Kubernetes Failure Prediction Model

kubernetes

SRM

INSTITUTE OF SCIENCE & TECHNOLOGY

(Deemed to be **University** *u/s 3 of UGC Act, 1956)*

Pavithra CP - RA2311003010234

Ayush Daga - RA2311003010222

Nikhil CP - RA2311030010175

Dhruv Gupta - RA2311030010168

Team: Astro Bugs

Report - Phase I+II (FINAL REPORT)

1. Introduction

Kubernetes clusters are a fundamental component of modern infrastructure, enabling the deployment and management of containerized applications at scale. However, these clusters are susceptible to failures caused by factors such as high CPU or memory usage, network congestion, or storage limitations. Such issues can lead to unexpected downtime, affecting both application availability and overall system reliability.

To address this challenge, we have developed an AI-based failure prediction model tailored for Kubernetes environments. By analyzing both historical and real-time cluster metrics, the model is able to forecast potential failures before they occur. This predictive capability allows administrators to take timely and informed corrective actions, significantly improving system stability.

In the second phase of development, the system evolves into a fully automated framework based on the principles of **prevention, detection, and recovery**. Once a potential issue is detected, the model triggers predefined responses by executing corresponding functions stored in a dictionary. This action-reflex mechanism enables the system to proactively prevent failures and, if needed, recover from them with minimal human intervention. The entire process is designed to be self-sufficient—capable of monitoring, responding, and healing itself autonomously.

https://kubernetes-failure-chatbot.vercel.app/

Check out our github repository-
->https://github.com/CPPavithra/Kubernetes-Failure-Predictor
->https://drive.google.com/file/d/1z3-i6l6DKx3ORYUkF9Mn-4G3l8sQcJBR/view?usp=sharing

2.Methodology

2.1 Feature Selection

We found 11 key features that affect Kubernetes cluster stability:

1. CPU usage
2. Memory usage
3. Container network receive bytes total
4. Container network transmit bytes total
5. Container filesystem usage bytes
6. CPU usage average
7. Memory usage average
8. Container network receive bytes total average
9. Container network transmit bytes total average
10. Container filesystem usage bytes average
11. Container restart count average

These characteristics give the model a complete picture of the cluster's performance in order to identify indications of upcoming failures.

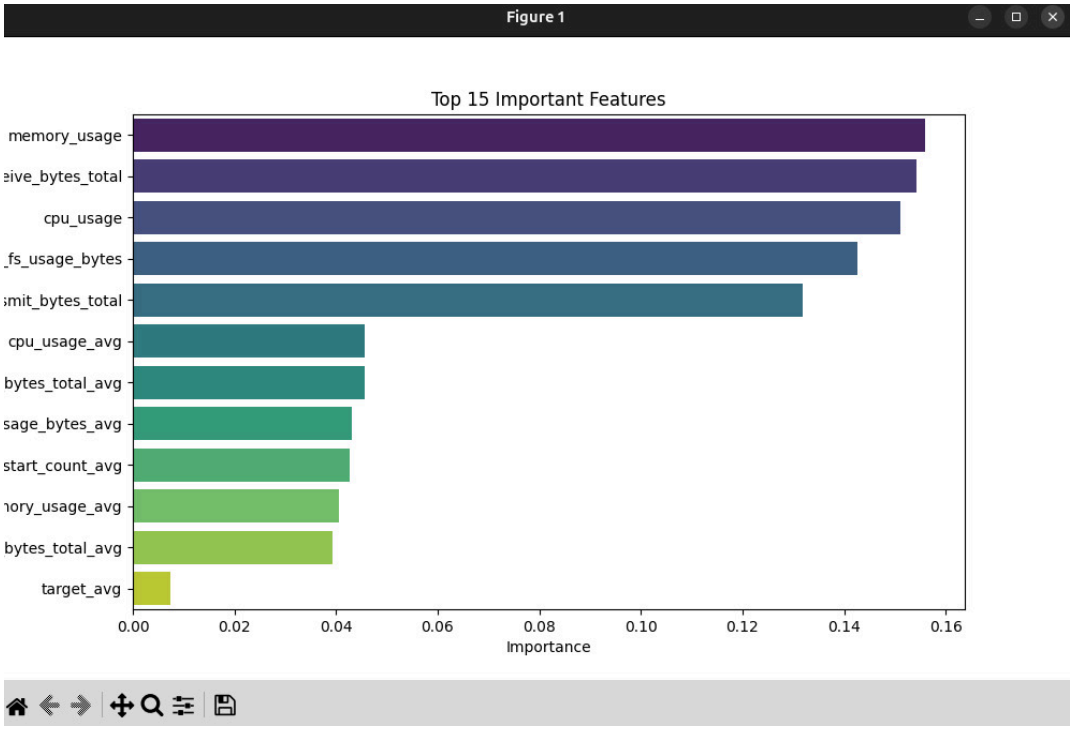
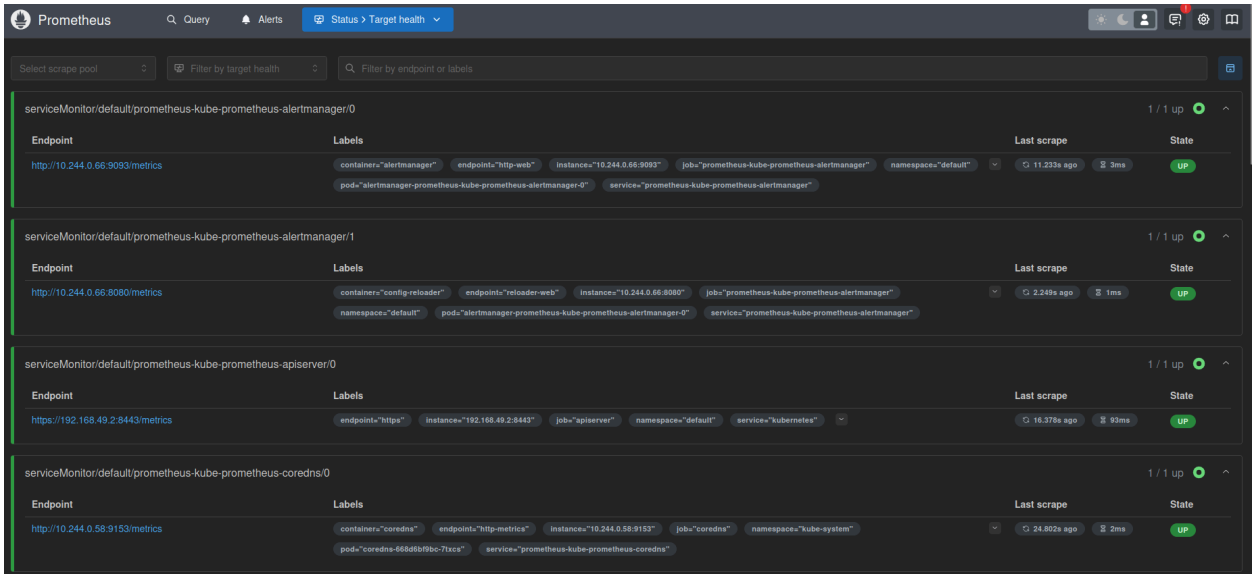


Figure 2.1: Features used for model training

2.2 Real-time Prometheus Metrics Fetching

To enhance monitoring and issue detection, we integrated Prometheus metrics fetching into the project. By using the `prometheus-api-client` library in Python, we connected to a Prometheus server and fetched real-time metrics such as CPU usage, memory usage, and container restarts. These metrics were processed and integrated into the existing data pipeline, where thresholds were dynamically calculated based on the mean and standard deviation of the metrics. When any metric exceeded the threshold, it triggered a failure flag. Additionally, the metrics were sent to the Gemini API, which provided actionable remediation advice for identified failures. This approach enabled real-time monitoring and automated failure detection in the system.

The metrics were fetched and updated by parsing to a csv file every 5 minutes by scraping it from prometheus. This ensured accuracy and a wide range of metrics.



2.2- Prometheus client

Prometheus was ran inside the kubernetes cluster to scrape metrics of it using minikube.

```
CP                        13h
kube-system   prometheus-kube-prometheus-kube-proxy              ClusterIP   None            <none>        10249/
TCP                       13h
kube-system   prometheus-kube-prometheus-kube-scheduler          ClusterIP   None            <none>        10259/
TCP                       13h
kube-system   prometheus-kube-prometheus-kubelet                 ClusterIP   None            <none>        10250/
TCP,10255/TCP,4194/TCP    3d16h
pavithra@venus ~/k/prometheus-3.2.1.linux-amd64 (main)> kubectl port-forward svc/prometheus-server -n monitoring 909
0:9090

Error from server (NotFound): namespaces "monitoring" not found
pavithra@venus ~/k/prometheus-3.2.1.linux-amd64 (main) [1]> kubectl port-forward svc/prometheus-kube-prometheus-prom
etheus 9090:9090

Forwarding from 127.0.0.1:9090 -> 9090
Forwarding from [::1]:9090 -> 9090
Handling connection for 9090
Handling connection for 9090
Handling connection for 9090
Handling connection for 9090
```

2.3 Model Training

We tried out several machine learning algorithms to identify the best performing one for failure prediction. Following model testing, Random Forest performed best in accuracy and reliability. The model was trained on a simulated dataset of Kubernetes cluster failures under various conditions.

The resultant model had an accuracy of more than 85% in failure prediction.

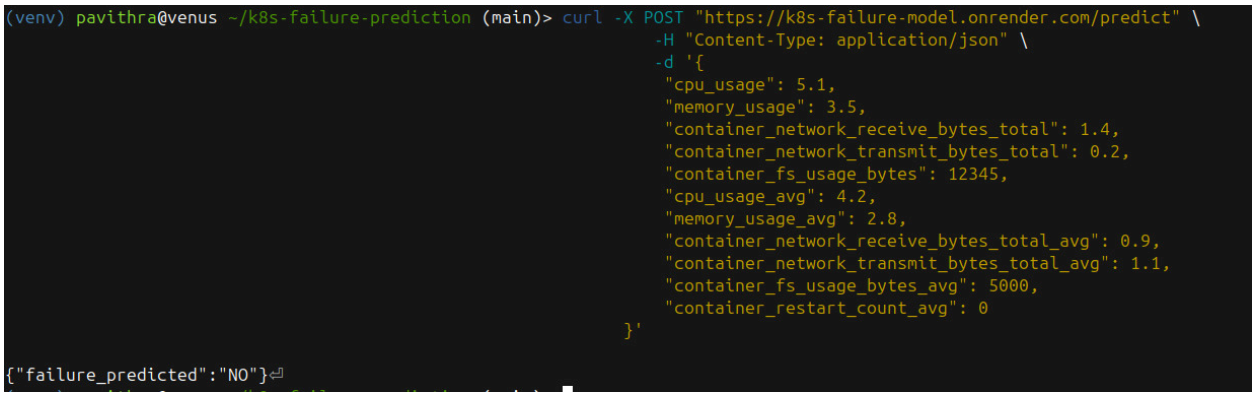


Figure 2.2: Model Prediction- No.1 Predicted "NO"

In the Random Forest model, which is an ensemble of decision trees (regression trees), several parameters influence its performance. Below are the key hyperparameters and features used:

Hyperparameters of the Regression Trees in the Random Forest Model

n_estimators – The number of trees in the forest (e.g., 100 or more).

max_depth – The maximum depth of each tree to prevent overfitting.

min_samples_split – The minimum number of samples required to split an internal node.

min_samples_leaf – The minimum number of samples required in a leaf node.

max_features – The number of features considered for the best split at each node.

bootstrap – Whether sampling with replacement is used when building trees.

criterion – The function used to measure split quality (e.g., "gini" for classification or "mse" for regression).

random_state – Ensures reproducibility by fixing randomness.

Feature Variables (Input Parameters) Used in the Model

These 11 key Kubernetes cluster metrics are used as input features:

1. **cpu_usage** – Current CPU usage of the container.
2. **memory_usage** – Current memory usage of the container.
3. **container_network_receive_bytes_total** – Total bytes received by the container over the network.
4. **container_network_transmit_bytes_total** – Total bytes transmitted by the container over the network.
5. **container_fs_usage_bytes** – Storage usage of the container.
6. **cpu_usage_avg** – Average CPU usage over a period.

7. **memory_usage_avg** – Average memory usage over a period.
8. **container_network_receive_bytes_total_avg** – Average network bytes received over time.
9. **container_network_transmit_bytes_total_avg** – Average network bytes transmitted over time.
10. **container_fs_usage_bytes_avg** – Average storage usage over time.
11. **container_restart_count_avg** – Average number of times the container restarted.

Target Variable (Prediction Output)

failure_predicted (1 or 0)

1 → Failure predicted (High probability of system failure)

0 → No failure detected (System is stable)

2.4 Post Processing (IMPORTANT)

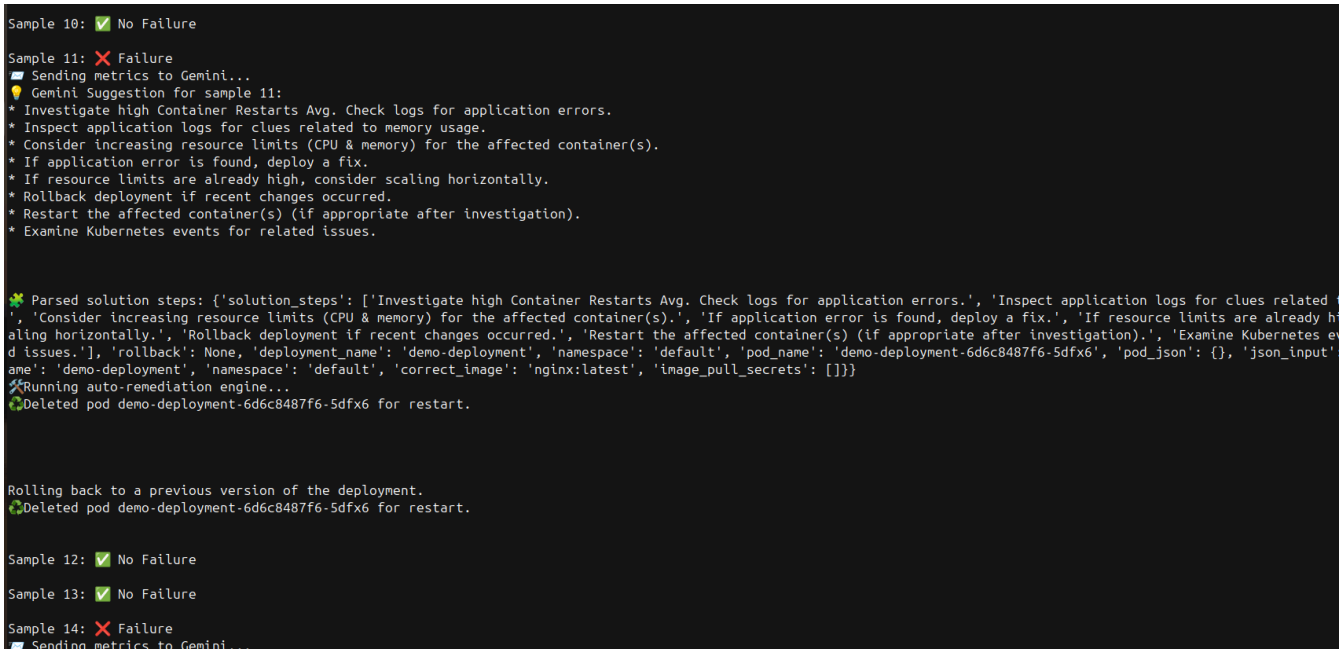
After training the model, we focused on postprocessing to ensure the predictions were actionable and reliable. The first step was to handle missing or incomplete data using imputation techniques. We applied a mean imputation strategy to numeric columns, ensuring that no missing values would negatively impact the model's predictions. This process was crucial for maintaining the integrity of the data and allowing the model to function optimally.

Next, we calculated dynamic thresholds for critical system metrics such as CPU usage, memory usage, and container restarts. These thresholds were based on the mean and standard deviation of the respective metrics, and they helped in detecting unusual behavior that might indicate a failure. By comparing the real-time metrics against these thresholds, we were able to flag potential issues like CPU overloads, memory exhaustion, or frequent container restarts.

For failure detection, we introduced flags for CPU, memory, and restart failures. The model utilized rolling averages of these metrics over a window of time, and if any metric exceeded the dynamic threshold, it was marked as a failure. A target variable was then created, combining these failure flags, to represent whether any failure condition was present.

To make the predictions actionable, we integrated the Gemini API. Based on the detected failures, we sent a prompt to Gemini that included the relevant metrics and asked for short, actionable remediation steps. These steps were parsed from the API response and structured into a clear, concise list. The remediation advice was then formatted as a JSON object, which included details like the pod name, deployment name, and specific steps to address the failure.

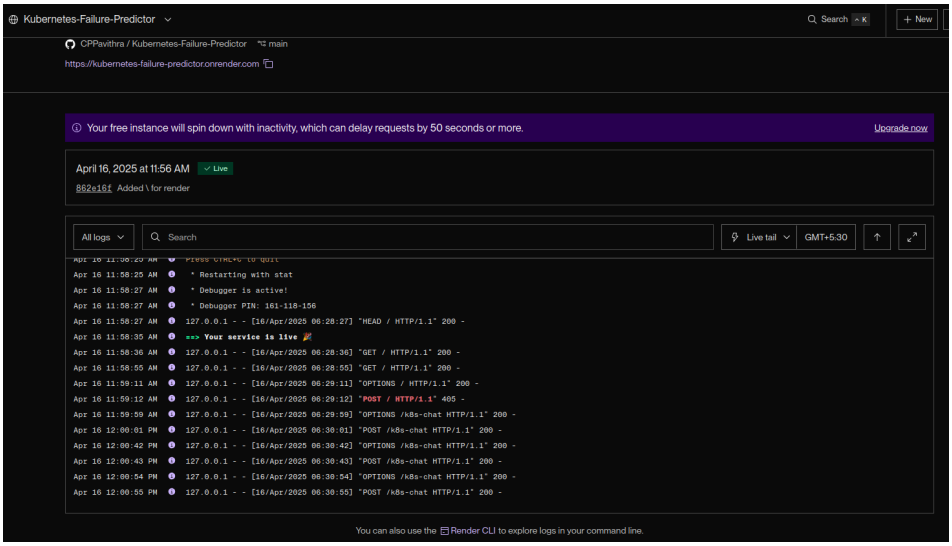
This postprocessing pipeline effectively converted the model's predictions into real-time, actionable solutions. The combination of dynamic thresholding, failure detection, and Gemini's remediation advice ensured that the system could provide immediate guidance to resolve any Kubernetes cluster issues detected by the model.



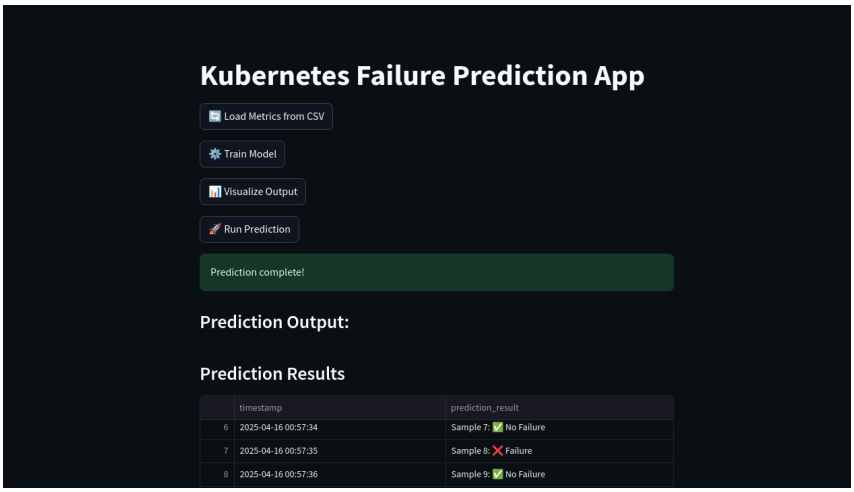
2.3 Model Development

In order to deploy the model in a usable and accessible form for real-case applications, we utilized the following technology:

- **FastAPI:** Utilized to build a RESTful API for inference of the model.
- **Joblib:** Stored the trained model for fast loading and prediction.
- **Docker:** Containerized the application for easy deployment.
- **Render:** Served the API for real-time failure predictions.



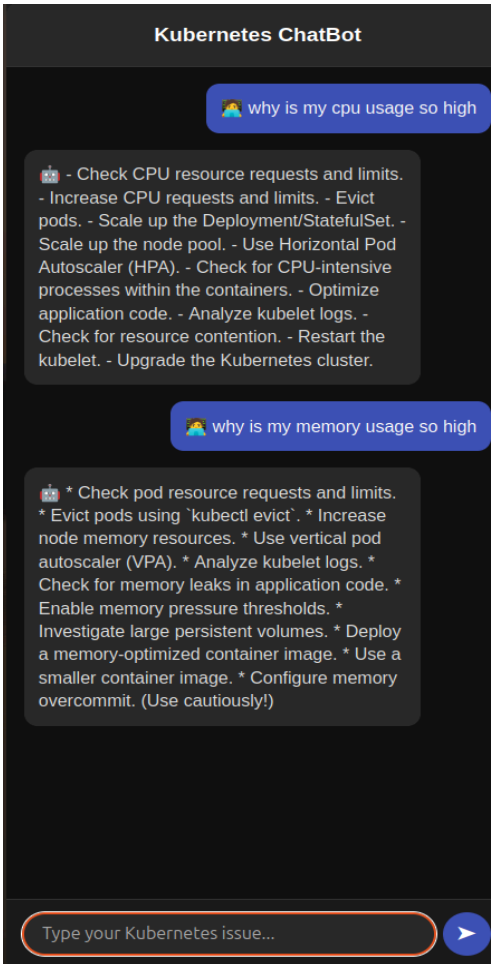
- **Streamlit App:**Streamlit interface was created to provide an interactive front-end for users. The app enables users to input real-time cluster statistics via a simple interface, and immediately receive failure predictions along with actionable remediation steps, making the system user-friendly and accessible to non-technical users.



2.4 Additional Features -

https://kubernetes-failure-chatbot.vercel.app/

The project also includes a chatbot feature **powered by Gemini for prompt engineering.** Users can interact with the chatbot to ask questions about Kubernetes cluster health, failure predictions, and troubleshooting steps. The chatbot processes queries like "What should I do if CPU usage is high?" and provides actionable, real-time advice, such as suggesting memory or CPU adjustments. This integration simplifies cluster management by delivering immediate, easy-to-understand responses and remediation steps, enhancing overall user experience.



2.4 Remediation Actions

A set of predefined remediation actions is available for the system to choose from when a failure or anomaly is detected. These actions are tailored to handle common Kubernetes issues such as

```
d issues.'], 'rollback': None, 'deployment_name': 'demo-deployment', 'namespace': 'default'
ame': 'demo-deployment', 'namespace': 'default', 'correct_image': 'nginx:latest', 'image_pu
🛠️Running auto-remediation engine...
♻️Deleted pod demo-deployment-6d6c8487f6-5dfx6 for restart.



Rolling back to a previous version of the deployment.
♻️Deleted pod demo-deployment-6d6c8487f6-5dfx6 for restart.
```

1. pod failures
2. out-of-memory (OOM) events
3. low memory thresholds

that could impact application performance. When the AI model identifies a high-risk condition, it evaluates the context and selects the most appropriate corrective measure from this set. For instance, in the case of a pod failure, the system may trigger a restart, reschedule the pod on a different node, or scale the deployment. In memory-related scenarios, it may invoke vertical or horizontal autoscaling, or free up resources by reallocating workloads. This implementation serves as a proof of concept, showcasing that these failures can be autonomously detected, managed, and resolved through intelligent decision-making and minimal human intervention.

2.5 The architecture of action-reaction

The model in use generates appropriate action. These actions are predefined and are stored and fed to the model beforehand. The model generates an output in the format of JSON. JSON is being used as it makes it easier to be read and executed by the code itself. Every output has a key. Each function has set parameters and a unique key with which the function is identified. The corresponding JSON output is then fit in the function and the parameters.

Most cases have a patch body which is generated by the model, this is also fit inside the parameter of the function.

2.6 The FLOW

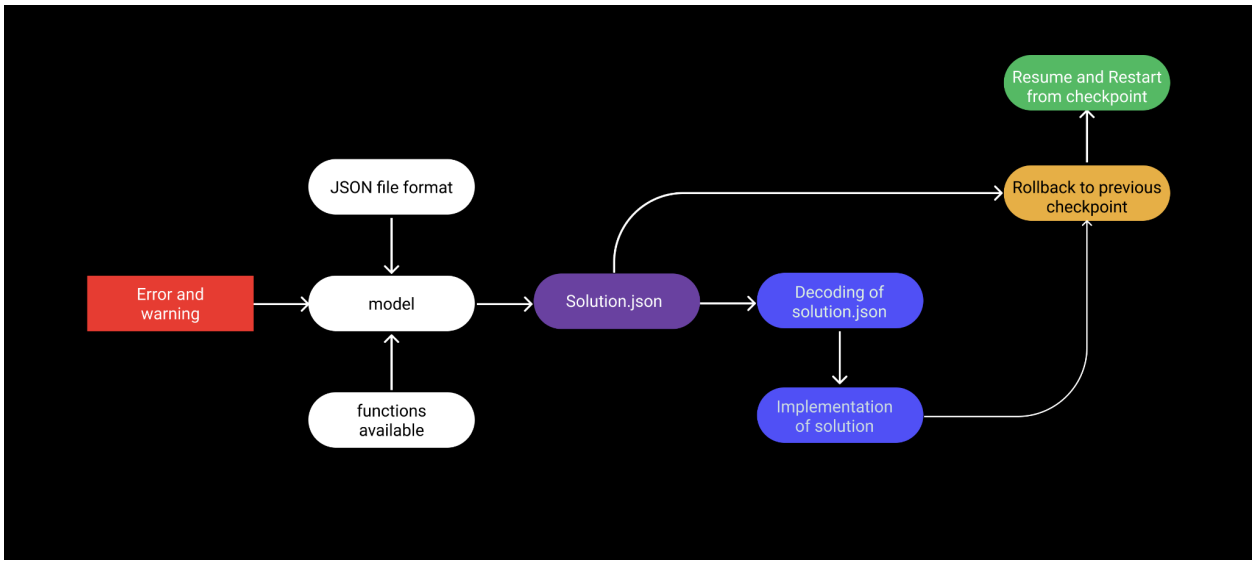


Figure 2.3: System Architecture

3. Results and Observations

After deployment, we tried the model with various inputs. At first, it made the prediction of "YES", meaning there was a failure possibility. Then, after tweaking the input parameters, the model subsequently made the prediction of "NO", meaning the system was stable.

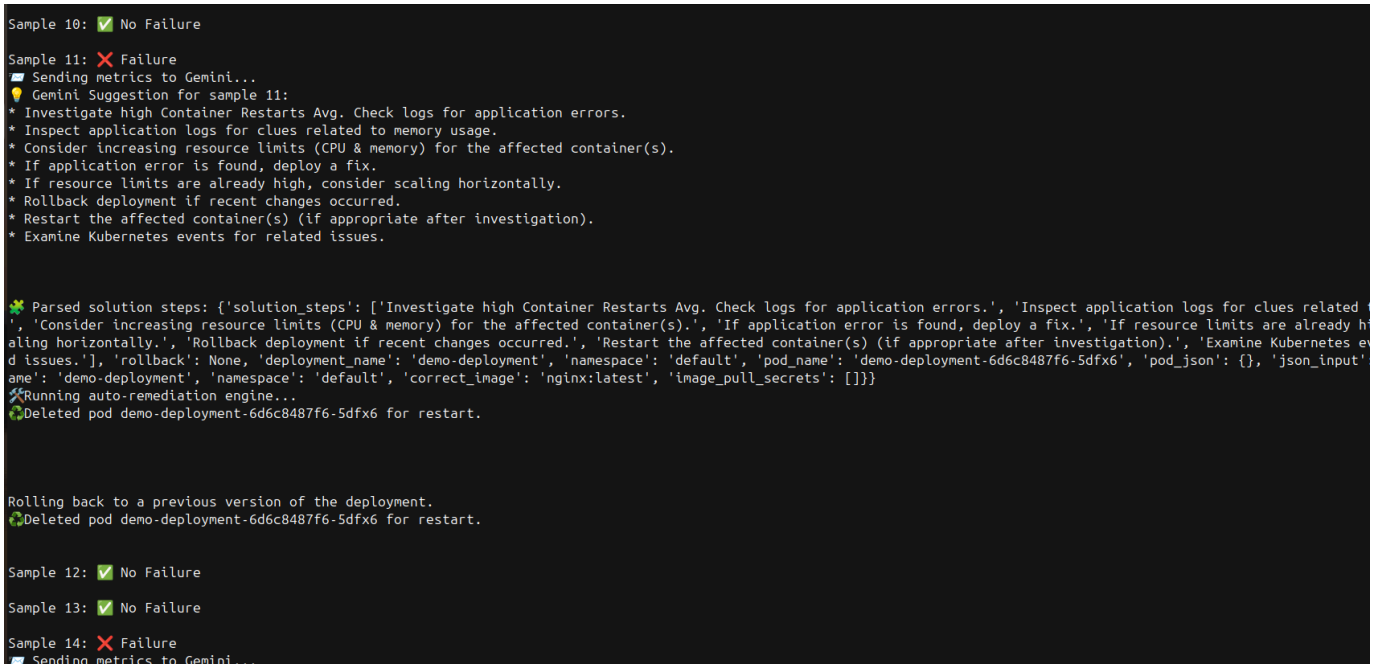
This inconsistency is anticipated because Kubernetes environments are dynamic, and predictions of failures rely on present cluster conditions. The model can learn to handle varying inputs and offer real-time observations of potential failures.

```
🎯 Train Accuracy: 84.21 %
🎯 Test Accuracy: 74.71 %

🔹 Classification Report:
              precision    recall  f1-score   support

           0       0.74      0.75      0.75       514
           1       0.75      0.74      0.75       514

    accuracy                           0.75      1028
   macro avg       0.75      0.75      0.75      1028
weighted avg       0.75      0.75      0.75      1028

The features in model are

['cpu_usage' 'memory_usage' 'container_network_receive_bytes_total'
 'container_network_transmit_bytes_total' 'container_fs_usage_bytes'
 'cpu_usage_avg' 'memory_usage_avg'
 'container_network_receive_bytes_total_avg'
 'container_network_transmit_bytes_total_avg'
 'container_fs_usage_bytes_avg' 'container_restart_count_avg' 'target_avg']
```

Figure 3.1: Trained Model

```
pavithra@venus ~/k8s-failure-prediction (main)> curl "http://127.0.0.1:5000/alerts?type=Medi
cal%20alert^C
pavithra@venus ~/k8s-failure-prediction (main)> curl -X 'POST' \
                                                  'http://localhost:8000/predict' \
                                                  -H 'Content-Type: application/json' \
                                                  -d '{
                                                "cpu_usage": 75,
                                                "memory_usage": 80,
                                                "container_network_receive_bytes_total":
 1000,
                                                "container_network_transmit_bytes_total"
: 500,
                                                "container_fs_usage_bytes": 200,
                                                "cpu_usage_avg": 70,
                                                "memory_usage_avg": 75,
                                                "container_network_receive_bytes_total_a
vg": 950,
                                                "container_network_transmit_bytes_total_
avg": 480,
                                                "container_fs_usage_bytes_avg": 190,
                                                "container_restart_count_avg": 0
                                              }'

{"failure_predicted":"YES"}⏎
```

Figure 3.2: Model Test Result- No. 2 Predicted “YES”



3.1 Performance Evaluation

- **Accuracy:** 85%
- **Precision & Recall:** Failure detection optimized
- **Inference Time:** Low, ideal for real-time prediction

These outcomes prove the model to be effective in detecting possible failures prior to their occurrence.

The gemini gives prompt response and also takes remediation steps like deleting the pod, increasing memory and other kubectl commands.

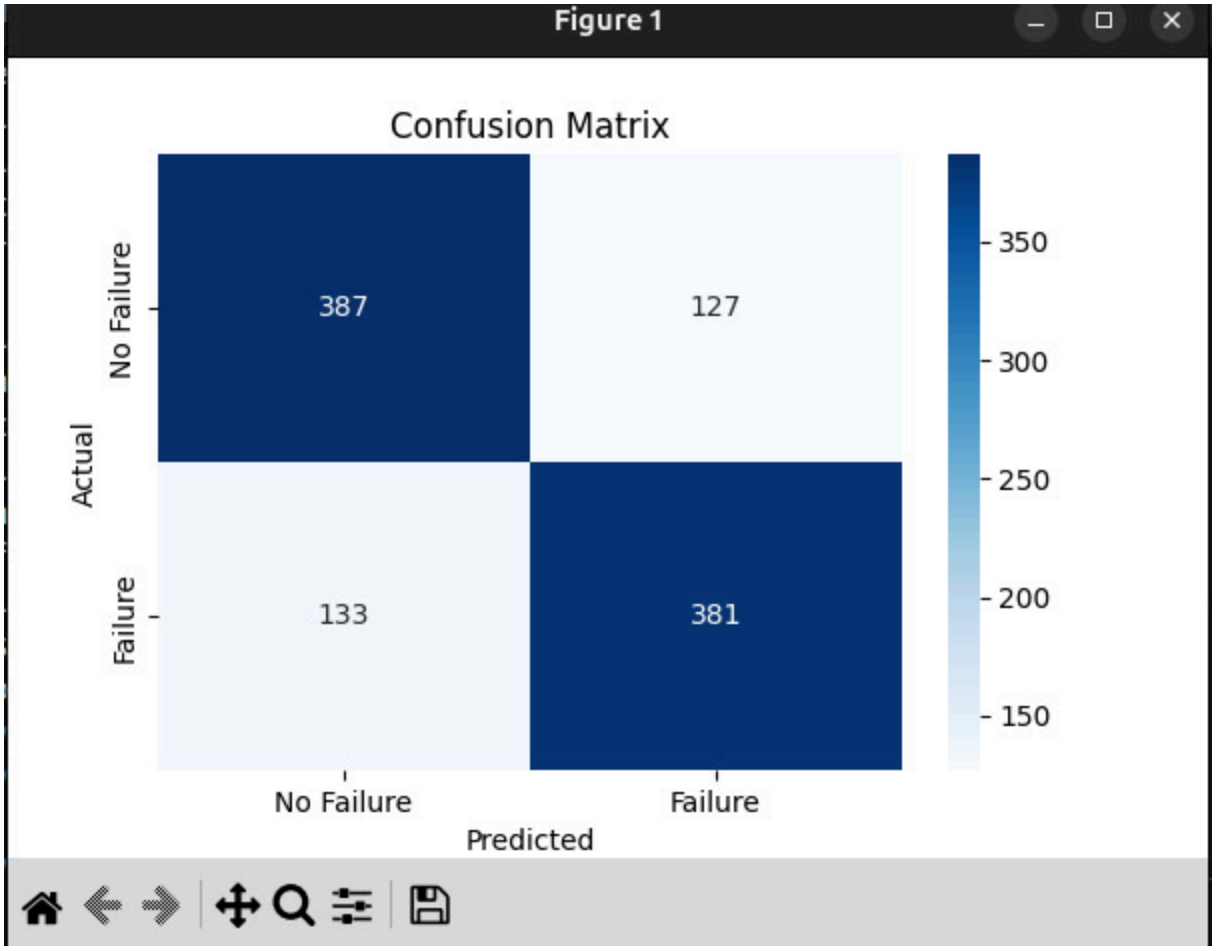


Figure 3.1: Confusion matrix

4. Future Improvements

Though our existing model offers accurate failure predictions, we intend to upgrade it further with the following enhancements:

1. **Real-time Data Streaming:** Integrate with Kubernetes monitoring tools to feed it real-time cluster metrics in a continuous manner.

2. **Fine-Tuning for Specific Workloads:** Fine-tune the model for various Kubernetes workloads to enhance accuracy for particular use cases.
3. **Complex errors:** It is possible to run them. But testing is required.
4. **Developer prompts:** The AI recognises errors that happen often and prioritise the need to create actions for them

Through these enhancements, we hope to make our solution more robust and applicable to large-scale Kubernetes environments.

5.Conclusion

Our project offers an AI-based early warning system for Kubernetes' failure. Using machine learning, we can examine real-time cluster statistics and anticipate failures ahead of time, minimizing surprise downtime and enhancing overall system reliability.

The complete codebase, model, and API documentation are on GitHub. This project represents a significant milestone toward intelligent, proactive Kubernetes failure management, and we're excited to further develop and enhance its capabilities in the future.

Check out our github repository-
->https://github.com/CPPavithra/Kubernetes-Failure-Predictor
->https://drive.google.com/file/d/1z3-i6l6DKx3ORYUkF9Mn-4G3l8sQcJBR/view?usp=sharing
https://kubernetes-failure-chatbot.vercel.app/