Latency-Aware Container Scheduling in Edge Cluster Upgrades: A Deep Reinforcement Learning Approach

Hanshuai Cui, Zhiqing Tang, Jiong Lou, Weijia Jia, *Fellow, IEEE,* and Wei Zhao, *Fellow, IEEE*

Abstract—In Mobile Edge Computing (MEC), Internet of Things (IoT) devices offload computationally-intensive tasks to edge nodes, where they are executed within containers, reducing the reliance on centralized cloud infrastructure. Cluster software upgrades are essential to maintain the efficient and secure operation of edge clusters. However, traditional cloud cluster upgrade strategies are ill-suited for edge clusters due to their geographically distributed nature and resource limitations. Therefore, it is crucial to properly schedule containers during edge cluster upgrades to minimize the impact on running tasks. This paper proposes a latency-aware container scheduling algorithm for efficient edge cluster upgrading. Specifically: 1) We formulate the online container scheduling problem for edge cluster upgrade to minimize the total task latency. 2) We propose a policy gradient-based reinforcement learning algorithm that addresses this problem by considering the characteristics of MEC, including heterogeneous resources, image distribution, and low-latency requirements. Subsequently, a location feature extraction method based on self-attention is designed to fully extract and utilize edge node distribution. 3) Experiments based on simulated and real-world data traces demonstrate that our algorithm reduces total task latency by approximately 30% compared to baseline algorithms.

Index Terms—Mobile edge computing, container scheduling, reinforcement learning, Internet of Things.

1 Introduction

In the era of the Internet of Things (IoT), Mobile Edge Computing (MEC) has emerged as a promising technology that brings computing and data storage closer to IoT devices [1]. This approach significantly reduces latency and bandwidth consumption associated with IoT devices and data center communications, making it more suitable for handling latency-sensitive tasks and services [2]. With the evolution of MEC, containers and Kubernetes are increasingly being used for service deployment [3]. Containers are lightweight and portable, frequently employed in MEC to deploy and manage applications while facilitating process and resource isolation [4]–[6]. Kubernetes [7] is a renowned open-source platform that offers robust tools for deploying, managing, and scaling containerized applications.

- *This work is supported in part by the Chinese National Research Fund (NSFC) under Grant 62272050 and 62302048; in part by the Guangdong Key Lab of AI and Multi-modal Data Processing, United International College (UIC), Zhuhai under Grant 2020KSYS007 sponsored by Guangdong Provincial Department of Education, and in part by the Interdisciplinary Intelligence SuperComputer Center of Beijing Normal University (Zhuhai). (Corresponding authors: Zhiqing Tang and Weijia Jia.)*

- *Hanshuai Cui is with School of Artificial Intelligence, Beijing Normal University, Beijing 100875, China, and also with Institute of Artificial Intelligence and Future Networks, Beijing Normal University, Zhuhai 519087, China. E-mail: hanshuaicui@mail.bnu.edu.cn*
- *Zhiqing Tang is with Institute of Artificial Intelligence and Future Networks, Beijing Normal University, Zhuhai 519087, China. E-mail: zhiqingtang@bnu.edu.cn*
- *Jiong Lou is with Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai 200240, China. E-mail: lj1994@sjtu.edu.cn*
- *Weijia Jia is with Institute of Artificial Intelligence and Future Networks, Beijing Normal University, Zhuhai 519087, China and also with Guangdong Key Lab of AI and Multi-Modal Data Processing, BNU-HKBU United International College, Zhuhai 519087, China. E-mail: jiawj@bnu.edu.cn*
- *Wei Zhao is with CAS Shenzhen Institute of Advanced Technology, Shenzhen 518055, China. E-mail: zhao.wei@siat.ac.cn*

An edge cluster consists of a network of interconnected edge nodes that collaborate with each other. Cluster upgrades can be performed for various reasons, such as security patches or the introduction of new features [8]. Common cluster upgrade strategies include in-place upgrades, canary upgrades, and rolling upgrades [9]. Such upgrades are essential but may negatively impact the IoT device experience. Especially in edge clusters, upgrades may negatively impact running containers due to geographical distribution and limited resources. In large-scale systems, upgrades take hours, or even much longer [10]. Considering that more than 10,000 system updates are carried out in the production environment per year [11], how to minimize the impact on running tasks during cluster upgrades poses an issue.

General resource unavailability at run-time usually refers to the exhaustion of computation or storage resources on a node, which does not affect the tasks already running on the node. However, during upgrades, a node cannot accept new tasks and can also not run tasks. It is necessary to reschedule them appropriately to guarantee the seamless execution of tasks during upgrades. This unavailability differs from the traditional resource shortage and poses significant challenges for scheduling policies. The default scheduling policy, taking into account factors such as resource availability, user preferences, and other constraints [12], does not meet the requirements of edge clusters in certain situations. Therefore, the first challenge is how to fully explore and utilize various information during edge cluster upgrades to enhance scheduling decisions. Given the limited storage and bandwidth resources at the edge, the distribution of images has a substantial impact on scheduling. Many scheduling policies consider migrating tasks from one edge node to another at run-time [13]–[15]. Most ignore the node upgrade status, but the node upgrade will affect its availability. For example, scheduling tasks to nodes that have been upgraded instead of nodes that have not yet been upgraded can reduce additional task migration latencies caused by node upgrades. This aspect is often ignored in traditional scheduling policies, which mainly focus on computation and communication resources without adequately considering the dynamic factor of node upgrades. However, simply considering the distribution of images and upgrade status is insufficient [16], the location information of the nodes is also crucial. Scheduling containers to distant edge nodes based solely on resource availability can lead to high communication latency. Therefore, the self-attention-based network is designed to extract the location information of nodes and tasks. This network can better understand the relative position relationship between edge nodes and tasks.

Another challenge lies in making online scheduling decisions that yield long-term benefits regarding reduced total task latency. Traditional scheduling algorithms primarily encompass rule-based, heuristic-based, or optimization-based methods [17]–[19]. Nonetheless, these algorithms cannot optimize long-term minimum latency in dynamic and diverse MEC environments. Recently, Reinforcement Learning (RL) algorithms have been widely applied to various optimization problems [20]. The policy gradient-based RL algorithm has exhibited promising outcomes for optimal resource scheduling problems in MEC [6]. Consequently, a policy gradient-based RL algorithm is proposed for making online scheduling decisions.

In this paper, we first model the Online Container Scheduling (OCS) problem for edge cluster upgrades to minimize the latency of IoT tasks while accounting for the geographic distribution, image locality, and limited resources of edge nodes. Second, the self-attention-based network is used to extract the location information of nodes and tasks. The policy network and value function of the RL agent are also meticulously designed. Then, we propose a policy gradient-based OCS algorithm. Finally, we implement a set of MEC scenarios based on simulated and real-world data to verify the effectiveness of the OCS algorithm and compare it with existing scheduling algorithms. Experimental results demonstrate that our proposed algorithm significantly reduces latency and outperforms all baseline algorithms.

In this extended version of our work [16], we focus on enhancing the understanding and effectiveness of latency-aware container scheduling in edge cluster upgrade scenarios. Firstly, we refine the problem modeling. This modeling is novel in that it fully considers the unique challenges of scheduling containers during edge cluster upgrades. Secondly, we improve the algorithm. This algorithm uniquely adapts to changing task locations, a feature not adequately addressed in previous works. Furthermore, we expand the scope of our experimental validation. These experiments not only verify the effectiveness of our algorithm but also its scalability and applicability in the real-world. Additionally, we introduce a comprehensive discussion on the practical deployment of our algorithm in a Kubernetes cluster. The discussion innovatively highlights challenges and considerations in practical implementation. In summary, the contributions of this paper are as follows:

1) We model the latency-aware container scheduling problem in edge cluster upgrade scenarios for the first time, including comprehensive motivations and case studies, to minimize total task latency.
2) To fully consider the distribution of nodes and the variation of task positions, a self-attention-based method is designed to extract the location information of nodes and tasks. Then, an OCS algorithm is proposed based on the policy gradient RL that continually makes online scheduling decisions. The upgrade status and the distribution of images are also taken into consideration.
3) We conduct large-scale experiments on simulated and real-world data traces to evaluate the effectiveness of the OCS algorithm. Our experimental results demonstrate that our proposed algorithm outperforms all baseline algorithms, reducing the total task latency by about 30%.

The remainder of the paper is organized as follows. Section 2 gives a brief overview of the related work and motivation. Section 3 presents the system model and problem formulation. Section 4 describes the OCS algorithm. The experimental settings and evaluation results are shown in Section 5. Section 6 gives some discussions. Finally, Section 7 concludes the paper and discusses future directions.

2 Related Work and Motivation

2.1 Container in Mobile Edge Computing

Containers are lightweight virtualization techniques that allow for the efficient deployment of applications in MEC environments [4]. As MEC continues to grow in popularity, it becomes evident that containers will play a critical role in enabling efficient and effective deployment at the edge. For example, Tang *et al.* [21] propose a container migration algorithm and architecture to support the fast migration of tasks. Regarding the problem of service migration in edge computing, Ma *et al.* [4] propose an edge computing platform architecture that supports the seamless migration of services. Lou *et al.* [13] introduce a method to jointly determine container assignment and layer sequencing to reduce container startup latency. Rossi *et al.* [22] present a Kubernetes-based orchestration tool to solve container deployment problems. Alameddine *et al.* [23] introduce a logic-based Benders decomposition approach to solve the complex dynamic task offloading and scheduling problem efficiently. Ayoub *et al.* [24] propose a multi-objective Integer Linear Programming (ILP) model and heuristic algorithms for efficient online Virtual Machine (VM) migration.

2.2 RL-based Scheduling

RL has received substantial attention and has been extensively employed in MEC for task scheduling. An RL agent can guarantee optimal resource allocation and enhanced system performance by perpetually learning from the environment and refining its policy. For example, Wang *et al.* [5] construct an RL-based microservice coordination scheme.

TABLE 1: Comparison with existing technologies

Paper	Research issue	Background	Optimization objective	Online	Distance	Resource availability	Methodology
[23]	Task offloading and scheduling	Low-latency IoT services	Latency				Benders decomposition
[15]	Task assignment and migration	Datacenter energy-saving	Energy	✓			RL-based
[24]	Virtual machine migration	Datacenter disaster resilience	Latency	✓	✓	✓	Heuristic-based
[25]	Virtual machine migration	Datacenter upgrades	Latency			✓	RL-based
[16]	Container scheduling	Edge cluster upgrades	Latency	✓		✓	RL-based
Ours	Container scheduling	Edge cluster upgrades	Latency	✓	✓	✓	RL-based, deep learning

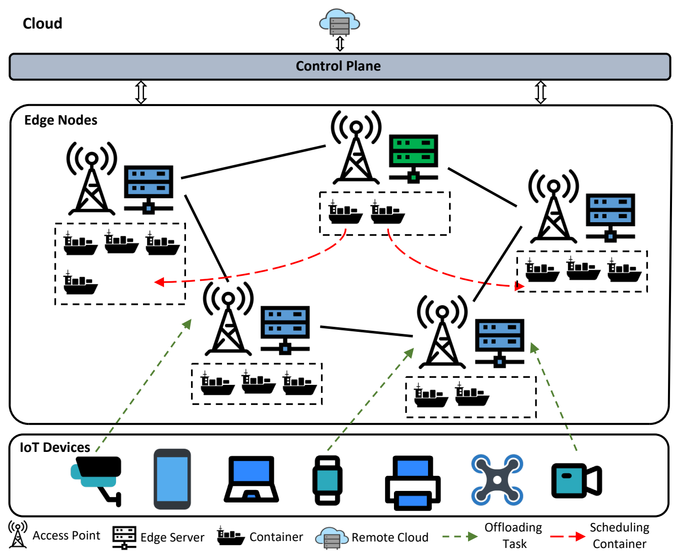


Fig. 1: An example of edge cluster upgrade.

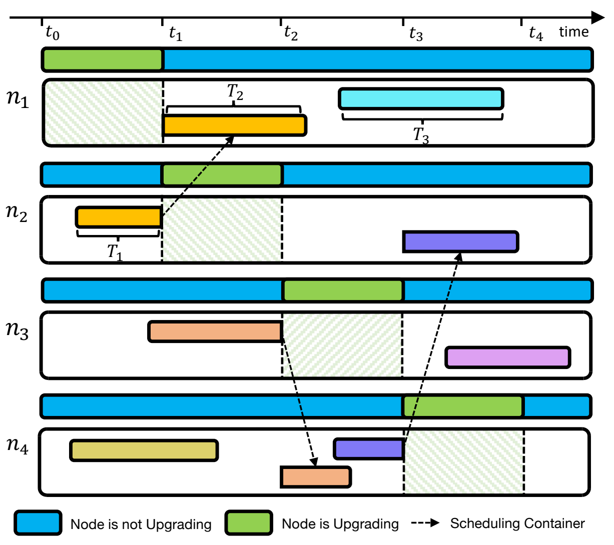


Fig. 2: Container scheduling in edge cluster upgrade.

Ho *et al.* [26] propose a MEC offloading framework that can jointly accomplish server selection, cooperative offloading, and handover. Aiming at the task migration problem in MEC, Liu *et al.* [27] design a distributed task migration algorithm based on the anti-fact multi-agent RL algorithm. Liu *et al.* [28] propose a load-balancing aware networking approach for efficient data processing in IoT edge systems and use an RL model. Ning *et al.* [29] propose an RL-based intent-driven traffic control system to optimize network resource orchestration and improve profits. Tang *et al.* [6] propose a layer dependency-aware learning scheduling algorithm based on container technology in MEC. Lou *et al.* [15] introduce an energy-efficient task scheduling method using RL for optimizing both task assignment and migration in data centers. Chen *et al.* [25] present a method that employs RL to optimize the scheduling of virtual machine migrations during datacenter upgrades. The detailed comparison between our approach and existing techniques is shown in TABLE 1.

2.3 Case Study

We model a one-round upgrade scenario for an edge cluster. As illustrated in Fig. 1, computation-intensive tasks from IoT devices are offloaded to edge nodes, where the results are processed and returned. Tasks are executed in containers, which require the corresponding image to be pulled locally before execution. All edge nodes in the cluster upgrade sequentially, with the edge node being upgraded in green and the edge node not being upgraded in blue. All containers on an edge node must be scheduled to another node before upgrading to ensure uninterrupted service. During the upgrade, new tasks are continuously offloaded to edge nodes, requiring decisions to be made regarding which node they are scheduled on. Meanwhile, resources (i.e., CPU, memory, etc.) on edge nodes are limited, and containers cannot be scheduled on nodes that do not meet resource requirements. Additionally, the node being upgraded is set as unschedulable. Different container images and resource quotas may be required for various tasks, so practical scheduling algorithms and resource management policies must be designed.

Fig. 2 illustrates the container scheduling process in an edge cluster upgrade scenario with four edge nodes. From left to right, it shows the progression of edge node upgrades over time. The green color represents an edge node currently upgrading and in an unschedulable state. The blue color represents an edge node that is not being upgraded, and tasks with resource requirements can be offloaded to this edge node to run in containers. Different colored rectangles within the edge nodes represent different tasks being executed. As can be seen from the figure, from time t_0 to t_1, edge node n_1 is upgrading. Container scheduling can be divided into two situations: 1) The edge node does not upgrade during task execution. T_3 is the execution time of a task offloaded from an IoT device to edge node n_1. Due to different geographical locations, image distribution, and resource availability, offloading tasks to different edge nodes can affect their execution time. 2) The edge node upgrades during task execution. T_1 is the execution time of a task offloaded from an IoT device to edge node n_2. At time

t_1, edge node n_1 completes its upgrade, and edge node n_2 begins its upgrade. To ensure uninterrupted service, all containers on the edge node must be scheduled to another edge node that is not undergoing an upgrade before the process begins. Therefore, the task is scheduled to the non-upgrading edge node n_1 to continue execution, and T_2 is the execution time of this task on edge node n_1. Therefore, unreasonable scheduling decisions can increase the total task latency and strain the bandwidth resources of edge nodes. We aim to make rational scheduling decisions to minimize the total execution time of all tasks, denoted as $\sum_{i=1} T_i$.

3 System model and problem formulation

3.1 System Model

For ease of reference, the main notations used in this paper are summarized in TABLE 2.

TABLE 2: Notations

Notation	Definition
$\mathbf{N}$	Edge node set
n	n^{th} edge node ($n \in \mathbf{N}$)
$C_n(t)$	CPU resource of edge node n at time t
$M_n(t)$	Memory resource of edge node n at time t
$D_n(t)$	Storage capacity of edge node n at time t
F_n	CPU frequency of edge node n
B_n	Bandwidth of edge node n
$p_n(t)$	Upgrade status of edge node n at time t
o_n	Location of edge node n
$\mathbf{K}$	Task set
k	k^{th} task ($k \in \mathbf{K}$)
c_k	CPU request of task k
m_k	Memory request of task k
f_k	CPU frequency request of task k
q_k	Image request of task k
s_{q_k}	Size of the image request of task k
d_k	Size of task k
t_k	Release time of task k
$o_k(t)$	Location of task k at time t
$\xi_{n,k}$	Uplink wireless transmission rate from task k to node n
$\mathbf{I}$	Image set
i	i^{th} image ($i \in \mathbf{I}$)
d_i	Size of image i
$T_{n,k}^{comm}$	Communication latency for task k on edge node n
$T_{n,k}^{down}$	Download latency for task k on edge node n
$T_{n,k}^{comp}$	Computation latency for task k on edge node n
$T_{n,k}^{total}$	Total latency for task k on edge node n
T_n^{queue}	Queuing download latency on edge node n
$x_{n,i}$	Whether image i is on edge node n
$y_{n,k}$	Whether task k is executed on edge node n

Edge Node. The set of edge nodes is defined as $\mathbf{N} = \{n_1, n_2, \ldots, n_{|\mathbf{N}|}\}$, where $|\cdot|$ indicates the number of elements in the set, e.g., $|\mathbf{N}|$ represents the number of edge nodes. The remaining CPU and memory resources in the edge node n can be represented by C_n and M_n. p_t is the upgrade status of the edge node n. Furthermore, the CPU frequency of edge node n is denoted as F_n, with the bandwidth defined as B_n. Besides, the number of images stored on the edge node is also influenced by the limitation of the storage capacity of the edge node D_n.

Task. The set of tasks offloaded by different IoT devices to the edge node is $\mathbf{K} = \{k_1, k_2, \ldots, k_{|\mathbf{K}|}\}$. Meanwhile, we assume that the resources requested by the task are the same as those occupied by the container. The CPU and memory resources requested by task k are c_n and m_n. Moreover, the data size of task k is d_k, the release time of task k is t_k, and the requested image of task k is q_k.

Container. The set of containers is denoted as $\mathbf{C} = \{c_1, c_2, \ldots, c_{|\mathbf{C}|}\}$. The set of images is denoted as $\mathbf{I} = \{i_1, i_2, \ldots, i_{|\mathbf{I}|}\}$, with each image associated with a container. The difference between a container and an image is only the writable container layer [30], so requesting a container is equivalent to requesting the corresponding image. The size of image i is denoted by d_i.

In MEC, each IoT device is connected to the edge node via a wireless link, while the edge node is connected to the remote cloud via a wired backhaul link [31]. The edge nodes are designed to process incoming tasks from various IoT devices quickly and efficiently through wireless. The image is stored in the remote cloud center and connected to the edge node through a wired connection. Therefore, the task is transmitted to the edge node through a wireless link, and the image is transmitted to the edge node through a wired connection.

3.2 Latency

Communication latency. In the communication model, the bandwidth allocation corresponds to the round-robin scheduling discipline and stands for equal resource sharing of these IoT devices associated with the edge node [32]. This design facilitates parallel processing for communication. In such a system, multiple tasks can be transmitted simultaneously, effectively eliminating transmission queuing latencies. The uplink wireless transmission rate $\xi_{n,k}$ from task k to node n is defined as [32]:

$$\xi_{n,k} = \frac{B_n}{U_n} log \left(1 + \frac{p_k h_{n,k}}{\sigma^2}\right), \tag{1}$$

where B_n is the bandwidth of edge node n, and U_n is the number of tasks transmitted to edge node n simultaneously. p_k is the transmission power, $h_{n,k}$ is the channel gain between the IoT device and the edge node, and σ represents the power of Gaussian white noise.

Therefore, the communication latency of task k transmitted to edge node n can be defined as follows:

$$T_{n,k}^{comm} = \frac{d_k}{\xi_{n,k}}. \tag{2}$$

where d_k represents the size of the data required to execute a task, which includes data files, configuration files, etc. Furthermore, similar to many studies [33], [34], we ignore the return communication latency of the result because the result is small compared with the task itself.

Download latency. Download latency refers to image download latency. Within Kubernetes, three different image-pulling policies are recognized: *IfNotPresent*, *Always*, and *Never*. Without a specified policy, *IfNotPresent* is the default. This implies that the image will be pulled from the remote repository if not stored locally. The download latency of the task can be obtained as follows:

$$T_{n,k}^{down} = x_{n,q_k} \times \left(\frac{s_{q_k}}{B_n} + T_n^{queue}\right), \tag{3}$$

where q_k is the image requested by task k and s_{q_k} is the size of the image required to process task k. It should be noted that d_k and s_{q_k} are distinct components; s_{q_k} is specifically the size of the image needed for task execution, while d_k includes all the other data required for the task. $x_{n,i} \in \{0,1\}$ is the binary variable to indicate whether image i is on edge node n. If $x_{n,i} = 1$, image i is on edge node n, otherwise not on edge node n. Each edge node is associated with a download queue, and the images in the queue are downloaded sequentially [13]. In this case, each download request will cause a queuing delay because it must wait for the previous image download to be completed. T_n^{queue} is the queuing download latency on edge node n. Therefore, if the image required to process the task is available locally, the download latency is 0.

Computation latency. Different tasks are executed in different containers, and all tasks are executed in parallel. The computation latency can be calculated as follows:

$$T_{n,k}^{comp} = \frac{f_k}{F_n}, \tag{4}$$

where f_k is the CPU frequency requested by task k, and F_n is the CPU frequency of edge node n.

In summary, the total latency of task k execution on node n can be denoted as:

$$T_k^{total} = T_{n,k}^{comm} + T_{n,k}^{down} + T_{n,k}^{comp}. \tag{5}$$

3.3 Problem Formulation and Analysis

Constraints. The containers need to be assigned certain resources to execute the tasks, while the total amount of resources on the edge node is limited. In instances where the resource limit of an edge node is surpassed, the functionality of the containers might be adversely affected. As a result, it becomes crucial to impose a constraint on the total quantity of resources allocated by the containers on an edge node. The resource limits on the edge node can be denoted as:

$$\sum_{k\in\mathbf{K}} y_{n,k} \times c_k \le C_n, \ \sum_{k\in\mathbf{K}} y_{n,k} \times m_k \le M_n, \ \forall n, \tag{6}$$

where the binary variable $y_{n,k} \in \{0,1\}$ indicates whether task k is executed on edge node n. If $y_{n,k} = 1$, the task k is executed on edge node n. Otherwise, the task k is not executed on edge node n.

Meanwhile, the storage space for the image on an edge node is limited, which can be defined as:

$$\sum_{i\in\mathbf{I}} x_{n,i} \times d_i \le D_n \ , \ \forall n. \tag{7}$$

Furthermore, like the previous studies [6], [35], [36], tasks are regarded as inseparable, so each task is scheduled to only one edge node, which can be expressed as:

$$\sum_{n\in\mathbf{N}} y_k^n = 1, \quad \forall k. \tag{8}$$

Problem Formulation. We aim to minimize the average total latency of the tasks during the edge cluster upgrade. The target is to find the best policy to minimize the latency while obeying the constraints. The problem is defined as:

Problem OCS.

$$\begin{aligned} \min T = \sum_{k\in\mathbf{K}} T_k^{total}&, \\ s.t.\ Eqs.\ (6)-(8)&, \\ x_{n,i} \in \{0,1\},&\ \forall n\in\mathbf{N},\ \forall i\in\mathbf{I}, \\ y_{n,k} \in \{0,1\},&\ \forall n\in\mathbf{N},\ \forall k\in\mathbf{K}. \end{aligned} \tag{9}$$

The objective of the OCS problem is to minimize the average total latency of the tasks during the edge cluster upgrade. In the OCS problem, $x_{n,i}$ and $y_{n,k}$ are variables indicating whether the image i is at the edge node n and whether the task k is scheduled to the edge node n, respectively. The OCS problem is a more complex variant of the bin-packing problem, where tasks must be scheduled to edge nodes to minimize the total latency. This problem is NP-hard, so the traditional algorithm may not get the optimal solution in a reasonable time [37]. Traditional heuristic algorithms fall short when dealing with the complexity of the OCS problem, and they cannot find the optimal solution in linear time. Meta-heuristic algorithms depend on a higher level of strategy to guide the search for solutions, but they struggle in the face of the unknown. In essence, as the number of edge nodes and tasks increases, there is a significant increase in the time required for these algorithms to find an acceptable solution.

The first-order transition probability of task resource requirements is an inherent property that is exploited in the scheduling problem. It is observed to remain quasi-static over extended periods, meaning that the state of the system changes is gradual and predictable over time [38]. Moreover, the arrival of tasks and the updating of the environment can be modeled as memoryless processes. Therefore, this problem can be modeled as an MDP [21]. This property simplifies the scheduling problem by reducing the dimensionality of the information needed for decision-making. RL is a powerful method that has proven effective in dealing with MDP problems [39]. Applying RL in container scheduling can offer significant improvements over traditional approaches.

4 Algorithms

4.1 Algorithm Settings

In this subsection, the settings in the RL algorithm are introduced, including state, action space, and reward.

State. The state s_t contains several parties, including node, task, and location states. Among them, the node state includes the resource and upgrade states. The resource state includes the CPU, memory, and storage capacity of the edge node at time t, as well as the CPU frequency and bandwidth of the edge node, which can be defined as:

$$\begin{aligned} s_t^{node,r} =& \{\mathbf{C}_n(t), \mathbf{M}_n(t), \mathbf{D}_n(t), \mathbf{F}, \mathbf{B}\} \\ =& \{C_1(t), C_2(t), \ldots, C_{|\mathbf{N}|}(t), M_1(t), M_2(t), \ldots, \\ & M_{|\mathbf{N}|}(t), D_1(t), D_2(t), \ldots, D_{|\mathbf{N}|}(t), \\ & F_1, F_2, \ldots, F_{|\mathbf{N}|}, B_1, B_2, \ldots, B_{|\mathbf{N}|}\}. \end{aligned} \tag{10}$$

The upgrade status of the node n at time t is denoted by the variable $p_n(t) \in \{0,1,2\}$. $p_n(t) = 0$ indicates that edge node n has not been upgraded at time t, $p_n(t) = 1$

indicates that edge node n is being upgraded at time t, and $p_n(t) = 2$ indicates that edge node n has been upgraded at time t. Then, the upgrade state of nodes can be denoted as:

$$s_t^{node,u} = \{\mathbf{P}_n(t)\} = \{p_1(t), p_2(t), \ldots, p_{|\mathbf{N}|}(t)\}. \tag{11}$$

Finally, the state for all edge nodes is defined as follows:

$$s_t^{node} = s_t^{node,r} \cup s_t^{node,u}. \tag{12}$$

The task state includes the requested resources and the status of the images requested to execute the task on each edge node, which can be denoted as:

$$\begin{aligned} s_t^{task,r} &= \{c_k, m_k, f_k, d_k, q_k\}, \\ s_t^{task,i} &= \{\mathbf{x}_{n,q_k}, \mathbf{t}_{n,q_k}\} = \{x_{1,q_k}, x_{2,q_k}, \ldots, \\ &\qquad x_{|\mathbf{N}|,q_k}, t_{1,q_k}, t_{2,q_k}, \ldots, t_{|\mathbf{N}|,q_k}\}, \end{aligned} \tag{13}$$

where t_{n,q_k} is the download time of the image in each edge node and can be calculated by Eq. (3).

Thus, the task state can be denoted as follows:

$$s_t^{task} = s_t^{task,r} \cup s_t^{task,i}. \tag{14}$$

The location state s_t^{loc} is a matrix sized $L \times W$, where L and W denote the length and width of the selected edge region, respectively. Within this matrix, the value of the location of the edge node is 1, the value of the location of the task is 2, and the value of the other locations is 0.

In summary, the state at time t is defined as:

$$s_t = s_t^{node} \cup s_t^{task} \cup s_t^{loc}. \tag{15}$$

Action space. The container used to execute tasks is scheduled by the scheduler. The OCS algorithm needs to determine which edge node to schedule. Therefore, the action space is the set of all edge nodes as follows:

$$a_t \in \mathbf{A} = \{1, 2, \ldots, |\mathbf{N}|\}. \tag{16}$$

Reward. Defining a proper reward is crucial in the RL algorithm. Since different tasks require different computation power, the completion time of tasks may vary by different orders of magnitude. Therefore, to improve the stability and effectiveness of the policy gradient algorithm, both the expected and actual latencies of the task are included in the reward, which can be defined as follows:

$$r_t = T_k^e - T_k^{total}, \tag{17}$$

where $T_k^e = \frac{f_k}{F_m}$ represents the expected total latency of the task, and F_m denotes the minimum value of the edge node CPU frequency. If the task is completed earlier than expected, the reward is positive, with the completion time being inversely proportional to the reward. Conversely, the reward is smaller. From a long-term perspective, the cumulative reward is $R_t = \sum_{t=0}^{T} \gamma^t r_t$, where γ is the discount factor with a value ranging between [0, 1].

4.2 Online Container Scheduling

Overview. The framework of the OCS algorithm is depicted in Fig. 3. Specifically, the node, task, and location states can be observed from the environment. After obtaining the features, they are embedded, concatenated, and fed into the policy network to make the corresponding scheduling decisions. The reward is subsequently obtained from the action taken. Finally, the policy gradient-based algorithm updates the policy network and value functions. Further specifics of these processes will be described in the following.

Feature Encoding. Feature encoding mainly comprises three components: node feature embedding, task feature embedding, and location feature encoding. Node and task feature embeddings are designed to map their respective features onto two distinctive embedding vectors.

The location feature is a two-dimensional matrix that can be interpreted as a single-channel image. The Vision Transformer (ViT) [40] is an application of the Transformer [41] architecture to the field of computer vision. ViT can directly capture the global dependencies between different areas of the image through the self-attention mechanism [40], which can help us to obtain the positional relationships between individual nodes, and between nodes and tasks.

The original location feature necessitates patch embedding. First, the location feature is segmented into N pieces of the shape (p, p, c), called "patch", where p is a predetermined parameter and c indicates the number of channels. Next, the patches thus obtained are flattened via a linear layer to condense the dimensions. Subsequently, a trainable position encoding is added to the final patches.

Following patch embedding is the stage of feature extraction. The output from patch embedding is used as the initial input to the stacked transformer encoders for global attention computation and feature extraction. The encoder is composed of two sublayers. The structure of the first sublayer comprises a Multi-Head Self-Attention (MSA) sublayer, a Layer Normalization (LN), and a residual connection. This first sublayer can be represented as:

$$\mathbf{z}'_\ell = \mathrm{MSA}\left(\mathrm{LN}\left(\mathbf{z}_{\ell-1}\right)\right) + \mathbf{z}_{\ell-1}, \quad \ell = 1 \ldots L, \tag{18}$$

where $\mathbf{z}_\ell$ denotes the features of transformer layers. The MSA sublayer exists in each encoder, an extension of Self-Attention (SA). It is crucial to note that Q, K, and V are obtained by multiplying the input $\mathbf{z}$ with three different trainable matrices. The computation of SA can be expressed as follows:

$$SA(\mathbf{z}) = softmax\left(\frac{QK^T}{\sqrt{e_k}}\right)V, \tag{19}$$

where e_k represents the embedding dimension of K, and the dot-product is actively scaled by $1/\sqrt{e_k}$ to standardize the variance of Q and K to 1.

MSA is the parallel computation of multiple self-attentions, termed "heads". Each head concentrates on a different aspect of the input and is subsequently concatenated. Using multi-head allows for a more nuanced extraction of features from different heads. Despite the overall computational workload being equivalent to that of a single head, the multi-head setup yields superior feature extraction results [41]. MSA can be computed as follows:

$$MSA(\mathbf{z}) = \left[SA_1(z); SA_1(z); \ldots; SA_k(z)\right]\mathbf{W}_O, \tag{20}$$

where $\mathbf{W}_O$ denotes a trainable matrix. Following this, the output from the first sublayer is fed into the second sublayer. The structure of the second sublayer consists of a Multi-Layer Perceptron (MLP), an LN, and a residual connection, which can be represented as follows:

$$\mathbf{z}_\ell = \mathrm{MLP}\left(\mathrm{LN}\left(\mathbf{z}'_\ell\right)\right) + \mathbf{z}'_\ell, \quad \ell = 1 \ldots L. \tag{21}$$

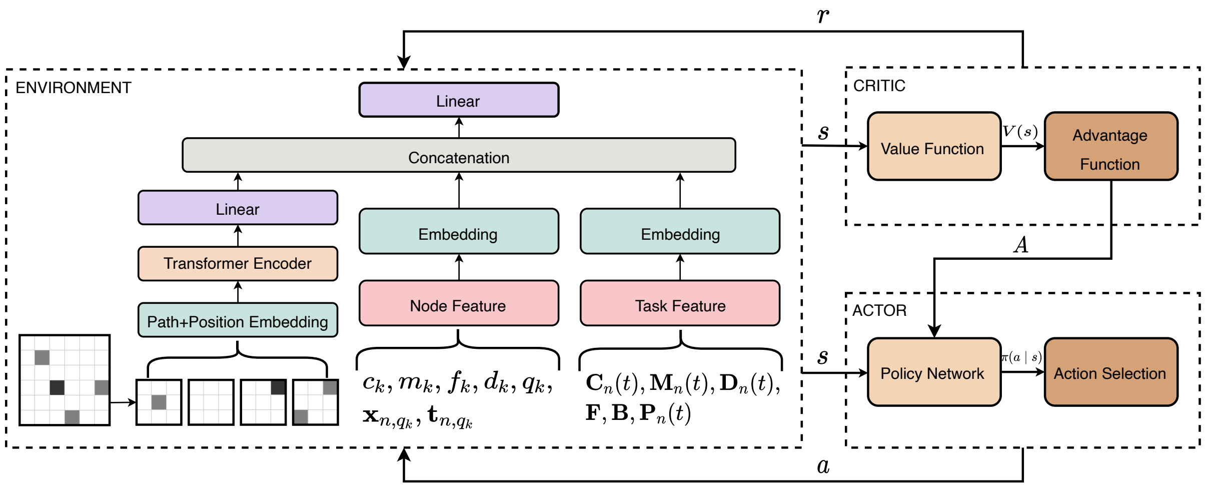


Fig. 3: Overview of the OCS algorithm.

Subsequently, we acquire the location feature encoding and concatenate it with the node and task feature embedding. We obtain the encoded feature after reducing the dimensions via a linear layer.

Training. The OCS algorithm is based on policy optimization. The policy gradient [42] is an RL algorithm that optimizes the policy for an expected return. Here, π_θ represents a policy with parameters θ. Supposing $J(\pi_\theta)$ is the objective function of the policy gradient, and the gradient of $J(\pi_\theta)$ is:

$$\nabla_\theta J\left(\pi_\theta\right)=\underset{\tau \sim \pi_\theta}{\mathrm{E}}\left[\sum_{t=0}^T \nabla_\theta \log \pi_\theta\left(a_t \mid s_t\right) A^{\pi_\theta}\left(s_t, a_t\right)\right], \tag{22}$$

where τ is a trajectory and $A^{\pi_\theta(s_t,a_t)}$ is the advantage function for the current policy.

While the policy gradient algorithm offers simplicity and efficiency, it can encounter training instability in practical scenarios. This instability often arises due to the indeterminacy of the step sizes in the policy gradient algorithm, potentially leading to suboptimal outcomes. The Trust Region Policy Optimization (TRPO) [43] is introduced to address this issue. The TRPO offers improved step size determination and policy updating, and the loss function can be customized as:

$$\mathcal{L}^{TRPO}\left(\theta_k, \theta\right)=\underset{s, a \sim \pi_{\theta_k}}{\mathrm{E}}\left[\frac{\pi_\theta(a \mid s)}{\pi_{\theta_k}(a \mid s)} A^{\pi_{\theta_k}}(s, a)\right]. \tag{23}$$

The TRPO has been successfully applied to various scenarios, but its computational complexity is substantial. Later, the Proximal Policy Optimization (PPO) algorithm [44] is proposed, which maintains effectiveness while significantly reducing computational complexity. Hence, the loss is customized into:

$$\begin{aligned}\mathcal{L}^{PPO}(\theta_k, \theta)=\underset{s, a \sim \pi_{\theta_k}}{\mathrm{E}}\Bigg[\Bigg(&\frac{\pi_\theta(a \mid s)}{\pi_{\theta_k}(a \mid s)} A^{\pi_{\theta_k}}(s, a), \\ &\operatorname{clip}\left(\frac{\pi_\theta(a \mid s)}{\pi_{\theta_k}(a \mid s)}, 1-\epsilon, 1+\epsilon\right) A^{\pi_{\theta_k}}(s, a)\Bigg)\Bigg],\end{aligned} \tag{24}$$

where $clip(x, y, z) = max(min(x, z), y)$ is a clip function to limit x to the range of $[y, z]$ and ϵ is a hyperparameter that represents the range of clips. Besides, PPO adopts the Generalized Advantage Estimator (GAE) [45] to compute the advantages, which can be calculated by:

$$\hat{A}_t=\delta_t+(\gamma \lambda) \delta_{t+1}+\cdots+\cdots+(\gamma \lambda)^{T-t+1} \delta_{T-1}, \tag{25}$$

where λ is the GAE parameter, $\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$ is the TD-error at time t. V is an approximate value function.

The OCS algorithm is presented in Algorithm 1. In each time slot, if IoT devices offload tasks to the edge cluster, their scheduled nodes need to be determined. Moreover, if an edge node starts upgrading in a certain time slot, the nodes for rescheduling all containers on that node also need to be determined sequentially. The replay memory $\mathbf{D}$ is first initialized for each episode. As shown in Lines 3 - 15, for each time slot t, the observation state s_t of the current time slot t is first obtained, then the action a_t is selected according to the policy, and the reward r_t is calculated. In addition, nodes that do not meet the scheduling requirements need to be filtered when selecting actions, e.g., upgrading or resources are insufficient. Afterward, the next state s_{t+1} is obtained. Finally, the transition is stored in the replay memory $\mathbf{D}$. As shown in Lines 18 - 25, for each training step k, the advantage estimation $\hat{A}_k$ is first computed based on the collected set of trajectories. Then, the stochastic gradient ascent algorithm with Adam [46] is used to maximize the objective function to update the policy. Finally, the results are output after all episodes are completed.

4.3 Computational Complexity Analysis

The OCS algorithm can be primarily divided into four parts: state observation, action selection, reward computation, and network update. The computational complexity of each part is analyzed in the following.

First, the state is shown in Eq. (15), and the complexity of this part can be calculated to be $O(|\mathbf{N}||\mathbf{I}|)$, where $|\mathbf{N}|$ and $|\mathbf{I}|$ represent the number of nodes and images, respectively. Second, action selection involves sequentially traversing all nodes. Therefore, action selection is a loop whose complexity can be represented as $O(|\mathbf{N}|)$. Following this is the reward calculation. The reward is calculated according to Eq. (17). The complexity of the reward calculation

Algorithm 1: The OCS Algorithm

```
Input: Initial policy parameters $\theta$, initial value
       function parameters $\phi$, clipping threshold $\epsilon$
Output: $a_t$
1  for episode ← 0,1,2,... do
2      Initialize replay memory $\mathbf{D} = \emptyset$ ;
3      for time slot ← 0,1,2,... do
4          Get the current state $s_t$ ;
5          Initialize the filtered set $\mathbf{A}' = \emptyset$;
6          for edge node $n$ ← 1,2,3,... do
7              if $p_n(t) = 1$ or the resources of the node are
               insufficient then
8                  Add the node to the filtered set:
                   $\mathbf{A}' \leftarrow \mathbf{A}' \cup \{n\}$
9              end if
10         end for
11         Select action $a_t$ from $\mathbf{A} \setminus \mathbf{A}'$ according to
           $\pi_\theta(a_t \mid s_t)$ ;
12         Execute action $a_t$ and obtain the reward $r_t$ ;
13         Get the next state $s_{t+1}$ ;
14         Store transition $(s_t, a_t, r_t, s_{t+1})$ in $\mathbf{D}$;
15     end for
16     Compute the cumulative reward:
       $R_t = \sum_{t=0}^{T} \gamma^t r_t$;
17     Compute the value function $V_\phi(s_t)$ for each state
       $s_t$;
18     for training step ← 0,1,2,... do
19         Compute the policy ratio $\rho_t(\theta) = \frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)}$ ;
20         Estimate advantages $\hat{A}_t$ by Eq. (25) ;
21         Compute and update the policy update by
           Eq. (24) ;
22         Compute the value function loss ues MSE
           function: $L_v(\phi) = \frac{1}{2}\|V_\phi(s_t) - R_t\|^2$;
23         Update the value function parameters:
           $\phi_{t+1} \leftarrow \phi_t$;
24     end for
25 end for
```

does not change with the number of nodes or tasks, so its complexity can be represented as $O(1)$. The node and task information is mapped through fully connected layers. Let there be L_1 number of hidden layers with G neurons per layer. The complexity of this part can be calculated to be $O(|\mathbf{N}||\mathbf{I}| \times G + L_1 \times G^2)$ [47].

As for the ViT, it includes several parts, of which the MSA operation has the highest computational complexity. Let D be the embedding dimension and N be the number of patches. The computational complexity of the MSA can be calculated as $O(N^2D + ND^2)$ [40], [48]. Here, the first term corresponds to the computation of self-attention, i.e., the calculation of the query, key, and value, and the second term corresponds to concatenating the outputs of the MSA. Assuming that the ViT contains L_2 layers, the complexity can be calculated as $O(L_2 \times (N^2D + ND^2))$.

For other parts, such as adding residual connections, performing layer normalization operations, and computing activation functions, their impact is generally much less than the parts mentioned above, so the computational complexity

TABLE 3: Hyperparameter Settings

Type	Hyperparameter	Value
Actor	Hidden layers	2 Full connection (128,64)
	Learning rate	1e-4
Critic	Hidden layers	2 Full connection (128,64)
	Learning rate	3e-4
	Loss Function	MSELoss
ViT	Input dimension	$L \times W$
	Hidden dimension	768
	Output dimension	10
Other	Discount factor γ	0.98
	GAE parameter λ	0.95
	Clipping threshold ϵ	0.2
	Batch size	32
	Activation function	ReLU
	Optimizer	Adam

of these operations can be negligible. Hence, the total complexity is $O(|\mathbf{N}||\mathbf{I}| \times G + L_1 \times G^2 + L_2 \times (N^2D + ND^2))$.

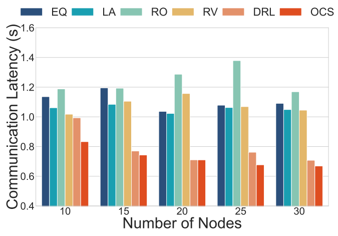
5 Evaluation

5.1 Experimental Settings

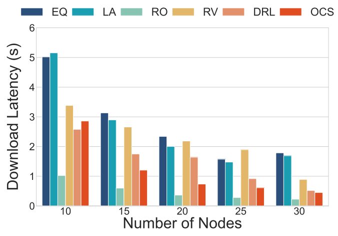
Parameter settings. Similar to [32], [49], we set the transmission power $p = 23dBm$ and the noise power spectrum density $\sigma = -174dBm/Hz$. According to the physical interference model [50], the channel gain between the IoT device and the edge node $h_{n,k}$ is $d_{n,k}^{-\alpha}$, where $d_{n,k}$ is the distance between the IoT device and the edge node and α is the path loss factor. The communication bandwidth between the IoT device and the edge node is [100, 200] Mbps.

The area of the selected edge region is $L \times W$, where L and W are the length and width of the selected edge region, respectively. The area of the selected edge region increases as the number of edge nodes increases, and the default area is $100m \times 100m$. All edge nodes are heterogeneous and randomly distributed, and the default number of edge nodes is 15. The CPU capacity of the edge node is set between [80,120] cores. The CPU frequency is set between [15,35] GHz, and the memory is set between [70,130] GB. The task is randomly generated in the selected edge region, and the task sizes are set from 10 KB to 10 MB. The types of requested images adhere to a normal distribution. If the requested image is not present on the assigned edge node, it must be downloaded from a remote repository, with the image size ranging from 300 MB to 1.5 GB. Initially, a specific number of tasks are carried out on each edge node, and the images corresponding to these tasks are also available on the edge nodes. The neural network input is scaled to the same order of magnitude. The hyperparameters of the OCS algorithm are listed in TABLE 3.

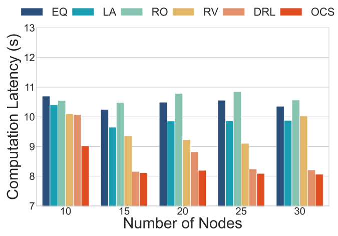
In addition to simulated data, real-world data traces are also used to increase the credibility of our experiment. The task data comes from the Alibaba Cluster Trace [51], which is collected from a large production cluster. After deleting missing values, filtering unreasonable values, and other preprocessing, 156,456 tasks are retained. The average CPU and memory of the task are 3.93 cores and 4.21GB, respectively, and the arrival times of the tasks are randomly generated. The container data is crawled from DockerHub [52], including 155 of the most commonly used images, with image sizes ranging from 1.84MB to 2.03GB.



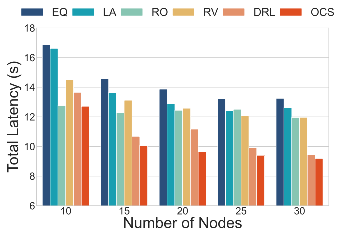
(a) Communication latency



(b) Download latency

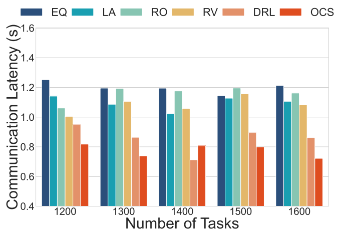


(c) Computation latency

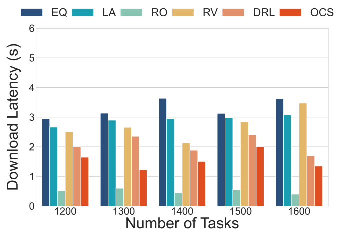


(d) Total latency

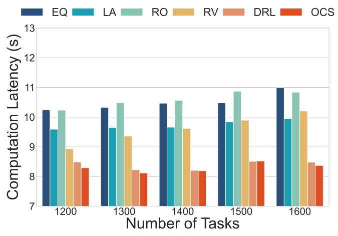
Fig. 4: Performance with different number of nodes



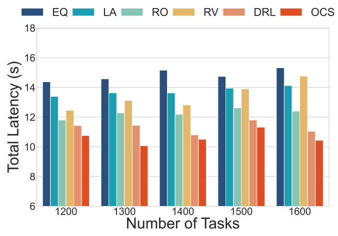
(a) Communication latency



(b) Download latency



(c) Computation latency



(d) Total latency

Fig. 5: Performance with different number of tasks

Baselines. We compare the OCS algorithm with several baseline algorithms to demonstrate the effectiveness of our proposed algorithm:

1) **EQ** (EqualPriority): This algorithm sets the weight of all nodes to 1, treating them as equal in priority for scheduling tasks. It does not consider any specific features of the nodes, such as resource availability or image locality.
2) **LA** (LeastAllocated): This is a scheduling policy related to the resource usage of the node. It selects the node with the least allocated resources for scheduling tasks, ensuring that nodes with more available resources are prioritized for new tasks.
3) **RO** (RO-min) [24]: This is a heuristic approach that performs migration and minimizes overall network resource occupation and optimizes online virtual machine migration for disaster resilience.
4) **RV** (Revan) [25]: This is an RL-based scheduler designed to minimize the total migration time of virtual machines during datacenter upgrades by adaptively choosing the best destination physical machine for each virtual machine migration.
5) **DRL** [16]: DRL algorithm is the PPO algorithm based on the actor-critic framework and contains no location information.

Among them, (1)-(2) are built-in scheduling policies in Kubernetes [12]. Moreover, LA is a greedy algorithm that selects nodes with more resources. By comparing the OCS algorithm with these baselines, we can demonstrate the advantages of our proposed algorithm in terms of minimizing total task latency.

5.2 Experimental Results

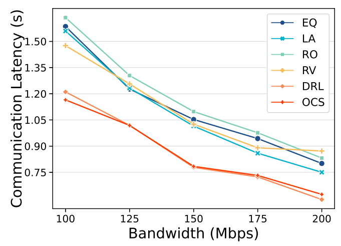
To validate the effectiveness of the OCS algorithm, a series of experiments are performed comparing its task latency performance against several baseline algorithms under varying conditions. Subsequently, the training process of the OCS algorithm is evaluated in detail.

Performance with different numbers of nodes. Fig. 4 shows the average task latency obtained through various container scheduling algorithms as the number of nodes increases, including communication latency, download latency, computation latency, and total latency. The average task latency decreases with an increasing number of nodes across all algorithms because more nodes provide more scheduling options, allowing containers to be allocated to more suitable nodes, like those closer or with more resources, thereby reducing the average total task latency.

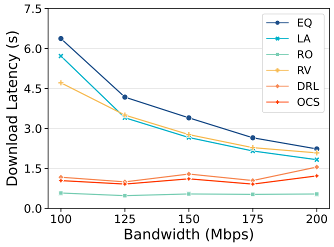
Specifically, Fig. 4(a) illustrates the communication latency of tasks. As shown in the figure, the communication latency of tasks from various container scheduling algorithms decreases as the number of nodes increases due to more scheduling options that favor closer nodes or those with higher network bandwidth. Unquestionably, the OCS algorithm, which considers location information, outperforms the others, decreasing communication latency by approximately 50% compared to the least efficient EQ algorithm.

Fig. 4(b) illustrates the download latency. The disparity in download latency among different algorithms is more significant than the variations observed in other latencies. As inferred from this figure, the RO scheduling algorithm significantly reduces download latency compared to other algorithms. This is because the RO algorithm accounts for the size of the image to be downloaded. Moreover, the download latencies for the DRL and OCS algorithms are less than the LA, RV, and EQ algorithms.

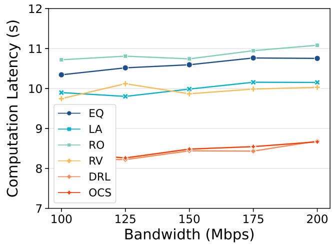
Fig. 4(c) depicts the computation latency. The disparity in computation latency among different algorithms is less pronounced than download and communication latencies.



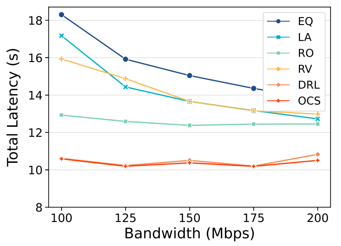
(a) Communication latency



(b) Download latency

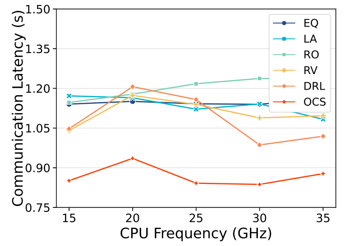


(c) Computation latency

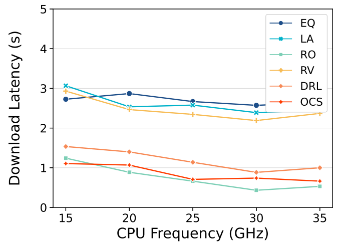


(d) Total latency

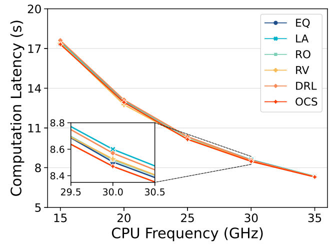
Fig. 6: Performance with different bandwidth



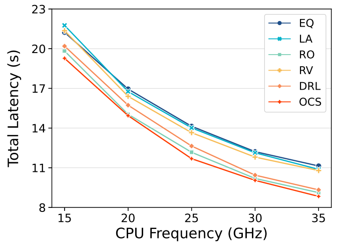
(a) Communication latency



(b) Download latency



(c) Computation latency



(d) Total latency

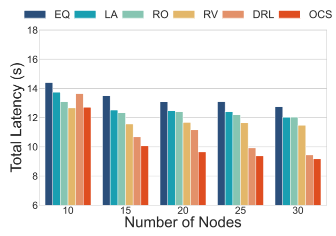
Fig. 7: Performance with different number CPU frequency

This is because containers executing distinct tasks operate independently on the node, precluding interference among tasks. The OCS algorithm continues to deliver superior performance, whereas the EQ and RO algorithms, which disregard the actual resources of the nodes, perform poorly.

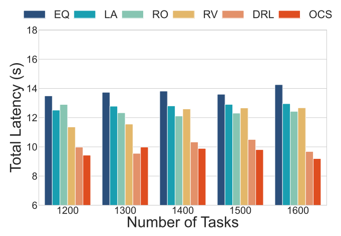
Fig. 4(d) presents the total latency of tasks. In our experiments, the OCS algorithm consistently outperformed regardless of the number of nodes in the environment. In summary, as the number of nodes increases, the total latency sorting is OCS < DRL < RO < RV < LA < EQ. More specifically, the average total latency for varying numbers of nodes reduces by 45%, 35%, 22%, 30%, and 6% when compared to the EQ, LA, RO, RV, and DRL algorithms, respectively. Hence, the OCS algorithm demonstrates superior performance, irrespective of the number of nodes.

Performance with different numbers of tasks. The variation of average task latency as the number of tasks increases is illustrated in Fig. 5. We analyze and present the variation in communication latency, download latency, and computation latency for different numbers of tasks in Figs. 5(a) - 5(c). In these figures, as the number of tasks increases, the influence of download latency gradually becomes dominant, emerging as the primary factor affecting the overall task latency. In contrast, the changes in communication latency and computation latency are relatively minor. This occurs because as the number of tasks increases, there is a greater demand for different types of images. Hence, in the design of the container scheduling algorithm, particular attention should be given to optimizing download latency to enhance the efficiency and performance of task scheduling. Additionally, The OCS algorithm has the lowest communication and computation latency, while the RO algorithm has the lowest download latency.

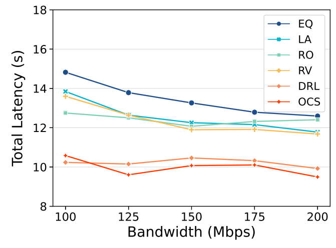
As illustrated in Fig. 5(d), the total latency of the task



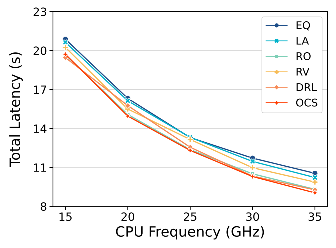
(a) Different number of nodes



(b) Different number of tasks



(c) Different bandwidth

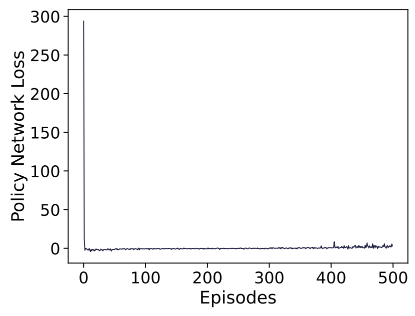


(d) Different CPU frequency

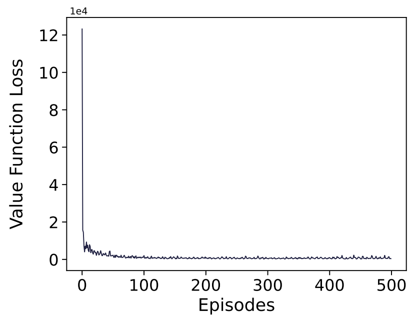
Fig. 8: Total latency with real-world data traces

is determined by considering the above latency in concert. As the number of tasks increases, the relative performance among different algorithms, in terms of total latency, follows the sequence: OCS < DRL < RO < LA < RV < EQ. In particular, in comparison to the DRL, RV, RO, LA, and EQ algorithms, the OCS algorithm reduces total scheduling latency by 5%, 41%, 19%, 39%, and 46%, respectively.

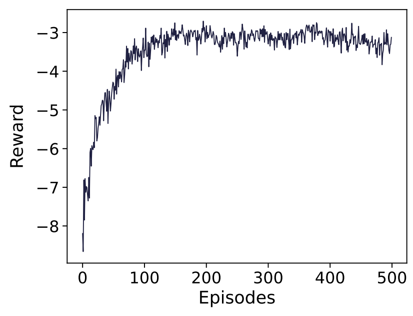
Performance with different bandwidth. It can be seen from Fig. 6 that the total latency decreases as the bandwidth increases. The main reason for this reduction is the decrease in download latency, which is directly influenced



(a) Policy network loss



(b) Value function loss



(c) Reward

Fig. 9: Policy network Loss, value function Loss, and reward of the OCS algorithm

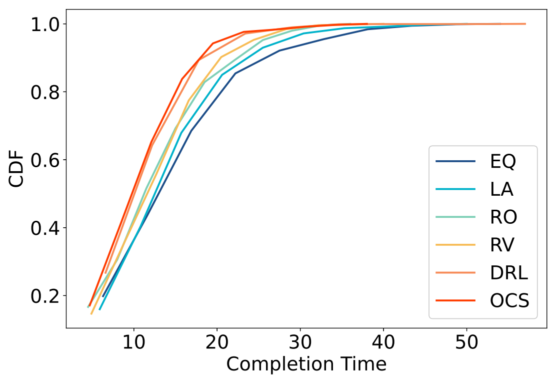


Fig. 10: CDF of total latency

by bandwidth. As bandwidth increases, the time required to download the container images is significantly reduced, thereby decreasing total latency. Overall, OCS outperforms all other algorithms in minimizing total task latency. In particular, OCS reduces total task latency by approximately 30% compared to the baseline algorithms.

Performance with different CPU frequency. Fig. 7 shows the total latency of different algorithms with the CPU frequency change. The results indicate a decrease in total task latency as the CPU frequency increases. This decrease can be attributed to the accelerated task execution resulting from the increased CPU frequency. Our OCS algorithm can maintain the best performance when the edge node CPU frequency changes.

Performance with real-world data traces. This experiment is completely based on real-world data traces. We separately calculate the total latency of various algorithms with changes in the number of nodes, the number of tasks, bandwidth, and CPU frequency, as shown in Fig. 8. As can be seen from the figure, when using real-world data traces, the total task latency obtained by container scheduling through the OCS algorithm is relatively low.

Fig. 8(a) illustrates the change in total task latency as the number of nodes increases. The results of all algorithms are similar when the number of nodes is small. As the number of nodes increases, the gap between the OCS algorithm and the baselines gradually widens. The reason is that as the number of schedulable nodes increases, the baseline algorithm may be unable to make optimal scheduling decisions. Fig. 8(b) shows the change in total task latency as the number of tasks increases. As the OCS algorithm can schedule tasks requesting the same image to the same node, it does not lead to a significant increase in total latency. Fig. 8(c) and Fig. 8(d) represent the change in total task latency with the variation in bandwidth and CPU frequency, respectively. As node resources increase, any scheduling policy will result in a decrease in total task latency. However, the OCS algorithm almost consistently outperforms the baseline algorithm.

In summary, the OCS algorithm delivers the lowest total latency under different conditions and outperforms other algorithms. Specifically, the OCS algorithm reduces the total latency than EQ, LA, RO, RV, and DRL algorithms by 39%, 29%, 32%, 27%, and 5% on average, respectively. These results demonstrate that our proposed algorithm can perform well in real production clusters.

Statistical analysis. TABLE 4 shows the statistical results of the paired t-test that determine the significance level of the proposed algorithm compared with other algorithms regarding total latency. Specifically, the table provides the mean and standard deviation (SD) of the baseline algorithms, alongside the t-statistic and corresponding p-value. Our null hypothesis is: "There is no significant difference in the performance between the OCS algorithm and the baseline algorithm". [53] The table shows a meaningful difference between the OCS algorithm and baselines, while the p-value is lower than 0.05 in all cases. There is statistically significant evidence at the 5% significance level to suggest that the proposed container scheduling algorithm performs better than the baseline algorithm. Thus, the null hypothesis is rejected, which proves the fact that differences are significant.

TABLE 4: Statistical comparison of the OSC algorithm with other baselines

Metrics	EQ	LA	RO	RV	DRL
Mean	14.23	13.56	12.52	12.50	11.27
SD	1.01	0.91	1.07	0.84	0.69
T-statistic	-8.59	-7.51	-4.40	-5.21	-2.25
P-value	1.79×10^{-6}	7.08×10^{-6}	8.61×10^{-4}	2.18×10^{-4}	0.043

Convergence of the OCS algorithm. Fig. 9 shows the training process of the OCS algorithm. As the training steps increase, both the policy network loss and value function loss decrease rapidly and eventually fluctuate near a specific value, indicating that the algorithm has converged. Fig. 9(a) illustrates the policy network loss. The change initially decreases and then increases, mainly because the policy network outputs relatively random policies in the early training phase, but with continued training, it learns more rational policies and stabilizes around a specific value.

Fig. 9(b) depicts the fluctuation of the value function loss. During the initial stages, the difference between the predicted and true values of the value function network is significant, resulting in a high-value function loss. However, after the 100-th episode, the value function network learns a more accurate value, resulting in a rapid decrease and subsequent stabilization of the loss. Fig. 9(c) illustrates the reward, indicating an initial sharp increase followed by a plateau. The algorithm identifies a promising policy and fine-tunes the policy network and value function. Occasional declines in reward during training may be due to the exploration of novel environments, making the current policy ineffective, but the algorithm quickly recalibrates, restoring normal function over time. Overall, the convergence rate of the OCS algorithm is remarkably rapid, suggesting its ability to learn optimal policies within a brief timeframe.

CDF of total latency. Fig. 10 presents the Cumulative Distribution Function (CDF) of total latency. The OCS algorithm shows a higher proportion of tasks with shorter total latency when compared to other algorithms. These results enhance the potential of our OCS algorithm as a promising solution for container scheduling in edge cluster upgrades.

Computation resources for different algorithms. As shown in TABLE 5, we use torch.profiler [54] to record the Random Access Memory (RAM), Video RAM (VRAM), and execution time for different algorithms. EQ and LA algorithms require the shortest execution time because they are simple judgments and comparisons. The introduction of the ViT network improves the performance of the OSC algorithm, although the cost is to increase the execution time. However, the computation resources and execution time required by the OCS algorithms are within acceptable limits, demonstrating that our algorithm has low complexity and can be run in real-time. Furthermore, it can be clearly discerned that the execution time does not increase significantly with the number of tasks.

TABLE 5: Computation resources of different algorithms

Algorithm	Tasks	RAM	VRAM	Execution Time
EQ	1000	-	-	0.92ms
LA	1000	-	-	1.24ms
RO	1000	-	-	4.01ms
RV	1000	315.50Kb	787.23Kb	1.93ms
DRL	1000	588.04Kb	630.00Kb	3.78ms
OCS	1000	1177.60Kb	1259.52Kb	17.93ms
OCS	2000	1064.00Kb	1265.00Kb	19.40ms
OCS	3000	1373.45Kb	1567.01Kb	20.98ms
OCS	5000	2644.99Kb	2922.50Kb	21.27ms

6 Discussion

6.1 Feasibility of deployment

In this section, we discuss the feasibility of deploying the OCS algorithm in Kubernetes cluster. A Kubernetes cluster generally comprises one master node and multiple worker nodes. The master node, usually equipped with superior processing capabilities, is responsible for managing the overall cluster. Meanwhile, the worker nodes are responsible for executing the tasks.

We can implement the OCS algorithm through the scheduling framework of Kubernetes. First, we need to program the OCS scheduler using the Go programming language. Next, we should package the scheduler files into a container image and push the image to a repository to ensure the consistency of the custom scheduler in different environments. Then, we should define a Deployment for the scheduler, which determines how the OCS scheduler runs in the edge cluster. When deploying the custom scheduler, it is necessary to write related Kubernetes configuration files, and set the correct resource quotas, permissions, and environment variable configurations. Finally, we should deploy the OCS algorithm in the master node and install Prometheus for monitoring. This setup allows the algorithm to receive various information (e.g., remaining resources, image locality) collected by Prometheus from each worker node. The algorithm makes scheduling decisions by processing this aggregated data and utilizes the Kubernetes API to schedule tasks.

Our proposed OCS algorithm is based on RL. Due to the large amount of coding work required to achieve interaction between RL and Kubernetes, we are currently striving to develop relevant code. However, additional effort is still needed to achieve comprehensive integration between RL and Kubernetes, which will be the focus of our future work.

6.2 Feasibility of real-time running

In this section, we discuss the feasibility of real time running of the OCS algorithm.

The OCS algorithm is an online algorithm. This design considers the dynamics and unpredictability of the tasks generated by IoT devices. The algorithm continuously receives and processes data, making real-time decisions without batch processing or waiting for a complete dataset. To ensure the efficiency and accuracy of our algorithm in real-time scenarios, we can first train our model using historical data. Using historical data in training enables it to handle various scenarios and variations in task generation. Once trained, the algorithm utilizes the learned model parameters to schedule in actual environments. The online nature of our algorithm, combined with its initial training phase, ensures that it can handle the randomness of tasks generated by IoT devices well. This algorithm ensures performance even in highly dynamic and unpredictable environments, as the algorithm constantly interacts with the environment.

For the real-time performance of the algorithm, please refer TABLE 5. The average task arrival time in the Alibaba Cluster Trace [51] is approximately 581ms, much longer than the time required to execute our algorithm. It reinforces our position that the interval between arriving tasks is greater than the execution time of the algorithm. Moreover, the number of tasks running concurrently is limited due to the resource capacity of the data center. Therefore, the proposed algorithm can run in real-time.

7 Conclusion

This paper proposes a latency-aware container scheduling algorithm for IoT services in edge cluster upgrades. First, we comprehensively model the OCS problem, considering communication, download, and computation latency. Second, a location feature extraction method based on ViT has been proposed, utilizing the distribution information of edge nodes. Then, a policy gradient-based RL algorithm is proposed to make online scheduling decisions, which fully considers the distinctive features of MEC. Finally, experiments are conducted on the simulated edge cluster, and the experimental results demonstrate that our algorithm achieves approximately 30% lower total latency than the baseline algorithm. The RL algorithm employed in our study, while effective in decision-making, requires extensive historical data to train the neural network. The algorithm may perform poorly in practical applications without insufficient training data. Besides, our validation relies mainly on simulations and real-world data traces, but more complex and variable factors in real environments may affect the performance of the algorithm. In future work, we will further consider the impact of real-time decision-making on algorithm performance and deploy this algorithm in the Kubernetes cluster.

References

[1] P. Porambage, J. Okwuibe, M. Liyanage, M. Ylianttila, and T. Taleb, "Survey on multi-access edge computing for internet of things realization," *IEEE Communications Surveys and Tutorials*, vol. 20, no. 4, pp. 2961–2991, 2018.

[2] L. Qian, Y. Wu, F. Jiang *et al.*, "Noma assisted multi-task multi-access mobile edge computing via deep reinforcement learning for industrial internet of things," *IEEE Transactions on Industrial Informatics*, vol. 17, no. 8, pp. 5688–5698, 2020.

[3] T. Goethals, F. De Turck *et al.*, "Extending kubernetes clusters to low-resource edge devices using virtual kubelets," *IEEE Transactions on Cloud Computing*, vol. 10, no. 4, pp. 2623–2636, Oct. 2022.

[4] L. Ma, S. Yi, N. Carter, and Q. Li, "Efficient live migration of edge services leveraging container layered storage," *IEEE Transactions on Mobile Computing*, vol. 18, no. 9, pp. 2020–2033, Sep. 2019.

[5] S. Wang, Y. Guo, N. Zhang, P. Yang, A. Zhou, and X. Shen, "Delay-aware microservice coordination in mobile edge computing: A reinforcement learning approach," *IEEE Transactions on Mobile Computing*, vol. 20, no. 3, pp. 939–951, Mar. 2021.

[6] Z. Tang, J. Lou, and W. Jia, "Layer dependency-aware learning scheduling algorithms for containers in mobile edge computing," *IEEE Transactions on Mobile Computing*, vol. 22, pp. 3444–3459, 2023.

[7] Kubernetes. [Online]. Available: https://kubernetes.io/

[8] A. Decan, T. Mens, A. Zerouali, and C. De Roover, "Back to the past–analysing backporting practices in package dependency networks," *IEEE Transactions on Software Engineering*, vol. 48, no. 10, pp. 4087–4099, 2021.

[9] L. Bass, I. Weber, and L. Zhu, *DevOps: A software architect's perspective*. Addison-Wesley Professional, 2015.

[10] D. Sun, A. Fekete, V. Gramoli, G. Li, X. Xu, and L. Zhu, "R2c: Robust rolling-upgrade in clouds," *IEEE Transactions on Dependable and Secure Computing*, vol. 15, no. 5, pp. 811–823, Sep. 2018.

[11] Y. Wang, S. Jiang, and B. Cui, "Tjosconf: Automatic and safe system environment operations platform," in *Proceedings of the 2022 11th International Conference on Software and Computer Applications (ICSCA)*, Jun. 2022, pp. 21–28.

[12] Kubernetes scheduler. [Online]. Available: https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/

[13] J. Lou, H. Luo, Z. Tang, W. Jia, and W. Zhao, "Efficient container assignment and layer sequencing in edge computing," *IEEE Transactions on Services Computing*, vol. 16, pp. 1118–1131, 2022.

[14] W.-K. Lai, Y.-C. Wang, and S.-C. Wei, "Delay-aware container scheduling in kubernetes," *IEEE Internet of Things Journal*, vol. 10, pp. 11 813–11 824, 2023.

[15] J. Lou, Z. Tang, and W. Jia, "Energy-efficient joint task assignment and migration in data centers: A deep reinforcement learning approach," *IEEE Transactions on Network and Service Management*, vol. 20, no. 2, pp. 961–973, 2023.

[16] H. Cui, Z. Tang, J. Lou, and W. Jia, "Online container scheduling for Low-Latency IoT services in edge cluster upgrade: A reinforcement learning approach," in *Proceedings of the 12th IEEE/CIC International Conference on Communications in China (ICCC)*, Aug. 2023, p. 6.

[17] A. Mehrabi, M. Siekkinen, and A. Ylä-Jääski, "Edge computing assisted adaptive mobile video streaming," *IEEE Transactions on Mobile Computing*, vol. 18, no. 4, pp. 787–800, 2018.

[18] X. Hu, C. Masouros, and K.-K. Wong, "Reconfigurable intelligent surface aided mobile edge computing: From optimization-based to location-only learning-based solutions," *IEEE Transactions on Communications*, vol. 69, no. 6, pp. 3709–3725, 2021.

[19] J. Chen, Y. Yang, C. Wang, H. Zhang, C. Qiu, and X. Wang, "Multitask offloading strategy optimization based on directed acyclic graphs for edge computing," *IEEE Internet of Things Journal*, vol. 9, no. 12, pp. 9367–9378, 2021.

[20] R. S. Sutton and A. G. Barto, "Reinforcement learning: An introduction," *IEEE Transactions on Neural Networks*, vol. 16, pp. 285–286, 2005.

[21] Z. Tang, X. Zhou, F. Zhang *et al.*, "Migration modeling and learning algorithms for containers in fog computing," *IEEE Transactions on Services Computing*, vol. 12, no. 5, pp. 712–725, Sep. 2019.

[22] F. Rossi, V. Cardellini, F. L. Presti, and M. Nardelli, "Geo-distributed efficient deployment of containers with kubernetes," *Computer Communications*, vol. 159, pp. 161–174, 2020.

[23] H. A. Alameddine, S. Sharafeddine *et al.*, "Dynamic task offloading and scheduling for low-latency iot services in multi-access edge computing," *IEEE Journal on Selected Areas in Communications*, vol. 37, no. 3, pp. 668–682, Mar. 2019.

[24] O. Ayoub, A. De Sousa, S. Mendieta, F. Musumeci, and M. Tornatore, "Online virtual machine evacuation for disaster resilience in inter-data center networks," *IEEE Transactions on Network and Service Management*, vol. 18, no. 2, pp. 1990–2001, Jun. 2021.

[25] C. Ying, B. Li, X. Ke, and L. Guo, "Raven: Scheduling virtual machine migration during datacenter upgrades with reinforcement learning," *Mobile Networks and Applications*, vol. 27, no. 1, pp. 303–314, Feb. 2022.

[26] T. M. Ho and K.-K. Nguyen, "Joint server selection, cooperative offloading and handover in multi-access edge computing wireless network: A deep reinforcement learning approach," *IEEE Transactions on Mobile Computing*, vol. 21, no. 7, pp. 2421–2435, Jul. 2022.

[27] C. Liu, F. Tang, Y. Hu, K. Li, Z. Tang, and K. Li, "Distributed task migration optimization in mec by extending multi-agent deep reinforcement learning approach," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 7, pp. 1603–1614, Jul. 2021.

[28] Q. Liu, T. Xia, L. Cheng, M. van Eijk, T. Ozcelebi, and Y. Mao, "Deep reinforcement learning for load-balancing aware network control in iot edge systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 6, pp. 1491–1502, Jun. 2022.

[29] Z. Ning, K. Zhang, X. Wang *et al.*, "Joint computing and caching in 5g-envisioned internet of vehicles: A deep reinforcement learning-based traffic control system," *IEEE Transactions on Intelligent Transportation Systems*, vol. 22, no. 8, pp. 5201–5212, Aug. 2021.

[30] I. Miell and A. Sayers, *Docker in practice*. Simon and Schuster, 2019.

[31] T. X. Tran, A. Hajisami, P. Pandey, and D. Pompili, "Collaborative mobile edge computing in 5g networks: New paradigms, scenarios, and challenges," *IEEE Communications Magazine*, vol. 55, no. 4, pp. 54–61, Apr. 2017.

[32] Y. Wang, X. Tao, X. Zhang, P. Zhang, and Y. T. Hou, "Cooperative task offloading in three-tier mobile computing networks: An admm framework," *IEEE Transactions on Vehicular Technology*, vol. 68, no. 3, pp. 2763–2776, Mar. 2019.

[33] M. Chen and Y. Hao, "Task offloading for mobile edge computing in software defined ultra-dense network," *IEEE Journal on Selected Areas in Communications*, vol. 36, no. 3, pp. 587–597, Mar. 2018.

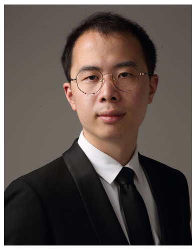
[34] J. Du, L. Zhao, J. Feng, and X. Chu, "Computation offloading and resource allocation in mixed fog/cloud computing systems with min-max fairness guarantee," *IEEE Transactions on Communications*, vol. 66, no. 4, pp. 1594–1608, Apr. 2018.

[35] Q. Chen, Z. Zheng, C. Hu, D. Wang, and F. Liu, "On-edge multi-task transfer learning: Model and practice with data-driven task

allocation," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 6, pp. 1357–1371, Jun. 2020.
[36] J. Liu, J. Ren, Y. Zhang, X. Peng, Y. Zhang, and Y. Yang, "Efficient dependent task offloading for multiple applications in mec-cloud system," *IEEE Transactions on Mobile Computing*, vol. 22, no. 4, pp. 2147–2162, Apr. 2023.
[37] D. S. Hochba, "Approximation algorithms for np-hard problems," *ACM Sigact News*, vol. 28, pp. 40–52, 1997.
[38] Z. Han, H. Tan, G. Chen *et al.*, "Dynamic virtual machine management via approximate markov decision process," in *Proceedings of the 35th IEEE International Conference on Computer Communications (INFOCOM)*. IEEE, 2016, pp. 1–9.
[39] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015.
[40] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn *et al.*, "An image is worth 16x16 words: Transformers for image recognition at scale," *arXiv preprint arXiv:2010.11929*, Jun. 2021.
[41] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proceedings of the 31st Advances in Neural Information Processing Systems (NIPS)*, vol. 30. Curran Associates, Inc., 2017.
[42] R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Proceedings of the 12th Advances in Neural Information Processing Systems (NIPS)*, vol. 12. MIT Press, 1999.
[43] J. Schulman, S. Levine, P. Abbeel *et al.*, "Trust region policy optimization," in *Proceedings of the 33rd International Conference on Machine Learning (ICML)*. PMLR, 2015, pp. 1889–1897.
[44] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, Aug. 2017.
[45] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel, "High-dimensional continuous control using generalized advantage estimation," *arXiv preprint arXiv:1506.02438*, 2015.
[46] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.
[47] I. Sarkar, M. Adhikari, S. Kumar, and V. G. Menon, "Deep reinforcement learning for intelligent service provisioning in software-defined industrial fog networks," *IEEE Internet of Things Journal*, vol. 9, no. 18, pp. 16 953–16 961, Sep. 2022.
[48] K. Han, A. Xiao, E. Wu, J. Guo, C. Xu, and Y. Wang, "Transformer in transformer," in *Proceedings of the 34th Advances in Neural Information Processing Systems (NIPS)*, vol. 34. Curran Associates, Inc., 2021, pp. 15 908–15 919.
[49] X. Chen, "Decentralized computation offloading game for mobile cloud computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 4, pp. 974–983, Apr. 2015.
[50] A. Goldsmith, *Wireless communications*. Cambridge University Press, 2005.
[51] Alibaba cluster trace program. [Online]. Available: https://github.com/alibaba/clusterdata/
[52] S. Fu, R. Mittal, L. Zhang, and S. Ratnasamy, "Fast and efficient container startup at the edge via dependency scheduling," in *Proceedings of the 3rd USENIX Workshop on Hot Topics in Edge Computing (HotEdge)*, 2020.
[53] J. Wang, F. Nan, P. Yang, Y. Yang, and J. Qi, "An effective approach for predicting p-value using high-dimensional snps data with small sample size," in *20th International Conference on Ubiquitous Computing and Communications (IUCC)*, Dec. 2021, pp. 339–344.
[54] Pytorch documentation. [Online]. Available: https://pytorch.org/docs/

Hanshuai Cui received the B.S. degree from School of Information Science and Engineering, Qufu Normal University, China, in 2020. He is currently pursuing the Ph.D. degree in School of Artificial Intelligence, Beijing Normal University, China. His current research interests include mobile edge computing, resource allocation, and reinforcement learning.

Zhiqing Tang received the B.S. degree from School of Communication and Information Engineering, University of Electronic Science and Technology of China, China, in 2015 and the Ph.D. degree from Department of Computer Science and Engineering, Shanghai Jiao Tong University, China, in 2022. He is currently an assistant professor with the Advanced Institute of Natural Sciences, Beijing Normal University, China. His current research interests include edge computing, resource scheduling, and reinforcement learning.



Jiong Lou received the B.S. degree and Ph.D. degree in the Department of Computer Science and Engineering, Shanghai Jiao Tong University, China, in 2016 and 2023. Since 2023, he has held the position of research assistant professor in the Department of Computer Science and Engineering, Shanghai Jiao Tong University, China. His current research interests include edge computing, task scheduling and container management.

Weijia Jia (Fellow, IEEE) is currently a Chair Professor, Director of BNU-UIC Institute of Artificial Intelligence and Future Networks, Beijing Normal University (Zhuhai) and VP for Research of BNU-HKBU United International College (UIC) and has been the Zhiyuan Chair Professor of Shanghai Jiao Tong University, China. He received BSc/MSc from Center South University, China, in 82/84 and Master of Applied Sci./PhD from Polytechnic Faculty of Mons, Belgium in 92/93, respectively, all in computer science. From 95-13, he worked at City University of Hong Kong as a professor. His contributions have been recognized as optimal network routing and deployment, anycast and QoS routing, sensors networking, AI (knowledge relation extractions; NLP, etc.), and edge computing. He has over 600 publications in the prestige international journals/conferences and research books, and book chapters. He has served as area editor for various prestige international journals, chair and PC member/skeynote speaker for many top international conferences. He is the Fellow of IEEE and the Distinguished Member of CCF.

Wei Zhao (Fellow, IEEE) completed his undergraduate studies in physics at Shaanxi Normal University, China, in 1977, and received his MSc and PhD degrees in Computer and Information Sciences at the University of Massachusetts at Amherst in 1983 and 1986, respectively. Prof. Zhao has served important leadership roles in academic including the Chief Research Officer at the American University of Sharjah, the Chair of Academic Council at CAS Shenzhen Institute of Advanced Technology, the eighth Rector of the University of Macau, the Dean of Science at Rensselaer Polytechnic Institute, the Director for the Division of Computer and Network Systems in the U.S. National Science Foundation, and the Senior Associate Vice President for Research at Texas A&M University. Prof. Zhao has made significant contributions to cyber-physical systems, distributed computing, real-time systems, and computer networks. His research results have been adopted in the standard of Survivable Adaptable Fiber Optic Embedded Network. Professor Zhao was awarded the Lifelong Achievement Award by the Chinese Association of Science and Technology in 2005.