Hindawi
Security and Communication Networks
Volume 2023, Article ID 7400235, 10 pages
https://doi.org/10.1155/2023/7400235

Research Article

Container Scaling Strategy Based on Reinforcement Learning

Huaijun Wang,[1,2] **Chenfei Zhang**,[1,2] **Junhuai Li**,[1,2] **Dan Bao**,[1,2] **and Jiang Xu**[3]

[1]*Collaborative Innovation Center of Modern Equipment Green Manufacturing in Shaanxi Province, Xi'an 710048, China*
[2]*Xi'an University of Technology, Xi'an 710048, China*
[3]*China National Heavy Machinery Research Institute Co., Ltd., Xi'an 710032, China*

Correspondence should be addressed to Junhuai Li; lijunhuai@xaut.edu.cn

Received 3 September 2022; Revised 8 October 2022; Accepted 13 October 2022; Published 26 May 2023

Academic Editor: Jianbo Du

Elasticity capability is one of the most important capabilities of cloud computing, which combines large-scale resource allocation capability to quickly achieve minute-level resource demand provisioning to meet the elasticity requirements of different scale scenarios. The elasticity capability is mainly determined by the container start-up speed and container scaling strategy together, where the container scaling strategy contains both vertical container scaling strategy and horizontal container scaling strategy. In order to make the container scaling policy more effective and improve the application service quality and resource utilization, we briefly introduce Kubernetes' horizontal pod autoscaling (HPA) strategy, analyze the existing problem of HPA, and develop a container scaling strategy based on reinforcement learning. First, we analyze the problems of Kubernetes' existing HPA container autoscaling strategy in the scale-up and scale-down phases, respectively. Second, the Markov decision model is used to model the container scaling problem. Then, we propose a model-based reinforcement learning algorithm to solve the container scaling problem. Finally, we compare the experimental results of the HPA scaling strategy and the model-based reinforcement learning strategy with the results from the resource utilization of the application, the change of the number of pods, and the application response time; through the experimental analysis, we verify that the reinforcement learning-based container scaling strategy can guarantee the application service quality and improve the utilization of the application resources more effectively than the HPA strategy.

1. Introduction

The vigorous development of IoT cannot be achieved without the support of edge computing technology, which is needed for data fusion, data analysis, network security, and data security at the edge of the network [1]. Meanwhile, in the face of the growing data transmission and computation demands of IoT [2], edge computing can effectively cope with the lack of computing power of IoT devices themselves and fully alleviate problems such as network congestion [3]. For example, in a real industrial application scenario, real-time control of the production floor site is achieved by placing edge gateways inside the workshop. It is also possible to build an industrial control platform through the edge cloud and use the data analysis and decision-making capabilities provided by the platform to provide a basis for real-time control decisions [4]. In many fields such as industrial control, most of the services are deployed to the platform as containers. Considering the problems of scattered IoT devices, network heterogeneity, and limited computing node resources [5], it is important to design a reasonable and effective container resource scheduling mechanism in order to provide guaranteed application services to IoT devices.

Kubernetes, as the industry's leading container orchestration management system [6], is widely used in edge computing scenarios [7, 8]; however, the native container scaling policy and container scheduling policy of Kubernetes are too simple to meet the fine-grained resource scheduling requirements in edge computing scenarios. The HPA container scaling service implements container scaling mainly by measuring CPU and memory utilization, which has problems such as response delay and poor scaling timeliness and cannot meet the business requirements in edge computing scenarios. The resource scheduling policy only collects the remaining CPU and memory metrics of nodes and calculates the priority of nodes with uniform weight values to complete container scheduling. The scheduling mechanism considers only a single indicator, and the weighting method is too simple, and it suffers from low utilization of cluster resources and resource scarcity, which cannot meet the individual resource requirements of applications in edge computing scenarios.

The operation of application container instances in a cluster consumes certain costs, including those caused by application response times exceeding the maximum threshold value, in addition to the cost of memory and CPU resources that the containers themselves need to consume. Therefore, a reasonable and effective container scaling operation affects the application service quality as well as resource utilization.

In recent years, with the rapid development of Internet technology, the number of network edge devices has been growing exponentially. Traditional centralized cloud computing exhibits various problems such as high latency, low bandwidth, and high energy consumption when dealing with these massive edge data, for which edge computing is proposed. Before the birth of edge computing, to make up for the shortcomings of traditional centralized cloud computing, the industry proposed various computing models, such as cloudlet [9], fog computing [10], and mobile edge computing [11], which have different architectures but all aim to make up for the shortcomings of cloud computing, meet the growing demand for data processing with the development of the Internet, and guarantee the quality of service for users [12].

In terms of energy consumption reduction and cost considerations, Wang and Wang [13] proposed a novel cluster-level control architecture to achieve coordination of energy consumption and performance of virtualized server clusters, and experimentally demonstrated that the approach can provide effective control of both application-level performance and underlying energy consumption. Chen et al. [14] proposed a premigration strategy based on three load dimensions, combining a hybrid genetic algorithm and knapsack problem to achieve multiple adaptations, and experimentally demonstrated that the algorithm can effectively improve resource utilization and reduce energy consumption. Jeong et al. [15] proposed an energy-efficient service scheduling algorithm for federated edge clouds, which aims to minimize service migration overhead and energy consumption, and experimentally demonstrated that the algorithm improves energy efficiency by 21% and reduces service violation rate by 80%. Feng et al. [16] studied the application of heterogeneous computing (HC) and wireless power transfer (WPT) to federated learning to address the performing efficient learning tasks on the devices and achieving longer battery life. Feng et al. [17] explored a min-max cost-optimal problem to guarantee the convergence rate of federated learning in terms of cost in wireless edge networks. The literature [18] proposes a task offloading scheme by exploiting multihop vehicle computation resources in VEC based on mobility analysis of vehicles. This offloading scheme can achieve significant improvement in terms of response delay by at least 34% compared with the other algorithms (e.g., local processing and random offloading). In mobile edge computing, Mao et al. [19] proposed a reconfigurable intelligent surface (RIS)-assisted secure MEC network framework to enhance the task offloading security. Wei et al. [20] designed a preprocessing approach to convert raw traffic data into available datasets for deep learning-based traffic classifiers, which tailors raw traffic data as training datasets by resolving its structure and content and pruning redundant information. The literature [21] model the resource allocation in vehicular cloud computing (VCC) as a multiobjective optimization with constraints that aims to maximize the acceptance rate and minimize the provider's cloud cost.

In terms of energy reduction and cost considerations, Wang and Wang [13] proposed a Co-Con cluster control architecture based on the feedback theory control to minimize cluster power consumption while ensuring the quality of service of applications in the cluster. Chen et al. [14] present a multiadaptive genetic algorithm-based resource policy for premigration scheduling of virtual machines, which effectively reduce the energy consumption of the cluster. Wu et al. [22] proposed Green Scheduling algorithm based on DVFS dynamic voltage frequency scaling technique and job priority, which can prevent overuse of resources and effectively reduce power consumption while meeting the minimum resource requirements of applications. Rossi et al. [23] proposed an interesting approach to solve the container scaling problem by constructing the state space and action space of the container scaling problem and used reinforcement learning to merge horizontal and vertical scaling. However, they did not consider the characteristics of the Kubernetes' environment.

The distinctive features of this work are as follows:

(1) We model the container scaling problem by the Markov decision process (MDP), which includes the design of state space, action space, and cost function.

(2) Based on model-basedreinforcement learning algorithm to realize the iteration of the cost value and container scaling strategy in the process of container scaling, we evaluate the best container scaling strategy by solving the optimal Q-value function in the iterative process.

(3) Simulations are executed with different system parameters to show the effectiveness of the proposed algorithm. Simulation results reveal that the reinforcement learning-based container scaling strategy can effectively guarantee the application service quality and improve the application resource utilization compared with the HPA strategy.

The remainder of this paper is organized as follows. In Section 2, we analyze the shortages of Kubernetes' own container scaling strategy in the changeable edge computing environment. In Section 3, we leverage the Markov decision model to describe the problem of container scaling policy mode. In Section 4, we propose the construction of the

Markov decision model. In Section 5, we present how to solve the optimal Q-value function in the iterative process of reinforcement learning, based on which the optimal container scaling policy is evaluated to achieve the optimal scaling of containers. In Section 6, we test and discuss the effect of two container scaling strategies in the same experimental environment. We conclude this paper in Section 7.

2. Kubernetes Autoscaling Strategy

The edge computing environment is characterized by low latency and large connections. The existing Kubernetes autoscaling policy cannot adjust to dynamic application load changes in a timely manner in an edge computing environment, and the application itself does not have high resource utilization.

In this section, we focus on the scaling strategy that comes with Kubernetes. First, we introduce the Kubernetes autoscaling service. Then, we study the scaling strategy and analyze the problems that exist in the expansion and scaling phases.

2.1. Kubernetes Autoscaling Service. The flow of a pod in Kubernetes from resource allocation to service access is as follows: the application uses a user-defined YAML file to fetch the corresponding image from the image repository; allocates the CPU, memory, and other resources required for its operation; and deploys the application to Kubernetes in the form of a pod. Based on different application scenarios, users can adjust the policy manually by adjusting the requests' and limits' parameters. The CPU is allocated in a lean configuration by default. When CPU resources are oversubscribed, if there are not enough resources on the host node, there will be a preemption phenomenon. Getting the entry address of an application requires creating a Service API object, where Ingress is an API object in the Kubernetes cluster that plays the role of a router, through which we can customize routing rules for forwarding, managing, and exposing services. The user distributes the route to the pod represented by the Service object through Ingress to access the application, and the specific request flow diagram is shown in Figure 1:

The Kubernetes autoscaling service, designed as a control loop, can be controlled periodically by the controller (Kube-controller-manager), and the period time can be modified with the horizontal-pod-autoscaler-sync-period flag (default is 30 seconds). During each cycle, Kubernetes' Control Manager queries the cluster for current resource utilization based on each specific metric defined by HPA. The control manager obtains the basic metrics from the resource metrics API or from user-defined APIs (including other metrics). For each pod resource metric value (e.g., CPU), the controller obtains data from the HPA policy for each pod targeted by the resource metric API. If target utilization is given, the controller will calculate the current utilization as a percentage of the resource requests for the containers in each pod. If the target raw value is defined, the raw metric is used directly. Then, the controller obtains the utilization on all target pods or the average of raw values (depending on the specified target type) and generates the ratio used to scale the number of copies required.

2.2. Kubernetes Autoscaling Strategy. Kubernetes comes with a container scaling policy, HPA, which requires the definition of an HPA object when creating automatic scaling behavior for a Pod replica set. To define an HPA object, you need to specify the parameters to ensure that the HPA works as intended, the main parameters are shown in Table 1.

Currently, the official version of Kubernetes only supports CPU as a scaling metric, and other types of scaling metrics are still in the testing stage. The automatic scaler HPA calculates the desired number of replicas by checking the CPU utilization of the corresponding Pod replica set and comparing it with a predefined target value, as shown below:

(1) Get the CPU usage of each replica in the replica set at the current time and sum up to calculate the CPU usage of the replica set, which is an absolute value, that is, how many units of CPU are used, and the usage unit is one thousandth
(2) Obtain the CPU allocation of each replica in the replica set, sum up and calculate the CPU allocation of the replica set, and calculate the CPU utilization of the replica set, which is calculated as the ratio of CPU usage to CPU allocation
(3) Calculate the desired number of replicas
(4) Check if the expected number of copies is greater than the upper limit MaxReplicas, and if it exceeds the upper limit, MaxReplicas will be the expected number of copies
(5) Check whether the expected number of replicas is less than MinReplicas, and if it is lower than the lower limit, MinReplicas is used as the expected number of replicas

In addition, to avoid system bumps caused by frequently triggering the scaling function, Kubernetes sets an energy-cooling time for autoscaling. By default, the interval between expansions is not less than 3 minutes, and the interval between scaling is not less than 5 minutes.

2.3. Problem Analysis of the Scale-Up Phase. Kubernetes requires a series of components to collaborate with each other to complete the scaling work, and the initialization time of pod t_{init} is calculated as follows:

$$t_{\text{init}} = \sum_{i=1}^{4} t_i. \tag{1}$$

When the application faces a large increase in requests, Kubernetes triggers the expansion and creates pods, and the time t_{init} is needed for the newly created pods to accept requests from users, as shown in Figure 2. At the moment of s1, the autoscaler checks that the load status of the pod replica set reaches the expansion standard and triggers the scale-up. At the

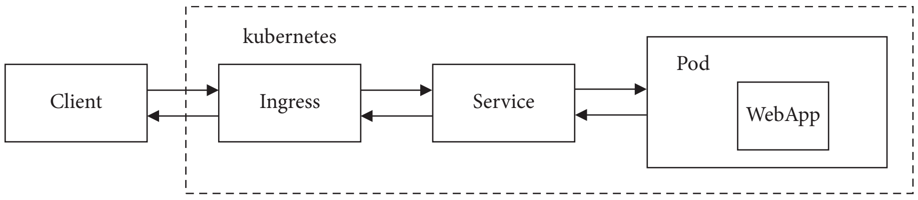


Figure 1: Web request workflow.

Table 1: HPA object parameters description.

Parameter name	Parameter value
MinReplicas	Minimum number of copies
MaxReplicas	Maximum number of copies
ScaleTargetRef	Scaling object
Target	Scaling indicators and target values

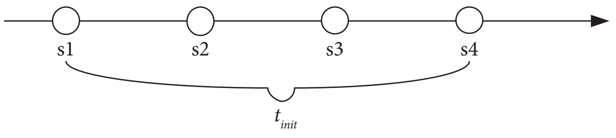


Figure 2: Scale-up time.

moment of s4, the scale-up is completed and the newly created replicas receive service requests. During the time of pod initialization, the replicas are overloaded and accumulate a large number of user requests, increasing user request time. Therefore, it is necessary to take the time of pod initialization into account when scale-up and prepare for the scale-up before the arrival of the load peak to reduce the response time of user requests, and thus ensure the stability of service quality.

2.4. Problem Analysis of the Scale-Down Phase. The purpose of the Kubernetes scale-down is essential to release the occupied resources by removing redundant pod nodes to improve the overall resource usage efficiency and reduce the overall application deployment resources. But in reality, although the application pod nodes are freed, the process only destroys the applications running on them, but the resources are not freed, as the underlying resource nodes are still within the Kubernetes resource pool. In a traditional private server room, the hardware purchased for the deployment of applications needs to go through preacquisition and shelving actions. Therefore, when the pod running on it is destroyed, it does not reduce the deployment and operation cost of these resources, and the only way to improve the overall utilization is to deploy other application services. In a public cloud environment, users only need to pay for these running resource instances, and when these idle nodes are higher than expected, they can effectively reduce operational costs by releasing these remaining nodes.

Kubernetes' flexible scaling is a natural fit with the public cloud's autoscaling, and Kubernetes' existing scaling strategy selects replicas to be deleted based on the priority of the replica state, unlike the preselection and preference process during expansion, which ignores the impact of deleting pods on cluster resources and actual business scenarios.

In order to achieve the scale-down of resource nodes, we expect to be able to concentrate as many pods as possible on certain nodes when scaling down and to shut down or delete the nodes of those pod nodes where fewer pod nodes are deployed by evicting the current node and calling the public cloud interface on it when it is in an unused state.

3. Container Scaling Policy Mode

The problem description for reinforcement learning using the Markov decision process consists of two main elements:

(1) Modeling the container scaling problem based on Markov decision models

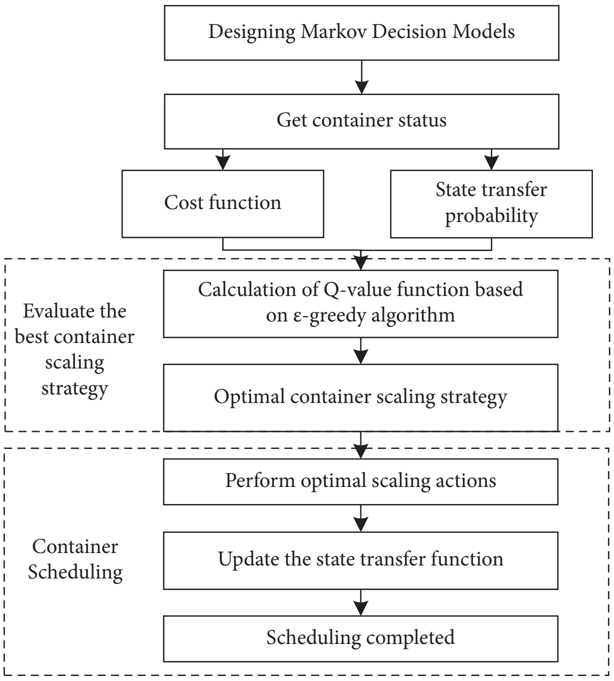
(2) Based on reinforcement learning algorithm to realize the iteration of the cost value and container scaling strategy in the process of container scaling, we evaluate the best container scaling strategy by solving the optimal Q-value function in the iterative process

The model is shown in Figure 3.

3.1. Modeling the Container Scaling Problem Based on Markov Decision Models. Markov decision processes are discrete-time stochastic control processes that provide a mathematical framework for modeling decision problems. Markov decision models are usually represented by five tuples $\{S, A, P, C, \gamma\}$. S denotes the state space, here the state refers to the container state. A denotes the action space, here the container scaling action. P denotes the state transfer function. $P(s_t, a_t)$ denotes the probability that the state s_{t+1} is at the next moment after the intelligence executes the action a_t in the current state s_t. C denotes the immediate payoff function. $C(s_t, a_t)$ denotes the reward received by the intelligence after the execution of action a_t. γ denotes the discount factor, where $\gamma \epsilon [0, 1]$. When the discount factor is closer to 0, it indicates that the intelligence places more importance on the short-term cumulative reward value; when the discount factor is closer to 1, it indicates that the intelligence places more importance on the long-term cumulative reward value.

The Markov model for container scaling consists of three steps: state space design, action space design, and cost function design.

3.2. Iteration of Cost Values and Container Scaling Policies during Container Scaling Based on Reinforcement Learning. The process of an agent interacting with the environment in reinforcement learning can be viewed as a time sequence. The agent has a starting state s_t, then does an action a_t, the



Designing Markov Decision Models
Get container status
Cost function
State transfer probability
Evaluate the best container scaling strategy
Calculation of Q-value function based on ε-greedy algorithm
Optimal container scaling strategy
Container Scheduling
Perform optimal scaling actions
Update the state transfer function
Scheduling completed

Figure 3: Container scaling policy mode.

environmental state changes to s_{t+1} and feeds back a reward r_{t+1}, and such interaction can go on forever.

In the reinforcement learning process, the change of state and the execution of a strategy have corresponding probabilities, indicating that they are all random events. The cumulative reward value obtained for each strategy execution trajectory is different, and the goal of reinforcement learning is to maximize the cumulative reward value by improving the strategy.

4. Construction of Markov Decision Model

4.1. State Space Design. The applications in the cluster are deployed and run in containers. The reinforcement learning intelligence perceives the state changes of the applications in the cluster environment and performs container scaling actions to effectively guarantee the quality of application services.

Define the state of the application at moment i as $s_i = (k_i, u_i, c_i)$, where k_i denotes the number of containers, and the range of k values is $\{1, 2, \cdots, K_{\max}\}$; $K_{\max}$ denotes the maximum number of copies of the application; u_i indicates the CPU utilization; and c_i indicates the amount of CPU computation granted to the container. Although u_i and c_i are real numbers, the discrete factors are set to disperse them for later strategy implementation. Assuming $u_i \epsilon \{0, \overline{u}, \cdots, L\overline{u}\}$ and $c_i \epsilon \{0, \overline{c}, \cdots, M\overline{c}\}$, $\overline{u}$ and $\overline{c}$ are suitable discrete factors. Define the state space S to store the state of all applications, which can be formulated as follows:

$$S = \{s_1, s_2, \ldots, s_n\}. \tag{2}$$

4.2. Action Space Design. The reinforcement learning intelligence senses the state of the application, performs container scaling operations, and optimizes the container scaling policy with the goal of minimizing expected costs.

For each state s in the application state space $S(s \epsilon S)$, there is a corresponding set of container scaling actions $A(s) \subseteq A$ that represent the container scaling actions performed by the intelligence in the application state s. A is the action space, defined as $A = \{-r, -1, 0, +1, +r\}$, consisting of 5 actions, where $+r$ indicates an increase in the amount of CPU resources and $-r$ indicates a decrease in the amount of CPU resources, i.e., $+1$ indicates a horizontal expansion to increase the number of container copies and -1 indicates a horizontal contraction to decrease the number of container copies, and $a = 0$ means to make no decision; action $A(s)$ is defined as follows:

$$A(s) = \{-r, -1, 0, +1, +r\}. \tag{3}$$

The action space A can be represented as follows:

$$A = \{A(s_1), A(s_2), \ldots, A(s_n)\}. \tag{4}$$

4.3. Cost Function Design. After a reinforcement learning intelligence makes an action a based on the current application's state s, the cost that the environment feeds the intelligence is also greatly related to the application's state s' at the next moment. We combine the different costs into a single weighted cost function, with different weights allowing us to express the relative importance of each cost term. Thus, we propose the immediate cost function $c(s, a, s')$ as a weighted value of several costs, representing the immediate cost spent to transform from state s to state s'. The weights of the immediate cost function range from $[0, 1]$, The mathematical expression of $c(s, a, s')$ is as follows:

$$\begin{aligned} c\left(s, a, s'\right) &= w_{\text{adp}} \frac{1_{\{\text{vertical-scaling}\}} c_{\text{adp}}}{c_{adp}} + w_{\text{perf}} \frac{1_{\{R(k+a_1, u', c+a_2) > R_{\max}\}} c_{\text{perf}}}{c_{perf}} + w_{\text{res}} \frac{(k + a_1)(c + a_2) c_{\text{res}}}{K_{\max} \bullet c_{\text{res}}}, \\ &= w_{\text{adp}} 1_{\{\text{vertical-scaling}\}} + w_{\text{perf}} 1_{\{R(k+a_1, u', c+a_2) > R_{\max}\}} + w_{\text{res}} \frac{(k + a_1)(c + a_2)}{K_{\max}}, \end{aligned} \tag{5}$$

where c_{adp} is the adaptation cost of the application to cover the cost of the application being unavailable when a certain container scaling operation is performed. c_{perf} denotes the cost of lost application performance, which is the cost to be paid when the response time of the application exceeds the response time limit $R_{\max}$. c_{res} is the cost of the resources used to run the application. The expense of this cost is proportional to the number of instances of the application and the share of CPU allocated to the container application. $1_{\{\ldots\}}$ is an indicator function. w_{adp}, w_{perf}, and w_{res} are nonnegative weights of different costs and $w_{\mathrm{adp}} + w_{\mathrm{perf}} + w_{\mathrm{res}} = 1$. In this paper, we consider these three weight settings are equally important; so, we have $w_{\mathrm{adp}} = w_{\mathrm{perf}} = w_{\mathrm{res}} = 0.33$. $R(k,u,c)$ is the application response time at state $s(k,u,c)$. In addition, the container scaling action a is decomposed into a horizontal scaling operation a_1 and a vertical scaling operation a_2.

5. Solving the Optimal *Q*-Value Function Based on the Iterative Process of Reinforcement Learning

When solving container scaling using reinforcement learning, the most important thing is how to solve the optimal Q-value function to evaluate the best container scaling policy to achieve optimal container scaling. There are many kinds of container scaling policies combining container state space and container scaling action space, and it is obviously not efficient enough to evaluate the policies by calculating the function values of each state-action pair one by one. This section focuses on computing Q-value functions using reinforcement learning algorithms.

First, we rely on the abovementioned system model and calculate the Q-value function directly using the Bellman equation:

$$Q(s,a) = \sum_{s' \epsilon S} p\left(s'|s,a\right)\left[c\left(s,a,s'\right) + \gamma \min_{a' \epsilon A(s_i)} Q\left(s_{i+1},a'\right)\right] \forall s \epsilon S, \forall a \epsilon A(s), \tag{6}$$

where γ is the discount rate. The unknown transfer probability and unknown cost consumption can be estimated empirically.

Estimating the transfer probability $p(s'|s,a)$ can be translated into estimating the transfer probability of CPU utilization $p(u_{i+1} = u'|u_i = u)$, so the transfer probability can be written as the following expression:

$$\begin{aligned} p\left(s'|s,a\right) &= P\left[s_{i+1} = \left(k',u',c'\right)|s_i = (k,u,c), a_i = a\right] \\ &= \begin{cases} p\left(u_{i+1} = u'|u_i = u\right) & k' = k + a_1 \wedge c' = c + a_2, \\ 0 \quad \text{otherwise}, \end{cases} \end{aligned} \tag{7}$$

where $a = (a_1,a_2)$ refers to the container scaling operation, including horizontal scaling operation and vertical scaling operation, which is defined based on the updated number of containers (a_1) and the updated CPU share (a_2). Since the CPU utilization u is taken in a discrete set, we will briefly denote the transfer probability as $P_{j,j'} = P[u_{i+1} = j'\overline{u}|u_i = j\overline{u}]$. $j, j' \epsilon [0,\ldots,L]$. Definition $n_{i,jj'}$ stands for the number of CPU utilization changes when the application changes from state $j\overline{u}$ to $j'\overline{u}$ in time interval $\{1,\ldots,i\}$, where j and $j' \epsilon [0,\ldots,L]$. The estimated value of the transfer probability in time interval i can also be noted in the form $\widehat{P}_{j,j'} = n_{i,jj'}/\sum_{l=0}^{L} n_{i,jl}$. Then, we can estimate the direct estimation of $\widehat{p}(s'|s,a)$ by (5).

When we estimate the immediate consumption cost $c(s,a,s')$, we observe that the immediate consumption cost is composed of two components, known and unknown costs, which can be written as follows:

$$c\left(s,a,s'\right) = c_k(s,a) + c_u\left(s'\right), \tag{8}$$

where $c_k(s,a)$ is a known cost depending on the current state and operation, and in this paper, it takes into account the adaptation cost and resource cost. $c_u(s')$ denotes the unknown cost, which depends on the next state s'. $c_u(s')$ is determined by the performance loss because our assumed application model is unknown and we estimate the unknown cost $c_u(s')$ online. Thus, at time i, the RL intelligences can directly acquire the costs c_i and estimate the immediate cost $c_{u,i}(s')$ of the next state at time i, which can be represented as follows:

$$c_{u,i}\left(s'\right) = c_i + c_{k,i}(s,a). \tag{9}$$

Then, update the unknown cost $\widehat{c}_{u,i}(s')$ using $c_{u,i}(s')$, which can be formulated as follows:

$$\widehat{c}_{u,i}\left(s'\right) \leftarrow (1-\alpha)\widehat{c}_{u,i-1}\left(s'\right) + \alpha c_{u,i}\left(s'\right). \tag{10}$$

The estimate of the unknown cost $\widehat{c}_{u,i}(s')$ is calculated based on the operation a in state s in (8). When the number of containers decreases, the CPU utilization increases and the CPU share decreases the cost spent to violate $R_{\max}$ is the expected cost of going from state $s = (k,u,c)$ to the next state $s' = (k',u',c')$. When updating $\widehat{c}_{u,i}(s'), \forall s \in S$, the following properties can be enforced:

$$\begin{aligned} \widehat{c}_{u,i}(s) &\le \widehat{c}_{u,i}\left(s'\right) \forall \mathrm{k} \ge \mathrm{k}', u \le u', c \ge c', \\ \widehat{c}_{u,i}(s) &\ge \widehat{c}_{u,i}\left(s'\right) \forall \mathrm{k} \le \mathrm{k}', u \ge u', c \le c'. \end{aligned} \tag{11}$$

From the above, the state transfer probabilities and cost functions can be obtained by estimation, so the Markov decision model for the container scaling problem in this paper is known. In other words, the reinforcement learning intelligence is "fully observing" the environment with known changes in the environment. The Q-value function is then computed using the policy iteration algorithm in model-based reinforcement learning to evaluate the optimal container scaling action. The core idea of the policy iteration algorithm is to use dynamic programming to solve the problem, so this paper chooses the greedy method to implement the calculation of the Q-value function and evaluate the container scaling strategy by the Q-value function. The ε-greedy strategy idea is that the selection behavior of an individual in a state is such that it can reach the state with the largest state value among all possible subsequent states. The state value here refers to the container scaling cost. In the ε-greedy strategy algorithm, the intelligence randomly selects the action with probability ε and chooses the best container scaling action with probability $1 - \varepsilon$. The strategy probability distribution can be written as follows:

$$\pi(s_t) = \begin{cases} \arg\max\limits_{a_t} & Q(s_t, a)\, p \le q, \\ a_{\text{random}} & \text{otherwise,} \end{cases} \tag{12}$$

where a_{random} means a randomly selected action, $p, q \in [0, 1]$ and p value determines the probability of exploration of the intelligence; the larger p value is, the smaller the probability of exploration by the intelligence. Finally, the reinforcement learning update strategy algorithm is given as the Algorithm 1.

The abovementioned pseudocode briefly describes the whole process of learning and updating the container scaling policy. Firstly, the current container state s is obtained, and the state transfer probability P and immediate cost function c are estimated based on the container state s. Then, for each state s is refined to the specific container scaling operation, the Q-value function is calculated with the goal of minimizing cost consumption and the optimal container scaling policy is evaluated to achieve optimal container scaling.

6. Simulation Results and Discussion

6.1. Experimental Environment. The hardware environment is a PC with i5 processor and 16G RAM. The software environment is Windows 10 operating system, and the programming language is Python 3.0. The container Cloud Platform cluster environment is a Kubernetes cluster, which contains one master node and three node nodes. The open source edge computing framework EdgeX Foundry is deployed on top of the Kubernetes cluster as a container.

6.2. Container Expansion Experiment. The effects of the two container scaling strategies are tested in the same experimental environment. The experimental parameter settings we used and the observed metrics of the final experimental results mainly include the change in the number of pod copies, the change in application CPU utilization, and the change in application response time.

6.2.1. Implementation of HPA Container Scaling Policy. The HPA autoscaling policy is chosen as the comparison policy to achieve automatic scaling of php-apache container applications in a clustered environment by creating HPA objects. HPA works under dynamic random workload requests, increasing or decreasing the number of replicas based on CPU resource metrics. Implementing container scaling using HPA policies requires configuring HPA parameters and creating HPA objects.

This command contains the scaling object parameter, the CPU utilization target value parameter, the maximum number of copies parameter, and the minimum number of copies parameter. The HPA object parameters are shown in Table 2.

6.2.2. Implementation Process of RLS Policy. The policy implementation process requires setting several parameter values for implementing policy learning and updating, which are the application response time limit value, the scaling time interval, the range of the number of container instances, and the greedy policy exploration probability.

We reasonably assume that applications receiving a large number of requests in the edge computing environment have high requirements for real-time response to requests, so this paper sets the response time limit value $R_{\max}$ to be no more than 140 ms at maximum; the automatic scaling service of Kubernetes, which is designed as a control loop, has a cycle time setting of 30 s in the controller (kube-controller-manager). In the comparison experiment, we test the effect of two container scaling strategies (RLS vs. HPA) in three aspects: resource utilization of the application, change in the number of pods, and response time of the application, so the execution time interval of both HPA and RLS is set to 30 s; considering the size of the state space of the reinforcement learning algorithm, the range of the container instances number is set to the maximum value of 10 and the minimum value is 1, which can satisfy the optimization goal of reinforcement learning and also alleviate the problem of state space explosion.

The specific learning and updating process of the policy is listed as follows:

(i) Continuously obtaining the number of real-time container instances, CPU utilization, and application response time for the container during the time interval to determine the current state.

(ii) The selection of the best action is solved using a greedy algorithm, where the high probability of exploration determines whether the best action can be selected quickly. The container scaling strategy of reinforcement learning aims to obtain the best container scaling action in the process of continuous learning and exploration, with emphasis on the later exploitation effect. Thus, using smaller exploration probabilities will result in the best action obtained by exploration, which is more likely to be exploited the next time, and small probabilities may be slow to explore upfront, but the best action selection will be better as time grows.

(iii) After executing the optimal scaling action, the current state, i.e., the number of container instances, CPU utilization, and application response time, is

```
(1) Update estimates $\widehat{\mathbf{P}}_{\mathbf{j,j'}}$ and $\widehat{\mathbf{c}}_{\mathbf{u,i}}(\mathbf{s_i})$
(2) for all $\mathbf{s} \in \mathbf{S}$ do
(3)     for all $\mathbf{a} \in \mathbf{A}(\mathbf{s})$ do
(4)     $Q(s,a) \leftarrow \sum_{s' \in S} \widehat{p}(s'|s,a) \cdot [\widehat{c}(s,a,s') + \gamma \min_{a' \in A(s_i)} Q(s',a')]$
(5)     end for
(6) end for
```

Algorithm 1: Update strategy algorithm based on reinforcement learning.

Table 2: HPA object parameters.

Parameter name	Parameter value
MinReplicas	1
MaxReplicas	10
scaleTargetRef	Php-apache
Target	50%

Table 3: Reinforcement learning parameter.

Parameter name	Parameter value
Greedy strategy exploration probability	0.06
$R_{\max}$	140 ms
Time interval	30 s
Range of container instances	Rounded from 1 to 10

recorded as a way to update the state transfer probability and the record of the cost spent.

(iv) After each update of the state transfer probability, all the actions and state records need to be updated.

The abovementioned implementation process, with continuous cyclic learning at time intervals, and as time grows, the effective information increases thereby enhancing the effectiveness of the container scaling strategy. The relevant parameter values are shown in Table 3.

6.3. Experimental Results and Analysis. Container scaling is used to improve the CPU utilization of the application, to ensure the quality of service of the application with the least cost of pod containers, and to avoid taking up too many cluster resources.

The reinforced learning container scaling strategy can better adapt to dynamic load changes, the allocation of pod resources can respond quickly compared to HPA, the early stage is a continuous learning phase, the number of pod changes varies greatly, but later tends to be a smooth state, and can ensure the quality of service of the application, and does not occupy too many resources. But HPA for the request reduction, the number of containers cannot be reduced in time, occupying too many cluster resources, and the application itself has low CPU utilization.

To observe the difference in container scaling results under two container scaling policies with the same application request load. The experiments are conducted according to the parameters designed above to achieve container scaling under the HPA container scaling strategy and reinforcement learning-based container scaling, respectively. The effects of the two container scaling strategies are compared by observing the change in the number of instances of container applications, the change in CPU utilization of container applications, and the change in response time of container applications under the application request load.

Figure 4 shows the trend of the number of container instances. From Figure 4, it can be seen that the container application load requests are randomly and dynamically changing and have a tendency to rise and fall sharply. In this random dynamic trend, the HPA container scaling policy has a more stable trend to meet the application load request with more container instances throughout. At the same time, the red box in Figure 4 also shows that the HPA policy has a time lag problem when scaling up and down, while the container instances under the reinforcement learning policy can flexibly adapt to dynamic workload changes in a timely manner, and with more valid information, the container scaling is more effective.

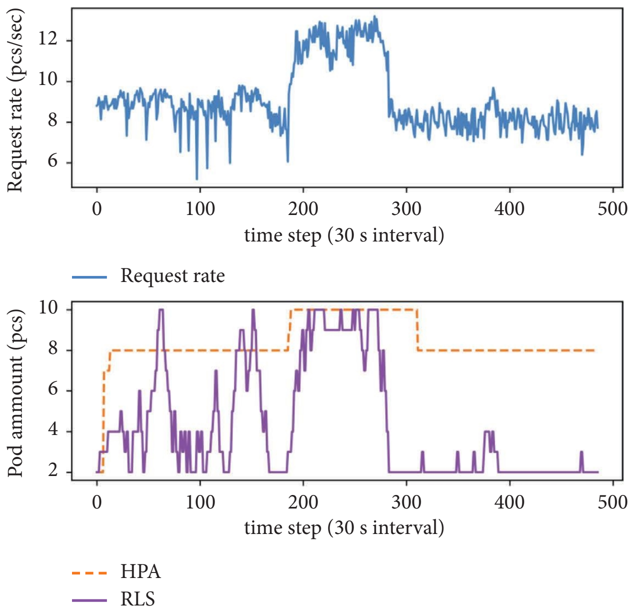


Figure 4: Trend of the number of container instances.

Figure 5 shows the trend of CPU utilization of container application. In Figure 5, it is obvious that the overall CPU utilization of the application is low under the HPA container scaling policy, and under the RLS container scaling policy, the dynamic change is obvious in the early stage due to less effective information, and it tends to be stable and high in the later stage.

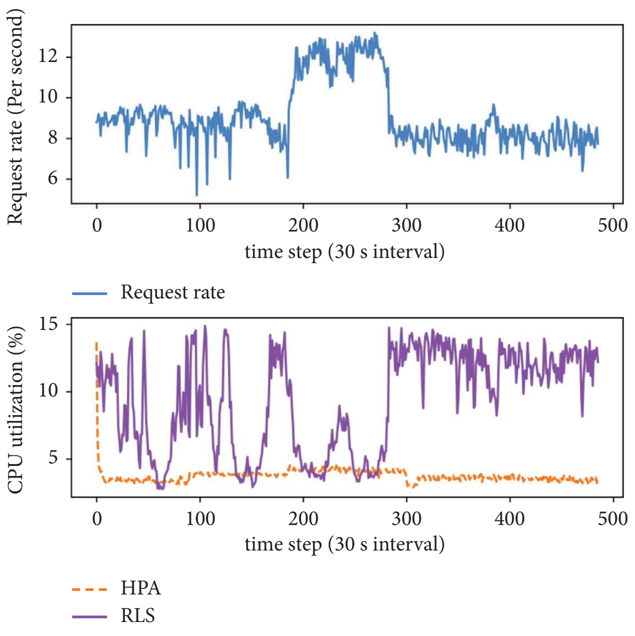


Figure 5: Trend of CPU utilization.

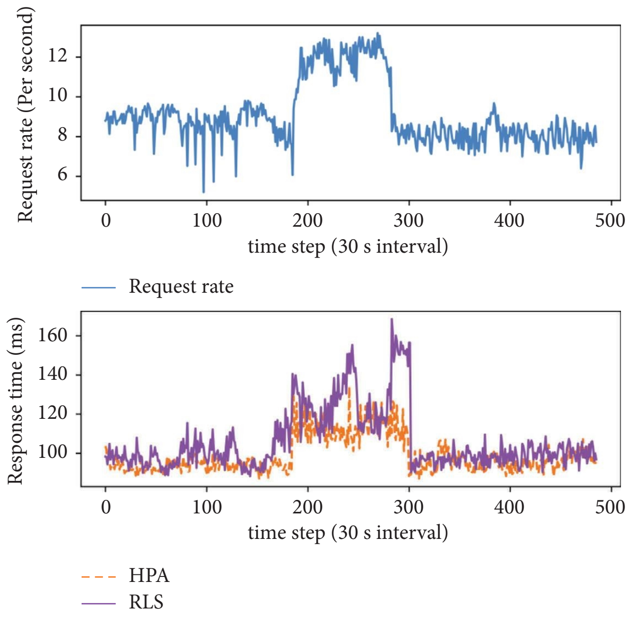


Figure 6: Trend of response time.

Figure 6 shows the trend of container application response time. In the early stage of RSL strategy, because the intelligent body does not interact with the environment for a long time, the effective information obtained is very little, and the optimal container scaling action is not obviously enough, so the application response time fluctuates a lot. In the later stage, with the continuous interaction and learning between the intelligence and the environment, the effective information obtained increases and the container scaling strategy is gradually optimized to select the container scaling action more accurately, thus meeting the application response time requirements.

Considering the limited resources in the edge computing environment, it is clear from the abovementioned analysis that the RLS strategy proposed in this paper has more advantages than the HPA strategy, which takes too much cluster resources as a condition to ensure the quality of service, and its timeliness is slightly lagged, its flexibility is poor, and its resource utilization is low. In contrast, the reinforcement learning strategy gradually optimizes the policy through continuous learning and decision making and finally guarantees the application service quality with less occupied resources, and it has high timeliness and high flexibility, which is more suitable for handling dynamic and randomly changing workloads in the edge computing environment.

7. Conclusions

This paper designs and implements a reinforcement learning-based container scaling strategy based on EdgeXFoundry, an edge computing framework, and Kubernetes, an open source container cloud platform, in conjunction with container scaling scenarios. Finally, we analyze the experimental results to prove that the reinforcement learning-based container scaling strategy can effectively guarantee the application service quality and improve the application resource utilization compared with the HPA automatic scaling strategy.

Data Availability

The data used to support the findings of this study are included within the article.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work was funded by the National Key R&D Program of China (Grant no. 2018YFB1703003), Natural Science Foundation of China (no. 61971347), Key Research and Development Program of Shaanxi Province (2022SF-353), Shaanxi Water Conservancy Technology Project (2020slkj-17), and Collaborative Innovation Center of Modern Equipment Green Manufacturing in Shaanxi Province (Grant no. 112-256092104).

References

[1] J. Bin, J. Li, and G. Yue, “Differential privacy for industrial Internet of things: opportunities, applications and challenges,” *IEEE Internet of Things Journal, to appear*, vol. 8.
[2] J. Fang and A. Ma, “IoT application modules placement and dynamic task processing in edge-cloud computing[J],” *IEEE Internet of Things Journal*, vol. 99, p. 1, 2020.
[3] M. Min, L. Xiao, and Y. Chen, “Learning-based computation offloading for IoT devices with energy harvesting,” *IEEE Transactions on Vehicular Technology*, vol. 68, no. 2, pp. 1930–1941, 2019.

[4] N. Soumyalatha and K. R. Manjunath, "Key technologies and challenges in iot edge computing," in *Proceedings of the 2019 Third International conference on I-SMAC*, pp. 61–65, Palladam, India, December 2019.

[5] C. K. K. Raymond, G. Stefanos, and P. J. Hyuk, "Cryptographic solutions for industrial internet-of-things: research challenges and opportunities," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 8, pp. 3567–3569, 2018.

[6] P. Claus, "Containerization and the PaaS cloud," *IEEE Cloud Computing*, vol. 2, no. 3, pp. 24–31, 2015.

[7] N. Makris, V. Passas, and C. Nanis, "On minimizing service access latency: employing MEC on the fronthaul of heterogeneous 5G architectures," in *Proceedings of the IEEE International Symposium on Local and Metropolitan Area Networks*, pp. 1–6, Paris, France, June 2019.

[8] R. Morabito, "Virtualization on Internet of things edge devices with container technologies: a performance evaluation," *IEEE Access*, vol. 5, pp. 8835–8850, 2017.

[9] M. Satyanarayanan, P. Bahl, and R. Caceres, "The case for VM-based cloudlets in mobile computing," *IEEE Pervasive Computing*, vol. 8, no. 4, pp. 14–23, 2009.

[10] F. Bonomi, R. Milito, and J. Zhu, "Fog computing and its role in the Internet of things," in *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, pp. 13–16, Helsinki, Finland, March 2012.

[11] Y. Yu, "Mobile edge computing towards 5G: vision, recent progress, and open challenges," *China Communications*, vol. 13, no. 2, pp. 89–99, 2016.

[12] M. B. Yassein, O. Alzoubi, and S. Rawasheh, "Challenges and issues of fog computing: a comprehensive review," *WSEAS Transactions on Computers*, vol. 19, pp. 86–97, 2020.

[13] X. Wang and Y. Wang, "Coordinating power control and performance management for virtualized server clusters," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 2, pp. 245–259, 2011.

[14] S. Chen, J. Wu, and Z. Lu, "A cloud computing resource scheduling policy based on genetic algorithm with multiple fitness," in *Proceedings of the 2012 IEEE 12th International Conference on Computer and Information Technology*, pp. 177–184, Chengdu, China, July 2012.

[15] Y. Jeong, K. E. Maria, and S. Park, "An energy-efficient service scheduling algorithm in federated edge cloud," in *Proceedings of the 2020 IEEE International Conference On Autonomic Computing And Self-Organizing Systems Companion*, pp. 48–53, Washington, DC, USA, August 2020.

[16] J. Feng, W. Zhang, and Q. Pei, "Heterogeneous computation and resource allocation for wireless powered federated edge learning systems," *IEEE Transactions on Communications*, vol. 70, no. 5, pp. 3220–3233, 2022.

[17] J. Feng, L. Liu, and Q. Pei, "Min-max cost optimization for efficient hierarchical federated learning in wireless edge networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 11, pp. 2687–2700, 2022.

[18] L. Liu, M. Zhao, and M. Yu, "Mobility-aware multi-hop task offloading for autonomous driving in vehicular edge computing and networks," *IEEE Transactions on Intelligent Transportation Systems, to appear*, vol. 24.

[19] S. Mao, L. Liu, and N. Zhang, "Reconfigurable intelligent surface-assisted secure mobile edge computing networks," *IEEE Transactions on Vehicular Technology*, vol. 71, no. 6, pp. 6647–6660, June 2022.

[20] W. Wei, H. Gu, W. Deng, and T. C. Abl, "A lightweight design for network traffic classification empowered by deep learning," *Neurocomputing*, vol. 489, pp. 333–344, 2022.

[21] W. Wei, R. Yang, and H. Gu, "Multi-objective optimization for resource allocation in vehicular cloud computing networks," *IEEE Transactions on Intelligent Transportation Systems, to appear*, vol. 23.

[22] C. M. Wu, R. S. Chang, and H. Y. Chan, "A green energy-efficient scheduling algorithm using the dvfs technique for cloud datacenters," *Future Generation Computer Systems*, vol. 37, no. 1, pp. 141–147, 2014.

[23] F. Rossi, M. Nardelli, and V. Cardellini, "Horizontal and vertical scaling of container-based applications using reinforcement learning," in *Proceedings of the 2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pp. 329–338, Milan, Italy, July 2019.