IJCRT.ORG ISSN : 2320-2882

INTERNATIONAL JOURNAL OF CREATIVE RESEARCH THOUGHTS (IJCRT)

An International Open Access, Peer-reviewed, Refereed Journal

A TREND DETECTION-BASED AUTO-SCALING METHOD FOR CONTAINERS IN HIGH-CONCURRENCY SCENARIOS

[1]Dr.K. Subbarao, [2]Karnatapu Vishnu Saketh, [3]Ravipati Lokesh Sai Kumar, [4]Shaik Sahil, [5]Gunukula Rakesh

[1]HOD & Professor, [2,3,4,5]Students

Department of CSE-Data Science

St. Ann's College of Engineering & Technology, Chirala, Andhra Pradesh, India

Abstract: With the rapid advancement of cloud computing and the widespread adoption of containerized applications, managing high-concurrency workloads efficiently has become a significant challenge in modern distributed systems. Traditional auto-scaling techniques, particularly reactive approaches such as threshold-based scaling, often fail to respond promptly to sudden and unpredictable workload variations, resulting in resource over-provisioning or under-provisioning and degraded system performance. To address these limitations, this paper proposes a trend detection-based auto-scaling method that integrates Long Short-Term Memory (LSTM)-based workload prediction with a trend detection mechanism. A cooldown strategy is also implemented to prevent frequent and unnecessary scaling operations, ensuring system stability. The proposed approach effectively combines prediction, trend awareness, and controlled execution to optimize resource utilization while maintaining performance under dynamic workload conditions in containerized environments.

Index Terms **-** Auto-scaling, LSTM, Trend Detection, Kubernetes, Containers, High-Concurrency, Cloud Computing, Workload Prediction, Pod Allocation.

I. INTRODUCTION

Cloud computing has become a fundamental technology for deploying scalable and efficient applications in modern computing environments. With the increasing adoption of containerization technologies such as Docker and orchestration platforms like Kubernetes, applications are deployed as containerized services that require dynamic resource management [3], [4]. In high-concurrency scenarios, where incoming request rates vary rapidly, efficient auto-scaling becomes critical to maintain system performance and resource utilization.

Auto-scaling enables systems to dynamically adjust resources based on workload demand. Traditional approaches such as the Kubernetes Horizontal Pod Autoscaler (HPA) rely on resource utilization metrics such as CPU and memory [1]. However, these methods are reactive in nature and respond to workload changes only after they occur. This leads to delayed scaling decisions, which may result in performance degradation during sudden workload spikes and inefficient resource utilization during periods of low demand [8].

To overcome these limitations, predictive auto-scaling techniques have been introduced, which use historical workload data to forecast future demand. Machine learning models, especially Long Short-Term Memory (LSTM) networks, are widely used for time-series prediction due to their ability to capture temporal dependencies [6], [7]. However, prediction-based methods alone are not sufficient to handle abrupt and irregular workload fluctuations in high-concurrency environments.

This paper proposes a trend detection-based auto-scaling system designed to improve the efficiency of resource allocation in containerized applications. The system integrates LSTM-based workload prediction with a trend detection mechanism to identify sudden changes in traffic patterns. Additionally, a cooldown mechanism is incorporated to prevent frequent and unnecessary scaling operations, ensuring system stability.

The proposed approach enhances responsiveness and achieves better resource utilization compared to traditional auto-scaling methods.

II. LITERATURE SURVEY

This section presents a review of existing research on cloud auto-scaling methods for containerized applications, with emphasis on Kubernetes-based scaling, predictive auto-scaling, machine learning approaches, and limitations of existing mechanisms in high-concurrency environments.

A. Auto-scaling Approaches in Cloud Environments

Cloud computing platforms rely on elasticity to allocate resources dynamically according to workload demand. In containerized environments, horizontal scaling is widely used because containers are lightweight, portable, and suitable for rapid deployment [2], [3], [4]. The Kubernetes Horizontal Pod Autoscaler (HPA) is one of the most commonly used auto-scaling mechanisms and performs scaling by monitoring resource utilization and adjusting the number of pod replicas [1]. However, traditional threshold-based scaling methods are often insufficient in scenarios where request rates fluctuate suddenly and irregularly.

Table 2.1: Existing Auto-Scaling Approaches

Approach	Description	Limitation
Horizontal Pod Autoscaler (HPA)	Scales pod replicas based on utilization thresholds	Reactive in nature and slow during sudden spikes
Vertical Scaling	Adjusts CPU and memory resources allocated to containers	May interrupt service and affect stability
Rule-based Scaling	Uses predefined thresholds and policies	Cannot adapt well to dynamic workload patterns

B. Reactive and Proactive Scaling Methods

Auto-scaling strategies are generally categorized as reactive and proactive methods [7]. Reactive scaling adjusts resources only after workload changes are observed, which may lead to delayed response during burst traffic conditions. Proactive scaling attempts to predict future demand using historical workload patterns and allocate resources in advance. Machine learning-based proactive approaches are increasingly preferred because they reduce startup delay and improve service continuity [6], [7]. However, even proactive methods may fail when the system encounters abrupt and short-lived traffic surges that are not well represented in historical patterns [8].

C. Machine Learning Models in Auto-Scaling

Recent studies have explored machine learning techniques for container auto-scaling, especially for forecasting HTTP workload and scaling container replicas accordingly [6], [7], [8]. Long Short-Term Memory (LSTM) models are widely used because they can capture temporal dependencies in workload data and are suitable for time-series prediction [6]. In Kubernetes environments, predictive scaling methods based on LSTM have shown better performance than purely reactive methods under regular traffic conditions [8]. Surveys on Kubernetes auto-scalers indicate that no single method performs optimally for all workload types, particularly when high volatility and irregular concurrency patterns are involved [7].

Table 2.2: Machine Learning Models for Auto-Scaling

Method	Strength	Limitation
ARIMA	Suitable for regular time-series forecasting	Weak in highly non-linear workload data
LSTM	Captures long-term temporal dependencies	Prediction may lag during abrupt spikes
Hybrid Predictive Models	Combine forecasting with control logic	More complex to design and tune

D. Research Gaps and Need for Proposed System

From the literature, it is evident that current Kubernetes auto-scaling methods face difficulty in high-concurrency scenarios where workload behaviour changes rapidly. Reactive methods respond too late, while predictive methods may not always capture short-term irregular trends [7], [8]. Existing studies indicate that

container-based cloud applications require scaling mechanisms that reduce resource shortage, improve responsiveness, and avoid unnecessary resource wastage [2], [4], [8]. There is therefore a clear requirement for a hybrid auto-scaling approach that integrates workload prediction with trend-aware correction, which this work addresses by combining LSTM-based prediction with a trend detection mechanism.

III. PROPOSED METHODOLOGY

The proposed system is designed to provide an efficient and intelligent auto-scaling mechanism for containerized applications operating under high-concurrency conditions. The system focuses on improving the limitations of traditional reactive scaling methods by incorporating predictive and analytical techniques. It utilizes historical workload data to forecast future demand and enhances prediction using a trend detection mechanism to handle sudden workload variations. By combining prediction, trend correction, and controlled execution, the proposed system ensures better responsiveness, improved resource utilization, and reduced system instability.

A. System Architecture Overview

The architecture of the proposed system consists of multiple modules that work together to perform data processing, prediction, analysis, and scaling. The workflow begins with input workload data collection and preprocessing, followed by LSTM-based workload prediction. Predicted values are refined through a trend detection module, and final scaling decisions are executed through the scaling module. A visualization module presents results for analysis. Table 3.1 summarizes the components of the proposed system.

Table 3.1: Components of the Proposed System

Module	Description
Input Data Module	Accepts workload dataset (QPS values) and prepares data for further processing.
Prediction Module	Uses LSTM model to forecast future workload based on historical data patterns.
Trend Detection Module	Identifies sudden spikes or drops in workload and adjusts predictions accordingly.
Cooldown Module	Prevents frequent scaling actions by introducing a delay between scaling operations.
Scaling Module	Determines the required number of pods based on predicted workload.
Visualization Module	Displays results in graphical format for analysis.

B. Data Input and Preprocessing

The system begins by accepting workload data, typically represented as Queries Per Second (QPS) values over time. This data is provided by the user in the form of an Excel file containing a 'requests' column. Before feeding the data into the prediction model, preprocessing is performed to ensure data consistency and quality. This includes handling missing or inconsistent values, normalizing data to improve model performance, and structuring data into a time-series format. Normalization is particularly important for LSTM models, as it ensures that input values fall within a specific range, allowing the model to learn patterns more effectively.

C. Workload Prediction Using LSTM

After preprocessing, the data is passed to the LSTM model, which is specifically designed for time-series prediction. The LSTM model analyzes historical workload patterns and captures temporal dependencies between data points. The model generates predicted QPS values for future time intervals, which form the basis for scaling decisions. By utilizing LSTM, the system performs proactive scaling by anticipating future demand instead of reacting to changes after they occur. The prediction process involves learning patterns from past workload behavior and generating a forecast that reflects the expected workload in upcoming time intervals.

D. Trend Detection Mechanism

Although the LSTM model provides accurate predictions under normal conditions, it may not effectively capture sudden workload changes. To address this limitation, a trend detection mechanism is introduced. This module monitors changes in predicted workload, detects sudden spikes or drops in traffic, and adjusts predicted values to reflect real-time behavior. The algorithm computes running mean and standard deviation

of QPS values and applies correction logic when significant deviations are detected. If a sudden increase in workload is identified, the module adjusts the predicted values to reflect this change, improving the responsiveness of the system to unexpected workload fluctuations and making scaling decisions more reliable.

E. Cooldown Mechanism

Frequent scaling actions can lead to system instability and resource wastage. To prevent this, the proposed system includes a cooldown mechanism. The cooldown mechanism introduces a controlled delay between consecutive scaling actions, prevents unnecessary scaling triggered by minor fluctuations, and ensures stability in resource allocation. A cooldown timer (CDT) is maintained, and scaling actions are suppressed while the cooldown period is active. This approach maintains a balance between responsiveness and system stability, avoiding oscillatory scaling behavior common in purely reactive approaches.

F. Scaling Decision and Pod Allocation

Based on the adjusted and predicted workload values, the system determines the number of pods required to efficiently handle incoming requests. The calculation involves dividing the total expected workload (QPS) by the per-pod handling capacity. The result is rounded upward to ensure that sufficient resources are available without causing service degradation. The scaling logic dynamically increases or decreases the number of pods depending on workload intensity, ensuring efficient resource utilization and optimal system performance. Table 1.1 provides a comparison of the existing and proposed systems, highlighting the improvements achieved by the proposed approach.

Table 1.1: Comparison of Existing vs. Proposed System

Aspect	Existing System	Proposed System
Scaling Type	Reactive	Proactive + Trend-based
Decision Basis	CPU/Memory thresholds	LSTM prediction + trend detection
Response Time	Delayed	Faster response
Handling Sudden Spikes	Poor	Improved
Stability	Frequent oscillations	Controlled using cooldown

IV. IMPLEMENTATION

The implementation of the proposed trend detection-based auto-scaling system focuses on developing a practical solution that integrates machine learning with dynamic scaling logic. The system is implemented using Python, where different modules are combined to perform data processing, prediction, trend analysis, and scaling operations in a sequential manner. The implementation follows a modular approach so that each component performs a specific function independently while contributing to the overall system workflow.

A. Software and Hardware Requirements

The system is developed using Python as the primary programming language. Supporting libraries include Pandas and NumPy for data processing, TensorFlow/Keras for LSTM model implementation, Scikit-learn for data preprocessing, and Matplotlib/Plotly for visualization. Streamlit is used as the user interface framework. The development environment consists of VS Code or Jupyter Notebook on Windows or Linux operating systems. The hardware requirements are minimal, requiring an Intel i5 processor or above, a minimum of 8 GB RAM, 256 GB or more of storage, and a 64-bit system configuration.

B. Data Input Module Implementation

The data input module acts as the starting point of the system. It allows the user to upload workload data in the form of an Excel file containing QPS values representing incoming request load over time. Once the dataset is uploaded, the system reads the file, extracts the required data, and verifies whether the dataset contains the necessary column. If the dataset is invalid or does not meet the required format, the system prevents further execution and prompts the user to provide correct input. The following code segment illustrates the data loading process:

```
uploaded_file = st.file_uploader("Upload Excel File (.xlsx)", type=["xlsx"])
df = pd.read_excel(uploaded_file)
if 'requests' not in df.columns:
    st.error("File must contain a 'requests' column")
```

C. LSTM Prediction Module Implementation

The prediction module forms the core of the implementation. The preprocessed data is provided as input to the trained LSTM model, which analyzes historical workload patterns and generates predicted QPS values for future time intervals. The use of LSTM enables the system to perform proactive scaling by anticipating future demand. The prediction function is implemented as follows:

```
def make_single_prediction(model, input_data):
    prediction = model.predict(input_data.reshape(1, input_data.shape[0], 1))
    return prediction
```

D. Trend Detection Module Implementation

The trend detection module enhances prediction results by identifying sudden changes in workload patterns. The module applies correction logic when significant deviations from expected behavior are detected. The core algorithm monitors the running mean and standard deviation of QPS values, applies FLAG counters to detect upward or downward trends, and adjusts predictions accordingly. A linear regression component is employed to identify the slope of workload change and inform scaling direction. This module directly improves the responsiveness and accuracy of scaling decisions in highly dynamic environments.

V. TESTING

The proposed auto-scaling system was subjected to systematic testing to validate the functionality of each module and the integrated system as a whole. Test cases were designed to cover data input validation, prediction accuracy, trend detection responsiveness, cooldown behavior, pod allocation logic, high-concurrency simulation, integration, and output visualization.

Table 6.1: Test Cases for the Proposed Auto-Scaling System

Test Case ID	Description	Input	Expected Output	Status
TC_01	Validate data input module	Workload dataset (QPS values)	Data loaded without errors	Pass
TC_02	Check LSTM prediction model	Historical workload data	Predicted future QPS values generated	Pass
TC_03	Validate trend detection mechanism	Sudden spike in workload	System detects trend and adjusts prediction	Pass
TC_04	Verify cooldown mechanism	Frequent scaling triggers	Scaling actions controlled and stabilized	Pass
TC_05	Test pod allocation logic	Predicted QPS values	Correct number of pods calculated	Pass
TC_06	System behavior under high concurrency	Large workload spikes	System scales efficiently without delay	Pass
TC_07	Integration testing	Combined module execution	All modules work together correctly	Pass
TC_08	Output visualization	Simulation results	Graphs and outputs displayed correctly	Pass

All test cases were executed successfully, confirming that the proposed system operates correctly under various conditions. The passing of TC_03 and TC_04 in particular validates the effectiveness of the trend detection and cooldown mechanisms as designed.

VI. RESULTS AND DISCUSSION

The proposed trend detection-based auto-scaling system was implemented and evaluated using a workload dataset consisting of QPS values representing real-world traffic patterns. The results demonstrate the capability of the system to process workload data, generate accurate predictions, detect workload trends, and make appropriate scaling decisions.

A. Dataset Upload and Preprocessing

The system provides a user-friendly interface for uploading workload datasets in Excel format. Upon upload, the QPS values are extracted and validated. A data preview is displayed alongside statistical information about the dataset. The interface allows users to interactively initiate the auto-scaling simulation, providing transparency in the input stage.

B. Workload Analysis

The QPS Over Time graph (Figure 7.2) visualizes the variation of QPS values over the duration of the dataset, revealing the dynamic nature of the workload and the presence of traffic fluctuations. The QPS Distribution graph (Figure 7.3) illustrates the frequency of different workload intensities, providing insights into the variability of the traffic patterns. A statistical summary (Figure 7.4) presents key measures including mean, median, standard deviation, and maximum values, characterizing the overall behavior of the workload data.

C. Prediction Accuracy

The LSTM prediction module generates forecasted QPS values based on historical patterns captured from the training data. The QPS Prediction Accuracy graph (Figure 7.5) presents a comparison between actual and predicted QPS values. The prediction results indicate that the LSTM model captures the general trend of the workload effectively. The integration of the trend detection mechanism further refines these predictions during periods of sudden workload change, improving the overall accuracy of the scaling decisions.

D. Pod Allocation

The Pod Allocation Timeline (Figure 7.6) illustrates the number of pods allocated over time based on the predicted and trend-adjusted workload values. The results demonstrate that the system dynamically increases pod count during high-traffic periods and reduces allocation during low-demand intervals. The cooldown mechanism ensures that pod allocation changes do not oscillate excessively, maintaining system stability throughout the simulation. These results confirm that the proposed system achieves improved scalability and more efficient resource management compared to conventional reactive auto-scaling approaches.

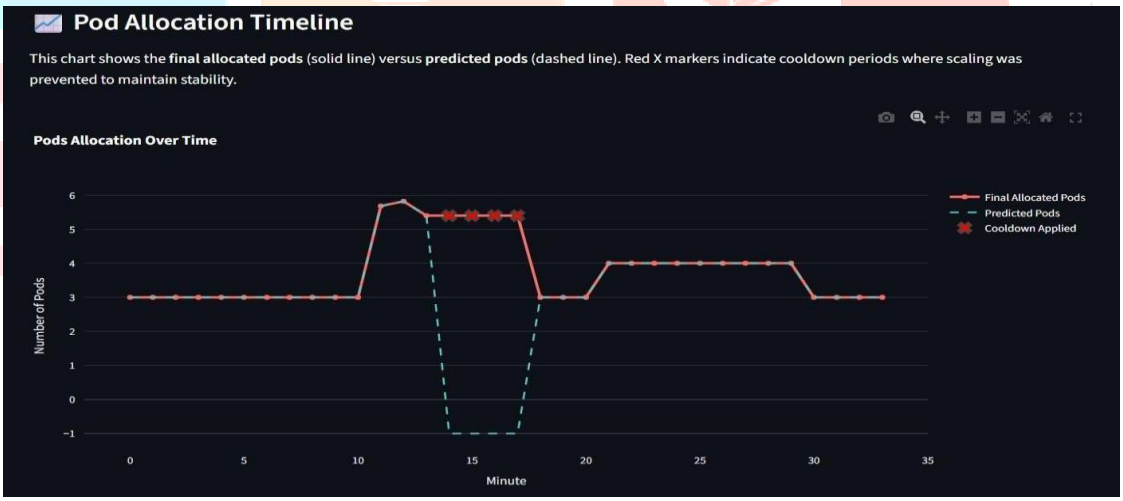

Figure 1. showing Pod Allocation Timeline

VII. CONCLUSION

This paper has presented a trend detection-based auto-scaling method for containerized applications in high-concurrency scenarios. The proposed system enhances traditional auto-scaling approaches by integrating LSTM-based workload prediction with a trend detection mechanism to handle sudden fluctuations in incoming requests. The inclusion of a cooldown strategy helps in avoiding unnecessary scaling actions and ensures system stability. The proposed method effectively improves resource utilization and scaling responsiveness compared to conventional reactive techniques. The modular design and successful testing of all components confirm the feasibility and correctness of the proposed approach. Overall, the system demonstrates the importance of combining predictive analysis with trend awareness to achieve efficient and reliable auto-scaling in modern cloud environments.

VIII. FUTURE SCOPE

The proposed trend detection-based auto-scaling system can be further enhanced in several directions. Incorporating multiple resource metrics such as CPU, memory, and network usage alongside QPS values would enable more comprehensive and accurate scaling decisions. Future work can focus on deploying the

system in real-time Kubernetes environments to validate its performance under practical production conditions. Advanced prediction models such as Gated Recurrent Units (GRU) or Transformer-based architectures can be explored to improve forecasting accuracy for highly dynamic and irregular workloads. The system can also be extended to support multi-service and microservices-based architectures, along with cost-aware scaling strategies to optimize resource utilization and operational expenses.

ACKNOWLEDGMENT

The authors express sincere thanks to Dr. K. Subbarao, Head of Department of CSE-Data Science, St. Ann's College of Engineering & Technology, Chirala, for his valuable guidance and support throughout the project. The authors are also grateful to the management of St. Ann's College of Engineering & Technology for providing a conducive research environment and excellent laboratory facilities.

REFERENCES

[1] Kubernetes, "Horizontal Pod Autoscaling," Kubernetes Documentation, Accessed: Dec. 17, 2023. [Online]. Available: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

[2] J. Dobies and J. Wood, Kubernetes Operators: Automating the Container Orchestration Platform. Sebastopol, CA, USA: O'Reilly Media, 2020.

[3] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and Linux containers," in Proc. IEEE Int. Symp. Performance Analysis of Systems and Software (ISPASS), Mar. 2015, pp. 171–172.

[4] N. Zhou, H. Zhou, and D. Hoppe, "Containerization for high performance computing systems: Survey and prospects," IEEE Transactions on Software Engineering, vol. 49, no. 4, pp. 2722–2740, Apr. 2023.

[5] S. Risco, G. Moltó, D. M. Naranjo, and I. Blanquer, "Serverless workflows for containerised applications in the cloud continuum," Journal of Grid Computing, vol. 19, no. 3, Sep. 2021.

[6] L. Toka, G. Dobreff, B. Fodor, and B. Sonkoly, "Machine learning-based scaling management for Kubernetes edge clusters," IEEE Transactions on Network and Service Management, vol. 18, no. 1, pp. 958–972, Mar. 2021.

[7] N. T. Tran, M. T. Kechadi, and A. Le-Khac, "A survey of auto-scaling in Kubernetes," Journal of Cloud Computing, 2022.

[8] N. Marie-Magdelaine and T. Ahmed, "Proactive autoscaling for cloud-native applications using machine learning," in Proc. GLOBECOM IEEE Global Communications Conf., Dec. 2020.

[9] N. Mungoli, "Scalable, distributed AI frameworks: Leveraging cloud computing for enhanced deep learning performance and efficiency," arXiv preprint arXiv:2304.13738, Apr. 2023.

[10] M.-N. Tran, D.-D. Vu, and Y. Kim, "A survey of autoscaling in Kubernetes," in Proc. 13th Int. Conf. Ubiquitous Future Netw. (ICUFN), Jul. 2022, pp. 263–265.

[11] M. Filho, E. Pimentel, W. Pereira, P. H. M. Maia, and M. I. Cortés, "Self-adaptive microservice-based systems landscape and research opportunities," in Proc. Int. Symp. Software Eng. Adapt. Self-Managing Syst. (SEAMS), May 2021, pp. 167–178.

[12] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle, "Autonomic vertical elasticity of Docker containers with ELASTICDOCKER," in Proc. IEEE 10th Int. Conf. Cloud Comput. (CLOUD), Jun. 2017, pp. 472–479.

[13] G. Rattihalli, M. Govindaraju, H. Lu, and D. Tiwari, "Exploring potential for non-disruptive vertical auto scaling and resource estimation in Kubernetes," in Proc. IEEE 12th Int. Conf. Cloud Comput. (CLOUD), Jul. 2019, pp. 33–40.

[14] A. Goli, N. Mahmoudi, H. Khazaei, and O. Ardakanian, "A holistic machine learning-based autoscaling approach for microservice applications," in Proc. 11th Int. Conf. Cloud Comput. Services Sci., 2021, pp. 190–198.

[15] W. Iqbal, A. Erradi, and A. Mahmood, "Dynamic workload patterns prediction for proactive auto-scaling of web applications," Journal of Network and Computer Applications, vol. 124, pp. 94–107, Dec. 2018.