DScaler: A Horizontal Autoscaler of Microservice Based on Deep Reinforcement Learning

Zhijiao Xiao
College of Computer Science and Software Engineering
Shenzhen University
Shenzhen, China
Email: cindyxzj@szu.edu.cn

Song Hu
College of Computer Science and Software Engineering
Shenzhen University
Shenzhen, China
Email: 2070276119@email.szu.edu.cn

***Abstract*—With the development of container technology, microservice architecture has become a powerful paradigm for cloud computing with efficient infrastructure management and large-scale service capabilities. Cloud providers require flexible resource management to meet dynamic workloads, such as autoscaling and provisioning. As one of the most popular open-source container orchestration systems, Kubernetes provides a built-in mechanism, Horizontal Pod Autoscaler (HPA), for dynamic resource autoscaling. However, the static rules of HPA are not adaptable to highly dynamic workloads. In this paper, we propose a deep reinforcement learning-based horizontal autoscaler(DScaler) for autoscaling of microservices deployed in Kubernetes. Under two workloads with different characteristics, our experiments show that the proposed approach reduces resource consumption by 19.90% and 10.80% while reducing SLA violations by 8.56% and 12.75% compared with HPA, respectively. In addition, our approach can significantly reduce resource consumption by about 60% compared to the existing reinforcement learning strategy while maintaining SLA within an acceptable level.**

***Index Terms*—Cloud Computing, Microservice, Autoscaling, Kubernetes, Deep Reinforcement learning**

I. Introduction

Microservices architecture is widely popular in the cloud data center. Unlike the traditional monolithic application, microservices are autonomous services deployed independently with a single and defined purpose and provide reliability, scalability, and agility benefits [1]. Container has become a commonly used deployment technology for microservices due to their fast start-up and lightweight while gaining significant popularity in mainstream cluster management systems such as Docker Swarm and Kubernetes. Kubernetes [2] is an open-source container orchestration platform that enables high availability and scalability through three autoscaling mechanisms e.g., vertical, horizontal, and cluster autoscaling. Horizontal autoscaling refers to dynamically adjusting the number of instances of microservices, aiming to improve the Quality of Service(QoS) to cope with fluctuating workloads. The autoscaling mentioned below is horizontal unless otherwise noted. The most common horizontal scaling is the Horizontal Pod Autoscaler(HPA) of Kubernetes, which deploys a static rule [3].

To guarantee QoS and reduce cloud resource waste, many autoscaling strategies have been proposed, such as rule-based, queuing theory-based and reinforcement learning-based scaling strategies. Rule-based strategies are manually designed with static rules (e.g., threshold rules) based on relevant performance metrics, which are simple to implement but difficult to adapt to systems with different characteristics and dynamic workloads. In addition, some studies used queuing theory to improve scaling system performance by studying statistical patterns in metrics such as user request arrival and response times. Such approaches assume that the arrival time satisfies a probabilistic model. However, their performance might degrade when the actual arrival time of requests differs significantly from the assumed distribution.

Reinforcement learning(RL), one of the paradigms and methodologies of machine learning, uses intelligent agent to learn optimal policy by interacting with the environment and obtaining reward signal. Since tabular RL algorithms (e.g., Q-learning, SARSA) are failed to handle the problems with large or continuous state and action space, Deep Reinforcement Learning(DRL) using Deep Neural Network(DNN) to approximate Q function is proposed. Motivated by advantages of DRL algorithms, we propose a DRL-based algorithm to solve horizontal scaling problem of microservices in container cloud. Our contributions of this paper are:

- First, we build an extensible system for autoscaling of microservices by integrating a horizontal autoscaler named DScaler into Kubernetes cluster.
- Second, we propose a horizontal autoscaling algorithm based on Deep Q Network(DQN) to solve the autoscaling problem of microservices.
- Finally, we conduct comparative experiments under workloads with different characteristics demonstrating that DScaler can effectively improve resource utilization while maintaining the QoS within an acceptable level.

The rest of this paper is organized as follows. Section II depicts related work about autoscaling in cloud environment. In section III, we define the problem and describe the autoscaling system. In section IV, we formulate the horizontal scaling problem of microservices as Markov Decision Process(MDP) and describe the autoscaling algorithm. Section V presents the experiments and validation results. Finally, section VI concludes the paper with a summary and future works.

II. Related Work

Autoscaling of microservice has received extensive attention from the industry and academia. We summarize previous researches and divide them into three categories, as discuss blow.

A. Rule-based strategies

Rule-based strategies typically model the scaling problem and construct scaling rules through domain knowledge. Nguyen et al. [3] analyzed the impact of different metrics, monitors, and metric sampling periods on the scaling performance of Horizontal Pod Autoscaler(HPA) of Kubernetes. Casalicchio et al. [4] analyzed the correlation between absolute and relative resource usage metrics under different workloads and proposed an improved Kubernetes horizontal pod autoscaling algorithm KHPA-A to control response time. Zheng et al. [5] presented a SLA-aware and microservice-centric autoscaling framework which needs to set different thresholds and rules in accordance with different microservices. Zhu et al. [6] proposed a bi-metric autoscaling approach for pods that considers CPU and thread pool utilization to improve the performance of web applications on Kubernetes. Rule-based methods are efficient and easy to implement in simple environment. However, developing appropriate rules in complex environments requires expert domain knowledge and experience which can bring additional human costs.

B. Queuing theory-based strategies

Some queuing models describe autoscaling systems by assuming memoryless arrival processes of user requests. Gias et al. [7] proposed an autoscaling system named ATOM that models microservice applications using a Layered Queuing Network and optimizes resources usage with a genetic algorithm. Tong et al. [8] proposed a holistic autoscaling strategy based on a balanced Jackson Queuing Network that optimizes SLA violation rate and resource cost of multi-service application. Ding et al. [9] proposed an autoscaling method named COPA, which models the scaling problem of microservice as a M/M/c model of queuing network according to workload and response time. Queuing theory allows evaluating application performance under different conditions of workload and number of instances by assuming a Markov arrival process. However, the accuracy of the performance estimate will decrease when the arrival time or service time deviate significantly from the assumed distribution.

C. Machine learning-based strategies

There have been some researches that build autoscaling methods using machine learning techniques. Yu et al. [10] proposed an autoscaling system using Bayesian optimization to find the optimal scaling actions. Abdullah et al. [11] proposed a burst-aware autoscaling algorithm based on decision tree to identify bursty workloads. Reinforcement learning, one of machine learning paradigms, is also used to handle with the autoscaling problem in cloud environment. Horovitz et al. [12] proposed a method based on Q-learning to dynamic change threshold of horizontal scaling. Compared with traditional RL, they greatly reduce state space by exploiting the monotonicity of Q function. Rossi et al. [13] proposed a model-based RL approach for controlling elasticity of container-based application, and discussed difference caused by different action space. A method combining ARIMA model and SARSA was proposed in [14]. They used ARIMA model to predict incoming requests of users and get scaling action by SARSA algorithm. FScaler [15] based on SARSA algorithm was proposed to solve dynamic resources scaling problem in Kubernetes fog cluster. Unlike these studies using tabular RL algorithms, we use a deep neural network to approximate Q function.

III. Problem Definition And System Design

In this section, we defined the horizontal autoscaling problem of microservice and described our autoscaling system architecture, based on Kubernetes and Istio [16].

A. Problem Definition

For microservices deployed with containers, multiple instances should be deployed to handle incoming user requests in parallel. Since the application workload changes dynamically over time, the number of microservices instances should be scaled accordingly to the workload to provide stable service to users. Horizontal scaling is the behavior of adding or removing instances to meet an acceptable response time for users and to optimize the consumption of container resources.

Optimization objectives. Optimization objectives of the proposed horizontal autoscaler are minimizing the Service Level Agreement(SLA) violation rate and maximizing resource utilization. SLA defines the quality of service that cloud service providers guarantee to the subscribers. Here, SLA is defined as an upper bound RT_{sla} of the response time of an application. The violation rate is defined as the ratio of violation time to total time as described by (1). T_{vio} is the period in which the response time of the application exceeds RT_{sla}, and T is the total time application runs. It is also essential to improve resource utilization. We focus on CPU resources in this work and use the average CPU utilization to measure resource utilization, which is defined as (2). N is the number of instances. U_i denotes the CPU utilization of the i_{th} instance at time t.

$$SLA_{vio} = \frac{T_{vio}}{T} \tag{1}$$

$$Avg_{cpu} = \frac{1}{T}\frac{1}{N}\sum_{t=0}^{T}\sum_{i=1}^{N} U_{ti} \tag{2}$$

B. System Design

We build an autoscaling system based on Kubernetes and Istio, as shown in Fig. 1. The system can be decomposed into Monitor-Analyze-Plan-Execute(MAPE) loops, a management structure widely used in self-adaptive systems [17]. Monitor continuously retrieves system information and updates knowledge. A analyzer used the knowledge to decide whether an adaptation is necessary. Then planner determines an adaptation plan for the application based on the optimization objective and delivers it to executor for execution.

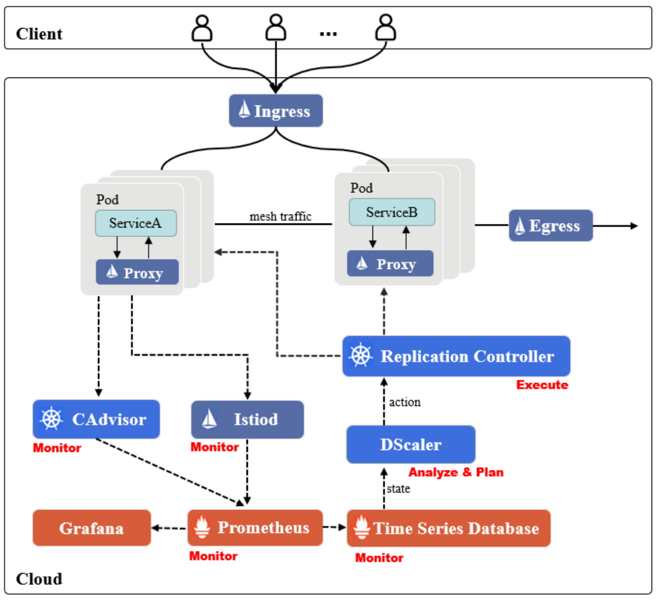


Fig. 1. System Architecture.

Monitor phase. The monitor needs to capture service-related metrics(e.g., response time, request rate) and container-related metrics(e.g., CPU utilization, number of instances) to get an accurate picture of application performance. By injecting a sidecar proxy container into each instance of service, the service-related metrics can be collected by Istiod, which is the main component of Istio's control plane. Cadvisor is in charge of collecting container-related metrics. In every monitor interval, Prometheus [18] grabs metrics data from CAdvisor and Istiod, then store them in the time series database, as shown in Fig. 1. DScaler can access metrics data from the database with PromQL, a query language for Prometheus. Grafana [19] is an open-source data visualization tool that exhibits real-time data for users on a website.

Analyze and plan phase. In every decision interval, the DScaler obtains current cluster metrics from the time series database and chooses the best action to scale in/out instances of application based on the optimization objective. Meantime, it also acquires the performance feedback of the previous scaling period from the database to adjust the model dynamically. The input of DScaler is the state of the system, including service-related metrics and container-related metrics, and output is the horizontal scaling action.

Execute phase. Replication Controller ensures a specified number of pod replicas are always available. The Replication Controller will terminate the redundant pods if there are too many pods. Through calling API provided by Kubernetes API Server, DScaler can deliver the scaling action to the Replication Controller for execution.

IV. Modeling DRL-based Horizontal Auto Scaling

A. Markov Decision Process Modeling

MDP is a mathematical model of sequential decision making which can be defined as a 5-tuple(S, A, P_a, R_a, γ) where S is a set of states, A is a set of actions, $P_a(s, s') = P(s_{t+1} = s'|s_t = s, a_t = a)$ represents the probability that action a in state s at time t will move to a new state s' at time $t+1$. $R_a(s, s')$ is the immediate reward received after s transforming to s', and $\gamma \in [0, 1]$ is a discount factor indicates agents' attention to future reward. We model the horizontal autoscaling problem as a MDP and the specific definitions are as follows.

State Space and Actions. The state space S contains all possible states, which reflects container-related and service-related metrics of the system. We define each state as a 3-tuple $s = (U, N, W)$, where U denotes current average CPU utilization of Pods, N is the number of currently available instances of service, and W is the user request rate. All three metrics were normalized to be used as input to the deep neural network. Given a state $s \in S$, the action space A includes all available actions which can be taken by the agent. Here each action at state s is defined as $a = (-2, -1, 0, +1, +2)$, where ± 1 and ± 2 indicate respectively scaling in/out one instance and scaling in/out two instances. To reduce the additional cost of frequent scaling actions, a also contains the action 0 meaning no scaling operation is performed in this decision cycle.

Reward Function. In order to ensure SLA and improve the resource utilization as much as possible, the reward function is defined as shown in (3), where $util$ is the average CPU utilization of Pods and $perf$ indicates performance on service quality.

$$Reward = util + perf \tag{3}$$

The application performance is measured by response time, as shown in (4), where RT_{sla} is the response time threshold defined in the SLA, measured in 95th percentile response time which is more stricter than the average response time. RT_{obs} represents the corresponding real response time. $penalty$ is a negative constant. When SLA violations occur, the agent is given a negative penalty signal for the inappropriate scaling action. On the contrary, the $perf$ is positive when the SLA requirement is satisfied.

$$perf = \begin{cases} \frac{1}{1+\frac{RT_{obs}}{RT_{sla}}} & \text{if } RT_{obs} < RT_{sla} \\ penalty & \text{if } RT_{obs} \geq RT_{sla} \end{cases} \tag{4}$$

B. Horizontal Auto Scaler based on DQN

For the above MDP model, we designed a horizontal autoscaling algorithm for DScaler based on DQN, as shown in Algorithm 1. The input of the algorithm is the state s_t of application at time t, and the output is the scaling action used to adjust the number of instances. The algorithm needs to initialize two structurally identical deep neural networks and an experience replay which can eliminate correlations of data and increase training efficiency. Considering the input dimension and the problem's scale, we used a full-connected neural network as an approximator of the Q function.

Algorithm 1 Horizontal Autoscaling Algorithm based on DQN

```
Input: state $s_t = (U_t, N_t, W_t)$ at time $t$
Output: scaling action $a_t$
 1: Initialize experience replay $D$
 2: Initialize Q-network $Q$ with random weights $\theta$
 3: Initialize target network $Q^-$ with random weights $\theta^-$
 4: while True do
 5:     Get state $s_t$ of service at time $t$
 6:     With probability $\epsilon$ select a random action $a_t$ otherwise
        select $a_t = argmax_a Q(s_t, a; \theta)$
 7:     Send action $a_t$ to API-Server and perform scaling action
 8:     Observe next state $s_{t+1}$ and obtain reward $r_t$
 9:     Store transition $(s_t, a_t, r_t, s_{t+1})$ in $D$
10:     Sample random minibatch of transitions from $D$
11:     Set $y_j = r_j + \gamma max_{a'} Q^-(s_{j+1}, a'; \theta^-)$
12:     Perform a gradient descent step on $(y_j - Q(s_j, a_j; \theta))^2$
        with respect to network parameters $\theta$
13:     Every $C$ steps reset $\theta^- = \theta$
14: end while
```

At each scaling period, the following operations are repeated. Firstly, the agent observes current state s_t, feeds it into the Q-network, and chooses a random action or the action with maximum value according to $\epsilon - greedy$ policy. In our algorithm, the ϵ will decrease over time, which means the agent will gradually reduce exploration behaviors. Secondly, action a_t will be sent to API-Server for scaling by the Replication Controller. The environment will move to the next state s_{t+1} and the reward will be calculated according to (3) and (4). Then the transition data (s_t, a_t, r_t, s_{t+1}) will be stored in experience replay. Finally agent randomly samples data from the experience replay and performs gradient descent based on the mean square error. To increase training stability, the parameters θ^- of the target network are frozen and periodically copied from θ.

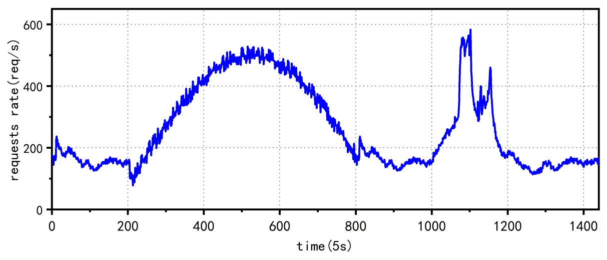
V. Experimental Results

In this section, we first introduce the experimental setup, then implement three algorithms(DScaler, A-SARSA [14], and HPA) for comparison experiments. Finally, we analyze experimental results from the perspectives of resource consumption and SLA violation rate.

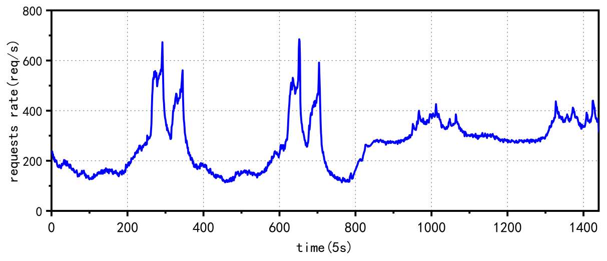
A. Experimental Setup

The experiment is set up on a physical machine with Intel Core i7-9700 (8 cores at 3.60 GHz) and 32GB of RAM. The Kubernetes cluster has one master node and two worker nodes, and each node is a 4-core and 8GB memory virtual machine with CentOS7. The resource allocation of every application instance is set to 0.5 core CPU and 512MB memory.

We deployed Bookinfo [20] application on the Kubernetes cluster. Bookinfo is the official sample application of Istio and emulates an online bookstore, and users can access the application to view information and ratings of books. We extract two workloads with different characteristics(slowly rise/down, rapidly rise/down, and smoothly fluctuate) from FIFA World Cup website access dataset [21], as shown in Fig. 2.



(a) workload 1



(b) workload 2

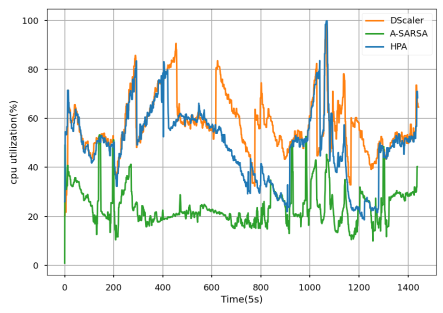
Fig. 2. Application workloads

B. Evaluation Metrics

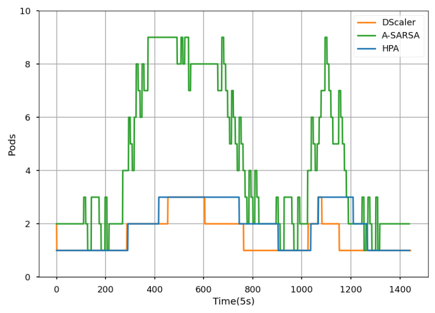
Evaluation metrics for algorithms include average CPU utilization, average resource consumption, and SLA violation rate. The average resource consumption is defined as the average number of instances allocated to the application during experiment time. The average CPU utilization is defined as (2). SLA violation rate is defined as (1), and the response time threshold in the SLA is set to 250ms. We are only concerned with whether the 95 percentile response time exceeds the threshold, not how much it exceeds in this work.

C. Experimental Result and Comparisons

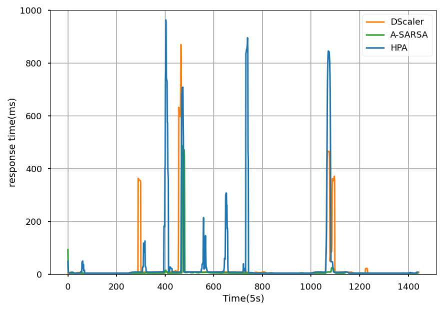
To evaluate the proposed DScaler more comprehensively, we deployed A-SARSA and Kubernetes HPA for comparative experiments. The Q-network and target network of DScaler contain one input layer, two hidden layers, and one output layer. Considering the scale of the horizontal scaling problem, the number of hidden layers and neurons should not be set too high. In our experiment, two hidden layers have 32 and 16 neurons, respectively. The parameters of the DScaler are as follows. The discount factor γ is set to 0.9, and the size of experience replay is set to 160, the update frequency of the target network is set to 100 steps, the penalty in the reward function is set to -2. For Kubernetes HPA, the target CPU utilization is set to 70%. The comparison experimental results of three algorithms are shown in Fig. 3, Fig. 4 and Table. I.



(a) CPU utilization under workload 1

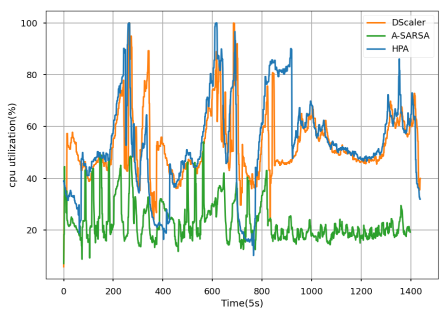


(b) Number of Pods under workload 1

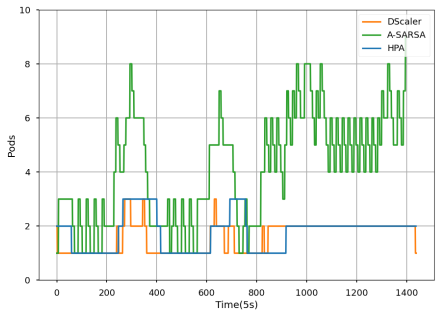


(c) 95th percentile response time under workload 1

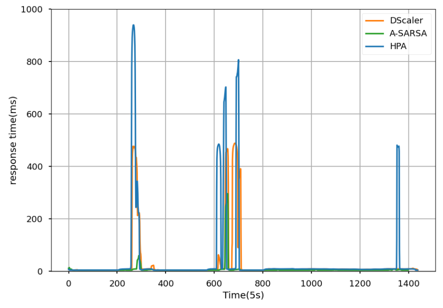
Fig. 3. Comparison of the DScaler, A-SARSA and HPA algorithms under workload 1.



(a) CPU utilizaion under workload 2



(b) Number of Pods under workload 2



(c) 95th percentile response time under workload 2

Fig. 4. Comparison of the DScaler, A-SARSA and HPA algorithms under workload 2.

TABLE I
COMPARISON OF PERFORMANCE METRICS.

Algorithms	Average CPU utilization	Average Resource consumption	SLA violation rate
DScaler (workload 1)	57.13%	1.53	3.74%
A-SARSA (workload 1)	23.84%	4.34	0.69%
HPA (workload 1)	49.27%	1.91	4.09%
DScaler (workload 2)	52.57%	1.57	4.79%
A-SARSA (workload 2)	22.80%	4.11	0.64%
HPA (workload 2)	53.34%	1.76	5.49%

DScaler vs A-SARSA. The average CPU utilization of DScaler under workload 1 is 57.13% which is significantly higher than A-SARSA, whose average CPU utilization is 23.84%, as shown in Table. I. The results under workload 2 are similar to workload 1. The average CPU utilization of DScaler is 52.5% which is much higher than A-SARSA, whose average CPU utilization is 22.80%. DScaler can handle both slowly rising and rapidly rising workloads with minimal resources to guarantee quality of service. Specifically, the resource consumption of DScaler is 64.75% less than A-SARSA under workload 1 and 61.80% less than A-SARSA under workload 2. It can be seen that A-SARSA usually uses more Pods to cope with the rising workload and does more scaling operations leading to jitter in system performance, as shown in shown in Fig. 3 and 4. This is because the Q table is not fine-grained enough to describe the system state when the state space is discretized. Overall, DScaler can balance resource consumption and quality of service for effectively reducing resource consumption while maintaining the SLA violation rate below 5% under both workloads.

DScaler vs HPA. Under workload 1, the average CPU utilization of DScaler improved by 7.86% over HPA. Also, the average resource consumption of DScaler is 19.90% less than HPA, which means DScaler is capable of responding to variations in workloads using fewer resources. Under workload 2, the resource consumption of DScaler is 10.80% less than HPA. In addition, DScaler also has a better performance in SLA violation compared to HPA. Under workload 1, DScaler reduced the violation rate by 8.56% compared to HPA. Under workload 2, DScaler also reduced the SLA violation rate by 12.75%. Due to the anti-jitter mechanism of HPA, the cluster will not scale down for a period after expansion, resulting in HPA being less timely than DScaler in scaling operations. In summary, HPA uses a static rule-based policy, which is simple to implement and inflexible, resulting in untimely and ineffective scaling operations. On the contrary, DScaler can accurately and efficiently scale horizontally by monitoring multiple metrics such as CPU utilization, request rate, and currently assigned Pods.

VI. Conclusion and Future Work

For highly dynamic workloads in cloud environments, this study proposed a horizontal scaling mechanism based on deep reinforcement learning, which can dynamically increase or decrease the number of application instances to accommodate different workloads. Comparative experiments were conducted under two different workloads, and our proposed algorithm maintained a balance between resource consumption and SLA violation rate, reducing consumption while maintaining the quality of service within an acceptable range.

The autoscaling strategy for this work is reactive, and we will consider workload prediction for proactive scaling using time series prediction methods in our future work. We will also explore the autoscaling strategies based on reinforcement learning for more complex scenarios such as multi-service applications.

Acknowledgment

This work is supported by the National Natural Science Foundation of China (NSFC) under Grant 62002230, Guangdong Natural Science Foundation under Grant 2019A1515012053, Shenzhen Science and Technology Foundation under Grant JCYJ20210324093212034.

References

[1] F. Auer, V. Lenarduzzi, M. Felderer, and D. Taibi, "From monolithic systems to Microservices: An assessment framework," *Information and Software Technology*, vol. 137, p. 106600, 2021.

[2] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes," *Communications of the Acm*, vol. 59, no. 5, pp. 50–57, 2016.

[3] T.-T. Nguyen, Y.-J. Yeom, T. Kim, D.-H. Park, and S. Kim, "Horizontal Pod Autoscaling in Kubernetes for Elastic Container Orchestration," *Sensors (Basel, Switzerland)*, vol. 20, no. 16, p. E4621, 2020.

[4] E. Casalicchio, "A study on performance measures for auto-scaling CPU-intensive containerized applications," *Cluster Computing-the Journal of Networks Software Tools and Applications*, vol. 22, no. 3, pp. 995–1006, 2019.

[5] T. Zheng, X. Zheng, Y. Zhang, Y. Deng, E. Dong, R. Zhang, and X. Liu, "SmartVM: a SLA-aware microservice deployment framework," *World Wide Web*, vol. 22, no. 1, pp. 275–293, 2019.

[6] C. Zhu, B. Han, and Y. Zhao, "A bi-metric autoscaling approach for n-tier web applications on kubernetes," *Frontiers of Computer Science*, vol. 16, no. 3, p. 163101, 2022.

[7] A. U. Gias, G. Casale, and M. Woodside, "ATOM: Model-Driven Autoscaling for Microservices," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pp. 1994–2004, 2019.

[8] J. Tong, M. Wei, M. Pan, and Y. Yu, "A Holistic Auto-Scaling Algorithm for Multi-Service Applications Based on Balanced Queuing Network," in *2021 IEEE International Conference on Web Services (ICWS)*, pp. 531–540, 2021.

[9] Z. Ding and Q. Huang, "COPA: A Combined Autoscaling Method for Kubernetes," in *2021 Ieee International Conference on Web Services, Icws 2021*, pp. 416–425, IEEE, 2021.

[10] G. Yu, P. Chen, and Z. Zheng, "Microscaler: Cost-Effective Scaling for Microservice Applications in the Cloud With an Online Learning Approach," *Ieee Transactions on Cloud Computing*, vol. 10, no. 2, pp. 1100–1116, 2022.

[11] M. Abdullah, W. Iqbal, J. L. Berral, J. Polo, and D. Carrera, "Burst-Aware Predictive Autoscaling for Containerized Microservices," *IEEE Transactions on Services Computing*, vol. 15, no. 3, pp. 1448–1460, 2022.

[12] S. Horovitz and Y. Arian, "Efficient Cloud Auto-Scaling with SLA Objective Using Q-Learning," in *2018 IEEE 6th International Conference on Future Internet of Things and Cloud (FiCloud)*, pp. 85–92, 2018.

[13] F. Rossi, M. Nardelli, and V. Cardellini, "Horizontal and Vertical Scaling of Container-Based Applications Using Reinforcement Learning," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pp. 329–338, 2019.

[14] S. Zhang, T. Wu, M. Pan, C. Zhang, and Y. Yu, "A-SARSA: A Predictive Container Auto-Scaling Algorithm Based on Reinforcement Learning," in *2020 IEEE International Conference on Web Services (ICWS)*, pp. 489–497, IEEE, 2020.

[15] H. Sami, A. Mourad, H. Otrok, and J. Bentahar, "FScaler: Automatic Resource Scaling of Containers in Fog Clusters Using Reinforcement Learning," in *2020 International Wireless Communications and Mobile Computing (IWCMC)*, pp. 1824–1829, 2020.

[16] Istio. [Online]. Available: https://istio.io/.

[17] O. Gheibi, D. Weyns, and F. Quin, "Applying Machine Learning in Self-adaptive Systems: A Systematic Literature Review," *Acm Transactions on Autonomous and Adaptive Systems*, vol. 15, no. 3, pp. 1–37, 2021.

[18] Prometheus. [Online]. Available: https://prometheus.io/.

[19] Grafana. [Online]. Available: https://grafana.com/.

[20] Bookinfo. [Online]. Available: https://istio.io/latest/zh/docs/examples/bookinfo/.

[21] M. Arlitt and T. Jin, "A workload characterization study of the 1998 World Cup Web site," *IEEE Network*, vol. 14, no. 3, pp. 30–37, 2000.