Proactive Hybrid Autoscaling for Container-Based Edge Applications in Kubernetes

Kaile Zhu[1], Shihao Shen[1], Shizhan Lan[2], Xiaofei Wang[1](✉), Cheng Zhang[3], Chao Qiu[1], and Victor Leung[4,5]

[1] College of Intelligence and Computing, Tianjin University, Tianjin, China
xiaofeiwang@tju.edu.cn
[2] China Mobile Guangxi Institute Co. Ltd., Shanghai, China
[3] Institute of Technology, Tianjin University of Finance and Economics, Tianjin, China
[4] College of Computer Science and Software Engineering, Shenzhen University, Shenzhen, China
[5] Department of Electrical and Computer Engineering, The University of British Columbia, Vancouver, Canada

Abstract. As the rising of the Internet of Things (IoT), edge computing is widely adopted in numerous applications. However, current autoscaling tools are not designed for edge applications and can not utilize the heterogeneous resources of edge nodes efficiently. In this paper, we propose a proactive hybrid autoscaler specifically optimized for edge computing scenario. With the Bidirectional Long Short Term Memory (Bi-LSTM) based load prediction model, the proposed autoscaler is able to predict the future workload and perform scaling operation before it arrives. In addition, a overload compensation algorithm is implemented to mitigate the Quality of Service (QoS) decreasing due to under-prediction. Then, a hybrid scaling method is applied to simultaneously modify the number of pods and their resource quotas without restarting. Experimental results with a real-world workload dataset shows the proposed load prediction model has better accuracy compared with the Long Short Term Memory model and the state-of-the-art statistical analysis model, Autoregressive Integrated Moving Average (ARIMA), which is also more than 350 times slower than our model in prediction speed. Finally, evaluation in a real Kubernetes cluster shows that the proposed proactive hybrid autoscaler outperforms the default Horizontal Pod Autoscaler (HPA) of Kubernetes in terms of both QoS and resource utilization efficiency.

Keywords: Kubernetes · Edge Computing · Autoscaling

1 Introduction

Over the last few decades, cloud computing has gained significant attention from the industry. A countless number of applications we use everyday are powered

© ICST Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2024
Published by Springer Nature Switzerland AG 2024. All Rights Reserved
V. C. M. Leung et al. (Eds.): Qshine 2023, LNICST 574, pp. 330–345, 2024.
https://doi.org/10.1007/978-3-031-65123-6_24

by cloud service providers. However, cloud computing faces latency issue caused by geographic limitations [27], which is crucial for latency-sensitive applications (e.g. live streaming, IoT applications).

Edge computing provides a promising solution to this issue through exploiting computing resources located closer to users, due to which it is getting increasingly popular [13,23]. Not only is the latency issue solved but also the computing workloads are dispatched, further reduced the pressure on cloud services [18,24,25]. Featuring low start-up time, running overhead and image size, container-based virtualization is widely used in edge computing. One of the advantages of containerization is elasticity, which allows changing the resources allocated to applications. In real world scenarios, the workload of both cloud-based and edge applications is unstable, so autoscaling is essential for both cloud computing and edge computing. However, how to appropriately scale is a significant challenge. Over-provisioning, allocating excessive resources, leads to reduced resource utilization, while under-provisioning, allocating insufficient resources, results in rising latency and degraded Quality of Service (QoS).

In this paper, we propose a proactive hybrid autoscaler which combines both horizontal and vertical scaling. It makes scaling decisions based on the prediction of future workloads and performs simultaneous horizontal and vertical scaling based on the current resource. Moreover, when there is a significant deviation between predicted and actual workloads, the approach switches to threshold-based scaling as compensation. The main contributions of this paper include:

- Proposed a Bi-LSTM based load prediction model which outperforms traditional prediction methods.
- Proposed a overload compensation algorithm which scales up when system is overloading, preventing QoS from decreasing.
- Proposed a hybrid scaling method which scales both horizontally and vertically, refining the granularity of resource allocation.

The rest is organized as follows: In Sect. 2, previous studies of autoscaling approaches are reviewed. Section 3 presents some background of the research. Section 4 introduces the system design of the proactive hybrid autoscaler. The load prediction model, overload compensation algorithm and the hybrid scaling method are discussed in Sect. 5. In Sect. 6, the experiment conducted to evaluating the proposed autoscaler is presented. Section 7 concludes the paper.

2 Related Work

Currently, mainstream cloud service providers have all introduced their own container auto-scaling services. Amazon was the first to provide dynamic creation and termination of cloud server instances in its Amazon EC2 cloud computing service in 2006. In 2009, it introduced the Autoscaling service, allowing users to specify thresholds or schedule scaling at specific time points [3]. Microsoft Azure provided its threshold-based auto-scaling service for the first time in 2013 [4]. In 2015, Kubernetes released version 1.1, which included container horizontal scaling capabilities based on thresholds [16]. Google also offers similar features

in Google Kubernetes Engine [11]. Alibaba Cloud introduced elastic scaling in 2014, supporting both threshold-based and schedule-based auto-scaling [7], while Tencent Cloud provided a similar auto-scaling solution in 2016 [8]. These auto-scaling solutions are all based on setting thresholds or time points for horizontal scaling, resulting in mature implementations with low risk. In response to these auto-scaling solutions, some researchers have proposed methods to improve auto-scaling performance by determining appropriate scaling thresholds [2]. Others have suggested using multiple-level thresholds to enhance performance [12].

However, reactive autoscalers are unable to handle scenarios where application workloads experience rapid changes. This calls for a proactive autoscaler that can perform scaling actions before the workload arrives. As a result, some researchers have proposed using machine learning to predict future workloads or the number of container instances for proactive scaling. H. Zhao et al. used the double exponential smoothing prediction algorithm to forecast the required number of container instances in advance [28]. M. Chen et al. employed the weighted random forest algorithm to predict data center workload demands [6]. Meanwhile, D.-H. LUONG et al. used Linear Regression (LR) and Autoregressive Integrated Moving Average (ARIMA) models to predict future workloads and determine the scaling strategy [19]. With the advancement of Deep Learning (DL) technology, various deep learning-based prediction methods have also been proposed. F. Qiu et al. used Deep Belief Network (DBN) combined with a regression layer to predict CPU utilization [20]. MT. Imam et al. utilized Time Delay Neural Network (TDNN) for load prediction [14]. Although these proactive scaling approaches allow scaling actions to be taken before the workload arrives, the predicted values can never be perfectly accurate. When the predicted values are lower than the actual workload, the autoscaler may make wrong scaling decisions, leading to a significant reduction in Quality of Service (QoS).

In addition, horizontal scaling itself has some drawbacks. For instance, when the workload is low, the resources allocated to a container instance cannot be fully utilized, resulting in reduced resource utilization. Furthermore, horizontal scaling can only change the number of container instances, and the granularity of controlling resource quotas is relatively coarse. To address this issue, some researchers have proposed using Checkpoint/Restore In Userspace (CRIU) [9]for non-disruptive vertical scaling of containers [1,21]. However, the CRIU technology is not perfect and has several serious flaws. For example, after several rounds of generating and restoring snapshots of containers, an error may occur when generating a new snapshot [21].

3 Background

3.1 Container

Containers are an operating system-level virtualization technology used to run multiple isolated instances of applications on a single host. Each container contains the application and all its dependencies, such as libraries, runtime environments, and system tools. The container runtime is responsible for creating, running, and managing containers. The implementation of container runtime

leverages various resource isolation mechanisms in the Linux kernel, such as namespaces, control groups (cgroups), and Union File System (UnionFS). Based on the above, the container runtime isolates the application and its dependencies in a separate user space, providing an independent environment similar to a virtual machine. Compared to traditional virtual machines, containers do not need to fully simulate an entire operating system. Instead, multiple containers share the same operating system kernel, resulting in reduced resource consumption. Additionally, containers do not require the operating system to start during their launch, making them faster to start compared to virtual machines.

3.2 Kubernetes

Kubernetes is an open-source platform for automating container deployment, scaling, and management. It provides a powerful container orchestration and management system that helps developers efficiently manage large-scale containerized applications. Initially developed by Google and built on top of Google's internal Borg system, Kubernetes was open-sourced and released in 2014. The primary goal of Kubernetes is to simplify the deployment and management of containerized applications while providing a highly scalable platform. Today, Kubernetes has become the de-facto standard in the container orchestration field, gaining widespread support and adoption.

4 System Design

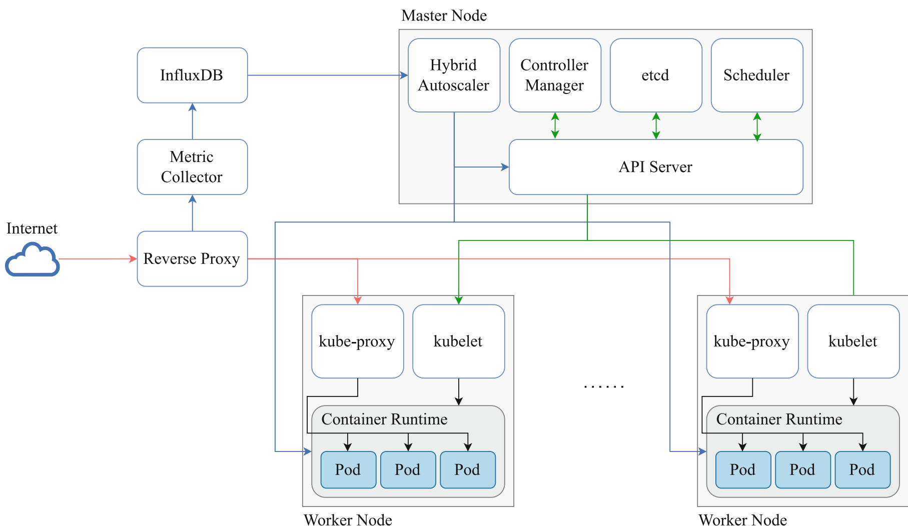

Fig. 1. System architecture

The system architecture is shown in Fig. 1. Requests coming from Internet are firstly received by the reverse proxy and then evenly dispatched to the corresponding pod in worker nodes. In this paper, we use Envoy [10], which is a high-performance reverse proxy developed in C++, providing functionalities such as data monitoring, traffic forwarding, and application discovery. Worker nodes host pods and manage them according to the command of the master node, which is responsible of organizing the cluster and keep it stable. There can be multiple master nodes in a cluster for high-availability in practice [17]. The proposed hybrid autoscaler dynamically modify the allocated resource of the application in response to the workload and the status of the cluster, which forms the monitor-analyze-plan-execute (MAPE) loop.

4.1 Monitor

The monitor system periodically collects various types of application metrics, such as the distribution of response latency, the queries per second (QPS) and the request error rate from the reverse proxy. Besides these, the resource usage is also collected from the kubernetes metrics server. Kubernetes metrics server is a component of Kubernetes that provides resource utilization data from nodes and pods in a cluster, from which our monitor system collects these metrics. All metric data is saved to a time-series database. In this paper, we use InfluxDB, which is an open-source time-series database designed for efficiently storing and analyzing high volumes of time-stamped data from various sources. It provides an intuitive and versatile data query API, which is used by the autoscaler.

4.2 Analysis

Based on the application workload collected before, the proposed autoscaler predicts the future workload using an Bi-LSTM based load prediction model, which is discussed in Sect. 6.1. In addition, the overload compensation algorithm runs periodically, checking the current status of the application using the collected metrics. Once it determines that the application is overloaded, the scaling decision will be made by an reactive scaling algorithm specifically designed for overload situation, until the application is not overloaded anymore. The overload compensation algorithm is discussed in Sect. 6.2.

4.3 Planning

The proposed hybrid autoscaler uses a hybrid scaling method which generate the target pod count and resource quota simultaneously from the target workload by modeling the application and thus the resource efficiency is increased. The hybrid scaling method is discussed in Sect. 6.3.

4.4 Execution

The scaling decisions are performed in this phase. The changes of pod count are carried out using the API provided by Kubernetes API server. The changes

of resource quota are carried out using the container runtime on worker nodes real-time, which doesn't require restarting pods.

5 Proposed Autoscaler

5.1 Load Prediction Model

Recurrent Neural Network(RNN) is widely used in time-series prediction tasks due to its recurrent architecture which enables it to remember dependencies in the input sequence. However, a simple RNN can only learn short-term dependencies because of the problems of gradient explosion and disappearance. To address the problem, Long Short Term Memory(LSTM), an variant of RNN, introduces gate functions, which enhances its ability to learn long-term dependencies.

Gate functions of the LSTM consist of the forget gate, the input gate and the output gate. The forget gate f_t controls how much information needs to be discarded from the previous state through a sigmoid function. The input gate i_t also includes a sigmoid function, determining how much information the current state should learn, and it uses a tanh function to generate the information to be learned. Then, the current status of cell C_t is updated by discarding old information and learning new information. Finally, the output gate o_t similarly contains a sigmoid function and a tanh function, producing the output h_t through the same mechanism as the input gate. Gate functions allow the model state C_t to be calculated through accumulation, thereby avoiding the problems of gradient explosion and disappearance. The equations of gate functions is listed below:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \tag{1}$$

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \tag{2}$$

$$\widetilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C) \tag{3}$$

$$C_t = f_t * C_{t-1} + i_t * \widetilde{C}_t \tag{4}$$

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \tag{5}$$

$$h_t = o_t * \tanh(C_t) \tag{6}$$

In the basic LSTM model, the model learns only in one direction of the input sequence, which may result in some deep-level dependencies not being extracted by the model. Consists of a forward LSTM and a backward LSTM, Bidirectional LSTM(Bi-LSTM) [22] traverses the input sequence from two directions simultaneously, and then combines the outputs of both models during the output phase, which allows the model to learn long-term dependencies from both directions of the input sequence, contributing to the improvement of the model's accuracy.

The proposed model is shown in Fig. 2, which has one Bi-LSTM layer which contains two opposite layers and one fully connected dense layer. The output tensor of Bi-LSTM is used as the input to the dense layer which then generates the prediction value.

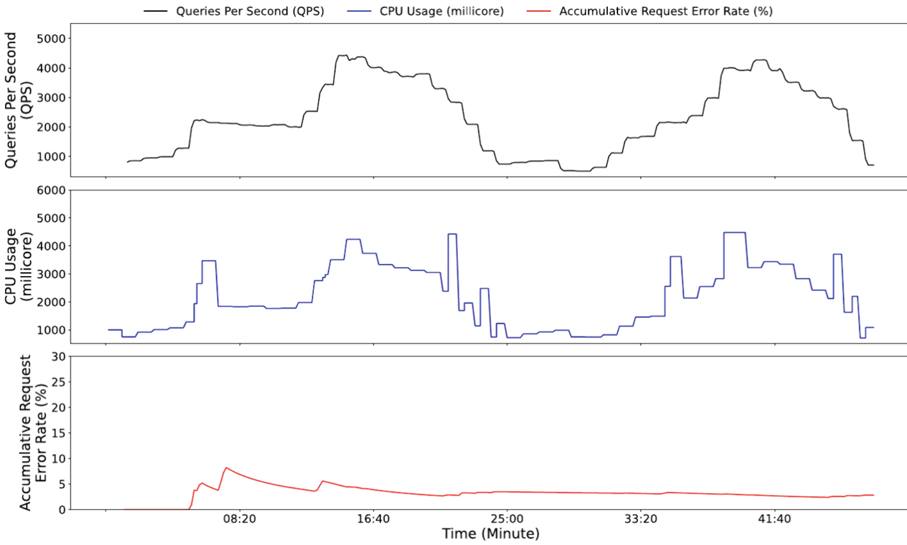

Fig. 2. Proposed Bi-LSTM model architecture

5.2 Overload Compensation Algorithm

It's impossible for any prediction methods to achieve 100% accuracy, which means that there will always be under-prediction, causing under-provisioning and Quality of Service (QoS) degrading. To alleviate this problem, we propose a overload compensation algorithm that automatically determines the status of application by analyzing its metrics and increases the resource quotas, thereby avoiding prolonged degradation of QoS.

The status of application is measured by the average request error rate e_{avg} in t seconds, which directly reflects the QoS status of application. A user-defined request error rate threshold x is introduced. When $e_{avg} > x$, it is determined that the application is overloaded, and the overload compensation algorithm starts resource provision. Likewise, when $e_{avg} < x$, the application is running normally, and the algorithm stops provisioning.

The resource provision is based on three metrics. First, traditional reactive scaling method is used to calculate R_{add} based on current CPU utilization. Second, $R_{history}$ is calculated by the latest queries per second (QPS) value collected in the time-series database. Third, the predicted workload generated by the load prediction model is used to calculate R_{pred}.

When the application is in a normal state, the proposed autoscaler uses R_{pred} as the scaling target R_{next}. When the application is in an overloaded state, the overload compensation algorithm takes the maximum value among R_{add}, $R_{history}$, and R_{pred} as R_{next}, as the Eq. 7. The algorithm is depicted in pseudo code in Algorithm 1.

$$R_{next} = \max(R_{add}, R_{history}, R_{pred}) \tag{7}$$

Algorithm 1. Overload Compensation Algorithm

Require: Request statistics in t seconds $Q[N]$
Ensure: The scaling target R_{next}

```
E[N] = errorRateCalc(Q[N])
isOverload = true
for i = 0 to N do
    if E[i] < x then
        isOverload = false
    end if
end for
if isOverload then
    R_now = getResource()
    R_add = addResourceCalc(R_now)
    R_history = historyResourceCalc(E[-1])
    R_pred = predictResourceCalc()
    R_next = max(R_now, R_history, R_pred)
else
    R_next = predictResourceCalc()
end if
return R_next
```

5.3 Hybrid Scaling Method

In previous research on container scaling methods, researchers typically optimized for either horizontal scaling or vertical scaling, without attempting to combine both approaches. In this paper, we establish a mathematical model of a test application based on the load test data, revealing the relationship of CPU usage, pod count and the queries per second (QPS). Hybrid scaling is enabled through this model, which greatly enhances the efficiency and flexibility of container scaling.

David et al. [5] has attempted to find the optimal upper limit of the CPU quota of each pod by plotting the graph of QPS and CPU usage. They discovered a linear relationship between these two factors. Building upon this finding, we conducted an experiment to investigate how the QPS relates to CPU usage and pod count.

Our test application is a simple HTTP service that extracts some string parameters from the request and concatenate them as return value. In terms of load simulating, we use k6 [15], an open-source load testing tool designed for developers to assess the performance and scalability of web applications by simulating virtual users and generating concurrent HTTP requests. The experimental results are shown in Fig. 3. It can be observed that the number of QPS and the CPU usage exhibit a linear relationship for each Pod count. As pod count increases, the number of QPS at the same CPU usage decreases. Based on the experimental data, the model is as follows:

$$qps = a * cpu * b^{pod_count} + c \tag{8}$$

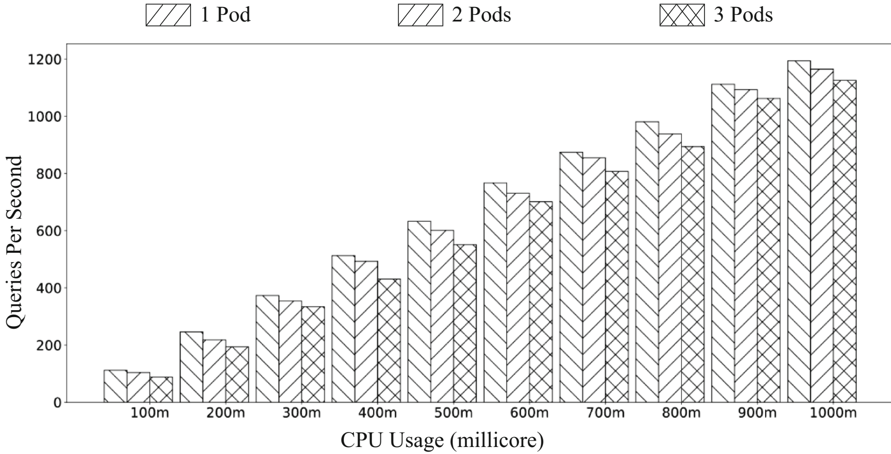

Fig. 3. The experiment result

In this model, a, b and c are model parameters obtained in experiment data. With the target QPS or total CPU quota known, the pod count and CPU quota of each pod can be calculated using this model to achieve hybrid scaling.

6 Experiment

In this section, we evaluate the proposed system design presented in Sect. 5. First, the proposed load prediction model is evaluated and its accuracy and prediction speed are compared with both the statistic-based ARIMA model and the original LSTM based model by using a real-world dataset. Subsequently, a complete proposed system in Sect. 5 is simulated and evaluated in a practical Kubernetes cluster to evaluate the performance in terms of provision accuracy and QoS improvement.

6.1 Dataset

The Wikipedia page view dataset [26] has been collected by Wikipedia since May 2015, containing hourly view counts per-article and per-project of all Wikimedia Foundation projects. The reason why we choose this dataset is that it's generated all by real web users. In the experiment, we used the view counts of the main page of Wikipedia in English and selected the view counts from September 8, 2022, 00:00:00 to September 30, 2022, 23:00:00 (23 days total) as our experiment dataset. Then we scaled the value to fit our experiment setup. The preprocessed dataset is shown in Fig. 4.

6.2 Experiment Setup

All parameters of the proposed Bi-LSTM based model are listed in Table 1. The parameter of compared LSTM model is the same as the proposed model except

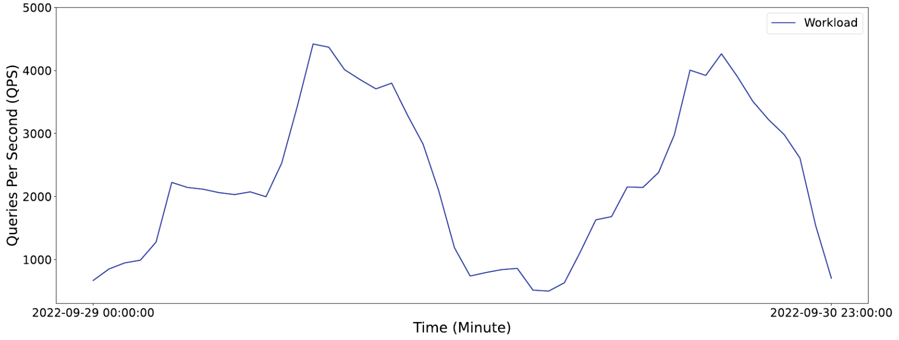

Fig. 4. The experiment dataset

Table 1. Parameters of the Proposed Model

Parameters	Value
Input size	16
Number of LSTM layers	2 (Forward and Backward)
Number of Dense layers	1
Units in hidden layers	128
Batch size	128
Epochs	640
Loss function	MSE
Optimizer	Adams
Activation function	tanh

the number of LSTM layers. We selected the request counts from September 8, 2022, 00:00:00 to September 28, 2022, 23:00:00 (3 weeks) as the training set, and the request counts from September 29, 2022, 00:00:00 to September 30, 2022, 23:00:00 (2 days) as the test set. Before training, the dataset is normalized by Z-score normalization to a range of $[-1, 1]$. The equation of Z-score normalization is:

$$z_t = \frac{x_t - \mu}{\sigma} \tag{9}$$

where x_t represents the request count sequence, μ represents the mean of x and σ represents the standard deviation of x. The paramaters p,d,q of ARIMA prediction model are chosen by grid searching.

In the second experiment, we have built the proposed hybrid autoscaler described in Sect. 5 on a realistic kubernetes cluster and compared it with the Kubernetes Horizontal Pod Autoscaler (HPA), the default autoscaler of Kubernetes. The target CPU Utilization of HPA is set at 80%. The cluster consists of one master node and two worker nodes, all of which shares the same hardware. The hardware configurations of each node in the cluster is listed in Table 3. The

proposed autoscaler runs on the master node. K6 is used for load simulation, similar to the tool discussed in Sect. 5.3. The reverse proxy continues receiving requests from k6 and dispatches them to all pods evenly. The test application in Sect. 5.3 runs in all pods. The monitor system collects hardware-level and application-level metrics and stores them in the time-series database InfluxDB. The hybrid autoscaler retrieves the data through InfluxDB's data query API, and then generates scaling decision with algorithms discussed in Sect. 6. Finally, the scaling operation is made through Kubernetes API server and the container runtime on worker nodes.

6.3 Evaluation Metrics

We use four metrics as Eqs. 10–13 respectively to evaluate the performance of the proposed model. Equation 10 computes the mean square error (MSE) by taking the average of the squared differences between predicted and observed values. The root mean square error (RMSE) in Eq. 11 represents the square root of twice the differences between the predicted and observed values. Equation 12 presents the mean absolute error (MAE), which is the average absolute difference between the predicted and actual observations, with equal weights for individual differences within the test sample. These metrics, MSE, RMSE, and MAE, serve as evaluation tools for model prediction errors, where lower values indicate higher prediction accuracy, and vice versa. Furthermore, we have the coefficient of determination, represented as R^2 in Eq. 13, which serves as a goodness-of-fit measure for linear regression models. R^2 is the square of the coefficient of multiple correlations between the predicted and observed values. A higher R^2 value indicates a better fit of the model to the data.

$$\mathrm{MSE}(y, \widehat{y}) = \frac{1}{n} \sum_{i=1}^{n} (y_i - \widehat{y}_i)^2 \tag{10}$$

$$\mathrm{RMSE}(y, \widehat{y}) = \sqrt{\frac{1}{n} \sum_{i=1}^{n} (y_i - \widehat{y}_i)^2} \tag{11}$$

$$\mathrm{MAE}(y, \widehat{y}) = \frac{1}{n} \sum_{i=1}^{n} |y_i - \widehat{y}_i| \tag{12}$$

$$R^2 = 1 - \frac{\sum_{i=1}^{n} (y_i - \widehat{y}_i)^2}{\sum_{i=1}^{n} (y_i - \overline{y}_i)^2} \tag{13}$$

In addition, to evaluate the performance of the proposed hybrid autoscaler, we use the accumulative request error rate e, the 95% tail latency $t_{95\%}$ and the total cpu usage r_{cpu}. Both the request error rate and 95% tail latency are Quality of Service (QoS) oriented metrics that are often used to measure the service quality of an application, and the total cpu usage means the resource utilization efficiency of an autoscaler (Table 2).

6.4 Experiment Results

Table 2. Model Performance Comparison

	MSE	RMSE	MAE	R^2	Prediction Time
ARIMA	0.074	0.27	0.22	0.91	498 ms
LSTM	0.065	0.26	0.20	0.92	1.35 ms
Proposed Model	0.042	0.20	0.14	0.95	1.39 ms

Load Prediction Model. Result of the first experiment is shown in Table 3. The proposed load prediction model achieves smaller MSE, RMSE and MAE values compared with the ARIMA model, which means the prediction accuracy of the proposed model is higher. Besides, the R2 value of the proposed model is also higher than the ARIMA model. This means the proposed model fits the data better than the ARIMA model. In addition, the proposed model has much faster prediction speed, which is more than 350 times faster than ARIMA. A comparison of the actual workload value and the predicted workload value of the ARIMA model, the original LSTM model and the proposed model is shown in Fig. 5, Fig. 6 and Fig. 7 respectively. As demonstrated, the ARIMA model can't obtain a stationary form of the historical workload, and thereby generates inaccurate prediction values. The LSTM based model can exploit temporal information to help prediction, resulting in reduced error. The proposed model utilizes Bi-LSTM architecture to extract long-term dependencies, which further enhances prediction accuracy.

Hybrid Autoscaler. The ARIMA model is not used in this experiment due to its slow prediction speed. The request loads are generated according to the test set mentioned above, with time interval scaled from one day to one minute, forming a 48-minute long experiment. The proposed hybrid autoscaler is compared with HPA in Table 3. It can be seen that our autoscaler achieves higher resource utilization efficiency while maintains QoS. Figure 8 and Fig. 9 shows the resource allocation and accumulative request error rate of our autoscaler and HPA respectively. By default, HPA only performs downscaling when the predefined metric values are below the threshold for at least 5 min, which is called cooldown delay [16]. It is shown in Fig. 9 that there's a lot of waste of resources due to belated downscaling operations.

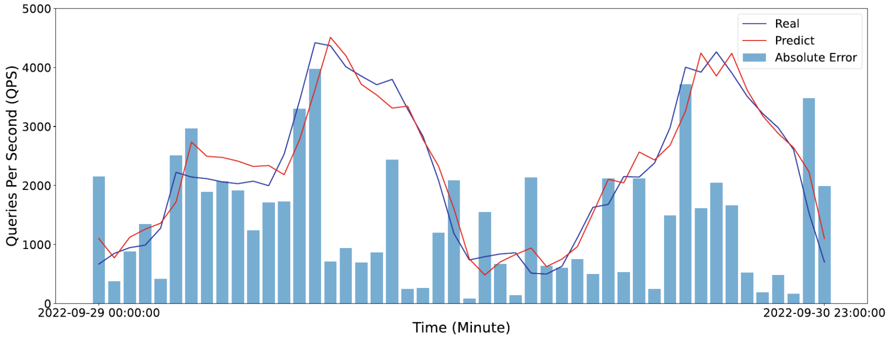

Fig. 5. Prediction result of the ARIMA model

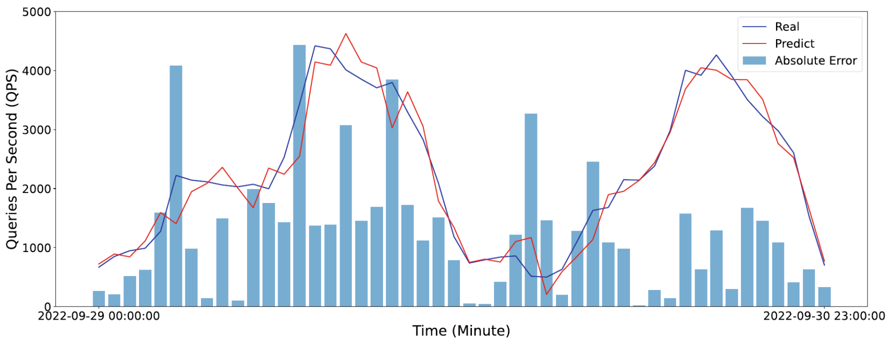

Fig. 6. Prediction result of the LSTM model

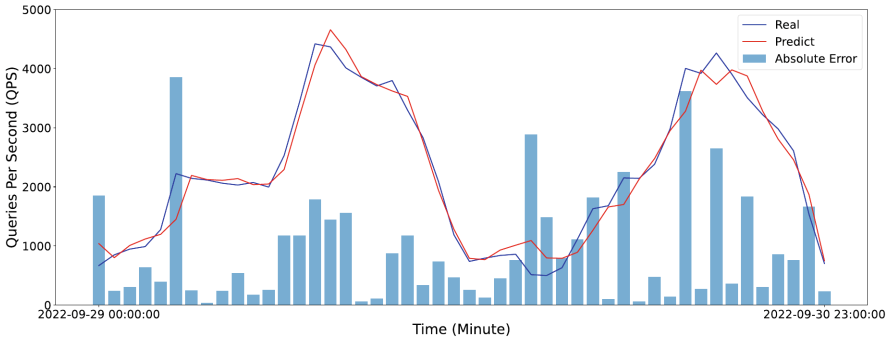

Fig. 7. Prediction result of the proposed model

Table 3. Autoscaler Performance Comparison

	e	$t_{95\%}$	r_{cpu}
Proposed Autoscaler	2.79%	45.08 ms	4701950 m
HPA	6.54%	99.62 ms	5253960 m

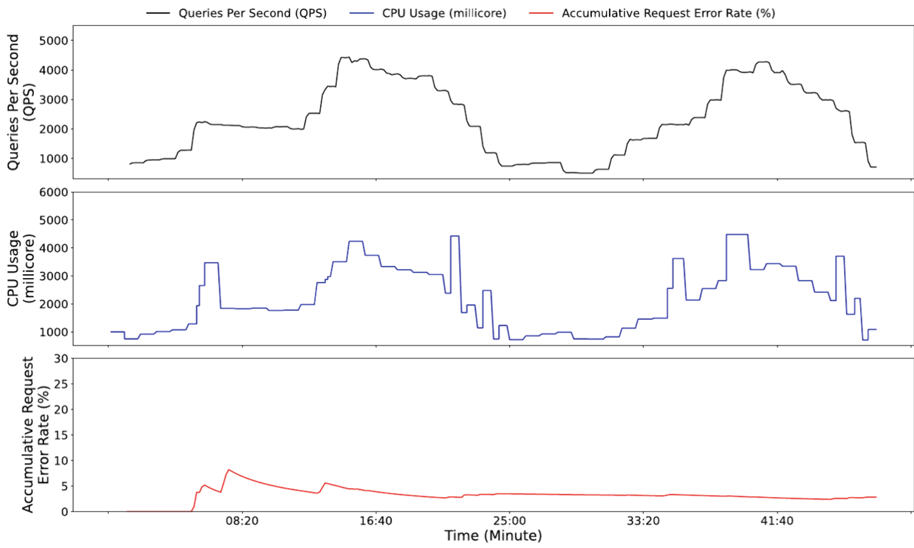

Fig. 8. The experiment result of the proposed autoscaler

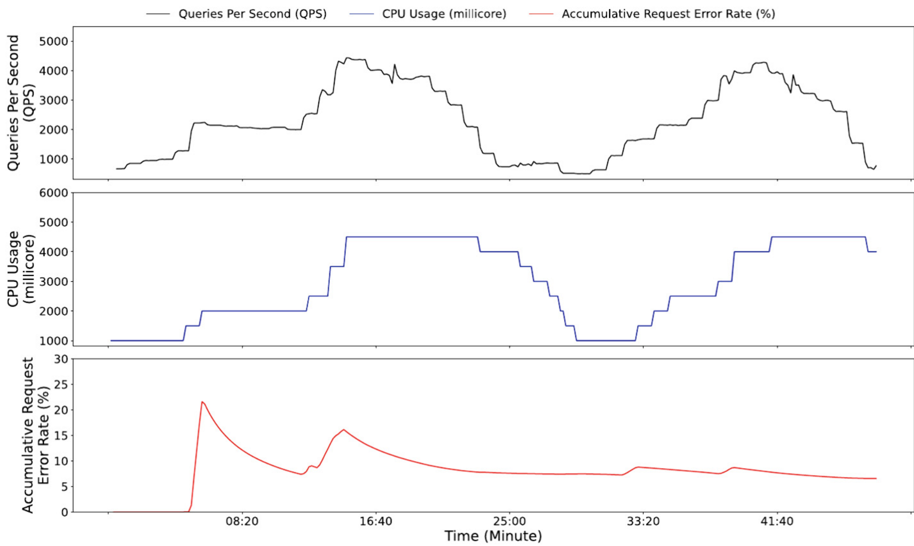

Fig. 9. The experiment result of HPA

7 Conclusion

Autoscaling in edge computing is as essential as it is in cloud computing. In this paper, we present a proactive hybrid autoscaler based on the Kubernetes container orchestration platform. A Bi-LSTM based load prediction model is used to forecast future workload and provide resources ahead of time. To alleviate overload caused by under-prediction, a overload compensation algorithm periodically check the status of application and increases resuorce quota when overloaded. Finally, a hybrid scaling method combines horizontal and vertical scaling, providing finer granularity of resource provision. The results of experiments evaluating the load prediction model and the autoscaler indicate that the proposed model is not only faster than the ARIMA model but also more accurate than both the LSTM based model and the ARIMA model. In addition, the proposed autoscaler achieves higher Quality of Service (QoS) while reduces resource cost than the default autoscaler, horizontal pod autoscaler (HPA) of Kubernetes.

Acknowledgments. This work was supported by the National Science Foundation of China (Grant 62072332), the China NSFC (Youth) (Grant 62002260), the China Postdoctoral Science Foundation (Grant 2020M670654), the Tianjin Xinchuang Haihe Lab (Grant 22HHXCJC00002) and the China Mobile Guangxi Development (Commissioning) Procurement Project (Grant CMGX-202201594).

References

1. Al-Dhuraibi, Y., Paraiso, F., Djarallah, N., Merle, P.: Autonomic vertical elasticity of docker containers with elasticdocker. In: 2017 IEEE 10th International Conference on Cloud Computing (CLOUD), pp. 472–479. IEEE (2017)
2. Al-Haidari, F., Sqalli, M., Salah, K.: Impact of CPU utilization thresholds and scaling size on autoscaling cloud resources. In: 2013 IEEE 5th International Conference on Cloud Computing Technology and Science, vol. 2, pp. 256–261. IEEE (2013)
3. Amazon: Auto scaling. https://aws.amazon.com/cn/autoscaling/. Accessed 25 July 2023
4. Autoscale, A.: Microsoft azure. https://azure.microsoft.com/en-us/products/virtual-machines/autoscale/. Accessed 25 July 2023
5. Balla, D., Simon, C., Maliosz, M.: Adaptive scaling of kubernetes pods. In: NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium, pp. 1–5. IEEE (2020)
6. Chen, M., Yuan, J., Liu, D., Li, T.: An adaption scheduling based on dynamic weighted random forests for load demand forecasting. J. Supercomput. **76**, 1735–1753 (2020)
7. Cloud, A.: Auto scaling: Automatically adjusts computing resources. https://www.alibabacloud.com/product/auto-scaling. Accessed 25 July 2023
8. Cloud, T.: Auto scaling. https://www.tencentcloud.com/products/as. Accessed 25 July 2023
9. CRIU: Criu. https://criu.org/. Accessed 25 July 2023
10. Envoy: Envoy proxy - home. https://www.envoyproxy.io/. Accessed 25 July 2023

11. Google: About gke scalability. https://cloud.google.com/kubernetes-engine/docs/best-practices/scalability. Accessed 25 July 2023
12. Hasan, M.Z., Magana, E., Clemm, A., Tucker, L., Gudreddi, S.L.D.: Integrated and autonomic cloud resource scaling. In: 2012 IEEE Network Operations and Management Symposium, pp. 1327–1334. IEEE (2012)
13. Huang, S., Wang, Z., Zhang, H., Wang, X., Zhang, C., Wang, W.: One for all: unified workload prediction for dynamic multi-tenant edge cloud platforms. In: Association for Computing Machinery's Special Interest Group on Knowledge Discovery and Data Mining (KDD). ACM (2023)
14. Imam, M.T., Miskhat, S.F., Rahman, R.M., Amin, M.A.: Neural network and regression based processor load prediction for efficient scaling of grid and cloud resources. In: 14th International Conference on Computer and Information Technology (ICCIT 2011), pp. 333–338. IEEE (2011)
15. k6, G.: Load testing for engineering teams — grafana k6. https://k6.io/. Accessed 25 July 2023
16. Kubernetes: Horizontal pod autoscaling. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/. Accessed 25 July 2023
17. Kubernetes: Kubernetes. https://kubernetes.io/. Accessed 25 July 2023
18. Liu, Z., Song, J., Qiu, C., Wang, X., Chen, X., He, Q., Sheng, H.: Hastening stream offloading of inference via multi-exit dnns in mobile edge computing. IEEE Trans. Mobile Comput. (2022)
19. Luong, D.H., Thieu, H.T., Outtagarts, A., Ghamri-Doudane, Y.: Predictive autoscaling orchestration for cloud-native telecom microservices. In: 2018 IEEE 5G World Forum (5GWF), pp. 153–158. IEEE (2018)
20. Qiu, F., Zhang, B., Guo, J.: A deep learning approach for vm workload prediction in the cloud. In: 2016 17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), pp. 319–324. IEEE (2016)
21. Rattihalli, G., Govindaraju, M., Lu, H., Tiwari, D.: Exploring potential for non-disruptive vertical auto scaling and resource estimation in kubernetes. In: 2019 IEEE 12th International Conference on Cloud Computing (CLOUD), pp. 33–40. IEEE (2019)
22. Schuster, M., Paliwal, K.K.: Bidirectional recurrent neural networks. IEEE Trans. Signal Process. **45**(11), 2673–2681 (1997)
23. Shen, S., et al.: A holistic qos view of crowdsourced edge cloud platform. In: International Symposium on Quality of Service (IWQoS). IEEE/ACM (2023)
24. Shen, S., Han, Y., Wang, X., Wang, S., Leung, V.C.: Collaborative learning-based scheduling for kubernetes-oriented edge-cloud network. IEEE/ACM Trans. Network. **31**, 2950–2964 (2023)
25. Shen, S., Ren, Y., Ju, Y., Wang, X., Wang, W., Leung, V.C.: Edgematrix: a resource-redefined scheduling framework for SLA-guaranteed multi-tier edge-cloud computing systems. IEEE J. Sel. Areas Commun. **41**(3), 820–834 (2022)
26. Wikipedia: Pageviews - wikitech. https://dumps.wikimedia.org/other/pageviews/readme.html. Accessed 25 July 2023
27. Zhang, H., et al.: How far have edge clouds gone? a spatial-temporal analysis of edge network latency in the wild. In: International Symposium on Quality of Service (IWQoS). IEEE/ACM (2023)
28. Zhao, H., Lim, H., Hanif, M., Lee, C.: Predictive container auto-scaling for cloud-native applications. In: 2019 International Conference on Information and Communication Technology Convergence (ICTC), pp. 1280–1282. IEEE (2019)