Layer-Aware Cost-Effective Container Updates With Edge-Cloud Collaboration in Edge Computing

Hanshuai Cui, Zhiqing Tang, *Member, IEEE*, Yuan Wu, *Senior Member, IEEE*, and Weijia Jia, *Fellow, IEEE*

***Abstract*—Containers have become popular for deploying applications in Edge Computing (EC) for their seamless integration and easy deployment. Frequent container updates are essential to enhance performance and introduce new challenges for cutting-edge applications such as large language models and digital twins. However, traditional container update methods result in substantial download costs and task interruptions, which are unacceptable for latency-sensitive tasks in resource-constrained EC. Existing work has largely overlooked the layered structure of container images. By leveraging this layered structure, duplicate downloads can be reduced, and various layers can be transferred from other edges, reducing burden on the remote cloud. In this paper, we model the layer-aware container update problem with edge-cloud collaboration to minimize update and scheduling costs. We present the Layer-aware Edge-cloud collaborative Container Update (LECU) algorithm based on reinforcement learning to make container update decisions. Moreover, a task scheduling algorithm is devised to schedule tasks affected by container updates to other edges, minimizing the impact of task interruptions. We implement our LECU algorithm on an edge system with real-world data traces to demonstrate its effectiveness and conduct larger-scale simulations to evaluate its scalability. Results demonstrate that our algorithms reduce container update and task scheduling costs by 14% and 19%, respectively, compared to baselines.**

***Index Terms*—Container update, edge computing, layer sharing, edge-cloud collaboration, reinforcement learning.**

I. Introduction

CONTAINERS have gained popularity in Edge Computing (EC) due to their efficiency in facilitating the deployment of services [1]. An image file containing the binaries, code, system tools, and configuration files must exist locally for running a container [2]. Recent cutting-edge applications, including large language models and digital twins, can be easily deployed to EC using containerization [3], [4]. These applications require rapid updates to meet user demands effectively. Besides, regular updates are also necessary to add new features, enhance security, and fix bugs. Real-world production cluster data [5], [6], collected from August 1 to October 24, 2022 using a one-minute sampling interval, shows 922 version updates occurred. The average interval between updates ranges from several days to weeks, with 95% of updates completed within eight hours. The container update process involves downloading the new image version from the cloud, halting the existing container, and initializing the newly one. Despite being lightweight, container updates may be slow in practice due to limited bandwidth for image downloads in EC environments. Moreover, many image downloads can place a heavy burden on the remote cloud. Therefore, finding a fast and effective way to update containers in EC is crucial.

Edge nodes have limited computation and storage resources, making them incapable of supporting large-scale parallel updates. Therefore, traditional update strategies for software are not suitable for container updates in EC. Container update strategies in cloud data centers are also unsuitable for EC. Unlike cloud data centers, EC environments impose stringent constraints on resource availability and network stability. For example, the blue-green update [7] requires running two container versions simultaneously, making it unsuitable for resource-limited EC environments. Similarly, the Rolling Update (RU) [8] involves taking one or more containers out of service for updating and then reincorporating them once the update is complete. The number of containers to update is often predefined and not dynamically adjusted according to the current load.

Several container update strategies specifically improved for EC have been proposed [9], [10]. However, two critical issues often overlooked in existing studies have been identified. First, images are composed of layers [11]. During container updates, only the modified layers need to be downloaded, whereas traditional software updates require downloading the entire

Received 30 December 2024; revised 19 June 2025; accepted 20 June 2025. Date of publication 25 June 2025; date of current version 3 October 2025. This work was supported in part by the National Natural Science Foundation of China (NSFC) under Grant 62272050 and Grant 62302048, in part by the Science and Technology Development Fund of Macau SAR under Grant FDCT 0158/2022/A, in part by the Research Fund of Guangxi Key Lab of Multi-Source Information Mining and Security under Grant MIMS24-07, in part by the Guangdong Key Lab of AI and Multi-modal Data Processing, Beijing Normal-Hong Kong Baptist University, Zhuhai under Grant 2023-2024 Grants sponsored by Guangdong Provincial Department of Education, in part by Institute of Artificial Intelligence and Future Networks and Engineering Center of AI and Future Education, Guangdong Provincial Department of Science and Technology, China, in part by Zhuhai Science-Tech Innovation Bureau under Grant 2320004002772, and in part by the Interdisciplinary Intelligence Super Computer Center of Beijing Normal University at Zhuhai. Recommended for acceptance by C. Lin. *(Corresponding authors: Zhiqing Tang; Weijia Jia.)*

Hanshuai Cui is with the School of Artificial Intelligence, Beijing Normal University, Beijing 100875, China, and also with the Institute of Artificial Intelligence and Future Networks, Beijing Normal University, Zhuhai 519087, China (e-mail: hanshuaicui@mail.bnu.edu.cn).

Zhiqing Tang is with the Institute of Artificial Intelligence and Future Networks, Beijing Normal University, Zhuhai 519087, China, and also with the Guangxi Key Lab of Multi-source Information Mining & Security, Guangxi Normal University, Guilin 541004, China (e-mail: zhiqingtang@bnu.edu.cn).

Yuan Wu is with the State Key Laboratory of Internet of Things for Smart City, University of Macau, Macau SAR, China (e-mail: yuanwu@um.edu.mo).

Weijia Jia is with the Institute of Artificial Intelligence and Future Networks, Beijing Normal University, Zhuhai 519087, China, and also with the Guangdong Key Lab of AI and Multi-Modal Data Processing, Beijing Normal-Hong Kong Baptist University, Zhuhai 519087, China (e-mail: jiawj@bnu.edu.cn).

Digital Object Identifier 10.1109/TMC.2025.3583153

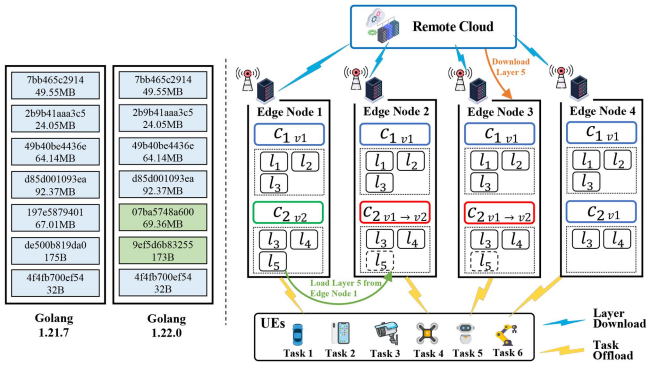


Fig. 1. Edge-cloud collaborative container update process. Left: Two versions of a Golang container image share common layers, with only two layers differing. Right: Edge nodes download layers from neighboring nodes instead of the remote cloud, reducing download latency.

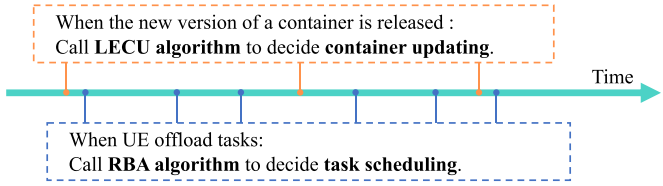


Fig. 2. Two-timescale container update framework. Container updates occur on a large timescale, while task scheduling operates on a small timescale.

installation package and reinstalling the software [12], leading to significant bandwidth waste. Fig. 1 shows two versions of the *Golang* image (used to provide a Golang programming environment), highlighting only two layers of differences between them. Second, distributed file systems can share layers across different nodes [13]. Container layers can be shared across multiple nodes, and employing efficient coordination of cross-node updates can reduce overall update latency. If a node lacks specific layers, it can load them from other nodes or download them from the remote cloud. However, the layered structure of containers, while enabling partial layer reuse, introduces complexity in coordinating distributed layer sharing across heterogeneous edge nodes in EC. The unique opportunity of layer sharing across edge nodes, which is not typically feasible in centralized cloud environments. Fig. 1 illustrates that nodes n_2 and n_3 need layer l_5 to update container c_2. If l_5 is loaded from node n_1, the burden of the cloud can be reduced. As a result, layer sharing and edge-cloud collaboration can enable more efficient container updates.

Although making layer-aware container update decisions with edge-cloud collaboration shows promise in shortening update time, several challenges remain to be addressed. *The first challenge is how to determine the optimal number of concurrent container updates.* Existing studies often rely on static update strategies that do not account for user mobility and load variability in EC [8]. Consequently, these methods cannot determine the optimal number of containers that can be updated simultaneously while ensuring that the remaining nodes still provide adequate service. Updating too many edge nodes can lead to unhandled tasks due to resource constraints, whereas updating too few prolongs the overall update duration. We have to face the dilemma of the longer total update time or the number of task interruptions caused by the container updates. Existing studies mainly focus on utilizing layer sharing to reduce container deployment or migration costs but overlook the effects of layer sharing during container updates [14], [15], [16], [17]. For downloading only a few layers, increasing the update proportion will expedite the process. Therefore, the update proportion should be adjusted dynamically based on the load of edge nodes and layer distribution.

The second challenge is how to jointly optimize the container update sequence and task scheduling to minimize update time and reduce update-related task interruptions. The update sequence affects the resources of nodes, thereby impacting task scheduling. Prioritizing the update of low-load edge nodes is advisable to reduce task interruptions. Meanwhile, later updated edge nodes can load layers from other edge nodes rather than download from the remote cloud, reducing the burden on the remote cloud. The update sequence and task scheduling should be considered comprehensively. However, existing studies often focus solely on updating strategies or task scheduling independently [18], [19], neglecting the mutual impacts of both problems. Reinforcement Learning (RL) algorithms have been extensively tackled diverse optimization problems [20], [21]. The reward function in RL enables comprehensive consideration of long-term benefits as well as the effects of layer sharing and edge-cloud collaboration. Therefore, we propose the Layer-aware Edge-cloud collaborative Container Update (LECU) algorithm based on the Soft Actor-Critic (SAC) RL algorithm to make container update decisions. Additionally, to prevent massive task disruptions caused by the updates, an efficient task scheduling algorithm named Resource Balance Allocation (RBA) is devised to balance the resource consumption across all the edge nodes.

This paper explores the layer-aware container update issue in the edge-cloud network. The proposed container update framework, shown in Fig. 2, employs the LECU algorithm for managing container updates upon new releases and the RBA algorithm for task scheduling when User Equipments (UEs) offload tasks to the edge node. We implement these algorithms on a small-scale edge system to validate their practicality and applicability. Docker facilitates performing the container update process, with images crawled from Docker Hub [22]. The state data is collected from the system, and an NVIDIA GPU is employed to train the RL agent. Once trained, the agent is deployed on the edge system to make update decisions whenever a container update request arises. Additionally, we conduct more extensive experiments to evaluate scalability. The results demonstrate that the LECU algorithm outperforms all baselines. The contributions are summarized below.

1) We formulate the layer-aware edge-cloud collaborative container update problem to reduce total update time and minimize task interruptions caused by container updates in a resource-constrained EC environment.

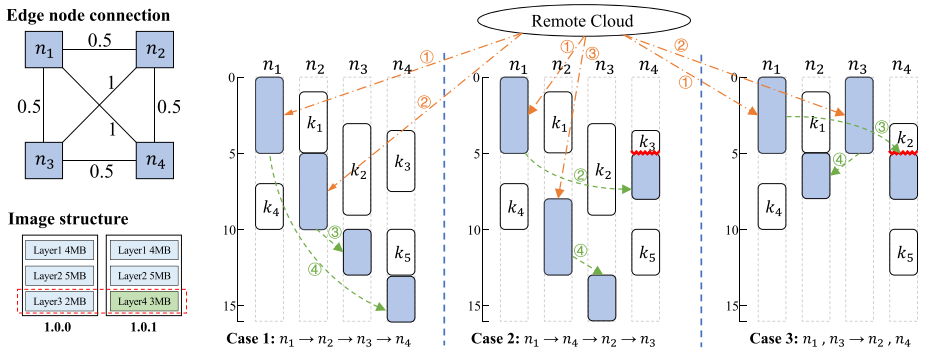


TABLE A: Example tasks

Task	Start time	Finish time
k_1	1	5
k_2	3	9
k_3	4	8
k_4	7	10
k_5	9	12

TABLE B: Update results under different cases

	Update proportion	Update sequence	Completed task	Update time
Case1	25%	1→2→3→4	5	16
Case2	25%	1→4→2→3	4	16
Case3	50%	1→3→2→4	3	8

Fig. 3. Motivation example comparing three update situations. Case 1 completes 5 tasks with 16 units update time. Case 2 interrupts task k_3, completing 4 tasks. Case 3 reduces update time to 8 units but interrupts task k_2 and offloads task k_3 to the cloud, completing only 3 tasks. Trade-offs between update time and task interruptions are highlighted.

2) We propose a two-timescale container update framework that employs the LECU algorithm for container update decisions and the RBA algorithm for task scheduling to minimize task interruptions during updates.
3) We validate the performance of our algorithms on an edge system using real-world data traces and assess their scalability through larger-scale simulations. Results show that the LECU algorithms outperform all baseline methods.

II. Related Work

A. Layer-Aware Container Scheduling

Containers are lightweight virtualization techniques that enable the efficient deployment of applications in EC environments [4]. Some recent works have focused on the layered structure of containers to reduce image download costs. For example, Ma et al. [23] propose an efficient live migration method for offloading services by leveraging a layered storage system, significantly reducing migration time and user-perceived interruptions. Tang et al. [11] present a container migration algorithm for edge-assisted vehicular networks, utilizing layer sharing to reduce total latency. Shi et al. [15] introduce an optimization algorithm to enhance reliability in microservice deployment with layer sharing. Zeng et al. [17] propose a cost-efficient algorithm for placing dependent microservices while considering the layered structure. Lou et al. [24] introduce a method for jointly determining container assignment and layer sequencing to minimize container startup latency. Gu et al. [14] investigate layer sharing and introduce a non-sequential layer fetching strategy to expedite microservice initialization. However, existing research has ignored the differences in layers during container updates and the potential for sharing layers between nodes [13], which could further minimize image downloads.

B. Container Update Strategies

Container update issues are still in the early stages in EC. Zhang et al. [9] present a chunk reuse mechanism to enhance container update efficiency and reduce network resource consumption. Al Maruf et al. [10] present an algorithm to determine the ideal quantity of fog nodes needed for Over-the-Air (OTA) updates. Chen et al. [19] present a method that employs RL to optimize the scheduling of virtual machine migrations during datacenter upgrades. Sun et al. [8] increase the robustness of rolling upgrades by improving error detection and predictability. Hassan et al. [18] present novel scheduling algorithms for OTA software updates. In our previous work [21], we propose a RL-based task scheduling algorithm for edge cluster upgrades. However, the existing studies failed to target the container layer update in the EC scenarios specifically, facing limited resources. In this paper, we jointly address container updates and task scheduling with layer sharing for the first time, aiming to reduce update time and task interruptions caused by updates.

III. Motivation

To better understand the scenario of container updates, a motivating example is illustrated in Fig. 3. An edge-cloud network consisting of a remote cloud and four edge nodes is considered. We aim to update the container version from 1.0.0 to 1.0.1. Considering the layer sharing between different containers, updating this container is equivalent to downloading layer l_4. For simplicity, it is assumed that each node executes only a single task simultaneously. The data transfer rate between edge nodes and the remote cloud is 0.6, while the edge nodes are interconnected with varying transmission rates. Table A in Fig. 3 lists five different tasks. Blue shaded areas represent the container update duration on each node. Orange arrows represent downloading layers from the remote cloud, while green arrows show layer transfers between nodes, with their directionality reflecting the layer transmission. For example, in Case 1, the arrow from edge node n_1 to edge node n_3 indicates that node n_3 loads shared layers from node n_1 instead of downloading them from the remote cloud. In addition, the circle numbers, e.g.,①, ②, ③, and ④, represent the update order of nodes. Task interruptions due to updates are highlighted in red. Three different update proportions and update sequences are considered as follows.

In Case 1, with an update proportion of 25%, one container is updated at a time in the sequence $n_1 \rightarrow n_2 \rightarrow n_3 \rightarrow n_4$. Nodes n_1 and n_2 download layers from the cloud, while nodes n_3 and n_4 load layers from n_2 and n_1, respectively. The total update time is 16 units, and the number of tasks completed is 5 units.

In Case 2, containers are updated in the sequence of $n_1 \rightarrow n_4 \rightarrow n_2 \rightarrow n_3$, with the same update proportion of 25%. At time 5, the container on node n_4 begins updating, forcing the eviction of the ongoing task k_3 on node n_4, which subsequently fails. The total update time remains 16 units, but the number of tasks completed is 4 units.

In Case 3, the update proportion is 50%, allowing two containers to update simultaneously. The container update sequence is $n_1, n_3 \rightarrow n_2, n_4$. Task k_3 is offloaded to the remote cloud due to the unavailability of edge nodes. At the same time, the task k_3 on the node n_4 is evicted, and the execution fails due to the node update. Consequently, only 3 units of tasks are completed. As more containers are updated simultaneously, the total update time decreases to 8 units.

More details of the container update results are shown in Table B in Fig. 3. This example highlights the need to balance container updates and task scheduling to optimize the tradeoff between task completion ratio and update time duration, illustrating the complexity and significance of container update problems and the interconnected decisions involved.

IV. System Model and Problem Formulation

A. Overview of the System

This section describes the system model for container updates and task scheduling. Then, the problem is formulated. Table I enumerates the key notations.

Edge node: Let $\mathbf{N} = \{n_1, n_2, \ldots, n_{|\mathbf{N}|}\}$ represent the set of edge nodes, where $|\cdot|$ indicates the size of a set. Hence, $|\mathbf{N}|$ stands for the overall number of such nodes. The remaining CPU cores and memory in the edge node n at time t are represented by $U_n(t)$ and $M_n(t)$, respectively. The initial CPU cores and memory in the edge node n are U_n^I and M_n^I, respectively. We denote the CPU frequency of edge node n as F_n, its bandwidth as B_n, and its storage capacity as D_n. The remote cloud, referred to as $n_{|\mathbf{N}|+1}$, can be considered an edge node with limitless capacity.

Container: The container set is denoted as $\mathbf{C} = \{c_1, c_2, \ldots, c_{|\mathbf{C}|}\}$. The image set is defined as $\mathbf{I} = \{i_1, i_2, \ldots, i_{|\mathbf{I}|}\}$. A container differs from an image only by the writable layer, so updating a container requires downloading the new version of the related image [25]. The layer set comprising each image is represented as $\mathbf{L} = \{l_1, l_2, \ldots, l_{|\mathbf{L}|}\}$, and d_l denotes the size of layer l.

Task: The tasks offloaded from different UEs to the edge node are grouped into the set $\mathbf{K} = \{k_1, k_2, \ldots, k_{|\mathbf{K}|}\}$. The resources that the task requires are assumed to be the same as those used by the container. At time t, task k requests u_k CPU resources and m_k memory resources. Additionally, d_k is the data size of task k, and t_k is its release time.

For applications sensitive to latency, minimizing task latency is a critical requirement. For example, autonomous vehicles require immediate access to perception, planning, and control containers to ensure safe navigation. Deploying all necessary containers on each edge node enables rapid access to services from nearby nodes and eliminates delays caused by transferring container images during emergency tasks. Similarly, edge AI servers deploy all containers, such as recognition, understanding, and processing modules, to minimize task latencies. While this design introduces additional storage overhead, it guarantees rapid access to critical functions. In EC, edge nodes are interconnected via a high-speed wired network to ensure reliable and low-latency layer sharing during container updates, which is common in EC deployments. In contrast, task offloading inherently involves wireless UEs (e.g., smartphones or IoT devices), necessitating a wireless uplink network [26]. The edge nodes are designed to process incoming tasks from various UEs quickly and efficiently through a wireless connection.

TABLE I
Summary of Key Notations

Section IV	
$\mathbf{N}$	Edge node set
$U_n(t)$	CPU cores of edge node n at time t
$M_n(t)$	Memory resource of edge node n at time t
U_n^I	Initial CPU cores of edge node n
M_n^I	Initial memory resource of edge node n
$D_n(t)$	Storage capacity of edge node n at time t
F_n	CPU frequency of edge node n
B_n	Bandwidth of edge node n
B_r	Bandwidth of the remote cloud
$B_{n,n'}$	Bandwidth between edge nodes n and n'
$w_n(t)$	load of the edge node n at time t
$\mathbf{K}$	Task set
u_k	CPU cores request of task k
m_k	Memory request of task k
f_k	CPU frequency request of task k
d_k	Size of task k
t_k^r	Release time of task k
t_k^f	Finish time of task k
o_k	Status time of task k
$\mathbf{C}$	Container set
$\mathbf{L}$	Layer set
d_l	Size of layer l
$s_{n,c}^{down}$	Download size
$T_{n,c}^{down}$	Download latency
$T_{n,c}^{inite}$	Initialization latency
$T_{n,c}^{update}$	Update latency
$T_{n,k}^{comm}$	Communication latency
$T_{n,k}^{comp}$	Computation latency
$T_{n,k}^{total}$	Total latency
t_c^s	Update start time of container c
t_c^f	Update finish time of container c
Section V	
s^t	State
ρ	Proportion of containers updated
$\mathbf{P}$	Update priority set
p_n	Update priority for node n
a_t	Action
r_t	Reward
R_t	Cumulative reward
$\mathcal{D}$	Replay buffer
$\mathcal{Q}_1$	Update sequence queue
$\mathcal{Q}_2$	Task scheduling queue
ϕ_i	Parameters of the Q-networks
$L(\phi_i)$	Loss functions of the Q-networks
π	Policy
θ	Parameters of policy network
$J(\theta)$	Loss function of the policy network

B. Container Update

Container updates involve three phases: downloading the new version of the image, pausing the old container, and initiating

the new one. Since the old container can be stopped while the new one starts, the process time is negligible.

Image download: Layers can be shared between various images, so only the changed layers in the new image need to be downloaded. The size of new layers necessary for updating container c in edge node n is:

$$s_{n,c}^{down} = \sum_{l \in \mathbf{L}} x_{c,l} \times (1 - y_{n,l}(t)) \times d_l, \tag{1}$$

where $x_{c,l}$ indicates whether container c contains layer l ($x_{c,l} = 1$) or not ($x_{c,l} = 0$). $y_{n,l}(t)$ represents the storage status of layer l on edge node n at time t (1 if stored, 0 otherwise).

An edge node updates a container by downloading the image from the remote cloud or loading it from other edge nodes via a distributed file system. Edge nodes dynamically select the fastest available source for downloading image layers. For each layer l, the node prioritizes the source with the highest bandwidth. If another edge node n' already stores layer l ($y_{n',l}(t) = 1$), the layer transferring between nodes n and n' is considered. Otherwise, the node downloads the layer from the cloud with bandwidth B_r. Download latency is the time required to transfer new image layers from neighboring edge nodes or the cloud. This collaboration minimizes download latency by leveraging distributed layer storage across the edge-cloud environment. The image download latency can be denoted as:

$$T_{n,c}^{down} = \sum_{l \in \mathbf{L}} \left(\frac{d_l \times x_{c,l}}{\max_{n' \in \mathbf{N}} \left(B_r, B_{n,n'} \times y_{n',l}(t) \right)} \right), \tag{2}$$

where B_r denotes the bandwidth of the remote cloud, and $B_{n,n'}$ signifies the bandwidth between edge nodes n and n'. The max operator is used to select the maximum bandwidth from the cloud B_r and other edge nodes $B_{n,n'} \times y_{n',l}(t)$. This ensures that each layer is transferred from the fastest available source, prioritizing edge nodes with existing layers to reduce reliance on the cloud.

Container initialization: While the container update process primarily consumes storage resources, there is an implicit demand on CPU, as the extraction of the new layers utilizes this resource [27]. For initialization latency, it describes the container startup time proportional to CPU frequency. Thus, container initialization latency can be obtained as:

$$T_{n,c}^{init} = \frac{\delta s_{n,c}^{down}}{F_n}, \tag{3}$$

where the CPU frequency of edge node n is F_n and δ is a constant. Therefore, the total update latency for container c on edge node n can be denoted as:

$$T_{n,c}^{update} = T_{n,c}^{down} + T_{n,c}^{init}. \tag{4}$$

Given that the update start time of container c on node n is $t_{n,c}^s$, the start time and finish time for the container update are:

$$t_c^s = \min_{n \in \mathbf{N}} t_{n,c}^s, \; t_c^f = \max_{n \in \mathbf{N}} \{ t_{n,c}^s + T_{n,c}^{update} \}, \tag{5}$$

where $\min_{n \in \mathcal{N}} t_{s_{n,c}}$ represents the earliest start time of the container update across all nodes, while $\max_{n \in \mathcal{N}} \{ t_{s_{n,c}} + T_{n,c}^{\text{update}} \}$ denotes the latest finish time. These operators capture the overall update time, ensuring the overall process completes only after all nodes finish their updates.

C. Task Scheduling and Computation

Computation-intensive tasks are offloaded by UEs to edge nodes for execution. The lifecycle of a task typically involves several stages: transferring task data (configuration files, input files, etc.) to the edge node over a wireless connection, executing the task in a container, and finally, the edge node returning the result to the user [28]. $\xi_{n,k}$, the uplink wireless transmission rate from task k to edge node n at time t, is denoted in [29] as:

$$\xi_{n,k}(t) = \frac{B_n}{Q_n(t)} log \left(1 + \frac{p_k h_{n,k}(t)}{\sigma^2} \right), \tag{6}$$

where Q_n denotes the number of tasks transmitted to edge node n at time t, with B_n representing the node's bandwidth. p_k corresponds to the transmission power, and the channel gain between UEs and edge node n at time t is $h_{n,k}(t) = d_{n,k}^{-\alpha}$ [30], where $d_{n,k}$ refers to the distance between them and α is the path loss coefficient. The power of Gaussian white noise is represented by σ. Communication latency corresponds to the time needed to transfer task data, determined by factors such as bandwidth contention and channel conditions. The communicating latency can be defined as:

$$T_{n,k}^{comm} = \int \frac{d_k}{\xi_{n,k}(t)} dt, \tag{7}$$

where d_k represents the data size required to execute the task. Generally, the latency of returning the result via communication is considered negligible and thus omitted [31], [32].

Tasks run concurrently within isolated containers. We adopt a weighted resource allocation approach, assigning computational resources to tasks in proportion to their CPU frequency requirements [33]. Computation latency reflects the processing time on edge nodes, which is inversely proportional to available computation resources and current workload intensity. The computation latency is defined as:

$$T_{n,k}^{comp} = \int \frac{f_k}{\frac{f_k}{w_n(t)+f_k} \times U_n} dt, \tag{8}$$

where f_k denotes the CPU frequency required by task k, while U_n represents the CPU cores available on edge node n. In addition, $w_n(t)$ is used to denote the load of edge node n at time t. The item $\frac{f_k}{w_n(t)+f_k}$ reflects the proportion of CPU resources allocated to task k. As $w_n(t)$ increases, it enables tasks to utilize a larger portion of the CPU cores U_n, thereby reducing the resource share available to each task and increasing latency. Conversely, under lighter loads, task k receives a larger resource fraction, minimizing latency. Edge nodes have finite CPU cores U_n, making proportional allocation critical for task scheduling. The model adapts to real-time workload changes, rendering it appropriate for edge environments where load balancing and task scheduling are closely linked.

Overall, the total task latency is:

$$T_k^{total} = T_{n,k}^{comm} + T_{n,k}^{comp}. \tag{9}$$

The release time of task k is denoted as t_k^r, so the finish time t_k^f can be denoted as:

$$t_k^f = T_k^{total} + t_k^r. \tag{10}$$

The start time of the update on edge node n is set to t_n^s, and the corresponding finish time is denoted as t_n^f. In real-world edge clusters, container updates induce the modification of the image file and the restart of the container. To ensure stability, containers undergoing updates are prohibited from executing new tasks. In this scenario, task execution and container update can be considered independent of each other. When a container has not yet initiated the update process, tasks can still be scheduled to it, meaning these tasks can execute on the container to be updated. However, once the container starts the update process, the update operation will terminate all ongoing tasks, causing them to fail. The status o_k of the task k can be represented as:

$$o_k = z_{n,k} \times \sum_{c \in \mathbf{C}} [\![t_k^s < t_{n,c}^s < t_k^f]\!], \tag{11}$$

where $[\![\cdot\cdot\cdot]\!]$ is Iverson bracket, which equals 1 if the condition is met; otherwise, it equals 0. $z_{n,k}$ denotes the scheduling status of task k on edge node n (1 for scheduled, 0 otherwise).

D. Problem Formulation and Analysis

Constraints: Each offloaded task requests a portion of resources, but available resources on an edge node are constrained. Exceeding the resource limits of an edge node can negatively impact container functionality. Therefore, it is essential to limit the total resource allocation to containers. The resource limits for the edge node are as follows:

$$\sum_{k \in \mathbf{K}} z_{n,k} \times u_k \le U_n(t), \sum_{k \in \mathbf{K}} z_{n,k} \times m_k \le M_n(t), \forall n, \forall t. \tag{12}$$

If no edge node possesses adequate resources to run a task, the task will be transferred to the remote cloud for execution, with its state o_k set to 0.

The storage capacity on an edge node for layers is limited and must not exceed the total available storage:

$$\sum_{l \in \mathbf{L}} y_{n,l}(t) \times d_l \le D_n(t) \ \ \forall n, \forall t. \tag{13}$$

As in previous studies [34], [35], tasks are indivisible and scheduled to a single edge node, represented as:

$$\sum_{n \in \mathbf{N} \cap \{n_{|\mathbf{E}|+1}\}} z_{n,k} = 1, \quad \forall k. \tag{14}$$

Problem formulation: To quantify the trade-offs between update efficiency and service quality, we define two optimization metrics: 1) Update cost $\mathcal{C}_\mathrm{u} = \sum_{c \in C}(t_f^c - t_s^c)$ is modeled as the total time spent on container updates, where t_f^c and t_s^c denote the finish and start timestamps of updating container c, respectively. This metric captures the cumulative time consumption induced by update operations; and 2) Scheduling cost $\mathcal{C}_\mathrm{s} = \sum_{k \in K} o_k$ measures service quality by penalizing task interruptions during updates. These metrics explicitly balance the trade-off between minimizing update time and maintaining task continuity. Our objective is to dynamically adjust the proportion and sequence of container updates, minimizing update time and reducing task interruptions due to updates. The weight λ balances container update and task scheduling costs. The problem is defined as:

$$\begin{aligned} \textit{Problem 1}. \quad & \min \mathcal{C} = \lambda \mathcal{C}_\mathrm{u} + (1-\lambda)\mathcal{C}_\mathrm{s}, \\ & s.t.\ Eqs.\ (12) - (14), \\ & x_{c,l} \in \{0,1\},\ \forall c \in \mathbf{C},\ \forall l \in \mathbf{L}, \\ & y_{n,l}(t) \in \{0,1\},\ \forall n \in \mathbf{N},\ \forall l \in \mathbf{L}, \\ & z_{n,k} \in \{0,1\},\ \forall n \in \mathbf{N},\ \forall k \in \mathbf{K}. \end{aligned}$$

Update cost is critical in resource-constrained edge environments, as prolonged updates strain bandwidth and delay service deployment. Scheduling cost aligns with the number of tasks interrupted due to container updates, where interruptions degrade user experience. As a complex variation of the bin-packing problem, traditional algorithms may not solve this problem efficiently within a reasonable time [36]. he task arrivals and container updates exhibit memoryless properties, which allows the problem to be modeled as a Markov Decision Process (MDP). RL algorithms can effectively address this complexity and yield improved solutions [37]. We can derive a value function to assess the expected cumulative reward by utilizing the collected states, actions, rewards, and a suitable discount factor. Therefore, the RL agent is able to refine its policy and make decisions prioritizing long-term objectives [11].

V. Proposed Algorithms

A. Algorithm Settings

This subsection introduces the LECU algorithm settings.

State: At time t, the state s_t comprises three components: the node state s_t^n, the container update state s_t^c, and the layer state s_t^l. Among these, the node state incorporates both the resource and transmission states. The resource state at time t contains available CPU, memory, and storage capacities, along with the CPU frequency, which is defined as:

$$\begin{aligned} s_t^{n,r} = & \{\mathbf{U}_n(t), \mathbf{M}_n(t), \mathbf{D}_n(t), \mathbf{F}\} \\ = & \{U_1(t), U_2(t), \ldots, U_{|\mathbf{N}|}(t), M_1(t), M_2(t), \ldots, M_{|\mathbf{N}|}(t), \\ & D_1(t), D_2(t), \ldots, D_{|\mathbf{N}|}(t), F_1, F_2, \ldots, F_{|\mathbf{N}|}\}. \end{aligned} \tag{15}$$

Nodes are interconnected through wired networks [32], with varying distances between edge nodes leading to different transmission rates. The transmission state $s_t^{n,b}$ is the transmission rate among edge nodes, which is represented as:

$$s_t^{n,b} = \begin{bmatrix} 0 & B_{1,2} & \cdots & B_{1,|\mathbf{N}|} \\ B_{2,1} & 0 & \cdots & B_{2,|\mathbf{N}|} \\ \vdots & \vdots & \ddots & \vdots \\ B_{|\mathbf{N}|,1} & B_{|\mathbf{N}|,2} & \cdots & 0 \end{bmatrix}. \tag{16}$$

The container update state includes the update start time and image ID of the container to be updated at the current time,

Algorithm 1: Proposed RBA Algorithm.

Input: $\mathbf{N}$, k,U_n^I,M_n^I,$U_n(t)$,$M_n(t)$
Output: $z_{n,k}$
1 **for** $n \in \mathbf{N}$ **do**
2 $\quad score \leftarrow \frac{U_n(t)}{U_n^I} + \frac{M_n(t)}{M_n^I}$;
3 **end for**
4 $\mathbf{N}_o \leftarrow$ sort $\mathbf{N}$ by score in the ascending order;
5 **for** $n \in \mathbf{N}_o$ **do**
6 $\quad$ **if** $u_k < U_n(t)$ *and* $m_k < M_n(t)$ **then**
7 $\quad\quad$ Assign k to n;
8 $\quad\quad$ $U_n(t) \leftarrow U_n(t) - u_k$;
9 $\quad\quad$ $M_n(t) \leftarrow M_n(t) - m_k$;
10 $\quad\quad$ Break;
11 $\quad$ **end if**
12 **end for**
13 **if** k *not assigned* **then**
14 $\quad$ Assign k to $n_{|\mathbf{N}|+1}$;
15 **end if**

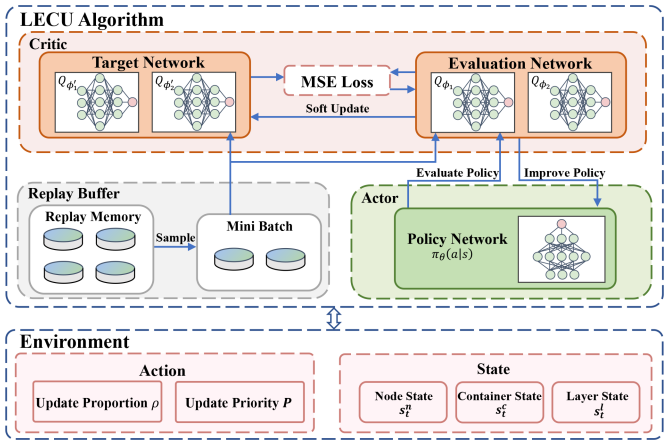


Fig. 4. Overview of LECU algorithm. The policy network generates container update decisions by embedding node states, layer distributions, and container versions. Dual critic networks evaluate actions using MSE loss, while a replay buffer stores transitions for batch training. Target networks enable stable policy updates through soft updates.

which can be defined as:

$$s_t^c = \{t_c^s, \mathrm{ID}_c\}. \tag{17}$$

The layer state includes the distribution of layers and the layers required for the currently updating container:

$$s_t^l = \{x_{c,1}, \ldots, x_{c,|\mathbf{L}|}, y_{n,1}(t), \ldots, y_{n,|\mathbf{L}|}(t)\}. \tag{18}$$

Thus, we define the state at time t as:

$$s_t = \{s_t^{n,r} \cup s_t^{n,b} \cup s_t^c \cup s_t^l\}. \tag{19}$$

Action: When container updates are required, the LECU algorithm determines both the proportion of containers to update simultaneously and the sequence of these updates. The action at time t is defined as:

$$a_t = \{\rho, \mathbf{P}\} \in \mathbf{A}, \tag{20}$$

where $\rho \in (0, 1]$ represents the proportion of containers updated simultaneously, and $\mathbf{P} = \{p_1, p_2, \ldots, p_{|\mathbf{N}|}\}$ is the updating priority of each container. Here, $p_n \in \mathbf{P}$ specifies the priority for the container on node n. If the container update proportion does not reach ρ, edge nodes with higher priorities are selected for updates.

Reward: Our goal is to minimize the duration of updates while reducing task interruptions caused by updates. At time t, the reward is given by:

$$r_t = -\lambda(t_c^f - t_c^s) - (1-\lambda) z_{n,k} \times [\![t_k^s < t_{n,c}^s < t_k^f]\!]. \tag{21}$$

The cumulative long-term reward can be written as $R_t = \sum_{t=0}^{T} \gamma^t r_t$, with γ representing the discount factor that ranges between 0 and 1.

B. Algorithm Design

Overview: The proposed container update framework in Fig. 2 shows that container updates occur on a large timescale, while task scheduling operates on a small timescale. Specifically, when a new container version is released, we invoke the LECU algorithm to determine the container upgrade sequence. During the task offloading process from UEs to edge nodes, the RBA algorithm is utilized to determine task scheduling. The details are as follows.

Algorithm 1 introduces the RBA algorithm, with inputs comprising the edge node set $\mathbf{N}$ and their associated resources. As depicted in Lines 1 to 3, each node is scored according to its remaining resources. Then, the edge nodes are sorted according to their scores, as shown in Line 4. In Lines 5 to 12, the algorithm verifies if each node has enough resources for task k. If so, the task is allocated to that node, and the loop ceases. In the end, if no edge node satisfies the scheduling criteria, the task is routed to the remote cloud, as shown in Lines 13 to 14.

Fig. 4 illustrates the framework of the LECU algorithm. It observes the states of node, container, and layer from the environment. These states are then embedded, concatenated, and input into the policy network, which makes the update decisions. Then, the reward is obtained from the action taken. As described in Algorithm 2, the update sequence queue $\mathcal{Q}_1$ and task scheduling queue $\mathcal{Q}_2$ are initialized. Queue $\mathcal{Q}_1$ contains all containers requiring updates, which are processed according to the update priority $\mathbf{P}$. Similarly, queue $\mathcal{Q}_2$ holds all tasks pending scheduling, which are processed in chronological order. As shown in Lines 2 to 6, when the count of containers undergoing updates is lower than the quantity specified by the algorithm, edge nodes are retrieved from $\mathcal{Q}_1$ to update the container. Then, as shown in Lines 7 to 12, tasks are taken from $\mathcal{Q}_2$. Should the current timestamp exceed the release time of task k, Algorithm 1 is called to schedule this task; otherwise, it is put back into $\mathcal{Q}_2$. Lastly, the LECU algorithm refreshes both the policy and target networks.

Training: Built upon a maximum entropy RL architecture, the LECU algorithm aims to identify the optimal policy that maximizes the expected long-term reward [38]. To reduce overestimation bias commonly present in Q-network-based methods, LECU employs two target networks.

Algorithm 2: Proposed LECU Algorithm.

Input: $\mathcal{Q}_1, \mathcal{Q}_2, a_t, \mathbf{N}$
Output: r_t
for $t \leftarrow 1, 2, \ldots$ **do**
  Get the number of containers being updated P;
  while $P < |\mathbf{N}| \times \rho$ **do**
    Get the first edge node n from $\mathcal{Q}_1$;
    Update the container on edge node n;
  end while
  Get the first task k from $\mathcal{Q}_2$;
  if $t_k > t$ **then**
    Call Algorithm 1 to schedule task k;
  else
    $\mathcal{Q}_2 \leftarrow put(k)$;
  end if
end for

Algorithm 3: Training of LECU Algorithm.

Input: s_0
Output: a_t
Initialize: $Q_{\phi_i}, Q'_{\phi'_i}$ for $i = 1, 2$, π_θ;
for *episode* $\leftarrow 1, 2, ..., M$ **do**
  for *step* $\leftarrow 1, 2, ..., N$ **do**
    Observe state s_t;
    Select action $a_t \sim \pi_\theta(s_t)$;
    Perform action a_t, then observe the reward r_t and the subsequent state s_{t+1};
    Call Algorithm 2 and store transition (s_t, a_t, r_t, s_{t+1}) in $\mathcal{D}$;
    if $|\mathcal{D}| > batch\_size$ **then**
      Sample a mini-batch of transitions from $\mathcal{D}$;
      Compute target value y by Eq. (23);
      Update Q_{ϕ_i} by Eq. (22);
      Update π_θ by Eq. (24);
      Softly update target networks by Eq. (25);
    end if
  end for
end for

To adjust the Q-network parameters ϕ_i, the LECU algorithm leverages the Mean Squared Error (MSE) loss [39] between the predicted Q-value $Q_{\phi_i}(s, a)$ and the calculated target value y. The Q-networks' loss functions are:

$$L(\phi_i) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{D}}[(Q_{\phi_i}(s, a) - y)^2], \tag{22}$$

where the expectation $\mathbb{E}_{(s,a,r,s')\sim\mathcal{D}}$ indicates the average value of samples. Furthermore, the target value y is given by:

$$y = r_t + \gamma \mathbb{E}_{a\sim\pi'_{\theta'}}[Q'_{\phi'_i}(s_{t+1}, a) - \alpha \log \pi'_{\theta'}(a|s_{t+1})], \tag{23}$$

where r_t denotes the instantaneous reward obtained after executing action a_t in state s_t, which transitions to state s_{t+1}. Then, the loss function for updating the policy network parameter θ is formulated as:

$$J(\theta) = \mathbb{E}_{s\sim\mathcal{D}, a\sim\pi_\theta}[Q_{\phi_i}(s, a) - \alpha \log \pi_\theta(a|s)]. \tag{24}$$

Instead of abruptly copying weights between the evaluation networks and target networks at discrete intervals, a weighted average is taken. The parameter τ enables the target networks to gradually track the evaluation networks. For the target networks, the soft update mechanism is formulated as:

$$\phi'_i \leftarrow \tau\phi_i + (1 - \tau)\phi'_i, \quad \theta' \leftarrow \tau\theta + (1 - \tau)\theta'. \tag{25}$$

Algorithm 3 presents the training procedure of the LECU algorithm. At the beginning of each episode, the replay buffer $\mathcal{D}$ is initialized. During each time step t, transitions are collected and stored in $\mathcal{D}$. As described in Lines 3 to 7, during step t, we first acquire the current state s_t, then select the action a_t based on the policy, and compute the reward r_t. Next, the subsequent state s_{t+1} is determined, and this transition is saved to the replay buffer $\mathcal{D}$. The training phase, elaborated in Lines 8–13, involves randomly sampling a batch of experience tuples from $\mathcal{D}$ and calculating the target Q-value for each tuple. Throughout training, the Q-network parameters ϕ_i and the policy network π_θ are updated. Lastly, the target networks undergo soft updates, and outputs are generated once all episodes are finished.

C. Computational Complexity Analysis

The LECU algorithm is composed of several modules. Their computational complexities are explored in the following analysis. First, (19) defines the state, which has a complexity of $O(|\mathbf{N}||\mathbf{L}|)$. Second, action selection involves determining the priority of container update of each node, leading to a complexity of $O(|\mathbf{N}|)$. For the reward, computation follows (21), and its complexity is $O(1)$, as it remains constant irrespective of edge node count. Finally, the node, container, and layer information is mapped through fully-connected layers. Letting L denote the number of hidden layers (each containing G neurons), the complexity of this part is $O(|\mathbf{N}||\mathbf{I}| \times G + L \times G^2)$.

The complexity of the RBA algorithm, which includes traversing each edge node and calculating the score, is $O(|\mathbf{N}|)$. Other operations, such as adding residual connections, performing normalization, and computing activation functions, generally have a much lower impact and thus can be considered negligible. Therefore, the total computational complexity is $O(|\mathbf{N}||\mathbf{I}| \times G + L \times G^2)$.

D. Training Cost Analysis

The training complexity of the LECU algorithm is dominated by policy updates. For each training step, the critic networks compute gradients over mini-batches with time complexity $O(B \times L \times G^2)$, where B is the batch size, L is the number of hidden layers, and G is the number of neurons per layer. Our experiments confirm convergence within 1500 episodes, ensuring practical feasibility. This training phase incurs computational costs related to GPU usage and time. During our experiments, the LECU algorithm is trained for about 1.5 hours on an NVIDIA RTX 4070 Super GPU. This training is conducted offline before deployment, ensuring that the algorithm operates efficiently during runtime.

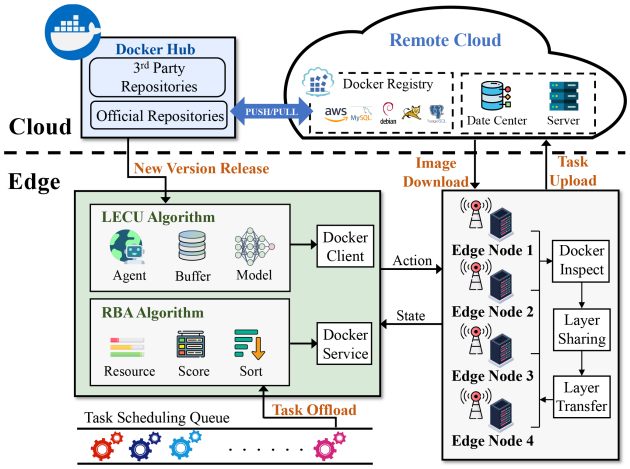


Fig. 5. Edge-cloud system implementation for container updates. A private Docker Registry hosts multi-version images, while edge nodes share layers via P2P protocols. The LECU agent is trained on GPUs to deploy dynamic update policies, and the RBA algorithm schedules tasks to balance resource utilization during updates.

VI. System Implementation

Overview: Fig. 5 illustrates our edge system, including four edge nodes and a cloud. We emulate a cloud on a server equipped with a 10-core Intel i9-10900 K 3.70 GHz CPU, 32 GB memory, and 1024 GB disk. Each edge node operates in separate Virtual Machines (VMs) with 4 cores, 8 GB memory, and 40 GB disk. Our algorithm is trained and executed on a workstation with the NVIDIA RTX 4070 Super GPU. We also develop a task generator for offloaded UE tasks and a version generator to release new image versions. Each time slot is 0.5 seconds, and the experiment consists of 1000 time slots.

Cloud: A private *Docker Registry* is deployed on the cloud. Typically, the required images are downloaded from the *Official Repositories* on *Docker Hub*. This approach may face major issues from network fluctuations, resulting in failed image downloads or incomplete updates. Therefore, a private *Docker Registry* is deployed. This registry hosts all the images used in our experiments. Edge nodes start the container update process when an updated container version is issued.

Edge: Containers on four edge nodes with *Docker* installed will be updated. To update a container, the process starts with *Docker Inspect* to retrieve all image layers. The edge node then uses the *Layer Sharing* mechanism to check and download only the layers that are not already available locally. However, efficient layer transmission in a distributed environment with varying network conditions is challenging. To address this, *Layer Sharing* utilizes a Peer-to-Peer (P2P) protocol, allowing edge nodes to communicate directly and create a decentralized network that improves stability and reliability. This approach is especially advantageous for the *Layer Transfer* process in edge-cloud collaboration, enabling more efficient downloading of layers from the remote cloud or loading from another edge node.

LECU algorithm comprises three components: `Agent`, `Buffer`, and `Model`. During training, the `Agent` acquires transitions from `Buffer` to train `Model`. During prediction, the `Agent` loads `Model` to determine updates and calls the *Docker Client* to download the new version of the image. However, ensuring the `Model` remains accurate in a changing environment is challenging. To address this, online learning and periodic retraining are implemented to keep the `Model` up-to-date with evolving conditions. Furthermore, the RBA algorithm receives tasks from the *Task Scheduling Queue*, evaluates edge node resources using *Score*, sorts edge nodes with *Order*, and schedules tasks to the highest-scoring edge node via *Docker Service*. If no edge node has sufficient resources, the task is uploaded to the *Cloud Server*.

TABLE II
Image and Layer Information

Type	Image	Layer
Number	23	609
Version	151	\
Max size	547.13 MB	406.78 MB
Min size	0.72 MB	0.03 KB
Average size	161.73 MB	30.02 MB

VII. Performance Evaluation

We analyze the LECU algorithm's effectiveness against various baselines in this section, utilizing larger-scale simulations and small-scale system implementations.

A. Dataset and Experiment Setup

Dataset: The container and layer data are crawled from Docker Hub [22], including 23 images and 609 layers. Each image has multiple versions, totaling 151 versions across all images. On average, images contain 4.03 layers each, with further statistical details in Table II. The task data originates from the Alibaba Cluster Trace [5], which is collected from a large production cluster. After preprocessing to remove missing and unreasonable values, 156,456 tasks remain, each averaging 3.93 CPU cores and 4.21 GB of memory, with randomly generated release times.

Parameter settings: We configure the noise power spectral density σ takes a value of -174 dBm/Hz, while the transmission power p as 23 dBm [40], [41]. Edge nodes exhibit transmission rates spanning 75 to 135 Mbps, and the bandwidth linking the remote cloud to edge nodes is set at 100 Mbps. For edge nodes, CPU capacity falls within 80–120 cores, with CPU frequencies ranging from 5 to 15 GHz [42]. Randomly generated tasks have sizes spanning 10 KB to 10 MB [21]. The weighting factor λ that balances update and scheduling costs is set to 0.6. The neural network input is normalized to a consistent scale. Table III lists the hyperparameters of the LECU algorithm.

Baselines: Several baselines are employed to compare the performance. The details are as follows.

1) *RU* [43]: The RU algorithm temporarily removes a subset of nodes from service to perform updates, then restores them to operation. Like Kubernetes default settings, the update proportion is 25% [44].

TABLE III
HYPERPARAMETER SETTINGS FOR THE LECU ALGORITHM

Type	Hyperparameter	Value
Actor	Learning rate	1e-4
	Hidden layers	2 Full connection (512,128)
Critic	Learning rate	3e-4
	Hidden layers	2 Full connection (512,128)
	Loss Function	MSE Loss
Other	Discount factor γ	0.99
	Activation function	ReLU
	Optimizer	Adam
	Soft update coefficient τ	5e-3
	Entropy parameter α	3e-4

2) *RULS* [43]: The RU algorithm incorporates layer sharing, which significantly reduces the download size of container updates.
3) *LS* [18]: The Local Search (LS) algorithm is a heuristic method that optimizes software updates for smart vehicles, focusing on update time-aware strategies.
4) *FB* [10]: The Fog Computing Based Update (FB) algorithm is designed to conserve computation resources through a resource-aware update strategy.

RU is widely adopted in production systems, making it a practical benchmark for real-world applicability. RU can be configured to specify the number of containers that can be updated at any time during the update process. Furthermore, RULS enhances RU by incorporating layer sharing, a key technology in our work, to reduce layer transmission. In addition to the rule-based or heuristic-based baselines, we also compare the LECU algorithm with several State-of-the-Art (SOTA) DRL algorithms. The details are as follows.

1) *PPO* [45]: Proximal Policy Optimization (PPO) is an on-policy RL algorithm that limits policy updates via a clipped objective function to ensure stable training.
2) *MPO* [46]: Maximum a Posteriori Policy Optimization (MPO) combines policy optimization with probabilistic inference, using entropy regularization and KL constraints for efficient exploration.
3) *DDPG* [47]: Deep Deterministic Policy Gradient (DDPG) is a model-free algorithm for continuous control, employing deterministic policies and target networks for stability.
4) *TD3* [48]: Twin Delayed DDPG (TD3) is an enhanced DDPG variant addressing overestimation bias via twin critics, delayed policy updates, and target smoothing.

Detailed comparison results with DRL algorithms in Fig. 10.

Evaluation metrics: To assess the performance of the LECU algorithm, we focus on the following evaluation metrics.

1) *Update cost:* It is defined as the total time spent on container updates across all nodes, capturing the cumulative latency caused by image downloads and container initialization.
2) *Scheduling cost:* It measures the number of tasks interrupted by container updates, emphasizing service continuity and user experience in latency-sensitive EC.
3) *Total cost:* It is the weighted sum of update cost and scheduling cost.
4) *Task latency:* It includes communicating latency and computation latency.
5) *Task interruption:* It indicates a task is interrupted if its execution window overlaps with a container update on its scheduled node.

B. Simulation Results

Performance with different numbers of edge nodes: The average total cost, comprising update cost, scheduling cost, and total cost, is shown in Fig. 6. Specifically, Fig. 6(b) and (c) illustrate the update and scheduling costs, respectively. For all algorithms, the update cost increases with the growing number of edge nodes, as more nodes result in more containers requiring updates. The LECU algorithm mitigates this increase more effectively by focusing on downloading only the necessary layers. The scheduling cost in our objective function quantifies the number of tasks interrupted by updates. To minimize this, the RBA algorithm proactively schedules tasks to edge nodes not to be updated, reducing the likelihood of future interruptions.

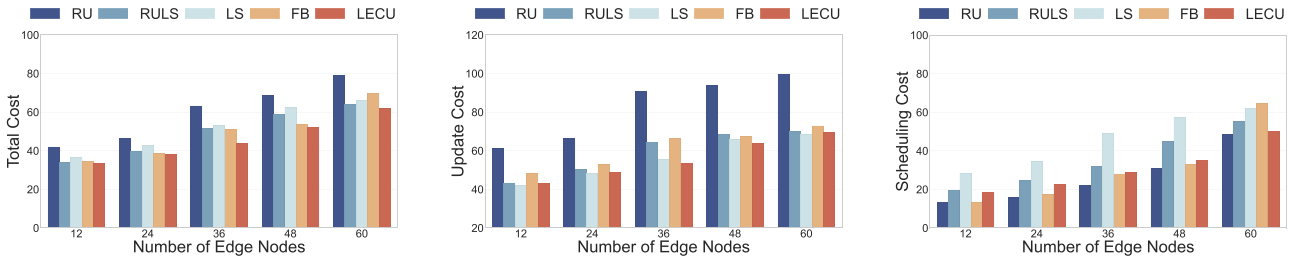
The total cost associated with container updates is illustrated in Fig. 6(a). The total cost ranking is LECU $<$ FB $<$ RULS $<$ LS $<$ RU as the number of edge nodes increases. Specifically, the LECU algorithm reduces the average total cost by 13%, 5%, 8%, and 30% compared to the FB, RULS, LS, and RU algorithms, respectively. LECU algorithm outperforms baselines across various numbers of edge nodes.

Performance with different CPU frequency: Fig. 7 shows the total cost as the CPU frequency of edge nodes varies. The results show that total costs decrease as CPU frequency increases due to faster task execution, reducing the likelihood of disruption by container updates. LECU algorithm consistently outperforms others in total cost as the CPU frequency of edge nodes varies, in the order: LECU $<$ LS $<$ RULS $<$ FB $<$ RU. In particular, compared to RU, RULS, LS, and FB algorithms, the LECU algorithm reduces total cost by 33%, 8%, 5%, and 11%, respectively.

Performance with different bandwidth: A decline in total cost with increasing bandwidth is illustrated in Fig. 8. The primary reason for this reduction is the decrease in update latency, which is directly affected by bandwidth. The time required to download container layers is significantly reduced as bandwidth increases. Overall, the LECU algorithm outperforms all other algorithms in minimizing total cost. In particular, compared to the baseline algorithms, the LECU algorithm reduces total cost by approximately 16%.

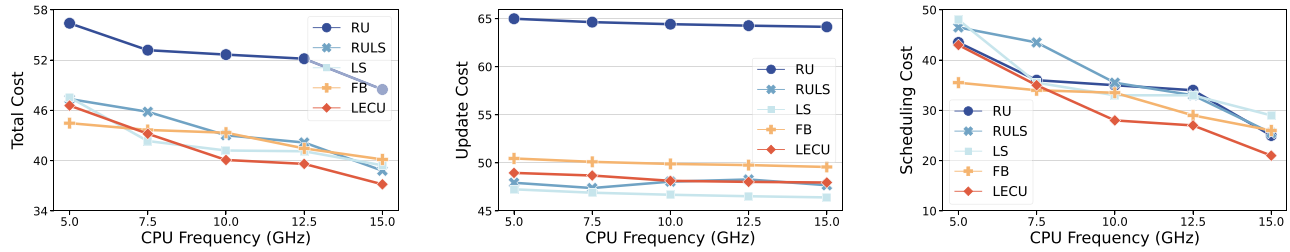
Convergence of LECU algorithm: The training process of the LECU algorithm is shown in Fig. 9. With an increase in training steps, the losses of both the policy network and the target Q-network decrease rapidly before stabilizing. Fig. 9(a) illustrates the reward, which initially rises sharply before leveling off. The algorithm identifies a promising policy and fine-tunes the policy network. Temporary declines in rewards during training may occur when exploring new environments, which can make the current policy less effective. However, the algorithm rapidly readjusts, restoring normal functionality over time.

Fig. 9(b) illustrates the policy network loss, which initially decreases before rising again. This fluctuation is due to the



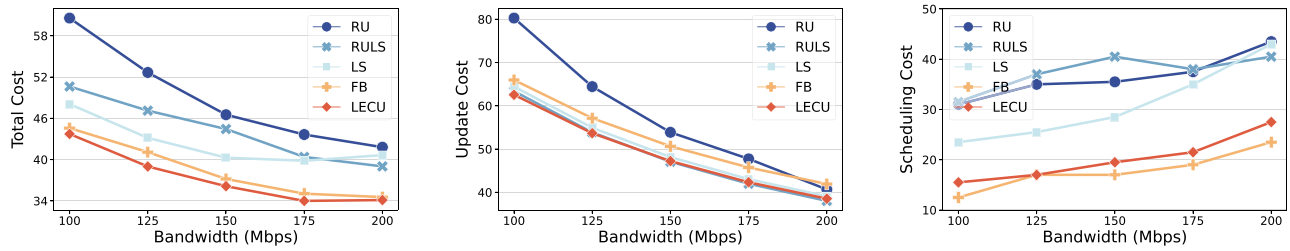
(a) Total Cost (b) Update Cost (c) Scheduling Cost

Fig. 6. Performance with different numbers of edge nodes.



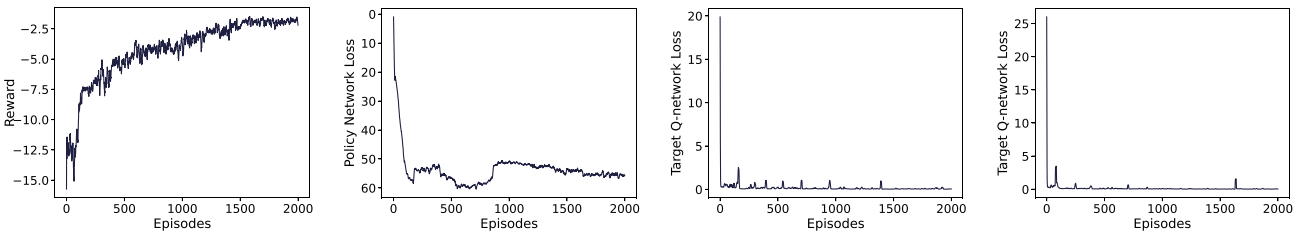
(a) Total Cost (b) Update Cost (c) Scheduling Cost

Fig. 7. Performance with different CPU frequency.



(a) Total Cost (b) Update Cost (c) Scheduling Cost

Fig. 8. Performance with different bandwidths.



(a) Reward (b) Policy Network π_θ Loss (c) Target Q-network ϕ_1 Loss (d) Target Q-network ϕ_2 Loss

Fig. 9. Reward, Policy network Loss, and Target Q-network Loss of LECU algorithm.

network producing relatively random policies early in training. As training progresses, it learns more rational policies and stabilizes around a specific value. Fig. 9(c) and (d) depict the losses of the target Q-networks. As the Q-network learns and its weights are updated based on state-action-reward transitions, the discrepancy between the evaluation and target networks decreases over time. This convergence leads to a reduction in the target Q-network loss. Overall, the LECU algorithm converges rapidly, demonstrating its ability to learn optimal policies in a short time.

Performance against different DRL algorithms: LECU algorithm is based on DRL, and we evaluate its performance against other DRL algorithms, including PPO [45], MPO [46], DDPG [47], and TD3 [48]. As shown in Fig. 10, the results demonstrate that LECU reduces the total cost by 20% over baselines, due to its layer-aware update mechanism and

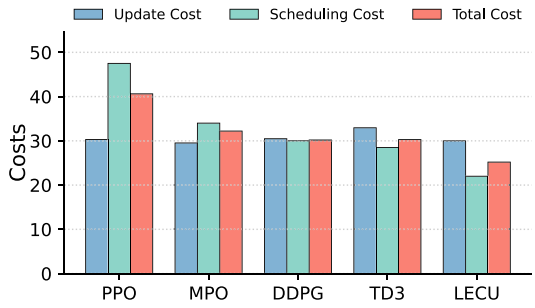


Fig. 10. Compared with different DRL algorithms.

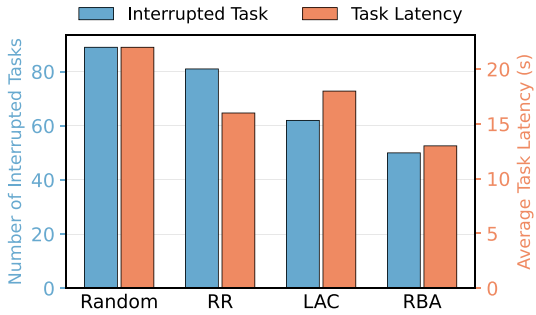
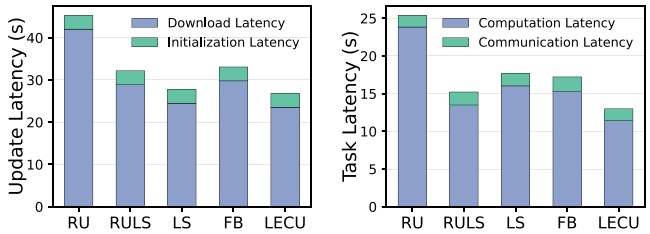


Fig. 11. Comparison with different task scheduling algorithms.



(a) Update Latency (b) Task Latency

Fig. 12. Latency performance of different algorithms.

edge-cloud collaboration. The LECU algorithm combines the RBA algorithm to achieve a lower total cost than these algorithms, further demonstrating their superiority in container updates.

Performance against different task scheduling algorithms: In Fig. 11, we compare the number of interrupted tasks and average task latency under four scheduling algorithms: Random, Round-Robin (RR), Least Active Calls (LAC), and our proposed RBA algorithm. The results demonstrate that RBA achieves the lowest number of interrupted tasks and the shortest average task latency. The performance gap highlights the ability of the RBA algorithm to balance resource utilization across edge nodes while prioritizing task continuity, making it particularly suitable for dynamic update scenarios.

Latency performance analysis: Update latency refers to the total time required to complete container updates, which includes download and initialization latencies. We measure both download and initialization latencies. Fig. 12(a) compares these components across algorithms, showing LECU reduces update latency by 40% compared to RU, due to layer sharing and dynamic update decision. Task latency denotes the total duration

TABLE IV
COMPUTATION RESOURCES OF DIFFERENT ALGORITHMS

Algorithm	RAM	VRAM	Execution Time
RU	-	-	2.20 ms
RULS	-	-	2.37 ms
LS	-	-	28.66 ms
FB	-	-	2.85 ms
PPO	219.87 Kb	4460.00 Kb	16.49 ms
MPO	658.71 Kb	3292.50 Kb	45.84 ms
DDPG	73.14 Kb	3292.50 Kb	14.10 ms
TD3	73.14 Kb	2268.50 Kb	8.75 ms
LECU	73.14 Kb	5340.50 Kb	20.38 ms

required to execute a task in edge nodes, including communication and computation latencies. We evaluate task latency as illustrated in Fig. 12(b). LECU achieves a 25% reduction in average task latency compared to the FB algorithm. This improvement is attributed to its RBA algorithm, which prioritizes scheduling tasks to edge nodes with sufficient computational resources.

Computation resources across diverse algorithms: The Random Access Memory (RAM), Video RAM (VRAM), and execution duration are presented in Table IV using *torch.profiler* [49]. The RU and RULS algorithms demonstrate the shortest runtime due to their reliance on straightforward decision-making and sorting processes. LECU achieves performance with 20.38 ms execution time, 73.14 Kb RAM, and 5340.50 Kb VRAM, which are acceptable for resource-constrained edge nodes. Moreover, the average container update interval in real-world trace is over 43.2 s [5], which vastly exceeds LECU runtime, demonstrating that our algorithm can be run in real-time.

C. System Results

We conduct extensive simulations to validate our LECU algorithm's effectiveness, demonstrating its strong performance under ideal conditions. However, theoretical validation alone doesn't fully capture its real-world behavior. Therefore, we perform additional experiments on our deployed edge system to evaluate the algorithm's feasibility and effectiveness in practical applications. For the real system experiments, we deploy 4 edge nodes with 4 cores CPU and 8 GB memory. The experiments are conducted within a single update cycle and involved generating 100 tasks with randomized release times distributed across 1000 time slots. The transmission rate between edge nodes varies between 50 and 100 Mbps, while the bandwidth connecting edge nodes and the remote cloud is capped at 80 Mbps. The following experiments are conducted on the implemented system.

In Fig. 13, we calculate the total cost incurred by various algorithms over a single update cycle. The experimental results closely align with those obtained in the simulation experiments. In this figure, the total cost of container updates for the LECU algorithm in the real system is relatively low. These findings further validate our prior conclusions.

Moreover, the update costs from various container updates are quantified in the system. The Cumulative Distribution Function (CDF) of update cost is demonstrated in Fig. 14. The LECU algorithm has a higher proportion of low update costs than other

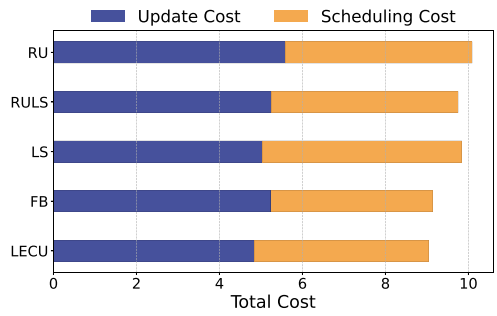


Fig. 13. System performance.

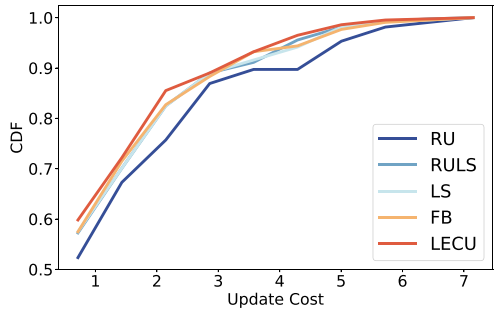


Fig. 14. CDF of the update cost.

algorithms. These results enhance the potential of our LECU algorithm as a promising solution for container updates.

VIII. Discussion

A. Feasibility of Real-Time Running

The LECU algorithm is an online optimization framework designed for dynamic edge environments. It processes container update and task scheduling in real-time through a two-timescale architecture, making immediate decisions based on current layer distributions and node resource states. Prior to actual deployment, LECU undergoes training with historical traces of container updates and edge node load variations. This initial training phase enables the RL model to learn effective update strategies, significantly reducing decision latency during actual operation. During runtime, LECU uses its pre-trained model with real-time environmental interaction. The online nature of our algorithm, combined with its initial training phase, ensures that it can handle the randomness of container updates well.

In summary, the LECU algorithm operates effectively in real-time scenarios. It utilizes historical data for initial training and continuously adapts to edge-cloud environments via dynamic interactions, thereby maintaining real-time responsiveness.

B. Applicability to Other Resource Allocation Problems

The components of the LECU algorithm, layer-aware resource optimization, RL-based decision-making, and edge-cloud collaboration, are broadly applicable to other resource allocation problems. For example, microservices in EC often share dependencies. Similar to container layer reuse, edge nodes can load shared dependencies from neighboring nodes, reducing deployment latency. The SAC-based model in LECU can adapt to other sequential decision-making scenarios, such as task offloading or service migration. The edge-cloud collaboration mechanism, which prioritizes local layer transfers over remote cloud downloads, can also enhance cost efficiency in distributed caching scenarios.

C. Cost-Effective Superiority of LECU

The LECU algorithm demonstrates superior cost-effectiveness in container updates by integrating dynamic update decisions, resource-balanced task scheduling, efficient layer sharing, and edge-cloud collaboration. This approach significantly enhances container update efficiency in EC while reducing associated costs. The details are as follows.

1) *Dynamic adjustment of update decision:* The LECU algorithm dynamically adjusts the update proportion and sequence based on real-time node load assessments. Under low-load conditions, multiple containers are updated concurrently, significantly reducing total update time. By balancing update latency with service continuity, LECU minimizes the cumulative cost of updates over time.
2) *Resource-balanced task scheduling:* Meanwhile, the RBA algorithm supports this process through resource-balanced task scheduling. It schedules tasks to nodes with the highest residual resource scores during updates, avoiding nodes under update. This approach reduces task interruptions, directly lowering service disruption costs.
3) *Benefits of layer sharing:* Furthermore, LECU substantially reduces update costs by leveraging layer sharing. When updating containers that share layers with others, only modified layers require downloading.
4) *Edge-cloud collaboration:* Edge-cloud collaboration optimally distributes image layers across available nodes. For locally unavailable layers, the algorithm prioritizes transfers from neighboring nodes over the remote cloud, mitigating cloud burden and reducing latency.

IX. Conclusion

This paper formulated the container update problem in edge-cloud networks with the objective of minimizing update and scheduling costs. Specifically, we investigated layer sharing and edge-cloud collaboration to accelerate the update process. We proposed the LECU algorithm, built upon the SAC RL framework, alongside a task scheduling algorithm designed to reduce task interruptions caused by updates. A real edge system was implemented to validate the performance of these algorithms, and large-scale simulations were conducted to assess their scalability. Both system implementation and simulation outcomes confirmed that the LECU algorithm surpassed current methods in update and scheduling cost efficiency. Future work will address update challenges related to advanced technologies such as large language models and digital twins, which entail larger-scale, more frequent updates and increased resource constraints.

REFERENCES

[1] L. Nkenyereye, K.-J. Baeg, and W.-Y. Chung, "Deep reinforcement learning for containerized edge intelligence inference request processing in IoT edge computing," *IEEE Trans. Services Comput.*, vol. 16, no. 6, pp. 4328–4344, Nov./Dec. 2023.

[2] N. Zhao et al., "Large-scale analysis of docker images and performance implications for container storage systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 4, pp. 918–930, Apr. 2021.

[3] Y. Chang et al., "A survey on evaluation of large language models," *ACM Trans. Intell. Syst. Technol.*, vol. 15, no. 3, pp. 1–45, 2024.

[4] Z. Tang, F. Mou, J. Lou, W. Jia, Y. Wu, and W. Zhao, "Joint resource overbooking and container scheduling in edge computing," *IEEE Trans. Mobile Comput.*, vol. 23, no. 12, pp. 10903–10917, Dec. 2024.

[5] Alibaba cluster trace program. 2017. [Online]. Available: https://github.com/alibaba/clusterdata/

[6] Q. Hua, D. Yang, S. Qian, J. Cao, G. Xue, and M. Li, "Humas: A heterogeneity- and upgrade-aware microservice auto-scaling framework in large-scale data centers," *IEEE Trans. Comput.*, vol. 74, no. 3, pp. 968–982, Mar. 2025.

[7] A. Buzachis, A. Galletta, A. Celesti, L. Carnevale, and M. Villari, "Towards osmotic computing: A blue-green strategy for the fast re-deployment of microservices," in *Proc. 24th IEEE Symp. Comput. Commun.*, 2019, pp. 1–6.

[8] D. Sun, A. Fekete, V. Gramoli, G. Li, X. Xu, and L. Zhu, "R2C: Robust rolling-upgrade in clouds," *IEEE Trans. Dependable Secure Comput.*, vol. 15, no. 5, pp. 811–823, Sep./Oct. 2018.

[9] H. Zhang et al., "An optimal container update method for edge-cloud collaboration," *Softw.: Pract. Exp.*, vol. 54, no. 4, pp. 617–634, 2024.

[10] M. Al Maruf, A. Singh, A. Azim, and N. Auluck, "Faster fog computing based over-the-air vehicular updates: A transfer learning approach," *IEEE Trans. Services Comput.*, vol. 15, no. 6, pp. 3245–3259, Nov./Dec. 2022.

[11] Z. Tang, F. Mou, J. Lou, W. Jia, Y. Wu, and W. Zhao, "Multi-user layer-aware online container migration in edge-assisted vehicular networks," *IEEE/ACM Trans. Netw.*, vol. 32, no. 2, pp. 1807–1822, Apr. 2024.

[12] L. Hu, A. Liu, M. Xie, and T. Wang, "UAVs joint vehicles as data mules for fast codes dissemination for edge networking in smart city," *Peer-to-Peer Netw. Appl.*, vol. 12, pp. 1550–1574, 2019.

[13] C. Zheng et al., "Wharf: Sharing docker images in a distributed file system," in *Proc. 18th ACM Symp. Cloud Comput.*, 2018, pp. 174–185.

[14] L. Gu, J. Huang, S. Huang, D. Zeng, B. Li, and H. Jin, "LOPO: An out-of-order layer pulling orchestration strategy for fast microservice startup," in *Proc. IEEE Conf. Comput. Commun.*, 2023, pp. 1–9.

[15] Y. Shi, Y. Yang, C. Yi, B. Chen, and J. Cai, "Toward online reliability-enhanced microservice deployment with layer sharing in edge computing," *IEEE Internet Things J.*, vol. 11, no. 13, pp. 23370–23383, Jul. 2024.

[16] T. Wang et al., "Propagation modeling and defending of a mobile sensor worm in wireless sensor and actuator networks," *Sensors*, vol. 17, no. 1, pp. 1–17.

[17] D. Zeng, H. Geng, L. Gu, and Z. Li, "Layered structure aware dependent microservice placement toward cost efficient edge clouds," in *Proc. IEEE Conf. Comput. Commun.*, 2023, pp. 1–9.

[18] M. Y. Hassan, S. Choudhury, and Z. M. Fadlullah, "On optimal scheduling of OTA software updates for smart vehicles leveraging fog computing," in *Proc. 17th Int. Wireless Commun. Mobile Comput.*, 2021, pp. 2044–2049.

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

[20] R. S. Sutton and A. G. Barto, "Reinforcement learning: An introduction," *Robotica*, vol. 17, no. 2, pp. 229–235, 1999.

[21] 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 *Proc. 12th IEEE/CIC Int. Conf. Commun. China*, 2023, pp. 1–6.

[22] Docker hub. 2024. [Online]. Available: https://hub.docker.com/

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

[24] J. Lou, H. Luo, Z. Tang, W. Jia, and W. Zhao, "Efficient container assignment and layer sequencing in edge computing," *IEEE Trans. Services Comput.*, vol. 16, no. 2, pp. 1118–1131, Mar./Apr. 2023.

[25] I. Miell and A. Sayers, *Docker in Practice*. New York, NY, USA: Simon and Schuster, 2019.

[26] Z. Chen, H. H. Yang, and T. Q. Quek, "Edge intelligence over the air: Two faces of interference in federated learning," *IEEE Commun. Mag.*, vol. 61, no. 12, pp. 62–68, Dec. 2023.

[27] S. Wu, C. Niu, J. Rao, H. Jin, and X. Dai, "Container-based cloud platform for mobile computation offloading," in *Proc. 31st IEEE Int. Parallel Distrib. Process. Symp.*, 2017, pp. 123–132.

[28] Y. Liang et al., "Collaborative edge server placement for maximizing QoS with distributed data cleaning," *IEEE Trans. Services Comput.*, vol. 18, no. 3, pp. 1321–1335, May/Jun. 2025.

[29] K. Qu, W. Zhuang, Q. Ye, W. Wu, and X. Shen, "Model-assisted learning for adaptive cooperative perception of connected autonomous vehicles," *IEEE Trans. Wireless Commun.*, vol. 23, no. 8, pp. 8820–8835, Aug. 2024.

[30] A. Goldsmith, *Wireless Communications*. Cambridge, U.K.: Cambridge Univ. Press, 2005.

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

[32] H. Cui, Z. Tang, J. Lou, W. Jia, and W. Zhao, "Latency-aware container scheduling in edge cluster upgrades: A deep reinforcement learning approach," *IEEE Trans. Services Comput.*, vol. 17, no. 5, pp. 2530–2543, Sep./Oct. 2024, doi: 10.1109/TSC.2024.3394689.

[33] J. Wang, J. Hu, G. Min, Q. Ni, and T. El-Ghazawi, "Online service migration in mobile edge with incomplete system information: A deep recurrent actor-critic learning approach," *IEEE Trans. Mobile Comput.*, vol. 22, no. 11, pp. 6663–6675, Nov. 2023.

[34] 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 Trans. Parallel Distrib. Syst.*, vol. 31, no. 6, pp. 1357–1371, Jun. 2020.

[35] 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 Trans. Mobile Comput.*, vol. 22, no. 4, pp. 2147–2162, Apr. 2023.

[36] D. S. Hochba, "Approximation algorithms for NP-hard problems," *ACM SIGACT News*, vol. 28, pp. 40–52, 1997.

[37] V. Mnih et al., "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[38] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," in *Proc. 37th Int. Conf. Mach. Learn.*, PMLR, 2018, pp. 1861–1870.

[39] O. Köksoy, "Multiresponse robust design: Mean square error (MSE) criterion," *Appl. Math. Comput.*, vol. 175, no. 2, pp. 1716–1729, 2006.

[40] X. Chen, "Decentralized computation offloading game for mobile cloud computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 4, pp. 974–983, Apr. 2015.

[41] 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 Trans. Veh. Technol.*, vol. 68, no. 3, pp. 2763–2776, Mar. 2019.

[42] Y. Chen, Y. Sun, C. Wang, and T. Taleb, "Dynamic task allocation and service migration in edge-cloud IoT system based on deep reinforcement learning," *IEEE Internet Things J.*, vol. 9, no. 18, pp. 16742–16757, Sep. 2022.

[43] Performing a rolling update. 2024. [Online]. Available: https://kubernetes.io/docs/tutorials/kubernetes-basics/update/update-intro/

[44] Kubernetes. 2024. [Online]. Available: https://kubernetes.io/

[45] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," 2017, *arXiv: 1707.06347*.

[46] A. Abdolmaleki et al., "Maximum a posteriori policy optimisation," 2018, *arXiv: 1806.06920*.

[47] T. P. Lillicrap et al., "Continuous control with deep reinforcement learning," 2015, *arXiv:1509.02971*.

[48] S. Fujimoto, H. Hoof, and D. Meger, "Addressing function approximation error in actor-critic methods," in *Proc. Int. Conf. Mach. Learn.*, PMLR, 2018, pp. 1587–1596.

[49] PyTorch documentation. 2024. [Online]. Available: https://pytorch.org/docs/

Hanshuai Cui received the BS degree from the School of Information Science and Engineering, Qufu Normal University, China, in 2020. He is currently working toward the PhD degree with the School of Artificial Intelligence, Beijing Normal University, China. His current research interests include mobile edge computing, resource allocation, and reinforcement learning.

Zhiqing Tang (Member, IEEE) received the BS degree from the School of Communication and Information Engineering, University of Electronic Science and Technology of China, China, in 2015, and the PhD degree from the 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.

Yuan Wu (Senior Member, IEEE) received the PhD degree in electronic and computer engineering from the Hong Kong University of Science and Technology, Hong Kong, in 2010. He is currently an associate professor with the State Key Laboratory of Internet of Things for Smart City, University of Macau, Macau SAR, China, and also with the Department of Computer and Information Science, University of Macau. His research interests include mobile edge computing and edge intelligence, and integrated sensing and communications. He was the recipient of the Best Paper Award from the IEEE ICC'2016, IEEE TCGCC'2017, IWCMC'2021, and IEEE WCNC'2023. He is on the editorial board of *IEEE Transactions on Wireless Communications*, *IEEE Transactions on Vehicular Technology*, and *IEEE Transactions on Network Science and Engineering*. He is the distinguished lecturer of IEEE Vehicular Technology Society (2025–2027).

Weijia Jia (Fellow, IEEE) received the BSc and MSc degrees from Center South University, China, in 1982 and 1984, and the master of applied sci. and PhD degrees from Polytechnic Faculty of Mons, Belgium, in 1992 and 1993, respectively, all in computer science. He 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 BNUHKBU United International College (UIC) and has been the Zhiyuan chair professor of Shanghai Jiao Tong University, China. He was the chair professor and the deputy director of the State Kay Laboratory of Internet of Things for Smart City, University of Macau. From 1993–1995, he joined German National Research Center for Information Science (GMD) in Bonn (St. Augustine) as a research fellow. From 1995–2013, he worked with the 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 more than 600 publications in the prestige international journals/conferences and research books, and book chapters. He has received the best product awards from the International Science & Tech. Expo (Shenzhen), in 2011/2012 and the 1st Prize of Scientific Research Awards from the Ministry of Education of China, in 2017 (list 2). 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 distinguished member of CCF.