Kubernetes Failure Prediction using ML, Prometheus, and Gemini Chatbot

Smart Infrastructure Monitoring and Troubleshooting

Team- Astro Bugs

Pavithra CP
Nikhil CP
Ayush Daga
Dhruv Gupta

Objective

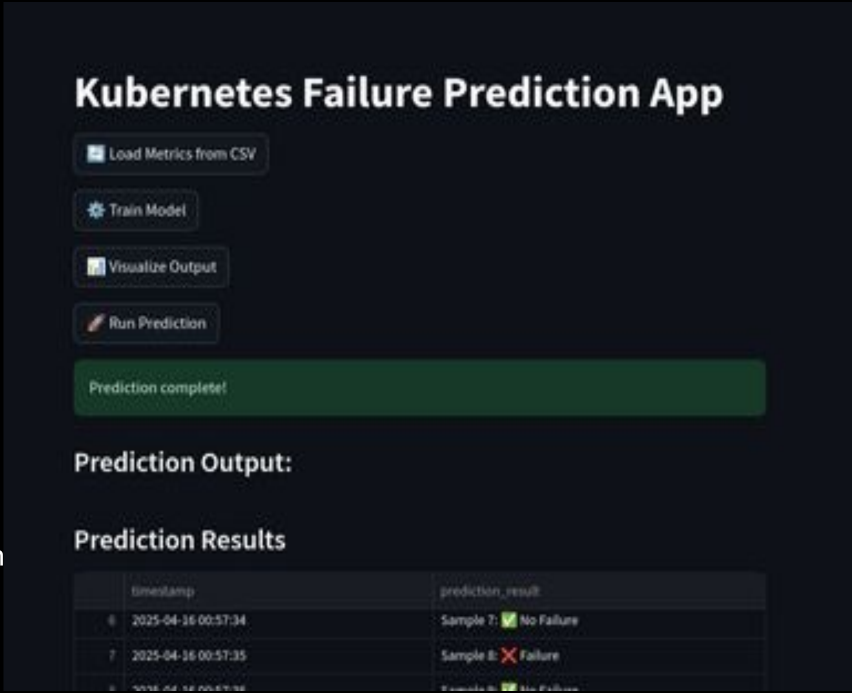
1. **Fetch real-time Prometheus metrics from a Kubernetes cluster**
 → Collect live telemetry data from Kubernetes using Prometheus for analysis.

2. **Train a machine learning model to predict failures**
 → Develop an ML model that anticipates Kubernetes cluster failures based on metrics.

3. **Provide automated, intelligent remediation advice using Gemini API**
 → Use Gemini API to generate smart, context-aware recovery suggestions.

4. **Deploy the system using FastAPI + Docker + Render**
 → Package and serve the solution as a containerized FastAPI app on Render.

5. **Enable a chatbot for real-time troubleshooting queries**
 → Integrate a chatbot to assist users with live issue diagnosis and solutions.



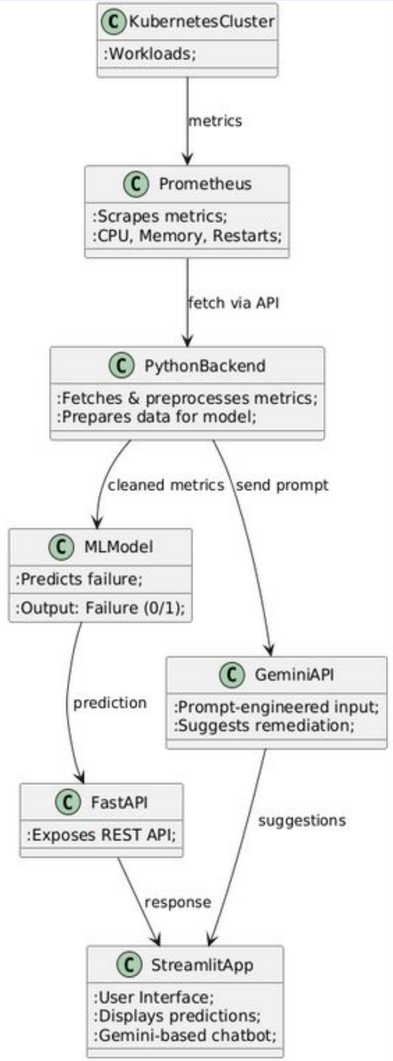
Kubernetes Failure Prediction App

Prediction complete!

Prediction Output:

Prediction Results

	timestamp	prediction_result
6	2025-04-16 00:57:34	Sample 7: ✅ No Failure
7	2025-04-16 00:57:35	Sample 8: ❌ Failure

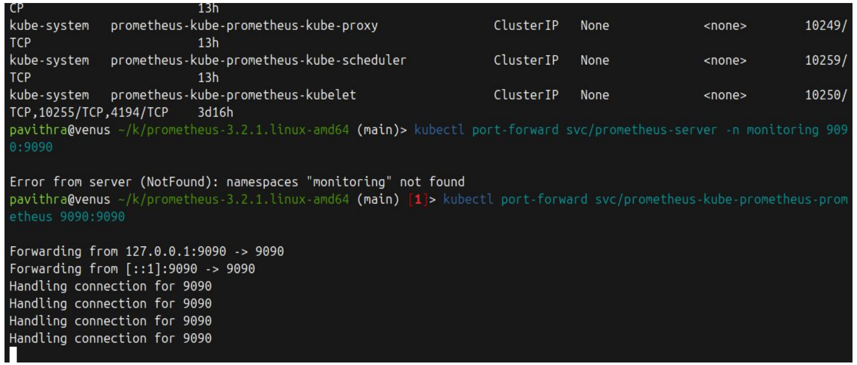


System Architecture

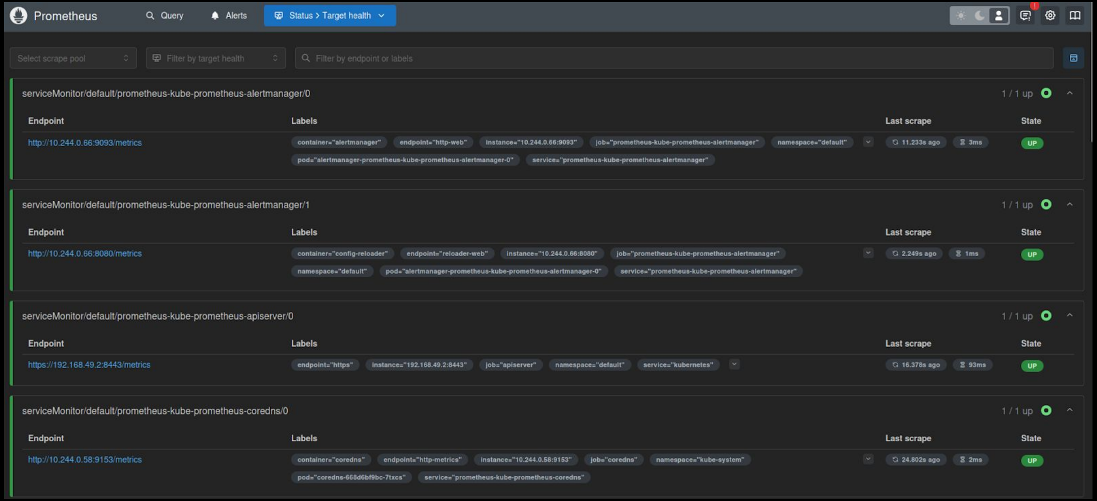
- Kubernetes Cluster running workloads
- Prometheus scrapes pod/deployment metrics
- Python backend fetches, preprocesses metrics
- ML model predicts failure likelihood
- Gemini suggests remediation
- FastAPI exposes REST API
- Streamlit UI + Gemini chatbot for user queries

Data Collection

- **Prometheus** used to monitor CPU, memory, restarts
- Custom Python script fetches metrics using Prometheus API
- Metrics stored in CSV format for processing
- Rolling averages and thresholds calculated dynamically
- Labelled target for failure prediction



```
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
```



serviceMonitor/default/prometheus-kube-prometheus-alertmanager/0 — 1 / 1 up

Endpoint	Labels	Last scrape	State
http://10.244.0.66:9093/metrics	container="alertmanager" endpoint="http-web" instance="10.244.0.66:9093" job="prometheus-kube-prometheus-alertmanager" namespace="default" pod="alertmanager-prometheus-kube-prometheus-alertmanager-0" service="prometheus-kube-prometheus-alertmanager"	11.233s ago 3ms	UP

serviceMonitor/default/prometheus-kube-prometheus-alertmanager/1 — 1 / 1 up

Endpoint	Labels	Last scrape	State
http://10.244.0.66:8080/metrics	container="config-reloader" endpoint="reloader-web" instance="10.244.0.66:8080" job="prometheus-kube-prometheus-alertmanager" namespace="default" pod="alertmanager-prometheus-kube-prometheus-alertmanager-0" service="prometheus-kube-prometheus-alertmanager"	2.249s ago 1ms	UP

serviceMonitor/default/prometheus-kube-prometheus-apiserver/0 — 1 / 1 up

Endpoint	Labels	Last scrape	State
https://192.168.49.2:8443/metrics	endpoint="https" instance="192.168.49.2:8443" job="apiserver" namespace="default" service="kubernetes"	16.378s ago 93ms	UP

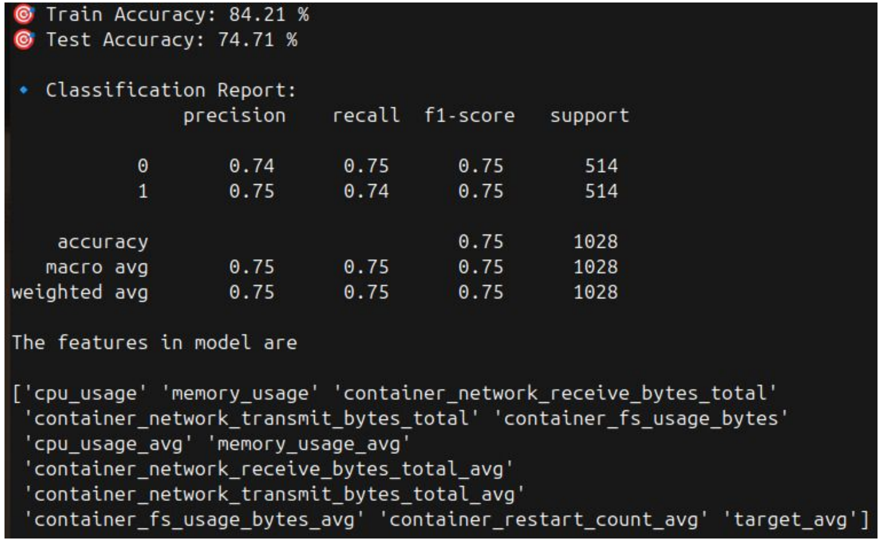
serviceMonitor/default/prometheus-kube-prometheus-coredns/0 — 1 / 1 up

Endpoint	Labels	Last scrape	State
http://10.244.0.58:9153/metrics	container="coredns" endpoint="http-metrics" instance="10.244.0.58:9153" job="coredns" namespace="kube-system" pod="coredns-668d6bf9bc-7txcs" service="prometheus-kube-prometheus-coredns"	24.802s ago 2ms	UP

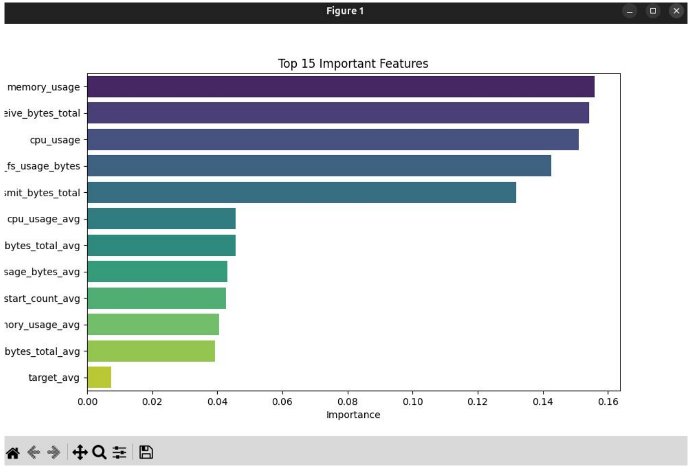
Prometheus was ran inside the kubernetes cluster to scrape metrics of it using minikube. The metrics were fetched and updated by parsing to a csv file every 5 minutes by scraping it from prometheus. This ensured accuracy and a wide range of metrics.

Model Training

1. Preprocessed data using pandas and NumPy
2. Features imputed using `SimpleImputer`
3. Model trained to classify failures (target = 1 or 0)
4. Saved using `joblib` for fast inference
5. Postprocessing includes rolling means, dynamic thresholds, and binary labels for failure events



```
🎯 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']
```



Note that;
The model's train and test accuracy will differ with the data collected in the csv as it is being scraped in real time. HOWEVER, it is ensured that it is always accurate and provides NO overfitting.

Post-Processing

1. Imputation of Missing Data

- Applied **mean imputation** to numeric columns to handle missing/incomplete data.
- Ensured **data integrity** and consistent model performance.

2. Dynamic Threshold Calculation

- Computed thresholds for key metrics:
  - **CPU usage**
  - **Memory usage**
  - **Container restarts**
- Based on **mean ± standard deviation** of each metric.
- Helped detect anomalies and potential failure conditions.

3. Failure Flag Generation

- Created **flags** for:
  - CPU failure
  - Memory failure
  - Restart failure
- Used **rolling averages** over a time window for better temporal detection.
- Triggered flags when metrics exceeded dynamic thresholds.

4. Target Variable Creation

- Combined all failure flags into a **single target variable**.
- Indicated if **any failure condition** was present.

5. Gemini API Integration for Remediation

- Upon detecting failures, sent a **prompt** to Gemini API.
- Included relevant metrics and requested **short, actionable solutions**.
- Parsed response into a **structured list of remediation steps**.

6. JSON Output Formatting

- Generated a **JSON object** with:
  - Pod name
  - Deployment name
  - Gemini-generated remediation steps

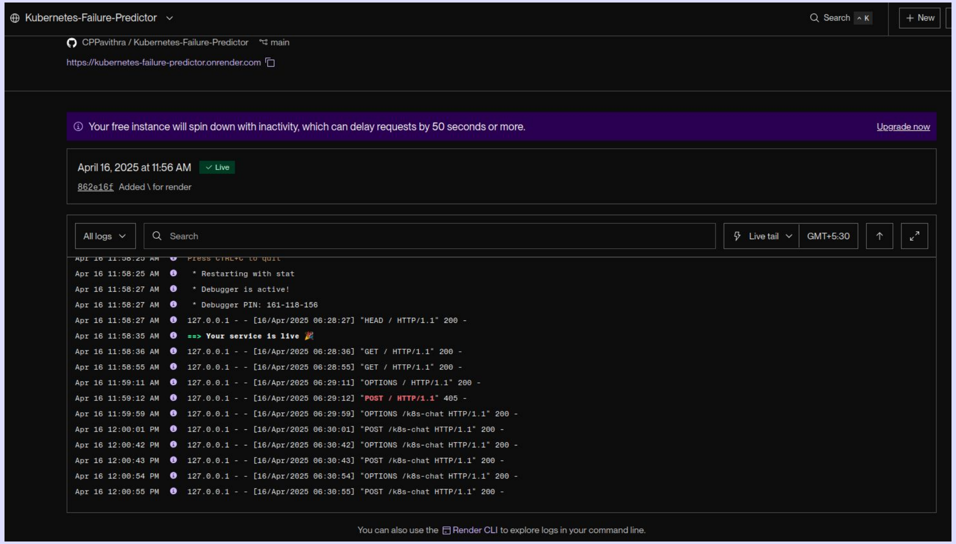
Final Outcome

- Transformed raw predictions into **real-time, actionable insights**.
- Enabled automated, intelligent **failure detection and guidance** for Kubernetes clusters.

Model Deployment

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.


Kubernetes-Failure-Predictor
Search ^K
+ New
CPPavithra / Kubernetes-Failure-Predictor
main
https://kubernetes-failure-predictor.onrender.com
Upgrade now
Live
All logs
Search
Live tail
GMT+5:30

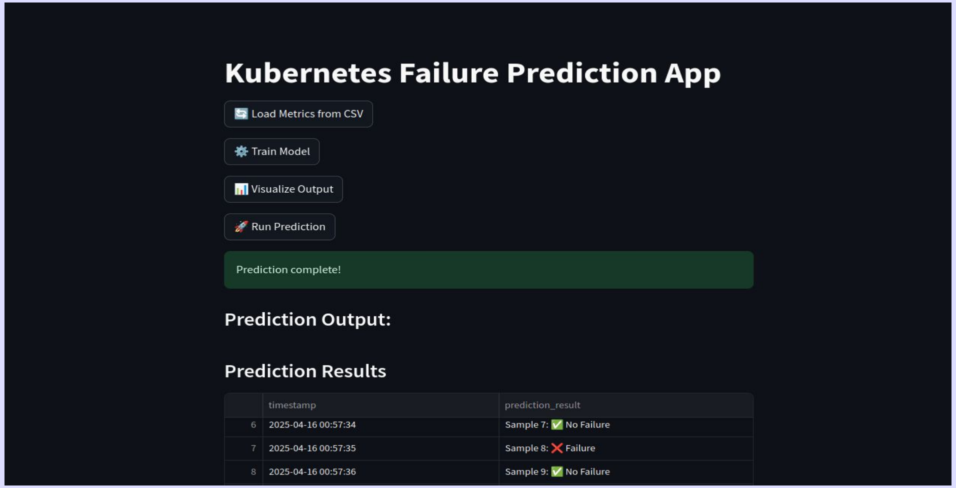
Your free instance will spin down with inactivity, which can delay requests by 50 seconds or more.

April 16, 2025 at 11:56 AM

862e16f Added \ for render

```
Apr 16 11:58:25 AM   Press CTRL+C to quit
Apr 16 11:58:25 AM    * Restarting with stat
Apr 16 11:58:27 AM    * Debugger is active!
Apr 16 11:58:27 AM    * Debugger PIN: 161-118-156
Apr 16 11:58:27 AM   127.0.0.1 - - [16/Apr/2025 06:28:27] "HEAD / HTTP/1.1" 200 -
Apr 16 11:58:35 AM   ==> Your service is live 🎉
Apr 16 11:58:36 AM   127.0.0.1 - - [16/Apr/2025 06:28:36] "GET / HTTP/1.1" 200 -
Apr 16 11:58:55 AM   127.0.0.1 - - [16/Apr/2025 06:28:55] "GET / HTTP/1.1" 200 -
Apr 16 11:59:11 AM   127.0.0.1 - - [16/Apr/2025 06:29:11] "OPTIONS / HTTP/1.1" 200 -
Apr 16 11:59:12 AM   127.0.0.1 - - [16/Apr/2025 06:29:12] "POST / HTTP/1.1" 405 -
Apr 16 11:59:59 AM   127.0.0.1 - - [16/Apr/2025 06:29:59] "OPTIONS /k8s-chat HTTP/1.1" 200 -
Apr 16 12:00:01 PM   127.0.0.1 - - [16/Apr/2025 06:30:01] "POST /k8s-chat HTTP/1.1" 200 -
Apr 16 12:00:42 PM   127.0.0.1 - - [16/Apr/2025 06:30:42] "OPTIONS /k8s-chat HTTP/1.1" 200 -
Apr 16 12:00:43 PM   127.0.0.1 - - [16/Apr/2025 06:30:43] "POST /k8s-chat HTTP/1.1" 200 -
Apr 16 12:00:54 PM   127.0.0.1 - - [16/Apr/2025 06:30:54] "OPTIONS /k8s-chat HTTP/1.1" 200 -
Apr 16 12:00:55 PM   127.0.0.1 - - [16/Apr/2025 06:30:55] "POST /k8s-chat HTTP/1.1" 200 -
```

You can also use the Render CLI to explore logs in your command line.



Kubernetes Failure Prediction App

Prediction complete!

Prediction Output:

Prediction Results

	timestamp	prediction_result
6	2025-04-16 00:57:34	Sample 7: ✅ No Failure
7	2025-04-16 00:57:35	Sample 8: ❌ Failure
8	2025-04-16 00:57:36	Sample 9: ✅ No Failure

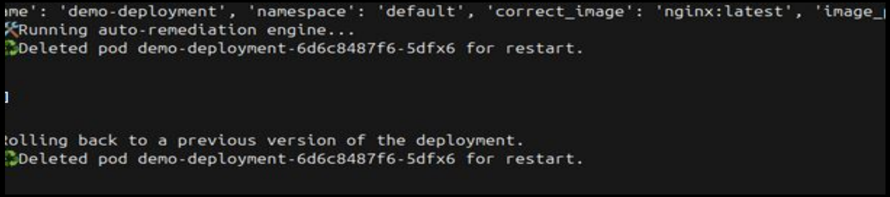
AI Integration for Remediation

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

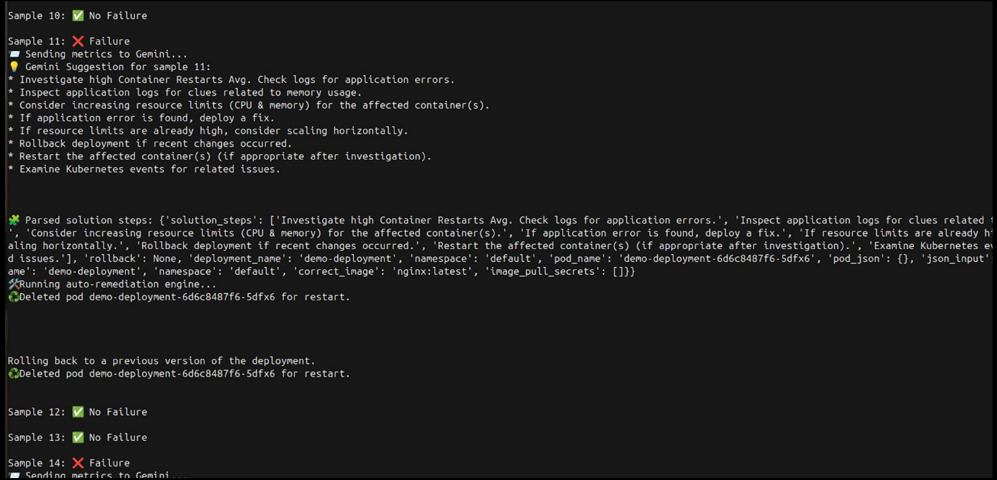
1. pod failures
2. out-of-memory (OOM) events
3. low memory thresholds

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.

- Gemini API used for **prompt-engineered remediation advice**
- Prompts include real-time metrics (e.g., CPU, memory, restarts)
- Gemini returns actionable bullet points for fixing the issue
- Advice parsed into JSON and shown on Streamlit



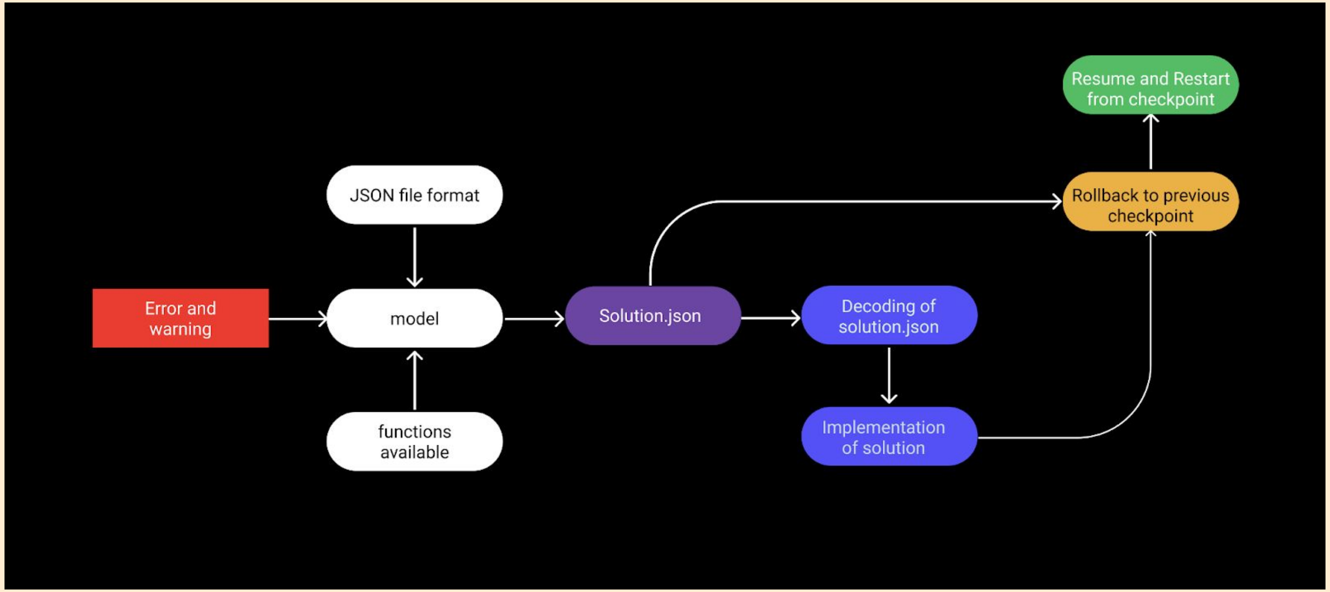
```
me': 'demo-deployment', 'namespace': 'default', 'correct_image': 'nginx:latest', 'image_
🛠Running auto-remediation engine...
♻Deleted pod demo-deployment-6d6c8487f6-5dfx6 for restart.

]

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



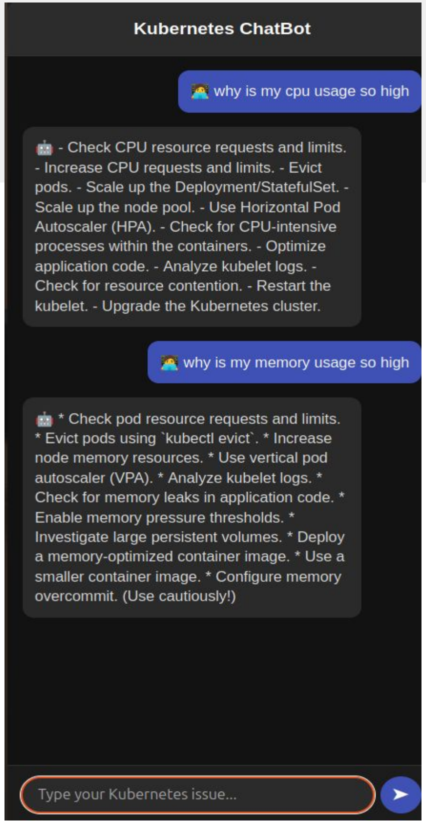
```
Sample 10: ✅ No Failure

Sample 11: ❌ Failure
📨 Sending metrics to Gemini...
💡 Gemini Suggestion for sample 11:
* Investigate high Container Restarts Avg. Check logs for application errors.
* Inspect application logs for clues related to memory usage.
* Consider increasing resource limits (CPU & memory) for the affected container(s).
* If application error is found, deploy a fix.
* If resource limits are already high, consider scaling horizontally.
* Rollback deployment if recent changes occurred.
* Restart the affected container(s) (if appropriate after investigation).
* Examine Kubernetes events for related issues.



🧩 Parsed solution steps: {'solution_steps': ['Investigate high Container Restarts Avg. Check logs for application errors.', 'Inspect application logs for clues related t
', 'Consider increasing resource limits (CPU & memory) for the affected container(s).', 'If application error is found, deploy a fix.', 'If resource limits are already hi
aling horizontally.', 'Rollback deployment if recent changes occurred.', 'Restart the affected container(s) (if appropriate after investigation).', 'Examine Kubernetes ev
d issues.'], 'rollback': None, 'deployment_name': 'demo-deployment', 'namespace': 'default', 'pod_name': 'demo-deployment-6d6c8487f6-5dfx6', 'pod_json': {}, 'json_input':
ame': 'demo-deployment', 'namespace': 'default', 'correct_image': 'nginx:latest', 'image_pull_secrets': []}}
🛠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.


Sample 12: ✅ No Failure

Sample 13: ✅ No Failure

Sample 14: ❌ Failure
📨 Sending metrics to Gemini...
```

Flow Diagram

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.


Resume and Restart from checkpoint
JSON file format
Rollback to previous checkpoint
Error and warning
model
Solution.json
Decoding of solution.json
functions available
Implementation of solution

Additional Feature- Chatbot



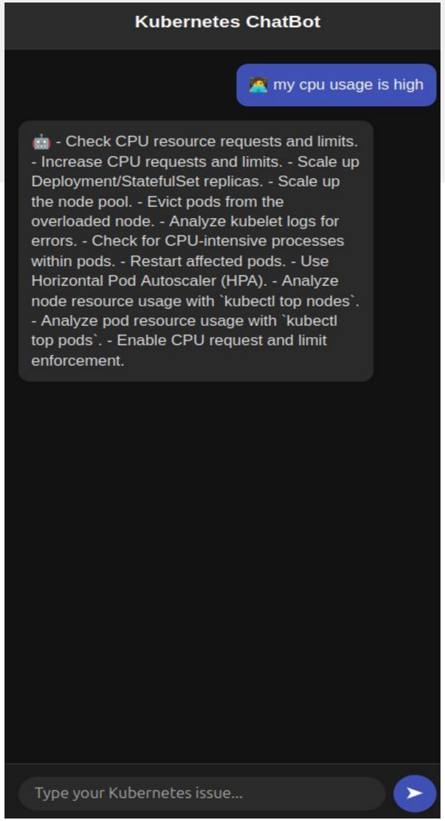
Kubernetes ChatBot

🧑‍💻 why is my cpu usage so high

🤖 - Check CPU resource requests and limits. - Increase CPU requests and limits. - Evict pods. - Scale up the Deployment/StatefulSet. - Scale up the node pool. - Use Horizontal Pod Autoscaler (HPA). - Check for CPU-intensive processes within the containers. - Optimize application code. - Analyze kubelet logs. - Check for resource contention. - Restart the kubelet. - Upgrade the Kubernetes cluster.

🧑‍💻 why is my memory usage so high

🤖 * Check pod resource requests and limits. * Evict pods using `kubectl evict`. * Increase node memory resources. * Use vertical pod autoscaler (VPA). * Analyze kubelet logs. * Check for memory leaks in application code. * Enable memory pressure thresholds. * Investigate large persistent volumes. * Deploy a memory-optimized container image. * Use a smaller container image. * Configure memory overcommit. (Use cautiously!)



Kubernetes ChatBot

🧑‍💻 my cpu usage is high

🤖 - Check CPU resource requests and limits. - Increase CPU requests and limits. - Scale up Deployment/StatefulSet replicas. - Scale up the node pool. - Evict pods from the overloaded node. - Analyze kubelet logs for errors. - Check for CPU-intensive processes within pods. - Restart affected pods. - Use Horizontal Pod Autoscaler (HPA). - Analyze node resource usage with `kubectl top nodes`. - Analyze pod resource usage with `kubectl top pods`. - Enable CPU request and limit enforcement.

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.

Future Scope and Conclusion

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
5. **Reinforcement learning**: for smarter remediation

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

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, while at the same time, providing AI powered remediations.

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.

Thank You.
Our complete project repo with detailed READMe, Explanation video and the project report can be accessed from the links below.

REPORT->

https://docs.google.com/document/d/1R6nR_AweptKE9sJPMdnFxIeO3jDxQfkfBhI2Ld4GCDc/edit?usp=sharing

GITHUB LINK-> https://github.com/CPPavithra/Kubernetes-Failure-Predictor