Simple and Explainable Machine-Learning based proactive Autoscaling for Kubernetes

Eileen Huang, Carnegie Mellon University
Harivallabha Rangarajan, Carnegie Mellon University
Hong Sng, Carnegie Mellon University
Jim (Chia-Tse) Shao, Carnegie Mellon University

***Abstract*—Kubernetes, the preeminent open-source platform for cloud resource management [1], provides reactive autoscaling out-of-the-box. Recent work on proactive autoscaling with LSTMs and Transformers have outperformed this default in resource allocation, but at the cost of the complexity, data-intensity, inference-time latency and opacity associated with large machine-learning (ML) models.**

We propose proactive autoscaling approaches focusing on two complementary goals: explainability and simplicity, demonstrating that the interpretable time-series forecasting model N-BEATS and classical ML methods like linear regression match the results of large ML models while remaining simple and explainable.

***Index Terms*—Machine Learning, Auto-scaling, Kubernetes**

I. Introduction

Cloud computing has revolutionized the management and deployment of applications by providing a powerful service abstraction. Instead of owning and maintaining physical infrastructure, organizations can access virtual resources over the Internet known as Infrastructure-as-a-Service (IaaS).

The "Cloud" allows for lower cost, greater scalability, and flexibility in compute due to economies of scale, but at the cost of additional complexity in resource management. Efficiently allocating compute and storage across dynamic workloads is difficult - resource mismanagement can lead to under-provisioning (causing performance bottlenecks) or over-provisioning (leading to wasted costs). The online algorithms for scaling aim to approximate the offline theoretical optimizations for when to scale horizontally, which refers to increasing and decreasing the number of compute units allocated based on the rate of client request.

Kubernetes, the leading cloud resource management platform, performs the allocation of lightweight compute units called containers through a rule-based process called Horizontal Pod Autoscaling (HPA). HPA scales the number of compute units (called 'Pods') based on metrics such as CPU utilization or Queries Per Second (QPS). However, such traditional reactive scaling methods can be suboptimal and lead to temporary service degradation due to delayed responses to fluctuating client demand. Proactive autoscaling addresses this limitation by predicting future workloads and preemptively scaling compute resources for higher throughput and lower response latency.

Author names arranged based on alphabetical orders.

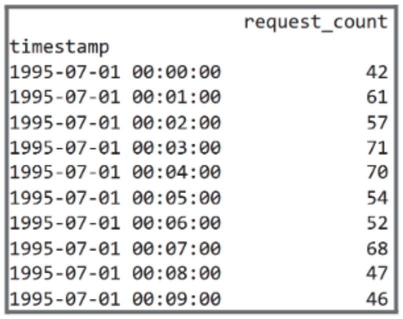
II. Related Works

Existing machine learning and deep learning-based approaches explore proactive autoscaling range from classical time-series models like ARIMA (Auto-Regressive Integrated Moving Average) to deep learning-based techniques such as bidirectional LSTMs and Transformers.

Bidirectional Long Short-Term Memory (Bi-LSTM) is a type of recurrent neural network (RNN) architecture, particularly useful for time-series prediction. It extends the traditional LSTM model by introducing two LSTM layers: one that processes input data in a forward direction (past to future) and another that processes data in a backward direction (future to past). This dual-layer setup enables Bi-LSTM to retain information from both past and future contexts, allowing it to capture temporal dependencies in both directions, which enhances prediction accuracy in complex time-series tasks. Dang-Quang and Yoo's Bi-LSTM-based autoscaling architecture [2] demonstrated higher accuracy and lower prediction error compared to ARIMA, and at the same time allowing for faster and more precise scaling in Kubernetes environments. More recent works have focused on techniques such as transformers that incorporate attention mechanisms.

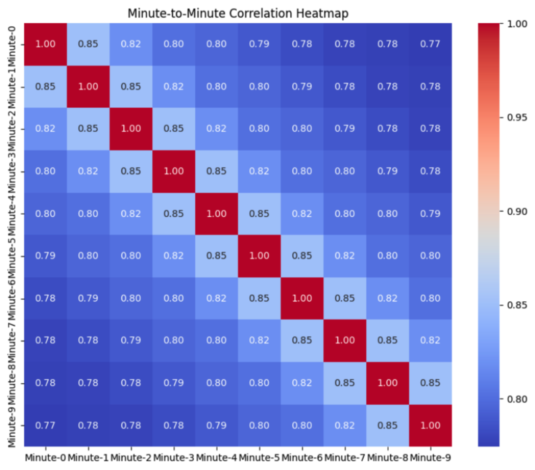
Transformers are a class of deep learning models originally developed for natural language processing (NLP) but have recently been adapted for time-series prediction. Unlike recurrent models such as LSTM or Bi-LSTM, which process data sequentially, transformers leverage an attention mechanism that enables them to process entire sequences at once. This approach allows transformers to capture relationships between data points over long sequences more effectively and efficiently, making them well-suited for time-series tasks where dependencies can span multiple time steps. Moreover, studies by Shim et al. [3] introduced transformer models as an alternative for handling long-sequence data, such as high-variability workload patterns observed in NASA and FIFA datasets, showing superior performance over Bi-LSTM and LSTM models due to their attention mechanisms.

Although these approaches demonstrate impressive performance as predictive agents, they lack the tenets of explainability and interpretability. A core focus of our work is on exploring simpler and more explainable models that match the performance of these larger and opaque techniques.



```
                           request_count
timestamp
1995-07-01 00:00:00                   42
1995-07-01 00:01:00                   61
1995-07-01 00:02:00                   57
1995-07-01 00:03:00                   71
1995-07-01 00:04:00                   70
1995-07-01 00:05:00                   54
1995-07-01 00:06:00                   52
1995-07-01 00:07:00                   68
1995-07-01 00:08:00                   47
1995-07-01 00:09:00                   46
```

Fig. 1. Dataset within minute scope



	Minute-0	Minute-1	Minute-2	Minute-3	Minute-4	Minute-5	Minute-6	Minute-7	Minute-8	Minute-9
Minute-0	1.00	0.85	0.82	0.80	0.80	0.79	0.78	0.78	0.78	0.77
Minute-1	0.85	1.00	0.85	0.82	0.80	0.80	0.79	0.78	0.78	0.78
Minute-2	0.82	0.85	1.00	0.85	0.82	0.80	0.80	0.79	0.78	0.78
Minute-3	0.80	0.82	0.85	1.00	0.85	0.82	0.80	0.80	0.79	0.78
Minute-4	0.80	0.80	0.82	0.85	1.00	0.85	0.82	0.80	0.80	0.79
Minute-5	0.79	0.80	0.80	0.82	0.85	1.00	0.85	0.82	0.80	0.80
Minute-6	0.78	0.79	0.80	0.80	0.82	0.85	1.00	0.85	0.82	0.80
Minute-7	0.78	0.78	0.79	0.80	0.80	0.82	0.85	1.00	0.85	0.82
Minute-8	0.78	0.78	0.78	0.79	0.80	0.80	0.82	0.85	1.00	0.85
Minute-9	0.77	0.78	0.78	0.78	0.79	0.80	0.80	0.82	0.85	1.00

Fig. 2. Minute-to-Minute Correlation Heatmap

III. Methodology

A. Dataset

The dataset used for this project is the NASA-HTTP logs, which contain detailed records of HTTP requests made to the NASA Kennedy Space Center WWW server over a two-month period [6]. This dataset supports our development of a predictive model for auto-scaling by providing a historical record of traffic patterns and request count. The dataset provides high-resolution, one-second granularity in request timestamps, which allows precise capture of time-based traffic fluctuations. This resolution is suitable for our goal of predicting minute-by-minute request counts for auto-scaling purposes, as it enables the model to learn both short-term spikes and regular patterns within the traffic data. The following images show the example dataset and the correlation between each minute, which is good for our machine learning models to capture temporal relationships.

The NASA-HTTP dataset consists of two separate log files: The first log captures all HTTP requests from July 1, 1995, 00:00:00 to July 31, 1995, 23:59:59, spanning a total of 31 days. The second log covers requests from August 1, 1995, 00:00:00 to August 7, 1995, 23:59:59, totaling 7 days. In this two-week dataset, a total of 3,461,612 requests were recorded. An important gap in the data occurs from August 1, 1995, 14:52:01 until August 3, 1995, 04:36:13, as the server was temporarily offline due to Hurricane Erin.

B. Machine Learning

In our approach to building a predictive model for autoscaling, we experimented with several complex deep learning models to predict the count of requests of the next minute. The models use the previous 10 minute request counts as input and output the next minute request count. We also sliced our train and test dataset using the same cut-off point as other papers. The models we implemented included Informer, Autoformer, N-Beats [5], and DeepAR, each designed to capture complex temporal patterns in the data. We evaluated each model based on mean squared error (MSE) and model efficiency, including training and inference times.

After the experiments are done, we will overlap this graph with results from NBeats / Linear Regression.

C. Evaluation Metrics

To assess the performance and suitability of our ML model for autoscaling in a Kubernetes environment, we will use the following evaluation metrics which also align with the benchmark we would like to compare with:

Mean Squared Error (MSE): This metric measures the average squared difference between the actual and predicted request counts. It penalizes larger errors more heavily, helping to identify how well the model captures general request patterns.

Root Mean Squared Error (RMSE) on request count: RMSE provides a more interpretable measure of error by taking the square root of MSE, offering an understanding of the average error magnitude in the same units as the request counts. Lower RMSE values indicate better predictive accuracy.

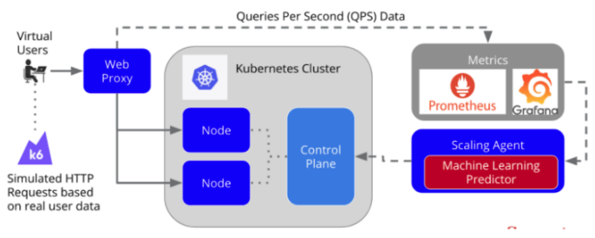
Mean Absolute Error (MAE) on request count: MAE calculates the average absolute difference between actual and predicted values, providing an intuitive measure of error by treating all deviations equally. This metric is less sensitive to large errors compared to MSE and RMSE.

R-squared (R^2) on request count: This metric evaluates the proportion of variance in the request counts that the model can explain, with a value closer to 1 indicating higher predictive strength and model reliability.

Model Prediction Time: In a Kubernetes autoscaling context, prediction latency is critical. We measure the time required for the model to generate each request count prediction, ensuring that predictions are fast enough for real-time scaling.

Number of Pods: To gauge the practical impact of our autoscaling strategy, we track the number of pods generated by the model's predictions. This helps us assess whether the model effectively scales resources based on predicted request load and maintains an efficient resource allocation.

Auto-Scaling Matrix: To provide a holistic view on our overall system performance, we need a more specific metric for determining how good our auto-scaling policy has improved compared to the non-scaling policy. The provisioning accuracy



Queries Per Second (QPS) Data
Virtual Users
Web Proxy
Kubernetes Cluster
Node
Node
Control Plane
Metrics
Prometheus
Grafana
Scaling Agent
Machine Learning Predictor
Simulated HTTP Requests based on real user data

Fig. 3. System Architecture

θ_O and θ_U describes the relative amount of resources being over-provisioned or under-provisioned.

$$\theta_U[\%] = \frac{100}{T}\sum_{t=1}^{T}\frac{max(r_t - p_t, 0)}{r_t} \tag{1}$$

$$\theta_O[\%] = \frac{100}{T}\sum_{t=1}^{T}\frac{max(p_t - r_t, 0)}{r_t} \tag{2}$$

IV. Experimental Setup

A. Architecture

In order to test our predictions, we built the testing infrastructure as figure 3. We deploy a mock application on a Kubernetes cluster, which contains several compute nodes that serve as backend servers for the NASA website. A cluster is simply a number of nodes managed by a control plane that handles scaling them horizontally.

Apache K6, an open-source load testing tool, is used to simulate client HTTP requests based on the NASA dataset. In this process, virtual users, an abstraction similar to threads, periodically send HTTP requests to our mock application based on the request rate of the current timestamp.

These requests are received by a web proxy, which performs two operations: first, it acts as a load balancer and routes these requests to one of the current compute nodes, where a mock application returns a 200-OK response; secondly, it sends QPS metric data to our metrics framework, which includes a Prometheus database and a Grafana visualization dashboard.

These QPS metrics are then used as input data to the machine learning predictor within our scaling agent, which then outputs the estimated number of compute nodes required. With this prediction, we notify the control plane to scale the number of nodes currently being provisioned. Therefore, the scaling agent autonomously handles autoscaling based on real-time client requests.

B. Implementation

1) Development Environment and Infrastructure: The experimental setup runs in GitHub Codespaces using a k3d (lightweight Kubernetes) cluster. The environment is automatically provisioned through a custom Docker container with all necessary development tools and dependencies. A single-server Kubernetes cluster is configured with exposed ports for monitoring (Prometheus: 30000, Grafana: 32000) and application services (MockApp: 30080, Heartbeat: 31080). The setup process is automated through initialization scripts that handle everything from creating a local Docker registry to deploying core services.

2) Application Stack and Monitoring: The core application consists of an REST API service for load generation. The monitoring stack includes Prometheus for metrics collection (scraping every 5 seconds), and Grafana for visualization for log aggregation. The web application is configured with specific resource limits (CPU: 1000m, Memory: 256Mi).

3) Load Testing Infrastructure: The load testing uses Grafana k6 to simulate realistic traffic patterns based on NASA web server logs. The system creates virtual users (VUs) equal to the maximum expected load, with each VU having a probability of sending requests that match historical data patterns.

4) Monitoring and Metrics Integration: The monitoring system collects metrics from multiple sources including application endpoints, Kubernetes system metrics, and custom metrics. Grafana dashboards are pre-configured to visualize application performance metrics and load test results.

5) Scaling Agent Monitoring and Logic: The scaling agent implements an automated scaling system that uses machine learning to predict and adjust the number of application replicas. The agent integrates with Kubernetes through the client.AppsV1Api() and monitors the application metrics using Prometheus, querying the request rate through the internal cluster DNS. The system maintains a 10-minute sliding window of request history using a deque data structure.

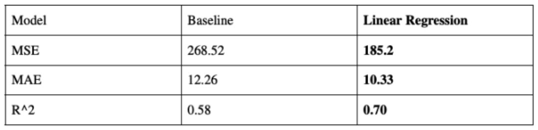
The RequestPredictor class implements a linear regression model that predicts the required number of replicas based on historical request patterns. The model takes a sequence of 10 minutes of request counts as input and predicts the next minute's request volume. The prediction is then converted to the number of needed replicas by assuming that each replica can handle 100 requests per minute (configurable). The model is pre-trained and loaded from a pickle file, making real-time predictions based on the observed traffic patterns.

V. Results

Despite the sophisticated architecture of other complex models (such as transformers), we found that a simpler linear regression model achieved comparable performance, with an MSE of 186.8. This performance was similar to the best performing deep learning models (e.g., N-Beats with an MSE of 185), but linear regression required significantly less computational time for both training and testing. This simplicity and efficiency make linear regression an appealing choice for our autoscaling task, where speed and resource efficiency are critical.

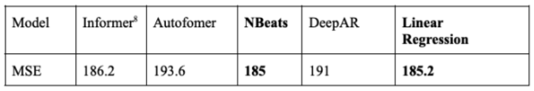
A. Comparison with other models

Figure 4 shows the comparison of Mean Squared Error (MSE), Mean Absolute Error (MAE), and $R\hat{2}$ Score with a baseline model that directly uses the request count from the previous minute as the request count from the current minute. We can see that the linear regression model outperformed



Model	Baseline	Linear Regression
MSE	268.52	**185.2**
MAE	12.26	**10.33**
R^2	0.58	**0.70**

Fig. 4. Baseline vs. Linear Regression



Model	Informer[8]	Autofomer	NBeats	DeepAR	Linear Regression
MSE	186.2	193.6	**185**	191	**185.2**

Fig. 5. Explored Model Performance Comparison

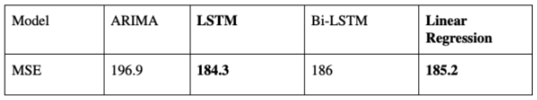
all metrics, demonstrating its capability to capture temporal trends.

Figure 5 summarizes the Mean Squared Error (MSE) scores across the Informer model we reproduced and other models we explored. Figure 6 demonstrates the MSE scores with other commonly used machine learning models in autoscaling from related works.

The prediction time of each sample with the linear regression model is 0.0014 seconds, and Figures 5 and 6 show that the linear regression model has similar results to the best one. Thus, choosing linear regression provides a balance between accuracy and efficiency, making it well-suited for deployment in a Kubernetes environment where real-time scaling decisions are important.

B. Explainable Feature Relationship

The linear regression model provided explainability by revealing how request counts from the previous 10 timestamps influence the prediction of the current request count. The bar chart in Figure 7 illustrates the magnitude of the regression coefficients for each time step, with $T-1$ (the most recent request count) showing the highest coefficient, indicating that it has the most significant impact on the predictions. As time steps move further into the past ($T-10$ to $T-2$), their influence diminishes, suggesting that recent traffic patterns are more critical in predicting the next request count. Thus, we can see that the linear regression model is explainable and trustworthy to understand scaling decisions in an autoscaling framework.



Model	ARIMA	LSTM	Bi-LSTM	Linear Regression
MSE	196.9	**184.3**	186	**185.2**

Fig. 6. Model Performance compared to other papers

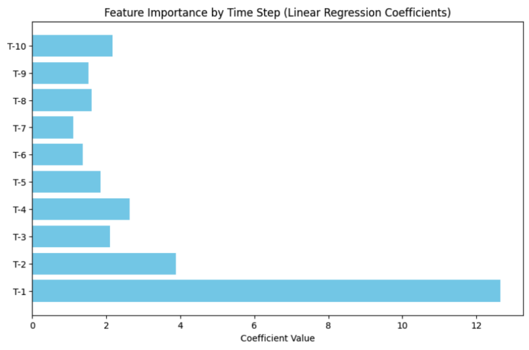


Fig. 7. Feature Importance by Time Step

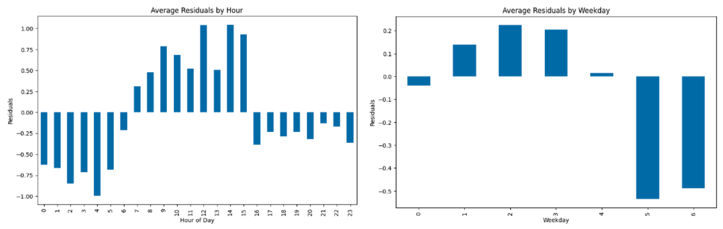


Fig. 8. Residuals by different Time Granularity

C. Residual analysis

The residual analysis revealed distinct patterns in the model's predictions. Positive residuals, where actual values exceeded predictions, were observed predominantly during midday hours and on weekdays, indicating under-predictions in these periods. Conversely, negative residuals, where predictions exceeded actual values, occurred mostly at night and on weekends, suggesting over-predictions during these times. These trends highlight areas where the model could be refined to improve accuracy in the future, such as incorporating additional features or adjusting the model to better capture temporal variations in traffic patterns.

D. Performance

Figure 9 shows the actual and predicted trends. We can see that the linear regression model is keeping up with the actual trend, proving its ability to capture the temporal trend. However, it still needs improvement in some peaks or lows, which is what we could improve in the future.

Figure 10 translates the ML model evaluation to a more systematic interpretation. Linear Regression Model performs worse in both under-provision and over-provision ratios, which means the system is potentially not cost efficient by choosing this model. We believe the reason this is not revealed in MAE is because the low request rate will have an amplification effect on equation 1 and 2.

E. System Performance

Similarly to the chosen benchmarks, we aim to provide a visualization of the results as Figure 11.

Backhand calculations show that proactive machine-learning based autoscaling can be about 20% closer to the optimal number of compute units (pods) required at any one time, resulting in both lower client request latency and a 20% decrease in cost associated with provisioning cloud resources.

This is a significant cost reduction that scales linearly with the number of resources deployed by a specific application,

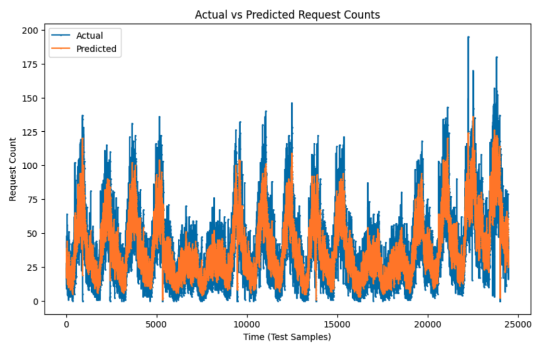
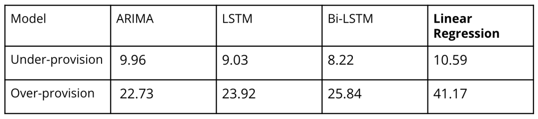


Fig. 9. Actual vs. Predicted Request



Model	ARIMA	LSTM	Bi-LSTM	**Linear Regression**
Under-provision	9.96	9.03	8.22	10.59
Over-provision	22.73	23.92	25.84	41.17

Fig. 10. Comparison with Auto-scaling Metrics

and thus has considerably large potential for further cost savings.

VI. Conclusion

In conclusion, our findings demonstrate the viability of simple and explainable machine learning-based proactive scaling methods.

Firstly, they present a significant upgrade over the Kubernetes default HPA, offering a 20% more efficient cloud resource allocation, which translates to reductions in both over-provisioning (thus reducing cost) and under-provisioning (thus reducing client request latency).

Furthermore, our methods also improve on other existing transformer or deep-learning based machine learning methods because they meet our goals of explainability and simplicity, showing that the interpretable time-series forecasting model N-BEATS and classical ML methods like linear regression can match the results of large ML models without the complexity and inference-time overhead of the latter. However, it is important to note that our models tend to be less resistant to outliers during spikes in demand compared to these larger ML models, perhaps because of their simplicity.

All in all, our research shows that it is possible to achieve a balance between optimality and complexity; providing an efficient machine-learning based approach to the functional goal of autoscaling cloud resources without sacrificing the non-functional goals of explainability and simplicity.

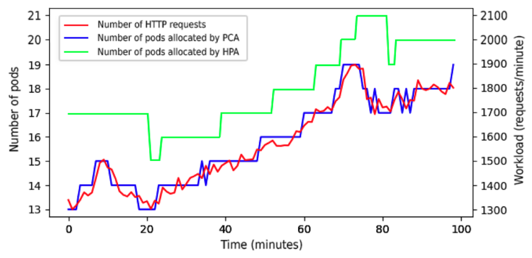


Fig. 11. Target Benchmark from [3]

References

[1] CNCF SURVEY 2020.

[2] Dang-Quang, N. -M., & Yoo, M. (2021). Deep Learning-Based Autoscaling Using Bidirectional Long Short-Term Memory for Kubernetes. Applied Sciences, 11(9), 3835. https://doi.org/10.3390/app11093835.

[3] S. Shim, A. Dhokariya, D. Doshi, S. Upadhye, V. Patwari and J. -Y. Park, ”Predictive Auto-scaler for Kubernetes Cloud,” 2023 IEEE International Systems Conference (SysCon), Vancouver, BC, Canada, 2023, pp. 1-8, doi: 10.1109/SysCon53073.2023.10131106.

[4] Boge, F.J. Two Dimensions of Opacity and the Deep Learning Predicament. Minds & Machines 32, 43–75 (2022). https://doi.org/10.1007/s11023-021-09569-4

[5] N-BEATS: Neural basis expansion analysis for interpretable time series forecasting. https://arxiv.org/abs/1905.10437

[6] Two Month's Worth of All HTTP Requests to the NASA Kennedy Space Center. Available online: fttp://ita.ee.lbl.gov/html/contrib/NASA-HTTP.html

[7] Herbst, N., more: Ready for Rain? A View from SPEC Research on the Future of Cloud Metrics. CoRR abs/1604.03470 (2016)

VII. Appendix

A. *Relevance to 15-712 Course Goals*

Uses machine learning to provide efficient, explainable and simple improvements to resource allocation in distributed systems and applications.

B. *Outline of work*

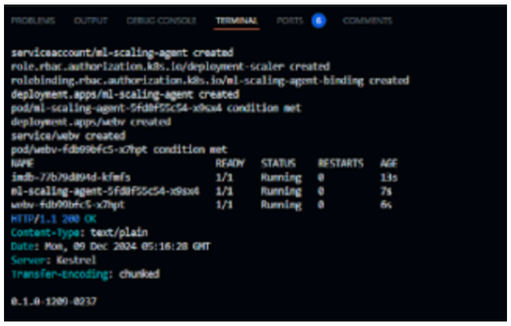
The project involves two major parts: machine learning model development and testing within a Kubernetes environment. Below is a breakdown of the tasks:

1) Eileen: Data Preprocessing: Preprocess and aggregating the NASA-HTTP data by minutes to create a time series suitable for training. Model Implementation and Training: Developing the ML model, using both complex and simpler models, and training it on the prepared data. Performance Evaluation: Evaluating model performance using metrics such as MSE, RMSE, and R² to identify the optimal model for integration.

2) Hari: Configuration Tuning: Setting parameters like cool-down time and fine-tuning the provisioning metrics to match scaling needs. Metric Provisioning Testing: Ensuring that metrics are appropriately captured and that they align with autoscaling requirements. Model Integration: Integrating the trained ML model to provide scaling predictions for K8s pods in real time.

3) Hong: Kubernetes Setup: Setting up the K8s cluster and ensuring the Kubernetes environment is ready for autoscaling tests. Load Testing: Implementing load testing tools to simulate varying levels of traffic for testing autoscaling behavior according to the chosen datasets Telemetry: Using Prometheus / Grafana to capture system metrics as input for our autoscaler + evaluate autoscaler performance Deployment: running the toy web application on the kubernetes cluster

Fig. 12. Experiment Snapshot: Load testing and pulling metrics



```
serviceaccount/ml-scaling-agent created
role.rbac.authorization.k8s.io/deployment-scaler created
rolebinding.rbac.authorization.k8s.io/ml-scaling-agent-binding created
deployment.apps/ml-scaling-agent created
pod/ml-scaling-agent-5fd8f55c54-x9sx4 condition met
deployment.apps/webv created
service/webv created
pod/webv-fdb99bfc5-x7hpt condition met
NAME                               READY   STATUS    RESTARTS   AGE
imdb-77b79d894d-kfmfs              1/1     Running   0          13s
ml-scaling-agent-5fd8f55c54-x9sx4  1/1     Running   0          7s
webv-fdb99bfc5-x7hpt               1/1     Running   0          6s
HTTP/1.1 200 OK
Content-Type: text/plain
Date: Mon, 09 Dec 2024 05:16:28 GMT
Server: Kestrel
Transfer-Encoding: chunked

0.1.0-1209-0237
```

Fig. 13. Experiment Snapshot: Environment Setup

4) Jim: HAProxy Setup: Configuring HAProxy to simulate and manage HTTP request traffic to the application. Toy Web Application: Developing a simple web application that sends acknowledgments for received requests, allowing for meaningful load testing and response tracking.