AIOps & MLOps DevOps Projects Part-1

AIOps Projects (AI in DevOps)

1. Log and Incident Management

Project 1. Intelligent Log Analysis: Use AI/ML to analyze logs from Kubernetes, Jenkins, or Docker and automatically detect anomalies.

Project 2. AI-Driven Log Parsing & Alerting: Train an NLP model to classify logs (info, warning, error, critical) and generate alerts in real time.

Project 3. AI-Driven Log Aggregation & Summarization: Use NLP to analyze and summarize logs from multiple sources (Kubernetes, Jenkins, CloudWatch).

Project 4. Self-Learning Incident Management System: Build a system that suggests automated fixes based on past incidents.

Project 5. AI-Driven Incident Response Playbook: Create a system that suggests incident resolution steps based on past issues.

---

2. Resource and Cost Optimization

Project 1. Predictive Auto-Scaling: Develop an AI-driven system to predict server/resource usage and auto-scale Kubernetes clusters.

Project 2. AI-Powered Cost Optimization: Use ML to analyze cloud billing data and recommend cost-saving measures.

Project 3. AI-Powered Cloud Resource Optimization: Train an ML model to recommend the best instance types and scaling configurations.

Project 4. AI-Assisted Infrastructure Cost Forecasting: Use time-series forecasting to predict cloud costs and prevent budget overruns.

Project 5. AI-Assisted Container Resource Allocation: Use reinforcement learning to optimize CPU/memory allocation in Docker containers.

---

3. Anomaly Detection & Failure Prediction

Project 1. Anomaly Detection in DevSecOps: Train an AI model to detect security vulnerabilities in containerized applications.

Project 2. Kubernetes Node Failure Prediction: Predict pod/node failures in Kubernetes clusters using AI-based anomaly detection.

Project 3. Anomaly Detection for Network Traffic: Use ML to identify unusual patterns in network traffic and detect potential DDoS attacks.

Project 4. Predictive Disk Failure Monitoring: Analyze disk I/O metrics using ML to predict hardware failures in advance.

Project 5. Smart CI/CD Failure Prediction: Train an AI model to analyze Jenkins pipeline logs and predict build failures before they occur.

---

4. Incident Prediction and Root Cause Analysis

Project 1. Incident Prediction & Root Cause Analysis: Build a machine learning model that predicts system failures based on historical monitoring data.

Project 2. AI-Based Root Cause Analysis (RCA): Build a model that correlates incidents, logs, and metrics to identify the root cause of failures.

---

5. Security and Compliance

Project 1. Automated Security Policy Enforcement with AI: Use AI to detect misconfigurations in firewall rules, IAM policies, and network security.

Project 2. AI-Powered SLA Compliance Monitoring: Analyze service response times and uptime metrics using ML to predict SLA violations.

---

6. Self-Healing and Automation

Project 1. Self-Healing Infrastructure: Use AI to detect and auto-remediate cloud infrastructure issues (e.g., restarting failed pods in Kubernetes).

Project 2. AI-Based Configuration Drift Detection: Build a model that monitors infrastructure-as-code (Terraform, Ansible) for unintended changes.

---

7. AI for Log Analysis & Monitoring

Project 1. AI-Powered Log Filtering & Categorization: Implementing AI to automatically filter out noise in logs and categorize relevant events for quicker analysis.

Project 2. Real-Time Anomaly Detection in Logs: AI system that processes logs in real time and raises alerts when unusual patterns or behavior are detected.

Project 3. Log Correlation for Performance Issues: Using AI to correlate logs from different services to identify root causes of performance degradation or service outages.

Project 4. AI-Based Multi-Source Log Aggregation: Aggregating logs from diverse sources (cloud, on-prem, containers, etc.) using AI to spot cross-system anomalies.

Project 5. Automated Log Tagging: Using AI to automatically tag logs with metadata for faster identification and analysis.

---

8. AI for Predictive Scaling & Performance Optimization

Project 1. Predictive Load Balancing: AI model that predicts incoming traffic and adjusts load balancing strategies accordingly to optimize resource usage and minimize latency.

Project 2. AI-Driven Predictive Resource Allocation: Using AI to dynamically allocate resources (CPU, memory, storage) based on predicted workloads in containers and VMs.

Project 3. Predictive Autoscaling with Customizable Metrics: AI-based auto-scaling system that considers custom application-specific metrics in addition to CPU/memory load.

Project 4. AI-Powered Resource Bottleneck Detection: AI to analyze performance metrics and detect resource bottlenecks that may affect scaling decisions.

Project 5. Multi-Tenant Cloud Optimization: Using AI to ensure efficient resource sharing in multi-tenant cloud environments without compromising performance.

---

9. AI for Incident Prediction & Automated Remediation

Project 1. Automated Health Checks with AI: AI-powered health check system that automatically checks infrastructure health and suggests fixes before failure.

Project 2. Dynamic Incident Severity Prediction: AI model that predicts the potential severity of an incident based on past data, helping teams prioritize responses.

Project 3. Proactive Failure Prevention System: AI-based system that uses failure trends to predict and prevent critical infrastructure failures before they happen.

Project 4. Predictive Incident Management in Multi-Cloud: AI to predict incidents across different cloud environments and suggest remediation actions.

Project 5. AI-Powered Predictive Alerting: Using machine learning models to identify patterns that precede incidents and proactively alert teams before failure occurs.

---

10. AI for CI/CD & DevSecOps

Project 1. AI-Driven Test Suite Optimization: Using AI to automatically optimize the sequence of tests in CI/CD pipelines to reduce the overall pipeline runtime.

Project 2. AI for Continuous Security Assessment: Real-time security vulnerability detection during the CI/CD pipeline, integrated into DevSecOps practices.

Project 3. AI-Based Dependency Vulnerability Scanning: Implement AI-based scanning of dependencies in code repositories for potential vulnerabilities or license compliance issues.

Project 4. Automated Code Quality Review with AI: AI models that scan code during CI/CD builds and provide insights into code quality, security, and performance improvements.

Project 5. AI-Enhanced Test Failure Analysis: Using AI to automatically analyze failed tests in CI/CD pipelines and suggest possible causes and fixes.

---

11. AI for Infrastructure & Network Monitoring

Project 1. AI-Powered Load Forecasting for Infrastructure: Predicting infrastructure load for upcoming days or weeks using historical data and adjusting resource allocation accordingly.

Project 2. Proactive Infrastructure Health Monitoring: AI model for identifying potential infrastructure failures before they occur by monitoring system health in real time.

Project 3. Network Traffic Anomaly Detection with AI: Using machine learning to detect outliers in network traffic data (e.g., unusual spikes or drops), potentially identifying attacks.

Project 4. Distributed Network Monitoring with AI: AI to monitor network performance across distributed environments (hybrid clouds, multi-region setups) and provide insights.

AIOps Projects (AI in DevOps)

1. Log and Incident Management

Project 1. Intelligent Log Analysis: Use AI/ML to analyze logs from Kubernetes, Jenkins, or Docker and automatically detect anomalies.

In modern cloud-native environments like Kubernetes, Jenkins, and Docker, logs are crucial for monitoring and troubleshooting applications. However, manually analyzing vast amounts of log data can be overwhelming. Intelligent log analysis powered by AI/ML automates the detection of anomalies, such as errors or unusual behavior, in real-time. By leveraging models like Isolation Forest and LSTM, this project aims to automatically identify issues from logs, reducing manual effort and enabling quicker responses. It also integrates real-time monitoring with Prometheus and visualization using Grafana, enhancing operational efficiency and system reliability.

Intelligent Log Analysis with AI/ML

1. Log Collection and Integration

Logs from Kubernetes, Jenkins, and Docker will be collected using respective tools and commands.

Kubernetes Logs:

To collect logs from Kubernetes:

```
kubectl logs <pod-name> -n <namespace> > kubernetes_logs.txt
```

Jenkins Logs:

Jenkins stores logs for jobs and system logs. You can extract logs using:

```
tail -f /var/log/jenkins/jenkins.log > jenkins_logs.txt
```

Docker Logs:

For Docker containers:

```
docker logs <container-id> > docker_logs.txt
```

Alternatively, set up Logstash or Fluentd to ingest logs from these services in real-time.

---

2. Log Preprocessing

Logs will be preprocessed to clean, parse, and structure them for analysis.

Python Script for Log Preprocessing:

Install necessary libraries:
pip install pandas numpy

Preprocess the logs by reading them, cleaning, and structuring them:
python

import pandas as pd

import numpy as np

def preprocess_logs(log_file):

    # Load logs

    logs_df = pd.read_csv(log_file, sep="|", header=None, names=["timestamp", "level", "message"])

    # Convert timestamp to datetime

logs_df['timestamp'] = pd.to_datetime(logs_df['timestamp'])

    # Create additional features

    logs_df['hour'] = logs_df['timestamp'].dt.hour

    logs_df['error_level'] = logs_df['level'].apply(lambda x: 1 if x == 'ERROR' else 0)

    return logs_df

logs_df = preprocess_logs('logs.txt')

print(logs_df.head())

---

3. Anomaly Detection

Unsupervised Learning Model (Isolation Forest)

Isolation Forest can be used for detecting anomalies in log patterns.

Install necessary libraries:
pip install scikit-learn

Use Isolation Forest to detect anomalies:
python

from sklearn.ensemble import IsolationForest

 # Prepare feature columns (hour and error_level)

X = logs_df[['hour', 'error_level']]

 # Initialize Isolation Forest model

model = IsolationForest(contamination=0.05)

 # Fit the model to the data

logs_df['anomaly'] = model.fit_predict(X)

 # Mark anomalies

anomalies = logs_df[logs_df['anomaly'] == -1]

print(anomalies)

Deep Learning Model (LSTM)

For more advanced anomaly detection, a Long Short-Term Memory (LSTM) model can be used for time-series data.

Install necessary libraries:
pip install tensorflow

LSTM model for anomaly detection in logs:
python

import tensorflow as tf

from sklearn.preprocessing import MinMaxScaler

```
# Normalize the data
scaler = MinMaxScaler(feature_range=(0, 1))
logs_df[['hour', 'error_level']] = scaler.fit_transform(logs_df[['hour', 'error_level']])

# Prepare the data for LSTM (time-series format)
X = logs_df[['hour', 'error_level']].values
X = X.reshape((X.shape[0], X.shape[1], 1))  # Reshaping for LSTM input

# Define the LSTM model
model = tf.keras.Sequential([
    tf.keras.layers.LSTM(50, activation='relu', input_shape=(X.shape[1], 1)),
    tf.keras.layers.Dense(1)
])
model.compile(optimizer='adam', loss='mean_squared_error')

# Train the model (using part of the data as the training set)
model.fit(X, X, epochs=10, batch_size=32)
```

---

4. Real-time Anomaly Detection

To integrate real-time log collection and anomaly detection:

Set up Fluentd or Logstash to Collect Logs:

Logstash example configuration:
yaml

```
input {
  file {
    path => "/var/log/containers/*.log"
    start_position => "beginning"
  }
}


output {
  elasticsearch {
    hosts => ["http://localhost:9200"]
    index => "logs"
  }
}
```

Prometheus and Alertmanager for Monitoring and Alerting:

1. Install Prometheus and Alertmanager for monitoring and alerting on anomalies.

Prometheus rule to alert when anomalies are detected:
yaml

```yaml
groups:
- name: anomaly_detection
  rules:
  - alert: AnomalyDetected
    expr: anomaly_rate > 5
    for: 1m
```

---

5. Visualization and Reporting

Grafana for Visualization:

Install Grafana:
sudo apt-get install grafana

- Create a Grafana dashboard that queries Elasticsearch for logs and displays anomalies in real-time.

Kibana for Log Exploration:

Install Kibana:
sudo apt-get install kibana

- Configure Kibana to connect to Elasticsearch and create visualizations for error trends, anomaly counts, and other key metrics.

---

6. Model Evaluation and Retraining

Model Evaluation:

Evaluate the anomaly detection model using classification metrics like Precision, Recall, and F1-Score:

python

from sklearn.metrics import classification_report

 # Assuming `y_true` is the actual labels and `y_pred` is the predicted anomalies

print(classification_report(y_true, y_pred))

Retraining the Model:

To ensure the model adapts to new log patterns, retrain it periodically with fresh logs:

python

model.fit(new_log_data, new_labels)

---

7. Complete Workflow for Logs, Model, and Alerting

1. **Log Collection:** Collect logs from Kubernetes, Jenkins, or Docker.
2. **Preprocessing:** Clean and structure the logs.
3. **Model Training:** Train an unsupervised model like Isolation Forest or a time-series LSTM model for anomaly detection.
4. **Real-time Detection:** Use Fluentd or Logstash for real-time log collection and integrate it with Prometheus for alerting.

5. **Visualization:** Use Grafana and Kibana for visualizing anomalies and log trends.
6. **Evaluation and Retraining:** Continuously evaluate and retrain the model as new logs come in.

Conclusion:

This project provides a comprehensive framework for analyzing logs from Kubernetes, Jenkins, and Docker, leveraging AI/ML models to detect anomalies. It integrates log collection, preprocessing, anomaly detection, and real-time monitoring with visualization tools like Grafana and Kibana. Additionally, it provides a feedback loop for evaluating and retraining the model as new data comes in.

Project 2. AI-Driven Log Parsing & Alerting: Train an NLP model to classify logs (info, warning, error, critical) and generate alerts in real time.

This project aims to teach how to use Artificial Intelligence (AI) to process logs (records of system activities) and classify them into categories like info, warning, error, and critical. Once classified, the system will alert you if something goes wrong (for example, when an error or critical event happens).

Steps to Build the Project

1. Setting Up Your Environment

- Install Python, which is the programming language we will use.
- Install libraries that will help us process and analyze text. These libraries are like tools that make tasks easier.

Command to install necessary libraries:

pip install scikit-learn pandas nltk tensorflow

2. Prepare Your Log Data

Logs are records that show what happens in a system. For example, a log could say "The server started" or "Database connection failed."

- Collect your logs, either from a file or a live system.
- Make sure your logs have a "log level" (like info, error) and a message (like "System started").

Example log data:

pgsql

25-02-06 00:12:45 [INFO] System started

2025-02-06 00:15:30 [ERROR] Database connection failed

3. Preprocessing the Data

The logs need to be cleaned up so the AI can understand them better. We'll:

- Make all the text lowercase (so the system doesn't get confused by different capitalizations).
- Remove punctuation and unnecessary words.

Example code to clean the logs:

python

from nltk.tokenize import word_tokenize

from nltk.corpus import stopwords

import string

```
def preprocess_text(text):
    text = text.lower()  # Convert everything to lowercase
    text = ''.join([char for char in text if char not in string.punctuation])  # Remove punctuation
    tokens = word_tokenize(text)  # Split the text into words
    stop_words = set(stopwords.words('english'))
    tokens = [word for word in tokens if word not in stop_words]  # Remove unnecessary words
    return ' '.join(tokens)


data['processed_message'] = data['message'].apply(preprocess_text)
```

4. Labeling the Log Levels

To help the AI understand the log's type, we need to label the log levels (info, warning, error, critical) into numbers. This makes it easier for the machine to work with the data.

Code to convert log levels to numbers:

python

```
from sklearn.preprocessing import LabelEncoder


le = LabelEncoder()
data['label'] = le.fit_transform(data['log_level'])
```

5. Train the AI to Classify Logs

Now, we train a machine learning model. This model learns from past logs and tries to classify new logs into categories like info, error, etc.

- Split the data into training data (which the model will learn from) and testing data (which we will use to check if the model is working well).
- We'll use a method called Logistic Regression to train the model. It's like teaching the AI how to recognize patterns in logs.

Example code to train the model:

python

```
from sklearn.model_selection import train_test_split

from sklearn.feature_extraction.text import TfidfVectorizer

from sklearn.linear_model import LogisticRegression

from sklearn.metrics import classification_report


X_train, X_test, y_train, y_test = train_test_split(data['processed_message'], data['label'], test_size=0.2)


tfidf = TfidfVectorizer(max_features=5000)

X_train_tfidf = tfidf.fit_transform(X_train)

X_test_tfidf = tfidf.transform(X_test)


model = LogisticRegression()

model.fit(X_train_tfidf, y_train)
```

```
y_pred = model.predict(X_test_tfidf)

print(classification_report(y_test, y_pred))
```

6. Real-Time Log Classification and Alerts

- Now, we set up the system to keep checking for new logs. Whenever a new log appears, the system will classify it and send an alert if it's an error or critical log.

Example code to classify and send alerts:

python

```
import time


def classify_and_alert(log_message):

    processed_message = preprocess_text(log_message)

    message_tfidf = tfidf.transform([processed_message])

    log_class = model.predict(message_tfidf)

    log_level = le.inverse_transform(log_class)[0]


    if log_level in ['error', 'critical']:

        send_alert(log_message, log_level)


def send_alert(log_message, log_level):
```

```python
    # Logic for sending alerts (email, SMS, Slack, etc.)
    print(f"ALERT: {log_level.upper()} log detected: {log_message}")


while True:
    new_log = get_new_log_from_file_or_stream()  # Implement log fetching logic
    classify_and_alert(new_log)
    time.sleep(1)  # Check for new logs every second
```

7. Testing and Deploying

- Test the system with some logs to see how it works.
- Once it works, you can deploy it on a server or in the cloud, where it can monitor logs in real-time.

8. Improvement & Scaling

- You can improve the model by training it with more data.
- You can also connect this system with log management tools like ELK Stack or Splunk for better monitoring.

Conclusion

This AI-driven log parsing and alerting system helps you monitor logs, detect problems, and get alerts when something goes wrong in real-time. It's a great starting point for learning about machine learning, AI, and how to handle logs in a system.

Project 3. AI-Driven Log Aggregation & Summarization: Use NLP to analyze and summarize logs from multiple sources (Kubernetes, Jenkins, CloudWatch).

Project Introduction

This project focuses on **log aggregation and summarization** using **Natural Language Processing (NLP)**. Logs from various sources like **Kubernetes, Jenkins, and AWS CloudWatch** are collected, analyzed, and summarized using AI. This helps in **quick issue detection, reducing noise, and improving observability**.

Tech Stack

- **Python** (FastAPI for API, Pandas for log processing)
- **NLP** (spaCy, OpenAI/GPT, Transformers for summarization)
- **Log Sources** (Kubernetes logs, Jenkins logs, AWS CloudWatch)
- **Elasticsearch** (Optional, for centralized storage)
- **Docker & Kubernetes** (Deployment)

Step 1: Environment Setup

Install dependencies:

 # Create and activate a virtual environment

python3 -m venv env

source env/bin/activate # On Windows, use `env\Scripts\activate`

 # Install necessary libraries

pip install fastapi uvicorn pandas transformers spacy boto3 elasticsearch

---

Step 2: Collect Logs from Different Sources

Kubernetes Logs

```
kubectl logs <pod-name> -n <namespace> > logs/k8s_logs.txt
```

Jenkins Logs

```
tail -n 100 /var/log/jenkins/jenkins.log > logs/jenkins_logs.txt
```

AWS CloudWatch Logs (Using Boto3)

python

```python
import boto3


def get_cloudwatch_logs(log_group, start_time, end_time):
    client = boto3.client('logs', region_name='us-east-1')
    response = client.filter_log_events(
        logGroupName=log_group,
        startTime=start_time,
        endTime=end_time
    )
    logs = [event['message'] for event in response['events']]
    return "\n".join(logs)
```

```
logs = get_cloudwatch_logs('/aws/lambda/my-function', 1700000000000,
1700100000000)

with open('logs/cloudwatch_logs.txt', 'w') as f:
    f.write(logs)
```

Step 3: Process & Clean Logs

```python
import pandas as pd


def clean_logs(file_path):
    with open(file_path, 'r') as f:
        logs = f.readlines()
    logs = [log.strip() for log in logs if log.strip()]
    return pd.DataFrame({'log_entry': logs})


df = clean_logs('logs/k8s_logs.txt')
print(df.head())  # Check processed logs
```

Step 4: Summarize Logs Using NLP

python

```
from transformers import pipeline


# Load a pre-trained summarization model
summarizer = pipeline("summarization", model="facebook/bart-large-cnn")


def summarize_logs(logs):
    text = " ".join(logs[:500])  # Limit to avoid token limit
    summary = summarizer(text, max_length=100, min_length=30,
do_sample=False)
    return summary[0]['summary_text']


logs = df['log_entry'].tolist()
summary = summarize_logs(logs)
print("Summary:", summary)
```

Step 5: Deploy as API using FastAPI

python

```
from fastapi import FastAPI
```

```python
app = FastAPI()


@app.post("/summarize/")
async def summarize_endpoint(logs: list[str]):
    summary = summarize_logs(logs)
    return {"summary": summary}


# Run API server
if __name__ == "__main__":
    import uvicorn
    uvicorn.run(app, host="0.0.0.0", port=8000)
```

Start API server:

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

Step 6: Dockerize & Deploy on Kubernetes

Dockerfile

```
FROM python:3.9
WORKDIR /app
COPY . /app
```

RUN pip install -r requirements.txt

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

Build & Run Docker Image

docker build -t log-summarizer .

docker run -p 8000:8000 log-summarizer

Kubernetes Deployment

yaml

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

```yaml
        app: log-summarizer
    spec:
      containers:
        - name: log-summarizer
          image: log-summarizer:latest
          ports:
            - containerPort: 8000
```

Apply in Kubernetes:

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

Conclusion

This project **automates log aggregation and summarization** using **NLP-based AI**. The API can be integrated with monitoring tools like **Grafana** for better observability.

Project 4. Self-Learning Incident Management System: Build a system that suggests automated fixes based on past incidents.

Introduction

Incident management is crucial for IT and DevOps teams. A **Self-Learning Incident Management System** automates issue resolution by analyzing past incidents and suggesting fixes. Using **Flask, MongoDB, and Machine Learning**, this project helps reduce downtime and improve operational efficiency.

Project Features

- **Incident Logging:** Users can report incidents with descriptions.
- **Database Storage:** Incidents are stored in **MongoDB**.
- **Machine Learning Model:** Suggests fixes based on past incidents.
- **Web Interface:** Users can log and view incident details.
- **REST API:** Allows integration with other tools.

Technology Stack

- **Backend:** Flask (Python)
- **Database:** MongoDB
- **Machine Learning:** scikit-learn (TF-IDF & Logistic Regression)
- **Frontend:** HTML, Bootstrap
- **Deployment:** Docker, Kubernetes (Optional)

Step-by-Step Guide

1. Setup Environment

```
mkdir incident-management

cd incident-management

python3 -m venv venv

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

pip install flask pymongo scikit-learn pandas nltk
```

---

2. MongoDB Installation (Ubuntu)

```
sudo apt update

sudo apt install -y mongodb

sudo systemctl start mongodb

sudo systemctl enable mongodb
```

Verify MongoDB is running:

```
mongo --eval "db.runCommand({ connectionStatus: 1 })"
```

---

3. Create a MongoDB Database

Connect to MongoDB and create a collection:

python

```
from pymongo import MongoClient

client = MongoClient("mongodb://localhost:27017/")
db = client["incident_db"]
collection = db["incidents"]

sample_incident = {
    "title": "Server Down",
```

"description": "The application server is not responding",

"solution": "Restart the server"

}

collection.insert_one(sample_incident)

print("Sample Incident Added!")

---

4. Create a Flask API

Create a file app.py:

python

```
from flask import Flask, request, jsonify
from pymongo import MongoClient
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
import pandas as pd

app = Flask(__name__)
```

 # Connect to MongoDB

```
client = MongoClient("mongodb://localhost:27017/")
```

```
db = client["incident_db"]
collection = db["incidents"]

# Train ML Model
def train_model():
    data = list(collection.find({}, {"_id": 0, "description": 1, "solution": 1}))
    df = pd.DataFrame(data)

    if df.empty:
        return None, None

    vectorizer = TfidfVectorizer()
    X = vectorizer.fit_transform(df["description"])
    y = df["solution"]

    model = LogisticRegression()
    model.fit(X, y)

    return model, vectorizer

model, vectorizer = train_model()
```

```
@app.route("/log_incident", methods=["POST"])
def log_incident():
    data = request.json
    collection.insert_one(data)
    return jsonify({"message": "Incident Logged!"})


@app.route("/suggest_fix", methods=["POST"])
def suggest_fix():
    if not model:
        return jsonify({"error": "No data to train the model"}), 400


    data = request.json
    desc_vector = vectorizer.transform([data["description"]])
    suggestion = model.predict(desc_vector)[0]


    return jsonify({"suggested_fix": suggestion})


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

---

5. Run Flask App

```
export FLASK_APP=app.py

flask run
```

The API will be available at http://127.0.0.1:5000.

---

6. Test API (Using curl or Postman)

Log an Incident

```
curl -X POST http://127.0.0.1:5000/log_incident \

-H "Content-Type: application/json" \

-d '{"title": "Database Error", "description": "Connection timeout issue", "solution": "Check network and restart DB"}'
```

Get Suggested Fix

```
curl -X POST http://127.0.0.1:5000/suggest_fix \

-H "Content-Type: application/json" \

-d '{"description": "The server is down"}'
```

---

7. Build a Simple Frontend

Create templates/index.html:

html

```html
<!DOCTYPE html>

<html>

<head>

    <title>Incident Management</title>

    <link rel="stylesheet"
href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css/bootstrap.min.css">

</head>

<body class="container mt-5">

    <h2>Incident Management System</h2>

    <form id="incidentForm">

        <input type="text" id="description" placeholder="Enter incident description"
class="form-control mb-2">

        <button type="button" class="btn btn-primary" onclick="suggestFix()">Get
Fix</button>

    </form>

    <h4 class="mt-3" id="solution"></h4>



    <script>

        async function suggestFix() {
```

```
            const desc = document.getElementById("description").value;
            const response = await fetch('/suggest_fix', {
                method: 'POST',
                headers: {'Content-Type': 'application/json'},
                body: JSON.stringify({"description": desc})
            });
            const data = await response.json();
            document.getElementById("solution").innerText = "Suggested Fix: " + data.suggested_fix;
        }
    </script>
</body>
</html>
```

Run the Flask app and open http://127.0.0.1:5000 in a browser.

---

8. Containerize with Docker

Create a Dockerfile:

dockerfile

```
FROM python:3.9
WORKDIR /app
```

```
COPY requirements.txt .

RUN pip install -r requirements.txt

COPY . .

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

Build and Run:

```
docker build -t incident-management .

docker run -p 5000:5000 incident-management
```

---

9. Deploy with Kubernetes (Optional)

Create deployment.yaml:

yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: incident-management
spec:
  replicas: 2
  selector:
    matchLabels:
```

```yaml
      app: incident-management
  template:
    metadata:
      labels:
        app: incident-management
    spec:
      containers:
      - name: incident-app
        image: incident-management:latest
        ports:
        - containerPort: 5000
```

Apply Deployment:

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

Code Explanation

1. **Flask API**: Handles logging incidents and suggesting fixes.
2. **MongoDB Storage**: Stores incidents and solutions.
3. **Machine Learning**: Uses **TF-IDF Vectorization** and **Logistic Regression** to suggest solutions.
4. **Frontend (HTML, Bootstrap)**: Simple form to get incident fixes.
5. **Docker & Kubernetes**: Containerization and deployment for scalability.

Conclusion

This **Self-Learning Incident Management System** helps automate issue resolution based on past incidents. By integrating **Flask, MongoDB, and Machine Learning**, it improves IT incident response, reducing downtime and manual effort.

Project 5. AI-Driven Incident Response Playbook: Create a system that suggests incident resolution steps based on past issues.

Incident management is crucial in IT operations. Traditional methods rely on manual playbooks, which can be time-consuming and inconsistent. This project introduces an **AI-Driven Incident Response Playbook**, which learns from past incidents and suggests resolution steps automatically.

We will use:

- **Python & Flask** (Backend API)
- **MongoDB** (Storing past incidents)
- **Machine Learning (Scikit-learn)** (AI model for recommendations)
- **Docker** (Containerization)
- **Jenkins** (CI/CD pipeline)

Step-by-Step Implementation

1. Install Dependencies

Ensure you have the required tools installed:

sudo apt update && sudo apt install python3 python3-pip docker.io -y

pip3 install flask pymongo scikit-learn joblib

---

2. Set Up MongoDB for Incident Storage

MongoDB will store previous incidents and their resolutions.

Start MongoDB

```
docker run -d --name mongo -p 27017:27017 mongo
```

Create Incident Database

```python
from pymongo import MongoClient


client = MongoClient("mongodb://localhost:27017/")
db = client["incident_db"]
collection = db["incidents"]


incident_data = {
    "issue": "Server down",
    "resolution": "Restart the service using systemctl restart apache2"
}


collection.insert_one(incident_data)
```

```
print("Sample incident inserted")
```

---

3. Build AI Model

The model will predict the best resolution based on historical data.

Train AI Model

python

```
import pandas as pd
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.neighbors import KNeighborsClassifier
import joblib


# Sample Data
data = [
    {"issue": "CPU usage high", "resolution": "Kill unnecessary processes"},
    {"issue": "Server down", "resolution": "Restart the service"},
    {"issue": "Memory leak", "resolution": "Check for memory-intensive apps"}
]


df = pd.DataFrame(data)


vectorizer = TfidfVectorizer()
```

```
X = vectorizer.fit_transform(df["issue"])
y = df["resolution"]

model = KNeighborsClassifier(n_neighbors=1)
model.fit(X, y)

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

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

---

4. Create Flask API

This API will suggest resolutions based on user input.

Install Flask

```
pip3 install flask
```

Create app.py

```python
from flask import Flask, request, jsonify
import joblib
```

```
import pymongo

app = Flask(__name__)

# Load model
model = joblib.load("incident_model.pkl")
vectorizer = joblib.load("vectorizer.pkl")

# MongoDB Connection
client = pymongo.MongoClient("mongodb://localhost:27017/")
db = client["incident_db"]
collection = db["incidents"]

@app.route("/predict", methods=["POST"])
def predict():
    data = request.json
    issue_text = data["issue"]

    vectorized_text = vectorizer.transform([issue_text])
    prediction = model.predict(vectorized_text)[0]
```

```python
    # Save to MongoDB
    collection.insert_one({"issue": issue_text, "suggested_resolution": prediction})

    return jsonify({"resolution": prediction})

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

---

5. Dockerize the Project

Create Dockerfile

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

Build & Run

```bash
docker build -t ai-playbook .
docker run -p 5000:5000 ai-playbook
```

---

6. Testing the API

Send an Incident

```
curl -X POST http://localhost:5000/predict -H "Content-Type: application/json" -d
'{"issue": "CPU usage high"}'
```

Expected Response

json

```json
{"resolution": "Kill unnecessary processes"}
```

---

7. Set Up CI/CD in Jenkins

Create Jenkinsfile

groovy

```groovy
pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                sh 'docker build -t ai-playbook .'
            }
        }
```

```
        stage('Test') {
            steps {
                sh 'docker run -d --name test-ai -p 5000:5000 ai-playbook'
                sh 'curl -X POST http://localhost:5000/predict -H "Content-Type: application/json" -d \'{"issue": "Server down"}\''
            }
        }
        stage('Deploy') {
            steps {
                sh 'docker tag ai-playbook your-dockerhub-username/ai-playbook:latest'
                sh 'docker push your-dockerhub-username/ai-playbook:latest'
            }
        }
    }
}
```

Run Pipeline

jenkins

Conclusion

This **AI-Driven Incident Response Playbook**: ✅ Uses AI to suggest solutions ✅ Stores incidents in MongoDB

✅ Exposes predictions via an API
✅ Runs in Docker for easy deployment

---

2. Resource and Cost Optimization

Project 1. Predictive Auto-Scaling: Develop an AI-driven system to predict server/resource usage and auto-scale Kubernetes clusters.

Introduction

Auto-scaling is essential in cloud environments to handle traffic spikes efficiently. This project builds an AI-driven predictive auto-scaler for Kubernetes clusters, using machine learning to forecast resource usage and adjust cluster size dynamically.

Step-by-Step Implementation

1. Prerequisites

Ensure you have the following installed:

- **Kubernetes (kind/minikube/EKS/GKE/AKS)**
- **kubectl** (Kubernetes CLI)
- **Prometheus** (for collecting metrics)
- **Grafana** (for visualization)
- **Python** (for ML model)
- **Flask** (to serve predictions)
- **Docker** (for containerization)
- **KEDA** (Kubernetes Event-Driven Autoscaling)
- **Helm** (for managing applications)

---

2. Set Up Kubernetes Cluster

kind create cluster --name auto-scaler

kubectl cluster-info

---

3. Deploy Prometheus for Metrics Collection

Install Prometheus using Helm:

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

helm repo update

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

Check Prometheus is running:

kubectl get pods -n default | grep prometheus

---

4. Deploy Grafana for Monitoring

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

Access Grafana at **http://localhost:3000**
(Default username: admin, password: prom-operator)

---

5. Collect Metrics Using Prometheus API

Check resource utilization:

```
kubectl top nodes

kubectl top pods
```

Prometheus Query Example:

```
http://localhost:9090/api/v1/query?query=node_cpu_seconds_total
```

---

6. Train a Machine Learning Model for Prediction

Install Python Dependencies:

```
pip install pandas scikit-learn flask requests
```

Train the ML Model (train_model.py)

```
import pandas as pd

import numpy as np

from sklearn.linear_model import LinearRegression

import joblib


# Simulated CPU Usage Data

data = pd.DataFrame({

    'timestamp': np.arange(1, 101),

    'cpu_usage': np.random.randint(30, 90, 100)
```

```
})

X = data[['timestamp']]
y = data['cpu_usage']

model = LinearRegression()
model.fit(X, y)

joblib.dump(model, 'cpu_predictor.pkl')
print("Model trained and saved.")
```

Run the script:

```
python train_model.py
```

---

7. Create a Flask API for Predictions

Create app.py:

python

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

```python
app = Flask(__name__)
model = joblib.load('cpu_predictor.pkl')


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


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

Run the API:

```
python app.py
```

Test API:

```
curl -X POST http://localhost:5000/predict -H "Content-Type: application/json" -d '{"timestamp": [101, 102, 103]}'
```

---

8. Containerize the Flask App

Create Dockerfile:

Dockerfile

```
FROM python:3.9

WORKDIR /app

COPY . /app

RUN pip install -r requirements.txt

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

Build and run the container:

```
docker build -t auto-scaler .

docker run -p 5000:5000 auto-scaler
```

---

9. Deploy Flask API in Kubernetes

Create deployment.yaml:

```
apiVersion: apps/v1

kind: Deployment

metadata:
```

```yaml
  name: auto-scaler
spec:
  replicas: 1
  selector:
    matchLabels:
      app: auto-scaler
  template:
    metadata:
      labels:
        app: auto-scaler
    spec:
      containers:
      - name: auto-scaler
        image: auto-scaler:latest
        ports:
        - containerPort: 5000
---
apiVersion: v1
kind: Service
metadata:
  name: auto-scaler-service
```

```yaml
spec:
  selector:
    app: auto-scaler
  ports:
  - protocol: TCP
    port: 80
    targetPort: 5000
  type: LoadBalancer
```

Apply deployment:

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

---

10. Configure KEDA for Auto-Scaling

Install KEDA:

```
helm repo add kedacore https://kedacore.github.io/charts
helm repo update
helm install keda kedacore/keda
```

Create scaledobject.yaml:

yaml

```yaml
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: auto-scaler
spec:
  scaleTargetRef:
    name: auto-scaler
  minReplicaCount: 1
  maxReplicaCount: 5
  triggers:
  - type: prometheus
    metadata:
      serverAddress: http://prometheus-server.default.svc.cluster.local
      query: avg(rate(node_cpu_seconds_total[2m])) * 100
      threshold: '70'
```

Apply scaling rule:

```
kubectl apply -f scaledobject.yaml
```

---

11. Test Auto-Scaling

Simulate load using **hey** (install via apt install hey):

hey -n 10000 -c 50 http://localhost:5000/predict

Check pods scaling up:

kubectl get pods -w

---

12. Monitor Auto-Scaling with Grafana

1. Open Grafana (http://localhost:3000)
2. Add Prometheus as a data source

Use queries like:
sql

avg(rate(node_cpu_seconds_total[2m])) * 100

---

1. **ML Model (train_model.py)**:
   - Generates fake CPU data
   - Trains a Linear Regression model
   - Saves the model using joblib
2. **Flask API (app.py)**:
   - Loads the trained model
   - Accepts a timestamp and predicts CPU usage
   - Returns prediction in JSON format
3. **Dockerfile**:

    - Defines a Python-based container
    - Copies app files and installs dependencies
    - Runs the Flask server
4. **Kubernetes (deployment.yaml)**:
    - Deploys the Flask app
    - Exposes it as a LoadBalancer service
5. **KEDA (scaledobject.yaml)**:
    - Uses Prometheus metrics to trigger auto-scaling
    - Scales when CPU usage exceeds 70%

Final Outcome

- **Machine Learning predicts CPU usage**
- **KEDA auto-scales Kubernetes pods based on predictions**
- **Prometheus collects real-time metrics**
- **Grafana visualizes performance**

Project 2. AI-Powered Cost Optimization: Use ML to analyze cloud billing data and recommend cost-saving measures.

Introduction

Cloud costs can quickly spiral out of control if not monitored effectively. This project leverages **Machine Learning** to analyze cloud billing data and suggest cost-saving strategies. By using **Python, Pandas, Scikit-Learn, and Matplotlib**, we'll process billing data, detect cost anomalies, and predict future cloud expenses.

Project Steps

Step 1: Setup the Environment

Install Required Packages

Ensure you have Python installed and set up a virtual environment:

python3 -m venv cost-opt-env

source cost-opt-env/bin/activate # On Windows: cost-opt-env\Scripts\activate

pip install pandas numpy scikit-learn matplotlib seaborn

Step 2: Prepare the Cloud Billing Data

Obtain your cloud billing data from AWS, Azure, or GCP. The format should include:

- **Service Name** (EC2, S3, RDS, etc.)
- **Cost** (USD)
- **Usage Hours**
- **Region**
- **Instance Type**

Example CSV File (cloud_billing.csv):

cs

Service, Cost, Usage_Hours, Region, Instance_Type

EC2, 120, 500, us-east-1, t3.medium

S3, 30, 200, us-east-1, N/A

RDS, 80, 300, us-west-2, db.m5.large

Step 3: Load and Preprocess Data

Create a Python script cost_optimization.py

python

```
import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.preprocessing import StandardScaler

from sklearn.cluster import KMeans


# Load cloud billing data

df = pd.read_csv("cloud_billing.csv")


# Check for missing values

print(df.isnull().sum())


# Convert categorical data to numerical values

df = pd.get_dummies(df, columns=["Service", "Region", "Instance_Type"], drop_first=True)
```

 # Scale the data

scaler = StandardScaler()

scaled_data = scaler.fit_transform(df.drop(columns=["Cost"]))

 # Display processed data

print(df.head())

Step 4: Detect Cost Anomalies with K-Means Clustering

We'll use **K-Means Clustering** to detect outliers (high-cost services).

python

 # Apply K-Means Clustering

kmeans = KMeans(n_clusters=3, random_state=42)

df["Cluster"] = kmeans.fit_predict(scaled_data)

 # Visualize clusters

plt.figure(figsize=(8,5))

sns.scatterplot(x=df["Usage_Hours"], y=df["Cost"], hue=df["Cluster"], palette="viridis")

plt.xlabel("Usage Hours")

plt.ylabel("Cost")

plt.title("Cloud Cost Clustering")

plt.show()

📌 **Interpretation:**

- Services in high-cost clusters can be **optimized** (switch to reserved instances, downgrade instance types, reduce unused services).

Step 5: Predict Future Cloud Costs using Linear Regression

We'll train a model to predict next month's cost based on historical usage.

python

from sklearn.model_selection import train_test_split

from sklearn.linear_model import LinearRegression

from sklearn.metrics import mean_absolute_error

 # Define input (X) and output (y) variables

X = df.drop(columns=["Cost", "Cluster"])

y = df["Cost"]

 # Split dataset 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)

```
# Train Linear Regression Model
model = LinearRegression()
model.fit(X_train, y_train)

# Make predictions
y_pred = model.predict(X_test)

# Evaluate model
mae = mean_absolute_error(y_test, y_pred)
print(f"Mean Absolute Error: {mae}")

# Predict cost for new data
new_data = np.array([[400, 1, 0, 0, 1]])  # Example usage hours & instance type
predicted_cost = model.predict(new_data)
print(f"Predicted Next Month's Cost: ${predicted_cost[0]:.2f}")
```

Step 6: Automate Cost-Saving Recommendations

We can automate cost-saving tips based on thresholds:

python

```
def suggest_cost_savings(row):
    if row["Cost"] > 100:
        return "Consider Reserved Instances or Auto-scaling"
    elif row["Usage_Hours"] > 400:
        return "Optimize Instance Usage or Rightsize"
    else:
        return "No changes needed"

df["Recommendation"] = df.apply(suggest_cost_savings, axis=1)
print(df[["Service", "Cost", "Recommendation"]])
```

Step 7: Deploy as a Flask API (Optional)

You can create a **Flask API** to accept billing data and return cost-saving recommendations.

Install Flask

```
pip install flask
```

Create app.py

python

```python
from flask import Flask, request, jsonify

import pandas as pd


app = Flask(__name__)


@app.route('/predict', methods=['POST'])

def predict():

    data = request.json

    df = pd.DataFrame([data])

    df["Recommendation"] = df.apply(suggest_cost_savings, axis=1)

    return jsonify(df.to_dict(orient="records"))


if __name__ == '__main__':

    app.run(debug=True)
```

Run API

```
python app.py
```

Test API using cURL

```
curl -X POST -H "Content-Type: application/json" -d '{"Service": "EC2", "Cost": 150, "Usage_Hours": 500, "Region": "us-east-1", "Instance_Type": "t3.medium"}' http://127.0.0.1:5000/predict
```

Conclusion

- We used **K-Means Clustering** to detect high-cost services.
- **Linear Regression** was used to **predict future costs**.
- **Automated cost-saving recommendations** help optimize cloud spending.
- **Optional API** enables integration with real-world applications.

Project 3. AI-Powered Cloud Resource Optimization: Train an ML model to recommend the best instance types and scaling configurations.

Introduction

Cloud computing offers flexibility, but choosing the right instance type and scaling strategy can be complex. This project focuses on training a Machine Learning (ML) model to analyze past resource usage data and recommend optimal cloud instance types and auto-scaling configurations. The goal is to minimize cost while maintaining performance.

Step-by-Step Guide

1. Setup the Environment

Prerequisites

- Python 3.x
- AWS CLI (or any cloud provider SDK)
- Jupyter Notebook
- Required Python libraries: pandas, numpy, scikit-learn, matplotlib, seaborn

Install Required Libraries

```
pip install pandas numpy scikit-learn matplotlib seaborn boto3
```

---

2. Collect and Prepare Data

Cloud resource optimization requires data such as:

- CPU, memory, and network usage logs
- Instance type and cost details
- Scaling history

Fetch Cloud Metrics Using AWS CLI

```
aws cloudwatch get-metric-statistics --namespace AWS/EC2 \

    --metric-name CPUUtilization --start-time 2024-02-01T00:00:00Z \

    --end-time 2024-02-07T00:00:00Z --period 300 --statistics Average \

    --dimensions Name=InstanceId,Value=i-1234567890abcdef \

    --region us-east-1
```

---

3. Load and Explore Data

```python
import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns
```

```python
# Load dataset (assuming we have a CSV file)

df = pd.read_csv("cloud_metrics.csv")


# Display first few rows

print(df.head())


# Basic statistics

print(df.describe())


# Visualize CPU usage

plt.figure(figsize=(10,5))

sns.lineplot(x=df["timestamp"], y=df["cpu_utilization"])

plt.title("CPU Utilization Over Time")

plt.show()
```

---

4. Feature Engineering

python

```python
# Extract useful features

df['hour'] = pd.to_datetime(df['timestamp']).dt.hour
```

df['day'] = pd.to_datetime(df['timestamp']).dt.dayofweek

 # Drop unnecessary columns

df.drop(columns=['timestamp'], inplace=True)

---

5. Train the Machine Learning Model

python

from sklearn.model_selection import train_test_split

from sklearn.ensemble import RandomForestRegressor

from sklearn.metrics import mean_absolute_error

 # Define input features and target variable

X = df.drop(columns=["instance_type"])

y = df["instance_type"] # Labels: Optimal instance types

 # 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 = RandomForestRegressor(n_estimators=100, random_state=42)

model.fit(X_train, y_train)

 # Predict and evaluate

predictions = model.predict(X_test)

print("Mean Absolute Error:", mean_absolute_error(y_test, predictions))

---

6. Make Predictions

python

 # Example: Predict best instance type for new usage data

new_data = [[30, 4]] # Example: CPU utilization 30%, Sunday

predicted_instance = model.predict(new_data)

print("Recommended Instance Type:", predicted_instance)

---

7. Deploy Model as an API (Flask)

python

from flask import Flask, request, jsonify

import pickle

```python
app = Flask(__name__)

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

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

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

Run API

```
python app.py
```

Test API

```
curl -X POST http://127.0.0.1:5000/predict -H "Content-Type: application/json" \
-d '{"features": [30, 4]}'
```

Conclusion

This project leverages ML to suggest the best cloud instances based on historical usage. It reduces cost and improves performance by recommending optimal scaling configurations.

Project 4. AI-Assisted Infrastructure Cost Forecasting: Use time-series forecasting to predict cloud costs and prevent budget overruns.

Introduction

Cloud cost forecasting is crucial for optimizing infrastructure expenses and avoiding budget overruns. This project leverages **time-series forecasting** techniques using **Python, Pandas, Matplotlib, Scikit-learn, and Facebook's Prophet** to analyze past cloud usage data and predict future costs.

By implementing **AI-assisted forecasting**, businesses can make informed decisions about resource allocation, cost-saving strategies, and scaling policies.

Project Setup and Execution

Step 1: Prerequisites

Ensure you have the required dependencies installed.

 # Update package list

sudo apt update

 # Install Python and pip if not already installed

sudo apt install python3 python3-pip -y

Create and activate a virtual environment (optional but recommended)

python3 -m venv cost_forecast_env

source cost_forecast_env/bin/activate

Step 2: Install Required Python Libraries

pip install pandas numpy matplotlib scikit-learn prophet

Step 3: Data Collection & Preprocessing

Create a file **cloud_cost_data.csv** with historical cost data.

Example CSV Format:

Date	Cost ($)
2024-01-01	1200

2024-02-01	1250
2024-03-01	1300
2024-04-01	1100
2024-05-01	1350

Step 4: Implement AI-Based Forecasting

Create a Python script forecast_cost.py and add the following code:

python

```
import pandas as pd

import matplotlib.pyplot as plt

from prophet import Prophet


# Load dataset

df = pd.read_csv("cloud_cost_data.csv")


# Rename columns for Prophet

df.rename(columns={"Date": "ds", "Cost ($)": "y"}, inplace=True)
```

 # Initialize Prophet model

model = Prophet()

model.fit(df)

 # Create future dataframe (next 6 months)

future = model.make_future_dataframe(periods=6, freq='M')

 # Predict future costs

forecast = model.predict(future)

 # Plot results

fig = model.plot(forecast)

plt.title("Cloud Cost Forecast")

plt.xlabel("Date")

plt.ylabel("Cost ($)")

plt.show()

 # Save forecast to CSV

forecast[['ds', 'yhat', 'yhat_lower', 'yhat_upper']].to_csv("cost_forecast.csv", index=False)

Step 5: Running the Project

Execute the script:

```
python3 forecast_cost.py
```

This will generate a forecast graph and save the predicted values in **cost_forecast.csv**.

Step 6: Explanation of Code

- **Loading Data**: Reads the historical cloud cost data from CSV.
- **Preprocessing**: Renames columns for compatibility with Prophet.
- **Training Model**: Fits the Prophet model to learn patterns from past data.
- **Forecasting**: Generates predictions for the next 6 months.
- **Visualization**: Displays a graph of historical and forecasted costs.
- **Saving Output**: Stores predicted values in a CSV file for further analysis.

Conclusion

This AI-based cost forecasting solution helps businesses anticipate infrastructure expenses, optimize cloud usage, and prevent unexpected budget spikes. You can further enhance the model by integrating real-time cloud billing data using APIs from AWS, GCP, or Azure.

Project 5. AI-Assisted Container Resource Allocation: Use reinforcement learning to optimize CPU/memory allocation in Docker containers.

Introduction

Managing CPU and memory allocation in Docker containers is challenging. Allocating too many resources wastes capacity, while allocating too few degrades performance. Reinforcement Learning (RL) can **dynamically** adjust these allocations based on real-time usage, maximizing efficiency.

We will build an AI model using **OpenAI Gym**, **Stable-Baselines3**, and **Docker SDK for Python** to optimize resource allocation.

Step 1: Setting Up the Environment

Install Dependencies

Ensure you have **Python 3.8+**, **Docker**, and required libraries installed.

 # Update system and install Docker

sudo apt update && sudo apt install docker.io -y

sudo systemctl start docker

sudo systemctl enable docker

 # Install Python and dependencies

python3 -m venv rl-container-env

source rl-container-env/bin/activate

pip install numpy pandas gym docker stable-baselines3

Check if Docker is working:

docker run hello-world

Step 2: Creating a Custom Gym Environment for Resource Allocation

Reinforcement Learning works by training an **agent** to interact with an **environment** and learn the best actions. We will create a **custom Gym environment** to simulate resource allocation for containers.

Create the Environment File

Create a new Python file docker_env.py

python

```
import gym

import docker

import numpy as np

from gym import spaces


class DockerResourceEnv(gym.Env):

    def __init__(self):

        super(DockerResourceEnv, self).__init__()


        # Connect to Docker

        self.client = docker.from_env()

        self.container_name = "test_container"
```

 # Action Space: CPU (0.1 to 2 cores), Memory (128MB to 2GB)

self.action_space = spaces.Box(low=np.array([0.1, 128]), high=np.array([2.0, 2048]), dtype=np.float32)

 # Observation Space: CPU usage and Memory usage

self.observation_space = spaces.Box(low=0, high=np.inf, shape=(2,), dtype=np.float32)

 # Start a test container

self.container = self.client.containers.run("nginx", detach=True, name=self.container_name, cpu_period=100000, cpu_quota=10000, mem_limit="128m")

def step(self, action):

cpu, memory = action

 # Apply new resource limits

self.container.update(cpu_quota=int(cpu * 100000), mem_limit=f"{int(memory)}m")

 # Simulate performance (use actual Docker stats)

stats = self.container.stats(stream=False)

```python
        cpu_usage = stats["cpu_stats"]["cpu_usage"]["total_usage"] / stats["cpu_stats"]["system_cpu_usage"]

        memory_usage = stats["memory_stats"]["usage"]


        reward = -abs(cpu_usage - 0.5) - abs(memory_usage / int(memory) - 0.5)  # Penalize large deviations


        return np.array([cpu_usage, memory_usage]), reward, False, {}


    def reset(self):

        return np.array([0.5, 128])


    def render(self, mode="human"):

        pass


    def close(self):

        self.container.stop()

        self.container.remove()
```

Step 3: Training the RL Model

Create a new file train_rl.py to train the model.

python

```python
import gym
from stable_baselines3 import PPO
from docker_env import DockerResourceEnv

# Create the environment
env = DockerResourceEnv()

# Load the RL model
model = PPO("MlpPolicy", env, verbose=1)
model.learn(total_timesteps=50000)

# Save the trained model
model.save("rl_docker_allocator")
env.close()
```

This trains an AI agent using the **Proximal Policy Optimization (PPO)** algorithm to optimize resource allocation.

Step 4: Running the AI Model for Real-Time Resource Allocation

Create run_ai.py to apply the trained model to live containers.

```python
from stable_baselines3 import PPO
from docker_env import DockerResourceEnv

# Load trained model
model = PPO.load("rl_docker_allocator")

# Create environment
env = DockerResourceEnv()

# Run optimization loop
obs = env.reset()
for _ in range(100):
    action, _states = model.predict(obs)
    obs, reward, done, _ = env.step(action)
    print(f"CPU: {action[0]}, Memory: {action[1]}, Reward: {reward}")

env.close()
```

Step 5: Testing the AI Model

Run the AI-powered resource allocator:

python run_ai.py

It will dynamically adjust CPU and memory allocation based on real-time container usage.

Code Explanation for New Learners

1. **Custom Gym Environment (docker_env.py)**
   - Defines an **RL environment** where Docker containers act as **agents**.
   - The RL **agent learns** to optimize CPU/memory.
   - Uses **Docker SDK** to control container resources dynamically.
2. **Training the RL Model (train_rl.py)**
   - Uses **Stable-Baselines3's PPO algorithm** to train an AI model.
   - The AI learns the best CPU/memory allocation over time.
3. **Applying AI Model (run_ai.py)**
   - Loads the trained AI model.
   - Dynamically **adjusts** CPU/memory allocation **based on live data**.

Conclusion

This project demonstrates how **AI and Reinforcement Learning** can optimize **container resource allocation** in real time. By training an RL model with **OpenAI Gym and Docker**, we can efficiently manage CPU and memory in Docker containers, improving performance and resource utilization.

---

3. Anomaly Detection & Failure Prediction

Project 1. Anomaly Detection in DevSecOps: Train an AI model to detect security vulnerabilities in containerized applications.

Anomaly Detection in DevSecOps involves identifying unusual patterns that may indicate security vulnerabilities in applications. Using **machine learning (ML) and security scanning tools**, we can train a model to predict vulnerabilities based on historical data.

---

2. Prerequisites

Ensure you have the following installed:

- Python (>=3.8)
- TensorFlow or PyTorch
- Docker & Kubernetes
- Trivy (for vulnerability scanning)
- Jupyter Notebook (for ML training)

---

3. Setup Environment

Install necessary dependencies:

sudo apt update && sudo apt install python3-pip -y

pip install tensorflow pandas numpy matplotlib scikit-learn seaborn trivy

---

4. Collect Security Data

Scan a Docker image using Trivy and save the output as a JSON file.

trivy image --format json -o vulnerabilities.json nginx:latest

This will provide a dataset containing vulnerabilities.

---

5. Preprocess Data

Convert JSON to CSV for ML training.

python

```
import json
import pandas as pd


# Load Trivy scan result
with open("vulnerabilities.json") as f:
    data = json.load(f)


# Extract relevant fields
df = pd.DataFrame([
    {
        "package": vuln["PkgName"],
        "severity": vuln["Severity"],
        "vulnerability_id": vuln["VulnerabilityID"],
    }
```

    for result in data["Results"] for vuln in result["Vulnerabilities"]

])

 # Save to CSV

df.to_csv("vulnerabilities.csv", index=False)

---

6. Train an AI Model

Using TensorFlow to detect vulnerabilities.

python

import tensorflow as tf

from sklearn.model_selection import train_test_split

from sklearn.preprocessing import LabelEncoder

 # Load dataset

df = pd.read_csv("vulnerabilities.csv")

 # Encode categorical data

le = LabelEncoder()

df["severity"] = le.fit_transform(df["severity"])

 # Train-test split

X_train, X_test, y_train, y_test = train_test_split(df[["severity"]], df["severity"], test_size=0.2, random_state=42)

 # Build a simple ML model

model = tf.keras.Sequential([

    tf.keras.layers.Dense(16, 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=10, batch_size=8)

---

7. Containerize the Application

Create a Dockerfile for the trained model:

dockerfile

FROM python:3.8

WORKDIR /app

COPY . /app

```
RUN pip install tensorflow pandas numpy scikit-learn
CMD ["python", "predict.py"]
```

Build and run the container:

```
docker build -t anomaly-detector .
docker run -it anomaly-detector
```

---

8. Deploy on Kubernetes

Create a deployment.yaml file:

yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: anomaly-detector
spec:
  replicas: 1
  selector:
    matchLabels:
      app: anomaly-detector
```

```yaml
  template:
    metadata:
      labels:
        app: anomaly-detector
    spec:
      containers:
      - name: anomaly-detector
        image: anomaly-detector:latest
        ports:
        - containerPort: 5000
```

Apply the deployment:

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

---

9. Monitor the Deployment

Check running pods and logs:

```
kubectl get pods
kubectl logs -f <pod-name>
```

10. Summary

- Used **Trivy** to scan for vulnerabilities.
- Processed the scan data for ML training.
- Built a **TensorFlow-based anomaly detection model**.
- **Containerized and deployed** it on Kubernetes.
- **Monitored and tested** the deployment.

This project **integrates AI into DevSecOps** to enhance **automated vulnerability detection** in CI/CD pipelines

---

Project 2. Kubernetes Node Failure Prediction: Predict pod/node failures in Kubernetes clusters using AI-based anomaly detection.

Kubernetes is widely used for managing containerized applications, but node failures can impact availability and performance. This project leverages AI-based anomaly detection to predict failures in advance, allowing proactive measures like workload redistribution or auto-scaling.

Project Overview

- **Use Case**: Monitor Kubernetes node metrics and detect anomalies using machine learning.
- **Technology Stack**: Kubernetes, Prometheus, Grafana, Python, Scikit-learn (or TensorFlow/PyTorch), Flask (optional for API), Docker.
- **Workflow**:
  - Collect real-time metrics from Kubernetes nodes using Prometheus.
  - Process data and extract features.
  - Train an anomaly detection model.
  - Deploy the model in Kubernetes for real-time predictions.

Step-by-Step Implementation

Step 1: Set Up a Kubernetes Cluster

If using a local cluster:

kind create cluster --name k8s-ai

kubectl cluster-info

For a cloud-based setup (EKS, AKS, GKE), follow their respective guides.

Step 2: Install Prometheus for Metrics Collection

Create a monitoring namespace:

kubectl create namespace monitoring

Deploy Prometheus:

kubectl apply -f
https://github.com/prometheus-operator/prometheus-operator/releases/latest/download/bundle.yaml

Verify Prometheus is running:

kubectl get pods -n monitoring

Step 3: Set Up Node Exporter to Collect Node Metrics

kubectl apply -f
https://raw.githubusercontent.com/prometheus/node_exporter/main/examples/kubernetes/node-exporter-daemonset.yaml

Check logs:

kubectl logs -l app=node-exporter -n monitoring

Step 4: Build the Machine Learning Model

Install dependencies:

pip install pandas scikit-learn prometheus-api-client flask

Python Script (Train the Model)

Create train_model.py:

python

```
import pandas as pd

import numpy as np

from sklearn.ensemble import IsolationForest

import joblib


# Simulated dataset (replace with Prometheus metrics in real implementation)

data = pd.DataFrame({

    'cpu_usage': np.random.normal(50, 10, 1000),

    'memory_usage': np.random.normal(60, 15, 1000),

    'disk_io': np.random.normal(30, 5, 1000),

})
```

 # Train an anomaly detection model

model = IsolationForest(contamination=0.05)

model.fit(data)

 # Save the model

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

print("Model trained and saved.")

Run the script:

python train_model.py

Step 5: Deploy the Prediction API in Kubernetes

Create predictor.py:

python

from flask import Flask, request, jsonify

import joblib

import numpy as np

app = Flask(__name__)

```
model = joblib.load("failure_prediction_model.pkl")


@app.route('/predict', methods=['POST'])
def predict():
    data = request.get_json()
    features = np.array([data['cpu_usage'], data['memory_usage'],
data['disk_io']]).reshape(1, -1)
    prediction = model.predict(features)
    result = "Anomaly detected (Possible Failure)" if prediction[0] == -1 else
"Normal"
    return jsonify({'prediction': result})


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

Run locally to test:

```
python predictor.py
```

Test API:

```
curl -X POST http://localhost:5000/predict -H "Content-Type: application/json" -d
'{"cpu_usage": 80, "memory_usage": 90, "disk_io": 50}'
```

Step 6: Containerize and Deploy in Kubernetes

Create a Dockerfile:

dockerfile

```
FROM python:3.9

WORKDIR /app

COPY predictor.py failure_prediction_model.pkl /app/

RUN pip install flask joblib numpy

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

Build and push the image:

```
docker build -t <your-dockerhub-username>/k8s-failure-predictor .

docker push <your-dockerhub-username>/k8s-failure-predictor
```

Create a Kubernetes Deployment (predictor-deployment.yaml):

yaml

```
apiVersion: apps/v1

kind: Deployment

metadata:

  name: predictor

  labels:
```

```yaml
    app: predictor
spec:
  replicas: 1
  selector:
    matchLabels:
      app: predictor
  template:
    metadata:
      labels:
        app: predictor
    spec:
      containers:
      - name: predictor
        image: <your-dockerhub-username>/k8s-failure-predictor
        ports:
        - containerPort: 5000
---
apiVersion: v1
kind: Service
metadata:
  name: predictor-service
```

```yaml
spec:
  selector:
    app: predictor
  ports:
    - protocol: TCP
      port: 80
      targetPort: 5000
  type: LoadBalancer
```

Apply the deployment:

kubectl apply -f predictor-deployment.yaml

Check running pods:

kubectl get pods

Step 7: Visualizing Anomalies with Grafana

Deploy Grafana:

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

Access Grafana:

```
kubectl port-forward svc/grafana 3000:80 -n monitoring
```

Login (default: admin/admin) and configure Prometheus as a data source.

Explanation for New Learners

- **Kubernetes Cluster**: Manages applications and resources.
- **Prometheus**: Collects real-time node metrics.
- **Node Exporter**: Exposes system-level metrics.
- **Machine Learning Model**: Detects anomalies using IsolationForest.
- **Flask API**: Serves predictions via REST API.
- **Docker & Kubernetes**: Containerizes and deploys the predictor service.
- **Grafana**: Visualizes anomalies for monitoring.

Project 3. Anomaly Detection for Network Traffic: Use ML to identify unusual patterns in network traffic and detect potential DDoS attacks.

Anomaly detection in network traffic is essential for cybersecurity. Machine learning models can analyze network patterns and identify unusual activities that may indicate potential attacks, such as Distributed Denial-of-Service (DDoS) attacks. This project will guide you through building an anomaly detection model using Python and Scikit-learn.

Project Steps

Step 1: Set Up Your Environment

Ensure you have Python installed and required libraries. Run the following commands:

pip install numpy pandas scikit-learn matplotlib seaborn

Step 2: Import Required Libraries

python

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.ensemble import IsolationForest

from sklearn.preprocessing import StandardScaler

from sklearn.model_selection import train_test_split

Step 3: Load and Prepare the Dataset

We'll use a synthetic network traffic dataset. You can also download a real dataset like the **CICIDS2017 dataset**.

python

 # Load dataset (simulated data for network traffic)

data = pd.read_csv("network_traffic.csv")

Display first few rows

print(data.head())

Check for missing values

print(data.isnull().sum())

Step 4: Data Preprocessing

Normalize and clean the data to prepare for model training.

python

```
# Select relevant features (assuming numerical columns)
features = ['packet_size', 'flow_duration', 'num_bytes', 'src_port', 'dst_port']
X = data[features]

# Normalize data
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Split into training and testing sets
X_train, X_test = train_test_split(X_scaled, test_size=0.2, random_state=42)
```

Step 5: Train the Isolation Forest Model

The Isolation Forest is an unsupervised learning algorithm for anomaly detection.

python

```
# Train Isolation Forest model

model = IsolationForest(contamination=0.05, random_state=42)

model.fit(X_train)


# Predict anomalies

y_pred = model.predict(X_test)


# Convert predictions (-1: Anomaly, 1: Normal) to readable format

y_pred = np.where(y_pred == -1, "Anomaly", "Normal")


# Add results to DataFrame

results = pd.DataFrame(X_test, columns=features)

results['Prediction'] = y_pred


# Display some predictions

print(results.head(10))
```

Step 6: Visualize Anomalies

python

 # Convert predictions to numeric values (1: Normal, -1: Anomaly)

results['Prediction'] = np.where(results['Prediction'] == "Anomaly", -1, 1)

 # Plot anomalies

plt.figure(figsize=(10, 6))

sns.scatterplot(x=results['flow_duration'], y=results['num_bytes'], hue=results['Prediction'], palette={1: "blue", -1: "red"})

plt.title("Anomalies in Network Traffic")

plt.xlabel("Flow Duration")

plt.ylabel("Number of Bytes")

plt.show()

---

1. **Importing Libraries**
    - We use numpy and pandas for data handling.
    - matplotlib and seaborn help in visualization.
    - IsolationForest detects anomalies based on data distribution.
2. **Loading and Preprocessing Data**
    - We select relevant numerical features for model training.
    - The data is scaled to ensure consistent value ranges.
3. **Training the Model**
    - The IsolationForest algorithm identifies outliers in the dataset.
    - A contamination value of 0.05 means 5% of data is considered anomalous.

4. **Predicting and Visualizing Results**
   - The model classifies network traffic as normal or anomalous.
   - A scatter plot visualizes unusual patterns in network traffic.

Project 4. Predictive Disk Failure Monitoring: Analyze disk I/O metrics using ML to predict hardware failures in advance.

Hard drive failures can lead to **data loss** and **downtime**. Predicting failures in advance helps in **preventive maintenance**. This project will use **Machine Learning (ML) to analyze disk I/O metrics** and predict potential failures based on SMART (Self-Monitoring, Analysis, and Reporting Technology) data.

Step 1: Set Up the Environment

1.1 Install Required Packages

Ensure your system has Python installed. Install the required libraries:

pip install pandas numpy scikit-learn matplotlib seaborn

For handling SMART data, install smartmontools:

sudo apt update && sudo apt install smartmontools

Step 2: Collect Disk Metrics

2.1 Enable SMART Monitoring

Check if SMART is enabled on your disk:

sudo smartctl -i /dev/sda

If it's disabled, enable it:

sudo smartctl -s on /dev/sda

2.2 Fetch SMART Data

To get disk health data:

sudo smartctl -A /dev/sda

To export SMART data to a file:

sudo smartctl -A /dev/sda > smart_data.txt

Step 3: Preprocess Data

3.1 Convert SMART Data to CSV

We extract attributes like **Reallocated Sectors, Power-On Hours, Temperature, and Error Rates** into a CSV.
Create extract_smart_data.py:

python

import os

import pandas as pd

```python
def parse_smart_data(file_path):
    data = {}
    with open(file_path, 'r') as file:
        for line in file:
            parts = line.split()
            if len(parts) > 9 and parts[0].isdigit():
                attr_id = int(parts[0])
                value = int(parts[9])  # Raw value
                data[attr_id] = value

    return data

# Read SMART data file
smart_data = parse_smart_data("smart_data.txt")

# Convert to DataFrame
df = pd.DataFrame([smart_data])

# Save as CSV
df.to_csv("smart_metrics.csv", index=False)
```

print("SMART data extracted and saved as smart_metrics.csv")

Run the script:

python extract_smart_data.py

Step 4: Train Machine Learning Model

4.1 Load and Prepare Data

Create train_model.py:

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

 # Load dataset (Assuming you have past failure data)

df = pd.read_csv("disk_failure_dataset.csv")

 # Define features and labels

X = df.drop(columns=["failure"]) # Features: SMART attributes

y = df["failure"] # Labels: 0 (healthy), 1 (failed)

 # Split into training and test sets

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

 # Train Random Forest Model

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

model.fit(X_train, y_train)

 # Predict and check accuracy

y_pred = model.predict(X_test)

accuracy = accuracy_score(y_test, y_pred)

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

 # Save model

import joblib

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

print("Model saved as disk_failure_model.pkl")

Run the script:

python train_model.py

Step 5: Predict Disk Failure in Real-Time

5.1 Create Prediction Script

Create predict_failure.py:

```python
import joblib
import pandas as pd
import subprocess

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

# Function to get live SMART data
def get_smart_metrics():
    result = subprocess.run(["sudo", "smartctl", "-A", "/dev/sda"], capture_output=True, text=True)
    data = {}
    for line in result.stdout.split("\n"):
        parts = line.split()
        if len(parts) > 9 and parts[0].isdigit():
            attr_id = int(parts[0])
```

value = int(parts[9])

data[attr_id] = value

return data

 # Get live disk data

smart_metrics = get_smart_metrics()

df_live = pd.DataFrame([smart_metrics])

 # Predict failure

prediction = model.predict(df_live)

status = "Failure Predicted! Backup your data!" if prediction[0] == 1 else "Disk is healthy."

print(status)

Run the script:

python predict_failure.py

Step 6: Automate with a Cron Job

To automate failure detection, schedule a cron job:

crontab -e

Add this line to run the prediction script every hour:

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

Step 7: Visualizing Disk Health Metrics

Create visualize_metrics.py:

python

```
import pandas as pd

import matplotlib.pyplot as plt

df = pd.read_csv("disk_failure_dataset.csv")

# Plot SMART attribute trends

plt.figure(figsize=(10, 6))

plt.plot(df["Power_On_Hours"], df["Reallocated_Sector_Ct"], marker="o", label="Reallocated Sectors")

plt.xlabel("Power-On Hours")

plt.ylabel("Reallocated Sectors")

plt.title("Disk Health Over Time")
```

plt.legend()

plt.show()

Run:

python visualize_metrics.py

Conclusion

We successfully:
✅ Collected SMART disk metrics
✅ Trained an ML model to predict failures
✅ Automated real-time failure prediction
✅ Visualized disk health trends

Project 5. Smart CI/CD Failure Prediction: Train an AI model to analyze Jenkins pipeline logs and predict build failures before they occur.

Introduction

CI/CD pipelines are critical in modern DevOps workflows, but frequent build failures slow down development. This project aims to **train an AI model** to analyze Jenkins pipeline logs and predict build failures before they happen, helping teams take preventive action.

We will:

- Collect Jenkins logs
- Preprocess and clean data
- Train an AI/ML model

- Deploy the model in a Jenkins pipeline for real-time failure prediction

---

Step-by-Step Guide

Step 1: Set Up Your Environment

Install Required Tools

Ensure you have:

- Python (3.8+)
- Jenkins (with logs available)
- Docker (optional for containerization)
- Jupyter Notebook (for model development)

Install dependencies:

pip install pandas numpy scikit-learn joblib flask

Step 2: Collect Jenkins Logs

Jenkins stores logs in /var/log/jenkins/jenkins.log or you can extract them from the Jenkins API.

To get logs using API:

curl -u USER:TOKEN
http://JENKINS_URL/job/JOB_NAME/lastBuild/consoleText > logs.txt

---

Step 3: Preprocess the Logs

Load and clean log data in Python:

```python
import pandas as pd
import re

def load_logs(file_path):
    with open(file_path, 'r') as f:
        logs = f.readlines()
    return logs

def preprocess_logs(logs):
    cleaned_logs = []
    for log in logs:
        log = re.sub(r'\d+', '', log)  # Remove numbers
        log = log.lower().strip()  # Convert to lowercase
        cleaned_logs.append(log)
    return cleaned_logs

logs = load_logs("logs.txt")
cleaned_logs = preprocess_logs(logs)
```

Step 4: Prepare Data for AI Model

Convert logs into numerical features for AI training.

python

```
from sklearn.feature_extraction.text import CountVectorizer

vectorizer = CountVectorizer()
X = vectorizer.fit_transform(cleaned_logs)
```

- X is now a matrix representation of logs for training.

Label **failed builds** as 1 and **successful builds** as 0:

python

```
y = [1 if 'error' in log or 'failed' in log else 0 for log in cleaned_logs]
```

Step 5: Train an AI Model

Use **Logistic Regression** to predict failures.

python

```
from sklearn.model_selection import train_test_split
```

```
from sklearn.linear_model import LogisticRegression

from sklearn.metrics import accuracy_score


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


model = LogisticRegression()

model.fit(X_train, y_train)


y_pred = model.predict(X_test)

print("Model Accuracy:", accuracy_score(y_test, y_pred))
```

Step 6: Save & Deploy the Model

Save the model:

```
python

import joblib

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

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

Step 7: Deploy AI Model in a Flask API

Create app.py to expose an API:

python

```
from flask import Flask, request, jsonify
import joblib

app = Flask(__name__)

model = joblib.load("failure_predictor.pkl")
vectorizer = joblib.load("vectorizer.pkl")

@app.route('/predict', methods=['POST'])
def predict():
    data = request.json['log']
    transformed_log = vectorizer.transform([data])
    prediction = model.predict(transformed_log)
    return jsonify({"failure": bool(prediction[0])})

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

Run the API:

python app.py

Step 8: Integrate AI Model into Jenkins

Modify your Jenkinsfile to send logs to the API:

groovy

```
pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                script {
                    def logText = sh(script: 'cat logs.txt', returnStdout: true).trim()
                    def response = sh(script: """
                        curl -X POST http://localhost:5000/predict -H "Content-Type: application/json" \
                        -d '{"log": "${logText}"}'
                    """, returnStdout: true).trim()

                    def failure = readJSON text: response
                    if (failure.failure) {
                        error "Build Failure Predicted! Stopping pipeline..."
```

```
                }
            }
        }
    }
  }
}
```

Step 9: Test Your Pipeline

Trigger a Jenkins build and check if the AI model predicts failures correctly.

Conclusion

This project helps prevent build failures in CI/CD by analyzing logs with AI. You can further:

- Train with real historical build logs.
- Use advanced NLP models (e.g., BERT) for better accuracy.
- Integrate with Slack for alerts.

---

4. Incident Prediction and Root Cause Analysis

Project 1. Incident Prediction & Root Cause Analysis: Build a machine learning model that predicts system failures based on historical monitoring data.

Introduction

Incident prediction and root cause analysis help organizations prevent system failures by leveraging machine learning on historical monitoring data. This project involves collecting system logs, training a model to predict failures, and providing insights into root causes.

Step 1: Setup Environment

Install Required Dependencies

Ensure you have Python and necessary libraries installed:

pip install pandas numpy scikit-learn matplotlib seaborn xgboost

---

Step 2: Data Collection

For this project, we'll assume a dataset containing system metrics like CPU usage, memory, disk I/O, network traffic, and failure logs. You can generate synthetic data if no real dataset is available.

Sample Dataset (system_logs.csv)

yaml

```
timestamp,cpu_usage,memory_usage,disk_io,network_traffic,error_code

2024-02-01 10:00:00,70,65,120,300,0

2024-02-01 10:05:00,85,75,140,400,1

2024-02-01 10:10:00,90,80,160,450,1

...
```

- error_code=1 → System failure
- error_code=0 → No failure

Step 3: Load & Preprocess Data

Python Code for Data Loading

python

```
import pandas as pd

import numpy as np


# Load data

df = pd.read_csv("system_logs.csv")


# Convert timestamp to datetime

df['timestamp'] = pd.to_datetime(df['timestamp'])


# Check for missing values

df.fillna(df.mean(), inplace=True)


print(df.head())
```

Step 4: Exploratory Data Analysis (EDA)

Before model training, visualize data trends.

Data Distribution

python

```
import matplotlib.pyplot as plt

import seaborn as sns


# Plot CPU Usage vs Failures

plt.figure(figsize=(8,5))

sns.boxplot(x=df["error_code"], y=df["cpu_usage"])

plt.title("CPU Usage vs System Failures")

plt.show()
```

Step 5: Feature Engineering

Convert categorical variables and scale numerical features.

python

```
from sklearn.preprocessing import StandardScaler


features = ["cpu_usage", "memory_usage", "disk_io", "network_traffic"]
```

X = df[features]

y = df["error_code"]

Scale data

scaler = StandardScaler()

X_scaled = scaler.fit_transform(X)

Step 6: Train Machine Learning Model

Using XGBoost for Prediction

python

from xgboost import XGBClassifier

from sklearn.model_selection import train_test_split

from sklearn.metrics import accuracy_score

 # Split data

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

 # Train model

```python
model = XGBClassifier()
model.fit(X_train, y_train)

# Predictions
y_pred = model.predict(X_test)

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

Step 7: Root Cause Analysis

Find key features contributing to failures.

python

```python
importances = model.feature_importances_

# Plot feature importance
plt.figure(figsize=(8,5))
sns.barplot(x=features, y=importances)
plt.title("Feature Importance in System Failures")
plt.show()
```

Interpretation:

- If CPU Usage has the highest importance, optimizing CPU-heavy processes can reduce failures.
- If Memory Usage is critical, increasing RAM or memory management tuning might help.

Step 8: Deployment (Optional)

You can deploy the model as a REST API using Flask:

Flask API for Real-time Prediction

python

```python
from flask import Flask, request, jsonify

import numpy as np


app = Flask(__name__)


@app.route('/predict', methods=['POST'])

def predict():

    data = request.json

    features = np.array([data["cpu_usage"], data["memory_usage"], data["disk_io"], 
data["network_traffic"]]).reshape(1, -1)

    prediction = model.predict(features)
```

```python
    return jsonify({"failure_prediction": int(prediction[0])})


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

Step 9: Run & Test API

Start the API:

```
python app.py
```

Test API with curl:

```
curl -X POST http://127.0.0.1:5000/predict -H "Content-Type: application/json" -d
'{"cpu_usage": 90, "memory_usage": 85, "disk_io": 180, "network_traffic": 500}'
```

Conclusion

This project provides a **real-world approach** to predicting system failures using ML and performing root cause analysis. You can extend it with **real-time monitoring, alert systems, or integrations with DevOps tools**.

Project 2. AI-Based Root Cause Analysis (RCA): Build a model that correlates incidents, logs, and metrics to identify the root cause of failures.

Introduction

Root Cause Analysis (RCA) is crucial in IT operations to diagnose failures by analyzing logs, metrics, and incidents. An AI-based RCA system automates this process using machine learning, helping teams quickly identify and resolve issues. In this project, we will develop a model that processes logs and metrics to determine the root cause of failures.

Project Steps

1. Setup Environment

Ensure Python and necessary dependencies are installed.

 # Update packages

sudo apt update && sudo apt upgrade -y

 # Install Python and virtual environment

sudo apt install python3 python3-pip python3-venv -y

 # Create and activate a virtual environment

python3 -m venv rca_env

source rca_env/bin/activate

 # Install required Python libraries

pip install numpy pandas scikit-learn tensorflow keras matplotlib seaborn loguru

---

2. Prepare Dataset

We will use synthetic log data or fetch logs from a real system.

python

```
import pandas as pd


# Simulated log data

data = {

    "timestamp": ["2024-02-10 10:00:00", "2024-02-10 10:01:00", "2024-02-10 10:02:00"],

    "service": ["Database", "API", "Server"],

    "log_message": ["Timeout error", "Slow response", "CPU overload"],

    "error_level": ["High", "Medium", "Critical"]

}


df = pd.DataFrame(data)

print(df.head())
```

---

3. Data Preprocessing

Convert text-based logs into numerical form using NLP techniques like TF-IDF.

python

```
from sklearn.feature_extraction.text import TfidfVectorizer

vectorizer = TfidfVectorizer()
log_features = vectorizer.fit_transform(df["log_message"])

print("Transformed log messages:", log_features.toarray())
```

---

4. Build Machine Learning Model

Use a simple classification model to identify failure patterns.

python

```
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier

# Simulated labels for training
labels = [1, 0, 1]  # 1 = Failure, 0 = No Failure

X_train, X_test, y_train, y_test = train_test_split(log_features, labels,
test_size=0.2, random_state=42)
```

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

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

---

5. Predict Root Causes

Make predictions on new log entries.

python

```
new_logs = ["Database connection lost", "Server overheating detected"]
new_features = vectorizer.transform(new_logs)

predictions = model.predict(new_features)
print("Predictions:", predictions)
```

---

6. Visualizing Results

Plot logs and failure trends.

python

```
import matplotlib.pyplot as plt

failure_counts = df["error_level"].value_counts()
failure_counts.plot(kind="bar", title="Error Levels Distribution", color=["red", "orange", "green"])
plt.show()
```

Explanation for Beginners

1. **Data Collection:** Logs from system services (Database, API, Server) are collected.
2. **Preprocessing:** Logs are converted into numerical form using TF-IDF.
3. **Model Training:** A RandomForest model is trained to detect failure patterns.
4. **Prediction:** The model predicts potential failures in new logs.
5. **Visualization:** Error levels are visualized for better insights.

This project provides a foundational AI-based RCA system, and it can be extended with deep learning models and real-time log streaming.

---

5. Security and Compliance

Project 1. Automated Security Policy Enforcement with AI: Use AI to detect misconfigurations in firewall rules, IAM policies, and network security.

Project Overview

This project automates security policy enforcement using AI by detecting misconfigurations in firewall rules, IAM policies, and network security settings. It leverages machine learning to analyze security policies and identify potential risks. The project can be integrated into DevOps pipelines to ensure continuous security compliance.

Project Implementation Steps

Step 1: Setup Environment

Ensure you have the necessary tools installed:

- Python 3.8+
- Virtual environment (venv)
- AWS CLI (for IAM policy analysis)
- Docker (for containerizing the application)
- Terraform (optional, for managing infrastructure)

Commands to Install Dependencies

 # Update the system

sudo apt update && sudo apt upgrade -y

 # Install Python and Virtual Environment

sudo apt install python3 python3-venv -y

 # Create a virtual environment

python3 -m venv venv

source venv/bin/activate

 # Install dependencies

pip install boto3 scikit-learn pandas requests flask

Step 2: Define AI Model for Policy Analysis

We will use machine learning to classify security configurations as **secure** or **misconfigured**.

Code: ai_security_model.py

python

```
import pandas as pd

from sklearn.ensemble import RandomForestClassifier

import joblib


# Sample dataset for training

data = {

    "rule_id": [1, 2, 3, 4, 5],

    "port": [22, 80, 443, 8080, 3389],

    "action": [1, 0, 1, 0, 1],  # 1 = Allow, 0 = Deny

    "risk_level": [3, 1, 2, 4, 5]  # Higher is riskier

}
```

df = pd.DataFrame(data)

 # Define features and labels

X = df[["port", "action"]]

y = df["risk_level"]

 # Train model

model = RandomForestClassifier()

model.fit(X, y)

 # Save model

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

print("Model trained and saved successfully.")

📌 **Explanation:**

- Creates a sample dataset with firewall rules
- Uses RandomForestClassifier to train a security risk model
- Saves the trained model for later use

Step 3: Detect Misconfigurations in IAM Policies

Using AWS IAM policies, we check for excessive permissions.

Code: iam_policy_checker.py

python

```python
import boto3
import json

# Initialize AWS IAM client
iam = boto3.client("iam")

def check_policy(policy_arn):
    policy = iam.get_policy(PolicyArn=policy_arn)
    policy_version = iam.get_policy_version(
        PolicyArn=policy_arn,
        VersionId=policy["Policy"]["DefaultVersionId"]
    )

    document = policy_version["PolicyVersion"]["Document"]

    # Analyze permissions
    for statement in document["Statement"]:
        if statement["Effect"] == "Allow" and statement["Action"] == "*":
            print(f"Warning: Overly permissive policy detected in {policy_arn}")
```

check_policy("arn:aws:iam::aws:policy/AdministratorAccess")

📌 **Explanation:**

- Retrieves IAM policies from AWS
- Checks for overly permissive permissions ("Action": "*")

🔹 **Commands to Run:**

export AWS_ACCESS_KEY_ID="your-access-key"

export AWS_SECRET_ACCESS_KEY="your-secret-key"

export AWS_REGION="us-east-1"

python iam_policy_checker.py

Step 4: Firewall Rule Misconfiguration Detection

This script analyzes firewall rules to detect open ports.

Code: firewall_analyzer.py

python

import json

firewall_rules = """

[

```
    {"port": 22, "protocol": "TCP", "action": "ALLOW"},
    {"port": 3389, "protocol": "TCP", "action": "ALLOW"},
    {"port": 443, "protocol": "TCP", "action": "ALLOW"}
]
"""

rules = json.loads(firewall_rules)

for rule in rules:
    if rule["port"] in [22, 3389]:
        print(f"Warning: High-risk port {rule['port']} is open.")
```

📌 **Explanation:**

- Reads firewall rules
- Detects risky open ports (22 for SSH, 3389 for RDP)

Step 5: Containerize the Application

Use Docker to package the security tool.

Dockerfile

```dockerfile
FROM python:3.8
```

```
WORKDIR /app

COPY . /app

RUN pip install -r requirements.txt

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

🔹 **Commands to Build and Run:**

```
docker build -t security-check .

docker run security-check
```

Step 6: Automate in CI/CD Pipeline (Jenkinsfile)

groovy

```groovy
pipeline {
    agent any
    stages {
        stage('Checkout') {
            steps {
                git 'https://github.com/your-repo/security-policy-check.git'
            }
        }
```

```
        stage('Run Security Checks') {
            steps {
                sh 'python firewall_analyzer.py'
                sh 'python iam_policy_checker.py'
            }
        }
        stage('Deploy') {
            steps {
                sh 'docker build -t security-check .'
                sh 'docker run security-check'
            }
        }
    }
}
```

📌 **Explanation:**

- Pulls code from GitHub
- Runs security scripts
- Builds and runs Docker container

Step 7: Monitor Security Violations with Grafana & Prometheus

Use Prometheus to log security findings and visualize in Grafana.

Commands to Set Up Prometheus

docker run -d -p 9090:9090 --name=prometheus prom/prometheus

Commands to Set Up Grafana

docker run -d -p 3000:3000 --name=grafana grafana/grafana

This project provides a full-stack **AI-powered security enforcement tool** that detects misconfigurations in firewall rules and IAM policies. You can integrate it with CI/CD for **automated security compliance** and **visual monitoring** using **Grafana and Prometheus**.

Project 2. AI-Powered SLA Compliance Monitoring: Analyze service response times and uptime metrics using ML to predict SLA violations.

Service Level Agreements (SLAs) define the expected performance and reliability of a service. This project builds an AI-powered monitoring system that analyzes response times and uptime metrics, using machine learning (ML) to predict SLA violations. It helps businesses proactively address performance issues before breaching SLAs.

Project Overview

We will develop a Python-based solution using Flask for the API, PostgreSQL for data storage, and Scikit-learn for ML-based SLA violation prediction. The system will:

- Collect real-time service response times and uptime metrics
- Store data in PostgreSQL
- Train an ML model to predict SLA violations

- Visualize insights using Grafana

Step-by-Step Implementation

Step 1: Install Dependencies

Ensure your system has Python and PostgreSQL installed. Then, install the required Python libraries:

```
pip install flask psycopg2 pandas scikit-learn requests matplotlib grafana-api
```

Step 2: Set Up PostgreSQL Database

Create the database and table to store service metrics.

```
CREATE DATABASE sla_monitor;

\c sla_monitor


CREATE TABLE service_metrics (

    id SERIAL PRIMARY KEY,

    timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP,

    response_time FLOAT,

    uptime BOOLEAN

);
```

Step 3: Create a Flask API to Collect Metrics

Create a server.py file to collect and store service metrics.

python

```python
from flask import Flask, request, jsonify

import psycopg2

from datetime import datetime


app = Flask(__name__)


# Database connection

conn = psycopg2.connect("dbname=sla_monitor user=postgres
password=yourpassword")

cur = conn.cursor()


@app.route('/metrics', methods=['POST'])

def collect_metrics():

    data = request.get_json()

    response_time = data['response_time']

    uptime = data['uptime']
```

```
    cur.execute("INSERT INTO service_metrics (response_time, uptime) VALUES (%s, %s)", (response_time, uptime))
    conn.commit()

    return jsonify({"message": "Metrics saved!"}), 201

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

Step 4: Collect Metrics from a Service

Write a script to simulate collecting data from an API:

python

```
import requests
import time
import random

API_URL = "http://127.0.0.1:5000/metrics"

while True:
```

```
    response_time = round(random.uniform(100, 1000), 2)  # Simulated response time (ms)

    uptime = random.choice([True, False])  # Simulated uptime status


    data = {"response_time": response_time, "uptime": uptime}

    requests.post(API_URL, json=data)


    time.sleep(5)  # Collect metrics every 5 seconds
```

Step 5: Train an ML Model to Predict SLA Violations

Create a script to analyze historical data and predict SLA violations using Scikit-learn.

python

```
import psycopg2

import pandas as pd

from sklearn.model_selection import train_test_split

from sklearn.ensemble import RandomForestClassifier

from sklearn.metrics import accuracy_score


# Connect to database
```

```python
conn = psycopg2.connect("dbname=sla_monitor user=postgres
password=yourpassword")

cur = conn.cursor()


# Load data

cur.execute("SELECT response_time, uptime FROM service_metrics")

data = cur.fetchall()

df = pd.DataFrame(data, columns=['response_time', 'uptime'])


# Prepare data

X = df[['response_time']]

y = df['uptime'].astype(int)


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)

model.fit(X_train, y_train)


# Test model

y_pred = model.predict(X_test)
```

print(f"Model Accuracy: {accuracy_score(y_test, y_pred) * 100:.2f}%")

 # Save model

import joblib

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

Step 6: Deploy ML Model as an API

Modify server.py to include an endpoint for prediction.

python

```
import joblib

import numpy as np


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


@app.route('/predict', methods=['POST'])

def predict_sla_violation():

    data = request.get_json()

    response_time = np.array(data['response_time']).reshape(-1, 1)
```

```
    prediction = model.predict(response_time)

    return jsonify({"sla_violation": bool(prediction[0])})
```

Step 7: Visualize Metrics with Grafana

Install Grafana:

```
sudo apt update

sudo apt install -y grafana

sudo systemctl start grafana-server
```

- Connect PostgreSQL to Grafana and create dashboards for response times and SLA violations.

Conclusion

This project builds an end-to-end AI-powered SLA monitoring system. It collects real-time metrics, trains an ML model, and predicts SLA violations while providing a Grafana dashboard for visualization.

---

6. Self-Healing and Automation

Project 1. Self-Healing Infrastructure: Use AI to detect and auto-remediate cloud infrastructure issues (e.g., restarting failed pods in Kubernetes).

Self-healing infrastructure is an approach where cloud environments **automatically detect and remediate failures** without human intervention. This ensures **high availability, reduced downtime, and improved system reliability**.

In this project, we will:

- Use **Prometheus** to monitor Kubernetes pods.
- Apply **AI/ML models** to predict failures.
- Use **Python automation** to trigger remediation (e.g., restarting failed pods).

Step-by-Step Implementation

Step 1: Set Up a Kubernetes Cluster

If you don't have a cluster, you can use kind (Kubernetes in Docker):

kind create cluster --name self-healing-cluster

To verify:

kubectl get nodes

Step 2: Deploy Prometheus for Monitoring

1. **Install Prometheus in Kubernetes:**

kubectl create namespace monitoring

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

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

2. **Verify the installation:**

kubectl get pods -n monitoring

Step 3: Deploy a Sample Application

Let's create a simple Nginx deployment to test self-healing:

yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
```

```yaml
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
```

Apply the deployment:

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

Step 4: Create an AI-based Failure Prediction Model

We will use a **simple Python AI model** to detect failures using Prometheus metrics.

1. **Install dependencies:**

```
pip install requests pandas scikit-learn
```

2. **Python script to collect metrics from Prometheus**:

python

```python
import requests

import pandas as pd

from sklearn.ensemble import RandomForestClassifier
```

```
import time
import json
import os

PROMETHEUS_URL = "http://localhost:9090/api/v1/query"

def fetch_pod_status():
    query = 'kube_pod_status_ready'
    response = requests.get(PROMETHEUS_URL, params={'query': query})
    data = response.json()

    pod_data = []
    for result in data['data']['result']:
        pod_name = result['metric']['pod']
        status = int(result['value'][1])  # 1 = Running, 0 = Failed
        pod_data.append([pod_name, status])

    return pd.DataFrame(pod_data, columns=['pod', 'status'])

def train_model():
    df = fetch_pod_status()
```

```
    X = df[['status']]
    y = df['status']  # Labels: 1 (healthy), 0 (failed)

    model = RandomForestClassifier()
    model.fit(X, y)

    return model

def detect_failure(model):
    df = fetch_pod_status()
    failed_pods = df[df['status'] == 0]['pod'].tolist()

    if failed_pods:
        print(f"Detected failed pods: {failed_pods}")
        for pod in failed_pods:
            restart_pod(pod)

def restart_pod(pod_name):
    print(f"Restarting pod: {pod_name}")
    os.system(f"kubectl delete pod {pod_name}")
```

```
if __name__ == "__main__":
    model = train_model()

    while True:
        detect_failure(model)
        time.sleep(10)
```

Step 5: Automate Self-Healing with a Kubernetes CronJob

1. **Create a Kubernetes CronJob to run the script periodically:**

yaml

```yaml
apiVersion: batch/v1
kind: CronJob
metadata:
  name: self-healing
spec:
  schedule: "*/1 * * * *"  # Runs every minute
  jobTemplate:
    spec:
      template:
```

```yaml
      spec:
        containers:
        - name: self-healing
          image: python:3.9
          command: ["python", "/app/self-healing.py"]
          volumeMounts:
          - name: script-volume
            mountPath: /app
        volumes:
        - name: script-volume
          configMap:
            name: self-healing-script
        restartPolicy: OnFailure
```

2. **Apply the CronJob**

```
kubectl apply -f self-healing-cronjob.yaml
```

Step 6: Test the Self-Healing System

1. **List the running pods:**

```
kubectl get pods
```

2. **Manually delete a pod to simulate failure:**

kubectl delete pod <nginx-pod-name>

3. **Check if the self-healing system restarts it:**

kubectl get pods

How the Code Works

- The **Python script** fetches Prometheus metrics and predicts failures.
- If a pod is detected as failed (status == 0), the script **automatically restarts** it using kubectl delete pod.
- A **Kubernetes CronJob** ensures the script runs periodically for continuous monitoring.
- The **AI model** is trained on historical data and improves failure predictions over time.

Conclusion

With this project, we created a **self-healing Kubernetes cluster** that: ✔ Monitors pod health using Prometheus
✔ Uses **AI-based failure detection**
✔ Auto-remediates failures using **Python automation**

Project 2. AI-Based Configuration Drift Detection: Build a model that monitors infrastructure-as-code (Terraform, Ansible) for unintended changes.

Configuration drift occurs when infrastructure configurations deviate from their intended state due to manual changes, updates, or other unexpected modifications. This project aims to **automate drift detection** using an AI-based model that identifies anomalies in Terraform and Ansible configurations.

Project Setup & Step-by-Step Execution

Step 1: Install Required Tools

Ensure you have the following installed:

- **Python 3**
- **Terraform**
- **Ansible**
- **Git**
- **Jenkins (Optional, for CI/CD Automation)**

Install Python & Required Libraries

sudo apt update

sudo apt install python3 python3-pip -y

pip install numpy pandas scikit-learn watchdog

Install Terraform

wget -O terraform.zip https://releases.hashicorp.com/terraform/1.6.0/terraform_1.6.0_linux_amd64.zip

unzip terraform.zip

sudo mv terraform /usr/local/bin/

terraform --version

Install Ansible

```
sudo apt install ansible -y
ansible --version
```

Step 2: Create a Terraform Configuration

Create a Terraform script to provision an AWS EC2 instance.

File: main.tf

hcl

```hcl
provider "aws" {
  region = "us-east-1"
}

resource "aws_instance" "web" {
  ami           = "ami-12345678"
  instance_type = "t2.micro"

  tags = {
    Name = "Drift-Detection-Instance"
```

```
  }
}
```

Initialize & Apply Terraform

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

Step 3: Create an Ansible Playbook

Ansible will configure the server.

File: playbook.yml

yaml

```
- name: Configure Web Server
  hosts: all
  become: yes
  tasks:
    - name: Install Nginx
      apt:
        name: nginx
```

state: present

Run Ansible Playbook

ansible-playbook -i inventory.ini playbook.yml

Step 4: Implement AI-Based Drift Detection

We will use **Python** and **Machine Learning** to detect unexpected changes.

File: drift_detector.py

python

import os

import hashlib

import pandas as pd

import numpy as np

from sklearn.ensemble import IsolationForest

 # Function to calculate hash of configuration files

def get_file_hash(file_path):

```python
    hasher = hashlib.md5()
    with open(file_path, "rb") as f:
        hasher.update(f.read())
    return hasher.hexdigest()

# List of configuration files to monitor
config_files = ["main.tf", "playbook.yml"]

# Generate initial baseline hashes
baseline = {file: get_file_hash(file) for file in config_files}

# Function to detect drift
def detect_drift():
    current_hashes = [get_file_hash(file) for file in config_files]
    baseline_hashes = list(baseline.values())

    # Convert to numerical representation
    data = np.array([baseline_hashes, current_hashes])
    df = pd.DataFrame(data.T, columns=["baseline", "current"])

    # Train Isolation Forest for anomaly detection
```

```python
    model = IsolationForest(contamination=0.1)
    model.fit(df)

    # Predict anomalies (drift)
    anomalies = model.predict(df)
    for i, file in enumerate(config_files):
        if anomalies[i] == -1:
            print(f"Configuration drift detected in: {file}")

# Run drift detection
detect_drift()
```

Step 5: Automate Drift Detection with Jenkins

Create a Jenkins pipeline to automate drift detection.

File: Jenkinsfile

groovy

```groovy
pipeline {
    agent any
    stages {
```

```
        stage('Checkout Code') {
            steps {
                git 'https://github.com/your-repo/drift-detection.git'
            }
        }
        stage('Run Drift Detector') {
            steps {
                sh 'python3 drift_detector.py'
            }
        }
    }
}
```

Explanation of Code

1. **Terraform Configuration (main.tf)**
    - Defines an AWS EC2 instance using Terraform.
    - terraform apply provisions the infrastructure.
2. **Ansible Playbook (playbook.yml)**
    - Installs Nginx on the EC2 instance.
    - Ensures infrastructure consistency.
3. **Drift Detector (drift_detector.py)**
    - Uses **MD5 hashing** to detect file changes.
    - Uses **Machine Learning (Isolation Forest)** to detect anomalies.
    - Compares current Terraform & Ansible configurations with the baseline.
4. **Jenkins Pipeline (Jenkinsfile)**
    - Automates drift detection.

    - Runs Python script to check for configuration drifts.

Conclusion

This project **automates drift detection** using **AI-based anomaly detection**. The model continuously monitors **Terraform & Ansible configurations**, alerting when unintended changes occur. By integrating with **Jenkins**, we ensure automated monitoring for infrastructure stability.

---

7. AI for Log Analysis & Monitoring

Project 1. AI-Powered Log Filtering & Categorization: Implementing AI to automatically filter out noise in logs and categorize relevant events for quicker analysis.

- **Introduction**
    - The goal of this project is to build an AI-powered system that processes log data, filters out noise, and categorizes important events using **Python, Machine Learning (ML), and NLP (Natural Language Processing)**.
    - This helps **DevOps engineers, SREs (Site Reliability Engineers), and security teams** quickly analyze logs and detect issues.
    - We'll use **Python, Flask (for API), Scikit-learn, NLP libraries (spaCy or NLTK), and a simple ML model** for classification.

Step-by-Step Guide

1. Set Up the Environment

Install dependencies:

pip install flask pandas numpy scikit-learn nltk spacy
python -m spacy download en_core_web_sm

---

2. Prepare Log Data

Logs are usually in text files. Example:

log

[2024-02-08 12:30:00] ERROR Database connection failed
[2024-02-08 12:31:00] INFO User login successful
[2024-02-08 12:32:00] WARNING Disk space running low

We'll preprocess logs to extract key parts.

---

3. Preprocessing Logs (Python Code)

python

```
import re
import pandas as pd
import spacy

nlp = spacy.load("en_core_web_sm")

# Sample logs
logs = [
    "[2024-02-08 12:30:00] ERROR Database connection failed",
    "[2024-02-08 12:31:00] INFO User login successful",
    "[2024-02-08 12:32:00] WARNING Disk space running low"
]

# Function to clean and extract log messages
```

```
def preprocess_log(log):
    log = re.sub(r"\[.*?\]", "", log)  # Remove timestamp
    return log.strip()

# Process logs
clean_logs = [preprocess_log(log) for log in logs]

# Convert logs to structured format
df = pd.DataFrame({"log": clean_logs})
print(df.head())
```

Explanation:

- We remove timestamps to focus on the message.
- Store logs in a structured format using Pandas.

---

4. Implement AI Model for Categorization

Using **TF-IDF Vectorization + Naïve Bayes Classifier**:

python

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.pipeline import make_pipeline

# Sample log data with labels
data = [
    ("ERROR Database connection failed", "Error"),
    ("INFO User login successful", "Info"),
    ("WARNING Disk space running low", "Warning"),
    ("ERROR Unable to reach API", "Error"),
    ("INFO Server restarted", "Info")
]
```

Splitting logs and labels
texts, labels = zip(*data)

 # Create text classification model
model = make_pipeline(TfidfVectorizer(), MultinomialNB())

 # Train model
model.fit(texts, labels)

 # Test on new log
test_log = ["CRITICAL: System overload detected"]
predicted_category = model.predict(test_log)[0]

print(f"Predicted Category: {predicted_category}")

Explanation:

- **TF-IDF (Term Frequency-Inverse Document Frequency)** converts logs into numerical format.
- **Naïve Bayes** is used for classification.
- The model predicts the category of an unseen log message.

---

5. Build Flask API for Real-Time Log Processing

python

```
from flask import Flask, request, jsonify

app = Flask(__name__)

@app.route('/classify', methods=['POST'])
def classify_log():
    data = request.json
```

```python
    log_message = data.get("log")

    if not log_message:
        return jsonify({"error": "No log provided"}), 400

    category = model.predict([log_message])[0]

    return jsonify({"log": log_message, "category": category})

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

Run the API:

```
python app.py
```

Test API (Using cURL or Postman):

```
curl -X POST http://127.0.0.1:5000/classify -H "Content-Type: application/json" -d
'{"log": "CRITICAL: System overload detected"}'
```

---

6. Deploying the API using Docker

Dockerfile:

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

Build & Run Docker Container:

```
docker build -t log-ai .
docker run -p 5000:5000 log-ai
```

Summary

✔ Preprocessed logs using regex & NLP
✔ Built a text classifier using Naïve Bayes
✔ Created a Flask API for real-time log categorization
✔ Deployed using Docker

Project 2. Real-Time Anomaly Detection in Logs: AI system that processes logs in real time and raises alerts when unusual patterns or behavior are detected.

This project builds an AI-based system that processes logs in real time, detects anomalies, and raises alerts when unusual behavior is found.

Tech Stack

- **Python** (for log processing & AI model)
- **Flask** (to expose API for log ingestion)
- **Scikit-learn** (for anomaly detection)
- **Elasticsearch & Kibana** (for storage & visualization)
- **Docker** (for containerization)

Step 1: Set Up the Environment

1. Install Dependencies

Run the following command:

pip install pandas numpy scikit-learn flask elasticsearch requests

If using Docker for Elasticsearch, run:

docker pull elasticsearch:8.11.2
docker run -d --name es -p 9200:9200 -e "discovery.type=single-node" elasticsearch:8.11.2

To check if Elasticsearch is running:

curl -X GET "localhost:9200"

Step 2: Prepare Log Data

Create a sample log file (logs.json):

json

```
[
   {"timestamp": "2025-02-08T12:00:00", "message": "User login", "status": 200, "response_time": 120},
   {"timestamp": "2025-02-08T12:01:00", "message": "File upload", "status": 200, "response_time": 350},
   {"timestamp": "2025-02-08T12:02:00", "message": "Failed login attempt", "status": 401, "response_time": 90}
]
```

Step 3: Implement Anomaly Detection

We'll use **Isolation Forest**, an unsupervised machine learning algorithm, to detect anomalies in logs.

Create anomaly_detector.py

python

```
import pandas as pd
import numpy as np
from sklearn.ensemble import IsolationForest
import json

class AnomalyDetector:
    def __init__(self):
        self.model = IsolationForest(contamination=0.1, random_state=42)

    def train(self, log_data):
        df = pd.DataFrame(log_data)
        features = df[['status', 'response_time']]
        self.model.fit(features)

    def predict(self, log_entry):
        df = pd.DataFrame([log_entry])
        features = df[['status', 'response_time']]
        result = self.model.predict(features)
        return "Anomaly" if result[0] == -1 else "Normal"
```

Explanation

- We use **Isolation Forest** to detect anomalies.
- The model trains on status and response_time fields.
- When new log data is received, the model predicts if it's an anomaly.

Step 4: Create an API to Ingest Logs

We will use **Flask** to expose an API that receives logs, analyzes them, and stores them in **Elasticsearch**.

Create app.py

python

```python
from flask import Flask, request, jsonify
from elasticsearch import Elasticsearch
from anomaly_detector import AnomalyDetector

app = Flask(__name__)
es = Elasticsearch("http://localhost:9200")
detector = AnomalyDetector()

@app.route('/train', methods=['POST'])
def train():
    data = request.get_json()
    detector.train(data)
    return jsonify({"message": "Model trained successfully"})

@app.route('/log', methods=['POST'])
def log_event():
    data = request.get_json()
    anomaly_result = detector.predict(data)

    # Store in Elasticsearch
    es.index(index="logs", document={"log": data, "anomaly": anomaly_result})

    return jsonify({"status": "logged", "anomaly": anomaly_result})

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

Explanation

- /train API trains the model using past logs.
- /log API:
  - Receives new log entries.
  - Predicts if they are anomalies.
  - Stores the results in **Elasticsearch**.

Step 5: Train and Test the Model

Train the Model

Run:

curl -X POST "http://127.0.0.1:5000/train" -H "Content-Type: application/json" -d @logs.json

Send a New Log for Analysis

curl -X POST "http://127.0.0.1:5000/log" -H "Content-Type: application/json" -d '{
  "timestamp": "2025-02-08T12:10:00", "message": "Unusual traffic spike",
"status": 500, "response_time": 2000
}'

Expected response:

json

{"status": "logged", "anomaly": "Anomaly"}

Step 6: Visualizing in Kibana

If using Kibana:

Start Kibana
docker run -d --name kibana --link es:elasticsearch -p 5601:5601 kibana:8.11.2

Open **http://localhost:5601**, go to **"Discover"**, and view logs.

Step 7: Running the Project

Run the Flask API

python app.py

Test with Log Data

- Use the **/train** API to train.
- Use the **/log** API to detect anomalies.

Conclusion

This project: ✅ Detects anomalies in real-time logs
✅ Uses **Isolation Forest** for AI-based detection
✅ Stores logs in **Elasticsearch** for analysis
✅ Exposes APIs using **Flask**

Project 3. Log Correlation for Performance Issues: Using AI to correlate logs from different services to identify root causes of performance degradation or service outages.

Modern applications generate logs across multiple services, making it difficult to pinpoint performance issues. **Log correlation using AI** helps analyze logs from various sources, detect patterns, and identify root causes of performance degradation or service outages.

In this project, we will:

- Collect logs from multiple services using **Fluentd** or **Filebeat**.
- Store logs in **Elasticsearch** for indexing and searching.
- Use **Python and Machine Learning (ML)** (Scikit-learn) to analyze logs and detect anomalies.
- Visualize insights with **Kibana** or **Grafana**.

Step-by-Step Implementation

Step 1: Setup Log Collection

We use **Fluentd** or **Filebeat** to collect logs from different services.

Install Fluentd (Ubuntu Example)

curl -fsSL https://toolbelt.treasuredata.com/sh/install-ubuntu-bionic-td-agent4.sh | sh

Install Filebeat (Alternative to Fluentd)

sudo apt-get install filebeat

Configure Filebeat to Send Logs to Elasticsearch
Edit /etc/filebeat/filebeat.yml:

yaml

output.elasticsearch:

```
  hosts: ["localhost:9200"]
  username: "elastic"
  password: "yourpassword"
```

Restart Filebeat:

```
sudo systemctl restart filebeat
```

Step 2: Store Logs in Elasticsearch

Install Elasticsearch (Ubuntu Example)

```
sudo apt-get install elasticsearch
sudo systemctl start elasticsearch
sudo systemctl enable elasticsearch
```

Verify installation:

```
curl -X GET "localhost:9200/_cat/indices?v"
```

Step 3: Visualize Logs in Kibana

Install Kibana

```
sudo apt-get install kibana
sudo systemctl start kibana
```

Access Kibana at: http://localhost:5601

Step 4: Implement AI-Based Log Correlation with Python

We use **Python** with **Scikit-learn** to detect performance anomalies.

Install Dependencies

pip install pandas numpy elasticsearch scikit-learn matplotlib seaborn

Python Code for Log Correlation

python

```
import pandas as pd
import numpy as np
from elasticsearch import Elasticsearch
from sklearn.ensemble import IsolationForest
import matplotlib.pyplot as plt
import seaborn as sns

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

# Fetch logs from Elasticsearch
query = {
    "size": 1000,
    "query": {
        "range": {
            "@timestamp": {
                "gte": "now-1d/d",
                "lt": "now/d"
            }
        }
    }
}

response = es.search(index="logs", body=query)
```

```python
logs = [hit["_source"] for hit in response["hits"]["hits"]]

# Convert logs to DataFrame
df = pd.DataFrame(logs)

# Feature extraction (Example: Response time)
df['response_time'] = df['message'].str.extract(r'Response time: (\d+)').astype(float)

# Detect anomalies using Isolation Forest
model = IsolationForest(contamination=0.05)
df['anomaly'] = model.fit_predict(df[['response_time']])

# Visualize anomalies
plt.figure(figsize=(10, 5))
sns.scatterplot(data=df, x=df.index, y="response_time", hue="anomaly",
palette={1: 'blue', -1: 'red'})
plt.title("Log Correlation for Performance Issues")
plt.show()

# Print potential issues
anomalies = df[df['anomaly'] == -1]
print("Potential performance issues detected:")
print(anomalies)
```

Step 5: Automate and Deploy the Solution

Run Python script every 5 minutes using cron:

```
crontab -e
```

Add:

```
*/5 * * * * /usr/bin/python3 /home/user/log_analysis.py
```

Deploy with Docker (Optional)

```
docker build -t log-analysis .
docker run -d --name log_analysis log-analysis
```

Explanation of Code

1. **Connect to Elasticsearch** to fetch logs.
2. **Extract performance metrics** (e.g., response time).
3. **Use Machine Learning (Isolation Forest)** to detect anomalies.
4. **Visualize performance issues** using Matplotlib.
5. **Print logs** of potential issues for debugging.

This project helps **DevOps teams** correlate logs, detect bottlenecks, and prevent outages.

Project 4. AI-Based Multi-Source Log Aggregation: Aggregating logs from diverse sources (cloud, on-prem, containers, etc.) using AI to spot cross-system anomalies.

Log aggregation is crucial for monitoring applications running in different environments like cloud, on-premises, and containers. This project builds an **AI-powered log aggregation system** that:

- Collects logs from multiple sources (AWS CloudWatch, Kubernetes logs, local files, etc.)
- Uses **Elasticsearch** for storage and **Kibana** for visualization
- Applies **AI (Machine Learning)** to detect anomalies in logs

---

Tech Stack

- **Python** (Flask for API, Pandas for data processing)
- **ELK Stack (Elasticsearch, Logstash, Kibana)**
- **Docker & Kubernetes** (for deployment)
- **Machine Learning** (scikit-learn for anomaly detection)

Project Setup with All Commands

1. Install Dependencies

Ensure Python, Docker, and Elasticsearch are installed.

 # Install Python dependencies
pip install flask pandas elasticsearch scikit-learn docker

2. Set Up Elasticsearch & Kibana

 # Pull and run Elasticsearch
docker run -d --name elasticsearch -p 9200:9200 -e "discovery.type=single-node" docker.elastic.co/elasticsearch/elasticsearch:8.0.0

 # Pull and run Kibana
docker run -d --name kibana -p 5601:5601 --link elasticsearch docker.elastic.co/kibana/kibana:8.0.0

3. Deploy Logstash

Create a logstash.conf file to read logs from various sources and push to Elasticsearch:

input {

```
  file {
    path => "/var/log/app.log"
    start_position => "beginning"
  }
}
filter {
  grok {
    match => { "message" => "%{TIMESTAMP_ISO8601:timestamp}
%{LOGLEVEL:level} %{GREEDYDATA:msg}" }
  }
}
output {
  elasticsearch {
    hosts => ["http://elasticsearch:9200"]
    index => "logs"
  }
}
```

Run Logstash:

```
docker run --rm -v $(pwd)/logstash.conf:/usr/share/logstash/pipeline/logstash.conf
--link elasticsearch logstash:8.0.0
```

4. Python Flask API to Aggregate Logs

Create app.py:

python

```python
from flask import Flask, request, jsonify
from elasticsearch import Elasticsearch

app = Flask(__name__)
es = Elasticsearch(["http://localhost:9200"])
```

```python
@app.route("/logs", methods=["POST"])
def ingest_logs():
    log_data = request.json
    es.index(index="logs", body=log_data)
    return jsonify({"message": "Log received"}), 200

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

Run API

```
python app.py
```

5. AI-Based Anomaly Detection

Create anomaly_detection.py:

python

```python
import pandas as pd
import numpy as np
from sklearn.ensemble import IsolationForest

def detect_anomalies(logs):
    df = pd.DataFrame(logs)
    df["length"] = df["message"].apply(len)

    model = IsolationForest(contamination=0.1)
    df["anomaly"] = model.fit_predict(df[["length"]])

    anomalies = df[df["anomaly"] == -1]
    return anomalies.to_dict(orient="records")
```

Use it in Flask API:

python

```
@app.route("/anomalies", methods=["GET"])
def get_anomalies():
    logs = es.search(index="logs", size=1000)["hits"]["hits"]
    log_messages = [{"message": log["_source"]["msg"]} for log in logs]
    anomalies = detect_anomalies(log_messages)
    return jsonify(anomalies)
```

6. Testing the System

```
# Send a sample log
curl -X POST "http://localhost:5000/logs" -H "Content-Type: application/json" -d
'{"message": "Error: Connection timeout"}'

# Get detected anomalies
curl -X GET "http://localhost:5000/anomalies"
```

Code Explanation

1. Flask API for Log Collection

- **Flask** is used to create API endpoints
- **/logs** endpoint receives logs and stores them in Elasticsearch

2. Elasticsearch for Log Storage

- Used to index and store log data
- Querying Elasticsearch retrieves logs for AI processing

3. Machine Learning for Anomaly Detection

- **IsolationForest** is trained to identify unusual log patterns

- It assigns **-1 (anomaly)** or **1 (normal)** based on log message lengths

---

Project 5. Automated Log Tagging: Using AI to automatically tag logs with metadata for faster identification and analysis.

Log files contain valuable insights, but manually analyzing them can be time-consuming. This project leverages **AI/ML** to **automatically tag logs** with metadata like severity, source, and category. This helps in **faster identification, filtering, and analysis** in DevOps and security monitoring.

Project Workflow

1. Collect log data
2. Preprocess logs (cleaning, tokenization)
3. Train an AI model to classify logs
4. Use the trained model to tag new logs automatically
5. Store results for further analysis

Step-by-Step Implementation

Step 1: Set Up the Environment

Ensure Python and required libraries are installed.

```
mkdir automated-log-tagging
cd automated-log-tagging
python3 -m venv env
source env/bin/activate  # On Windows: env\Scripts\activate
pip install pandas numpy scikit-learn nltk joblib
```

Step 2: Prepare Sample Log Data

Create a sample log file logs.txt:

```
nano logs.txt
```

Add some sample logs:

```pgsql
[ERROR] 2025-02-08 12:00:01 Database connection failed.
[INFO] 2025-02-08 12:05:02 User logged in successfully.
[WARNING] 2025-02-08 12:10:03 High memory usage detected.
[ERROR] 2025-02-08 12:15:04 Unauthorized access attempt.
```

Step 3: Preprocess the Log Data

Create preprocess.py to clean and prepare the logs.

```python
import re
import pandas as pd
import nltk
from nltk.tokenize import word_tokenize
nltk.download('punkt')

def preprocess_log(log):
    """Clean and tokenize logs"""
    log = re.sub(r"[\[\]]", "", log)  # Remove brackets
    log = log.lower()
    tokens = word_tokenize(log)
    return " ".join(tokens)

def load_logs(filename):
    """Load logs from file"""
```

```
    with open(filename, "r") as file:
        logs = file.readlines()
    return [preprocess_log(log.strip()) for log in logs]

if __name__ == "__main__":
    logs = load_logs("logs.txt")
    df = pd.DataFrame(logs, columns=["log"])
    df.to_csv("processed_logs.csv", index=False)
    print("Logs preprocessed and saved.")
```

Run the script

```
python preprocess.py
```

Step 4: Train a Simple AI Model

Create train_model.py to train a log classifier using scikit-learn.

python

```
import pandas as pd
import joblib
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.pipeline import make_pipeline
from sklearn.model_selection import train_test_split

# Load processed logs
df = pd.read_csv("processed_logs.csv")

# Add labels manually (ERROR, INFO, WARNING)
df["label"] = ["ERROR", "INFO", "WARNING", "ERROR"]
```

Split data
X_train, X_test, y_train, y_test = train_test_split(df["log"], df["label"], test_size=0.2, random_state=42)

# Create model pipeline
model = make_pipeline(TfidfVectorizer(), MultinomialNB())

# Train model
model.fit(X_train, y_train)

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

print("Model trained and saved.")

Run the training:

python train_model.py

Step 5: Automatically Tag New Logs

Create tag_logs.py to tag logs using the trained model.

python

import joblib
import pandas as pd

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

def tag_log(log):
    """Predict log category"""

```
    return model.predict([log])[0]

# Load new logs
df = pd.read_csv("processed_logs.csv")
df["predicted_label"] = df["log"].apply(tag_log)

# Save results
df.to_csv("tagged_logs.csv", index=False)
print("Logs tagged and saved.")
```

Run the tagging:

```
python tag_logs.py
```

Step 6: View Tagged Logs

```
cat tagged_logs.csv
```

Example Output:

pgsql

```
log,predicted_label
"error 2025-02-08 database connection failed.",ERROR
"info 2025-02-08 user logged in successfully.",INFO
"warning 2025-02-08 high memory usage detected.",WARNING
"error 2025-02-08 unauthorized access attempt.",ERROR
```

- **Data Preprocessing:** Cleans logs by removing unwanted characters and tokenizing words.

- **Model Training:** Uses **TF-IDF (Term Frequency-Inverse Document Frequency)** for feature extraction and **Naïve Bayes** for classification.
- **Log Tagging:** Predicts the category (ERROR, INFO, WARNING) for new logs.

---

8. AI for Predictive Scaling & Performance Optimization

Project 1. Predictive Load Balancing: AI model that predicts incoming traffic and adjusts load balancing strategies accordingly to optimize resource usage and minimize latency.

Load balancing distributes network traffic across multiple servers to ensure no single server is overwhelmed. Traditional load balancing techniques rely on static rules or real-time traffic metrics. However, predictive load balancing uses **AI/ML models** to anticipate traffic surges and adjust strategies proactively, **minimizing latency and optimizing resource usage**.

Key Technologies Used:

- **Python** (for AI model and API)
- **Flask** (to serve predictions)
- **Scikit-learn / TensorFlow** (for training ML models)
- **Nginx / HAProxy** (as load balancers)
- **Docker & Kubernetes** (for deployment)
- **Prometheus & Grafana** (for monitoring)

Step 1: Setting Up the Environment

Before starting, install the required dependencies:

 # Update system and install required packages
sudo apt update && sudo apt install python3 python3-pip docker-compose -y

 # Install Python dependencies
pip3 install flask numpy pandas scikit-learn tensorflow joblib requests

---

Step 2: Building the AI Model

The AI model predicts traffic based on historical data.

2.1: Create Training Data

Create a dataset (traffic_data.csv) with columns: time, requests_per_minute, cpu_usage, memory_usage, response_time, and server_allocation.

python

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

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

 # Define features and target variable
X = df[['time', 'requests_per_minute', 'cpu_usage', 'memory_usage', 'response_time']]
y = df['server_allocation']

```
# 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 = RandomForestRegressor(n_estimators=100)
model.fit(X_train, y_train)

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

Step 3: Creating API to Serve Predictions

We create a Flask API to serve predictions to the load balancer.

python

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

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

app = Flask(__name__)

@app.route('/predict', methods=['POST'])
def predict():
    data = request.get_json()
    features = np.array([[data['time'], data['requests_per_minute'],
                          data['cpu_usage'], data['memory_usage'], data['response_time']]])

    prediction = model.predict(features)[0]
```

```python
    return jsonify({"server_allocation": int(prediction)})

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

Run the API:

```
python3 api.py
```

Step 4: Configuring Nginx as a Load Balancer

Modify **nginx.conf** to use the AI-powered decision-making API.

nginx

```nginx
http {
    upstream backend_servers {
        server server1.example.com;
        server server2.example.com;
        server server3.example.com;
    }

    server {
        listen 80;
        location / {
            proxy_pass http://backend_servers;
        }

        location /predict {
            proxy_pass http://127.0.0.1:5000;
        }
    }
}
```

Restart Nginx:

```
sudo systemctl restart nginx
```

Step 5: Automating with Docker & Kubernetes

5.1: Create Dockerfile

dockerfile

```
FROM python:3.9
WORKDIR /app
COPY . /app
RUN pip install -r requirements.txt
CMD ["python3", "api.py"]
```

Build and Run Container:

```
docker build -t predictive-load-balancer .
docker run -d -p 5000:5000 predictive-load-balancer
```

5.2: Deploy with Kubernetes

Create deployment.yaml:

yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: predictive-load-balancer
spec:
```

```yaml
  replicas: 2
  selector:
    matchLabels:
      app: load-balancer
  template:
    metadata:
      labels:
        app: load-balancer
    spec:
      containers:
      - name: load-balancer
        image: predictive-load-balancer
        ports:
        - containerPort: 5000
```

Apply Deployment:

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

Step 6: Monitoring with Prometheus & Grafana

6.1: Install Prometheus

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

6.2: Configure Prometheus for API Metrics

Modify prometheus.yml:

yaml

```yaml
scrape_configs:
```

```
  - job_name: 'load-balancer-api'
    metrics_path: '/metrics'
    static_configs:
      - targets: ['localhost:5000']
```

Restart Prometheus:

```
sudo systemctl restart prometheus
```

6.3: Install Grafana

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

Login to Grafana (http://localhost:3000), add Prometheus as a data source, and create dashboards.

Conclusion

This project demonstrates how **AI-driven predictive load balancing** optimizes resource allocation by anticipating traffic surges. It integrates:

- **Machine Learning for Traffic Prediction**
- **Flask API for Predictions**
- **Nginx Load Balancer**
- **Docker & Kubernetes for Deployment**
- **Prometheus & Grafana for Monitoring**

Project 2. AI-Driven Predictive Resource Allocation: Using AI to dynamically allocate resources (CPU, memory, storage) based on predicted workloads in containers and VMs.

This project focuses on **AI-Driven Predictive Resource Allocation**, where AI models analyze past workloads and predict future resource demands. Based on predictions, the system dynamically adjusts **CPU, memory, and storage** allocation for **containers and VMs** to optimize performance and cost efficiency.

Step-by-Step Guide

1. Prerequisites

- Ubuntu 20.04+ (or any Linux-based OS)
- Docker & Kubernetes (for containerized environments)
- Python 3.8+ (for AI model development)
- TensorFlow/PyTorch (for predictive modeling)
- Prometheus & Grafana (for monitoring)
- Kubernetes Horizontal Pod Autoscaler (HPA) & Vertical Pod Autoscaler (VPA)
- Terraform (for infrastructure automation)
- Ansible (for automation)
- Jupyter Notebook (for model development)

---

2. Project Setup

Step 1: Install Required Tools

```
# Update packages
sudo apt update && sudo apt upgrade -y

# Install Docker
sudo apt install docker.io -y
sudo systemctl start docker
sudo systemctl enable docker
```

 # Install Kubernetes (kind for local setup)
curl -Lo ./kind https://kind.sigs.k8s.io/dl/v0.20.0/kind-linux-amd64
chmod +x kind
sudo mv kind /usr/local/bin/

 # Install kubectl
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 Prometheus & Grafana
kubectl apply -f
https://raw.githubusercontent.com/prometheus-operator/prometheus-operator/main/bundle.yaml
kubectl apply -f
https://raw.githubusercontent.com/grafana/grafana/main/deploy/kubernetes/grafana.yaml

 # Install Terraform
wget
https://releases.hashicorp.com/terraform/1.5.0/terraform_1.5.0_linux_amd64.zip
unzip terraform_1.5.0_linux_amd64.zip
sudo mv terraform /usr/local/bin/

---

3. AI Model Development (Predicting Resource Usage)

Step 2: Install Python Libraries

pip install numpy pandas tensorflow torch matplotlib seaborn scikit-learn

Step 3: Load & Preprocess Data

python

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
import tensorflow as tf
from tensorflow import keras

# Load dataset (Assuming CSV format with 'CPU', 'Memory', 'Storage', 'Timestamp')
data = pd.read_csv("resource_usage.csv")

# Convert timestamp to numerical values
data['Timestamp'] = pd.to_datetime(data['Timestamp'])
data['Timestamp'] = data['Timestamp'].astype(int) // 10**9  # Convert to Unix time

# Normalize data
scaler = MinMaxScaler()
data_scaled = scaler.fit_transform(data)

# Split dataset
X = data_scaled[:, :-1]
y = data_scaled[:, -1]
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Build AI Model
model = keras.Sequential([
    keras.layers.Dense(64, activation='relu', input_shape=(X_train.shape[1],)),
    keras.layers.Dense(32, activation='relu'),
    keras.layers.Dense(1)  # Predict next resource allocation
])
```

```
model.compile(optimizer='adam', loss='mse')
model.fit(X_train, y_train, epochs=50, batch_size=16, validation_data=(X_test,
y_test))

# Save model
model.save("resource_predictor.h5")
```

---

4. Deploy AI Model in Kubernetes

Step 4: Create a Flask API for AI Model

python

```
from flask import Flask, request, jsonify
import tensorflow as tf
import numpy as np

app = Flask(__name__)

# Load trained model
model = tf.keras.models.load_model("resource_predictor.h5")

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

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

Step 5: Create Dockerfile for Deployment

dockerfile

```
FROM python:3.8-slim

WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt

COPY app.py .
COPY resource_predictor.h5 .

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

Step 6: Build and Push Docker Image

```
docker build -t myrepo/resource-predictor:latest .
docker push myrepo/resource-predictor:latest
```

Step 7: Deploy to Kubernetes

yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ai-resource-predictor
spec:
  replicas: 1
  selector:
    matchLabels:
      app: ai-resource-predictor
  template:
    metadata:
      labels:
        app: ai-resource-predictor
```

```yaml
    spec:
      containers:
      - name: ai-resource-predictor
        image: myrepo/resource-predictor:latest
        ports:
        - containerPort: 5000
```

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

---

5. Implement Auto-Scaling Based on Predictions

Step 8: Enable Kubernetes HPA

```
kubectl autoscale deployment ai-resource-predictor --cpu-percent=50 --min=1
--max=5
```

Step 9: Enable Kubernetes VPA

yaml

```yaml
apiVersion: autoscaling.k8s.io/v1
kind: VerticalPodAutoscaler
metadata:
  name: ai-resource-predictor-vpa
spec:
  targetRef:
    apiVersion: "apps/v1"
    kind: Deployment
    name: ai-resource-predictor
  updatePolicy:
    updateMode: "Auto"
```

```
kubectl apply -f vpa.yaml
```

---

6. Monitor Resource Allocation

Step 10: Setup Prometheus & Grafana Dashboards

```
kubectl port-forward svc/prometheus 9090
kubectl port-forward svc/grafana 3000
```

- Open Grafana at http://localhost:3000
- Add Prometheus as a data source
- Create a dashboard with metrics:
    - container_memory_usage_bytes
    - container_cpu_usage_seconds_total
    - container_fs_usage_bytes

---

7. Automate Infrastructure with Terraform

Step 11: Create Terraform Script

hcl

```hcl
provider "aws" {
  region = "us-east-1"
}

resource "aws_instance" "k8s_node" {
  ami           = "ami-0abcdef1234567890"
  instance_type = "t3.medium"

  tags = {
```

```
    Name = "KubernetesNode"
  }
}
```

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

Conclusion

This project **predicts future resource usage** and **automatically scales Kubernetes workloads** using AI. It improves **efficiency, cost optimization, and performance** for dynamic cloud environments.

Project 3. Predictive Autoscaling with Customizable Metrics: AI-based auto-scaling system that considers custom application-specific metrics in addition to CPU/memory load.

Autoscaling is essential in cloud environments to manage application performance and cost efficiently. Traditional autoscaling methods rely on CPU and memory utilization, but predictive autoscaling enhances this by using AI-based models to forecast future resource demands.

This project implements a **Predictive Autoscaling System** that uses machine learning models to scale resources based on both system (CPU/Memory) and custom application-specific metrics, such as request rates, latency, or database queries per second.

Project Overview

- **Step 1**: Setup Kubernetes cluster (K3s/Kind/Minikube)

- **Step 2**: Install and configure Prometheus for monitoring metrics
- **Step 3**: Train and deploy a Machine Learning model for prediction
- **Step 4**: Implement a custom Kubernetes autoscaler using Python
- **Step 5**: Deploy a sample application and test autoscaling

---

Step 1: Setup Kubernetes Cluster

Using Kind (Kubernetes in Docker)

kind create cluster --name predictive-autoscale
kubectl cluster-info --context kind-predictive-autoscale

Step 2: Install Prometheus for Metrics Collection

Deploy Prometheus using Helm

helm repo add prometheus-community
https://prometheus-community.github.io/helm-charts
helm repo update
helm install prometheus prometheus-community/kube-prometheus-stack
--namespace monitoring --create-namespace

Verify Installation

kubectl get pods -n monitoring

Step 3: Train and Deploy a Machine Learning Model

We use a simple **Linear Regression Model** trained with past CPU usage and request rates to predict future resource needs.

Python Code for Training (train_model.py)

python

```
import numpy as np
import pandas as pd
from sklearn.linear_model import LinearRegression
import pickle

# Sample Data: CPU Usage & Requests
data = {
    "cpu_usage": [20, 30, 50, 60, 80],
    "request_rate": [100, 200, 400, 600, 900],
    "replicas": [1, 2, 3, 4, 5]  # Expected scaling
}

df = pd.DataFrame(data)

# Train Model
X = df[["cpu_usage", "request_rate"]]
y = df["replicas"]

model = LinearRegression()
model.fit(X, y)

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

Deploy Model as a Microservice

Create a Flask API to serve predictions.

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

autoscaler_service.py

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

app = Flask(__name__)

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

@app.route("/predict", methods=["POST"])
def predict():
    data = request.get_json()
    cpu_usage = data["cpu_usage"]
    request_rate = data["request_rate"]

    prediction = model.predict(np.array([[cpu_usage, request_rate]]))
    return jsonify({"recommended_replicas": int(round(prediction[0]))})

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

Run API

```
python autoscaler_service.py
```

Test the API:

```
curl -X POST http://localhost:5000/predict -H "Content-Type: application/json" -d
'{"cpu_usage": 60, "request_rate": 700}'
```

Step 4: Implement Custom Kubernetes Autoscaler

We create a **Python script that fetches Prometheus metrics and scales deployments**.

autoscaler.py

python

```
import requests
import json
import subprocess

PROMETHEUS_URL =
"http://prometheus-server.monitoring.svc.cluster.local:9090/api/v1/query"
PREDICTOR_URL =
"http://autoscaler-service.default.svc.cluster.local:5000/predict"
DEPLOYMENT_NAME = "my-app"
NAMESPACE = "default"

def get_metrics():
    cpu_query =
'sum(rate(container_cpu_usage_seconds_total{namespace="default"}[5m]))'
    request_query = 'sum(rate(http_requests_total{namespace="default"}[5m]))'

    cpu_response =
requests.get(f"{PROMETHEUS_URL}?query={cpu_query}").json()
    request_response =
requests.get(f"{PROMETHEUS_URL}?query={request_query}").json()

    cpu_usage = float(cpu_response["data"]["result"][0]["value"][1])
    request_rate = float(request_response["data"]["result"][0]["value"][1])

    return cpu_usage, request_rate

def scale_deployment(replicas):
```

```
    cmd = f"kubectl scale deployment {DEPLOYMENT_NAME}
--replicas={replicas}"
    subprocess.run(cmd, shell=True)

def main():
    cpu_usage, request_rate = get_metrics()

    payload = {"cpu_usage": cpu_usage, "request_rate": request_rate}
    prediction_response = requests.post(PREDICTOR_URL, json=payload).json()
    recommended_replicas = prediction_response["recommended_replicas"]

    scale_deployment(recommended_replicas)

if __name__ == "__main__":
    main()
```

Run Autoscaler in a Cron Job

Create a Kubernetes CronJob to run every minute.

yaml

```yaml
apiVersion: batch/v1
kind: CronJob
metadata:
  name: predictive-autoscaler
spec:
  schedule: "* * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: autoscaler
            image: myrepo/autoscaler:latest
```

```
    command: ["python", "autoscaler.py"]
  restartPolicy: OnFailure
```

Step 5: Deploy a Sample Application

```
kubectl create deployment my-app --image=nginx
kubectl expose deployment my-app --type=LoadBalancer --port=80
```

Step 6: Test Predictive Autoscaling

Increase traffic:

```
kubectl run load-test --image=busybox --restart=Never -- wget -qO- http://my-app
```

Check replicas:

```
kubectl get deployment my-app
```

Summary

- We **set up Kubernetes** and installed **Prometheus** to collect metrics.
- We **trained a predictive ML model** to estimate the required replicas.
- We **built a Flask API** to serve predictions.
- We **created a Python-based Kubernetes autoscaler** that dynamically scales deployments.
- We **automated the scaling process** with a Kubernetes **CronJob**.

Project 4. AI-Powered Resource Bottleneck Detection: AI to analyze performance metrics and detect resource bottlenecks that may affect scaling decisions.

Scaling applications efficiently requires understanding resource usage. This project uses **AI/ML techniques** to analyze system performance metrics (CPU, memory, network, and disk usage) and detect **resource bottlenecks** that may impact scaling decisions. We will use **Python, Prometheus, Grafana, and Scikit-Learn** for data collection, visualization, and AI-based anomaly detection.

Project Setup & Steps

1. Install Required Tools

Ensure your system has the following installed:

- **Python** (v3.8+)
- **Prometheus** (for monitoring)
- **Grafana** (for visualization)
- **Docker** (optional for containerization)

Install required Python packages:

pip install pandas numpy scikit-learn prometheus_api_client flask

---

2. Set Up Prometheus for Data Collection

Create a Prometheus configuration file prometheus.yml:

yaml

```
global:
  scrape_interval: 5s
```

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

Run Prometheus using Docker:

```
docker run -p 9090:9090 -v
$(pwd)/prometheus.yml:/etc/prometheus/prometheus.yml prom/prometheus
```

---

3. Fetch Performance Metrics

Use Python to query Prometheus and retrieve system metrics.

Create a file fetch_metrics.py:

python

```
from prometheus_api_client import PrometheusConnect
import pandas as pd
import time

# Connect to Prometheus
prom = PrometheusConnect(url="http://localhost:9090", disable_ssl=True)

def fetch_metrics():
    query_cpu = '100 - (avg by (instance)
(irate(node_cpu_seconds_total{mode="idle"}[5m])) * 100)'
    query_memory = 'node_memory_Active_bytes /
node_memory_MemTotal_bytes * 100'

    cpu_usage = prom.custom_query(query=query_cpu)
    memory_usage = prom.custom_query(query=query_memory)
```

```
    return cpu_usage, memory_usage

if __name__ == "__main__":
    while True:
        cpu, mem = fetch_metrics()
        print("CPU Usage:", cpu)
        print("Memory Usage:", mem)
        time.sleep(10)
```

Run the script:

```
python fetch_metrics.py
```

---

4. Implement AI Model for Bottleneck Detection

Modify bottleneck_detector.py:

python

```
import numpy as np
import pandas as pd
from sklearn.ensemble import IsolationForest

# Simulated sample data
data = {
    "cpu": [20, 30, 50, 90, 95, 15, 40, 80, 85, 10],
    "memory": [40, 50, 75, 85, 90, 35, 60, 80, 95, 20]
}

df = pd.DataFrame(data)

# Train Isolation Forest for anomaly detection
model = IsolationForest(contamination=0.2)
```

```python
df["anomaly"] = model.fit_predict(df[["cpu", "memory"]])

# Print detected anomalies
print(df[df["anomaly"] == -1])
```

Run:

```
python bottleneck_detector.py
```

---

5. Build a Flask API for Live Bottleneck Detection

Create app.py:

```python
from flask import Flask, jsonify
from bottleneck_detector import model, df

app = Flask(__name__)

@app.route("/detect", methods=["GET"])
def detect():
    anomalies = df[df["anomaly"] == -1].to_dict(orient="records")
    return jsonify({"bottlenecks": anomalies})

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

Run Flask API:

```
python app.py
```

Test with:

curl http://127.0.0.1:5000/detect

---

6. Visualize in Grafana

- Connect Grafana to Prometheus
- Create dashboards to monitor CPU and Memory usage

Conclusion

This project uses **Prometheus for monitoring, Flask for API, and AI (Isolation Forest) to detect bottlenecks** in real-time. The insights help in **scaling decisions**, ensuring **efficient resource utilization**.

Project 5. Multi-Tenant Cloud Optimization: Using AI to ensure efficient resource sharing in multi-tenant cloud environments without compromising performance.

Multi-tenant cloud environments host multiple users (tenants) on a shared infrastructure, making efficient resource allocation crucial. AI-driven optimization ensures fair resource distribution, cost savings, and performance stability without compromising security.

This project will leverage **Python, Kubernetes, Prometheus, Grafana, and Machine Learning (ML)** to build an AI-based resource allocation system.

Project Steps with Commands

Step 1: Set Up the Environment

Ensure you have the necessary tools installed:

- Python 3.x
- Kubernetes (kind or Minikube)
- Docker
- Helm
- Prometheus & Grafana

 # Install Python dependencies
pip install numpy pandas scikit-learn flask requests kubernetes prometheus_client

 # Install Kubernetes cluster (if not already)
kind create cluster --name multi-tenant

 # Install Prometheus & Grafana for monitoring
helm repo add prometheus-community
https://prometheus-community.github.io/helm-charts
helm repo update
helm install prometheus prometheus-community/kube-prometheus-stack

---

Step 2: Create a Kubernetes Multi-Tenant Setup

Create Namespaces for Tenants
kubectl create namespace tenant-a
kubectl create namespace tenant-b

Define Resource Quotas for Each Tenant

Save this as quota.yaml:

yaml

```yaml
apiVersion: v1
kind: ResourceQuota
metadata:
  name: tenant-quota
  namespace: tenant-a
spec:
  hard:
    cpu: "2"
    memory: "4Gi"
    pods: "10"
```

Apply it:

```
kubectl apply -f quota.yaml
```

Step 3: Deploy Sample Workloads

Create a simple web app (Flask) and deploy it in Kubernetes.

Flask App (app.py)

python

```python
from flask import Flask
import os

app = Flask(__name__)

@app.route("/")
def home():
    return f"Running in {os.environ.get('TENANT', 'default')} namespace"

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

Dockerize the App

```
# Dockerfile
FROM python:3.9
WORKDIR /app
COPY app.py .
RUN pip install flask
CMD ["python", "app.py"]
```

```
docker build -t multi-tenant-app .
docker tag multi-tenant-app myrepo/multi-tenant-app:latest
docker push myrepo/multi-tenant-app:latest
```

Deploy in Kubernetes

yaml

```yaml
# deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: tenant-app
  namespace: tenant-a
spec:
  replicas: 2
  selector:
    matchLabels:
      app: tenant-app
  template:
    metadata:
      labels:
```

```yaml
      app: tenant-app
  spec:
    containers:
    - name: tenant-app
      image: myrepo/multi-tenant-app:latest
      ports:
      - containerPort: 5000
```

Apply it:

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

Step 4: AI-Based Optimization Model

Create an AI model to predict and optimize resource allocation.

AI Model (optimize.py)

python

```python
import numpy as np
from sklearn.linear_model import LinearRegression

# Sample data (CPU usage vs. requests)
X = np.array([10, 20, 30, 40, 50]).reshape(-1, 1)  # Requests
y = np.array([1, 2, 2.5, 3, 4])  # CPU usage in cores

model = LinearRegression()
model.fit(X, y)

def predict_cpu(requests):
    return model.predict(np.array([[requests]]))[0]

# Example prediction
```

```
print(f"Predicted CPU for 60 requests: {predict_cpu(60)} cores")
```

Step 5: Monitor and Optimize in Real-Time

Expose Prometheus Metrics

python

```python
# metrics.py
from prometheus_client import start_http_server, Gauge
import random
import time

cpu_usage = Gauge("cpu_usage", "Current CPU usage")

def monitor():
    start_http_server(8000)
    while True:
        cpu_usage.set(random.uniform(1, 4))  # Simulating CPU usage
        time.sleep(5)

monitor()
```

View Metrics in Prometheus

```
kubectl port-forward svc/prometheus 9090
```

Access: **http://localhost:9090**

Visualize in Grafana

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

Access: **http://localhost:3000** (Default Login: admin/admin)

Conclusion

This project sets up an AI-driven multi-tenant cloud resource optimization system.

- **AI predicts CPU needs**
- **Prometheus monitors usage**
- **Kubernetes enforces quotas**
- **Grafana visualizes performance**

---

9. AI for Incident Prediction & Automated Remediation

Project 1. Automated Health Checks with AI: AI-powered health check system that automatically checks infrastructure health and suggests fixes before failure.

Automated Health Checks with AI is a system that monitors infrastructure (servers, databases, applications) using AI. It detects issues like high CPU usage, low memory, or failing services and suggests or applies fixes automatically.

Technologies Used:

- Python (Flask for API, TensorFlow for AI model)
- Prometheus (Monitoring)
- Grafana (Visualization)
- Docker (Containerization)
- Kubernetes (Orchestration)
- Jenkins (CI/CD)

Step-by-Step Implementation

1. Install Dependencies

Ensure Python, Docker, and Kubernetes are installed.

```
sudo apt update && sudo apt install -y python3 python3-pip docker.io kubectl
pip3 install flask prometheus_client tensorflow numpy pandas
```

---

2. Set Up Prometheus for Monitoring

Create a prometheus.yml config file:

yaml

```yaml
global:
  scrape_interval: 15s

scrape_configs:
  - job_name: 'health-checks'
    static_configs:
      - targets: ['localhost:8000']
```

Run Prometheus in Docker:

```
docker run -d --name=prometheus -p 9090:9090 -v
$(pwd)/prometheus.yml:/etc/prometheus/prometheus.yml prom/prometheus
```

---

3. Create a Flask API for Health Checks

python

```python
from flask import Flask, jsonify
import psutil
import tensorflow as tf
import numpy as np

app = Flask(__name__)
```

```python
# AI Model (Dummy Model for Prediction)
model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(10, activation='relu', input_shape=(3,)),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

@app.route('/health', methods=['GET'])
def check_health():
    cpu = psutil.cpu_percent(interval=1)
    memory = psutil.virtual_memory().percent
    disk = psutil.disk_usage('/').percent

    prediction = model.predict(np.array([[cpu, memory, disk]]))
    health_status = "Critical" if prediction[0][0] > 0.5 else "Healthy"

    return jsonify({'cpu': cpu, 'memory': memory, 'disk': disk, 'status': health_status})

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=8000)
```

Run the API:

```
python3 health_check.py
```

---

4. Set Up Grafana for Visualization

Run Grafana:

```
docker run -d --name=grafana -p 3000:3000 grafana/grafana
```

Log in to http://localhost:3000 and configure Prometheus as a data source.

---

5. Deploy in Kubernetes

Create a deployment file health-check-deployment.yaml:

yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: health-check
spec:
  replicas: 2
  selector:
    matchLabels:
      app: health-check
  template:
    metadata:
      labels:
        app: health-check
    spec:
      containers:
        - name: health-check
          image: your-dockerhub-username/health-check:latest
          ports:
            - containerPort: 8000
```

Apply it:

kubectl apply -f health-check-deployment.yaml

---

6. Automate with Jenkins

Create a Jenkinsfile:

groovy

```groovy
pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                sh 'docker build -t your-dockerhub-username/health-check .'
            }
        }
        stage('Push') {
            steps {
                withDockerRegistry([credentialsId: 'docker-hub', url: '']) {
                    sh 'docker push your-dockerhub-username/health-check'
                }
            }
        }
        stage('Deploy') {
            steps {
                sh 'kubectl apply -f health-check-deployment.yaml'
            }
        }
    }
}
```

Run Jenkins Pipeline.

- **Flask API**: Hosts a simple server that checks CPU, memory, and disk usage.
- **AI Model**: Uses TensorFlow to analyze the system's health and predict failures.
- **Prometheus**: Collects real-time system metrics.
- **Grafana**: Visualizes data from Prometheus.

- **Kubernetes**: Deploys and scales the application.
- **Jenkins**: Automates build and deployment.

Project 2. Dynamic Incident Severity Prediction: AI model that predicts the potential severity of an incident based on past data, helping teams prioritize responses.

Incident management is crucial in IT operations, cybersecurity, and customer support. A quick response to critical incidents can prevent business losses. This project develops a **Machine Learning (ML) model** to predict incident severity using historical data, helping teams prioritize responses efficiently.

Technologies Used

- **Python** (for data processing and model training)
- **Pandas, NumPy** (for data handling)
- **Scikit-learn** (for machine learning)
- **Flask** (to create an API for predictions)
- **Docker** (for containerization)
- **Jupyter Notebook** (for experimentation)

---

2. Steps to Build the Project

Step 1: Set Up the Environment

Install required libraries:

pip install pandas numpy scikit-learn flask joblib

Step 2: Prepare Dataset

For simplicity, we use a CSV dataset with fields like:

- **incident_type** (e.g., network failure, security breach)
- **time_of_day** (morning, afternoon, night)
- **affected_users** (number of users impacted)
- **downtime_minutes** (how long the issue lasted)
- **severity** (Low, Medium, High)

Example Dataset (incident_data.csv):

incident_type	time_of_day	affected_users	downtime_minutes	severity
network_issue	morning	100	30	Medium
security_breach	night	500	120	High
hardware_fail	afternoon	50	20	Low

Step 3: Load & Process Data

Create a script (data_processing.py) to preprocess data.

python

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

# Load data
df = pd.read_csv("incident_data.csv")

# Encode categorical values
encoder = LabelEncoder()
df["incident_type"] = encoder.fit_transform(df["incident_type"])
```

df["time_of_day"] = encoder.fit_transform(df["time_of_day"])
df["severity"] = encoder.fit_transform(df["severity"]) # Convert labels to numbers

 # Split data
X = df.drop(columns=["severity"])
y = df["severity"]
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

print("Data processed successfully!")

Step 4: Train the ML Model

Create a script (train_model.py) to train a classification model.

python

from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score
import joblib

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

 # Save model
joblib.dump(clf, "incident_model.pkl")

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

Step 5: Build API for Prediction

Create a Flask API (app.py) to take input and predict severity.

python

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

app = Flask(__name__)

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

@app.route("/predict", methods=["POST"])
def predict():
    data = request.get_json()
    df = pd.DataFrame([data])
    prediction = model.predict(df)
    severity_map = {0: "Low", 1: "Medium", 2: "High"}
    return jsonify({"severity_prediction": severity_map[prediction[0]]})

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

Step 6: Test the API

Run the Flask app:

```
python app.py
```

Then, send a test request using **Postman** or **cURL**:

```
curl -X POST http://127.0.0.1:5000/predict -H "Content-Type: application/json" -d
'{"incident_type": 1, "time_of_day": 2, "affected_users": 200,
"downtime_minutes": 45}'
```

Expected Response:

json

```json
{"severity_prediction": "Medium"}
```

Step 7: Containerize the Application

Create a Dockerfile for the API:

dockerfile

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

Build & Run the Docker container:

```
docker build -t incident-severity .
docker run -p 5000:5000 incident-severity
```

- **Data Preprocessing:** Converts raw data into a usable format.
- **Label Encoding:** Transforms categorical data (e.g., "morning") into numbers.
- **Model Training:** Uses past incidents to learn patterns.

- **Flask API:** Exposes a web service to take new incidents as input and predict severity.
- **Docker:** Ensures the project runs the same way everywhere.

Project 3. Proactive Failure Prevention System: AI-based system that uses failure trends to predict and prevent critical infrastructure failures before they happen.

Introduction

In critical infrastructure systems like manufacturing plants, cloud servers, or railway tracks, failures can cause significant downtime and financial loss. A **Proactive Failure Prevention System** leverages **machine learning** to predict failures before they happen. The system analyzes past failure data, identifies trends, and alerts users about potential failures so preventive actions can be taken.

Project Breakdown

1. **Set up the environment** (Python, dependencies, database)
2. **Collect and store sensor data** (Simulated dataset)
3. **Train a Machine Learning model** (Failure prediction using Scikit-Learn)
4. **Build an API using Flask** (Serve ML predictions)
5. **Store predictions in MongoDB** (Historical tracking)
6. **Deploy using Docker** (Containerize and run anywhere)

Step-by-Step Implementation

1. Set Up the Environment

Install the required dependencies:

```
# Update the system and install dependencies
sudo apt update && sudo apt install python3-pip -y

# Create and activate a virtual environment
python3 -m venv venv
source venv/bin/activate

# Install necessary Python libraries
pip install flask pandas scikit-learn pymongo numpy joblib
```

---

2. Prepare the Sensor Data

We'll create a **simulated dataset** representing sensor readings and failure records.

sensor_data.csv (Example dataset)

```
temperature,pressure,vibration,failure
80,100,0.5,1
60,85,0.3,0
75,95,0.4,1
50,70,0.2,0
```

---

3. Train a Machine Learning Model

We'll use a **Random Forest Classifier** to predict failures based on sensor data.

train_model.py

python

```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 dataset
df = pd.read_csv("sensor_data.csv")

# Features and target variable
X = df[['temperature', 'pressure', 'vibration']]
y = df['failure']

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

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

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

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

Run the script:

```
python train_model.py
```

---

4. Create a Flask API

Flask will serve predictions via an API.

app.py

python

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

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

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

app = Flask(__name__)

@app.route('/predict', methods=['POST'])
def predict():
    data = request.json
    temperature = data["temperature"]
    pressure = data["pressure"]
    vibration = data["vibration"]

    # Make prediction
    features = np.array([[temperature, pressure, vibration]])
    prediction = model.predict(features)[0]

    # Store in MongoDB
    collection.insert_one({"temperature": temperature, "pressure": pressure,
"vibration": vibration, "prediction": int(prediction)})

    return jsonify({"failure": bool(prediction)})
```

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

---

5. Run MongoDB

Start MongoDB to store predictions.

```
sudo systemctl start mongod
mongo --eval 'use failure_db'
```

---

6. Test the API

Run the Flask app:

```
python app.py
```

Send a test request:

```
curl -X POST "http://127.0.0.1:5000/predict" -H "Content-Type: application/json"
-d '{"temperature": 75, "pressure": 90, "vibration": 0.4}'
```

Expected output:

json

```
{"failure": true}
```

---

7. Deploy Using Docker

Create a **Dockerfile**:

dockerfile

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

Build and run the container:

```
docker build -t failure-predictor .
docker run -p 5000:5000 failure-predictor
```

- **Machine Learning (ML) Model:** We trained a model to predict failures using historical data.
- **Flask API:** The API accepts real-time sensor data and predicts failure risks.
- **MongoDB:** Stores historical predictions to analyze failure trends.
- **Docker:** Enables the application to run in any environment.

Conclusion

This project showcases how **AI-driven predictive maintenance** can prevent failures. By continuously improving the ML model and integrating real-time IoT sensor data, this system can be scaled for **smart manufacturing, cloud reliability, and critical infrastructure monitoring**.

Project 4. Predictive Incident Management in Multi-Cloud: AI to predict incidents across different cloud environments and suggest remediation actions.

Cloud environments generate vast amounts of logs and monitoring data. This project builds an **AI-powered system** that **predicts incidents** across AWS, Azure, and GCP and suggests remediation actions.

Technologies Used

- **Machine Learning (ML):** Python, Scikit-learn, Pandas
- **Cloud APIs:** AWS CloudWatch, Azure Monitor, GCP Logging
- **Infrastructure:** Docker, Kubernetes, Terraform
- **Monitoring:** Prometheus, Grafana
- **DevOps Tools:** Jenkins, GitHub Actions

---

2. Project Setup

Install Required Tools

```
sudo apt update && sudo apt install python3-pip -y
pip install pandas numpy scikit-learn flask requests boto3 google-cloud-monitoring azure-mgmt-monitor joblib
```

---

3. Collecting Incident Data

AWS CloudWatch Logs

python

```python
import boto3

client = boto3.client('logs')

def get_logs(log_group, start_time, end_time):
    response = client.filter_log_events(
        logGroupName=log_group,
        startTime=start_time,
```

```
        endTime=end_time
    )
    return response['events']

logs = get_logs('/aws/lambda/error-logs', 1700000000, 1700003600)
print(logs)
```

Azure Monitor Logs

python

```
from azure.mgmt.monitor import MonitorManagementClient
from azure.identity import DefaultAzureCredential

credential = DefaultAzureCredential()
client = MonitorManagementClient(credential, "<Subscription_ID>")

def get_logs():
    logs =
client.metrics.list("subscriptions/<Subscription_ID>/resourceGroups/<ResourceGr
oup>/providers/Microsoft.Compute/virtualMachines/<VM_Name>")
    return logs

print(get_logs())
```

---

4. Machine Learning Model

Preprocessing Data

python

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

```
import joblib

# Load dataset
data = pd.read_csv("incident_logs.csv")

# Feature selection
X = data[['cpu_usage', 'memory_usage', 'response_time']]
y = data['incident_occurred']

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

# Train Model
model = RandomForestClassifier()
model.fit(X_train, y_train)

# Save model
joblib.dump(model, 'incident_predictor.pkl')
```

---

5. API for Predictions

Flask API

python

```
from flask import Flask, request, jsonify
import joblib

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

@app.route('/predict', methods=['POST'])
def predict():
    data = request.get_json()
```

```
    prediction = model.predict([[data['cpu_usage'], data['memory_usage'],
data['response_time']]])
    return jsonify({'incident_predicted': bool(prediction[0])})

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

Test API

```
curl -X POST http://localhost:5000/predict -H "Content-Type: application/json" -d
'{"cpu_usage": 85, "memory_usage": 70, "response_time": 500}'
```

---

6. Docker & Kubernetes Deployment

Dockerfile

dockerfile

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

Build and Push Docker Image

```
docker build -t your_dockerhub/incident-predictor:latest .
docker push your_dockerhub/incident-predictor:latest
```

Kubernetes Deployment

yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: incident-predictor
spec:
  replicas: 2
  selector:
    matchLabels:
      app: incident-predictor
  template:
    metadata:
      labels:
        app: incident-predictor
    spec:
      containers:
      - name: predictor
        image: your_dockerhub/incident-predictor:latest
        ports:
        - containerPort: 5000
---
apiVersion: v1
kind: Service
metadata:
  name: incident-predictor
spec:
  type: LoadBalancer
  ports:
  - port: 80
    targetPort: 5000
  selector:
    app: incident-predictor
```

Deploy to Kubernetes

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

---

7. Monitoring with Prometheus & Grafana

Prometheus Config

yaml

```yaml
scrape_configs:
  - job_name: 'incident-predictor'
    metrics_path: /metrics
    static_configs:
      - targets: ['incident-predictor:5000']
```

Start Prometheus

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

Start Grafana

```
docker run -d -p 3000:3000 grafana/grafana
```

---

8. Auto-Remediation with AWS Lambda

- If an incident is predicted, **AWS Lambda** triggers an action.

AWS Lambda Code

python

```
import boto3

def lambda_handler(event, context):
    client = boto3.client('ec2')
    instances = ['i-0abcd1234efgh5678']
    response = client.reboot_instances(InstanceIds=instances)
    return response
```

Trigger Lambda from API

Modify **Flask API** to trigger **AWS Lambda** if an incident is predicted:

python

```
import boto3

lambda_client = boto3.client('lambda')

def trigger_lambda():
    response = lambda_client.invoke(FunctionName="AutoRemediationLambda")
    return response
```

---

9. CI/CD with Jenkins

Jenkins Pipeline

groovy

```
pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                sh 'docker build -t your_dockerhub/incident-predictor:latest .'
            }
        }
        stage('Push') {
            steps {
                withDockerRegistry([credentialsId: 'docker-hub-credentials', url: '']) {
                    sh 'docker push your_dockerhub/incident-predictor:latest'
                }
            }
        }
        stage('Deploy') {
            steps {
                sh 'kubectl apply -f deployment.yaml'
            }
        }
    }
}
```

---

10. Conclusion

- **Predict incidents using AI.**
- **Deploy in Kubernetes** for scalability.
- **Monitor with Prometheus & Grafana.**
- **Automate remediation using AWS Lambda.**
- **CI/CD with Jenkins.**

Project 5. AI-Powered Predictive Alerting: Using machine learning models to identify patterns that precede incidents and proactively alert teams before failure occurs.

In modern IT operations, system failures can lead to downtime, loss of revenue, and customer dissatisfaction. This project focuses on **AI-powered predictive alerting**, where we use **machine learning models** to analyze system logs and metrics, identify patterns leading to failures, and **proactively alert** teams before incidents occur.

This project is useful for **DevOps engineers, SREs, and IT teams** to implement predictive monitoring instead of reactive troubleshooting.

Tech Stack

- **Programming Language**: Python
- **Machine Learning**: Scikit-learn, Pandas, NumPy
- **Data Visualization**: Matplotlib, Seaborn
- **Alerting**: Prometheus & Alertmanager
- **Deployment**: Docker, Kubernetes
- **Data Storage**: PostgreSQL or InfluxDB
- **Logging & Monitoring**: Grafana, Prometheus

Project Steps

Step 1: Setup Environment

Install the necessary dependencies:

pip install pandas numpy scikit-learn matplotlib seaborn prometheus-client flask requests

---

Step 2: Collect & Preprocess Data

We'll use system logs or synthetic failure logs.

Example dataset structure (CSV)

Timestamp	CPU Usage (%)	Memory Usage (%)	Disk I/O (MB/s)	Error Count	Failure (1/0)
10:01:00	85	76	120	5	0
10:02:00	90	80	130	10	1

Load dataset in Python

python

```
import pandas as pd
df = pd.read_csv("system_logs.csv")
print(df.head())
```

Preprocessing

python

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

X = df.drop(columns=["Failure"])
y = df["Failure"]

scaler = StandardScaler()
```

```
X_scaled = scaler.fit_transform(X)

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

Step 3: Train Machine Learning Model

We'll use a **Random Forest classifier** to predict failures.

python

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score

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

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

Step 4: Deploy Model as an API

We'll use **Flask** to create an API for real-time predictions.

app.py
python

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

app = Flask(__name__)
```

```
model = joblib.load("predictor.pkl")
scaler = joblib.load("scaler.pkl")

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

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

Save the model

python

```
import joblib
joblib.dump(model, "predictor.pkl")
joblib.dump(scaler, "scaler.pkl")
```

Step 5: Alerting with Prometheus & Alertmanager

Expose metrics for monitoring

Modify app.py:

python

```
from prometheus_client import Counter, start_http_server

failure_alerts = Counter('system_failure_alerts', 'Number of predicted failures')

@app.route('/predict', methods=['POST'])
```

```
def predict():
    data = request.json["features"]
    scaled_data = scaler.transform([data])
    prediction = model.predict(scaled_data)[0]

    if prediction == 1:
        failure_alerts.inc()  # Increment alert count

    return jsonify({"prediction": int(prediction)})

if __name__ == "__main__":
    start_http_server(8000)  # Expose metrics at port 8000
    app.run(host="0.0.0.0", port=5000)
```

Configure Prometheus to scrape Flask app

Edit **prometheus.yml**:

yaml

```
scrape_configs:
  - job_name: 'predictive_alerts'
    static_configs:
      - targets: ['localhost:8000']
```

Run Prometheus:

```
./prometheus --config.file=prometheus.yml
```

Alertmanager Rules

Create alert_rules.yml:

yaml

```yaml
groups:
  - name: system_alerts
    rules:
      - alert: SystemFailure
        expr: system_failure_alerts > 0
        for: 1m
        labels:
          severity: critical
        annotations:
          summary: "Potential system failure detected!"
```

Run Alertmanager:

```
./alertmanager --config.file=alertmanager.yml
```

Step 6: Containerize & Deploy

Dockerfile

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

Build & Run:

```
docker build -t predictive-alerts .
docker run -p 5000:5000 predictive-alerts
```

Deploy on Kubernetes:

```
kubectl create deployment predictive-alerts --image=predictive-alerts
kubectl expose deployment predictive-alerts --type=NodePort --port=5000
```

Conclusion

This project enables **proactive incident management** by:

- **Analyzing system logs** to detect failure patterns
- **Predicting failures using AI models**
- **Alerting teams via Prometheus & Alertmanager**
- **Deploying the solution using Docker & Kubernetes**

---

10. AI for CI/CD & DevSecOps

Project 1. AI-Driven Test Suite Optimization: Using AI to automatically optimize the sequence of tests in CI/CD pipelines to reduce the overall pipeline runtime.

In modern CI/CD pipelines, running a full test suite can be time-consuming, delaying deployments. This project leverages **AI to optimize test execution order**, prioritizing tests based on past failures, execution time, and code changes. By running critical tests first, we can detect failures earlier and **reduce the overall pipeline runtime**.

Project Setup

Tech Stack

- **Python** (Machine Learning & Optimization)
- **Pytest** (Test framework)
- **GitHub Actions/Jenkins** (CI/CD)
- **SQLite** (Storing test history)

- **Docker** (Containerization)

Step 1: Set Up the Project

```
mkdir ai-test-optimizer && cd ai-test-optimizer
python3 -m venv venv
source venv/bin/activate  # On Windows: venv\Scripts\activate
pip install pytest numpy pandas scikit-learn sqlite3
```

This creates a virtual environment and installs necessary dependencies.

Step 2: Create a Sample Test Suite

Create a tests/ directory with sample test cases.

```
mkdir tests
```

Example: Sample Pytest Test Cases (tests/test_sample.py)

python

```python
import time
import random

def test_fast():
    """A fast test case"""
    time.sleep(1)
    assert True

def test_slow():
    """A slow test case"""
    time.sleep(3)
```

```
    assert True

def test_unstable():
    """A test that sometimes fails"""
    time.sleep(2)
    assert random.choice([True, False])
```

- test_fast() runs quickly
- test_slow() takes more time
- test_unstable() is flaky

Step 3: Store Test History in SQLite

We store execution time and failure history in a database to optimize the order of execution.

Create Database and Logger (test_logger.py)

python

```
import sqlite3
import time

DB_FILE = "test_history.db"

def setup_db():
    """Initialize the test history database"""
    conn = sqlite3.connect(DB_FILE)
    cursor = conn.cursor()
    cursor.execute("""
        CREATE TABLE IF NOT EXISTS test_history (
            test_name TEXT PRIMARY KEY,
            avg_runtime REAL,
            failure_count INTEGER
```

```
    )
  """)
  conn.commit()
  conn.close()

def log_test_result(test_name, runtime, failed):
  """Update test execution history"""
  conn = sqlite3.connect(DB_FILE)
  cursor = conn.cursor()

  cursor.execute("SELECT avg_runtime, failure_count FROM test_history
WHERE test_name=?", (test_name,))
  row = cursor.fetchone()

  if row:
    avg_runtime, failure_count = row
    new_runtime = (avg_runtime + runtime) / 2
    new_failures = failure_count + (1 if failed else 0)
    cursor.execute("UPDATE test_history SET avg_runtime=?, failure_count=?
WHERE test_name=?",
            (new_runtime, new_failures, test_name))
  else:
    cursor.execute("INSERT INTO test_history (test_name, avg_runtime,
failure_count) VALUES (?, ?, ?)",
            (test_name, runtime, 1 if failed else 0))

  conn.commit()
  conn.close()

setup_db()
```

This script:

- Creates an SQLite database to track test runtime and failures

- Logs test execution results

Step 4: AI Model to Prioritize Tests

We use **scikit-learn** to prioritize tests based on past failures and execution time.

Create AI Model (ai_test_optimizer.py)

python

```
import sqlite3
import pandas as pd
from sklearn.preprocessing import MinMaxScaler

DB_FILE = "test_history.db"

def get_prioritized_tests():
    """Fetch and sort tests based on AI-driven priority"""
    conn = sqlite3.connect(DB_FILE)
    df = pd.read_sql_query("SELECT * FROM test_history", conn)
    conn.close()

    if df.empty:
        return []

    # Normalize data
    scaler = MinMaxScaler()
    df[["avg_runtime", "failure_count"]] = scaler.fit_transform(df[["avg_runtime",
"failure_count"]])

    # Prioritize: Sort by failures (descending) & runtime (ascending)
    df["priority_score"] = df["failure_count"] - df["avg_runtime"]
    df = df.sort_values(by="priority_score", ascending=False)
```

```python
    return df["test_name"].tolist()

print(get_prioritized_tests())
```

This script:

- Fetches test data from the database
- Normalizes runtime and failure count
- Assigns priority (run failure-prone tests first, fast tests before slow ones)

Step 5: Run Tests in Optimized Order

Modify the test runner to execute prioritized tests.

Run Optimized Test Execution (run_tests.py)

python

```python
import pytest
import time
from ai_test_optimizer import get_prioritized_tests
from test_logger import log_test_result

def run_test(test_name):
    """Run a single test and log results"""
    start = time.time()
    result = pytest.main(["-q", f"tests/{test_name}.py"])
    end = time.time()

    log_test_result(test_name, end - start, result != 0)

def run_tests():
    """Run tests in AI-optimized order"""
    test_order = get_prioritized_tests()
```

```
    if not test_order:
        test_order = ["test_sample"]  # Default if no history

    for test in test_order:
        run_test(test)

if __name__ == "__main__":
    run_tests()
```

This script:

- Fetches prioritized tests
- Runs them one by one
- Logs results in the database

Step 6: Integrate with CI/CD (GitHub Actions or Jenkins)

GitHub Actions Workflow (.github/workflows/test_optimization.yml)

yaml

```yaml
name: AI-Test-Optimization
on: [push, pull_request]

jobs:
  run-tests:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout Code
        uses: actions/checkout@v3

      - name: Set Up Python
        uses: actions/setup-python@v4
```

```
    with:
      python-version: "3.9"

  - name: Install Dependencies
    run: |
      python -m venv venv
      source venv/bin/activate
      pip install pytest numpy pandas scikit-learn sqlite3

  - name: Run Optimized Tests
    run: |
      source venv/bin/activate
      python run_tests.py
```

Step 7: Run Everything

Run the following commands to test locally:

```
python test_logger.py       # Initialize database
python ai_test_optimizer.py # Check test order
python run_tests.py         # Run optimized tests
```

Conclusion

- This **AI-driven approach** prioritizes failure-prone and fast tests to **detect bugs earlier and reduce pipeline runtime**.
- The system continuously **learns from test results**, improving efficiency over time.
- It can be **integrated into any CI/CD pipeline** like Jenkins, GitHub Actions, or GitLab CI.

Project 2. AI for Continuous Security Assessment: Real-time security vulnerability detection during the CI/CD pipeline, integrated into DevSecOps practices.

Introduction

As security threats evolve, organizations must integrate continuous security assessment within their CI/CD pipelines. This project implements **AI-driven real-time security vulnerability detection**, ensuring DevSecOps compliance. By integrating AI-based tools, we automate security scanning and risk analysis at various CI/CD stages.

Project Overview

Technology Stack

- **CI/CD Tools**: Jenkins/GitHub Actions/GitLab CI
- **AI/ML for Security**: OpenAI API, ML Models (Scikit-learn, TensorFlow)
- **Security Tools**: OWASP Dependency-Check, Trivy, SonarQube
- **Containerization**: Docker, Kubernetes
- **Infrastructure as Code**: Terraform
- **Monitoring**: Prometheus, Grafana
- **Database**: PostgreSQL/MongoDB (for storing vulnerabilities)
- **Scripting**: Python, Shell

Step-by-Step Implementation

Step 1: Setup CI/CD Pipeline

1.1 Install Jenkins/GitHub Actions/GitLab CI

 # Install Jenkins (Ubuntu)
sudo apt update
sudo apt install openjdk-11-jdk -y
wget -q -O - https://pkg.jenkins.io/debian-stable/jenkins.io.key | sudo apt-key add -
sudo sh -c 'echo deb http://pkg.jenkins.io/debian-stable binary/ >
/etc/apt/sources.list.d/jenkins.list'
sudo apt update
sudo apt install jenkins -y
sudo systemctl start jenkins
sudo systemctl enable jenkins

For **GitHub Actions** or **GitLab CI**, configure .github/workflows/security.yml or .gitlab-ci.yml.

Step 2: AI-based Security Scanning

2.1 Integrate OWASP Dependency-Check for Vulnerability Analysis

 # Install OWASP Dependency-Check
wget
https://github.com/jeremylong/DependencyCheck/releases/download/v7.0.4/dependency-check-7.0.4-release.zip
unzip dependency-check-7.0.4-release.zip
cd dependency-check/bin
./dependency-check.sh --project "AI-Security-Scan" --scan /path/to/project

2.2 Automate Security Scanning in CI/CD
yaml

 # GitHub Actions - .github/workflows/security.yml

```yaml
name: Security Scan

on: [push]

jobs:
  security-check:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Run OWASP Dependency-Check
        run: ./dependency-check/bin/dependency-check.sh --project "AI-Security"
--scan .
```

Step 3: AI Integration for Threat Analysis

3.1 Build AI Model for Security

python

```python
# ai_security_model.py - Machine Learning Model for Security Analysis
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Load vulnerability dataset
data = pd.read_csv("vulnerability_data.csv")
X = data.drop(columns=["Risk_Level"])
y = data["Risk_Level"]

# Train ML Model
```

```python
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)
model.fit(X_train, y_train)

# Evaluate model
predictions = model.predict(X_test)
print("Model Accuracy:", accuracy_score(y_test, predictions))

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

Step 4: AI-based Risk Prediction in Pipeline

4.1 Integrate AI Model into CI/CD

yaml

```yaml
- name: AI Security Check
  run: python security_check.py
```

4.2 Security Assessment with AI

python

```python
# security_check.py - Use AI model in CI/CD
import joblib
import pandas as pd

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

# Scan new code vulnerabilities
```

```
new_scan = pd.read_csv("new_vulnerabilities.csv")
risk_predictions = model.predict(new_scan)

# Generate security report
for idx, risk in enumerate(risk_predictions):
    print(f"Vulnerability {idx+1}: Risk Level - {risk}")
```

Step 5: Deploy Secure Infrastructure using Terraform

5.1 Define Secure Cloud Resources

hcl

```
# main.tf - Terraform configuration
provider "aws" {
  region = "us-east-1"
}

resource "aws_s3_bucket" "security_logs" {
  bucket = "ai-security-logs"
  acl    = "private"
}
```

5.2 Apply Terraform Configuration

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

Step 6: Security Monitoring & Alerts

6.1 Setup Prometheus & Grafana

```
docker run -d -p 9090:9090 --name prometheus prom/prometheus
docker run -d -p 3000:3000 --name grafana grafana/grafana
```

6.2 Monitor Vulnerabilities in Real-time

yaml

```yaml
# Prometheus Alert for High-Risk Vulnerabilities
groups:
  - name: security_alerts
    rules:
      - alert: HighSeverityVulnerability
        expr: security_risk > 8
        for: 2m
        labels:
          severity: critical
        annotations:
          summary: "High-risk security vulnerability detected!"
```

Project Summary

✅ **Implemented CI/CD security scanning** with OWASP Dependency-Check
✅ **Integrated AI model** for real-time threat assessment
✅ **Automated security risk classification** using Machine Learning
✅ **Deployed secure infrastructure** with Terraform
✅ **Monitored vulnerabilities** using Prometheus & Grafana

Project 3. AI-Based Dependency Vulnerability Scanning: Implement AI-based scanning of dependencies in code repositories for potential vulnerabilities or license compliance issues.

Dependency vulnerabilities in software projects can lead to security risks and compliance violations. Traditional scanning tools like **OWASP Dependency-Check**, **Snyk**, or **Trivy** detect vulnerabilities, but AI can improve detection accuracy and predict potential risks. This project builds an **AI-powered** scanner that integrates machine learning models with existing vulnerability databases to enhance security scanning.

Project Steps

1. **Set Up Environment**
   - Install Python and required libraries
   - Set up a virtual environment
2. **Get Project Dependencies**
   - Clone a sample code repository
   - Extract dependencies (Maven, npm, pip, etc.)
3. **Collect Vulnerability Data**
   - Use sources like the National Vulnerability Database (NVD)
   - Parse Common Vulnerabilities and Exposures (CVE) data
4. **AI-Based Vulnerability Analysis**
   - Train a simple AI model to predict risk levels
   - Use NLP to analyze package descriptions
5. **Implement License Compliance Check**
   - Extract license information from dependencies
   - Cross-check against approved licenses
6. **Generate Reports and Alerts**
   - Store results in a database
   - Send alerts for critical vulnerabilities
7. **Integrate with CI/CD Pipeline**
   - Automate scanning in GitHub Actions or Jenkins

Step-by-Step Implementation

1. Set Up Environment

Install Python and create a virtual environment:

```
sudo apt update && sudo apt install python3 python3-venv -y
python3 -m venv venv
source venv/bin/activate
pip install --upgrade pip
```

Install required dependencies:

```
pip install requests beautifulsoup4 pandas scikit-learn tensorflow nltk
```

---

2. Clone a Sample Repository & Extract Dependencies

Clone a test project (Java, Node.js, Python, etc.):

```
git clone https://github.com/your-test-repo.git
cd your-test-repo
```

Extract dependencies:
For Python (pip):

```
pip freeze > requirements.txt
```

For Node.js (npm):

```
npm list --json > dependencies.json
```

For **Java (Maven)**:

```
mvn dependency:tree -DoutputType=text -DoutputFile=dependencies.txt
```

---

3. Fetch Vulnerability Data

Fetch vulnerability data from the **National Vulnerability Database (NVD)**:

python

```
import requests

NVD_API = "https://services.nvd.nist.gov/rest/json/cves/1.0"
def get_cve_data():
    response = requests.get(NVD_API)
    return response.json()

cve_data = get_cve_data()
print(cve_data)  # Sample CVE JSON output
```

---

4. AI-Based Vulnerability Detection

Use AI to classify dependency risks:

python

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression

# Sample training data
vulnerabilities = ["Critical SQL Injection vulnerability in package X",
                   "Minor dependency update issue in package Y"]
labels = [1, 0]  # 1 = High Risk, 0 = Low Risk

vectorizer = TfidfVectorizer()
X_train = vectorizer.fit_transform(vulnerabilities)
model = LogisticRegression()
model.fit(X_train, labels)

# Predict new risks
```

```
def predict_risk(description):
    X_test = vectorizer.transform([description])
    return model.predict(X_test)[0]

print(predict_risk("Security flaw found in package Z"))  # Output: 1 (High Risk) or 0 (Low Risk)
```

---

5. License Compliance Check

Extract and verify licenses:

python

```
import json

def check_license():
    with open("dependencies.json", "r") as f:
        data = json.load(f)

    for package, info in data["dependencies"].items():
        print(f"Package: {package}, License: {info.get('license', 'Unknown')}")

check_license()
```

---

6. Generate Reports

Save results in a CSV file:

python

```
import pandas as pd
```

```
results = [{"package": "numpy", "risk": "High"}, {"package": "requests", "risk": "Low"}]
df = pd.DataFrame(results)
df.to_csv("scan_results.csv", index=False)
```

---

7. Integrate with CI/CD (Jenkins Example)

Add this to your Jenkinsfile:

groovy

```groovy
pipeline {
    agent any
    stages {
        stage('Dependency Scan') {
            steps {
                sh 'python3 scan.py'
            }
        }
        stage('Check Results') {
            steps {
                sh 'cat scan_results.csv'
            }
        }
    }
}
```

Conclusion

This project builds an **AI-based Dependency Vulnerability Scanner** that:

- **Extracts** dependencies from code repositories
- **Fetches** vulnerability data from NVD

- **Uses AI** to classify risk levels
- **Checks licenses** for compliance
- **Generates reports** and integrates with **CI/CD**

Project 4. Automated Code Quality Review with AI: AI models that scan code during CI/CD builds and provide insights into code quality, security, and performance improvements.

Objective: Implement an AI-driven code quality review system in a CI/CD pipeline to analyze code for security, performance, and best practices.

Step-by-Step Guide

Step 1: Set Up the Project

Create a directory for the project

```
mkdir ai-code-review
cd ai-code-review
```

Initialize a Git repository

```
git init
```

Set up a Python virtual environment

```
python3 -m venv venv
source venv/bin/activate  # On Windows: venv\Scripts\activate
```

Install dependencies

```
pip install openai flake8 bandit
```

Step 2: Implement AI-Powered Code Review Script

- Create a Python script code_review.py to analyze code using **Flake8** (for style), **Bandit** (for security), and **OpenAI API** (for AI-driven insights).

python

```
import os
import openai
import subprocess

openai.api_key = "your_openai_api_key"

def run_command(command):
    """Execute a shell command and return output"""
    result = subprocess.run(command, shell=True, capture_output=True, text=True)
    return result.stdout.strip()

def analyze_code():
    """Run static analysis tools"""
    flake8_result = run_command("flake8 . --exclude=venv")
    bandit_result = run_command("bandit -r .")

    return f"Flake8 Report:\n{flake8_result}\n\nBandit Security
Report:\n{bandit_result}"

def ai_code_review(code_analysis):
    """Send analysis to OpenAI for insights"""
    response = openai.ChatCompletion.create(
        model="gpt-4",
        messages=[{"role": "system", "content": "You are an expert code reviewer."},
                  {"role": "user", "content": f"Analyze this report and provide
suggestions:\n{code_analysis}"}]
    )
    return response["choices"][0]["message"]["content"]

if __name__ == "__main__":
```

```
    report = analyze_code()
    ai_suggestions = ai_code_review(report)
    print("=== AI Code Review Suggestions ===")
    print(ai_suggestions)
```

Step 3: Set Up a CI/CD Pipeline in GitHub Actions

- **Create .github/workflows/code_review.yml**

yaml

```
name: AI Code Review

on: [push, pull_request]

jobs:
  review:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout Code
        uses: actions/checkout@v4

      - name: Set Up Python
        uses: actions/setup-python@v4
        with:
          python-version: '3.10'

      - name: Install Dependencies
        run: |
          python -m venv venv
          source venv/bin/activate
          pip install openai flake8 bandit

      - name: Run AI Code Review
        run: python code_review.py
```

```
    env:
      OPENAI_API_KEY: ${{ secrets.OPENAI_API_KEY }}
```

Step 4: Commit and Push Code

```
git add .
git commit -m "Add AI Code Review"
git push origin main
```

Step 5: Review AI Code Analysis in GitHub Actions

Once the GitHub Action runs, it will analyze your code, check for issues, and provide AI-generated suggestions.

1. **analyze_code()**
   - Runs flake8 for style checks.
   - Runs bandit for security scans.
   - Collects reports for AI processing.
2. **ai_code_review()**
   - Sends the analysis to OpenAI's GPT-4 model for review.
   - Receives feedback on improvements.
3. **CI/CD Pipeline**
   - Runs automatically on every push/pull request.
   - Installs dependencies and executes the review script.

Project 5. AI-Enhanced Test Failure Analysis: Using AI to automatically analyze failed tests in CI/CD pipelines and suggest possible causes and fixes.

Introduction

In CI/CD pipelines, test failures can slow down development. This project automates test failure analysis using **AI**. It collects failure logs from Jenkins, processes them using **NLP (Natural Language Processing)**, and uses **OpenAI GPT** to suggest possible causes and fixes.

Step 1: Setting Up the Environment

Prerequisites

- Jenkins installed and running
- Python (>=3.8) installed
- Docker installed
- OpenAI API key

Required Python Libraries

pip install openai requests flask

Step 2: Jenkins Job Setup

Jenkinsfile Configuration

This pipeline will run tests and send failure logs to our AI-powered analysis tool.

groovy

```
pipeline {
    agent any
    stages {
        stage('Checkout') {
            steps {
                git 'https://github.com/your-repo/your-project.git'
            }
        }
```

```
        stage('Run Tests') {
            steps {
                script {
                    def testResult = sh(script: 'pytest --tb=short > test_output.log; echo $?',
returnStatus: true)
                    archiveArtifacts artifacts: 'test_output.log', fingerprint: true
                    if (testResult != 0) {
                        sh 'curl -X POST -F "file=@test_output.log"
http://localhost:5000/analyze'
                        error("Tests failed. Check AI analysis.")
                    }
                }
            }
        }
    }
}
```

- Runs tests with **pytest**
- Captures failures in **test_output.log**
- Sends the log to the AI-powered Flask service

Step 3: Creating the AI Service with Flask

Flask API (ai_analysis.py)

python

```
from flask import Flask, request, jsonify
import openai
import os

app = Flask(__name__)
```

```
# Set your OpenAI API key
openai.api_key = os.getenv("OPENAI_API_KEY")

@app.route('/analyze', methods=['POST'])
def analyze():
    if 'file' not in request.files:
        return jsonify({'error': 'No file uploaded'}), 400

    file = request.files['file']
    log_data = file.read().decode('utf-8')

    prompt = f"Analyze the following test failure logs and suggest possible causes and fixes:\n\n{log_data}"

    response = openai.ChatCompletion.create(
        model="gpt-4",
        messages=[{"role": "user", "content": prompt}]
    )

    ai_suggestion = response['choices'][0]['message']['content']
    return jsonify({'suggestion': ai_suggestion})

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

- Reads the **test logs**
- Sends them to **GPT-4**
- Returns **possible causes & fixes**

Step 4: Running the AI Service in Docker

Dockerfile

dockerfile

```
FROM python:3.8
WORKDIR /app
COPY ai_analysis.py .
RUN pip install flask openai
CMD ["python", "ai_analysis.py"]
```

Build and Run the Container

```
docker build -t ai-test-analyzer .
docker run -d -p 5000:5000 --env OPENAI_API_KEY=your_api_key
ai-test-analyzer
```

Step 5: Running the Complete Setup

Start Jenkins Pipeline

1. **Push your code to GitHub**
2. **Trigger the Jenkins job**
3. **Jenkins runs tests, collects failures**
4. **Failed logs sent to AI service**
5. **AI suggests fixes in Jenkins logs**

Example Output

Test Failure Log (test_output.log) makefile

```
AssertionError: Expected 200 but got 500
```

AI Suggestion

pgsql

Possible Cause: The API endpoint might be returning a 500 due to an unhandled exception.
Fix: Check application logs for errors. Validate input parameters. Ensure database connection is active.

Summary

- **Automates test failure analysis** using AI
- **Saves developers time** debugging failures
- **Easily integrates into CI/CD pipelines**

---

11. AI for Infrastructure & Network Monitoring

Project 1. AI-Powered Load Forecasting for Infrastructure: Predicting infrastructure load for upcoming days or weeks using historical data and adjusting resource allocation accordingly.

This project predicts infrastructure load (such as CPU, memory, or network usage) for upcoming days or weeks using historical data. The goal is to optimize resource allocation by analyzing past trends and forecasting future demands with machine learning.

Step 1: Setting Up the Environment

Before starting, ensure you have Python and essential libraries installed.

Install Required Packages

pip install pandas numpy scikit-learn matplotlib seaborn tensorflow

---

Step 2: Data Collection & Preprocessing

We assume the dataset contains historical infrastructure usage data, including timestamps, CPU load, memory usage, and network activity.

Load the Dataset

python

```
import pandas as pd

# Load dataset
df = pd.read_csv("infrastructure_usage.csv", parse_dates=["timestamp"])

# Display first few rows
print(df.head())
```

Handle Missing Data

python

```
df = df.fillna(method="ffill")  # Forward fill missing values
```

Feature Engineering

python

```
df["hour"] = df["timestamp"].dt.hour
df["day_of_week"] = df["timestamp"].dt.dayofweek
df["month"] = df["timestamp"].dt.month
```

Step 3: Data Visualization

Plot CPU Usage Over Time

python

```python
import matplotlib.pyplot as plt

plt.figure(figsize=(10,5))
plt.plot(df["timestamp"], df["cpu_load"], label="CPU Load")
plt.xlabel("Time")
plt.ylabel("CPU Load")
plt.title("CPU Load Over Time")
plt.legend()
plt.show()
```

Step 4: Train-Test Split

python

```python
from sklearn.model_selection import train_test_split

X = df[["hour", "day_of_week", "month", "cpu_load"]].values
y = df["cpu_load"].shift(-1).fillna(0).values  # Predicting next time step

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, shuffle=False)
```

Step 5: Building a Machine Learning Model

We will use **LSTM (Long Short-Term Memory)**, a type of neural network effective for time-series forecasting.

Prepare Data for LSTM

python

```
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense
```

 # Reshape data for LSTM

```
X_train = np.reshape(X_train, (X_train.shape[0], 1, X_train.shape[1]))
X_test = np.reshape(X_test, (X_test.shape[0], 1, X_test.shape[1]))
```

Define LSTM Model

python

```
model = Sequential([
    LSTM(50, return_sequences=True, input_shape=(1, X_train.shape[2])),
    LSTM(50, return_sequences=False),
    Dense(25),
    Dense(1)
])

model.compile(optimizer="adam", loss="mean_squared_error")
model.fit(X_train, y_train, epochs=10, batch_size=32)
```

Step 6: Model Evaluation & Prediction

python

```
predictions = model.predict(X_test)

plt.figure(figsize=(10,5))
plt.plot(y_test, label="Actual Load")
plt.plot(predictions, label="Predicted Load", linestyle="dashed")
```

```python
plt.xlabel("Time")
plt.ylabel("CPU Load")
plt.title("Infrastructure Load Forecasting")
plt.legend()
plt.show()
```

Step 7: Deployment (Optional - Using Flask)

To deploy the model as an API, create a Flask app.

Install Flask

```
pip install flask
```

Create app.py

```python
from flask import Flask, request, jsonify
import numpy as np
import tensorflow as tf

app = Flask(__name__)
model = tf.keras.models.load_model("load_forecasting_model.h5")

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

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

Run the API

```
python app.py
```

Test API with Curl

```
curl -X POST http://127.0.0.1:5000/predict -H "Content-Type: application/json" -d
'{"features": [10, 3, 7, 50]}'
```

Conclusion

This project used LSTM to forecast infrastructure load and built an API for real-world integration. It helps DevOps teams optimize resource allocation and prevent over-provisioning or downtime.

Project 2. Proactive Infrastructure Health Monitoring: AI model for identifying potential infrastructure failures before they occur by monitoring system health in real time.

Infrastructure failures in IT systems can lead to downtime, security risks, and financial losses. A **Proactive Infrastructure Health Monitoring System** leverages **AI and real-time monitoring** to detect potential failures before they occur. It analyzes system health metrics, predicts issues, and alerts administrators to take preventive action.

In this project, we will build an **AI-driven monitoring system** using **Python, Flask, Prometheus, Grafana, and Machine Learning (Scikit-learn/PyTorch)**. This system collects system health metrics (CPU, memory, disk usage), trains an AI model to predict failures, and visualizes real-time data.

Project Setup & Steps

Step 1: Install Dependencies

Before starting, ensure you have Python and necessary tools installed.

```
sudo apt update && sudo apt upgrade -y
sudo apt install python3 python3-pip -y
pip install flask prometheus_client psutil pandas scikit-learn matplotlib
```

Step 2: Build the System Metrics Collector

Create a Python script to collect CPU, memory, and disk usage metrics.

Create metrics_collector.py

python

```python
from flask import Flask, Response
import psutil
from prometheus_client import Gauge, generate_latest

app = Flask(__name__)

# Define Prometheus metrics
cpu_usage = Gauge("cpu_usage", "CPU Usage Percentage")
memory_usage = Gauge("memory_usage", "Memory Usage Percentage")
disk_usage = Gauge("disk_usage", "Disk Usage Percentage")

@app.route("/metrics")
def metrics():
    cpu_usage.set(psutil.cpu_percent(interval=1))
    memory_usage.set(psutil.virtual_memory().percent)
    disk_usage.set(psutil.disk_usage("/").percent)
    return Response(generate_latest(), mimetype="text/plain")
```

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

Run the Metrics Collector

```
python3 metrics_collector.py
```

Your system's health metrics will be available at http://localhost:5000/metrics.

Step 3: Train an AI Model to Predict Failures

We will use a simple machine learning model to predict system failures based on collected data.

Create train_model.py

python

```
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

# Generate synthetic data
data = {
    "cpu_usage": [10, 20, 50, 90, 95, 80, 60, 40],
    "memory_usage": [30, 40, 50, 85, 90, 70, 60, 50],
    "disk_usage": [40, 50, 60, 80, 85, 70, 65, 55],
    "failure": [0, 0, 0, 1, 1, 1, 0, 0]  # 1 = Failure, 0 = Normal
}

df = pd.DataFrame(data)
```

# Split dataset
X = df[["cpu_usage", "memory_usage", "disk_usage"]]
y = df["failure"]
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)
model.fit(X_train, y_train)

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

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

Run the Model Training

python3 train_model.py

The trained model will be saved as failure_prediction_model.pkl.

Step 4: Deploy an API for AI Predictions

We will create a Flask API that takes real-time metrics and predicts potential failures.

Create predict_failure.py

python

from flask import Flask, request, jsonify

```python
import joblib
import psutil

app = Flask(__name__)

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

@app.route("/predict", methods=["GET"])
def predict():
    # Get real-time system metrics
    data = {
        "cpu_usage": psutil.cpu_percent(interval=1),
        "memory_usage": psutil.virtual_memory().percent,
        "disk_usage": psutil.disk_usage("/").percent,
    }

    # Make prediction
    prediction = model.predict([[data["cpu_usage"], data["memory_usage"], data["disk_usage"]]])
    result = "Failure predicted! Take action!" if prediction[0] == 1 else "System is healthy."

    return jsonify({"metrics": data, "prediction": result})

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5001)
```

Run the AI Prediction API

```
python3 predict_failure.py
```

Now, visit http://localhost:5001/predict to see real-time predictions.

Step 5: Setup Prometheus for Monitoring

Prometheus will scrape our metrics and store them for analysis.

Install Prometheus

```
wget
https://github.com/prometheus/prometheus/releases/latest/download/prometheus-linux-amd64.tar.gz
tar -xvf prometheus-linux-amd64.tar.gz
cd prometheus-linux-amd64
```

Edit prometheus.yml

Add the following under scrape_configs:

yaml

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

Run Prometheus

```
./prometheus --config.file=prometheus.yml
```

Prometheus UI will be available at http://localhost:9090.

Step 6: Setup Grafana for Visualization

Grafana will display real-time system health data.

Install Grafana

```
sudo apt install -y software-properties-common
```

sudo add-apt-repository "deb https://packages.grafana.com/oss/deb stable main"
sudo apt update
sudo apt install grafana -y

Start Grafana

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

Access Grafana UI

Visit http://localhost:3000 (default username/password: admin/admin).

Add Prometheus as a Data Source

- Go to **Settings > Data Sources > Add Prometheus**
- URL: http://localhost:9090

Create Dashboards

- Import a dashboard and select **cpu_usage, memory_usage, and disk_usage** as metrics.

Final Architecture

1. **Metrics Collector** (Flask) → Sends system health data to Prometheus
2. **AI Model** (Scikit-learn) → Predicts failures
3. **Prediction API** (Flask) → Provides real-time failure warnings
4. **Prometheus** → Stores and queries metrics
5. **Grafana** → Visualizes data for monitoring

Step 7: Automate with Docker (Optional)

Create Dockerfile

dockerfile

```
FROM python:3.9
WORKDIR /app
COPY . .
RUN pip install -r requirements.txt
CMD ["python3", "metrics_collector.py"]
```

Build & Run the Container

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

Conclusion

This **Proactive Infrastructure Health Monitoring System** allows organizations to predict and prevent system failures using **AI-driven monitoring**. By integrating **Flask, Prometheus, Grafana, and ML models**, we gain **real-time insights** into system health, reducing downtime risks.

Project 3. Network Traffic Anomaly Detection with AI: Using machine learning to detect outliers in network traffic data (e.g., unusual spikes or drops), potentially identifying attacks.

Network security is a crucial aspect of modern digital infrastructure. Detecting anomalies in network traffic can help identify potential security threats, such as **DDoS attacks, data exfiltration, or unauthorized access**.

In this project, we will use **Machine Learning (ML)** to detect unusual traffic patterns using unsupervised learning techniques like **Isolation Forest** and **One-Class SVM**.

Project Setup

1. Install Required Libraries

Before starting, install the necessary Python libraries:

pip install pandas numpy scikit-learn matplotlib seaborn

Step-by-Step Implementation

Step 1: Import Libraries

python

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.ensemble import IsolationForest
from sklearn.svm import OneClassSVM
from sklearn.preprocessing import StandardScaler
```

Step 2: Load and Explore the Dataset

For this project, we will use a synthetic dataset. However, you can also use real datasets like **CICIDS2017** or **KDDCup99**.

python

```
# Create synthetic network traffic data
np.random.seed(42)
normal_traffic = np.random.normal(loc=50, scale=10, size=(1000, 2))
```

anomalous_traffic = np.random.normal(loc=100, scale=20, size=(50, 2)) # Simulating attacks

 # Combine normal and anomalous traffic
data = np.vstack((normal_traffic, anomalous_traffic))
labels = np.array([0] * 1000 + [1] * 50) # 0 = normal, 1 = anomaly

 # Convert to DataFrame
df = pd.DataFrame(data, columns=['Packets_Per_Second', 'Bytes_Per_Second'])
df['Anomaly'] = labels

 # Display first few rows
print(df.head())

 # Plot data distribution
sns.scatterplot(x=df['Packets_Per_Second'], y=df['Bytes_Per_Second'], hue=df['Anomaly'])
plt.title('Network Traffic Data')
plt.show()

📌 **Explanation:**

- We generate normal traffic using a normal distribution.
- We introduce anomalies to simulate unusual traffic patterns.
- The dataset contains two features: **Packets per Second** and **Bytes per Second**.

Step 3: Preprocess the Data

python

```
scaler = StandardScaler()
df[['Packets_Per_Second', 'Bytes_Per_Second']] = scaler.fit_transform(df[['Packets_Per_Second', 'Bytes_Per_Second']])
```

📌 **Why Standardization?**

- Since ML models work better with normalized data, we use **StandardScaler** to bring all values into a common range.

Step 4: Train the Isolation Forest Model

python

```
iso_forest = IsolationForest(contamination=0.05, random_state=42)
df['Anomaly_Score'] = iso_forest.fit_predict(df[['Packets_Per_Second',
'Bytes_Per_Second']])

# Replace -1 with 1 for anomaly detection
df['Anomaly_Detected'] = (df['Anomaly_Score'] == -1).astype(int)

# Display detected anomalies
print(df[df['Anomaly_Detected'] == 1].head())
```

📌 **Explanation:**

- **Isolation Forest** isolates anomalies by recursively partitioning data.
- **contamination=0.05** assumes 5% of data is anomalous.
- The model predicts -1 for anomalies and 1 for normal data.

Step 5: Train One-Class SVM Model (Alternative Approach)

python

```
oc_svm = OneClassSVM(nu=0.05, kernel="rbf", gamma='scale')
df['SVM_Anomaly_Score'] = oc_svm.fit_predict(df[['Packets_Per_Second',
'Bytes_Per_Second']])
```

```
df['SVM_Anomaly_Detected'] = (df['SVM_Anomaly_Score'] == -1).astype(int)

# Display detected anomalies
print(df[df['SVM_Anomaly_Detected'] == 1].head())
```

📌 **Explanation:**

- **One-Class SVM** is another unsupervised anomaly detection method.
- It learns the normal behavior and flags deviations.

Step 6: Visualize the Anomalies

python

```
plt.figure(figsize=(10, 6))
sns.scatterplot(data=df, x='Packets_Per_Second', y='Bytes_Per_Second',
hue='Anomaly_Detected', palette={0: 'blue', 1: 'red'})
plt.title('Anomaly Detection using Isolation Forest')
plt.show()
```

📌 **Visualization:**

- Normal traffic points are shown in **blue**.
- Detected anomalies are marked in **red**.

Step 7: Evaluate the Model

python

```
from sklearn.metrics import classification_report

print("Isolation Forest Report:")
print(classification_report(df['Anomaly'], df['Anomaly_Detected']))
```

```
print("One-Class SVM Report:")
print(classification_report(df['Anomaly'], df['SVM_Anomaly_Detected']))
```

📌 **Evaluation Metrics:**

- **Precision:** How many detected anomalies are actual anomalies?
- **Recall:** How many actual anomalies were detected?
- **F1-Score:** Balances precision and recall.

Conclusion

🔹 This project demonstrated how **Machine Learning** can detect network anomalies.
🔹 **Isolation Forest** and **One-Class SVM** help find outliers in network traffic data.
🔹 The model can be extended using real-time data from **Wireshark, NetFlow, or cloud monitoring logs**.
🔹 Future improvements include deep learning models like **Autoencoders** for better accuracy.

Project 4. Distributed Network Monitoring with AI: AI to monitor network performance across distributed environments (hybrid clouds, multi-region setups) and provide insights.

In modern IT infrastructure, network monitoring is crucial, especially in **hybrid cloud** and **multi-region setups**. Traditional monitoring tools often struggle with **scalability** and **real-time insights**. This project leverages **AI-powered monitoring** to:

- Track **network performance** across distributed environments
- Detect **anomalies** in network traffic
- Provide **predictive insights** using **Machine Learning (ML)**

We'll use:

- **Python** for backend development
- **Prometheus & Grafana** for monitoring & visualization
- **Scapy** & **TShark** for packet analysis
- **TensorFlow/PyTorch** for AI-based anomaly detection
- **Docker & Kubernetes** for deployment

Step-by-Step Implementation

Step 1: Install Dependencies

Ensure you have Python, Prometheus, and Grafana installed.

```
# Update system
sudo apt update && sudo apt upgrade -y

# Install Python & Virtual Environment
sudo apt install python3 python3-pip python3-venv -y

# Create a virtual environment
python3 -m venv venv
source venv/bin/activate

# Install required Python libraries
pip install scapy tensorflow pandas numpy matplotlib prometheus_client flask requests
```

Step 2: Set Up Prometheus for Network Metrics Collection

Download & install **Prometheus**

```
wget
https://github.com/prometheus/prometheus/releases/latest/download/prometheus-linux-amd64.tar.gz
tar -xvf prometheus-linux-amd64.tar.gz
cd prometheus-linux-amd64
```

Configure Prometheus (prometheus.yml)

yaml

```yaml
global:
  scrape_interval: 10s

scrape_configs:
  - job_name: "network-monitor"
    static_configs:
      - targets: ["localhost:8000"]  # Flask API exposing network metrics
```

Start Prometheus

sh

```sh
./prometheus --config.file=prometheus.yml
```

Step 3: Build the Network Monitoring Script (Python API)

Create network_monitor.py

python

```
from flask import Flask, jsonify
from prometheus_client import start_http_server, Gauge
import scapy.all as scapy
import time
import random

app = Flask(__name__)

# Prometheus metrics
packet_count = Gauge('network_packet_count', 'Number of packets captured')
packet_size = Gauge('network_packet_size', 'Total size of packets captured')

def capture_traffic():
    packets = scapy.sniff(count=10)
    total_size = sum(len(p) for p in packets)

    packet_count.set(len(packets))
    packet_size.set(total_size)

@app.route('/metrics')
def metrics():
    capture_traffic()
    return jsonify({'packet_count': packet_count._value.get(), 'packet_size': packet_size._value.get()})

if __name__ == '__main__':
    start_http_server(8000)
    app.run(host='0.0.0.0', port=5000)
```

Step 4: Implement AI for Anomaly Detection

Create anomaly_detection.py

python

```
import pandas as pd
import numpy as np
import tensorflow as tf
from sklearn.preprocessing import MinMaxScaler

# Simulated network data
data = np.array([[random.randint(100, 5000), random.randint(10, 200)] for _ in
range(100)])
df = pd.DataFrame(data, columns=["packet_size", "latency"])

# Normalize data
scaler = MinMaxScaler()
df_scaled = scaler.fit_transform(df)

# Create simple autoencoder for anomaly detection
model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(8, activation='relu', input_shape=(2,)),
    tf.keras.layers.Dense(4, activation='relu'),
    tf.keras.layers.Dense(8, activation='relu'),
    tf.keras.layers.Dense(2, activation='sigmoid')
])

model.compile(optimizer='adam', loss='mse')
model.fit(df_scaled, df_scaled, epochs=10, batch_size=8)

# Predict on new data
new_data = np.array([[4500, 180]])  # Example high packet size & latency
new_data_scaled = scaler.transform(new_data)
reconstruction = model.predict(new_data_scaled)

# Compute anomaly score
anomaly_score = np.mean(np.abs(new_data_scaled - reconstruction))
print("Anomaly Score:", anomaly_score)
```

Step 5: Deploy on Docker & Kubernetes

Dockerfile

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

Build & Run Docker Container

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

Deploy to Kubernetes (network-monitor.yaml)

yaml

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

```yaml
    containers:
      - name: network-monitor
        image: network-monitor:latest
        ports:
          - containerPort: 5000

---
apiVersion: v1
kind: Service
metadata:
  name: network-monitor-service
spec:
  selector:
    app: network-monitor
  ports:
    - protocol: TCP
      port: 80
      targetPort: 5000
  type: LoadBalancer
```

Deploy on Kubernetes:

```
kubectl apply -f network-monitor.yaml
```

Step 6: Visualize Metrics in Grafana

Install **Grafana**

```
sudo apt install -y grafana
sudo systemctl start grafana-server
sudo systemctl enable grafana-server
```

Configure Data Source:

- Go to http://localhost:3000
- Login (admin/admin)
- Add **Prometheus** as a data source
- Query: {network_packet_count} & {network_packet_size}

- **Flask API (network_monitor.py)**:
  - Captures **network packets** and exposes **Prometheus metrics**
  - Used to integrate with **Grafana**
- **AI Model (anomaly_detection.py)**:
  - Uses **TensorFlow Autoencoder** for detecting **unusual network activity**
- **Docker & Kubernetes**:
  - **Docker**: Packages the app into a container
  - **Kubernetes**: Deploys across distributed cloud environments
- **Grafana**:
  - Visualizes **network metrics**

Conclusion

This project provides **real-time network monitoring** with **AI-powered anomaly detection**. It integrates with **Prometheus & Grafana** for visualization and can be **scaled** across **multi-cloud & hybrid environments** using **Kubernetes**.