International Journal of Scientific Research and Engineering Development-– Volume 9 Issue 6, Nov – Dec 2026
Available at www.ijsred.com

RESEARCH ARTICLE OPEN ACCESS

Proactive ML-Based Load Balancing for Cloud Microservices

Prof. Roopa B, Madhusudhan U, Rakesh D, Goutham R, Kiran Kumar
Bangalore Institute of Technology. Bangalore, Karnataka, India

----------**********----------

Abstract:
Cloud-based applications developed using microservices architecture experience continuous variations in workload due to changing user requests. During peak traffic conditions, increased resource utilization can cause server overload, higher response time and reduced application performance. Traditional load-balancing and autoscaling techniques generally follow a reactive approach, where additional resources are allocated only after predefined resource-utilization thresholds are reached. This may result in delayed responses during sudden increases in workload. To address these challenges, this paper presents a Proactive Machine Learning-Based Load Balancing framework for cloud microservices. The proposed system uses a Long Short-Term Memory (LSTM) network to analyse historical workload information and predict future CPU utilization. The prediction process considers five important parameters: current users, CPU utilization, memory usage, requests per second and response time. A rule-based mechanism converts the predicted CPU utilization into recommended backend pod counts to support proactive resource management. The system is developed using Python and Fast API for backend services and React for the frontend interface. A workload simulator generates different traffic conditions, while Prometheus and Grafana provide resource monitoring and visualization. Docker and Kubernetes support containerization and deployment, with Horizontal Pod Autoscaler (HPA) configuration for replica management. The framework integrates workload generation, metric collection, machine learning-based prediction and container orchestration to support improved scalability and resource utilization in cloud-hosted applications.

Keywords: Cloud Computing, Microservices, Load Balancing, Machine Learning, Long Short-Term Memory (LSTM), Predictive Autoscaling, Kubernetes, Prometheus, Grafana, CPU Utilization.

----------**********----------

I. Introduction

Cloud computing has become an important technology for developing and deploying modern applications. Applications developed using microservices architecture consist of multiple independent services that communicate with each other to perform different operations. This architecture provides flexibility, scalability and easier application maintenance.However, cloud-hosted applications frequently experience changes in workload due to variations in user activity. During normal operating conditions, the available computing resources may be sufficient to handle incoming requests. During peak traffic periods, the number of requests may increase
significantly, resulting in higher CPU utilization, memory consumption and response time. Traditional load-balancing techniques such as Round Robin, Weighted Round Robin and Least Connections distribute incoming requests among available application instances. Although these techniques help distribute workloads, they do not necessarily predict future resource requirements.Similarly, conventional autoscaling mechanisms generally depend on predefined resource-utilization thresholds. Additional application instances are created only after the observed resource utilization reaches the configured threshold. This reactive approach may introduce delays because starting new containers and preparing them to handle requests requires additional time.

II. Research Background

Cloud resource management plays an important role in maintaining application performance and service availability. Effective resource allocation ensures that sufficient computing resources are available to handle incoming requests while avoiding unnecessary

 ©IJSRED: All Rights are Reserved

resource consumption.In traditional cloud environments, resource management is mainly performed using static allocation and reactive autoscaling techniques.

Static resource allocation provides a fixed number of application instances regardless of workload variations. This approach may result in resource wastage during low traffic conditions and insufficient capacity during peak traffic. Reactive autoscaling monitors system parameters such as CPU and memory utilization and adjusts the number of application instances when predefined thresholds are crossed. Kubernetes Horizontal Pod Autoscaler is commonly used to increase or decrease the number of pod replicas according to configured metrics.However, reactive autoscaling does not directly anticipate future workload changes. When sudden traffic increases occur, additional resources may not become available immediately.

Machine learning-based workload prediction provides an alternative approach by analysing historical resource measurements and estimating future resource requirements. Time-series forecasting techniques can identify patterns in workload behaviour and provide predictions that support earlier resource allocation.

Long Short-Term Memory (LSTM) is a recurrent neural network architecture suitable for sequential data processing. Its internal memory mechanism helps identify temporal relationships between previous and current workload observations.

The proposed project combines LSTM-based CPU prediction with monitoring and Kubernetes deployment configurations to explore proactive resource management for cloud microservices.

III. Literature Review

Several research studies have investigated machine learning-based autoscaling, predictive load balancing and cloud resource management.

Abdelhamid et al. (2025) proposed a Kubernetes autoscaling approach using machine learning-based traffic load prediction. Their study investigates the integration of predicted traffic information with Kubernetes Horizontal Pod Autoscaler to improve resource allocation. The work highlights the importance of predictive information in reducing the limitations of traditional reactive scaling. [1]

Bandaru Palli et al. (2025) presented predictive load balancing in microservices architecture using LSTM networks. Their research focuses on workload forecasting to improve application performance and reduce latency.

The study demonstrates the relevance of deep learning techniques for analysing sequential workload information. [2]

Wang et al. (2024) proposed Deep Scaling, an approach for autoscaling microservices while maintaining stable CPU utilization in large-scale production cloud systems. Their work investigates deep learning-based resource management and scaling decisions in cloud environments. [3]

Chawla (2024) examined reinforcement learning-based adaptive load balancing for dynamic cloud environments. The study focuses on using feedback from system conditions to improve workload distribution and resource utilization. [4]

Kallel et al. (2024/2025) investigated deep reinforcement learning techniques for container scheduling in cloud environments. Their research explores intelligent container placement and resource allocation to improve scheduling efficiency. [5]

Santos et al. (2023) presented gym-hpa, an approach that uses reinforcement learning for efficient autoscaling in complex microservice-based Kubernetes applications. The study highlights the potential of learning-based approaches for improving autoscaling decisions. [6]

Research Gap

From the reviewed studies, it is evident that workload prediction, adaptive resource allocation and container orchestration are important areas of cloud resource management. Several existing approaches focus on improving application scalability through reactive autoscaling, machine learning-based workload forecasting and intelligent resource allocation. However, each approach has certain limitations when applied independently.

Traditional reactive autoscaling techniques mainly depend on current CPU and memory utilization. When the incoming workload increases suddenly, the system detects the resource demand only after utilization reaches a predefined threshold. As a result, the creation of additional containers may be delayed, which can lead to temporary resource

 ©IJSRED: All Rights are Reserved

shortages, increased response time and reduced application performance. Although reactive scaling is useful for handling general workload variations, it may not respond efficiently to unexpected traffic spikes.

IV. Problem Statement

Cloud-based microservices applications experience continuously changing workloads due to variations in user requests and application usage patterns. During normal operating conditions, the available backend instances may be sufficient to process incoming requests. However, during sudden increases in traffic, the existing application instances may not have enough computing resources to handle the increased demand. This can lead to higher CPU utilization, increased memory consumption, longer response times and reduced application performance. Traditional resource-management techniques generally use reactive autoscaling mechanisms to manage these workload variations. In such approaches, the system monitors resource parameters such as CPU and memory utilization and increases or decreases the number of application instances when predefined thresholds are reached. Although this mechanism helps maintain resource availability, it has a limitation: additional resources are allocated only after the workload has already increased. The time required to detect the increased demand, create new containers and make them ready to process requests may result in temporary performance degradation.

Another challenge is the inability of conventional reactive mechanisms to anticipate future workload requirements. Without analysing historical workload patterns, the system may not identify upcoming traffic increases in advance. This can result in inefficient resource utilization, where insufficient resources are available during peak workloads or unnecessary resources remain allocated during low-traffic periods.

V. System Architecture and Methodology

The proposed system follows a modular architecture consisting of frontend services, backend services, workload generation, metric collection, machine learning prediction, monitoring and container orchestration.

The overall architecture integrates these components to monitor application behaviour, predict future resource requirements and provide pod recommendations.

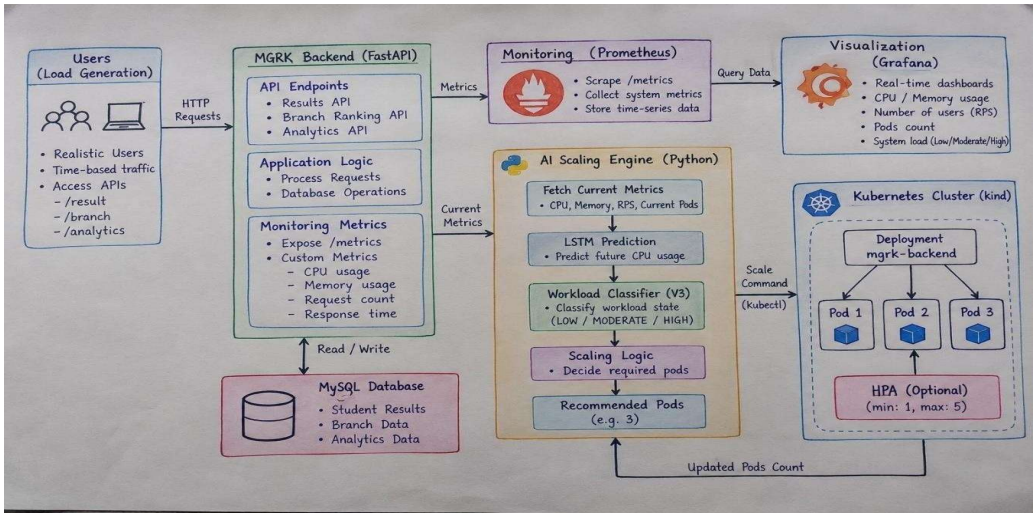


Fig 1: Overall architecture of proactive ML-based load balancing

 ©IJSRED: All Rights are Reserved

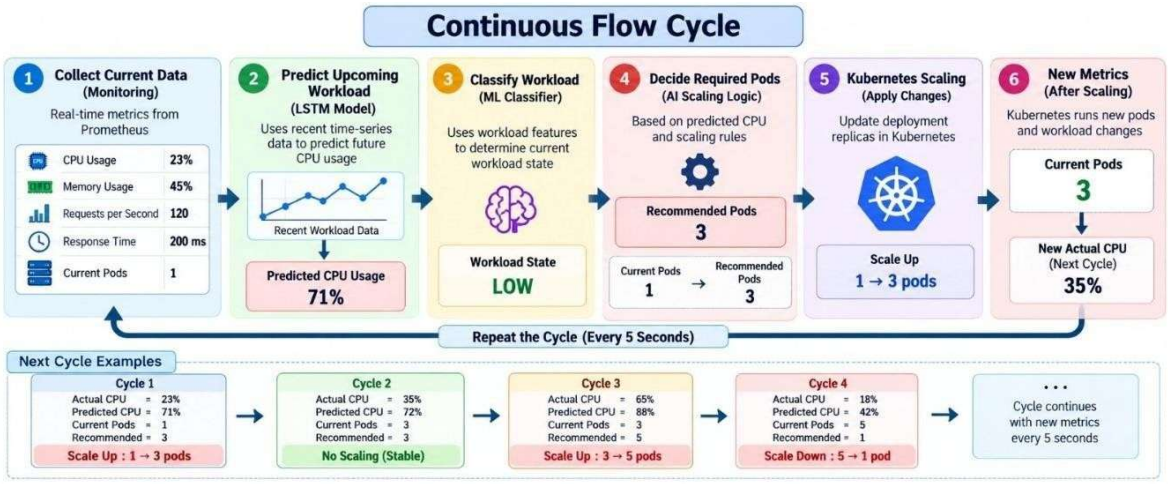


Fig 2: Continuous Flow Cycle

1. Frontend Application

The frontend is developed using React.js and React Router. It provides a web-based interface through which users can access the university service.

The frontend consists of the following modules:

- Home Page: Displays the main application interface.
- Results Page: Allows users to retrieve student results using their University Seat Number (USN).
- Branch Ranking: Displays student ranking information based on average marks.
- Analytics Page: Provides application-related analytical information. The frontend communicates with backend services through API requests.

2. Backend Services

The backend is developed using Python and FastAPI. It handles incoming requests, processes application operations and provides monitoring-related information.

The backend supports the following API endpoints:

- GET / : Returns the API status.
- GET /api/health : Provides service health information.
- GET /api/result/{usn} : Retrieves student result information.
- GET /api/branch-ranks : Provides branch ranking information.
- GET /api/college-ranks : Provides college ranking information.
- GET /api/analytics : Provides application analytics.
- /metrics : Exposes monitoring metrics in a Prometheus-compatible format.

The backend also tracks active requests and collects CPU and memory utilization information.

3. Workload Generation

- A workload simulator is developed to generate different levels of application traffic.
- The simulator supports light, medium and heavy workload profiles. These profiles perform different computational operations to represent variations in application demand.
- The load-testing configuration uses a starting population of 20 users and supports a maximum of 500 simulated users. The user population increases in increments of 20 at two-second intervals.
- Simulated users access different backend operations, including student results, branch rankings and college rankings.
- This process generates workload variations that can be analysed by the monitoring and prediction components.

4. Metric Collection and Data Processing

The system collects workload information at regular intervals and stores the observations in a time-series CSV dataset.

The dataset contains the following parameters:

- Timestamp
- Current Users
- CPU Utilization
- Memory Utilization
- Requests Per Second
- Response Time
- Load Level
- Most Requested API

The LSTM predictor uses five numerical input features:

1. Current Users
2. CPU Utilization
3. Memory Utilization
4. Requests Per Second
5. Response Time

The collected observations are arranged chronologically and processed before being supplied to the prediction

 ©IJSRED: All Rights are Reserved

model.

5. LSTM-Based CPU Prediction

The proposed system uses a Long Short-Term Memory network to analyse historical workload patterns and predict future CPU utilization.

LSTM is selected because it can learn temporal relationships from sequential observations. Unlike models that consider only individual measurements, LSTM can retain relevant information from previous time steps.

The prediction process consists of the following steps:

1. Collect historical workload observations.
2. Read the latest time-series dataset.
3. Select the five required numerical features.
4. Apply the saved data scaler.
5. Arrange the observations into a sequence.
6. Load the trained LSTM model.
7. Predict future CPU utilization.
8. Convert the predicted value into the original CPU percentage scale.

The implemented predictor uses a sequence of ten observations containing five input features.

6. Pod Recommendation Mechanism

- After predicting CPU utilization, the system applies a rule-based mechanism to recommend the required number of backend pods.
- The predicted CPU utilization is compared with predefined thresholds to determine the recommended pod capacity.
- This mechanism provides an additional predictive signal for resource-management decisions and supports preparation for changing workloads.
- The recommendation is intended to complement conventional Kubernetes scaling mechanisms rather than replace their safety controls.

7. Monitoring and Visualization

Prometheus and Grafana are integrated to monitor application performance and resource utilization. Prometheus collects metrics exposed by the backend and prediction services.

Grafana provides dashboards for visualizing important parameters, including:

- CPU utilization.
- Memory utilization.
- Active requests.
- Current pod count.
- Predicted CPU utilization.
- Recommended pod count.
- Workload variations.

These monitoring facilities help observe actual resource conditions and prediction-related information.

8. Containerization and Kubernetes Deployment

- Docker is used to package backend application components and their dependencies into containers.
- Kubernetes is used to manage containerized application services through Deployments and Services.
- Horizontal Pod Autoscaler configuration supports adjustments in backend replica counts according to configured resource metrics.
- The project also includes Prometheus and custom-metrics-adapter-related configuration files to support monitoring and predictive metric integration.
- The successful operation of predictive autoscaling depends on the correct configuration and validation of the Kubernetes metrics API, adapter rules and HPA settings.

Performance Evaluation

The performance evaluation framework is designed to study the behaviour of the proposed system under different workload conditions.

The system supports workload generation, resource monitoring, prediction and pod recommendation. Performance can be evaluated using the following parameters:

1. CPU Utilization: Measures processor usage during different workload conditions.
2. Memory Utilization: Measures memory consumption by application services.
3. Response Time: Represents the time taken to process application requests; direct per-request measurements are required for reliable latency evaluation.
4. Prediction Error: Evaluates the difference between actual and predicted CPU utilization.
5. Pod Count: Observes the number of backend replicas required under changing workloads.
6. Scaling Response Time: Measures the time required to make additional backend capacity available. The LSTM prediction model can be evaluated using Mean Absolute Error (MAE), Root Mean Square Error (RMSE) and Mean Absolute Percentage Error (MAPE).

The evaluation should compare reactive and prediction-assisted scaling under identical workload conditions. Repeated experiments are required to establish whether the predictive approach consistently improves resource utilization and application responsiveness. The evaluation is intended to determine whether incorporating historical workload information and LSTM-based CPU prediction can provide useful advance information for backend resource-management decisions. It also examines whether the pod recommendation mechanism can support Kubernetes-

 ©IJSRED: All Rights are Reserved

based resource allocation under changing traffic conditions.

The framework provides the necessary components for analysing prediction accuracy, resource utilization and scaling behaviour. However, actual performance improvements must be established through controlled experiments and measured results. The final evaluation should report the observed values rather than assume that prediction-assisted scaling always performs better than reactive autoscaling.

VI. Results

The implemented framework provides an integrated environment for workload simulation, resource monitoring, machine learning-based prediction and Kubernetes deployment configuration. It combines multiple technologies to analyse application workload variations and support proactive resource-management decisions in cloud microservices environments.The system begins by generating different workload conditions through a workload simulator, which represents variations in the number of users and incoming application requests. These requests are processed by the backend services developed using Python and FastAPI. During workload execution, important parameters such as active users, CPU utilization, memory utilization, requests per second and response time are collected for further analysis.

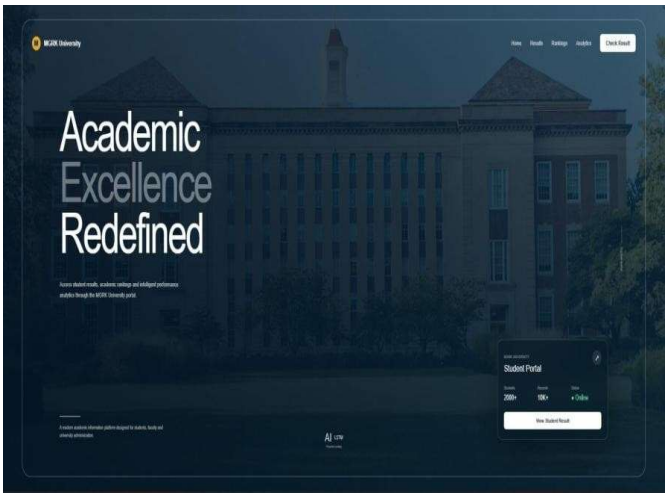


Fig 3: University Service Home Page

The home page serves as the main interface of the application, providing users with easy access to its functionalities. It is developed using React.js to provide a user-friendly and interactive experience. It allows users to navigate between different modules, such as student results, branch rankings and college analytics. The interface communicates with the backend services to retrieve and display the required information.

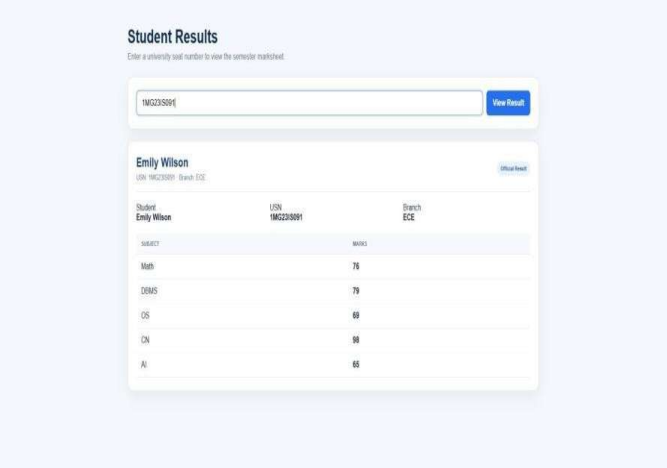


Fig 4: Student Results Page

The results page allows users to enter their University Seat Number (USN) to access their academic results. It sends the entered USN to the FastAPI backend for processing and retrieves the corresponding student information. The page displays the subject-wise marks and overall result details. It provides a simple and convenient way for students to check their academic performance.

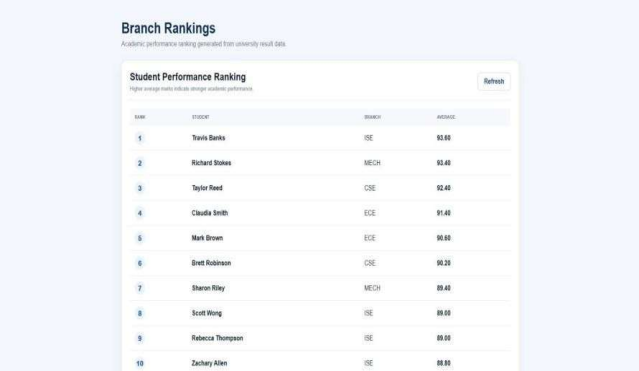


Fig 5: Branch Ranking Page

The branch ranking module displays the ranking of students based on their academic performance and average marks. It retrieves student details from the backend and arranges them according to their scores.

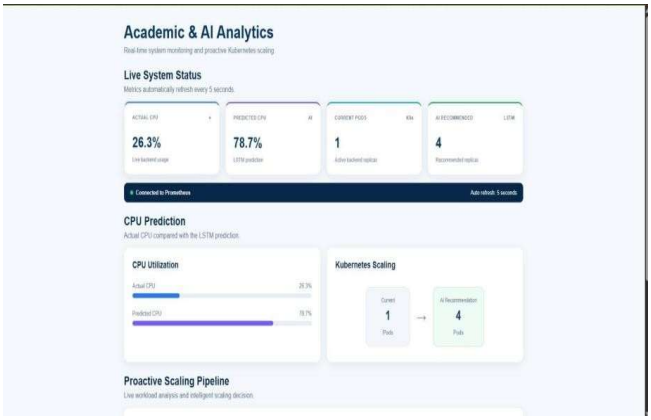


Fig 6: College Ranking and Analytics Page

The analytics module provides detailed information about college rankings based on student performance and

 ©IJSRED: All Rights are Reserved

average marks. It helps users compare the academic performance of different colleges.



Fig 7: Monitoring Dashboard

Prometheus collects real-time system metrics, including CPU utilization, memory usage, and application traffic. Grafana uses the collected metrics to create interactive dashboards and visual graphs. These tools help monitor system performance and resource utilization. They also help identify workload variations and potential performance issues in the application.

```
NAME              REFERENCE                    TARGETS                MINPODS   MAXPODS   REPLICAS   AGE
mgrk-backend-hpa  Deployment/mgrk-backend  cpu: 84%/60%, 71210m/60  1         5         5          81d
mgrk-backend-hpa  Deployment/mgrk-backend  cpu: 85%/60%, 72011m/60  1         5         3          81d
mgrk-backend-hpa  Deployment/mgrk-backend  cpu: 85%/60%, 79358m/60  1         5         4          81d
mgrk-backend-hpa  Deployment/mgrk-backend  cpu: 84%/60%, 86004m/60  1         5         5          81d
mgrk-backend-hpa  Deployment/mgrk-backend  cpu: 85%/60%, 79939m/60  1         5         4          81d
mgrk-backend-hpa  Deployment/mgrk-backend  cpu: 85%/60%, 72832m/60  1         5         3          81d
mgrk-backend-hpa  Deployment/mgrk-backend  cpu: 85%/60%, 69498m/60  1         5         5          81d
mgrk-backend-hpa  Deployment/mgrk-backend  cpu: 84%/60%, 69835m/60  1         5         3          81d
mgrk-backend-hpa  Deployment/mgrk-backend  cpu: 84%/60%, 72835m/60  1         5         3          81d
mgrk-backend-hpa  Deployment/mgrk-backend  cpu: 85%/60%, 85459m/60  1         5         5          81d
```

Fig 8: LSTM Prediction Output

The prediction component analyzes historical and real-time workload data to estimate future CPU utilization. It uses a machine learning model to identify workload patterns and predict upcoming resource requirements. The predicted CPU utilization is passed to the pod recommendation mechanism. This helps determine the required number of pods to handle future workloads efficiently.

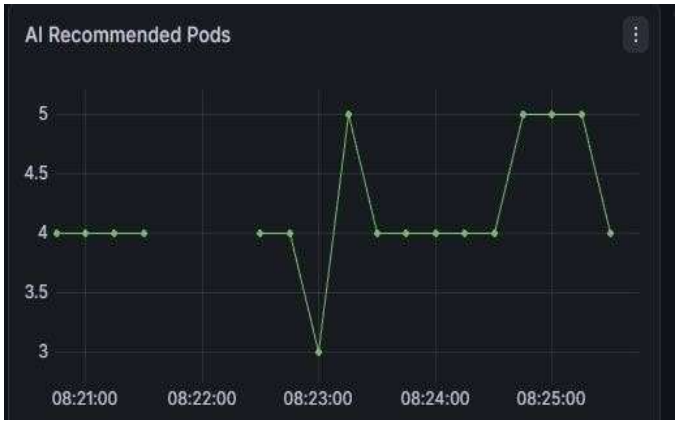


Fig 9: Kubernetes Deployment and HPA

The Kubernetes configuration includes backend Deployment, Service and Horizontal Pod Autoscaler resources for container management and replica scaling. The results demonstrate the implementation of the major components required for a prediction-supported resource-management framework. However, the numerical performance improvements and live predictive autoscaling behaviour must be established through reproducible experiments before making quantitative performance claims.

VII. Conclusion

This paper presents a Proactive Machine Learning-Based Load Balancing framework for cloud microservices. The proposed approach combines workload monitoring, historical data analysis, LSTM-based CPU prediction and rule-based pod recommendations to support proactive resource management. The system is developed using Python, Fast API and React.js, while Prometheus and Grafana provide monitoring and visualization facilities. Docker and Kubernetes support containerization and deployment, with Horizontal Pod Autoscaler configuration for managing backend replicas. The major contribution of this work is the integration of multivariate workload information with an LSTM-based prediction component to estimate future CPU utilization and recommend backend capacity. The framework provides a foundation for studying predictive resource management in cloud-hosted applications. Future work includes training and validating the prediction model using larger real-world workload datasets, comparing different forecasting algorithms, integrating real response-time measurements, implementing workload-aware scaling policies and conducting controlled experiments between reactive and prediction-assisted autoscaling.

References

[1] M. Abdelhamid, W. A. Aziz, and J. Soliman, “Kubernetes Autoscaling with Machine Learning based

 ©IJSRED: All Rights are Reserved

on traffic l load prediction," 2025.
[2] G. Bandaru Palli et al., "Predictive Load Balancing in Microservices Architecture Using LSTM Networks," 2025.
[3] Z. Wang, S. Zhu, J. Li, W. Jiang, K. K. Ramakrishnan, M. Yan, X. Zhang, and A. X. Liu, "Deep scaling: Autoscaling microservices with stable CPU utilization for large scale production cloud systems," IEEE/ACM Transactions on Networking, vol. 32, no. 5, pp. 3961–3976, 2024.
[4] K. Chawla, "Reinforcement Learning-Based Adaptive Load Balancing for Dynamic Cloud Environments," arXiv preprint, 2024.
[5] A. Kallel, M. Rekik, and M. Khemakhem, "Deep Reinforcement Learning for Container Scheduling in Cloud Environments," 2024/2025.

 ©IJSRED: All Rights are Reserved