Received: 30 December 2022 | Revised: 14 September 2023 | Accepted: 21 September 2023

DOI: 10.1002/spe.3284

SPECIAL ISSUE PAPER

WILEY

DRS: A deep reinforcement learning enhanced Kubernetes scheduler for microservice-based system

Zhaolong Jian[1] | Xueshuo Xie[1,2] | Yaozheng Fang[1] | Yibing Jiang[1] | Ye Lu[3] | Ankan Dash[4] | Tao Li[1,2] | Guiling Wang[4]

[1]College of Computer Science, Nankai University, Tianjin, China

[2]State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China

[3]College of Cyber Science, Nankai University, Tianjin, China

[4]Department of Computer Science, New Jersey Institute of Technology, Newark, New Jersey, USA

Correspondence
Tao Li, College of Computer Science, Nankai University, Tianjin, China.
Email: litao@nankai.edu.cn

Funding information
National Key Research and Development Program of China, Grant/Award Number: 2018YFB2100304; National Natural Science Foundation, Grant/Award Number: 62272248; Open Project Fund of State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Grant/Award Numbers: CARCH201905, CARCHA202108; Zhejiang Lab, Grant/Award Number: 2021KF0AB04; Tianjin Graduate Scientific Research Innovation Project, Grant/Award Number: 2022BKY028

Summary

Recently, Kubernetes is widely used to manage and schedule the resources of microservices in cloud-native distributed applications, as the most famous container orchestration framework. However, Kubernetes preferentially schedules microservices to nodes with rich and balanced CPU and memory resources on a single node. The native scheduler of Kubernetes, called Kube-scheduler, may cause resource fragmentation and decrease resource utilization. In this paper, we propose a deep reinforcement learning enhanced Kubernetes scheduler named DRS. We initially frame the Kubernetes scheduling problem as a Markov decision process with intricately designed *state*, *action*, and *reward* structures in an effort to increase resource usage and decrease load imbalance. Then, we design and implement DRS mointor to perceive six parameters concerning resource utilization and create a thorough picture of all available resources globally. Finally, DRS can automatically learn the scheduling policy through interaction with the Kubernetes cluster, without relying on expert knowledge about workload and cluster status. We implement a prototype of DRS in a Kubernetes cluster with five nodes and evaluate its performance. Experimental results highlight that DRS overcomes the shortcomings of Kube-scheduler and achieves the expected scheduling target with three workloads. With only 3.27% CPU overhead and 0.648% communication delay, DRS outperforms Kube-scheduler by 27.29% in terms of resource utilization and reduces load imbalance by 2.90 times on average.

KEYWORDS

deep reinforcement learning, Kubernetes scheduler, microservice scheduling, resource awareness

1 | INTRODUCTION

With the development of cloud computing and container ecology, cloud-native has become a new computing paradigm.[1,2] The microservice architecture is used by developers to create custom apps, which are then packaged as containers and sent to the cloud servers for execution. The microservice architecture uses multiple light-weight and loosely-coupled

Abbreviations: A3C, Asynchronous Advantage Actor-Critic; DPG, Deterministic Policy Gradient; DQN, Deep Q-Network; MDP, Markov decision process; PPO, Proximal Policy Optimization; SAC, Soft Actor-Critic; SARSA, State-Action-Reward-State-Action; TRPO, Trust Region Policy Optimization;

microservice components to replace the traditional integrated application. These microservices communicate with each other through the network.[3,4] In this paradigm, developers can focus on developing and delivering microservices without paying attention to the underlying infrastructure. Since microservice architecture has greatly increased the number of microservices, efficient management of these microservices becomes an essential research problem.[5-7] Typically, these microservices are managed and scheduled by the container orchestration frameworks, such as Kubernetes[*], Docker Swarm[†], and Mesos.[8] Among them, Kubernetes launched by Google is the most famous one and is widely used as the infrastructure of mainstream cloud service providers (e.g., Alibaba, Amazon, and Microsoft). It has become the de facto standard of cloud-native.[9]

A major challenge in running a big Kubernetes cluster is resource scheduling.[10,11] A smart scheduling algorithm can assist companies in allocating resources fairly for each microservice and increasing cluster resource utilization, both of which can lower costs for businesses. However, Kube-scheduler adopts a simple scheduling algorithm. Kube-scheduler first filters out nodes with insufficient resources according to the resource requirement of the microservice. Then it scores each node using a set of predefined policies and selects the node with the highest score to execute the microservice. The core policy of Kube-scheduler is to balance the usage of CPU and memory on the node. Node with more CPU and memory resources and similar CPU and memory utilization will be preferred. Kube-scheduler has the following defects. Firstly, Kube-scheduler only focuses on the CPU and memory utilization and ignores other important resources, such as network and disk. When running network-intensive and io-intensive applications, Kube-scheduler may lead to uneven load between nodes. Secondly, Kube-scheduler considers the resource utilization of a single node rather than the entire cluster. In a large Kubernetes cluster, using the Kube-scheduler may cause resource fragmentation, decrease resource utilization, and further increase the cost of the enterprise. Therefore, Kubernetes urgently needs a load-balanced resource scheduling algorithm to alleviate the above weaknesses.

In both industry and academia, the resource scheduling problem is considered as an NP-hard problem.[12,13] In different scenarios, since the workload of the cluster is different, the application characteristics are also different. And with different scheduling targets, it is difficult to design a general scheduling algorithm. For the resource scheduling problem, the traditional solution is to design heuristic algorithms. The heuristic algorithms prioritize generality and are easy to understand and implement. Examples include Round-Robin,[14] First Fit,[15] and Max-Min, which perform well under specific workloads. However, heuristic algorithms usually use fixed parameters and scheduling policies. They can not be adapted to the changing environment, and thus can not achieve the desired performance. Some research proposes meta-heuristic algorithms to improve the heuristic algorithms. They combine heuristic algorithms with random algorithms and local search algorithms, such as the ant colony algorithm,[16] simulated annealing,[17] and genetic algorithm.[18] However, designing these meta-heuristics is complex and cumbersome. Generally, people first design a simple heuristic algorithm according to the scheduling scenario. Then they try to adjust the parameters of the heuristic algorithm according to the application characteristics and the scheduling target. This process relies on repeated manual testing and tuning, which also requires the developer to have expert knowledge about the cluster status and workload.[19-21]

Recently, since reinforcement learning has shown good performance in making sequential decisions, it has been successfully applied to solve the resource scheduling problem of the computing cluster.[22-24] In this paper, we introduce the advantage of reinforcement learning to Kubernetes scheduling and propose DRS, a Deep Reinforcement learning based Kubernetes Scheduler. We first model the Kubernetes scheduling problem as a Markov decision process. We respectively define the resource usage and the schedulable nodes as the state space and the action space to translate the system status into some learnable features. Considering the deficiency of Kube-scheduler, we use the utilization of network and disk as a part of the state in DRS. We use the changes in overall resource utilization and the load imbalance between nodes as the feedback of the scheduling action. After taking each scheduling action, DRS perceives the change of the system state and gets feedback to learn the effect of the action. With multi-dimensional resource awareness, DRS can analyze the potential impact of the scheduling action on the usage of each resource. With the meticulously designed the model, DRS can learn scheduling policy through interaction with the Kubernetes cluster and achieve the high-level scheduling target for improving the average resource utilization and balancing the utilization of different nodes. Our concrete contributions can be summarized as follows:

1. We model the scheduling problem of Kubernetes as a Markov decision process. DRS uses the resource usage of each node and the resource requirements of pod as the *state* and uses the schedulable node as the *action space*.

[*] http://kubernetes.io/.

[†] https://docs.docker.com/engine/swarm/.

After taking action, DRS calculates the average resource utilization (*AvgUtil*) and the standard deviation of resource utilization between nodes (*Imbalance*) as the *reward* of *action*. With this model, DRS scheduler can automatically learn the expected scheduling policy from the running information of the system, without relying on expert knowledge.

2. We propose a load-balancing scheduling method based on multi-resource awareness. DRS monitors six easy-to-obtain metrics to measure the usage of CPU, memory, network, and disk, including CPU utilization, memory utilization, package receiving rate, package transmission rate, disk reading rate, and disk writing rate. This method enhances the resource awareness of DRS with only 3.27% CPU overhead.
3. We design DRS into the scheduler, monitor, and decision maker, and implement them in a Kubernetes cluster with five nodes. A comprehensive assessment shows that compared with Kube-scheduler, DRS can improve the overall resource utilization by 27.29% and reduce the load imbalance by 2.90× on average, with only 0.648% communication latency.

The rest of this paper is organized into the following sections. Section 2 introduces the preliminary information about Kubernetes scheduling and deep reinforcement learning. Section 3 introduces the scheduling problem of DRS and models it as a reinforcement learning process. Section 4 presents the basic design of DRS and introduces the implementation of each component. Performance evaluation and result discussions are given in Section 5. Section 6 reviews the research work related to Kubernetes scheduling optimization and reinforcement learning scheduling. The conclusion is drawn in Section 7.

2 | PRELIMINARY

In this section, we introduce the technical background related to our work. We first introduce the knowledge of Kubernetes and the workflow of Kube-scheduler. Then, we introduce the concept of reinforcement learning and its basic principle. Finally, we describe the detail of deep reinforcement learning and the DQN algorithm used in DRS.

2.1 | Kubernetes

Kubernetes is an open-source container orchestration framework launched by Google. With support for container scheduling, lifecycle management, elastic expansion, and network management, Kubernetes can help users to manage containerized applications easily and efficiently. As shown in Figure 1, the nodes in a Kubernetes cluster can be divided into two types: master and worker. The master node runs key components responsible for cluster management (Controller Manager), container scheduling (Scheduler), and state storage (etcd), forming the control plane of the cluster. The worker node is the computing node of the cluster, which can communicate with the control plane by running the Kubelet and Kube-proxy components. The worker is responsible for actually executing the user-deployed containerized applications(Pods). These pods will run in the container runtime (e.g., docker, contained, CRI-O) of the worker nodes.

Pod is an important concept in Kubernetes, which is a container group composed of one or more containers. These containers can share virtual environment resources such as Namespace and Cgroup in the pod. With the concept of pod, users can specify the image and resource required by the application to generate a pod and submit it to the master node. The scheduling process is also shown in Figure 1. Kubernetes divide the scheduling process into the scheduling cycle and the binding cycle and each cycle is further divided into serval stages. The scheduler on the master node organizes pods waiting for scheduling as a queue. The scheduling cycle is a serial process in which the scheduler fetches the pod from the scheduling queue one by one. The functions of each stage in the scheduling cycle are as follows:

Stage 1: Pre-Filter. As part of node filtering, the scheduler preprocesses the information of the pod and the nodes at this stage.
Stage 2: Filter. At this stage, the scheduler filters out all nodes that cannot meet the resource requirements of the pod.
Stage 3: Post-Filter. At this stage, the reserved nodes are considered schedulable nodes. This stage is usually used for notification.
Stage 4: Scoring. At this stage, the scheduler scores each node using the pre-defined rules.
Stage 5: Normalize Scoring. At this stage, the scheduler normalizes the node score to [0,100] as the final score.
Stage 6: Reserve. At this stage, the scheduler reserves resources on the chosen node to prevent resource overruns from asynchronous binding.

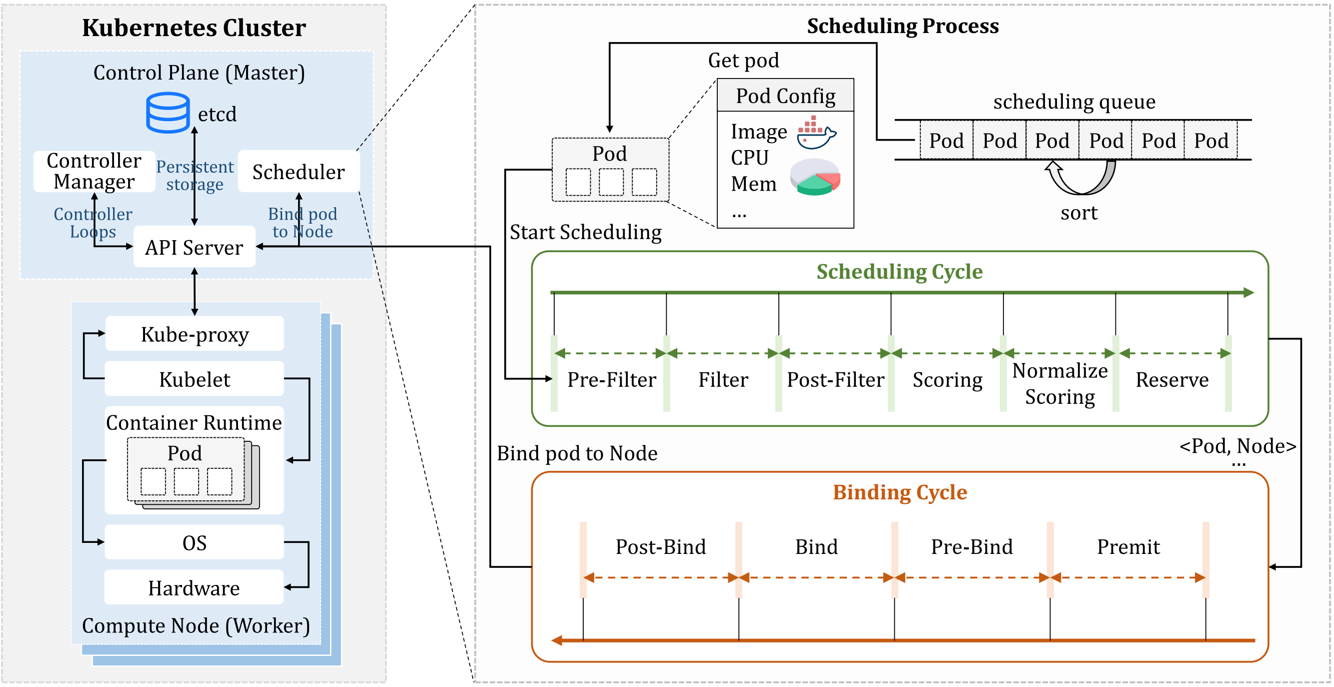


FIGURE 1 The architecture and scheduling process of Kubernetes.

At the end of the scheduling cycle, the scheduler informs the API server of the scheduling result. Then the pod will start the binding cycle, which is an asynchronous process. The functions of each stage in the binding cycle are as follows:

Stage 1: Permit. At this stage, the pod binding will be delayed or prevented until at appropriate time.
Stage 2: Pre-Bind. This is a pre-processing stage, it is used for preparing the resources on the chosen node.
Stage 3: Bind. At this stage, the pod will be bound to the chosen node.
Stage 4: Post-Bind. Since the scheduler has reserved resources on the node at the reserve stage, these resources will be released at this stage if the binding fails.

Users can customize the functionality of the above stages using the Kubernetes Scheduling Framework (KSF) to achieve customized scheduling. DRS is achieved by designing our own filtering and scoring function.

2.2 | Deep reinforcement learning

2.2.1 | Reinforcement learning process

Reinforcement learning is a branch of machine learning. Unlike supervised learning and unsupervised learning, reinforcement learning does not require a deterministic label for training data. Rather, it learns the optimal action policy at a given environment state by interacting with the environment as an agent. The reinforcement learning process can be described as a Markov decision process (MDP), which can be represented by a five-tuple $< S, A, R, P, \rho_0 >$:

1. S represents the *state* space, which contains all the possible states of the environment.
2. A represents the *action* space, which describes the possible actions of the agent.
3. R represents the *reward* function. At time t, the reward is given based on the agent's action and the change in the environment state:

$$r_t = R(s, a, s') \tag{1}$$

TABLE 1 Categories of reinforcement learning algorithms.

Type	Value-based	Policy-based	Actor-Critic
Algorithm	Q-learning,[25] SARSA,[26] DQN[27-30]	REINFORCE,[31] TRPO,[32] PPO[33]	A3C,[34] DPG,[35] SAC[36]
Mechanism	① Calculate the value function ② Select the action with the maximum value	① Maximizes the cumulative return selection action ② Update the policy	① Actor (policy-based) updates the policy and selects an action ② Critic (value-based) calculates the value with the action and updates each step
Advantage	① High sample utilization ② Value function estimation with a small variance ③ Avoid local optimum	① Easy computable policy function ② Random exploratory ③ Stable and easy to converge	① High sample utilization ② Value function estimation with a small variance ③ Fast training speed
Disadvantage	① Easy to overfitting ② Limited complexity in dealing with problems ③ Poor convergence properties	① Low sample utilization ② Easy to fall into local optimum ③ Inefficient policy evaluation with large variance	① Poor stability ② Sensitive to hyperparameters
Applicable scenario	Discrete action space	Discrete/continuous action space	Discrete/continuous action space

4. P is the transition function of the environment state. When the environment state is s and the agent takes action a, we use $P(s'|s, a)$ to represent the probability that the environment state changes from s to s'.
5. ρ_0 represents the initial state of the environment.

The agent observes the state of the environment and decides which action to take according to its existing policy. The policy is similar to a function that maps the state to an action. This action is applied to the environment and the state of the environment will be changed. This change is generally random. After the change of environment state, the agent can observe the new state and get a feedback signal, called *reward*. Reward is used to measure the quality of the action taken by the agent. The agent adjusts the policy according to the reward and tries to find an optimal policy that can help it obtain the maximum cumulative reward:

$$max \sum_{t=1}^{T} \gamma^t r_t \tag{2}$$

In this equation, r represents the reward and γ represents the discount factor. When $0 < \gamma < 1$, it means that the short-term reward is more important than the future reward.

2.2.2 | Mainstream reinforcement learning algorithms

According to the agent's action selection method, reinforcement learning algorithms can be classified into three categories: value-based, policy-based, and actor-critic. Table 1 presents a summary of the key characteristics of mainstream reinforcement learning algorithms. The value-based approach entails implicitly constructing the optimal policy by deriving the optimal value function and selecting the action associated with the maximum value. The pioneering deep reinforcement learning algorithm, known as the Deep Q-Network (DQN), has expanded the applicability of the value-based method to high-dimensional and continuous-space problems.[27] Subsequently, numerous variants of the DQN algorithm have been developed.[28-30] The policy-based algorithms directly explore the optimal strategy throughout the value function and update the policy parameters by maximizing the cumulative reward. The actor-critic algorithm combines the value-based (critic) and policy-based (actor) methods and simultaneously learns the policy and value function. The actor adjusts the policy based on the feedback received from the critic's value function, while the critic updates the value function through a single-step update mechanism.

Owing to the distinct mechanisms, these three types of algorithms possess their own set of advantages and disadvantages, rendering them suitable for different scenarios. The value-based algorithms exhibit high sample utilization, and low variance in value function estimation, and are less susceptible to local optima. However, this algorithm is only applicable to discrete action space problems, is prone to overfitting, and is limited in its capacity to handle complex problem domains. Additionally, the value-based algorithm suffers from poor convergence properties due to its sensitivity to value function changes. In contrast to value-based algorithms, policy-based algorithms can handle both discrete and continuous space problems and demonstrate better convergence properties. Nevertheless, the policy-based algorithms typically exhibit large variance, low sample utilization, and high susceptibility to fall into local optima. Actor-critic, considered a variant of the policy-based method, incorporates the value function as a reference for policy gradient, addressing the issue of estimation variance found in policy-based algorithms. Actor-critic algorithms combine the strengths of both policy-based and value-based algorithms, resulting in low value function estimation variance, high sample utilization, and fast training speed. However, the actor-critic algorithms have some inherent drawbacks, such as the actor's limited exploration of samples and the critic's susceptibility to overfitting. Furthermore, when combined with the actor, the critic faces challenges in convergence and exhibits inferior convergence properties. In this paper, we focus on the resource scheduling problem of the Kubernetes cluster. We employ a reinforcement learning algorithm to select the scheduling action and the action space is discrete. With limited historical data and the small sample size, the problem aligns well with the characteristics of the value-based reinforcement learning algorithms. Hence, we utilize the value-based DQN algorithm in DRS and delicately design the model to obtain the optimal value function and finally improve the scheduling effect.

2.2.3 | Q-learning and deep Q networks

Q-Learning is a value-based reinforcement learning algorithm that uses the state-action function $Q^{\pi}(s,a)$ to represent the value function more precisely.[25] $Q^{\pi}(s,a)$ represents the expected reward that the agent can obtain from taking action a with state s and following policy π:

$$Q^{\pi}(s,a) = E_{\pi}[G_t|S_t = s, A_t = a] = E_{\pi}\left[\sum_{k=0}^{\infty}\lambda^k R_{t+k+1}|S_t = s, A_t = a\right] \tag{3}$$

Accordingly, the optimal state-action function can be defined as:

$$Q_*(s,a) = \max_{\pi} Q_{\pi}(s,a) \forall s \in S, \forall a \in A \tag{4}$$

The optimal policy can be achieved by greedily selecting actions that maximize the value of the state-action function:

$$\pi_* = \arg\max_{a} Q_*(s,a) \tag{5}$$

On this basis, Google DeepMind combines Q-Learning with the deep neural network and proposes Deep Q Networks (DQN) algorithm.[27] The detail of the DQN algorithm is shown in Algorithm 1. When the state space and action space are large, using the Q-Learning algorithm will cause dimension disaster, a solution to find an approximate function of Q function:

$$Q_{\theta}(s,a) \approx Q_{\pi}(s,a) \tag{6}$$

DQN algorithm uses the deep neural network to fit the function $Q_{\theta}(s,a)$. Figure 3 shows the entire process of deep reinforcement learning using the DQN algorithm.

3 | MODEL DESIGN

To realize the load-balancing scheduling of Kubernetes with reinforcement learning, we model the scheduling process as a Markov decision process. Following the Markov property, we believe that the state of the cluster at a certain time is only related to the state at the previous time and the resource requirements of the pods. We delicately design the *state*, *action*, and *reward* as follows:

Algorithm 1. DQN algorithm[27]

```
Initialize experience pool $D$ to capacity $N$
Initialize action-value function $Q$ with random weights
for episode = 1, M do
    Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
    for $t = 1, T$ do
        With probability $\epsilon$ select a random action $a_t$, otherwise select $a_t = \max_a Q_\theta(\phi_t, a)$
        Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$
        Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma max_{a'} Q_\theta(\phi_{j+1}, a') & \text{for non-terminal } \phi_{j+1} \end{cases}$
    end for
end for
```

3.1 | State

We consider a cluster with n nodes, $State_t$ represents all the possible states of the environment at a certain time t. $State$ needs to reflect the current resource utilization of the cluster, as well as the resource requests of the pod which is waiting for scheduling. Therefore, $State_t$ is defined as a collection of the resource utilization of each node ($Node_t^i$) and the resource requests of the pod (Pod_t). $State_t$ is a one-dimensional vector, in which each value is normalized to [0,100].

$$State_t = [Node_t^1, Node_t^2, \ldots, Node_t^n, Pod_t] \tag{7}$$

For the resource utilization of the node $Node_t^i$, we consider not only the utilization of CPU and memory but also the utilization of network and disk. We use six metrics to reflect the utilization of these four resources for easy measurement. The metrics include CPU utilization (CPU_t^i), memory utilization (Mem_t^i), the package receiving and transmission rate of the network card ($Recv_t^i$, $Tran_t^i$), and the reading and writing rate of the disk ($Read_t^i$, $Write_t^i$). Due to the instability of the real-time utilization, we use the average utilization of several samples to express these metrics. The timeframe of the sample should be small to reflect the real-time state changes and consider the characteristics of the resources and applications. In DRS, the utilization of the CPU and network exhibit rapid and substantial fluctuations, so we collect the data five times with intervals of 0.1 s. Conversely, the utilization of memory and disk tends to be stable. Thus, we collect these data three times at intervals of 0.1 s.

$$Node_t^i = [CPU_t^i, Memory_t^i, Network_t^i, Disk_t^i] = [CPU_t^i, Mem_t^i, Recv_t^i, Tran_t^i, Read_t^i, Write_t^i] \quad i = 1, 2, \ldots, n \tag{8}$$

For the resource request of pod Pod_t, we also consider the above six metrics. We calculate the utilization according to the resource requirements in the configuration file of the pod and its running history.

$$Pod_t = [CPU_t, Memory_t, Network_t, Disk_t] = [CPU_t, Memory_t, Recv_t, Tran_t, Read_t, Write_t] \tag{9}$$

3.2 | Action

The scheduler plays the role of the agent in reinforcement learning. It will take action each time when receiving a new pod scheduling request. The agent's action is to select a node for the pod to execute, so the action space is all the possible nodes in the cluster.

$$Action_t \in \{node_i | i = 1, 2, \ldots, n\} \tag{10}$$

As shown in Section.2.1, the decision-making process occurs in the scheduling cycle. After the node is selected, the scheduler will enter the binding cycle. If the pod cannot be bound to the selected node, it will be added to the back off the queue for rescheduling. The time interval for rescheduling will gradually increase with the number of failures.

3.3 | Reward

The reward function directly reflects the target of the scheduling algorithm. Therefore, the reward function of DRS should give higher rewards for decisions that can improve the average resource utilization of the cluster and reduce the degree of load imbalance. We use $Util_t^i$ to represent the resource utilization of node i at time t. We use the symbol C to represent the number of distinct resource types under consideration. DRS considers six resource utilization types, thereby $C = 6$. In the subsequent processing, we normalize the resource utilization to distribute it evenly in [0,100]. Therefore, we use the average value of the six resource utilization to represent $Util_t^i$.

$$Util_t^i = \frac{\sum_{c=1}^{C} (Util_c)_t^i}{C}, Util_c \in \{CPU, Mem, Recv, Tran, Read, Write\} \tag{11}$$

$AvgUtil_t$ represents the overall resource utilization of the cluster, that is, the average of the resource utilization on each node ($Util_t^i$).

$$AvgUtil_t = \frac{\sum_{i=1}^{n} Util_t^i}{n} = \frac{\sum_{i=1}^{n} \sum_{c=1}^{C} (Util_c)_t^i}{nC} \tag{12}$$

We calculate the standard deviation of the utilization of each resource on each node and then sum the weighted values to indicate the degree of load imbalance at time t. It can be defined as Equation (13). When learning the scheduling policy, DRS takes into equal consideration the usage of CPU, memory, network, and disk. The normalization for the values of the six metrics ensures approximate values across them. Consequently, the $weight^k$ is set to $\frac{1}{c}$.

$$ImBalance_t = \sum_{c=1}^{C} weight^c * std((Util_c)_t^1, (Util_c)_t^2, \ldots, (Util_c)_t^n) \tag{13}$$

Finally, we can define the reward function as follows, in which α, β are the factors.

$$Reward_t = R(State_t, Action_t) = \alpha AvgUtil_t - \beta ImBalance_t \tag{14}$$

α and β are the empirical values set according to the cluster state, used to scale the $AvgUtil$ and $Imbalance$ to a similar size.

4 | IMPLEMENTATION

In this section, we first give an overview of DRS and introduce the complete scheduling process using DRS. Then, we describe the implementation details of DRS decision maker and its training process. Finally, we introduce the method to obtain the utilization of each resource and the monitoring algorithm of DRS monitor.

4.1 | DRS overview

The architecture overview of DRS is shown in Figure 2. We deploy DRS scheduler in the Kubernetes cluster as the second scheduler using the Kubernetes Scheduling Framework. DRS scheduler follows the Kubernetes scheduling process mentioned in Section 2.1. In addition to DRS scheduler, DRS also includes two modules: DRS decision maker and DRS monitor. DRS decision maker is a module independent of Kubernetes, which continuously receives requests from the scheduler. DRS monitor is a module that runs on each worker node of the cluster. It continuously monitors and records the usage of the CPU, memory, network, and disk on the node, and will return the current state of the cluster when receiving a state acquisition request.

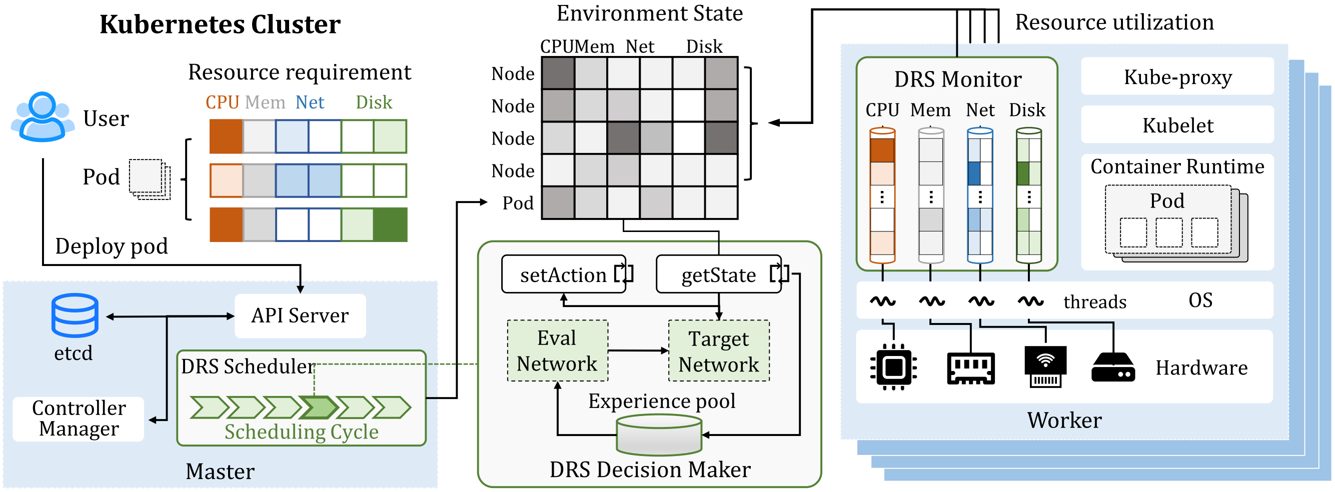


FIGURE 2 The architecture overview of DRS.

We give an example to show the complete process of pod scheduling using DRS. Firstly, the user specifies the image file and resource requirements to deploy the application as a pod. Then, the user submits the request to deploy the pod to the master node. At the filter stage of the scheduling cycle, the DRS scheduler on the master node filters out the nodes with insufficient resources according to the resource requirement of the pod. Next, DRS scheduler sends the request to DRS decision maker at the scoring stage of the scheduling cycle. The request contains the resource requirements of the current pod. After receiving the request, the DRS decision maker sends a message to the DRS monitor on the worker nodes to obtain the node resource utilization. When receiving the return of all nodes, the decision maker will aggregate and normalize the node state and the pod resource requirements into the environment state vector $State_t$. The color of the state in Figure 2 represents the degree of resource dependence. The darker the color, the higher the resource utilization. Then, the target network of DRS decision maker gives the action $Action_t$ and sends it to the scheduler. DRS scheduler scores the selected node with the highest score and continues the scheduling cycle. After the pod is bound to the selected node, the microservice can start running.

4.2 | DRS decision maker

DRS decision maker is the core component of DRS. The decision maker communicates to the DRS scheduler and the DRS monitor on each worker node through socket connections. The decision maker runs the DQN algorithm. The detail of the training process is shown in Algorithm 2 and Figure 3. The decision maker maintains an experience pool D with a capacity of N to store historical information about the environment. In our experiments, N is set to 300. In the beginning, the decision maker initializes the Q function with random values, which is used for action selection. The decision maker establishes socket connections with other components for subsequent communication. When the training process starts, the decision maker continuously listens for messages from DRS scheduler. On receiving the scheduling request, the decision maker processes the pod information. Then the decision maker sends requests to DRS monitor to obtain the current node state $Node_t^i$ and generate the cluster state vector $State_t$. The values in the vector are normalized according to the preset utilization range. DRS decision maker uses all possible actions to calculate Q, and select the maximum Q value using ϵ-greedy strategy. With ϵ probability, it selects an action randomly. After returning the action $Action_t$ to the scheduler, the decision maker will wait for T_i, and then obtain the state of each node again. We set T_i to 5 s, which is to ensure that the pod has been bound to the corresponding node and started running. $(State_t, Action_t, Reward_t, State_{t+1})$ of this scheduling process will be stored in the experience pool. When the experience pool is full, the decision maker will sample random transitions from the experience pool and start training the neural network. The evaluation network evaluates the losses and performs a gradient descent step. The target network will copy the parameters of the evaluation network for weight update every K iteration. In our implementation, K is set to 50. This process will repeat until the end of training. The training process of the DRS decision maker will repeat M epochs, and each epoch waits to execute the scheduling request for T times (same as T in Algorithm 1). Therefore, the time complexity of the DRS algorithm is approximately $O(MT)$, which is related to the number of epochs and the number of time steps.

Algorithm 2. The training algorithm of DRS decision maker

```
Initialize experience pool $D$ to capacity $N$
Initialize action-value function $Q$ with random weights
Initialize socket connections with DRS monitor and DRS scheduler
for episode = 1, M do                                                        ▷ Initialize the environment state
    for t = 1, T do
        if get scheduling request at time $t$ then
            Get state of each worker node $\{Node_t^1, Node_t^2, ..., Node_t^n\}$                    ▷ Step ① in Figure 3
            Combine the node states with the pod resource requirements as the state vector $State_t$
            With probability $\epsilon$ select a random action $Action_t$, otherwise select $Action_t = \max_a Q_\theta(\phi_t, a)$
            Return the result $Action_t$ to DRS scheduler                                      ▷ Step ② in Figure 3
            Get state of each worker node $\{Node_{t+1}^1, Node_{t+1}^2, ..., Node_{t+1}^n\}$ after time $T_i$
            Calculate the reward $Reward_t$
            Store transition $(State_t, Action_t, Reward_t, State_{t+1})$ in $D$                ▷ Step ③ in Figure 3
            if $D$.counter > capacity $N$ then
                Sample random minibatch of transitions from $D$                              ▷ Step ④ in Figure 3
                Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma max_{a'} Q_\theta(\phi_{j+1}, a^{'}) & \text{for non-terminal } \phi_{j+1} \end{cases}$    ▷ Step ⑤ in Figure 3
                Perform a gradient descent step on $(y_j - Q_\theta(\phi_j, a_j))^2$             ▷ Step ⑥ in Figure 3
            end if
        end if
    end for
    Copy parameters of eval Network to target Network per $K$ iterations            ▷ Step ⑦ in Figure 3
end for
```

4.3 | DRS monitor

According to our model, DRS monitor is responsible for monitoring six metrics related to resource utilization. We use the *top* command to get the real-time CPU utilization. For memory usage, we get the overall memory size and the memory used from the */proc/meminfo* and calculate memory utilization. For the network utilization, we get the data size sent and received by the network card from the */proc/net/dev* and record the time. Then we calculate the average rate between two queries as the result. For disk performance, we use the *iotop* tool to get the reading and writing rate. DRS monitor runs multiple threads to continuously monitor and record the resource utilization of the node. In order to obtain the average resource utilization within a certain period of time, we use four queues to record the usage of CPU, memory, network, and disk respectively. When a query request is received, the corresponding records are taken from the queue header to calculate the average value.

The monitoring process of DRS monitor is shown in Algorithm 3. We set a minimum value AT. Each time when returning the node state, we calculate the average value of the first AT elements in each queue. When the monitor start running, it initializes a thread for each resource. These threads obtain resource utilization every 1 s and maintain the resource queue. The monitor blocks until the number of elements in each queue are greater than AT. Then the monitor establishes a socket connection to the decision maker, binding the decision maker's IP and port. Finally, DRS monitor continuously listens for requests from the decision maker. Once receiving the request to query the node state, the monitor will calculate the average resource utilization and send the result as a response.

5 | EVALUATION

In this section, we evaluate the performance of DRS. Firstly, we introduce the experimental environment and deploy microservices. Then we analyze the resource requirements of these microservices and set the experimental parameters accordingly. To evaluate the performance of DRS, we compare it with Kuber-scheduler and some traditional scheduling methods, including Random and Round Robin. The evaluation metrics include the reward, the resource utilization of

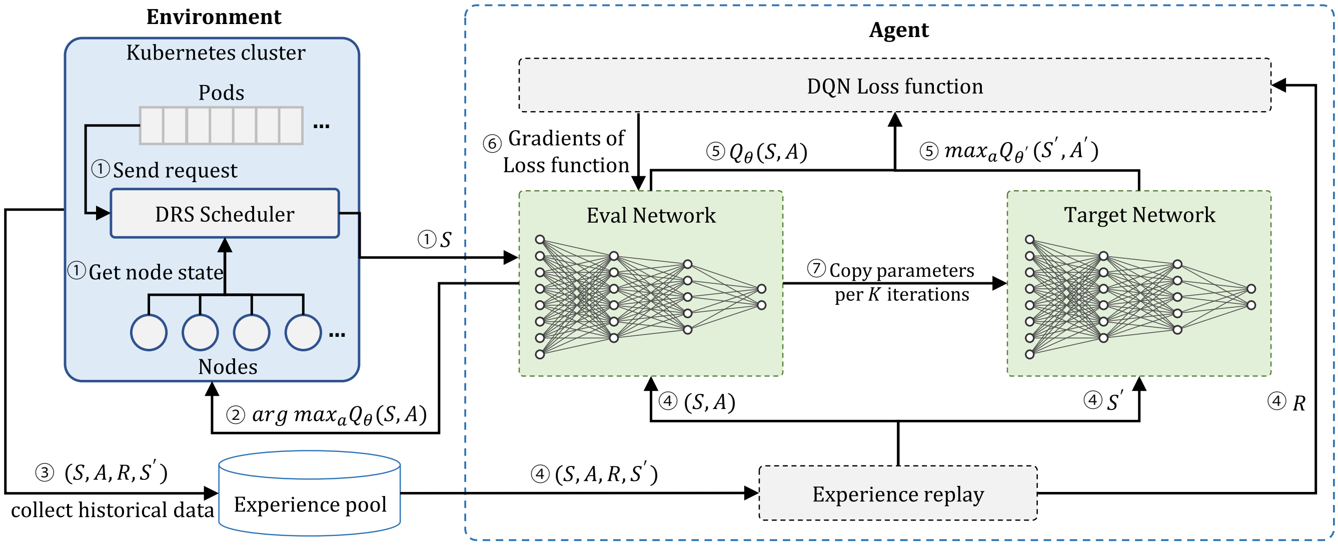


FIGURE 3 The training algorithm of DRS decision maker.

Algorithm 3. The monitoring process of DRS

```
Set the avgtime $AT$                                    ▷ The minimum number of queue elements
Initialize queues $Q_{CPU}, Q_{Mem}, Q_{Net}, Q_{Disk}$     ▷ Recard the CPU, memory, network, and disk utilization
Initialize the threads $T_{CPU}, T_{Mem}, T_{Net}, T_{Disk}$
Start the threads $T_{CPU}, T_{Mem}, T_{Net}, T_{Disk}$     ▷ Monitor the resource utilization and maintain the queue
Wait until the size of each queue is greater than $AT$     ▷ Wait until there are enough elements in each queue
Initialize the socket connection $socket$
$socket.bind(ip, port)$                                 ▷ Bind the socket connection with DRS~decision maker
$socket.listen()$
while True do
    if Receive a request for the node state then
        Get the first AT elements from each queue $CPU, Mem, Recv, Tran, Read, Write$
        $State = [avg(CPU), avg(Mem), avg(Recv), avg(Tran), avg(Read), avg(Write)]$
        $socket.send(State)$
    end if
end while
```

the cluster, the makespan of microservices, and the scheduling latency. Through these tests, we intend to answer the following questions:

1. Whether DRS performs better than other schedulers with different workloads?
2. How much additional overhead does the DRS monitor bring?
3. How much performance improvement does DRS bring by considering network and disk pressure?
4. How does DRS perform in terms of scheduling latency?

5.1 | Experimental setup

5.1.1 | Hardware and software

We use five virtual machines to build the testbed and deploy Kubernetes (version v1.23.4) on Ubuntu 22.04 OS. The implementation details of the testbed is shown in Figure 4. We label these nodes as *Node0*, *Node1*, *Node2*, *Node3*, and

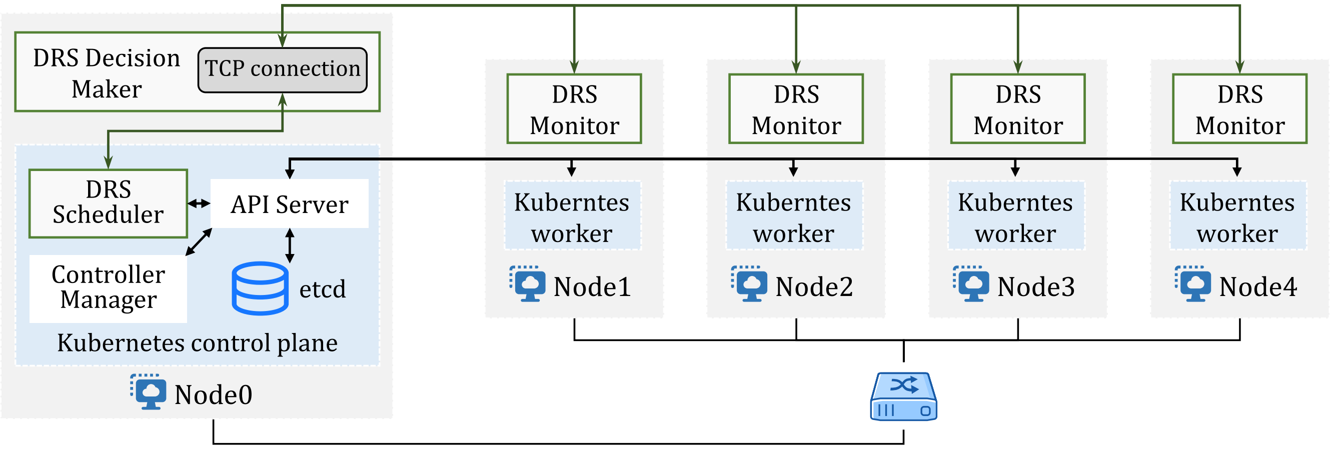


FIGURE 4 The implementation of DRS testbed.

TABLE 2 Hardware configuration used in the experiment.

Node	Node type	CPU	Memory	Network	Disk
Node0	Master	4 * AMD R7-4800H@2.90GHz	8GB	8.2MB/s	R:678MB/s, W:661MB/s
Node1	Worker	2 * AMD R7-4800H@2.90GHz	4GB	8.4MB/s	R:663MB/s, W:610MB/s
Node2	Worker	2 * Intel Core i5-9400@2.90GHz	2GB	8.6MB/s	R:280MB/s, W:269MB/s
Node3	Worker	2 * Intel Core i5-9400@2.90GHz	4GB	8.1MB/s	R:305MB/s, W:299MB/s
Node4	Worker	4 * Intel Core i5-9400@2.90GHz	4GB	8.5MB/s	R:318MB/s, W:303MB/s

Node4. These nodes connect with each other via a local area network. *Node0* is the master node responsible for running the scheduler and deploying applications. The other nodes play the role of the worker node, which performs specific applications. We modify the source code of Kubernetes, recompile it to generate DRS scheduler, and use the binary file to build the docker image. We deploy DRS scheduler as the second scheduler of the cluster running with Kube-scheduler[‡]. We can choose DRS scheduler by setting the *spec.schedulerName* field in the pod's configuration file. DRS scheduler and DRS decision maker run on the master node. DRS monitor runs on all of the worker nodes. Both the DRS scheduler and the DRS monitor connect with the DRS decision maker through the TCP socket connections. Table 2 shows the configuration of each node in detail.

5.1.2 | Application

In the experiments, we set up three microservices with different resource requirements. The details of these microservices are shown in Table 3. *Video Scale* microservice uses the FFmpeg tool to scale a video to a certain resolution. A lot of CPU resources will be used by this process. *Transmission* microservice continuously sends some data to remote servers and receives corresponding responses. This will increase the pressure on network communication. *Data write* microservice reads a certain size of data from the disk and writes a copy using the Linux dd command, which brings greater pressure on disk writing. In the following sections, we refer to these microservices as *Video*, *Network*, and *Disk*. We encapsulate these microservices and the corresponding running scripts as Docker images for deployment.

5.2 | Workload construction

To evaluate the performance of DRS, we construct three workloads using the three microservices mentioned above. As shown in Section 3, the resource requirement of the pod is a part of the environment state. Therefore, we perform several

[‡] https://kubernetes.io/docs/tasks/extend-kubernetes/configure-multiple-schedulers/.

TABLE 3 Microservices used in the experiment.

Application	Type	Desrciption	Docker image
Video scale	CPU-intensive	Scale the video to a certain size with ffmpeg	jolyonjian/apps:cpu-1.0
Transmission	Network-intensive	Transfer data of a certain size to the server	jolyonjian/apps:net-1.0
Data write	IO-intensive	Read a file on the disk and write a copy	jolyonjian/apps:io-1.0

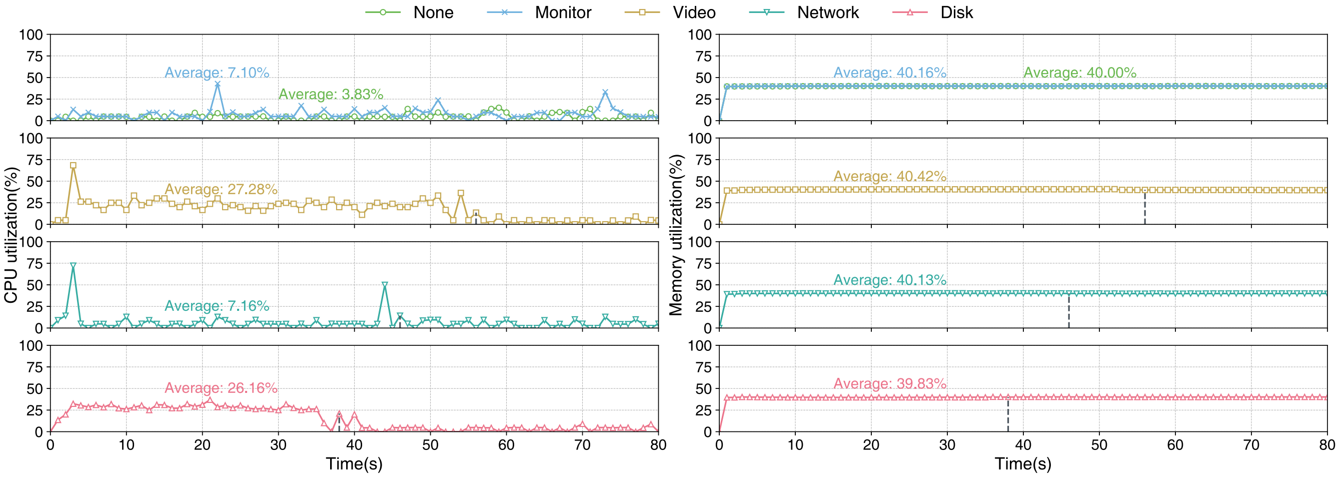


FIGURE 5 The CPU and memory utilization of the applications.

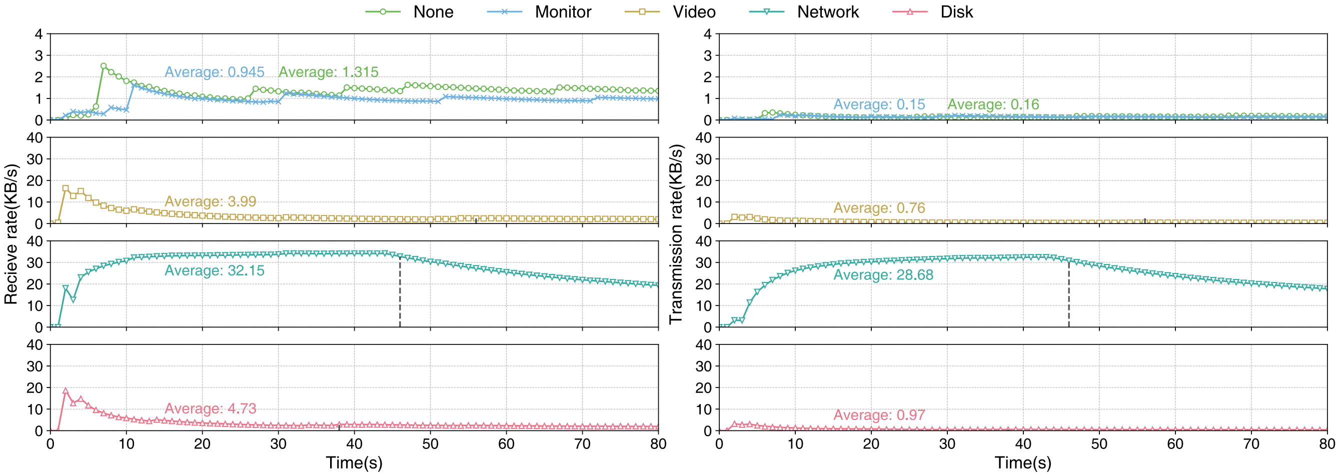


FIGURE 6 The receive and transmission rate of the applications.

tests to check for the resource utilization of each microservice. We choose *Node1* to run these microservices. We set the *limit.CPU* of the pod to 500 m, which means Kubernetes will allocate 0.5 CPU for each application. This setting ensures that the node can run multiple microservices simultaneously. Figures 5–7 show the utilization of CPU, memory, network, and disk for each of the three microservices. We also evaluate the resource utilization for running the Kubernetes components and running DRS monitor for comparison (labeled as *None* and *Monitor*). In the experiments for the *Video*, *Network*, and *Disk*, the execution time of these three applications is represented in Figures 5–7 by the black dotted line, with an average duration of 50 s. In order to ensure the completion of application execution, we monitor the resource utilization for a duration of 80 s.

As shown in Figure 5, we calculate the average resource utilization during the microservice execution. When the node only runs the Kubernetes components, the CPU utilization is 3.83%. When the node runs the Kubernetes component

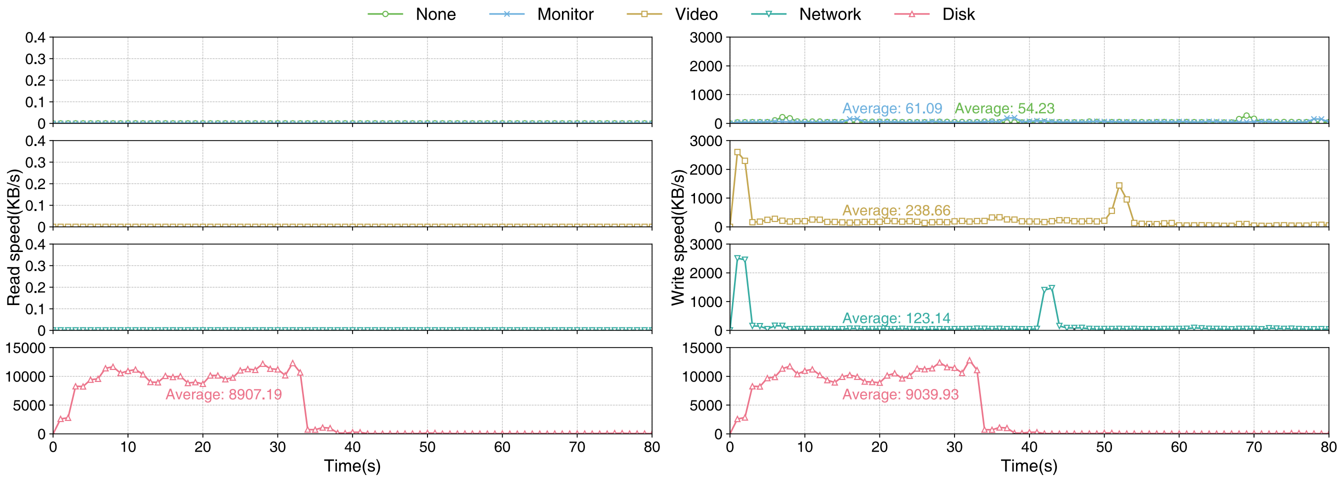


FIGURE 7 The disk read and write rate of the applications.

and DRS monitor, the CPU utilization is 7.10%. Since the monitor needs to respond to the queries for resource utilization periodically, the CPU utilization increase periodically. After deploying a microservice, the node will process the execution of the pod. Therefore, the CPU utilization will first reach a peak and then stabilize. The execution of the CPU-intensive *Video* consumes all the allocated CPU resources (500 m). The CPU utilization of *Video* is 27.28%. The *Video* application is responsible for processing a single video, after which the execution concludes. The resource utilization of the *Video* and *Disk* applications decreases at around 55 and 40 s, respectively, due to the completion of application execution. *Network* consumes fewer CPU resources and the CPU utilization is only 7.16%. It is worth noting that *Disk* also consumes all CPU resources due to the heavy operations of disk read and write. The CPU utilization of *Disk* is 26.16%. The memory utilization of different applications is almost the same, ranging from 39.83% to 40.42%.

Figure 6 shows the network utilization of the microservices, including the package receiving rate and transmission rate. When the node runs the Kubernetes components and DRS monitor, the network overhead is few. The package receiving rate of *None* and *Monitor* is 1.315 and 0.945 KB/s, respectively. The package transmission rate of *None* and *Monitor* is 0.16 and 0.15 KB/s respectively. When running *Video* and *Disk*, the node receives the pod deployment message sent by the master node. Then, the two applications run normally without data transmission. The package receiving rate of *Video* and *Disk* is 3.99 and 4.73 KB/s respectively and the package transmission rate is 0.76 and 0.97 KB/s respectively. *Network* continuously sends messages to the server and receives responses. The device network card continues to process data packets even after the completion of application execution, resulting in a gradual decline in the package sending and receiving rate. The package receiving rate is 32.15 KB/s, and the package transmission rate is 28.68 KB/s.

Figure 7 shows the network utilization of the microservices, including disk reading rate and writing rate. When running *None*, *Monitor*, *Video*, or *Network*, there is no disk read operations and their disk reading rates are almost zero. The writing rate of *None* is 54.23 KB/s, and that of *Monitor* is 61.09 KB/s. At the beginning and end of the microservice execution, disk writing operations increase the writing rate. The disk writing rate of *Video* and *Network* is 238.66 and 123.14 KB/s respectively. For *Disk*, its disk reading rate and writing rate is 8907.19 and 9039.93 KB/s, respectively.

Based on the aforementioned findings, it is clear that the cost of the DRS monitor, which amounts to roughly 3.27% CPU overhead, is quite low. We set the maximum value for each resource and convert the resource utilization of a single microservice to about 25%. In addition, we also test other *limit.CPU* settings. The results show that resource utilization is positively related to CPU limitation. Therefore, we randomly specify 200–500 m CPU resources for the pod when constructing the workload. We construct three different workloads to evaluate the performance of DRS. The workloads are as follows:

1. Even workload. The three types of applications are deployed at a ratio of 1:1:1, with a fixed time interval. In our tests, we choose to deploy an application every 20 s.
2. Random workload. We deploy three types of applications according to the ratio of 1:1:1. The time interval T between applications follows a normal distribution: $T \sim N(\mu, \sigma^2)$, $\mu = 20, \sigma = 1$.

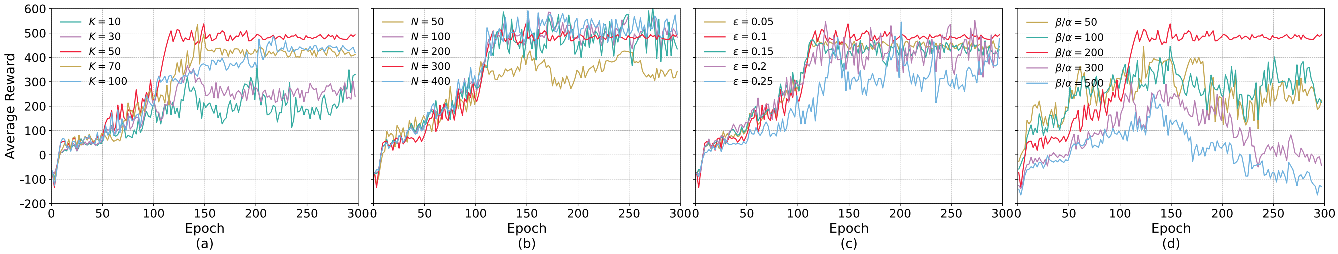


FIGURE 8 The average reward under different parameter settings.

3. CPU workload. We deploy three types of applications according to the ratio of 4:1:1. Similarly, the time interval between applications is set to 20 s.

5.3 | Experimental results

We first explore the optimal setting of key parameters in DRS to obtain the expected scheduling effect. Then, we select five scheduling algorithms for comparison, including Kube-scheduler, Random, Round Robin, Double DQN, and Dueling DQN. Among these, Random and Round Robin are heuristic algorithms. We follow the DRS scheduling process and use the corresponding scheduling policy instead of the neural network in the DRS decision maker to take action. Double DQN and Dueling DQN are enhanced versions of the DQN algorithm. We fine-tune the DRS decision maker to employ these two algorithms. Through this implementation method, we can evaluate the reward and resource utilization of these algorithms.

5.3.1 | Hyperparameter tuning

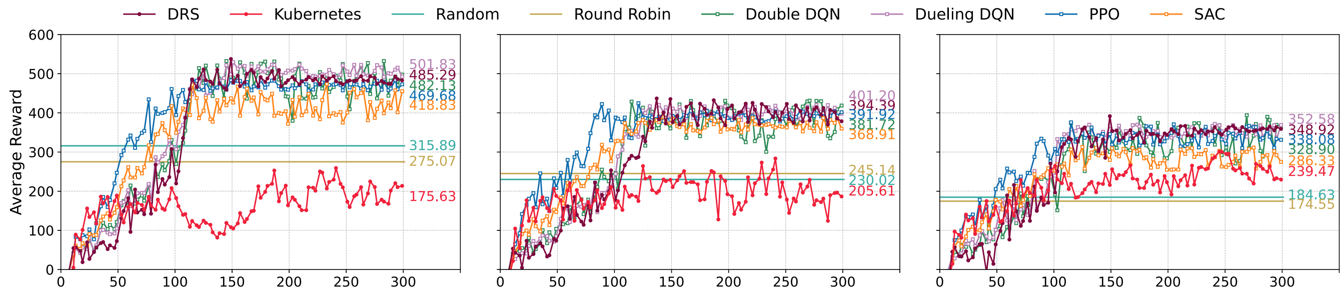
We investigate the settings of various parameters in DRS. These parameters include the update frequency K of the Q network in the DQN algorithm, the capacity N of the experience pool, the ϵ in the ϵ-greedy algorithm, and α and β in the reward function. After conducting thorough experimentation, we determine the following optimal parameter values: $K = 50$, $N = 300$, $\epsilon = 0.1$, and $\beta/\alpha = 200$. We adjust each of the five parameters individually and analyze their impact on the scheduling effect based on the results of the reward. The results are shown in Figure 8.

The DQN algorithm employs two sets of Q networks, namely the training network and the target network. The value K represents the update frequency of the parameters in the target network. As the training process involves continuous changes to the network parameters, updating the target network at every step can adversely affect the stability of the algorithm. As shown in Figure 8A, with the increase of K, the average reward initially increases and then begins to decline. For small values of K, the stability of the network decreases, resulting in significant reward fluctuations. Consequently, we select the configuration that maximizes the average return, with K set to 50.

Figure 8B shows the impact of the capacity N of the experience pool on the average reward. The observations indicate that the model converges after approximately 100 epochs. When N is less than 100, the experience pool's capacity is limited, necessitating the deletion of some memories before the model can converge. Therefore, the average returns are lower in such cases. when N exceeds 100, the model converges before the experience pool reaches the maximum capacity. Consequently, there is no need to delete any memory and the average reward obtained under different N settings remains relatively similar. Taking into account subsequent experiments involving other algorithms, we set N to 300 to ensure the stability of the algorithm.

ϵ is set to improve the exploration capability of the model and ensure that the replay process can gather sufficient training data. In the case of DRS, the action space is small. The model does not need an overly strong exploration ability and the ϵ can be set to a small value. The results obtained under different ϵ settings are shown in Figure 8C. It can be observed that with the increase of ϵ, the volatility of the average reward increases. Therefore, we set ϵ to 0.1 which corresponds to the maximum average reward.

α and β are the parameters of the reward function and play a crucial role. They can directly impact the scheduling effect. It's worth noting that the specific numerical value of the reward is not as important as its trend. Therefore, we can



(a) Average rewards under even workload.

(b) Average rewards under random workload.

(c) Average rewards under CPU workload.

FIGURE 9 The reward with even workload, random workload, and CPU workload.

focus on the parameter β/α. According to the scheduling target of DRS, a positive reward should be obtained when the average resource utilization increases and the load imbalance decreases. Based on our testing, the resource utilization $AvgUtil_t$ is approximately 40%, while the load imbalance $ImBalance_t$ is around 0.05. To strike a balance between the gaps of $AvgUtil_t$ and $ImBalance_t$, we need to set β/α to a relatively large value. Figure 8D shows the results when the β/α is set from 50 to 500. When β/α is set to 50 or 100, the model can converge but the average reward is low. When β/α is set to 300 or 500, the model excessively emphasizes the influence of $ImBalance_t$, resulting in failure to converge. Consequently, we ultimately set β/α to 200.

5.3.2 | Reward

We first evaluate the *reward* of the scheduling algorithm under three workloads. For reinforcement learning scheduling, *reward* is the direct reflection of the scheduling effect. We deploy 300 pods as 300 epochs of DRS training process and update the parameters of the target network every 50 epochs. As shown in Figure 9, with the three workloads, DRS can converge after 100 epochs and reach a stable reward. It means that DRS can effectively adapt to different workloads and learn scheduling policy quickly. At the same time, compared with the heuristic algorithms, DRS has the largest reward. This result proves that DRS can improve the resource utilization of the cluster and reduce the degree of load imbalance between nodes. With random workload, the time interval between applications follows a normal distribution, which may lead to small or large resource consumption at a certain time. Therefore, resource utilization fluctuates greatly, and the reward of the scheduling algorithm decreases. With CPU workload, the CPU requirement of the pod is large, and the requirements of other different resources are uneven. Therefore, the overall resource utilization decreases significantly, and so does the reward.

The scheduling effect of Kube-scheduler is poor under conditions of even workload. And with the CPU workload, Kube-scheduler has good scheduling effect, which is close to DRS. This is because Kube-scheduler only considers a single node's usage of CPU and memory, which causes uneven resource usage. This scheduling policy performs well when the application needs more CPU resources. However, when the microservice needs more network or disk resources, the scheduling quality will decline. The scheduling effect of Random and Round Robin is similar under three workloads. When the microservice resource requirements are balanced, the scheduling effect is good. However, the overall resource utilization fluctuates greatly due to the randomness of the scheduling result. When there is an imbalance in the resource requirements for the microservices, the resource imbalance will worsen and the scheduling effect will be poor.

We also implemented four representative RL algorithms including the DQN variants, PPO algorithm, and SAC algorithm to compare with DRS. The Double DQN algorithm aims to address the issue of Q value overestimation present in the DQN algorithm. The average reward achieved by the Double DQN algorithm is slightly lower than the DQN algorithm, and it exhibits a higher level of fluctuation. The Dueling DQN algorithm calculates the state-value function and the advantage function of the action separately. This enables the agent to learn the differences resulting from different actions. Compared with the DQN algorithm, Dueling DQN demonstrates a more frequent and accurate learning of the state value function, resulting in a slightly higher average reward. Overall, the results obtained from Double DQN and Dueling DQN are relatively similar to those of the DQN algorithm. Indeed, these two algorithms can be readily implemented within DRS to adapt to various scenarios. In the scenario with discrete action space, the DQN algorithm employed

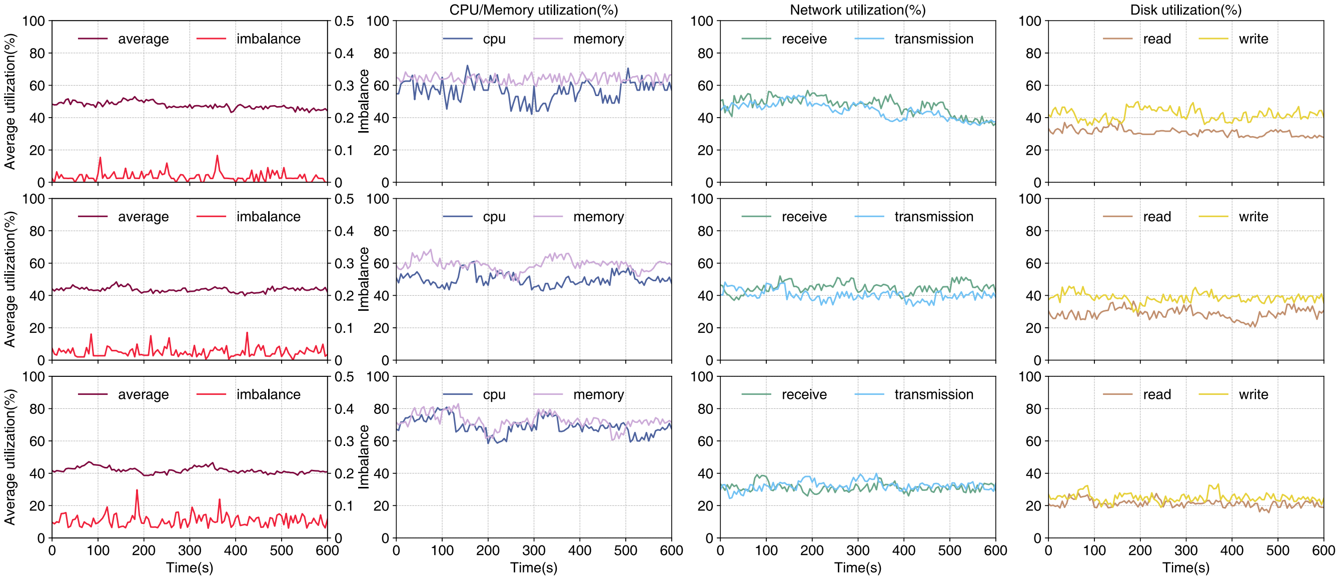


FIGURE 10 The resource utilization of DRS scheduler with even workload, random workload, and CPU workload.

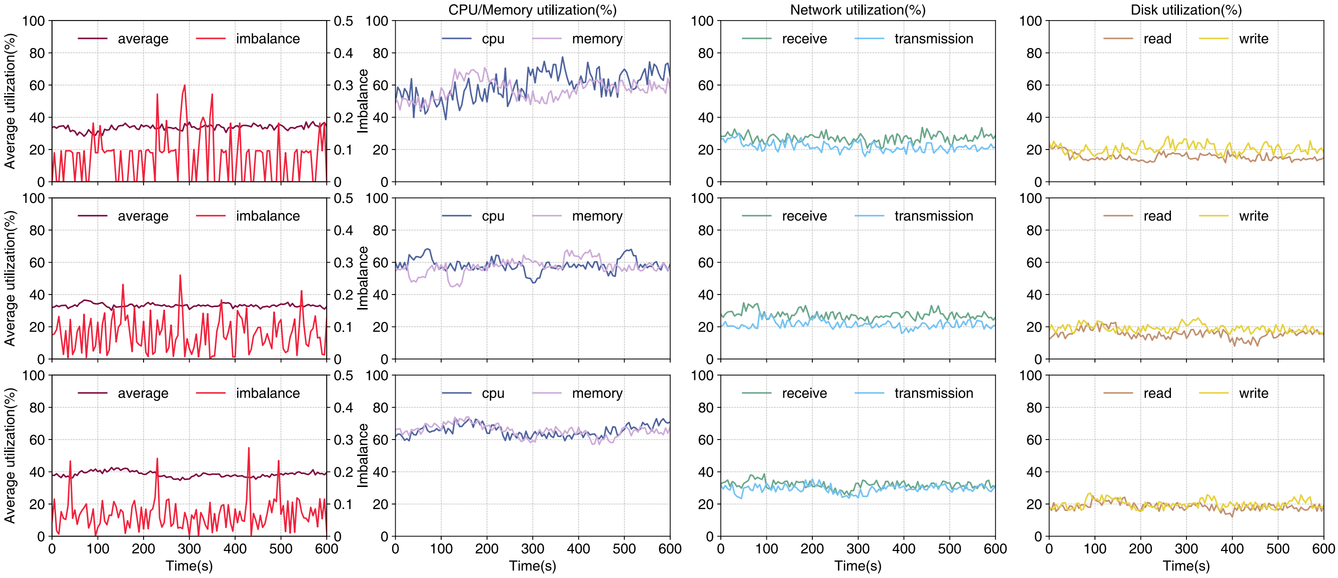


FIGURE 11 The resource utilization of Kube-scheduler with even workload, random workload, and CPU workload.

by DRS proves to be more effective compared with the PPO algorithm and the SAC algorithm. The PPO algorithm exhibits a marginally swifter rate of convergence, boasting stability, and its reward is modestly lower than those of DRS. The SAC algorithm can facilitate random exploration within the learning process. But when confronted with scenarios characterized by a small and discrete action space, the reward of the SAC algorithm is also lower than DRS.

5.3.3 | Resource utilization

According to Figure 9, we can see that after the 250th epoch, the effects of the scheduling algorithms have been stable. So we monitor the utilization of each resource from the 250th epoch. The results are shown in Figures 10–13. Average utilization and resource imbalance between nodes are the basis for calculating *reward*. They are calculated using the equation presented in Section 3.

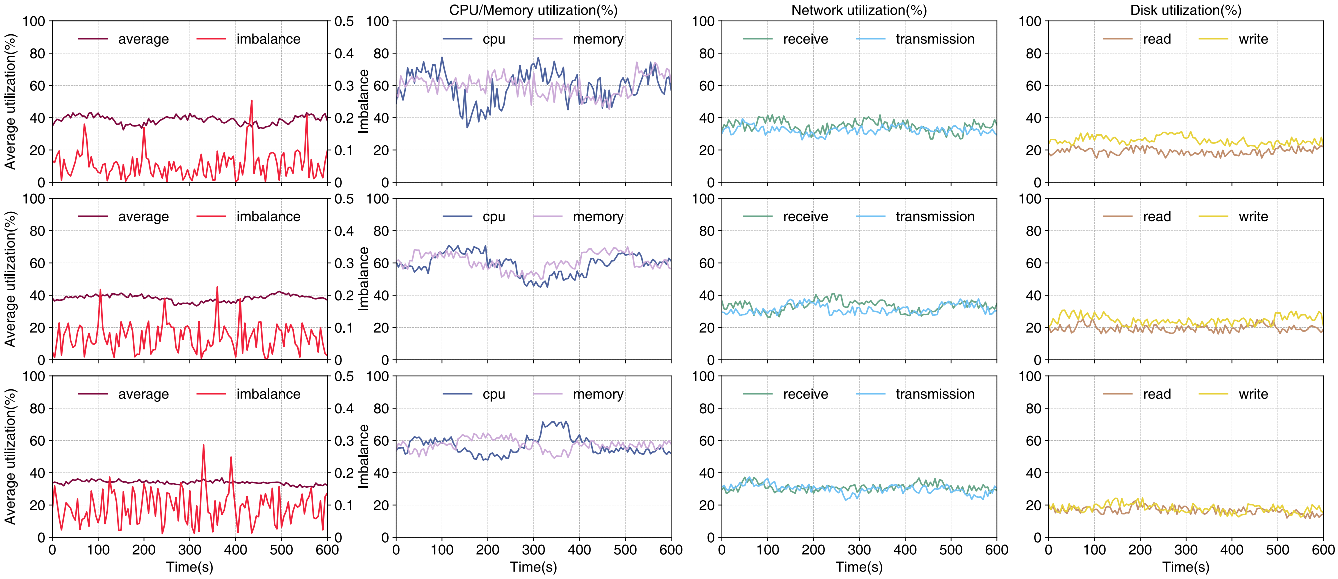


FIGURE 12 The resource utilization of Random scheduler with even workload, random workload, and CPU workload.

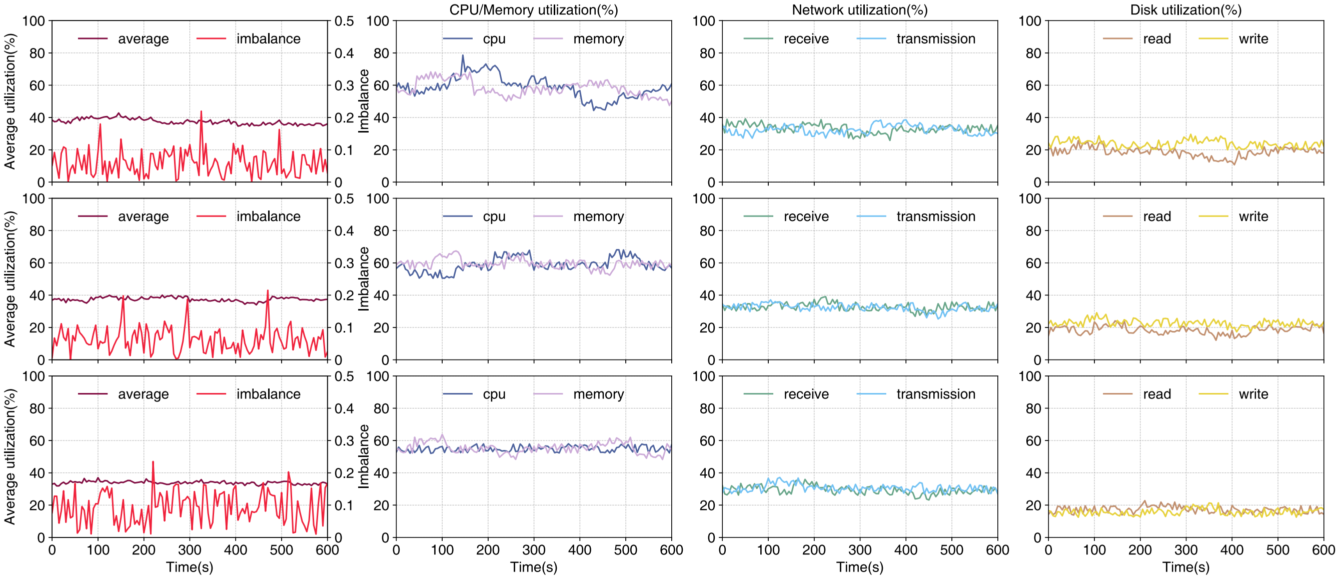


FIGURE 13 The resource utilization of round robin scheduler with even workload, random workload, and CPU workload.

Figure 10 shows the resource utilization of DRS with different workloads. With the even workload, the CPU and memory utilization is 56.80% and 64.05% respectively. The package receiving and transmission rates of the network are 47.51% and 44.42%, respectively. And the disk reading and writing rates are 31.00% and 41.82%, respectively. With the random workload, the CPU and memory utilization is 50.02% and 59.05% respectively. The package receiving and transmission rates of the network are 44.85% and 40.26%, respectively. And the disk reading and writing rates are 28.86% and 38.62%, respectively. With the CPU workload, the CPU and memory utilization is 69.17% and 72.43% respectively. The package receiving and transmission rates of the network are 31.15% and 32.36%, respectively. And the disk reading and writing rates are 21.36% and 24.96%, respectively. With these three workloads, the average resource utilization is 47.60%, 43.61%, and 41.90%, respectively. The imbalance is 0.02, 0.03, and 0.05, respectively. We can see that with the even workload, the resource utilization of DRS is high and balanced. When the CPU load is heavy, the utilization of CPU and memory will increase and fluctuate. And the utilization of the network and disk will decrease significantly. Nevertheless, DRS maintains a high average resource utilization and a low load imbalance with different workloads, and achieves the expected scheduling target well.

Figure 11 shows the resource utilization of Kube-scheduler with different workloads. With the even workload, the CPU and memory utilization is 59.61% and 57.13% respectively. The package receiving and transmission rates of the network are 27.14% and 21.94%, respectively. And the disk reading and writing rates are 15.45% and 20.34%, respectively. With the random workload, the CPU and memory utilization is 58.30% and 57.33% respectively. The package receiving and transmission rates of the network are 27.38% and 21.50%, respectively. And the disk reading and writing rates are 15.64% and 19.12%, respectively. With the CPU workload, the CPU and memory utilization is 65.44% and 65.58% respectively. The package receiving and transmission rates of the network are 31.94% and 29.43%, respectively. And the disk reading and writing rates are 18.48% and 20.00%, respectively. With the three workloads, the average resource utilization is 33.60%, 33.21%, and 38.48%, respectively. The imbalance is 0.08, 0.08, and 0.07, respectively. The Kube-scheduler only considers the usage of CPU and memory, and the utilization of network and disk is low. When the CPU load is heavy, the CPU utilization is more balanced and the scheduling effect is better. The load imbalance of the Kube-scheduler is 4.39×, 3.06×, and 1.25× than that of DRS respectively with the three workloads. At the same time, DRS's average resource consumption outperforms Kube-scheduler by 41.67%, 31.31%, and 8.90%, respectively. DRS performs well for all three workloads, while Kube-scheduler only performs well for the CPU workload.

Figure 12 shows the resource utilization of Random with different workloads. With the even workload, the CPU and memory utilization is 59.45% and 59.85% respectively. The package receiving and transmission rates of the network are 34.78% and 32.32%, respectively. And the disk reading and writing rates are 19.19% and 25.91%, respectively. With the random workload, the CPU and memory utilization is 59.24% and 60.85% respectively. The package receiving and transmission rates of the network are 33.37% and 31.51%, respectively. And the disk reading and writing rates are 19.60% and 24.78%, respectively. With the CPU workload, the CPU and memory utilization is 56.83% and 57.31% respectively. The package receiving and transmission rates of the network are 30.99% and 29.95%, respectively. And the disk reading and writing rates are 16.69% and 17.86%, respectively. With the three workloads, the average resource utilization is 38.58%, 38.22%, and 33.94%, respectively. The imbalance is 0.06, 0.07, and 0.09, respectively. Due to the randomness of scheduling, resource utilization of Random fluctuates. Therefore, the load imbalance of Random is relatively high, which is 3.21×, 2.77×, and 1.73× than that of DRS for the three workloads. For all three workloads, Random performs similarly but is less efficient than DRS. The average resource utilization of DRS is improved by 23.37%, 14.09%, and 23.47% over Random.

Figure 13 shows the resource utilization of Round Robin with different workloads. With the even workload, the CPU and memory utilization is 58.74% and 58.19% respectively. The receiving and transmission rates of the network are 33.18% and 32.42%, respectively. And the disk read and write rates are 18.67% and 23.82%, respectively. With the random workload, the CPU and memory utilization is 58.85% and 59.61% respectively. The receiving and transmission rates of the network are 32.78% and 32.37%, respectively. And the disk read and write rates are 18.40% and 22.74%, respectively. With the CPU workload, the CPU and memory utilization is 54.88% and 55.36% respectively. The receiving and transmission rates of the network are 29.36% and 30.69%, respectively. And the disk read and write rates are 17.25% and 15.78%, respectively. With the three workloads, the average resource utilization is 37.50%, 37.46%, and 33.88%, respectively. The imbalance is 0.06, 0.06, and 0.09, respectively. The Round Robin scheduling performance is similar to that of Random. The load imbalance of Round Robin is relatively high but stable, which is 3.40×, 2.50×, and 1.72× than that of DRS with the three workloads. Compared with Round Robin, DRS improves the average resource utilization by 26.92%, 16.43%, and 23.66% respectively.

In general, DRS has the highest resource utilization, especially network and disk utilization. Kube-scheduler has the lowest network and disk utilization, which aggravate the resource imbalance between nodes. The scheduling results of Random and Round Robin are somewhat random, and their resource utilization is moderate. DRS has a considerably reduced resource imbalance than other algorithms in this area. With a balanced load, Kube-scheduler has the highest resource imbalance. Enhancing the awareness of network and disk usage helps DRS improve its understanding of the cluster state and further adapt to different load conditions. In summary, DRS achieves the expected scheduling target well. Compared with Kube-scheduler, DRS brings an improvement of 27.29% in resource utilization and reduces the load imbalance by 2.90×.

5.3.4 | Latency

We evaluate the scheduling latency of DRS and Kube-scheduler with even workload, random workload, and CPU workload respectively. We also deploy 300 pods and the results are shown in Table 4. Makespan represents the total time from deploying the first pod to the end of the execution of all pods. For Kube-scheduler, the makespan contains two parts:

TABLE 4 Latency breakdown of DRS and Kube-scheduler with different workloads.

Workload	Scheduler	Type	Latency (×10^3)	Percentage (%)
Even	DRS	Makespan (s)	14.950	100
		Make decision (ms)	1.572	0.011
		Communication (ms)	10.171	0.647
	Kube-scheduler	Makespan (s)	14.038	100
		Scheduling (ms)	1.621	0.012
Random		Makespan (s)	15.703	100
	DRS	Make decision (ms)	1.583	0.010
		Communication (ms)	10.378	0.656
	Kube-scheduler	Makespan (s)	15.304	100
		Scheduling (ms)	1.579	0.010
CPU		Makespan (s)	18.843	100
	DRS	Make decision (ms)	1.593	0.009
		Communication (ms)	10.220	0.642
	Kube-scheduler	Makespan (s)	19.037	100
		Scheduling (ms)	1.566	0.008

the pod execution latency and the scheduling latency. For DRS, the makespan contains three parts: the pod execution latency, the decision latency, and the communication latency. According to the results of our tests, the scheduling latency of Kube-scheduler is about 4.3 ms. The scheduling latency of the DRS is about 38.2 ms, of which the communication latency is 35 ms.

As shown in Table 4, with the even load, the makespan of DRS is about 1.06× that of Kube-scheduler. DRS does not aim to reduce the makespan. To improve resource utilization, DRS may schedule the pod to the node with poor performance, thus increasing the makespan. The scheduling latency of DRS is 7.24× that of Kube-scheduler, but their decision latency is similar. The difference in latency comes from the socket communication overhead between DRS components. Since the communication latency is far less than the pod execution latency, the communication latency only accounts for 0.647% of the makespan. With random workload, the makespan of DRS is 1.03× that of Kube-scheduler. The decision latency and communication latency of DRS account for 0.010% and 0.656% of the makespan respectively. The scheduling latency of Kube-scheduler accounts for 0.010% of the makespan. With the CPU workload, the scheduling latency of Kubernetes accounts for 0.008% of makespan. The decision latency and communication latency of DRS account for 0.009% and 0.642% of makespan respectively. Kubernetes may wait for node resources when scheduling pods, making its makespan greater than DRS. It is worth noting that the decision latency of DRS is affected by the calculation scale of the neural network. Under a fixed reinforcement learning model, the scheduling latency of DRS is stable and not affected by the complexity of the cluster state. Although the scheduling latency of DRS is greater than that of Kubernetes, it is negligible for makespan. When the pod execution latency increases, the proportion of scheduling latency will further decrease. DRS improves the resource utilization of the cluster with a negligible latency overhead. In summary, the performance advantages of DRS can be comprehended from two aspects. Primarily, DRS demonstrates comparable execution latency compared to the Kube-scheduler. Furthermore, through the comparison between communication latency and the makespan, the results highlight that DRS introduces only negligible additional overhead.

5.3.5 | Comparsion with other RL scheduling algorithms

To facilitate a comprehensive assessment of the performance of DRS, we conducted a series of experiments. Specifically, we use the reward settings of three previous works (RLSK,[37] DeepRM,[22] and Multi DeepRM[38]) respectively under the aforementioned experimental environment and workloads. We conduct a comparative analysis across three dimensions,

TABLE 5 Comparison between DRS and other RL scheduling algorithms.

Workload	Algorithm	CPU (%)	Memory (%)	Network (%)	Disk (%)	Resource (%)	Imbalance	Makespan (s)
Even	DRS	56.80	64.05	45.97	36.41	**50.81**	**0.02**	14.95
	RLSK	59.45	59.85	33.55	22.55	43.85	0.06	14.98
	DeepRM	59.61	57.13	24.54	17.89	39.79	0.09	**13.74**
	Multi DeepRM	58.99	57.27	23.68	19.99	39.98	0.09	14.23
Random	DRS	50.05	59.05	42.55	33.74	**46.35**	**0.03**	15.70
	RLSK	59.24	60.85	32.44	22.19	43.68	0.07	16.13
	DeepRM	58.30	57.33	24.44	17.38	39.36	0.11	13.75
	Multi DeepRM	56.49	60.54	26.73	19.62	40.84	0.10	**13.43**
CPU	DRS	69.17	72.43	31.75	23.16	**49.13**	**0.05**	18.84
	RLSK	56.83	57.31	30.47	17.28	40.47	0.09	19.07
	DeepRM	65.44	65.58	30.68	19.24	45.24	0.18	**12.97**
	Multi DeepRM	62.82	58.03	32.77	21.12	43.69	0.16	13.83

including resource utilization, load imbalance, and makespan, to compare their performance. The results are shown in Table 5.

The primary objective of the RLSK framework is to achieve a balanced resource allocation in the multi-cluster environment. RLSK only focuses on two resource types: CPU and memory. RLSK exhibits a modest degree of inefficiency in harnessing network and disk resources, leading to disparities in their utilization across various resources. This disparity results in a higher resource imbalance. Furthermore, in our experimental environment with a single cluster, RLSK demonstrates lower resource utilization and a marginally extended makespan in comparison to DRS. DeepRM and Multi DeepRM, tailored for single-cluster and multi-cluster scenarios, respectively, are designed to expedite task completion through advanced scheduling policies. In our single-cluster environment, they exhibit comparable performance. Notably, their reward settings prioritize task completion time and favor the allocation of tasks to the highest-performing nodes, leading to improved makespan in comparison with DRS and RLSK. However, this scheduling policy emphasis on task completion time induces a notable skew in resource utilization, resulting in a significantly elevated resource imbalance. Moreover, the average resource utilization also falls short of DRS. In summary, under the specific experimental conditions and workloads, the performance of DRS aligns most closely with our pre-established target. Additionally, the results also prove the influence of reward setting on the scheduling effect and performance of the RL algorithm.

6 | RELATED WORK

Recently, academia and industry have done a lot of research on the resource scheduling problem of Kubernetes. In these works, researchers design customized scheduling algorithms or schedulers to overcome the limitations of the Kubernetes scheduler. Their optimization direction includes improving the overall resource utilization, balancing the load, reducing the completion time of tasks, reducing the decision time, and so on. This research can be roughly divided into two types. One is to improve the scheduling algorithm of Kubernetes using traditional methods, and the other type is to use machine learning methods (especially reinforcement learning). In this section, we introduce the traditional Kubernetes scheduling optimization and reinforcement learning scheduling and compare them with DRS. Table 6 compares the characteristics of DRS and representative related work.

6.1 | Kubernetes scheduling optimization

Some research proposed customized scheduling algorithms for different scheduling scenarios. Stratus[39] is a cluster scheduler specially designed to orchestrate batch jobs on virtual clusters. It packages tasks based on the required resources and

TABLE 6 Detailed characteristics of representative related work.

		Resource Metrics						
Ref.	**Algorithm**	**CPU**	**Mem.**	**Net.**	**IO**	**GPU**	**Env.**	**Characteristic**
KCSS[41]	TOPSIS	✓	✓	✗	✓	✗	Simulation (1C, 4N)	Multi-criteria node selection considering resource, power, task, and timestamp
KubeCG[43]	heuristic	✓	✓	✗	✗	✓	Testbed (1C, 6N)	Resource scheduling in heterogeneous cluster using timeline and historical data
DeepRM[22]	REINFORCE	✓	✓	✗	✗	✗	Testbed (1C, 1N)	Modeling system states as images, using deep reinforcement learning
Multi DeepRM[38]	REINFORCE	✓	✓	✗	✗	✗	Testbed (1C, 2N)	Deep reinforcement learning scheduling for multi-resource and multi-machine jobs
RLSK[37]	DQN	✓	✓	✗	✗	✗	Simulation (3C)	Load balancing scheduling in multi-cluster using deep reinforcement learning
HDRL[45]	DQN	✓	✓	✗	✓	✗	Simulation (1C, 40N)	Hierarchical framework for resource allocation and power management
DRS	DQN	✓	✓	✓	✓	✗	Testbed (1C, 5N)	Reinforcement learning scheduling for load balancing with multi-resource metrics

estimated processing time to minimize the cluster cost of using the public cloud. Weiguo Zhang et al. implemented the model extraction of Kubernetes scheduling module.[40] They combine the ant colony algorithm and particle swarm optimization algorithm to improve the K8s scheduling model. Some work enhances the Kubernetes scheduler by sensing the static and dynamic information of the hardware. KCSS[41] introduces multiple selection criteria for Kubernetes scheduling, including CPU, memory, disk utilization, power consumption, pod number, and so forth. It provides the scheduler with an enhanced global view of cluster resources. In order to choose the best node, KCSS employs the TOPSIS algorithm, which ultimately results in a positive outcome. NetMARKS[42] uses Istio Service Mesh to collect the dynamic network metrics of the Kubernetes cluster so that the scheduler can understand the dependencies between pods. NetMARKS can schedule delay-sensitive tasks in edge clusters and has achieved good results under different workloads. KubeCG[43] improves the awareness of Kubernetes about GPU resources. It optimizes container deployment by considering the timeline of Pod and the historical execution information, so as to decrease the execution latency caused by waiting for GPU resources.

6.2 | Reinforcement-learning-based scheduling

With the development of artificial intelligence, machine learning methods such as reinforcement learning have shown good effects in solving sequence decision-making problems. DQ-HEFT[44] combines deep Q-learning with the heuristic HEFT method to solve the workflow scheduling problem in cloud computing. This algorithm can achieve better makespan and speed. DQ-HEFT, however, also has its limits. When there are many tasks, the Q table's value will be updated excessively, slowing down scheduling. Decima[23] models an event-driven simulation environment based on the Spark cluster. It uses an extensible graph neural network to extract features of tasks and can handle tasks with a dependency of any size. DeepRM[22] models the state information of the resource scheduling system into an image, as an input for the convolutional neural network. The convolutional neural network extracts the image information quickly. Through the reinforcement learning method, they iteratively update the parameters in the neural network to generate the final scheduling policy. On this basis, DeepRM is also extended to multiple clusters and multiple machines.[38] RLSK[37] is also a multi-cluster job scheduler, which aims to improve the average resource utilization between clusters and balance the utilization of different resources in the cluster. It achieves resource balance inside and outside the cluster using DQN algorithm.

Ning Liu et al. used a two-layer deep reinforcement learning model to jointly complete the management of cluster resources and energy consumption.[45] One of the deep reinforcement learning models is responsible for analyzing the cluster state and reasonably scheduling tasks. The other one models the energy consumption of the machines to analyze their state. This model will shut down or start the machines for reasonable energy consumption management.

Through the analysis of the above work, we can see that hardware awareness of the worker nodes is a key factor in Kubernetes scheduling.[9] Adding resource indicators of node resources can help the scheduler understand the cluster state, thereby improving the scheduling performance. However, traditional scheduling algorithms still need to spend a lot of time adjusting the parameters of the algorithm after designing indicators. This makes the design of the scheduling algorithm very complex and tedious. Using reinforcement learning we can solve this problem and make the scheduler learn the scheduling policy automatically. In this paper, we aim to reduce the imbalance of resource usage caused by the Kubernetes scheduler through the advantages of reinforcement learning. Inspired by the previous work, we expand the resource indicators of nodes and increase the consideration of networks and disks. When selecting indicators, we choose indicators that are easy to obtain, such as package transmission rate, to avoid large monitoring costs. Compared with other work using reinforcement learning, we re-model the application scenario of DRS and achieve a good scheduling effect through the reasonable design of state, action, and reward.

For performance evaluation, as mentioned in literature,[9] the research on the Kubernetes scheduling algorithm lacks standardized evaluation methods. Existing RL scheduling algorithms are specifically designed to address distinct scheduling problems. The differences between scheduling targets and workloads make the research isolated and difficult to compare fairly. Applying these algorithms directly to our scenario may adversely impact the effectiveness of the scheduling process, potentially leading to diminished scheduling performance. Moreover, the absence of source code and parameter settings impedes the implementation and evaluation of these algorithms. The new proposed algorithm is usually compared with the simple heuristic algorithms or the default algorithm of Kubernetes, as we do in this paper.

7 | CONCLUSIONS

Kubernetes, which is a container orchestration framework, is actually the de facto standard of cloud-native. Nevertheless, Kubernetes' native scheduler suffers from a serious drawback of uneven resource utilization. In this paper, we propose DRS, which is a Kubernetes scheduler based on deep reinforcement learning. DRS consists of three parts, which include scheduler, monitor, and decision maker. DRS scheduler follows the Kubernetes Scheduling Framework and implements user-defined filtering and scoring functions. DRS monitor keeps track of how much each node's CPU, memory, network, and disk resources are being used. Compared with Kube-scheduler, DRS is more comprehensive in the awareness of resource utilization. DRS decision maker is trained using the DQN algorithm and can automatically learn scheduling policy from the state of the cluster. By designing the reward function of DRS, DRS achieves the high-level scheduling target of improving cluster resource utilization and reducing load imbalance. Experimental results show that DRS can adapt to different workloads, and achieve better scheduling effects than other common scheduling algorithms. For average resource utilization, DRS achieves an improvement of 27.29% compared with Kube-scheduler. For load imbalance, DRS is 2.90× less than Kube-scheduler. At the same time, DRS only brings 3.27% CPU overhead and 0.648% communication latency.

DRS possesses commendable adaptability and scalability. In terms of its design and implementation methods, DRS has the capacity to accommodate various clusters and scheduling targets, through the redesign of the reinforcement learning model. The design of the RL scheduling algorithm is closely related to the environment, particularly in intricate scenarios. While extending DRS to different scenarios, the model parameters should be adjusted in conjunction with the specific environment. This adjustment can be guided by the design principles and adjustment strategies of DRS, which offer valuable insights and serve as a reference for achieving optimal performance in different scenarios. In terms of training methods, DRS is acquired through online training, where the *State* transitions are derived from interactions with the real environment. In the offline training scenario, DRS can be trained with the historical data on system operations. We can create a simulated environment and utilize the historical data for interactive network training. In the absence of historical data, DRS will encounter the challenge of the cold start. The general solution is to collaborate with traditional scheduling algorithms and gather operational data (*State* and *Action*) based on the design of DRS until the prerequisites for model training are satisfied. In the future, we will continue to optimize the scheduling algorithm from the perspective of resource awareness and scalability. For example, we can improve the awareness of Kubernetes about new hardware (such as GPU and FPGA) to further improve the performance. Or we can introduce distributed scheduling and multi-level scheduling to prevent single-point failure.

AUTHOR CONTRIBUTIONS

All authors contributed to this paper equally.

ACKNOWLEDGMENTS

This work is partially supported by the National Key Research and Development Program of China (2018YFB2100304), National Natural Science Foundation (62272248), the Open Project Fund of State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences (CARCH201905, CARCHA202108) and Sponsored by Zhejiang Lab (2021KF0AB04), Tianjin Graduate Scientific Research Innovation Project (2022BKY028).

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are openly available in github at https://github.com/JolyonJian/DRS.

ORCID

Zhaolong Jian https://orcid.org/0000-0003-1543-3207

REFERENCES

1. Gu R, Zhang K, Xu Z, et al. Fluid: dataset abstraction and elastic acceleration for cloud-native deep learning training jobs. Paper presented at: 2022 IEEE 38th International Conference on Data Engineering (ICDE). IEEE. 2022:2182-2195.
2. Boudi A, Bagaa M, Pöyhönen P, Taleb T, Flinck H. AI-based resource management in beyond 5G cloud native environment. *IEEE Netw*. 2021;35(2):128-135.
3. Zhao H, Deng S, Liu Z, Yin J, Dustdar S. Distributed redundancy placement for microservice-based applications at the edge. *IEEE Trans Serv Comput*. 2020;15(3):1732-1745.
4. Monteiro A, Almeida W, Hazin R, Lima dA, eSKG S, Ferraz F. A survey on microservice security-trends in architecture, privacy and standardization on cloud computing environments. *Int J Adv Secur*. 2018;11(3 & 4):201-213.
5. Fu K, Zhang W, Chen Q, Zeng D, Guo M. Adaptive resource efficient microservice deployment in cloud-edge continuum. *IEEE Trans Parallel Distrib Syst*. 2021;33(8):1825-1840.
6. Fu K, Zhang W, Chen Q, et al. Qos-aware and resource efficient microservice deployment in cloud-edge continuum. Paper presented at: 2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS). IEEE. 2021:932-941.
7. Baarzi AF, Kesidis G. Showar: right-sizing and efficient scheduling of microservices. Proceedings of the ACM Symposium on Cloud Computing. ACM. 2021:427-441.
8. Hindman B, Konwinski A, Zaharia M, et al. Mesos: a platform for {fine-grained} resource sharing in the data Center. Paper presented at: 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11). USENIX. 2011.
9. Carrión C. Kubernetes scheduling: taxonomy, ongoing issues and challenges. *ACM Comput Surv*. 2022;55(7):1-37.
10. Han Y, Shen S, Wang X, Wang S, Leung VC. Tailored learning-based scheduling for kubernetes-oriented edge-cloud system. Paper presented at: IEEE INFOCOM 2021-IEEE Conference on Computer Communications. IEEE. 2021:1-10.
11. Rejiba Z, Chamanara J. Custom scheduling in Kubernetes: a survey on common problems and solution approaches. *J ACM*. 2022;55(7):1-37.
12. Mor B, Shabtay D, Yedidsion L. Heuristic algorithms for solving a set of NP-hard single-machine scheduling problems with resource-dependent processing times. *Comput Ind Eng*. 2021;153:107024.
13. Guo W, Tian W, Ye Y, Xu L, Wu K. Cloud resource scheduling with deep reinforcement learning and imitation learning. *IEEE Internet Things J*. 2020;8(5):3576-3586.
14. Chen J, Wang D, Zhao W. A task scheduling algorithm for Hadoop platform. *J Comput*. 2013;8(4):929-936.
15. Song W, Xiao Z, Chen Q, Luo H. Adaptive resource provisioning for the cloud using online bin packing. *IEEE Trans Comput*. 2013;63(11):2647-2660.
16. Azad P, Navimipour NJ. An energy-aware task scheduling in the cloud computing using a hybrid cultural and ant colony optimization algorithm. *Int J Cloud Appl Comput*. 2017;7(4):20-40.
17. Najafizadeh A, Salajegheh A, Rahmani AM, Sahafi A. Multi-objective task scheduling in cloud-fog computing using goal programming approach. *Clust Comput*. 2022;25(1):141-165.
18. Pezzella F, Morganti G, Ciaschetti G. A genetic algorithm for the flexible job-shop scheduling problem. *Comput Oper Res*. 2008;35(10):3202-3212.
19. Madhura R, Uthariaraj VR, Elizabeth BL. An efficient list-based task scheduling algorithm for heterogeneous distributed computing environment. *Softw: Pract Exper*. 2023;53(2):390-412.
20. Kaur A, Auluck N. Scheduling algorithms for truly heterogeneous hierarchical fog networks. *Softw: Pract Exper*. 2022;52(11):2411-2438.

21. Javanmardi S, Shojafar M, Persico V, Pescapè A. FPFTS: a joint fuzzy particle swarm optimization mobility-aware approach to fog task scheduling algorithm for internet of things devices. *Softw: Pract Exper*. 2021;51(12):2519-2539.
22. Mao H, Alizadeh M, Menache I, Kandula S. Resource management with deep reinforcement learning. Proceedings of the 15th ACM Workshop on Hot Topics in Networks. ACM. 2016:50-56.
23. Mao H, Schwarzkopf M, Venkatakrishnan SB, Meng Z, Alizadeh M. Learning scheduling algorithms for data processing clusters. Proceedings of the ACM Special Interest Group on Data Communication. 2019:270-288.
24. Chen Z. RIFLING: a reinforcement learning-based GPU scheduler for deep learning research and development platforms. *Softw: Pract Exper*. 2022;52(6):1319-1336.
25. Watkins CJ, Dayan P. Q-learning. *Mach Learn*. 1992;8:279-292.
26. Rummery GA, Niranjan M. *On-Line Q-Learning Using Connectionist Systems*. 37. University of Cambridge, Department of Engineering; 1994.
27. Mnih V, Kavukcuoglu K, Silver D, et al. Playing atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602. 2013.
28. Van Hasselt H, Guez A, Silver D. Deep reinforcement learning with double q-learning. Vol 30. AAAI. 2016.
29. Wang Z, Schaul T, Hessel M, Hasselt H, Lanctot M, Freitas N. Dueling network architectures for deep reinforcement learning. PMLR. 2016 1995-2003.
30. Hessel M, Modayil J, Van Hasselt H, et al. Rainbow: combining improvements in deep reinforcement learning. Vol 32. AAAI. 2018.
31. Williams RJ. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Reinforce Learn*. 1992;8:229-256.
32. Schulman J, Levine S, Abbeel P, Jordan M, Moritz P. Trust region policy optimization. PMLR. 2015:1889-1897.
33. Schulman J, Wolski F, Dhariwal P, Radford A, Klimov O. Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347. 2017.
34. Mnih V, Badia AP, Mirza M, et al. Asynchronous methods for deep reinforcement learning. PMLR. 2016:1928-1937.
35. Silver D, Lever G, Heess N, Degris T, Wierstra D, Riedmiller M. Deterministic policy gradient algorithms. PMLR. 2014:387-395.
36. Haarnoja T, Zhou A, Abbeel P, Levine S. Soft actor-critic: off-policy maximum entropy deep reinforcement learning with a stochastic actor. PMLR. 2018:1861-1870.
37. Huang J, Xiao C, Wu W. Rlsk: a job scheduler for federated kubernetes clusters based on reinforcement learning. Paper presented at: 2020 IEEE International Conference on Cloud Engineering (IC2E). IEEE. 2020:116-123.
38. Chen W, Xu Y, Wu X. Deep reinforcement learning for multi-resource multi-machine job scheduling. *arXiv preprint arXiv:1711.07440*. 2017.
39. Chung A, Park JW, Ganger GR. Stratus: Cost-aware container scheduling in the public cloud. Proceedings of the ACM symposium on cloud computing. ACM. 2018:121-134.
40. Wei-guo Z, Xi-lin M, Jin-zhong Z. Research on kubernetes' resource scheduling scheme. Proceedings of the 8th International Conference on Communication and Network Security. ACM. 2018:144-148.
41. Menouer T. KCSS: Kubernetes container scheduling strategy. *J Supercomput*. 2021;77(5):4267-4293.
42. Wojciechowski Ł, Opasiak K, Latusek J, et al. NetMARKS: network metrics-AwaRe kubernetes scheduler powered by service mesh. Paper presented at: IEEE INFOCOM 2021-IEEE Conference on Computer Communications. IEEE. 2021:1-9.
43. El Haj AG, Gil-Castiñeira F, Costa-Montenegro E. KubCG: a dynamic Kubernetes scheduler for heterogeneous clusters. *Softw: Pract Exper*. 2021;51(2):213-234.
44. Kaur A, Singh P, Singh Batth R, Peng LC. Deep-Q learning-based heterogeneous earliest finish time scheduling algorithm for scientific workflows in cloud. *Softw: Pract Exper*. 2022;52(3):689-709.
45. Liu N, Li Z, Xu J, et al. A hierarchical framework of cloud resource allocation and power management using deep reinforcement learning. Paper presented at: 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS). IEEE. 2017:372-382.

How to cite this article: Jian Z, Xie X, Fang Y, et al. DRS: A deep reinforcement learning enhanced Kubernetes scheduler for microservice-based system. *Softw Pract Exper*. 2023;1-25. doi: 10.1002/spe.3284