Use Case: AI & ML Agent for Predicting and Remediating Kubernetes Cluster Issues

Team Name : RaKri
Team Members : YUVA SRI B, VIDYA B, SRUTHI B

1. Issue Detection & Remediation Workflow (Detailed)

The workflow in Phase 2 is designed to mimic a real-time AI-driven self-healing system for Kubernetes clusters. This workflow explains how the system transitions from data ingestion to automatic remediation using the previously trained model.

1. Real-Time Cluster Data Ingestion (Simulation)

In real-world Kubernetes environments, cluster data such as CPU usage, memory consumption, network I/O, and pod health is continuously logged by monitoring systems.
Since live telemetry isn't available during development, we simulate this by streaming a pre-collected CSV file (`sys_failure.csv`) containing time-series system metrics.

- Python reads the CSV line-by-line with a delay (`time.sleep()`), simulating real-time log ingestion.
- Each row represents the current snapshot of system health metrics.

2. Model Loading and Prediction Execution

The trained model from Phase 1 (`phase1_model.pkl`) is loaded using `joblib`. This model has learned from historical failures and is capable of identifying patterns that suggest an upcoming **service disruption**, **node/pod failure**, or similar cluster issues.

- Each row of input metrics is preprocessed (dropping timestamp or labels).
- The model receives this input and returns a binary prediction:
  - `0` = Healthy
  - `1` = Issue Detected

3. Prediction Analysis and Condition Evaluation

When the model predicts a failure (i.e., output is `1`), the system performs a **conditional check** to determine the type or severity of the issue.

For example:

- If CPU usage > 85% and prediction == 1 → likely an overloaded node
- If memory usage is consistently high with failure prediction → potential pod crash risk

4. Remediation Action Trigger via Kubernetes API

Upon confirming the issue, the Python script connects to the Kubernetes cluster using the official Kubernetes Python client. It then performs **real-time remediation** by executing one or more of the following:

- **Scaling up a deployment** (e.g., increasing pod replicas)
- **Restarting a pod**
- **Tainting a node**
- **Sending alerts or logs**

5. Kubernetes Responds to the Remediation

After the remediation command is issued, Kubernetes applies the change. For example:

- The deployment is scaled from 2 to 5 replicas
- New pods are scheduled and started
- Cluster state is updated and can be verified with `kubectl get pods`

Feedback Loop: Actions are logged on CLI and also optionally verified by checking the state of deployments/pods again.

6. Continuous Monitoring

The entire loop can run continuously or on a timed schedule to act like a lightweight **self-healing controller agent**.

2. Prerequisites Installation

Set up your system with the necessary tools and configurations to run the AI-powered Kubernetes Remediation Agent.

1. Python & Pip Installation
   ```
   python --version
   pip --version
   ```
2. Install Required Python Libraries
   Install the following packages for model loading and Kubernetes interaction:
   ```
   pip install kubernetes pandas scikit-learn joblib
   ```
3. Install `kubectl` (Kubernetes CLI)
4. Start Minikube or Kubernetes Cluster

minikube start
kubectl get nodes

5. Test `kubectl` Connection

bash

```
kubectl get pods -A
```

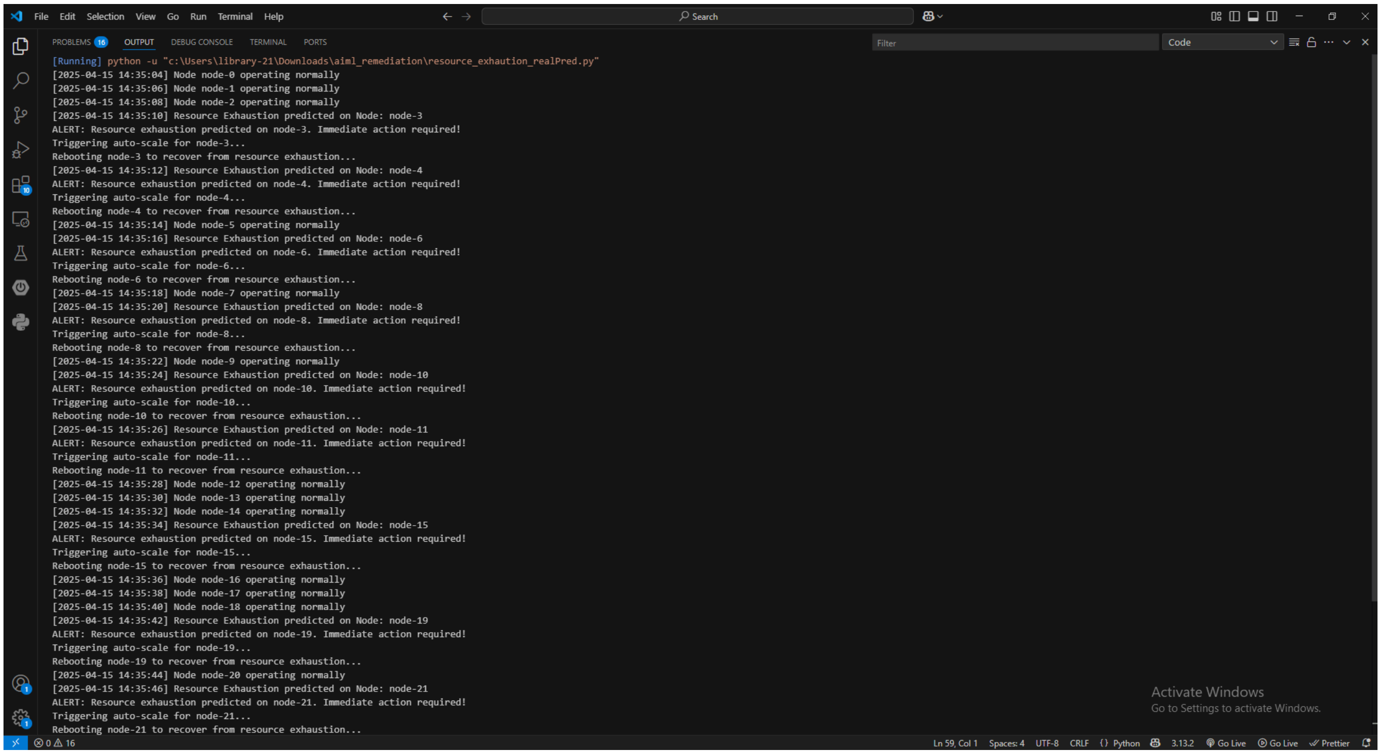
2. Resource Exhaustion Detection

This detection is designed to **predict and respond to resource exhaustion** in Kubernetes nodes using a pre-trained **Isolation Forest model**. It simulates the real-time monitoring of system metrics and performs automated responses based on prediction results.

1. **Model Loading**
   - A pre-trained machine learning model is used to detect abnormal resource usage.
   - The model expects three input features: **CPU usage**, **Memory usage**, and **Disk usage**.
2. **Metric Simulation**
   - System metrics are randomly generated to simulate real-time usage patterns.
   - These values represent live readings from multiple Kubernetes nodes.
3. **Prediction**
   - The simulated metrics are passed to the model to check for signs of resource exhaustion.
   - The model returns a prediction indicating whether the system is operating normally or experiencing abnormal behavior.
4. **Remediation Actions**
   - If resource exhaustion is predicted:
     - An **alert** is sent (simulated as a console message).
     - An **auto-scaling** action is triggered to add resources (simulated).
     - As a last resort, a **node reboot** is initiated (also simulated).
   - If no issue is detected, the node is logged as operating normally.
5. **Continuous Monitoring**
   - The script continuously monitors multiple nodes in a loop.

- ○ Every few seconds, it checks one node's simulated metrics and reacts based on the prediction.

OUTPUT :



Normal Operation Logs

Lines like the ones below indicate that the node is functioning properly, with no signs of resource exhaustion:

- `Node node-1 operating normally`
- `Node node-5 operating normally`
- `Node node-8 operating normally`

These messages are printed every few seconds for different nodes, showing that the system is being actively monitored.

Resource Exhaustion Detected

Whenever a node shows signs of **high resource usage (CPU, memory, or disk)** that could lead to failure, the script:

1. Predicts a potential issue.
2. Logs a message like:

        ◦ `Resource Exhaustion predicted on Node: node-4`

3. Triggers automatic actions:
    ◦ **Sends an alert**:
      `ALERT: Resource exhaustion predicted on node-4. Immediate action required!`
    ◦ **Initiates auto-scaling**:
      `Triggering auto-scale for node-4…`
    ◦ **Simulates a reboot**:
      `Rebooting node-4 to recover from resource exhaustion...`

These steps show that the system is **not just detecting problems, but also actively responding** to try to fix them.

Here is a **simple documentation (explanation only)** of what the code does — specifically designed for **network usage anomaly detection and remediation**, **without including any code**.

3. Network Usage Remediation

Overview

This script performs real-time monitoring of **network usage in a Kubernetes cluster** using a pre-trained machine learning model (ARIMA). The system detects high usage and automates responses like scaling pods, restarting problem pods, and enabling autoscaling.

1. Model and Data Setup

- A trained **ARIMA model** is loaded, which was previously trained to forecast network usage patterns.
- The dataset (`sys_failure.csv`) containing time-series logs of system metrics is loaded for monitoring.

2. Data Preprocessing

- Percentage values from **CPU, Memory, and Disk usage** columns are cleaned and converted to numeric format.
- **Categorical fields** such as Network Usage, Pod Status, System Logs, etc., are encoded numerically to be compatible with model input.
- The **timestamp** column is converted into datetime format and set as the index to enable time-series forecasting.

3. Remediation Functions

The system defines the following actions to take in response to high network usage:

- **Scale Pods**: Increases the number of replicas of a deployment to manage load.
- **Restart Pod**: Deletes and restarts a specific pod to clear potential issues.
- **Enable Autoscaling**: Automatically adjusts the number of replicas based on CPU usage between defined minimum and maximum limits.

4. Real-Time Monitoring Simulation

The script uses a **sliding window approach** to simulate real-time prediction:

- For every new time point in the dataset, the last 10 network usage values are used to forecast the next value.
- The system compares the **predicted value** and **actual usage** against a threshold (80%).

5. Anomaly Detection and Action

If either the forecasted or actual **network usage exceeds 80%**, the system takes immediate action:

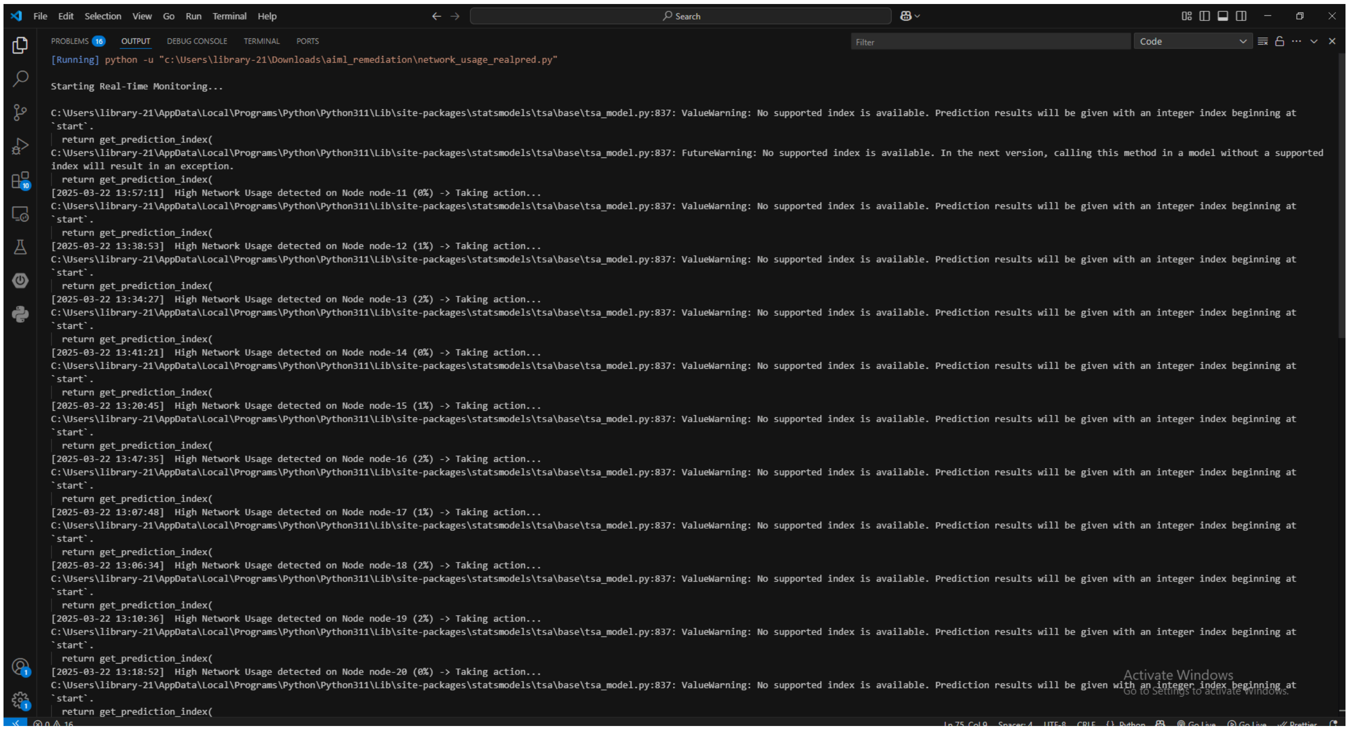
1. Logs a message highlighting high usage on a specific node.
2. **Triggers scaling** of the deployment.
3. **Restarts** the affected pod.
4. **Enables autoscaling** for better future load management.

Otherwise, it logs that the node is healthy and continues monitoring.

6. Time Delay Simulation

To mimic real-time operation, a small **delay of 1 second** is introduced between each iteration, representing a live system continuously checking network health.

OUTPUT :



- The script is **running real-time monitoring** of network usage for different nodes.
- When **high network usage** is detected (e.g., above 80%)-
- This means the script:
    - Detected that **Node node-12** is using **91%** of network bandwidth.
    - It is **taking action** like:
        - Scaling up pods,
        - Restarting affected pod,
        - Enabling autoscaling.

4. Node/Pod Failure Detection

1. **Model Loading and Preprocessing**:
    - The script begins by **loading the pre-trained model** (`pod_random_forest.pkl`) that was used for predicting node/pod failures.
    - It then loads the **system failure dataset** (`sys_failure.csv`) which contains columns like CPU, Memory, Disk, and Network usage data.
    - Columns that represent **percentage values** (CPU, Memory, Disk Usage) are cleaned to remove percentage signs and converted to **float** type for proper processing.

    - **Label encoding** is applied to the `Network_Usage` column to transform the categorical values into numerical ones, as required by the model.
2. **Kubernetes API Setup**:
    - The script loads the **Kubernetes configuration** (`kubeconfig`) to access the Kubernetes cluster and initialize the Kubernetes API client.
    - With this setup, the script can interact with the Kubernetes cluster to perform actions like restarting pods.

3. **Pod Restart Function**:

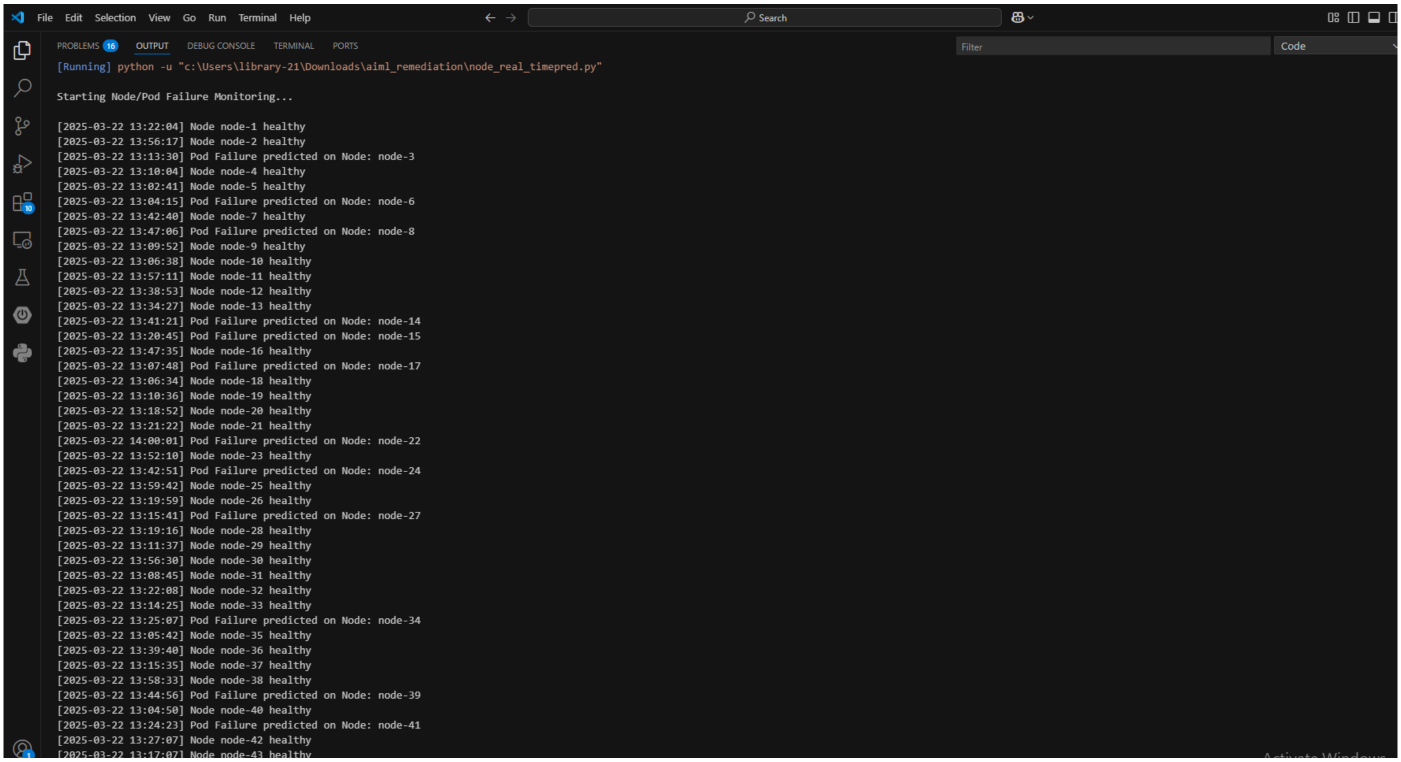
    - The `restart_failed_pods` function takes a **node name** as input, finds all pods running on that node, and checks their status.
    - If a pod is found to be **running** (but not in the `kube-system` namespace), the pod is deleted (restarted) using the Kubernetes API.

4. **Real-time Prediction and Action**:

    - The script runs in a **real-time prediction loop**, processing each row of the dataset sequentially.
    - For each row:
        - The feature columns (`CPU_Usage`, `Memory_Usage`, `Disk_Usage`, `Network_Usage`) are extracted to form the input data for the model.
        - The model then **predicts** whether a **pod failure** is likely (output 1) or not (output 0).
    - If a pod failure is predicted:
        - The **pod restart function** is triggered to **restart the pods** on the affected node.
        - A message indicating the **predicted failure** is printed with the timestamp and node details.
    - If no failure is predicted:
        - A message is printed indicating the node is **healthy**.

5. **Real-Time Simulation**:
    - The script includes a `time.sleep(0.2)` to simulate a real-time delay between each prediction, making it behave like a monitoring system that evaluates the system at regular intervals.

OUTPUT :



The script monitors the health of nodes and pods in a Kubernetes cluster by using a **pre-trained model** to predict failures. It loads data from a **CSV file** that contains system resource usage (CPU, Memory, Disk, and Network usage). Based on these inputs, the model predicts whether a pod failure will occur.

- If a **failure is predicted**, the script **restarts the affected pods** by interacting with the Kubernetes API.
- If no failure is predicted, it prints that the node is healthy.

The system simulates a real-time environment by processing the data in intervals and taking automated actions based on predictions.

Service Disruptions

1. **Load the model**:
   The trained Isolation Forest model (`service_iso_forest.pkl`) is loaded to predict service disruptions.

2. **Preprocess the data**:
   The system failure logs are cleaned and prepared by:

   - Converting percentage columns (like CPU, Memory, Disk usage) to float.

- Encoding categorical columns (such as `Network_Usage`, `Pod_Status`, `K8s_Event_Log`, etc.) using `LabelEncoder`.

3. **Kubernetes interaction**:
   The script connects to the Kubernetes API using the `kubeconfig` file and retrieves pod information.

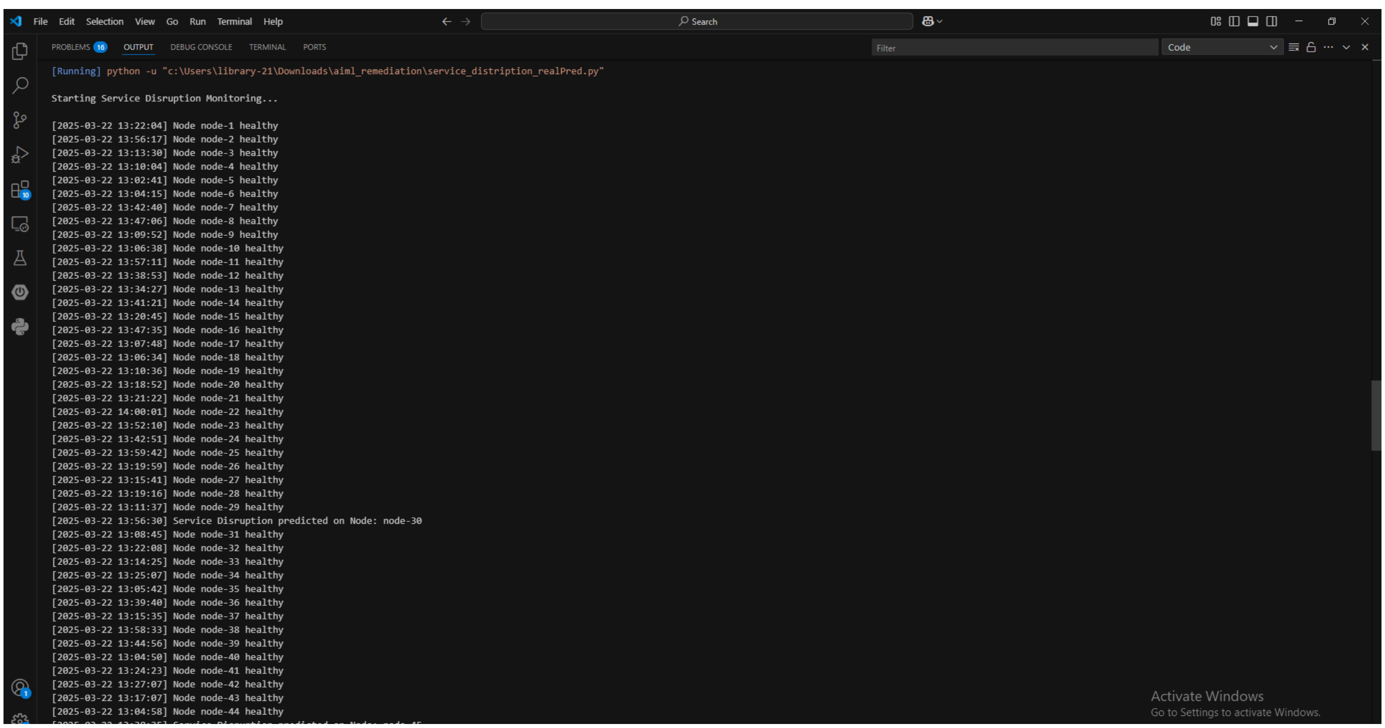
4. **Predict anomalies**:
   The model predicts whether there is a **service disruption** based on the input data. If an anomaly is detected (represented by `-1`), the script takes the following actions:

   - **Automated remediation**: The script restarts the affected pods on the node to address the disruption.
   - **Recommended actions**: It also provides a set of suggested actions, like checking system logs and optimizing network configurations.

5. **Real-time monitoring**:

   The script processes the data in real-time, simulating delays between each prediction and providing continuous monitoring of node health and automated actions.

OUTPUT :



This script monitors **service disruptions** in a Kubernetes cluster by using a **pre-trained Isolation Forest model**. It processes system failure logs (like CPU, memory, disk, and network usage), detects anomalies (service disruptions), and triggers automated actions such as **restarting affected pods**.

Additionally, it provides **recommended actions** for further investigation, like checking system logs and optimizing network configurations.

The script continuously checks for disruptions in real-time, ensuring the system remains healthy with minimal downtime.