 1.4, 0.2]}'

7. Check Prometheus metrics

curl "http://localhost:5000/metrics"

8. Run data drift detection

python drift_monitor.py

Project 2. Data Drift Monitoring & Alerting System: Continuously track dataset changes and trigger model retraining if necessary.

1. Introduction

Data Drift occurs when the data used to make predictions changes significantly from the data the model was trained on. This can cause the model's performance to degrade over time.

A **Data Drift Monitoring & Alerting System** helps detect these changes, alerts the team, and triggers model retraining if necessary.

---

2. Project Overview

We will build a system that:
✅ Continuously monitors input data for statistical changes
✅ Compares real-time data with training data
✅ Sends alerts when significant drift is detected
✅ Triggers model retraining if drift crosses a threshold

Tech Stack:

- Python
- Pandas, NumPy, SciPy (for data analysis)
- Scikit-learn (for model training)
- EvidentlyAI (for drift detection)
- FastAPI (for real-time monitoring API)
- Docker (for containerization)
- Prometheus & Grafana (for visualization & alerting)

---

3. Step-by-Step Implementation

Step 1: Set Up Environment

```
mkdir data-drift-monitoring && cd data-drift-monitoring
python -m venv venv
source venv/bin/activate  # On Windows: venv\Scripts\activate
```

pip install pandas numpy scipy scikit-learn evidently fastapi uvicorn prometheus-client

Step 2: Prepare Data

Create a dataset with training and simulated real-time data.

python

```
import pandas as pd
import numpy as np

# Generate training data
np.random.seed(42)
train_data = pd.DataFrame({
    "feature1": np.random.normal(50, 10, 1000),
    "feature2": np.random.normal(30, 5, 1000),
    "target": np.random.choice([0, 1], size=1000)
})
train_data.to_csv("train_data.csv", index=False)

# Generate simulated new data (with drift)
new_data = pd.DataFrame({
    "feature1": np.random.normal(60, 10, 1000),  # Shift in mean
    "feature2": np.random.normal(30, 5, 1000),
    "target": np.random.choice([0, 1], size=1000)
})
new_data.to_csv("new_data.csv", index=False)
```

Step 3: Detect Data Drift

python

```
from evidently.report import Report
from evidently.metric_preset import DataDriftPreset
```

```
train_data = pd.read_csv("train_data.csv")
new_data = pd.read_csv("new_data.csv")

drift_report = Report(metrics=[DataDriftPreset()])
drift_report.run(reference_data=train_data, current_data=new_data)
drift_report.save_html("drift_report.html")
```

👉 **Check drift_report.html** in a browser.

Step 4: Build an API to Monitor Drift

python

```
from fastapi import FastAPI
import pandas as pd
from evidently.report import Report
from evidently.metric_preset import DataDriftPreset

app = FastAPI()

train_data = pd.read_csv("train_data.csv")

@app.post("/check-drift/")
async def check_drift(new_data: dict):
    new_df = pd.DataFrame([new_data])
    drift_report = Report(metrics=[DataDriftPreset()])
    drift_report.run(reference_data=train_data, current_data=new_df)

    drift_result = drift_report.as_dict()
    drift_detected = drift_result['metrics'][0]['result']['dataset_drift']

    return {"data_drift": drift_detected}

# Run API
```

uvicorn filename:app --reload

Step 5: Set Up Alerts with Prometheus & Grafana

1. Install Prometheus & Grafana
2. Create a Prometheus exporter in Python

python

```
from prometheus_client import start_http_server, Gauge
import time

data_drift_gauge = Gauge("data_drift", "Data Drift Detected")

def monitor_drift():
    while True:
        drift_status = check_drift(new_data.to_dict())
        data_drift_gauge.set(1 if drift_status["data_drift"] else 0)
        time.sleep(30)

if __name__ == "__main__":
    start_http_server(8000)  # Prometheus port
    monitor_drift()
```

Step 6: Automate Model Retraining

python

```
from sklearn.ensemble import RandomForestClassifier
import pickle

def train_model():
    model = RandomForestClassifier()
    model.fit(train_data[["feature1", "feature2"]], train_data["target"])
```

```
    with open("model.pkl", "wb") as f:
        pickle.dump(model, f)

if __name__ == "__main__":
    train_model()
```

---

4. Code Explanation

- **Drift Detection**: Uses EvidentlyAI to compare live data with training data.
- **FastAPI**: Exposes an API that checks drift in real-time.
- **Prometheus**: Captures drift metrics & sends alerts.
- **Model Retraining**: Automatically updates the model when drift occurs.

🚀 Next Steps

- Deploy the system using **Docker & Kubernetes**
- Store drift logs in **MongoDB/PostgreSQL**
- Send **Email or Slack Alerts** when drift is detected

---

12. Model Rollback and Failure Management

Project 1. Intelligent Model Rollback System: Automatically roll back ML models if performance degrades in production.

In machine learning (ML) deployment, a new model might not always perform better in production. The **Intelligent Model Rollback System** helps detect performance degradation and automatically rolls back to the previous stable version. This ensures reliability and prevents a bad model from affecting business operations.

Project Workflow

1. **Train and Deploy Model** - Train an ML model and deploy it using a CI/CD pipeline.
2. **Monitor Performance** - Continuously track performance metrics (e.g., accuracy, loss, F1-score).
3. **Compare with Baseline** - If the new model underperforms compared to the previous version, trigger a rollback.
4. **Rollback Mechanism** - Automatically replace the faulty model with the last stable version.

Tech Stack

- **Python** - Core programming language
- **Flask** - API for serving the model
- **Docker** - Containerization
- **Prometheus & Grafana** - Monitoring performance metrics
- **GitHub Actions / Jenkins** - CI/CD pipeline
- **MLflow** - Model versioning and tracking

Step-by-Step Implementation

Step 1: Set Up the Project Directory

```
mkdir ml-rollback-system && cd ml-rollback-system
```

Step 2: Create a Virtual Environment

```
python3 -m venv venv
```

source venv/bin/activate # On Windows: venv\Scripts\activate

Step 3: Install Dependencies

pip install flask scikit-learn joblib requests prometheus-client mlflow

Step 4: Train and Save the Model

train_model.py

python

import joblib

import mlflow

from sklearn.datasets import load_iris

from sklearn.model_selection import train_test_split

from sklearn.ensemble import RandomForestClassifier

from sklearn.metrics import accuracy_score

Load dataset

data = load_iris()

X_train, X_test, y_train, y_test = train_test_split(data.data, data.target, test_size=0.2, random_state=42)

 # Train model

```
model = RandomForestClassifier(n_estimators=100, random_state=42)
model.fit(X_train, y_train)

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

# Log model in MLflow
mlflow.set_tracking_uri("http://localhost:5000")
mlflow.set_experiment("model_rollback")
with mlflow.start_run():
    mlflow.sklearn.log_model(model, "model")
    mlflow.log_metric("accuracy", accuracy)

# Save model locally
joblib.dump(model, "model.pkl")
print(f"Model trained with accuracy: {accuracy}")
```

Step 5: Create Model API Using Flask

app.py

python

```python
from flask import Flask, request, jsonify
import joblib
import numpy as np

app = Flask(__name__)

# Load initial model
model = joblib.load("model.pkl")

@app.route('/predict', methods=['POST'])
def predict():
    data = request.json['data']
    prediction = model.predict([np.array(data)])
    return jsonify({'prediction': prediction.tolist()})

if __name__ == '__main__':
    app.run(debug=True, host='0.0.0.0', port=8080)
```

Run the API:

```
python app.py
```

Test API with sample input:

curl -X POST "http://localhost:8080/predict" -H "Content-Type: application/json" -d '{"data": [5.1, 3.5, 1.4, 0.2]}'

Step 6: Automate Rollback if Performance Drops

rollback.py

python

```
import mlflow

import joblib

import shutil


mlflow.set_tracking_uri("http://localhost:5000")

experiment = mlflow.get_experiment_by_name("model_rollback")

runs = mlflow.search_runs(experiment_ids=[experiment.experiment_id], order_by=["metrics.accuracy DESC"])
```

 # Get last two models

```
if len(runs) > 1:

    latest_model = runs.iloc[0]
```

```
    previous_model = runs.iloc[1]

    latest_accuracy = latest_model['metrics.accuracy']
    previous_accuracy = previous_model['metrics.accuracy']

    # If new model performs worse, rollback
    if latest_accuracy < previous_accuracy:
        model_path = latest_model['artifact_uri'].replace("file://", "")
        backup_model_path = previous_model['artifact_uri'].replace("file://", "")

        shutil.copy(backup_model_path + "/model.pkl", "model.pkl")
        print("Rollback performed: Restored the previous model.")
    else:
        print("No rollback needed: New model is better.")
else:
    print("Only one model exists, rollback not applicable.")
```

Run rollback script:

```
python rollback.py
```

Step 7: Containerize the Application Using Docker

Dockerfile

```
FROM python:3.9

WORKDIR /app

COPY . /app

RUN pip install flask joblib numpy scikit-learn requests mlflow

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

Build and run container:

```
docker build -t ml-rollback .

docker run -p 8080:8080 ml-rollback
```

Step 8: Automate Deployment with CI/CD (GitHub Actions)

.github/workflows/deploy.yml

yaml

```
name: Deploy ML Model

on:

  push:

    branches:

      - main
```

```yaml
jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout repository
        uses: actions/checkout@v3

      - name: Set up Python
        uses: actions/setup-python@v3
        with:
          python-version: 3.9

      - name: Install dependencies
        run: pip install flask joblib scikit-learn requests mlflow

      - name: Train and Deploy Model
        run: |
          python train_model.py
          python rollback.py
```

Explanation of Code

- **train_model.py**: Loads the dataset, trains the ML model, and saves it using MLflow.
- **app.py**: A simple API to serve predictions using Flask.
- **rollback.py**: Checks the performance of the latest model and restores the previous model if necessary.
- **Dockerfile**: Converts the app into a container for deployment.
- **GitHub Actions Workflow**: Automates model training, rollback checks, and deployment.

Conclusion

The **Intelligent Model Rollback System** ensures that ML models in production are reliable. If performance drops, the system automatically restores the previous stable version, preventing degradation in results.

Project 2. AI-Enhanced Model Rollback Strategy: Automatically roll back to the best-performing model based on real-time inference results.

Machine learning models in production can degrade over time due to data drift, bugs, or unexpected performance drops. This project implements an **AI-Enhanced Model Rollback Strategy** that continuously monitors real-time inference results and **automatically rolls back** to the best-performing model if the current model underperforms.

Key Features

- **Real-time Monitoring:** Tracks model inference accuracy and response time.
- **Automated Rollback:** Switches to a previously stored model if performance drops below a threshold.
- **Model Versioning:** Maintains multiple model versions for quick rollback.

- **Logging & Alerts:** Records performance metrics and alerts users before rollback.

Step-by-Step Implementation

Step 1: Setup Environment

Install required dependencies:

pip install tensorflow numpy pandas scikit-learn flask

Step 2: Train and Save Multiple Models

We simulate multiple model versions for rollback.

train_model.py

python

```
import tensorflow as tf

import numpy as np

import pandas as pd

from sklearn.model_selection import train_test_split

from sklearn.preprocessing import StandardScaler

import os


# Generate synthetic dataset
```

```
np.random.seed(42)
X = np.random.rand(1000, 10)
y = (X.sum(axis=1) > 5).astype(int)

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

# Scale features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Train and save multiple model versions
for version in range(1, 4):
    model = tf.keras.models.Sequential([
        tf.keras.layers.Dense(16, activation='relu', input_shape=(10,)),
        tf.keras.layers.Dense(8, activation='relu'),
        tf.keras.layers.Dense(1, activation='sigmoid')
    ])
```

model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

model.fit(X_train, y_train, epochs=5, verbose=0)

 # Save model

model_dir = f"models/version_{version}"

os.makedirs(model_dir, exist_ok=True)

model.save(f"{model_dir}/model.h5")

print(f"Model version {version} saved successfully.")

Explanation

- Generates a **synthetic dataset** with 10 features and a binary target.
- Trains **3 different versions** of a simple neural network model.
- Saves each model in a separate versioned folder (models/version_X/model.h5).

Step 3: Deploy Model with Real-time Monitoring

We use Flask to create an API that loads the latest model and monitors performance.

app.py

python

```python
from flask import Flask, request, jsonify
import tensorflow as tf
import numpy as np
import os
import json

app = Flask(__name__)

# Load the latest model
def get_latest_model():
    versions = sorted(os.listdir("models"), reverse=True)
    latest_model_path = f"models/{versions[0]}/model.h5"
    return tf.keras.models.load_model(latest_model_path), versions[0]

model, model_version = get_latest_model()
performance_log = {}  # Store inference results

@app.route('/predict', methods=['POST'])
def predict():
    global model, model_version
    data = request.get_json()
```

```
    input_data = np.array(data['features']).reshape(1, -1)


    prediction = model.predict(input_data)[0][0]
    response = {'prediction': float(prediction), 'model_version': model_version}


    # Simulate performance tracking
    performance_log[model_version] = performance_log.get(model_version, []) +
[prediction]


    return jsonify(response)


@app.route('/rollback', methods=['POST'])
def rollback():
    global model, model_version
    versions = sorted(os.listdir("models"), reverse=True)


    if len(versions) > 1:
        print(f"Rolling back from {model_version} to {versions[1]}")
        model = tf.keras.models.load_model(f"models/{versions[1]}/model.h5")
        model_version = versions[1]
        return jsonify({"message": f"Rolled back to version {model_version}"})
    return jsonify({"message": "No older version available for rollback"}), 400
```

```python
if __name__ == '__main__':
    app.run(debug=True)
```

Explanation

- **/predict endpoint:** Accepts a JSON input and returns model predictions.
- **Performance Logging:** Tracks predictions per model version.
- **Rollback Mechanism:** If performance declines, /rollback switches to the previous version.

Step 4: Test API and Rollback Mechanism

Start the Flask API

```
python app.py
```

Make a Prediction

```
curl -X POST http://127.0.0.1:5000/predict -H "Content-Type: application/json" -d
'{"features": [0.2, 0.5, 0.1, 0.3, 0.7, 0.8, 0.2, 0.6, 0.9, 0.4]}'
```

Trigger Rollback

```
curl -X POST http://127.0.0.1:5000/rollback
```

Final Thoughts

- This project enables **automatic model rollback** based on real-time inference tracking.
- The **Flask API monitors model performance**, and if a decline is detected, the rollback mechanism is triggered.
- **Enhancements** could include:
    - Using a **database** (e.g., SQLite, MongoDB) to store inference logs.
    - Implementing a **cron job** or **Prometheus/Grafana** for automated rollback monitoring.

Project 3. Self-Healing ML Pipelines: Detect and resolve ML training failures automatically.

Machine Learning (ML) pipelines automate model training and deployment. However, failures occur due to data drift, resource limits, or bad hyperparameters. A **self-healing ML pipeline** detects these failures and resolves them automatically by **retrying, scaling resources, or adjusting parameters**.

---

2. Project Overview

We will build an ML pipeline that:

- **Trains a model** using Scikit-learn.
- **Monitors failures** (e.g., training crashes).
- **Auto-restarts training** with adjustments if needed.
- Uses **Kubernetes, Python, and MLFlow** for tracking.

Tech Stack

- Python
- Scikit-learn (for ML model)
- MLFlow (for logging & tracking)
- Kubernetes (for self-healing)

- Docker (for containerization)
- Jenkins (for CI/CD)

---

3. Step-by-Step Implementation

Step 1: Install Dependencies

```
pip install scikit-learn mlflow kubernetes requests
```

Step 2: Create a Python Script for ML Training

File: train_model.py

```
import mlflow

import numpy as np

import time

from sklearn.ensemble import RandomForestClassifier

from sklearn.model_selection import train_test_split

from sklearn.metrics import accuracy_score


# Simulating failure condition

def should_fail():

    return np.random.rand() < 0.3  # 30% chance of failure
```

```
# Load dataset
X, y = np.random.rand(1000, 10), np.random.randint(0, 2, 1000)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

# MLflow experiment tracking
mlflow.set_experiment("Self-Healing-ML")

with mlflow.start_run():
    model = RandomForestClassifier(n_estimators=50)

    if should_fail():
        print("Training failed! Retrying...")
        time.sleep(5)  # Simulate delay before retry
        exit(1)  # Simulate crash

    model.fit(X_train, y_train)
    predictions = model.predict(X_test)
    acc = accuracy_score(y_test, predictions)

    mlflow.log_metric("accuracy", acc)
```

```
    print(f"Model trained with accuracy: {acc}")
```

What it does:

- **Trains a RandomForest model.**
- **Logs accuracy in MLFlow.**
- **Fails 30% of the time** (simulating real-world issues).

Step 3: Dockerize the ML Training Script

File: Dockerfile

dockerfile

```
FROM python:3.8

WORKDIR /app

COPY train_model.py .

RUN pip install scikit-learn mlflow

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

Build and Push Docker Image

```
docker build -t my_ml_train:latest .

docker run my_ml_train
```

Step 4: Deploy on Kubernetes with Self-Healing

File: ml-job.yaml

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: ml-training-job
spec:
  template:
    spec:
      containers:
      - name: ml-training
        image: my_ml_train:latest
        restartPolicy: OnFailure
```

- **If training fails, Kubernetes automatically retries it.**

Deploy on Kubernetes

```
kubectl apply -f ml-job.yaml
kubectl get pods
```

Step 5: Automate with Jenkins

File: Jenkinsfile

groovy

```groovy
pipeline {
    agent any
    stages {
        stage('Build Docker Image') {
            steps {
                sh 'docker build -t my_ml_train:latest .'
            }
        }
        stage('Push to Docker Hub') {
            steps {
                withDockerRegistry([credentialsId: 'docker-hub', url: '']) {
                    sh 'docker push my_ml_train:latest'
                }
            }
        }
        stage('Deploy to Kubernetes') {
            steps {
                sh 'kubectl apply -f ml-job.yaml'
```

```
            }
        }
    }
}
```

- **Jenkins automates the pipeline** (build → push → deploy).

---

4. Code Explanation

1️⃣ train_model.py

- Loads dataset, trains model, logs accuracy.
- Uses **MLFlow** to track metrics.
- **30% chance of failure**, mimicking real-world errors.

2️⃣ Dockerfile

- Creates a **Docker image** for training.

3️⃣ ml-job.yaml

- **Kubernetes Job** auto-retries if training fails.

4️⃣ Jenkinsfile

- Automates **CI/CD**:
  - Builds Docker image.
  - Pushes to Docker Hub.
  - Deploys on **Kubernetes**.

Final Outcome

✅ **Self-healing ML pipeline** that retries training on failure!

---

13. Automated Model Training & Tuning

Project 1. Automated Hyperparameter Optimization: Implementing systems that automatically tune the hyperparameters of machine learning models based on past training runs, improving accuracy and reducing human effort.

Automated Hyperparameter Optimization

Hyperparameter optimization is a crucial step in machine learning, where we fine-tune the parameters of a model that are not learned during training. Automated hyperparameter tuning helps improve model accuracy while reducing human effort.

Project Overview

We will implement an **automated hyperparameter optimization** system using **Optuna**, a powerful hyperparameter tuning library. This project will involve:

1. **Building a Simple ML Model**: Using Scikit-Learn with a dataset.
2. **Implementing Hyperparameter Optimization**: Using Optuna for tuning.
3. **Automating the Process**: Running multiple trials and selecting the best parameters.

Step 1: Install Dependencies

Run the following command to install the required libraries:

pip install numpy pandas scikit-learn optuna

Step 2: Load Dataset and Prepare Data

We will use the **Breast Cancer dataset** from Scikit-Learn.

python

```
import numpy as np

import pandas as pd

from sklearn.model_selection import train_test_split

from sklearn.ensemble import RandomForestClassifier

from sklearn.metrics import accuracy_score

from sklearn.datasets import load_breast_cancer


# Load dataset

data = load_breast_cancer()

X = data.data

y = data.target


# Split into training and testing sets

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

Step 3: Define an Objective Function for Optuna

Optuna requires an **objective function** that takes a set of hyperparameters, trains a model, and returns a performance metric.

python

```
import optuna


def objective(trial):

    # Define hyperparameters to optimize

    n_estimators = trial.suggest_int("n_estimators", 50, 300)

    max_depth = trial.suggest_int("max_depth", 2, 20)

    min_samples_split = trial.suggest_int("min_samples_split", 2, 10)


    # Train model

    model = RandomForestClassifier(n_estimators=n_estimators,
max_depth=max_depth, min_samples_split=min_samples_split, random_state=42)

    model.fit(X_train, y_train)


    # Evaluate model

    y_pred = model.predict(X_test)

    accuracy = accuracy_score(y_test, y_pred)
```

return accuracy # Optuna will maximize this value

Step 4: Run the Hyperparameter Optimization

Now, we run Optuna to find the best hyperparameters.

python

 # Create study and optimize

study = optuna.create_study(direction="maximize") # We want to maximize accuracy

study.optimize(objective, n_trials=20) # Run 20 trials

 # Print the best hyperparameters

print("Best hyperparameters:", study.best_params)

Step 5: Train the Model with the Best Parameters

Once the best hyperparameters are found, we train the final model.

python

 # Retrieve best hyperparameters

best_params = study.best_params

 # Train the final model

final_model = RandomForestClassifier(**best_params, random_state=42)

final_model.fit(X_train, y_train)

 # Evaluate performance

y_pred_final = final_model.predict(X_test)

final_accuracy = accuracy_score(y_test, y_pred_final)

print(f"Final Model Accuracy: {final_accuracy:.4f}")

Step 6: Automating the Process with a Script

You can save this entire process in a Python script (hyperparameter_optimization.py) and run it automatically.

python hyperparameter_optimization.py

Code Explanation

- **Loading Dataset**: We use Scikit-Learn's **load_breast_cancer()** dataset.
- **Splitting Data**: We split data into **training** and **testing** sets.
- **Objective Function**: This is where we define **which hyperparameters to optimize**.
- **Running Optimization**: Optuna will **try different hyperparameter values** and find the best combination.
- **Final Model Training**: Once we have the best hyperparameters, we **train the final model**.

Conclusion

This project **automates hyperparameter tuning**, improving model accuracy with minimal manual effort. You can modify it for **other ML models** like XGBoost, SVM, or Deep Learning.

Project 2. CI/CD for Model Training Pipelines: Automating the entire process of model training, from data preprocessing to model evaluation, using CI/CD pipelines.

Machine Learning (ML) model training involves multiple steps like data preprocessing, model training, evaluation, and deployment. Automating this process using **CI/CD pipelines** ensures that models are trained and deployed consistently without manual intervention. This project focuses on implementing a **CI/CD pipeline** for model training using **GitHub Actions, Docker, and Jenkins**.

Project Setup and Steps

Step 1: Install Dependencies

Ensure you have the following installed:

- Python (>=3.8)
- pip (Python package manager)
- Docker
- GitHub Actions (or Jenkins)
- AWS S3 (for dataset storage, optional)

Run the following to install dependencies:

pip install numpy pandas scikit-learn joblib

Step 2: Project Structure

```
ci-cd-ml-training/
│── data/                    # Folder for dataset
│── model/                   # Folder to store trained models
│── src/                     # Source code
│   ├── preprocess.py        # Data preprocessing script
│   ├── train.py             # Model training script
│   ├── evaluate.py          # Model evaluation script
│── Dockerfile               # Docker setup for training
│── requirements.txt         # Python dependencies
│── Jenkinsfile              # Jenkins CI/CD pipeline
│── .github/workflows/       # GitHub Actions pipeline
│   ├── ci-cd-model.yml      # GitHub Actions YAML file
│── README.md                # Project documentation
```

Step 3: Write Code

Preprocessing Data (preprocess.py)

python

```python
import pandas as pd
```

```
from sklearn.model_selection import train_test_split


def load_and_preprocess_data():
    # Load dataset (replace with actual dataset)
    df = pd.read_csv("data/sample.csv")

    # Feature selection
    X = df.drop(columns=["target"])
    y = df["target"]

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

    return X_train, X_test, y_train, y_test
```

Train Model (train.py)

python

```
import joblib
from sklearn.linear_model import LogisticRegression
from preprocess import load_and_preprocess_data
```

```python
def train_model():
    X_train, _, y_train, _ = load_and_preprocess_data()

    # Train model
    model = LogisticRegression()
    model.fit(X_train, y_train)

    # Save model
    joblib.dump(model, "model/model.pkl")

    print("Model training complete and saved!")

if __name__ == "__main__":
    train_model()
```

Evaluate Model (evaluate.py)

python

```python
import joblib
from sklearn.metrics import accuracy_score
from preprocess import load_and_preprocess_data
```

```python
def evaluate_model():
    _, X_test, _, y_test = load_and_preprocess_data()
    model = joblib.load("model/model.pkl")

    # Make predictions
    y_pred = model.predict(X_test)

    # Evaluate accuracy
    acc = accuracy_score(y_test, y_pred)
    print(f"Model Accuracy: {acc:.2f}")

if __name__ == "__main__":
    evaluate_model()
```

Step 4: Create Dockerfile

dockerfile

```dockerfile
FROM python:3.8-slim
WORKDIR /app
```

```
COPY requirements.txt requirements.txt
RUN pip install -r requirements.txt
COPY . .
CMD ["python", "train.py"]
```

Build and run the Docker container:

```
docker build -t ml-training .
docker run --rm ml-training
```

Step 5: CI/CD Pipeline

GitHub Actions (.github/workflows/ci-cd-model.yml)

yaml

```yaml
name: ML Model CI/CD

on:
  push:
    branches:
      - main

jobs:
  build:
```

```yaml
runs-on: ubuntu-latest
steps:
  - name: Checkout code
    uses: actions/checkout@v3

  - name: Set up Python
    uses: actions/setup-python@v3
    with:
      python-version: '3.8'

  - name: Install dependencies
    run: pip install -r requirements.txt

  - name: Train model
    run: python src/train.py

  - name: Evaluate model
    run: python src/evaluate.py

  - name: Save model artifact
    uses: actions/upload-artifact@v3
```

```
      with:
        name: trained-model
        path: model/model.pkl
```

Jenkins Pipeline (Jenkinsfile)

groovy

```groovy
pipeline {
    agent any
    stages {
        stage('Checkout Code') {
            steps {
                git 'https://github.com/user/ml-ci-cd.git'
            }
        }
        stage('Install Dependencies') {
            steps {
                sh 'pip install -r requirements.txt'
            }
        }
        stage('Train Model') {
            steps {
```

```
            sh 'python src/train.py'
          }
        }
        stage('Evaluate Model') {
          steps {
            sh 'python src/evaluate.py'
          }
        }
        stage('Archive Model') {
          steps {
            archiveArtifacts artifacts: 'model/model.pkl', fingerprint: true
          }
        }
    }
}
```

Step 6: Run and Deploy

GitHub Actions

1. Push the code to GitHub
2. GitHub Actions will trigger the pipeline automatically
3. The model will be trained and saved as an artifact

Jenkins

1. Start Jenkins: systemctl start jenkins
2. Add the repository as a Jenkins job
3. Trigger the pipeline
4. The trained model will be stored in Jenkins artifacts

Explanation for Beginners

1. **Preprocessing** (preprocess.py) loads the dataset, selects features, and splits it into training and testing sets.
2. **Training** (train.py) uses **Logistic Regression** to train a model and save it.
3. **Evaluation** (evaluate.py) checks how well the model performs.
4. **Dockerfile** ensures the entire training process runs in an isolated environment.
5. **GitHub Actions** automatically trains and evaluates the model whenever you push code.
6. **Jenkins Pipeline** can also automate the workflow similarly to GitHub Actions.

Conclusion

This project automates the entire model training process using **CI/CD pipelines**. It ensures that models are trained, evaluated, and stored automatically, improving efficiency and reliability.

Project 3. Automated Model Versioning: Creating an automated system that manages different versions of models, ensuring that only validated models are deployed into production.

When developing machine learning models, managing different versions is crucial to ensure reliability, reproducibility, and easy rollback to previous versions if needed. Automated model versioning ensures that only validated models are deployed into production, reducing risks and maintaining performance consistency.

In this project, we will:

- Train a simple ML model
- Automatically track its versions using **MLflow**
- Validate models before deployment
- Deploy only the validated models

Project Implementation

Step 1: Install Dependencies

Install the required libraries:

pip install numpy pandas scikit-learn mlflow flask gunicorn

Step 2: Set Up MLflow for Model Versioning

MLflow is a tool that helps with experiment tracking and model versioning.

Start MLflow Tracking Server:

mlflow server --backend-store-uri sqlite:///mlflow.db --default-artifact-root ./mlruns --host 0.0.0.0 --port 5000

Step 3: Train and Version a Model

Create a file train_model.py to train a simple model and log it into MLflow.

train_model.py

python

```
import numpy as np

import pandas as pd

import mlflow

import mlflow.sklearn

from sklearn.model_selection import train_test_split

from sklearn.ensemble import RandomForestClassifier

from sklearn.metrics import accuracy_score


# Set MLflow tracking URI

mlflow.set_tracking_uri("http://127.0.0.1:5000")


# Load sample data

data = pd.DataFrame({

    "feature1": np.random.rand(100),

    "feature2": np.random.rand(100),

    "label": np.random.randint(0, 2, 100)

})
```

```
X = data[["feature1", "feature2"]]

y = data["label"]


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


# Train a model

model = RandomForestClassifier(n_estimators=10, random_state=42)

model.fit(X_train, y_train)


# Evaluate the model

y_pred = model.predict(X_test)

accuracy = accuracy_score(y_test, y_pred)


# Log model with MLflow

with mlflow.start_run():

    mlflow.log_metric("accuracy", accuracy)

    mlflow.sklearn.log_model(model, "model")


    # Register model
```

lflow.register_model("runs:/{}/model".format(mlflow.active_run().info.run_id), "RandomForestModel")

print(f"Model trained and logged with accuracy: {accuracy}")

Run the script:

python train_model.py

Step 4: Validate and Deploy Only Best Models

We want to ensure only models with accuracy above **80%** are deployed.

validate_deploy.py

python

import mlflow

from mlflow.tracking import MlflowClient

mlflow.set_tracking_uri("http://127.0.0.1:5000")

client = MlflowClient()

 # Fetch latest model version

model_name = "RandomForestModel"

latest_version = client.get_latest_versions(model_name, stages=["None"])[0]

Get model accuracy

run_id = latest_version.run_id

metrics = client.get_run(run_id).data.metrics

accuracy = metrics.get("accuracy", 0)

Validate model

if accuracy > 0.80:

client.transition_model_version_stage(name=model_name, version=latest_version.version, stage="Production")

print(f"Model version {latest_version.version} deployed to production.")

else:

print(f"Model version {latest_version.version} did not meet accuracy threshold.")

Run the validation:

python validate_deploy.py

Step 5: Serve the Production Model via API

Once a model is validated and deployed, we expose it via a REST API.

model_api.py

python

```python
import mlflow.pyfunc
from flask import Flask, request, jsonify

app = Flask(__name__)

# Load the production model
model_name = "RandomForestModel"
model = mlflow.pyfunc.load_model(f"models:/{model_name}/Production")

@app.route("/predict", methods=["POST"])
def predict():
    data = request.json
    features = [data["feature1"], data["feature2"]]
    prediction = model.predict([features])
    return jsonify({"prediction": int(prediction[0])})

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5001)
```

Run the API server:

```
python model_api.py
```

Test prediction:

```
curl -X POST http://127.0.0.1:5001/predict -H "Content-Type: application/json" -d
'{"feature1": 0.5, "feature2": 0.8}'
```

Conclusion

This project automates model versioning using MLflow and ensures that only validated models are deployed into production. It helps maintain reliability and enables easy rollback to previous models if needed.