High-Level Information and Metrics-Aware Kubernetes Autoscaling for Microservice Application

Brianaldo Phandiarta
School of Electrical Engineering and Informatics
Bandung Institute of Technology
Bandung, Indonesia
13520113@std.stei.itb.ac.id

Achmad Imam Kistijantoro
School of Electrical Engineering and Informatics
Bandung Institute of Technology
Bandung, Indonesia
imam@staff.stei.itb.ac.id

***Abstract*—This research presents a proactive autoscaling system for Kubernetes that improves application availability by directly incorporating high-level metrics, specifically Service Level Objective (SLO) tail latency, and high-level information, specifically historical traffic patterns. The system leverages the PatchTST time series model for traffic forecasting and a neural network regression model for predicting service tail latency. Replica counts are adjusted based on the ratio of tail latency to the SLO threshold. Experimental results demonstrate that while the tail latency still exceeds the SLO threshold, this approach improves application availability by 0.0411%, from 99.7402% to 99.7813%, and reduces the average tail latency from 302.05 ms to 277.62 ms. However, reliance on tail latency ratios for replica adjustment introduced some instability, indicating a need for further refinement. This study underscores the potential of incorporating predictive analytics and high-level metrics into Kubernetes autoscaling strategies to optimize performance.**

Keywords—Kubernetes; proactive autoscaling; traffic patterns, tail latency;

I. Introduction and Related Works

The rapid advancement of cloud computing technology has significantly transformed how organizations manage data and deploy applications. Cloud computing services, including Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS), are central to modern IT operations. Among these, PaaS, which leverages containerization technologies, is particularly beneficial for deploying applications that adopt a microservices architecture. In this context, Kubernetes, developed by Google, is a widely adopted container orchestration solution.

Autoscaling is commonly used on Kubernetes to adjust computing resources dynamically. Kubernetes' autoscaling feature enables resource allocation based on application demand. However, traditional Kubernetes autoscaling is reactive, responding only after demand changes, which can result in resources being unprepared when they are needed. Proactive autoscaling is essential to address this issue. We can predict future resource needs by analyzing high-level application-specific information, such as historical traffic patterns. Therefore, it is crucial to find an approach to transform and utilize this high-level information, turning it into actionable insights that guide scaling decisions and ensure resources are allocated in advance to meet anticipated demand.

Traditional Kubernetes autoscaling primarily relies on low-level metrics, such as CPU and memory usage, which may only partially capture the application's performance needs. High-level metrics, like Service Level Objectives (SLOs), are crucial as they more directly reflect service quality. However, the current Kubernetes autoscaling mechanisms do not fully integrate these high-level metrics, leading to the need to map high-level metrics to low-level metrics. Mapping high-level metrics, such as SLOs, to low-level metrics, like CPU and memory usage, is a complex process, especially in a microservices architecture. Each microservice may have different resource requirements even when they share the same SLOs, making it challenging to develop a one-size-fits-all approach. This complexity necessitates more sophisticated approaches.

Several studies have explored various approaches to autoscaling in Kubernetes and cloud environments. Dang-Quang and Yoo [1] developed an autoscaling system using Bidirectional Long Short-Term Memory (Bi-LSTM) to forecast future HTTP request loads within applications. Bartelucci et al. [2] investigated high-level metrics, specifically the request latency SLO, for autoscaling in Kubernetes. Their findings showed that SLO-aware autoscaling could provide more responsive and effective scaling, particularly in reducing latency compared to traditional Horizontal Pod Autoscaler (HPA) methods. Rahman and Lama [3] focused on predicting the end-to-end tail latency in microservices, where the complexity of request handling forms a Directed Acyclic Graph (DAG). They found that including virtual machine (VM)-level metrics alongside pod-level metrics improved prediction accuracy, with deep neural networks (NN) outperforming other regression models. Furthermore, they used these models to determine the optimal number of replicas to meet latency targets.

While these studies have advanced autoscaling strategies by incorporating high-level metrics and information, challenges remain in effectively utilizing high-level metrics and application-specific information. This research addresses these challenges by developing a proactive autoscaling system that integrates high-level metrics like SLOs and information like historical traffic patterns. The goal is to provision just before the resource is needed and ease autoscaling decision-making by utilizing high-level metrics.

The remainder of this paper is organized as follows: Section II and III presents the proposed solution. Section IV details the implementation of the system, including the experimental setup. Section V discussed the results compared to HPA. Finally, Section VI concludes the paper with a summary of the findings and potential future work.

II. High-Level Information and Metrics-Aware Autoscaling

The proposed solution addresses two critical aspects of autoscaling: the need for proactive scaling based on predictive models and the direct use of high-level metrics for decision-making.

A. Traffic Patterns as the High-Level Information

Proactive autoscaling is only achievable with the implementation of a forecasting model that can predict future resource demands. The Data-Information-Knowledge-Wisdom (DIKW) hierarchy underscores the value of historical data in this process. By analyzing historical data, actionable information can be derived. Since historical data is intrinsically linked to time, a time series model is the perfect tool for this purpose, enabling us to predict future resource demands and implement proactive autoscaling.

Due to their predictability and stability, traffic patterns were selected as the primary high-level information for guiding autoscaling decisions. Unlike other metrics, such as CPU usage, which can fluctuate because of scaling actions, traffic patterns remain consistent and are unaffected by these changes. This stability makes traffic patterns ideal for time series modeling, as the time series model works by detecting patterns from previous data. By anticipating traffic increases, the system can proactively scale resources, avoiding the delays and disruptions associated with reactive scaling strategies.

B. Tail Latency as the High-Level Metric

Latency, particularly tail latency, was chosen as the key high-level metric for autoscaling due to its significant impact on user experience and its sensitivity to changes in resource availability. Tail latency, which represents the worst-case response times experienced by users, provides a more critical measure than average latency, as it highlights the performance experienced by the most affected users. Percentile-based measurements of latency are preferred over mean values because they offer a more accurate representation of service quality across all users, especially in identifying performance issues that could affect a subset of users.

To address the complexity of mapping high-level metrics like Service Level Objectives (SLOs) to low-level metrics such as CPU and memory usage, the proposed solution directly utilizes high-level metrics like tail latency as the basis for scaling decisions. This approach simplifies the decision-making process by eliminating the need for complex mappings between different types of metrics.

Incorporating proactive autoscaling based on this approach requires predicting tail latency based on forecasted traffic. However, predicting tail latency is challenging due to its non-linear behavior under varying system conditions. Tail latency can spike unpredictably as CPU usage increases, making traditional mathematical models insufficient. To overcome this challenge, the proposed solution employs a regression model to predict tail latency using the forecasted traffic and current system conditions.

C. Autoscaling Strategy

This solution's autoscaling strategy is based on the HPA principles, enhanced by using predicted latency values. The system uses the formula 1 to determine the required number of replicas. Where R_{new} is the target number of replicas, R_{old} is the current number of replicas, L_q is the current latency at the q-th percentile, θ_{latency} is the target latency threshold, and R_{max} is the maximum allowed number of replicas.

$$R_{\text{new}} = min\left(\left\lceil R_{\text{old}} \times \frac{\left\lfloor \frac{L_q}{\theta_{\text{latency}}} \times 100 \right\rfloor}{100} \right\rceil, R_{max}\right) \quad (1)$$

III. Autoscaling System Architecture

The autoscaling system follows the Monitor-Analyze-Plan-Execute (MAPE) loop architecture. The flow begins with the system monitoring relevant metrics from the microservices, such as CPU usage at both the pod and VM levels, the number of replicas per microservice, and tail latency. The system forecasts future traffic at specified intervals based on the collected traffic metrics. It then predicts tail latency using the traffic forecasts and other gathered metrics. Finally, the system makes scaling decisions by adjusting the number of replicas according to the predicted latency. This loop continuously repeats while giving several cooling periods to mitigate isolation issues, ensuring the system dynamically responds to changing conditions in real time.

The system consists of several key components, which are integrated into a Kubernetes cluster alongside the microservices. These components can be seen in Fig. 1. The main components of the autoscaling solution include the Autoscaler system and the Monitoring component. The Monitoring component is responsible for collecting crucial metrics, including CPU usage at the pod and VM levels, the number of replicas for each microservice, and tail latency. Based on the kube-prometheus-stack, this component ensures that all necessary data is gathered to feed into the predictive models and decision-making processes. The Autoscaler system itself consists of.

1) TrafficForecasterModel Component: This component uses time series models to forecast future traffic for each service within the microservices architecture. It predicts traffic patterns by analyzing historical traffic data, which is critical for anticipating future resource needs and making proactive scaling decisions. PatchTST [4] is chosen as the time series model because of its result on the multivariate data [5], as we need to forecast each microservice's traffic.

2) LatencyPredictorModel Component: This component predicts tail latency using the traffic forecasts generated by the TrafficForecasterModel component and the application's

current resource usage metrics. It employs an estimated tail latency using a regression model based on a deep neural network [3].

3) MetricsFetcher Component: This component serves as the interface for retrieving the metrics collected by the Monitoring component. It provides the necessary data for their prediction to the TrafficForecasterModel and LatencyPredictorModel components.

4) ResourceManager Component: This component manages the computational resources of the microservices by adjusting the number of replicas.

5) Controller Component: This component is the central component that orchestrates the autoscaling process. It connects and coordinates the other components—MetricsFetcher, TrafficForecasterModel, LatencyPredictorModel, and ResourceManager—ensuring the system operates cohesively. The Controller oversees the entire MAPE loop, making real-time scaling decisions based on the data and predictions provided by the other components.

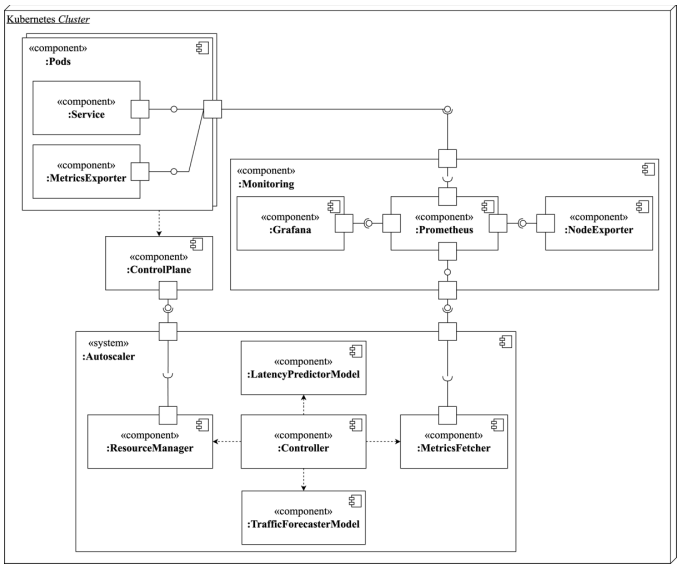

Fig. 1. System Autoscaling Component Diagram

IV. IMPLEMENTATION

The implementation was conducted on a virtual machine with the following specifications: x86/64 architecture, 8 cores, 16 vCPUs, 16 GB RAM, and an 80 GB boot disk, running Ubuntu 20.04. The Kubernetes cluster was deployed using Kubernetes v1.30.0, Minikube v1.33.0, and Docker v26.0.1. Testing was performed on a microservices architecture generated using the µBench tool [6], simulating the Sock Shop application [7], a small to medium-sized microservices application with services like Frontend, Catalogue, Order, Payment, User, Shipping, and Queue. Each service was configured for CPU-intensive tasks and probabilistic service calls using 200 ms of vCPU and 200 MB of memory. Traffic simulation was based on the Online Shopping Store—Web Server Logs dataset [8] that was processed with Locust. The implementation is done with Python 3.11.9, with the core component implementation details as follows.

A. TrafficForecasterModel Component

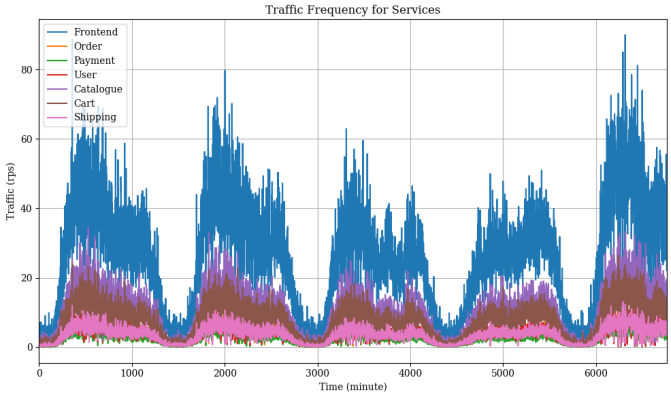

Fig. 2. Traffic Pattern

The implementation of the TrafficForecasterModel component begins with data collection. Data is gathered through simulation. The PatchTST model is implemented using the transformers library [9]. Training the PatchTST model involves configuring 40 parameters. However, only a few key configurations are crucial as they depend on the data patterns seen in Fig. 2. These key configurations are presented in Table I. The context length configuration, representing the amount of historical data used for prediction, is set to 1 day (24 hours x 60 minutes). The patch length and patch stride configurations, which determine the segment length and stride between segments, are chosen to optimize the learning process through iteration. The number of input channel configurations corresponds to the number of input variables, which is 7, matching the number of microservices. The remaining parameters use the default values provided by the library. The model is trained using three days of data, validated on one day of data, and tested using the rest of the data.

TABLE I. PATCHTST CONFIGURATION

Configuration	Value
Context length	1440
Patch length	15
Patch stride	5
Number input channels	7

The model is evaluated using the Mean Squared Error (MSE) and Mean Absolute Error (MAE). The model achieved an average MSE of 23.4091 and an average MAE of 2.5093 across all services, with details in Table II and the illustration in Fig. 3. These relatively low MSE and MAE values indicate that the model effectively captures and predicts the traffic patterns for the microservices.

TABLE II. TRAFFICFORECASTERMODEL TEST RESULT

Microservice	MSE	MAE
Frontend	121.5753	7.9937
Order	2.8081	1.996
Payment	1.3998	0.8457
User	3.0104	1.2389
Catalogue	21.0776	3.2956
Cart	11.4444	2.4115
Shipping	2.5483	1.1473

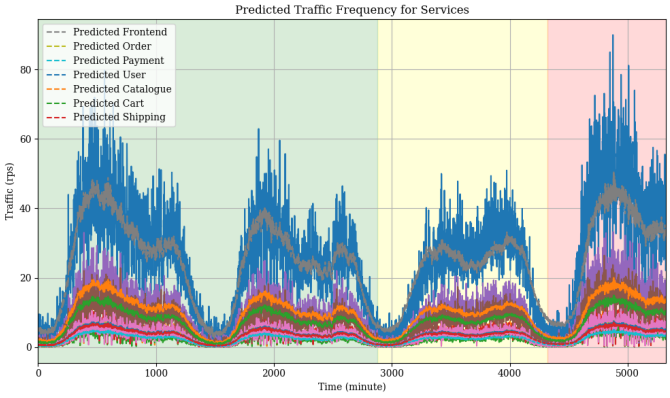

Fig. 3. Forecasted Traffic

B. *LatencyPredictorModel Component*

Like the TrafficForecasterModel, implementing the LatencyPredictorModel component begins with data collection and preparation. Data for model training is collected through load testing on the microservices, manually increasing the number of pod replicas while simultaneously adding workload. This data is aggregated per minute and collected from the Monitoring component. Feature engineering is then applied, creating a traffic-per-replica feature by dividing traffic by the number of pod replicas, allowing the model to adapt to higher traffic levels not present in the training data. A log-transformed tail latency feature is also created to reduce model sensitivity to outliers. The prepared data is then randomly split into training (80%), validation (10%), and test (10%) datasets.

The model is implemented using the TensorFlow library [10]. It consists of 1 input layer, 5 hidden layers, and 1 output layer. The input layer has 29 neurons corresponding to the input features: node-level CPU usage, 7 pod replica counts, 7 aggregate CPU usage values, 7 aggregate traffic values, and 7 traffic-per-replica features. The hidden layers have 32, 64, 128, 256, and 32 neurons, respectively, each using the ReLU activation function. The output layer contains 7 neurons, corresponding to the logarithm of the tail latency for each microservice. Model training is performed using the mean absolute error loss function to reduce sensitivity to outliers, and the process runs for 100 epochs with early stopping if no improvement is seen after 10 epochs.

The model achieved an average Mean Absolute Percentage Error (MAPE) of 2.07% and an R^2 value of 0.9801, indicating that the regression model can accurately predict tail latency across the microservices. The details of the evaluation can be seen in Table III. However, it is noteworthy that some services, specifically Payment and Shipping, exhibited lower R^2 values despite having very small MAPE. This discrepancy is attributed to the lack of variability in the training data for these services, which can limit the model's ability to generalize well in those cases.

TABLE III. LATENCYPREDICTORMODEL TEST RESULT

Microservice	MAPE	R^2
Frontend	1,71%	0,9052
Order	3,30%	0,7510
Payment	0,00%	0,2563
User	3,47%	0,7157
Catalogue	2,63%	0,9094
Cart	3,18%	0,8153
Shipping	0,00%	0,2432

C. *Controller Component*

The implementation of the Controller component follows the algorithm outlined in Algorithm I. The algorithm starts with initialization to ensure the autoscaling system is connected to Prometheus and Kubernetes. The scaling process then iterates through several steps: fetching all metrics using the fetchMetrics function, forecasting traffic for each service with the forecastTraffic function, predicting tail latency using the predictLatency function, determining the number of replicas for each service with the scaleStrategy function based on Equation III.1, scaling each service with the scale function, and performing a cooling down phase with the coolingDown function. The Controller component also allows for the flexibility to adjust thresholds for each microservice.

Algorithm I: Controller Component Algorithm

```
initialization;                          /* Connect to Prometheus and Kubernetes */
while true do
    metrics ← fetchMetrics();
    forecastedTraffic ← forecastTraffic(metrics[traffic])
    predictedLatency ← predictLatency(metrics, forecastedTraffic);
    foreach service ∈ microservices do
        targetReplica ← scaleStrategy(service, predictedLatency[service]);
        scale(service, targetReplica);
    end
    coolingDown();
end
```

V. RESULT

The system was tested by comparing the Kubernetes Horizontal Pod Autoscaler (HPA) with the developed autoscaling system. The tests were conducted over a 6-hour load test. The Kubernetes HPA was configured with a CPU utilization threshold of 70%, while the developed autoscaling system was configured with latency percentile thresholds of 250 ms, 150 ms, 100 ms, 100 ms, 100 ms, 100 ms, and 100 ms for the Frontend, Order, Payment, User, Catalogue, Cart, and Shipping services, respectively.

Testing with the HPA resulted in an availability of 99.7402%, whereas testing with the developed autoscaling system achieved an availability of 99.7813%, representing an improvement of 0.0411%. As shown in Fig. 4, the autoscaling system was able to forecast traffic trends and proactively scale

resources, as observed particularly at the 300-minute mark for the Frontend service in Fig. 7. The proactive nature of the autoscaling system is further supported by its ability to predict spikes in tail latency, as depicted in Fig. 5. This resulted in a more stable tail latency compared to the HPA, as illustrated in Fig. 6. However, oscillations were observed after scaling, as the autoscaling system attempted to reduce the number of replicas when the tail latency significantly dropped below the threshold, as seen in Fig. 7.

Despite the system's proactive capabilities, Fig. 5 shows that the autoscaling system struggled to predict tail latency accurately, often underestimating the actual latency values. This led to situations where the autoscaling system's configured thresholds were not met. Nonetheless, there was a reduction in average tail latency from 302.05 ms to 277.62 ms, as shown in Fig. 6.

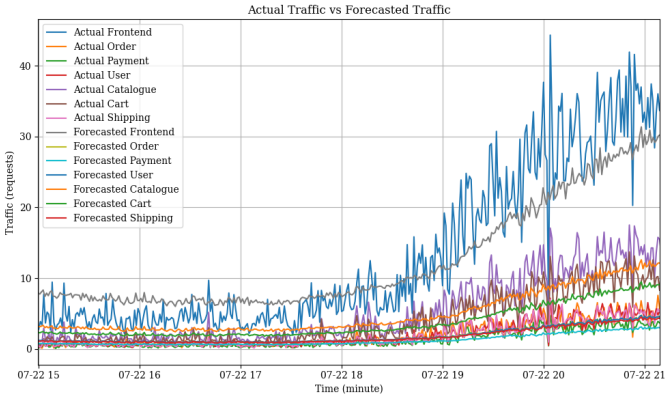

Fig. 4. Actual and Forecasted Traffic

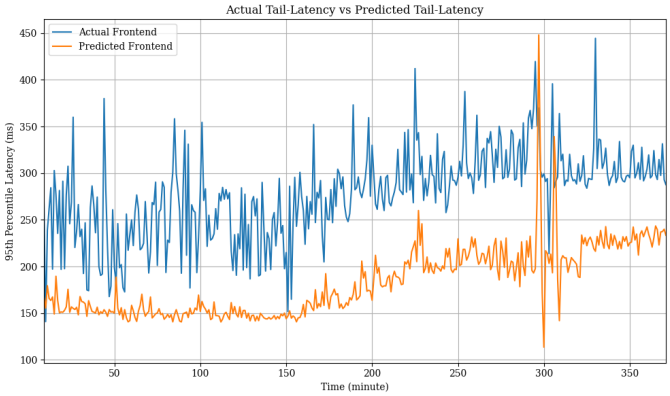

Fig. 5. Actual and Predicted 95th Tail Latency (Frontend)

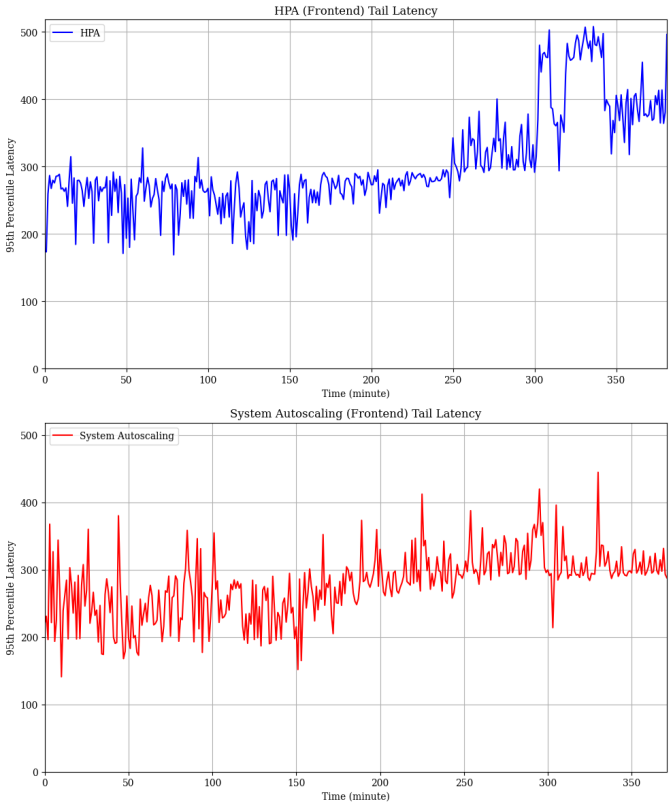

Fig. 6. HPA and System Autoscaling 95th Tail Latency (Frontend)

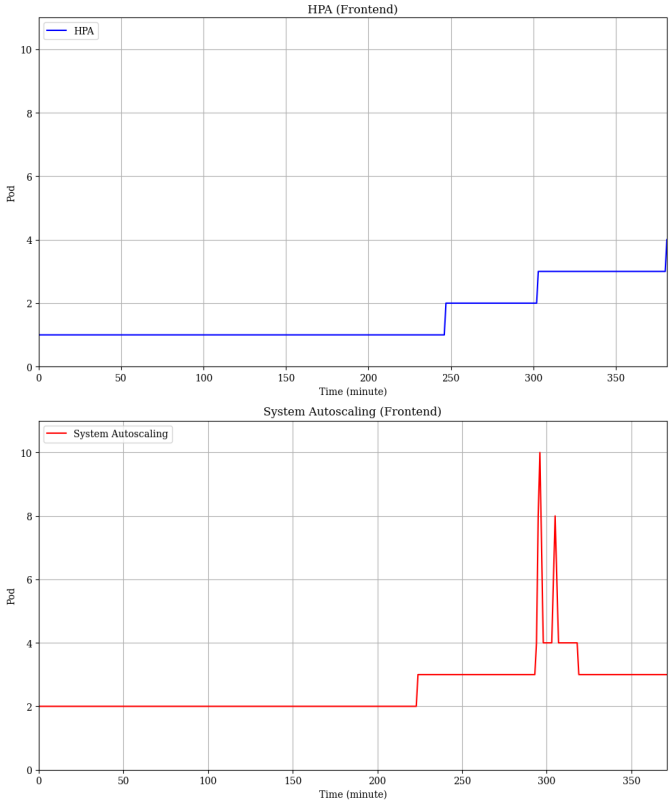

Fig. 7. HPA and System Autoscaling Replica (Frontend)

VI. Conclussion

This study developed an autoscaling system that leverages high-level information and metrics to optimize resource management in microservices. Autoscaling can effectively utilize application-specific information, such as traffic patterns, by employing time series models like PatchTST. These models can predict future traffic and guide scaling decisions in microservices. Additionally, high-level metrics, specifically Service Level Objectives (SLOs) related to tail latency, can be directly used for autoscaling. A neural network-based regression model can perform multi-output regression to predict the tail latency for each service within the microservices architecture.

The impact of using high-level information and metrics in resource control includes several benefits. The autoscaling strategy becomes proactive, addressing provisioning delays and improving application availability. Furthermore, the strategy is not limited to low-level metrics, such as CPU or memory, as it does not require mapping from high-level to low-level metrics, demonstrating behavior comparable to HPA-based autoscaling. However, it was found that the autoscaling strategy proposed by Bartelucci et al. [2] is not directly applicable for determining the number of replicas needed to meet tail latency SLOs, as tail latency does not behave proportionally like computing resources.

For future research and development, several suggestions arise. The decision-making process for scaling replicas could be enhanced using optimization methods that leverage the developed models, aiming to find the optimal number of replicas required. Further research is needed to explore the interdependencies of SLOs, particularly tail latency SLOs, to establish appropriate latency thresholds for services with interdependencies. Additionally, testing should be expanded to more complex microservices, such as those with more intricate inter-service dependencies.

References

[1] N.-M. Dang-Quang and M. Yoo, "Deep Learning-Based Autoscaling Using Bidirectional Long Short-Term Memory for Kubernetes," Applied Sciences, vol. 11, no. 9, p. 3835, Apr. 2021, doi: https://doi.org/10.3390/app11093835.

[2] Nicolo Bartelucci, Paolo Bellavista, T. Pusztai, A. Morichetta, and Schahram Dustdar, "High-Level Metrics for Service Level Objective-aware Autoscaling in Polaris: a Performance Evaluation," 2022 IEEE 6th International Conference on Fog and Edge Computing (ICFEC), May 2022, doi: https://doi.org/10.1109/icfec54809.2022.00017.

[3] J. Rahman and P. Lama, "Predicting the End-to-End Tail Latency of Containerized Microservices in the Cloud," Jun. 2019, doi: https://doi.org/10.1109/ic2e.2019.00034.

[4] Y. Nie, N. H. Nguyen, P. Sinthong, and J. Kalagnanam, "A Time Series is Worth 64 Words: Long-term Forecasting with Transformers," arXiv.org, Mar. 05, 2023. https://arxiv.org/abs/2211.14730

[5] A. Casolaro, V. Capone, G. Iannuzzo, and F. Camastra, "Deep Learning for Time Series Forecasting: Advances and Open Problems," Information, vol. 14, no. 11, p. 598, Nov. 2023, doi: https://doi.org/10.3390/info14110598.

[6] A. Detti, L. Funari, and L. Petrucci, "μBench: An Open-Source Factory of Benchmark Microservice Applications," IEEE transactions on parallel and distributed systems, vol. 34, no. 3, pp. 968–980, Mar. 2023, doi: https://doi.org/10.1109/tpds.2023.3236447.

[7] Weaveworks, "Sock Shop: A Microservices Demo Application," [Online]. Available: https://github.com/microservices-demo/microservices-demo. [Accessed: Aug. 17, 2024].

[8] Z [1]F. Zaker, "Online Shopping Store - Web Server Logs," Harvard Dataverse, Jan. 2019, doi: https://doi.org/10.7910/dvn/3qbyb5.

[9] "PatchTST," Huggingface.co, 2014. https://huggingface.co/docs/transformers/en/model_doc/patchtst (accessed Aug. 18, 2024).

[10] TensorFlow, "API Documentation | TensorFlow Core v2.4.1," TensorFlow. https://www.tensorflow.org/api_docs