Kubernetes Autoscaling with Machine Learning based on traffic load prediction

Conference Paper · May 2025

CITATIONS
0

READS
192

3 authors, including:

Wagdy Anis Aziz
Ain Shams University
50 PUBLICATIONS **43** CITATIONS
SEE PROFILE

John N. Soliman
Ain Shams University
19 PUBLICATIONS **28** CITATIONS
SEE PROFILE

Kubernetes Autoscaling with Machine Learning based on traffic load prediction

Yousef M. Abdelhamid
Faculty of Media Engineering and Technology
German University in Cairo
Cairo, Egypt
yousef.abdellatif@student.guc.edu.eg

Wagdy A. Aziz
Faculty of Engineering
Ain Shams University/Orange-Egypt
wagdy.anis@orange.com

John N. Soliman
M.Sc., Faculty of Engineering
Ain Shams University
john.n.soliman@ieee.org

***Abstract*—This paper explores a proactive approach to Kubernetes autoscaling by integrating Machine Learning (ML) traffic prediction with the Horizontal Pod Autoscaler (HPA) [1]. Traditional reactive autoscaling methods respond to current resource usage, which can lead to delayed responses during traffic surges and inefficient resource allocation. This study leverages Amazon Web Services (AWS), particularly Amazon EKS, SageMaker, and CloudWatch, to develop a predictive scaling pipeline. A simulated traffic dataset was used to train an ML model capable of forecasting future request volumes based on time-based and system metrics. These predictions were published to CloudWatch as custom metrics and used to drive HPA decisions in advance of traffic changes. Testing under varying load conditions demonstrated improved responsiveness and resource optimization. The results support the feasibility of predictive autoscaling in Kubernetes environments using cloud-native ML tools.**

Index Terms—Kubernetes, machine learning, autoscaling, AWS, EKS, SageMaker, CloudWatch, proactive scaling

I. Introduction

A. Cloud Computing

In recent years, the emergence of cloud computing has revolutionized the way businesses deploy and manage their applications. What once required dedicated physical infrastructure, costly hardware, and manual maintenance can now be handled by remote data centers with just a few clicks. This shift has significantly lowered the barrier of entry for startups and streamlined operations for established enterprises.

Cloud computing allows data and services to be accessed over the internet through a network of remote servers, rather than relying on local machines. Users can store, manage, and process their data without having to worry about the underlying physical infrastructure. Services such as storage, databases, computing power, and networking are offered on-demand by providers such as Amazon Web Services (AWS), Google Cloud Platform (GCP), and Microsoft Azure.

The popularity of cloud computing lies in its key benefits: scalability, flexibility, cost-efficiency, and reliability. Businesses can quickly scale their resources up or down based on traffic, launch applications across global regions, and only pay for what they use. Moreover, cloud platforms offer high availability, disaster recovery, and robust security features to meet the needs of modern workloads.

Today, cloud computing plays a vital role in supporting a wide range of applications — from video streaming and social media to enterprise-level data analytics and machine learning. This paper explores how cloud services, particularly those offered by AWS, can be used in combination with container orchestration and predictive machine learning models to address challenges in autoscaling cloud-native applications.

B. Kubernetes and Autoscaling

As applications grow in complexity and user demand fluctuates unpredictably, managing deployment and scalability becomes a critical challenge. Kubernetes, an open-source container orchestration platform, has emerged as the industry standard for automating the deployment, scaling, and operation of containerized applications. It enables developers and operators to abstract infrastructure concerns and focus on building resilient and scalable services.

Kubernetes introduces the concept of declarative infrastructure, where users define the desired state of their applications, and the platform actively ensures that this state is maintained. It handles container scheduling, rolling updates, service discovery, and most importantly, scaling. One of its key features, the Horizontal Pod Autoscaler (HPA) adjusts the number of running pods in a deployment based on real-time resource utilization metrics such as CPU and memory.

However, traditional autoscaling mechanisms like HPA are reactive by design. They respond to current usage trends rather than anticipating future demand, which can lead to delayed scaling during sudden traffic spikes or underutilized resources during off-peak times. In environments with high variability in traffic, this can result in performance degradation or unnecessary costs.

To address this limitation, this research explores a proactive autoscaling strategy that integrates machine learning (ML) models for predicting traffic load in advance. By combining Kubernetes with AWS services such as Elastic Kubernetes Service (EKS), CloudWatch [2], and SageMaker [3], a predictive autoscaler can be implemented. This system enables intelligent scaling decisions based on forecasted traffic, allowing the platform to prepare for upcoming demand rather than reacting to it. Such an approach aims to optimize performance, improve

resource efficiency, and enhance the overall reliability of cloud-native applications [4].

II. Background

A. Kubernetes Scaling Models

To better understand the direction of this study, it is important to examine the different scaling methods Kubernetes provides for managing workloads. While Kubernetes supports various forms of scaling, this section will focus on three of the most widely used and relevant models.

1) Horizontal Pod Autoscaler (HPA): HPA is the most commonly used scaling method in Kubernetes. It automatically adjusts the number of pod replicas in a deployment or replica set based on observed metrics, such as CPU utilization or custom application-level metrics. For instance, if CPU usage consistently exceeds a threshold, HPA increases the number of pods to handle the load. HPA is ideal for stateless applications where demand can fluctuate significantly over short periods.

2) Vertical Pod Autoscaler (VPA): In contrast to HPA, which scales the number of pods, VPA adjusts the CPU and memory requests and limits of individual pods. It is suitable for workloads with consistent traffic where optimizing resource allocation is more important than adjusting the number of instances. However, VPA can be disruptive because it often requires restarting pods to apply new resource values.

3) Cluster Autoscaler (CA): The Cluster Autoscaler operates at the infrastructure level. It adjusts the number of nodes in the Kubernetes cluster based on pending pods and resource requirements. If pods cannot be scheduled due to resource constraints, CA provisions new nodes. Similarly, it scales down by removing underutilized nodes, reducing costs. This model works alongside HPA and VPA to provide full-stack elasticity in a Kubernetes environment.

B. Benefits of Predictive Autoscaling

Predictive autoscaling augments traditional reactive methods by introducing foresight into resource planning, which enables clusters to scale proactively based on predicted demand. This section outlines several key benefits of incorporating traffic prediction into the autoscaling process.

1) Improved Responsiveness: Unlike reactive autoscaling models that respond only after traffic changes occur, predictive autoscaling allows the system to prepare ahead of time. For example, if a sudden spike in traffic is expected at 6 PM, resources can be provisioned minutes in advance, ensuring smooth performance.

2) Reduced Latency and Downtime: By preemptively scaling up during forecasted peak times, predictive autoscaling minimizes the risk of delayed response times or pod unavailability. This is especially useful for user-facing applications where latency impacts customer experience.

3) Cost Efficiency: While over-provisioning ensures availability, it also leads to unnecessary costs. Predictive autoscaling finds a balance by scaling resources only when necessary and scaling down during anticipated low-traffic periods. This dynamic allocation reduces cloud expenses over time.

4) Better Resource Utilization: Predictive models analyze usage patterns to anticipate demand, leading to more efficient use of CPU, memory, and storage resources. This prevents underutilization of resources and helps optimize infrastructure sizing.

C. Machine Learning in Autoscaling

Traditional Kubernetes autoscaling reacts to current resource usage like CPU or memory, but this can lead to delays during traffic spikes. Machine learning (ML) can enhance autoscaling by predicting future traffic based on historical patterns, such as time of day or past request volumes. With this predictive approach, autoscalers can adjust resources ahead of demand, avoiding performance lags and reducing unnecessary overprovisioning. By using ML models trained on traffic data, autoscaling becomes more proactive and efficient, improving responsiveness during high traffic and saving resources during low periods. This approach helps optimize resource management and enhances the overall user experience.

III. Methodology

For this solution, the infrastructure setup was divided into three key phases: planning, implementation, and testing. During the planning phase, the core components and functionalities of the system were defined—such as the structure of the Kubernetes cluster, which AWS services would be integrated (e.g., SageMaker, CloudWatch, EKS), and how autoscaling behavior would be influenced by predicted traffic load rather than reactive thresholds.

A. Planning and System Architecture

The first step was to determine the infrastructure and services needed for the solution. The goal was to design a responsive and cost-effective architecture that could scale based on predicted traffic loads. Amazon EKS, CloudWatch, and SageMaker were selected for orchestration, monitoring, and machine learning tasks. After comparing regions, the me-central-1 region was chosen for its support of the required services and compatibility with the AWS Free Tier. A visual diagram of the proposed architecture was then created using Lucidchart, incorporating key AWS components to guide the implementation.

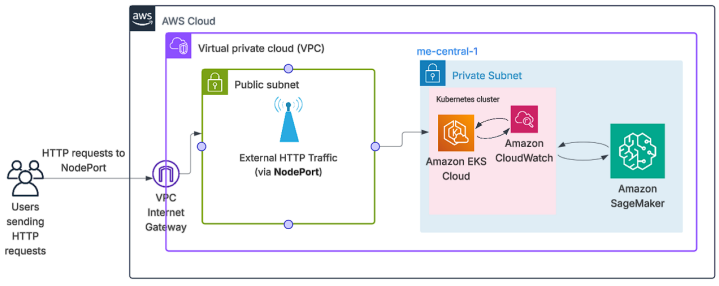

Fig. 1: Project Architecture

B. Infrastructure Setup and Test App Deployment

With our infrastructure determined, the next step would be the actual implementation and creation of the specified resources. The first step was to create an EKS cluster [5] in the AWS region me-central-1, ensuring that the cluster was properly configured with the required VPC, subnets, and IAM roles for the nodes to communicate effectively with other AWS services. The AWS CLI was used for this setup, and the cluster was verified using the kubectl command to ensure that all nodes were up and running. Then, once the EKS cluster was ready, the next step involved deploying the Flask-based test application [6] , test-app, which had already been Dockerized and pushed to Amazon Elastic Container Registry (ECR). The deployment was done using Kubernetes manifests to define the deployment and service configurations. The Flask app was set up to respond to HTTP requests and simulate real traffic, which was critical for testing the autoscaling behavior under varying loads.

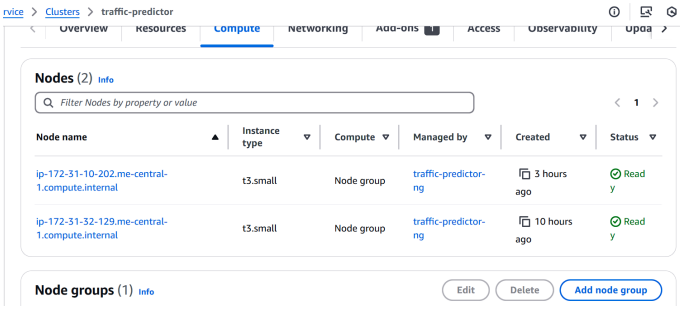

Fig. 2: EKS Cluster

C. CloudWatch Monitoring and Dataset Collection

CloudWatch Container Insights was integrated with the EKS cluster to monitor application and infrastructure performance [7] . The CloudWatch agent collected key metrics like CPU usage, memory usage, which were visible in the CloudWatch console and used for scaling decisions.

1) CloudWatch Setup: The aws-cloudwatch-metrics Helm chart was used to deploy the CloudWatch agent, enabling the collection of container metrics.

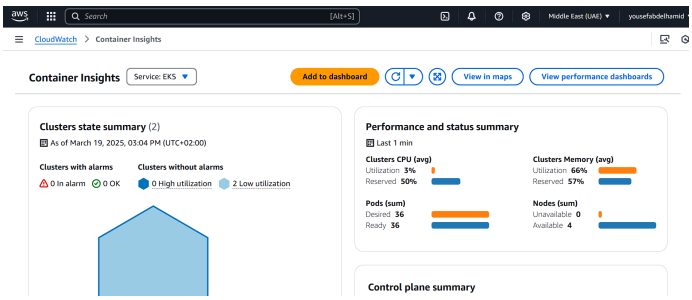

Fig. 3: CloudWatch Console

2) Dataset Collection: A simulated traffic dataset was generated, including features such as hour of the day, day of the week, request count, and resource usage. This data was pre-processed for machine learning model training, ensuring clean and normalized data.

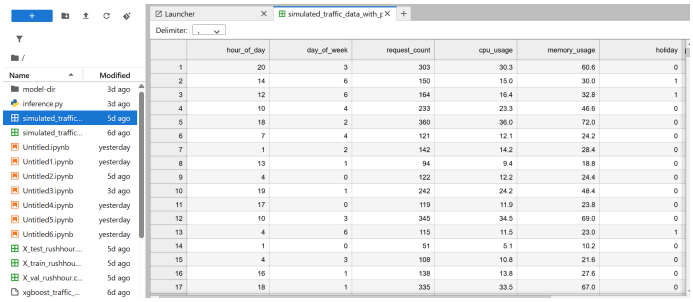

Fig. 4: Training Dataset

D. Machine Learning Model Development and Deployment

In this phase, a machine learning model was developed using Amazon SageMaker to predict traffic loads [8] [9] , which would inform Kubernetes autoscaling decisions. XGBoost was chosen for its efficiency in regression tasks.

1) Model Training and Evaluation: The model was trained using historical traffic data and features like hour of day, CPU, and memory usage. After training, the model was evaluated using Mean Absolute Error (MAE) to ensure accurate predictions.

```
[96]    train-mae:0.33472       eval-mae:0.37575
[97]    train-mae:0.33445       eval-mae:0.37565
[98]    train-mae:0.33417       eval-mae:0.37550
[99]    train-mae:0.33388       eval-mae:0.37549
✅ Test MAE: 0.35
✅ Model saved as xgboost-traffic-predictor-rush.model
```

Fig. 5: Training Completion and Evaluation

2) Model Deployment: The trained model was deployed to an Amazon SageMaker Endpoint for real-time inference. This allowed the EKS cluster to query the model and use the predicted traffic load to adjust the autoscaling behavior.

Fig. 6: Endpoint Deployment

E. Integration with EKS and Automation

The integration of the machine learning model with the EKS cluster [10] was done through a Python script running inside the cluster. This script continuously fetched real-time metrics, sent them to the SageMaker endpoint for traffic predictions, and adjusted the scaling parameters accordingly.

1) Model Integration with EKS: A Python script inside the EKS cluster was responsible for collecting current metrics, such as CPU usage and request count, from CloudWatch. These metrics were then sent to the SageMaker endpoint to retrieve traffic predictions. The predicted request count was pushed as a custom metric [11] on Cloudwatch.

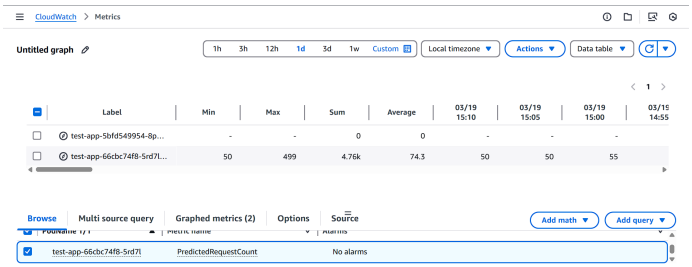

Fig. 7: Custom Metrics

2) Modified Horizontal Pod Autoscaler (HPA): The HPA [12] was configured to scale pods based on the predicted request count. The integration allowed the system to scale proactively based on forecasted traffic, preventing delays during sudden spikes or drops in load.

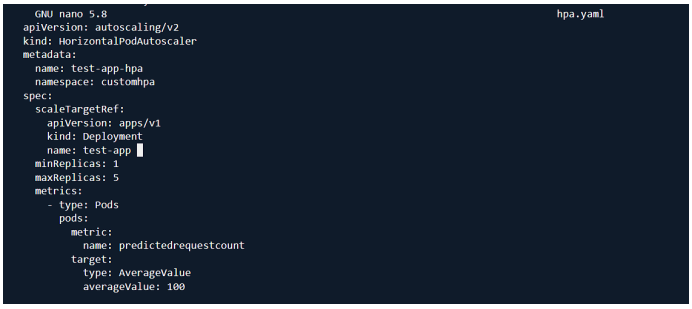

Fig. 8: Modified HPA

3) Automation with CronJobs: The entire process, from fetching metrics to making predictions and adjusting scaling parameters, was automated using Kubernetes CronJobs. This ensured that predictions were continuously updated, and the autoscaling process was handled automatically without manual intervention.

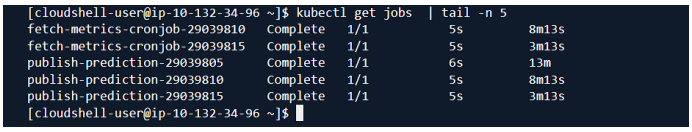

Fig. 9: Cronjobs Running

IV. Test Use Cases

The traffic prediction model was trained using a dataset that simulates realistic usage patterns of a web application. Specifically, the dataset reflected consistent surges in traffic on weekends and during the afternoon hours of each day, particularly between 3:00 PM and 5:00 PM. Conversely, the dataset assumed significantly lower traffic volumes during very early and very late hours, typically from 11:00 PM until 12:00 PM the following day. These patterns were deliberately embedded in the training process to allow the model to learn and generalize behaviors that would be critical for driving autoscaling decisions. In both test cases, the system was tested by stress testing with http requests on the running flask application. [13] [14] These two scenarios confirm that the ML-enhanced HPA system can effectively anticipate traffic trends and scale Kubernetes workloads proactively, offering significant improvements over traditional reactive autoscaling approaches.

A. Low Traffic Scenario

In the first scenario, the system was tested under conditions of low expected demand, particularly during the late-night and morning hours when traffic was expected to be minimal. During these times, the model correctly predicted reduced traffic. The HPA observed these metrics and scaled the deployment down accordingly. This demonstrated the system's ability to reduce resource usage and operational costs without manual intervention or real-time metric delays.

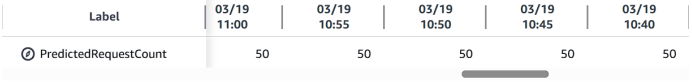

Fig. 10: Predicted traffic values at around 10:30 am

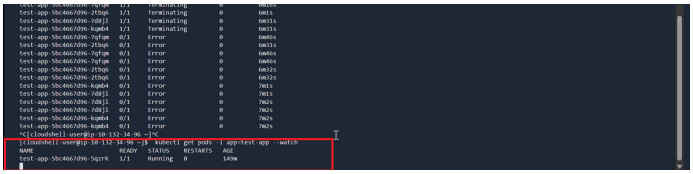

Fig. 11: System scaled down to 1 pod to save resources

B. High Traffic Scenario

The second testing scenario focused on weekend and afternoon surge handling. The goal was to verify whether the system could scale the application in advance of high traffic periods. The HPA responded to elevated predicted request counts at around 2 pm by scaling up the number of pods before the simulated rush began at 3 pm. This proactive scaling prevented performance degradation and ensured the application remained responsive during peak usage.

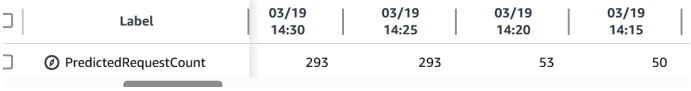

Fig. 12: Predicted traffic values at around 2:30 pm

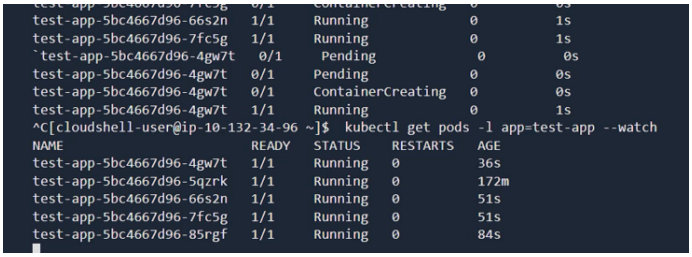

Fig. 13: HPA scaled up to get ready for predicted traffic surge at 3 pm

The test was considered successful, as none of the pods restarted or failed due to high CPU pressure during the stress test, indicating that the system scaled appropriately and maintained stability under load.

V. Conclusion and Future Work

A. Conclusion

This paper presented the development of a machine learning-based autoscaling solution for Kubernetes that proactively adjusts resources based on predicted traffic loads. By integrating Amazon SageMaker for traffic forecasting, CloudWatch for monitoring, and Amazon EKS for container orchestration, the system aimed to overcome the reactive nature of standard HPA. A synthetic dataset reflecting realistic usage patterns was used to train the model, allowing it to anticipate peak and low-demand periods with high accuracy.

Testing in both high and low traffic scenarios demonstrated the system's effectiveness: it scaled down to reduce resource waste during low demand and scaled up preemptively to maintain performance under heavy load. No pod failures occurred, indicating stability and resilience. Overall, the results support that ML-driven autoscaling enhances cost efficiency and responsiveness, offering a smarter alternative to traditional reactive methods in Kubernetes environments.

B. Future Work

While this paper successfully demonstrated the feasibility of predictive autoscaling using machine learning in Kubernetes, several areas remain open for future exploration and enhancement. First, a synthetic dataset with predetermined traffic patterns was used to train the current algorithm. To increase the accuracy and generalizability of the model, real-world traffic records from operational systems may be included in subsequent iterations. Better management of seasonality, anomalies, and multi-step prediction windows may also be possible by incorporating time-series forecasting models such as LSTM or Prophet. Lastly, the effectiveness of this predictive scaling system under increasingly complex, interdependent workload patterns might also be assessed by implementing it in a multiservice microservices environment. Additionally, this would require handling prediction at the service level instead of globally and improving metric granularity. By addressing these areas, future work can expand this system into a productiongrade, intelligent autoscaling solution that adapts to changing traffic, improves reliability, and reduces cloud costs across diverse Kubernetes workloads.

References

[1] K. Documentation, "Horizontal pod autoscaling," https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/, 2024, accessed: 2025-03-21.

[2] AWS, "Scaling kubernetes deployments with amazon cloudwatch metrics," https://aws.amazon.com/blogs/compute/scaling-kubernetes-deployments-with-amazon-cloudwatch-metrics/, 2022, accessed: 2025-03-20.

[3] ProjectPro, "Amazon sagemaker: Everything you need to know," https://www.projectpro.io/article/amazon-sagemaker/76, 2023, accessed: 2025-03-20.

[4] B. Sigoure, "Scaling kubernetes deployments using custom metrics," https://blog.cloudflare.com/scaling-kubernetes-deployments-using-custom-metrics/, 2023, accessed: 2025-03-21.

[5] AWS, "Amazon eks user guide," https://docs.aws.amazon.com/eks/latest/userguide/, 2024, accessed: 2025-03-20.

[6] shahrukhcoder357, "Deploying a flask app on amazon eks with load balancer as a service using docker image," https://medium.com/@shahrukhcoder357/deploying-a-flask-app-on-amazon-eks-with-load-balancer-as-a-service-using-docker-image-77219776696e, 2023, accessed: 2025-03-20.

[7] AWS, "Deploy container insights on amazon eks," https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/deploy-container-insights-EKS.html, 2023, accessed: 2025-03-20.

[8] C. Adamson, "Deploying machine learning models on aws: A guide to amazon sagemaker," https://medium.com/@christopheradamson253/deploying-machine-learning-models-on-aws-a-guide-to-amazon-sagemaker-ca50717bb341, 2023, accessed: 2025-03-20.

[9] AWS, "Train and deploy machine learning models on kubernetes with amazon sagemaker operators," https://aws.amazon.com/blogs/machine-learning/train-and-deploy-machine-learning-models-on-kubernetes-with-amazon-sagemaker-operators/, 2023, accessed: 2025-03-21.

[10] AWS.Documentations, "Integrating amazon sagemaker into kubernetes workflows using sagemaker operators," https://aws.amazon.com/blogs/machine-learning/integrating-amazon-sagemaker-into-kubernetes-workflows-using-sagemaker-operators/, 2022, accessed: 2025-03-21.

[11] Stack Overflow, "How to add custom metrics to pre-existing cloudwatch?" https://stackoverflow.com/questions/79394733/how-to-add-custom-metrics-to-pre-existing-cloudwatch, 2024, accessed: 2025-03-20.

[12] mabar, "Today i learned: Enabling horizontal pod autoscaler (hpa) in kubernetes," https://medium.com/mabar/today-i-learned-enabling-horizontal-pod-autoscaler-hpa-in-kubernetes-489126696990, 2022, accessed: 2025-03-20.

[13] saantoryuu, "Load testing using hey," https://dev.to/saantoryuu/load-testing-using-hey-c84, 2023, accessed: 2025-03-20.

[14] rakyll, "Hey - http load generator, apache benchmark (ab) replacement," https://github.com/rakyll/hey, 2023, accessed: 2025-03-20.