Layer Reused Collaborative Scheduling for Container-dependent Tasks in Industrial Real-time Systems

HAOTONG ZHANG, South China University of Technology, Guangzhou, China
WEIWEI LIN, South China University of Technology, Guangzhou, China and Pengcheng Laboratory, Shenzhen, China
YUEBIN HUANG, South China University of Technology, Guangzhou, China
HAIJIE WU, South China University of Technology, Guangzhou, China
CHENJIETING WU, South China University of Technology, Guangzhou, China
KEQIN LI, Department of Computer Science, State University of New York at New Paltz, New Paltz, United States

Renowned for their lightweight and portability, containers are increasingly deployed in edge computing to deliver low-latency and privacy-preserving computing services for industrial real-time systems. While layer reuse can potentially improve the execution efficiency of container-dependent tasks, the existing work suffers from two issues: (1) inefficient disk space utilization, which compromises image pull efficiency and reuse utility, and (2) inadequate consideration of resource capacity, leading to practical limitations. This article addresses the container-dependent task scheduling problem in edge computing-assisted industrial real-time systems by introducing a novel Layer Reuse-based Collaborative Scheduling (LRCS) framework. First, to ensure practical applicability, we comprehensively consider resource capacities and the cost terms of the task completion time. Second, a framework that collaborates scheduling and execution nodes in a weakly coupled manner is proposed. At the execution nodes, LRCS maximizes disk utilization by evicting layers only when disk space is insufficient. The eviction is modeled as a 0-1 knapsack problem, where dynamic layer value assessment enhances the reuse rate. At the scheduling node, LRCS formulates container-dependent task scheduling as a multidimensional online bin-packing problem. A value-based learning algorithm optimizes long-term processing efficiency by adjusting the scheduling order of online tasks. Experimental validation on a testbed using real device data and comparisons with state-of-the-art layer reuse-based algorithms demonstrate that the proposed algorithm outperforms all baselines.

CCS Concepts: • **Computer systems organization → Real-time system architecture**; • **Theory of computation → Models of computation**; • **Computing methodologies → Distributed algorithms;**

This work is supported by Shandong Provincial Natural Science Foundation Project (ZR2024LZH012), Guangxi Key Research and Development Project(2024AB02018), Guangdong Provincial Natural Science Foundation Project (2025A1515010113) and the Major Key Project of PCL, China under Grant PCL2025A11.

Authors' Contact Information: Haotong Zhang, South China University of Technology, Guangzhou, China; e-mail: sezhanght@mail.scut.edu.cn; Weiwei Lin (corresponding author), South China University of Technology, Guangzhou, Guangdong, China and Pengcheng Laboratory, Shenzhen, Guangdong, China; e-mail: linww@scut.edu.cn; Yuebin Huang, South China University of Technology, Guangzhou, China; e-mail: 202030430097@mail.scut.edu.cn; Haijie Wu, South China University of Technology, Guangzhou, Guangdong, China; e-mail: 202030442496@mail.scut.edu.cn; Chenjieting Wu, South China University of Technology, Guangzhou, Guangdong, China; e-mail: 202030442489@mail.scut.edu.cn; Keqin Li, Department of Computer Science, State University of New York at New Paltz, New Paltz, New York, United States; e-mail: lik@newpaltz.edu.

ACM 1539-9087/2026/09-ART83
https://doi.org/10.1145/3779131

Additional Key Words and Phrases: Industrial real-time systems, container-dependent tasks, collaborative scheduling, layer reuse

ACM Reference Format:
Haotong Zhang, Weiwei Lin, Yuebin Huang, Haijie Wu, Chenjieting Wu, and Keqin Li. 2026. Layer Reused Collaborative Scheduling for Container-dependent Tasks in Industrial Real-time Systems. *ACM Trans. Embedd. Comput. Syst.* 25, 5, Article 83 (September 2026), 25 pages. https://doi.org/10.1145/3779131

1 Introduction

With the rapid development of IoT and automation technologies, the data generated by industrial real-time systems has surged. This has led to higher demands for processing efficiency and data security, rendering traditional centralized computing paradigms inadequate. Edge computing, which processes and stores data closer to the edge of the network, seamlessly integrates with industrial real-time systems [1]. It not only leverages the vast array of devices in industrial environments but also offers significant advantages in low latency [2] and privacy protection [3]. Edge computing has been deeply merging with industrial real-time systems in terms of machinery health [4], rapid decision-making [5], real-time interactions [6], and product quality [7].

Container technology has been widely adopted in edge computing to shield edge devices from hardware and software heterogeneity [8–11]. Benefiting from its lightweight, isolation, and portability, containers enable edge devices to serve as a unified infrastructure, delivering flexible containerized services for various tasks. As a result, the execution efficiency of container becomes crucial to ensuring the quality of service for the container-dependent task. The execution efficiency of container is mainly affected by startup time and running time, where the optimization of startup time has been neglected by many studies [12, 13]. However, the startup latency has a considerable impact on the completion time of the container-dependent task. Fuerst et al. [14] compared the startup time and running time of containers for multiple types of applications, and the startup time can be up to 80% of the run time. Therefore, optimization of container startup delay is necessary in edge computing-assisted industrial real-time systems.

To shrink container startup time, some work focused on reusing container instances in edge clusters. Xu et al. [15] and Chen et al. [16] avoided the cold-start process by selecting nodes where instances have already been deployed. To further enhance the utility of instance reuse, some work has delved into memory-efficient container retention strategy [17, 18]. Nevertheless, due to the diversity of industrial tasks, container retention may have very limited effect in industrial real-time systems. Nodes in these systems, i.e., embedded devices and host computers, do not have comparable memory capacity to servers. So, a cluster only has a restricted number of instances cached because the main overhead of instances is memory. Even if the cached instances have great hit rates, there are still a large number of containers requiring a cold start. In fact, they ignored that the main time overhead during container startup is the image pull process. Harter et al. [19] found that the image pull time occupies 76% of the container startup time in a benchmark test including 57 commonly used containers. This drives us to optimize the execution efficiency of container-dependent tasks by focusing on the image pull time.

Containerization technology represented by Docker applies a layered structure when building images. The union mount file system, e.g., AUFS, overlay2, enables different container to share the same layer, which is extremely beneficial for storage and transmission. Specifically, when a node needs to start a container, it can reuse the layers that already exist locally and just pull the missing layers. Funari et.al [20] compares various container scheduling policies, i.e., random, image locality, and layer locality, among which layer locality has the most reuse data volume. However, layer reuse is still ignored by many work, even if they take the image pull time into account [17, 21, 22].

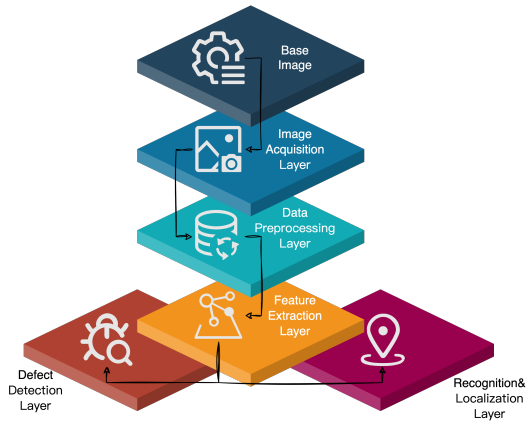

Fig. 1. Workflow and image structure of the industrial applications.

It is promising to optimize the startup delay through layer reuse in industrial real-time systems. Industrial task workflows often exhibit varying degrees of similarity, leading to substantial data duplication at the image layer granularity. To further reveal the potential of layer reuse, we analyze the relationship between the workflow and image structure of the industrial applications. As shown in Figure 1, G1 and G2 are industrial vision applications for defect detection and workpiece localization, respectively. Due to the similarity of the application types, they may share the same system dependencies and libraries, which results in an identical base image. Furthermore, subtasks in the workflow are sequentially added to the base image as upper layers. Consistent subtask sequences produce identical upper layers, which can be reused during container startup. Industrial applications with similar workflows are common in practice [23–25]. Building on this relationship between application similarity and image structure, some approaches have even reduced network and storage overhead by deliberately increasing the number of identical layers [26].

Despite the great potential of layer reuse on optimizing the efficiency of container-dependent tasks in industrial real-time systems, existing work suffers from two problems. First, not maximizing the utilization of disk space in exchange for image pulling time due to inadequate usage of disk and historical images. Second, oversimplification of resource capacity leads to practice difficulties in industrial real-time systems. Some work allocates a portion of the disk space as a registry, which does not maximize the utilization of the disk and brings additional overhead when registry adjustment occurs [27–29]. While the others limit the reuse scope in the running container. Once the container instance is destroyed, its image is also erased, resulting in significant data waste that could be reused by subsequent tasks [20, 30, 31]. In contrast, few works expand the reuse scope by leveraging layers from tasks previously dispatched, which have the potential to maximize disk utilization. Nevertheless, existing work has yet to realize it [17, 32]. In addition, most of the works mentioned above primarily focus on the resource overhead of images on disk, while other resource constraints are often oversimplified. However, during the container runtime stage, container instances and computations are also constrained by memory and CPU capacity [21]. This simplification limits the practical applicability of these approaches.

To address these issues, two key challenges need to be tackled. The first is maximizing disk utilization while ensuring that the disk requirements of images are met. This necessitates a feasible disk management mechanism. Besides, since the tasks arriving on the nodes in the future are unknown, it is impossible to predict which layers will be reused. Therefore, the proper layer cache must be determined within the limited space. The second is developing efficient scheduling

algorithms suitable for industrial real-time systems, considering disk, memory, and CPU capacities. On one hand, the algorithm must meet the rapid decision-making demands of real-time systems. On the other hand, it is essential to optimize task execution efficiency from a long-term perspective.

In this article, we formulate the **container-dependent task scheduling problem** (**CDTSP**) in edge computing-assisted industrial real-time systems. To ensure usability in practice, the model comprehensively considers resource capacity in terms of network, disk, memory, and CPU. Moreover, the image pulling, container running, and task waiting are all taken into account. A **layer reuse-based collaborative scheduling** (**LRCS**) framework is proposed to minimize long-term online task completion time. LRCS operates in a weakly coupled manner between the execution node and the scheduling node. At the execution node, LRCS employs an **evict-when-insufficient** (**EWI**) mechanism to maximize disk usage and enhances the reuse rate through a **dynamic value assessment-based layer caching** (**DVALC**) algorithm. At the scheduling node, the LRCS adopts an **active waiting** (**AW**) mechanism integrated with a **deep Q-network** (**DQN**) algorithm, which optimizes the long-term processing efficiency by adjusting the scheduling order of online tasks. The proposed LRCS is validated on a testbed using real device data and compared with state-of-the-art layer reuse-based algorithms. Experimental results demonstrate that the proposed algorithm outperforms all baselines.

The contributions are summarized as follows:

— We formulate the CDTSP in edge computing-assisted industrial real-time systems, providing a comprehensive analysis of resource constraints. The impact of image pulling, container running, and task waiting are all considered in the model.
— To enable efficient continuous online decision-making, we propose a novel LRCS framework featuring weakly coupled collaboration between the scheduling node and the execution node. At the execution node, an EWI mechanism is introduced to maximize disk utilization, while a DVALC algorithm is employed to enhance the layer reuse rate. At the scheduling node, an AW mechanism-integrated DQN is utilized to optimize long-term execution efficiency.
— A testbed built on a real device is used for evaluation. The proposed LRCS is compared with state-of-the-art layer reuse-based algorithms. Experimental results show that LRCS is significantly better than the baselines in terms of task completion time and reuse data volume.

The rest of the article is organized as follows. Section 2 reviews related work. Section 3 formulates the system model and problem. Section 4 presents the proposed LRCS framework. Section 5 evaluates the proposed algorithm. Section 6 concludes the article.

2 Related Work

In this section, we review existing work on container-dependent task scheduling that utilizes layer reuse. We categorize them into node registry-based layer reuse, mounted container-based layer reuse, and node cache-based layer reuse, and explain their mechanisms. Additionally, we discuss the benefits and limitations of each category through comparison and identify the research gap.

2.1 Node Registry-based Layer Reuse

In node registry-based layer reuse, a certain volume of the node disk is allocated for long-term layer storage, effectively establishing an image registry on the edge node. The layers in the registry are updated based on specific triggering conditions.

Hu et al. [33] make a joint decision on request scheduling and layer reservation in a vehicle-infrastructure collaboration scenario. A proper request shifting and layer retention strategy can efficiently save the system overhead in the region. However, this work does not explicitly give a

limit on the total number of the retained layers. It can lead to limitations in selecting resources for tasks with high disk consumption, since the occupancy of disk by the retained layers in a time slot is not adjustable. Xiao et al. [34] took cold-start latency, function execution latency, inter-function transfer latency, image caching cost, and other overheads into account to determine the function images to be saved in the edge server. Meanwhile, in order to avoid the overhead generated by frequent image layout changes, the work adopts a lazy update approach to readjust the function image layout when the overhead of cold-start is larger than the overhead of non-cold-start. However, storing function images is a coarser-grained way of data reuse compared to the layer. Yin et al. [27] analyzed the hierarchical relationship of multiple container image structures and described it as a tree structure. Furthermore, a fixed-size space is partitioned on the edge server to pre-store layers, and the stored layers are adjusted in two phases based on the tree structure. During the initialization phase, layer size and the number of child nodes are considered to decide which layers to store. In the runtime phase, if the downloaded layers exceed the threshold, the layer registry is updated by combining the layer size and reuse frequency.

2.2 Mounted Container-based Layer Reuse

In mounted container-based layer reuse, layers from containers currently mounted on a node can be reused. After a task completes, its layers will be removed if they are not reused by other containers.

Funari et al. [20] greedily selected resources to maximize the reuse of the image layer for subsequent containers based on the container information being mounted on the node, and prevented containers from being centrally scheduled to a few nodes through fairness constraints. Ma et.al [31] selects the target node for migration based on the reusability of the layer during container migration, making the migration efficiency effective. Tang et.al [30] considered the correlation of task sequences within a time slot in terms of layers. By characterizing the layer information that has been mounted and is being pulled on edge nodes, online tasks are scheduled using policy-based reinforcement learning. However, due to the diversity and frequent changes in edge services, using the layer information as the network inputs results in poor scalability and dimensionality explosion. In other words, adding any new layer requires restructuring and re-training the policy network.

2.3 Node Cache-based Layer Reuse

In node cache-based layer reuse, disk utilization is comparable to memory management in the operating system. Layers update dynamically depending on the tasks being executed and are only evicted when disk space is insufficient for a new task.

Fuerst et.al [14] optimized the cold start problem of FaaS functions by using an instance keep-alive approach. When the memory is insufficient for a new function, some container instances are evicted according to priority, considering function usage frequency, recent usage time, cold start cost, and memory preservation overhead. Although this work performs instance caching in memory rather than layer caching on disk, which is not directly related to the layer reuse problem in this article, the resource utilization approach remains instructive. Mou et al. [35] used an LFU-based algorithm to manage the disk at the image granularity and adjusted the cache with a multi-action DQN. While the LFU algorithm is traditional in memory management, image-based disk caching algorithms should be considered more carefully since image sizes are inconsistent. And like [30], the study directly uses the image information as an input to the reinforcement learning network, which can lead to dimensionality catastrophe and limited flexibility. Dolati et.al [32] utilizes layer sharing relationships to orchestrate the placement of VNF containers. Instead of immediately cleaning up layers after a task is finished, only a limited amount of space is cleared based on the priority of the layer when a new task arrives. The priority of each layer is accumulated from reuse times and

Table 1. Comparison of Advantages and Limitations Across Categories

Category	Node Registry	Mounted Container	Node Cache	Ours
Disk Utilization	↑	↓	↑	↑
Tuning Flexibility	↓	↑	↑	↑
Tuning Effect	↓	↓	↓	↑
Reuse Utility	↓	↓	↓	↑

the original priority multiplied by a discount factor. However, similar to [35], this work does not address the issue of differing layer sizes.

2.4 Comparison and Research Gap

Table 1 compares the advantages and limitations of the three categories mentioned above with the algorithm proposed in this article. The comparison focuses on disk utilization, tuning flexibility, and tuning effects, which ultimately influence the utility of layer reuse. Specifically, the node registry-based layer reuse utilizes a certain volume of disk capacity but faces issues with periodic data updates: the data in the registry cannot be modified during the interval. Registry tuning is usually triggered when reuse efficiency is low. If the conditions are too strict, the registry will be adjusted frequently, but if they are too lenient, reuse efficiency might already be low before any adjustments happen. Additionally, fixing storage space through partitioning can make disk constraints on edge resources more restrictive. Moreover, the layers that are adjusted need to be re-pulled, which causes unnecessary network load. These issues result in its poor flexibility in tuning.

In contrast, mounted container-based layer reuse avoids the above problem because the image data stored on disk is always adjusted based on the mounted containers on the node. However, since layers not reused by any container will be deleted, it also has notable limitations. Layer reuse essentially trades disk space for network transmission time, but mounted container-based layer reuse does not fully utilize storage resources. The number of containers mounted on a node within a time slot is often limited by CPU and memory rather than disk, resulting in fewer layers available for reuse.

Node cache-based layer reuse can effectively enhance disk utilization and provide more reusable image layers for containers. However, this approach has not been thoroughly studied or emphasized. Current research mainly uses some operating system memory management algorithms, which do not fully leverage the advantages of layer reuse. In reality, because of the differences in layer sizes, assessing layer value involves many factors, and this issue needs further exploration. Moreover, these methods primarily aim to optimize reuse mechanisms in network-limited scenarios. At the same time, assumptions about the computing limitations of edge devices are overly simplified, which can restrict their effectiveness in industrial real-time systems.

In summary, this article focuses on fully utilizing disk space while ensuring that cache layers within nodes can flexibly and effectively adjust to changes in container types. While enhancing layer reuse utility, it also considers computational constraints to ensure applicability in real-world industrial real-time systems.

3 System Model and Problem Formulation

The processing of a container-dependent task in edge computing-assisted industrial real-time systems is shown in Figure 2. Step a: The industrial terminal sent a task to the scheduling node in the edge cluster. Step b: The scheduling node assigns the task to an execution node according to a certain strategy. Step c: The execution node checks the container image required by the task. Due to the layered structure of the image, the layers existing locally can be utilized directly, while the

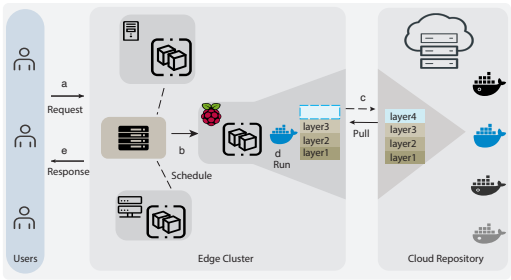

Fig. 2. The processing of container-dependent tasks in edge computing-assisted industrial real-time systems.

Table 2. Notations in Section 3

Symbol	Description
$c \in \mathcal{C} = \{1, 2, \dots, C\}$	Container and container set
$l \in \mathcal{L} = \{1, 2, \dots, L\}$	Layer and layer set
$\phi_{c,l} = \{0, 1\}$	Variable to indicate whether container c contains layer l
d_l	The data volume of layer l
$e \in \mathcal{E} = \{1, 2, \dots, E\}$	Execution node set
$\psi_{e,l} = \{0, 1\}$	Variable to indicate whether node e stores layer l
D_e	Disk size of node e
B_e	bandwidth of node e
$b_e(t) = \{0, 1\}$	Variable to indicate whether the network of node e is idle at t
$U_e(t)$	Remaining CPU capacity of the node e
$M_e(t)$	Remaining memory capacity of the node e
$r \in \mathcal{R} = \{r, r+1, \dots\}$	Real-time task and queue
$\rho_{c,r} = \{0, 1\}$	variable to indicate whether task r corresponding to container c
$u_{c,e}$	CPU requirement of container c on node e
$m_{c,e}$	Memory requirement of container c on node e
$\tau_{c,e}$	Run time of container c on node e
$\chi_{e,r} = \{0, 1\}$	Decision variable to indicate whether assign task r to node e
$D_{e,r}^{pull}$	The data volume to be pulled of task r on node e
$D_{e,r}^{reuse}$	The data volume to be reused of task r on node e
$T_{e,r}^{pull}$	The pull time of task r on node e
$T_{e,r}^{run}$	The run time of task r on node e
$T_{e,r}^{wait}$	The waiting time of task r on node e
$T_{e,r}^{comple}$	The completion time of task r on node e

others need to be pulled from the registry. Step d: When the node obtains all the required layers, it instantiates the container and processes the task. Step e: The processing results are returned to the scheduling node, which responds to the terminal. In the following, we formulate the CDTSP in industrial real-time system. For ease of reference, the main notations used in this section are summarized in Table 2.

3.1 System Model

3.1.1 Container and Layer. With the above procedure, we can describe the containers in the edge cluster as a set, denoted as $c \in \mathcal{C} = \{1, 2, \dots, C\}$. The layers that construct the container in $\mathcal{C}$

can be described as a set, denoted as $l \in \mathcal{L} = \{1, 2, \dots, L\}$. The size of layer l is d_l, and the binary variable $\phi_{c,l} = \{0, 1\}$ is used to describe the relationship between the layer and the container, $\phi_{c,l} = 1$ if the layer l is in the image of the container c, and 0 otherwise.

3.1.2 Edge Cluster. The execution nodes in the cluster are denoted as $e \in \mathcal{E} = \{1, 2, \dots, E\}$. The disk capacity of the node is D_e. The layers stored in the node are described by binary variable $\psi_{e,l} = \{0, 1\}$, $\psi_{e,l} = 1$ indicates that the layer l is stored on the node e, and 0 otherwise. The bandwidth of the node e is B_e, and we use binary variable $b_e(t) = \{0, 1\}$ to denote the real-time network occupancy, where $b_e(t) = 1$ indicates the link is occupied, 0 otherwise. The real-time remaining CPU and memory capacity of the node is denoted as $U_e(t)$, $M_e(t)$, respectively.

3.1.3 Real-time Task. The scheduling node receives the pending task and places it into the queue, denoted as $r \in \mathcal{R} = \{r, r+1, \dots\}$. At each time slot t, the scheduling node takes out the head task r of the queue and tries to assign it to an execution node according to a certain strategy. The relationship between task and container is described by the binary variable $\rho_{c,r} = \{0, 1\}$, $\rho_{c,r} = 1$ indicates that task r needs to be processed by the container c, 0 otherwise. The execution time of a task on a node is platform and resource usage dependent. In practice, tasks can be executed in parallel on nodes, but there is a complex performance interference between container instances. In order to focus on studying the reuse mechanism and scheduling strategy of the layer, the performance interference can be simplified. It is assumed that a task can be completed within a bounded execution time if the node has sufficient computing power. Otherwise, the task may experience excessive delays or fail to complete within a reasonable timeframe. The computing power requirement of the task specifically includes CPU and memory, using the variable $u_{c,e}$ and $m_{c,e}$ to denote the CPU and memory requirements of the container c on the node e, respectively. Due to the heterogeneity of the edge nodes, the execution time of a task is varying on different nodes, and the run time of the container c on the node e is represented as $\tau_{c,e}$. The scheduling decision is expressed as $\chi_{e,r} = \{0, 1\}$, $\chi_{e,r} = 1$ indicates that task r is assigned to the node e, 0 otherwise.

3.2 Cost and Constraints

In industrial real-time systems, task processing efficiency is directly related to system performance, so we consider task completion time to be the cost. For scheduling task r to node e, the data volume to be pulled can be expressed as

$$D_{e,r}^{pull} = \sum_{e=1}^{E} \sum_{l=1}^{L} \sum_{c=1}^{C} \rho_{c,r} \phi_{c,l} (1 - \psi_{e,l}) d_l, \tag{1}$$

then the pull time of task r is follows:

$$T_{e,r}^{pull} = \frac{D_{e,r}^{pull}}{B_e}. \tag{2}$$

The pulling process exclusively occupies the network and cannot be interrupted, so the link of node n should be originally idle during the pulling time period of task r, which is represented as

$$b_e(t) = 0, \quad \forall t \in T_{e,r}^{pull}. \tag{3}$$

Subsequently, if the node e meets the computing power requirements of container c, the run time of task r can be denoted as

$$T_{e,r}^{run} = \sum_{c=1}^{C} \rho_{c,r} \tau_{c,e}. \tag{4}$$

The node should always satisfy the CPU and memory constraints during the run time

$$\begin{cases} \sum_{c=1}^{C} \rho_{c,r} u_{c,e} \leq U_e(t), & \forall t \in T_{e,r}^{run}, \\ \sum_{c=1}^{C} \rho_{c,r} m_{c,e} \leq M_e(t), & \forall t \in T_{e,r}^{run}. \end{cases} \tag{5}$$

Besides, the layers stored in a node cannot exceed the disk capacity at any time

$$D_e \geq \sum_{l=1}^{L} \psi_{l,e}(t) d_l, \quad \forall t, \forall n \in \mathcal{E}. \tag{6}$$

Moreover, each real-time task corresponds to a unique container

$$\sum_{c=1}^{C} \rho_{c,r} = 1, \tag{7}$$

and task r is scheduled to the unique node

$$\sum_{e=1}^{E} \chi_{e,r} = 1, \quad \forall r \in \mathcal{R}. \tag{8}$$

Note that the task may suffer from the waiting time due to the resource constraint in Equations (3) and (5). It can be denoted as $T_{e,r}^{wait}$.

3.3 Problem Formulation

The processing of the task in the edge cluster includes request transmission, task waiting, image pulling, container running, and result return. We ignore the time for request transmission and result return, since the data volume of the request and the result is much less than the image. Then the completion time of a container-dependent task can be expressed as

$$T_{e,r}^{comple} = T_{e,r}^{wait} + T_{e,r}^{pull} + T_{e,r}^{run}. \tag{9}$$

Furthermore, our optimization objective is to minimize the system cost, which is the total completion time of the tasks:

$$P1 : \min_{\chi} \sum_{r=1}^{|\mathcal{R}|} \chi_{e,r} T_{e,r}^{comple} \quad s.t. \quad (3), (5), (6), (7), (8). \tag{10}$$

We refer to this problem as the CDTSP, which covers the heterogeneity and capacity constraints of the CPU, memory, disk, and network of the nodes, as well as the image pull, container running, and waiting cost of the task.

3.4 Problem Analysis

We can analyze the CDTSP based on Equation (9). On one hand, the term $T_{e,r}^{pull}$ is mainly related to the transmission data volume $D_{e,r}^{pull}$ according to Equation (2), which can be further expressed as

$$D_{e,r}^{pull} = \sum_{e=1}^{E} \sum_{l=1}^{L} \sum_{c=1}^{C} \rho_{c,r} \phi_{c,l} d_l - \sum_{e=1}^{E} \sum_{l=1}^{L} \sum_{c=1}^{C} \rho_{c,r} \phi_{c,l} \psi_{e,l} d_l, \tag{11}$$

where the former item is the data volume of the full image, while the latter item is the reused data volume on the node. Considering the image size is fixed, the optimization of $T_{e,r}^{pull}$ can be

reformulated as the problem of maximizing the system long-term reused data volume,

$$P2 : \max_{\chi_{e,r}} \lim_{r\to\infty} \sum_{r=1}^{|\mathcal{R}|} \sum_{e=1}^{E} \sum_{c=1}^{C} \sum_{l=1}^{L} \chi_{e,r}\rho_{c,r}\phi_{c,l}\psi_{e,l}(t)d_l. \tag{12}$$

In P2, the reused data volume depends not only on the decision variable $\chi_{e,r}$, but also on the layer cache $\psi_{e,l}(t)$ of the assigned node. We can strip away the scheduling decision $\chi_{e,r}$ and discuss it from the perspective of the node, then P2 can be transformed into the following:

$$P3 : \max \lim_{r\to\infty} \sum_{r=1}^{|\mathcal{R}'|} \sum_{c=1}^{C} \sum_{l=1}^{L} \rho_{c,r}\phi_{c,l}\psi_{e,l}(t)d_l, \tag{13}$$

where $\mathcal{R}'$ is the queue of tasks dispatched to the node, which is unknown in advance due to the uncertainty of the real-time task and the scheduling decision. While $\psi_{e,l}(t)$ of a node contains two types of layers at any moment. One type is used by the running container, denoted as $\psi_{e,l}^{using}(t)$, and the other type is used by the once-running container, denoted as $\psi_{e,l}^{used}(t)$. However, owing to disk capacity constraint, i.e., Equation (6), not all of the layers ever used will be retained. Therefore, we need to address how to manage the layers cached without queue information, which is crucial to maximize the system long-term reused data volume.

On the other hand, the term $T_{e,r}^{run}$ and $T_{e,r}^{wait}$ are mainly affected by the heterogeneity and limitations of resource. The scheduling decisions need to harmonize the short-term cost of the current task with the utilization of the cluster, which is actually an online multidimensional bin-packing problem. Taking the above analysis into account, online decision $\chi_{e,r}$ should be made from a long-term perspective with the consideration on the above three terms. In this problem, if we discretize the system state with a proper time slot, then the system state $\mathcal{S}_r$ can be considered memoryless between slots, i.e., consistent with the first-order transfer property. Therefore, we reconstruct P1 as the following Markov decision process problem:

$$P4 : \min \mathbb{E}\left\{\sum_{r=1}^{|\mathcal{R}|} T_r^{comple}(S_r, \chi_{n,r})\right\} \quad s.t. \quad (3), (5), (6), (7), (8). \tag{14}$$

4 The Proposed LRCS

In this section, we propose the LRCS framework which collaborates at the execution node and the scheduling node in a weakly-coupled manner. At the execution node, LRCS maximizes layer-reused utility by fully utilizing the disk space and enhancing the reuse rate of the layer. At the scheduling node, LRCS optimizes the long-term processing efficiency by adjusting the scheduling order of online tasks.

4.1 LRCS at Execution Node

4.1.1 EWI Mechanism. Optimizing pull time through layer reuse essentially uses disk space to tradeoff data transmission time. Therefore, the node should maximize the usage of the disk to provide as many layers as possible for reuse. Based on this idea, we propose the EWI mechanism. Specifically, the node only destroys the container instance but does not erase the image file after a task is completed. As the node processes a series of tasks, the layers stored in the disk increase. While the node evicts layers only if the disk space is insufficient. We can clarify the trigger and scope of eviction in the EWI mechanism.

— Trigger of eviction: Eviction will only happen when a new task is dispatched to the node. At this time, the node evicts the layers according to a strategy to ensure that the remaining disk space can meet the requirement of the new task.
— Scope of eviction: As analyzed in the previous section, the layers in the disk at any moment contain two types. Layers used by running containers are not evictable, while layers used by once-running containers are evictable. Thus, the scope of eviction is $\psi_{e,l}^{used}(t)$.

4.1.2 Problem Reformulation. EWI mechanism maximizes disk utilization and reserves more reusable layers for nodes. At the same time, the trigger and scope of eviction have been clarified. We introduce the decision variable $\lambda_{e,l}(t) = \{0, 1\}$ to denote the eviction strategy, and then P2 can be further redefined as

$$P5 : \max_{\lambda_{e,l}} \lim_{r\to\infty} \sum_{r=1}^{|\mathcal{R}'|} \sum_{c=1}^{C} \sum_{l=1}^{L} \rho_{c,r}\phi_{c,l}\psi_{e,l}(t,\lambda)d_l, \tag{15}$$

where $\psi_{e,l}(t,\lambda)$ is the layer cache after eviction. Explicitly, the layer cache $\psi_{e,l}(\tau) = \psi_{e,l}^{used}(\tau) + \psi_{e,l}^{using}(\tau)$ at τ will change to

$$\psi_{e,l}(t,\lambda) = ((\psi_{e,l}^{used}(\tau) - \lambda_{e,l}(t)) + ((\psi_{e,l}^{using}(\tau) + \phi_{c,l}(1 - \psi_{e,l}(t)) \tag{16}$$

after eviction and new task pulling. Besides, $\lambda_{l,n}(t)$ needs to satisfy following constraints:

$$\lambda_{e,l}(t) \le \psi_{e,l}^{used}(\tau), \tag{17}$$

which means the using layers will not be evicted, and

$$\sum_{l=1}^{L} \lambda_{e,l}(t)d_l \ge \sum_{l=1}^{L} \phi_{c,l}(1 - \psi_{e,l}(t))d_l, \tag{18}$$

which means disk space can meet the requirement of new task after eviction.

However, we still face node task queues $\mathcal{R}'$ is unknown in P5, which drives us to utilize the historical usage and intrinsic information of the layer to determine $\lambda_{e,l}(t) = \{0, 1\}$. Inspired by the principle of locality [36] and the tree structure of multi-container image, we extract the following factors that can reflect the future access probability of a layer:

— Recent access time Π_l^{recent}: reflects short-term access hotspots. In situations where the distribution of container types changes frequently, e.g., shopping malls, stations, and so on, layers that have recently been accessed have a greater likelihood of being accessed in the future.
— Historical access frequency $\Pi_l^{frequency}$: reflects the distribution of access over a period. In situations where the distribution of container types is stable, e.g., schools, companies, and so on, the layers that are accessed more frequently have a greater likelihood of being accessed in the future.
— Number of child nodes Π_l^{child}: reflects the probability of reuse in the tree structure of images, in the situations where the distribution of container types is lacking, e.g., when the system is first started, the base layer has a greater probability of being used because it is shared by more containers.

Here, we need to clarify the meaning of the number of child nodes. Containers leverage union mount file systems to enhance data reusability. When modifications are made to the contents of base layers, a new layer with the modified content copied and rewritten is appended on top, thereby ensuring the reusability of the base layers. As different base images often have different appended layers, it can be deduced that a key condition for reusability between two layers is that their base

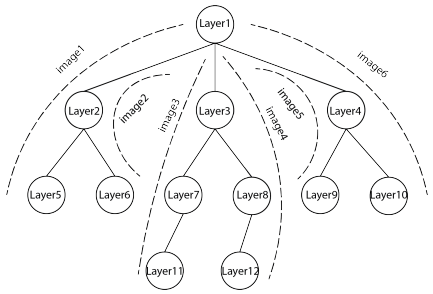

Fig. 3. Layer reuse relationship of batch containers in the tree structure.

layer sequences must be identical [27, 37]. Building on this relationship, a tree structure is a suitable way to represent a set of images. As illustrated in Figure 3, by treating the layers as nodes and the sequence relationships between them as edges, the composition of images and their layer reuse relationships can be depicted. For a collection of containers, these relationships form a forest composed of multiple trees. Then, Π_l^{child} is got, including all nodes in the subtree with l as the root node.

Using the above information, we can roughly evaluate the probability of a layer being used in the future. In other words, layers that are more likely to be reused have greater value, which can be expressed as a weighted sum of the above factors:

$$\mathbb{v}_l = \kappa_1 \Pi_l^{recent} + \kappa_2 \Pi^{frequency} + \kappa_3 \Pi^{child}. \tag{19}$$

Furthermore, combined with the size of the layers, P2 can actually be reformulated as a 0-1 knapsack problem. we use Equation (17) to determine the range of optional items and Equation (18) to determine the knapsack capacity, and

$$P6 : \max_{\lambda_{e,l}} \sum_{l=1}^{L} (\psi_{e,l}^{used}(\tau) - \lambda_{e,l}) \mathbb{v}_l \quad s.t. \quad (17), (18). \tag{20}$$

There are numerous polynomial-time approximate solutions for the knapsack problem [38]. Considering the importance of scheduling efficiency in online system, we directly use a greedy way to solve P6.

4.1.3 DVALC Algorithm. In P6, the value assessment of the layers has an important impact on the eviction and reuse data volume. Indeed, the factors we consider are not in a superimposed relationship but rather favor one of them depending on the task type distribution and the online decision. As the previous analysis shows, Π_l^{child} is more representative of the layer value when the container types are uniformly distributed. $\Pi_l^{frequency}$ is more important when the container types show a stable non-uniform distribution. Π_l^{recent} is more adaptable when the distribution of container type changes frequently. Therefore, we set the value assessment weights $\mathbb{W}_0 = [\kappa_1, \kappa_2, \kappa_3]$ more finely based on the reuse data volume, so that the value assessment can be dynamically adapted to the distribution of container types in different situations.

The update of the weights occurs when the eviction is triggered and the node redetermines the value assessment based on a better configuration between evictions. Based on $\mathbb{W}_0$, we first generate three sets of weight configurations favoring different value factors, and set δ to 2/3 empirically,

$$\begin{cases} \mathbb{W}_1 = [\kappa_1 + \delta, \kappa_2 - \delta/2, \kappa_3 - \delta/2], \\ \mathbb{W}_2 = [\kappa_1 - \delta/2, \kappa_2 + \delta, \kappa_3 - \delta/2], \\ \mathbb{W}_3 = [\kappa_1 - \delta/2, \kappa_2 - \delta/2, \kappa_3 + \delta]. \end{cases} \tag{21}$$

ALGORITHM 1: DVALC algorithm

Input: $\rho_{c,r}$, $\phi_{c,l}$, $\psi_{e,l}(\tau)$, $\mathcal{R}''$
Output: $\psi_{e,l}(t,\lambda)$
Initialization $\mathbb{W}_0$ and get $\mathbb{W}_1$-$\mathbb{W}_3$ by Equation (21)
for *each evict trigger* **do**
  Get $D_{e,r}^{reuse}$ by Equation (22)
  if *not first trigger* **then**
    $\mathbb{W}_0 \leftarrow \arg\max_{\mathbb{W}_i} D_{i,e}^{reuse}$
    Update $\mathbb{W}_1$-$\mathbb{W}_3$ by Equation (21)
  Get $\psi_{e,l}^i(t,\lambda)$ by Equation (16)
  Update layer cache to $\psi_{e,l}^0(t,\lambda)$

Furthermore, at each eviction trigger, these weights are used to calculate different value assessments $\mathbb{v}_l^i,\ i \in \{0,1,2,3\}$, and obtain the layer cache $\psi_{e,l}^i(t,\lambda),\ i \in \{0,1,2,3\}$ after eviction. $\mathbb{W}_0$ is used for the actual eviction, so $\psi_{e,l}^0(t,\lambda)$ is the real layer cache after the eviction. The reuse data volume for each layer cache is counted when the next eviction occurs:

$$D_e^{reuse} = \sum_{}^{\mathcal{R}''} \sum_{c=1}^{C} \sum_{l=1}^{L} \rho_{c,r}\phi_{c,l}\psi_{e,l}^i(t,\lambda)d_l \quad i \in \{0,1,2,3\}, \tag{22}$$

where $\mathcal{R}''$ is the task set between evictions.

The weight configuration with the largest reuse data volume is used for the value assessment of the next eviction and taken as the new weight configuration $\mathbb{W}_0$. At the same time, $\mathbb{W}_1 - \mathbb{W}_3$ is updated based on Equation (21). The DVACL algorithm is shown in Algorithm 1.

4.1.4 *Theoretical Analysis.*

Theorem 1. *The effective condition for DVALC algorithm is*

$$\sum_{l=1}^{L}(1-\phi_{c,l})\psi_{e,l}^{used}(t)d_l \geq \sum_{l=1}^{L}\phi_{c,l}(1-\psi_{e,l}(t))d_l. \tag{23}$$

Proof. Given a container c scheduled to node e, the layer that can be evicted is $\psi_{e,l}^{used}(t)$. The data volume of the layer in $\psi_{e,l}^{used}(t)$ that will not be used by c is $\sum_{l=1}^{L}(1-\phi_{c,l})\psi_{e,l}^{used}(t)d_l$, which should be larger than the data volume that c needs to pull. Otherwise, DVALC algorithm will fail because it cannot clear out enough disk space. □

Theorem 2. *The time complexity of DVALC is $O(\mu \log \mu)$.*

Proof. DVALC runs on execution nodes, so its complexity is considered independently based on each node's layer cache. According to Equation (17), evicted layers are identified within $psi_{e,l}^{used}(\tau)$, and thus, the value assessment is also only for these layers. The number of layers in $\psi_{e,l}^{used}(\tau)$ is typically independent of the total number of layers in the system, because there is a limit to how many layers can be cached on a node due to disk capacity constraints. We can determine a bound on the number of layers in $\psi_{e,l}^{used}(\tau)$ based on the size of the smallest layer, denoted as $\mu \leq \lceil D_e / \min_{l\in\mathcal{L}}\{d_l\}\rceil$. As in Algorithm 1, each run of DVALC involves two processes: layer value assessment and greedy-based knapsack solving. During the value assessment stage, DVALC calculates the value of each layer in $\psi_{e,l}^{used}(\tau)$ according to Equation (19), with a complexity of $O(\mu)$. In the greedy-based knapsack solving phase, DVALC first sorts the layers by value density, with a

complexity of $O(\mu \log \mu)$. Then, evicted layers are selected sequentially in ascending order of value until Equation (18) is satisfied, with a complexity of $O(\mu)$. Each value assessment weight $\mathbb{W}_i$ follows this process and does not affect the overall complexity since the number of weight configurations remains constant. In summary, the complexity of DVALC at each eviction trigger is $O(\mu \log \mu)$. □

4.2 LRCS at Scheduling Node

4.2.1 AW Mechanism. As analyzed in Section 3.4, the scheduling of real-time tasks is a multidimensional online bin-packing problem. Due to resource constraints, task r may not start executing immediately after arrival, which causes the term $T_{e,r}^{wait}$. There are two different ways in which the scheduling node addresses task waiting in resource-constrained situations. One way is to schedule the task as soon as it arrives, and the task will wait on the node, which has been used in a lot of work [17, 30, 39]. The alternative is to hold the task r in the queue $\mathcal{R}$ and schedule r again later. This way changes the constraints of the classical online bin-packing problem by introducing information about subsequent tasks into the scheduling of r. Keeping task r in queue $\mathcal{R}$ until a node can immediately process it adapts to our problem because of the following advantages:

- $T_{e,r}^{wait}$ would have been unavoidable if no nodes in the cluster could handle r. Keeping it waiting in the queue does not increase this cost.
- If a task waits on a node, the node will not evict its layers because they are known to be used in the future. According to Theorem 1, the evictable layers $\psi_{e,l}^{used}$ will gradually disappear as the tasks pile up, which can lead to DVALCA failure.
- Using task look-ahead, the online bin-packing problem will have more potential for optimization, and we illustrate this with an example.

Given a queue $\mathcal{R} = \{1, 2, 3, 4\}$, the resource occupancy of the tasks is [0.3, 0.6, 0.4, 0.7]. The complete online bin-packing algorithm may make a suboptimal decision when we aim to use as few slices as possible to ensure that more resources can be available for subsequent tasks. The classical first fit (i.e., greedily choosing the first slice that satisfies the capacity requirement in each decision) and best fit [40] (i.e., greedily choosing the slice with the smallest remaining capacity in each decision) both result in the following scheduling due to not utilizing the information of the subsequent tasks: $\{1, 2\}$, $\{3\}$, $\{4\}$. The optimal scheduling, on the other hand, is clearly:$\{1, 4\}$, $\{2, 3\}$. In fact, the online bin-packing algorithm with a look-ahead offers better competitive rates than the complete online algorithm [41]. Some look-ahead algorithms implement sliding windows and make decisions in favor of subsequent tasks, e.g., through dynamic programming or metaheuristic [42]. However, such computational overhead is clearly unaffordable in industrial real-time systems.

AW mechanism is introduced to utilize look-ahead information more efficiently. In the above example, even the complete online algorithms of first fit and best fit can make optimal scheduling if the task order is rearranged to $\mathcal{R} = \{2, 3, 4, 1\}$. Inspired by this idea, we utilize look-ahead information by rearranging the scheduling order of tasks. Specifically, the scheduling node still takes out the head task r of the queue $\mathcal{R}$ each time. However, there are two possible decisions in AW mechanism, one is scheduling r to an execution node and starting processing immediately, and the other is inserting r into the tail of the queue. This decision-making approach does not require considering the complete queue information at each time, but is effective in adjusting the order of tasks. Indeed, in order to reach a more optimal task ordering, it is possible for AW to make tasks actively wait even if there exist nodes that can process task r immediately.

4.2.2 Algorithm Settings. AW mechanism extends the decision options of the MDP problem in P4, which gives the problem a wider optimization space. However, we still need to determine online decisions, and reinforcement learning algorithms perform well for solving such problems. Among many reinforcement learning algorithms, **Deep-Q learning** (**DQL**) has the advantage of

lightweight and fast computation, which is well suited for resource-constrained conditions and fast decision-making requirements in industrial sites. In this subsection, we give the algorithm settings for DQN.

In RL, agents perform action $\mathcal{A}_t$ at time slot t according to environment state $\mathcal{S}_t$, and thus get the reward $\mathcal{RW}_t$ given by the environment. Subsequently, the environment will transit to new state $\mathcal{S}_{t+1}$. Combined with a suitable discount, the long-term cumulative reward of agent is expressed as

$$U_t = \sum_{t=0}^{\mathcal{T}} \gamma^t \mathcal{RW}_t. \tag{24}$$

The action-value function can evaluate the long-term cumulative discount reward of action $\mathcal{S}_t$ made by the agent under state $\mathcal{A}_t$, denoted as

$$Q_\pi(\mathcal{S}_t, \mathcal{A}_t) = \mathbb{E}[U_t | \mathcal{S}_t, \mathcal{A}_t]. \tag{25}$$

In DQL, we approximate the optimal action-value function by DQN, denoted as

$$Q_\pi^*(\mathcal{S}_t, \mathcal{A}_t) = \max_\pi Q_\pi(\mathcal{S}_t, \mathcal{A}_t), \tag{26}$$

which can guide the intelligentsia to make online decisions that will benefit long-term return:

$$\mathcal{A}_t^* = \arg\max_{\mathcal{A}_t} Q_\pi^*(\mathcal{S}_t, \mathcal{A}_t). \tag{27}$$

To train DQN, we need to give state, action, and reward.

State: The state space needs to contain resource information and task information at time slot t. The resource information mainly contains the remaining CPU capacity $U_e(t)$, remaining memory capacity $M_e(t)$, node bandwidth B_e, and bandwidth idleness $b_e(t)$ of each node at time slot t. Thus, the resource information can be denoted as

$$\mathcal{S}_e = \{(U_e(t), M_e(t), B_e, b_e(t)) | e \in \mathcal{E}\}. \tag{28}$$

The task information mainly contains CPU requirements $u_{c,e}$, memory requirements $m_{c,e}$, run time $\tau_{c,e}$, reuse data volume $D_{e,r}^{reuse}$, transferred data volume $D_{e,r}^{pull}$ at each node and the actively waiting times w_r. Thus, the task information can be denoted as

$$\mathcal{S}_r = \{u_{c,e}, m_{c,e}, \tau_{c,e}, D_{e,r}^{reuse}, D_{e,r}^{pull}, w_r | e \in \mathcal{E}\}. \tag{29}$$

Then the state of the environment at time slot t can be denoted as

$$\mathcal{S}_k = \mathcal{S}_e \cup \mathcal{S}_r. \tag{30}$$

To ensure learning stability and model generalization, we preprocess the aforementioned state information. Specifically, the remaining CPU capacity $U_e(t)$, the remaining memory capacity $M_e(t)$, the CPU demand $u_{c,e}$, and the memory demand $m_{c,e}$ are normalized based on the overall resource capacity of the node. The runtime $\tau_{c,e}$, the reuse data volume $D_{e,r}^{reuse}$, the transmission data volume $D_{e,r}^{pull}$, and the bandwidth B_e are standardized with logarithmic variation and sliding window-based Z-score, considering their long-tailed distribution characteristics.

Action: Based on the state of the environment, the agent will make the corresponding scheduling decision $\mathcal{A}_t$, and the action space $\mathcal{A}$ is denoted as

$$\mathcal{A} = e_0 \cup \mathcal{E}. \tag{31}$$

For $\mathcal{A}_t \in \mathcal{E}$, the task will be scheduled to execute node immediately. For $\mathcal{A}_t = e_0$ the task will actively wait.

Reward: Depending on the actions made by the agent, the environment gives a reward. In Equation (10), we set the optimization objective to minimize the total completion time of tasks.

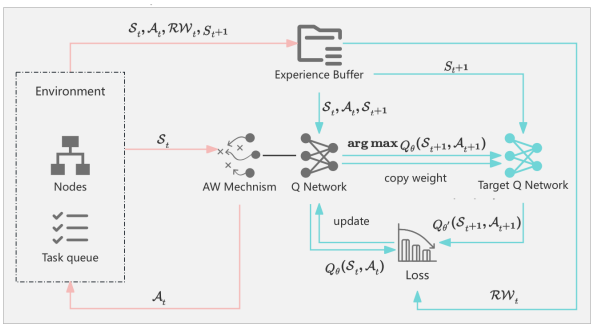

Fig. 4. AW mechanism-integrated DQN.

However, the final execution of a task may go through several decisions, only the last one is successfully executed, and all the decisions before that are to make the task wait. This will result in the environment giving sparse rewards. Therefore, we need to shape the rewards based on the effect of different decisions on the task completion time.

For successful execution of the task, the decision leads to the image pulling time and running time of the task, and the reward can be denoted as

$$\mathcal{RW}_{success} = -(T_r^{pull} + T_r^{run}). \tag{32}$$

For the case where a task fails to execute due to resource constraints or actively wait, the decision results in a waiting time for the task. Specifically, the waiting time is related to the number of tasks in front of r after it is inserted into the queue tail, and the reward can be denoted as

$$\mathcal{RW}_{wait} = -w_r * \frac{queue\_length * average\_time}{E}, \tag{33}$$

where the w_r is to avoid tasks waiting too many times from a fairness perspective. The latter part is used to measure the waiting time incurred by actively wait. Where *queue_length* is the number of tasks ahead of r, and *average_time* is the average sum of pulling time and running time of the completed tasks.

Since the reward function is nonlinear, agents may exhibit certain tendencies on waiting or executing. In order to balance such tendencies, we reshape the rewards as follows:

$$RW = \xi_1 1 \cdot \{A_k \in \mathcal{E}\}\mathcal{RW}_{success} + \xi_2 1 \cdot \{A_k = e_0\}\mathcal{RW}_{wait}, \tag{34}$$

where $1 \cdot \{\}$ is the Iverson bracket, which is equivalent to 1 when the condition is satisfied.

4.2.3 AW Mechanism-integrated DQN. The training and decision making of AW mechanism-integrated DQN is shown in Figure 4. At time slot t, the agent inputs the environment state $\mathcal{S}_t$ into the DQN, which outputs the Q value of all the actions. Q value is the evaluation of the cumulative discounted future reward. However, if the agent greedily chooses the action by Equation (27), it may miss out on better long-term returns due to not fully exploring the environment. Therefore, the ϵ-greedy strategy is adopted to balance the exploration and exploitation [43]. Specifically, the agent chooses a random action with a decaying probability.

At time slot $t+1$, the agent gets reward $\mathcal{RW}_t$ for the action $\mathcal{A}_t$, and the environment transit to state $\mathcal{S}_{t+1}$. The above transition $(\mathcal{S}_t, \mathcal{A}_t, \mathcal{RW}_t, \mathcal{S}_{t+1})$ is used to update the weight of the AW mechanism-integrated DQN to reduce the estimation bias of Q value. Based on the **temporal**

ALGORITHM 2: AW mechanism-integrated DQN Scheduling Algorithm

Input: Q_θ, $\mathcal{S}_t$
Output: $\mathcal{A}_t$, $\mathcal{RW}_t$
Generate random number $\eta \in [0, 1]$
if $\eta > \epsilon$ **then**
  Select $\mathcal{A}_t$ by Equation (27)
else
  Select $\mathcal{A}_t$ randomly
if $\mathcal{A}_t \in \mathcal{E}$ *and condition* > 0 **then**
  Schedule r by $\mathcal{A}_t$
  Give $\mathcal{RW}_t$ by Equation (32)
else
  Insert r to queue tail
  Give $\mathcal{RW}_t$ by Equation (33)
Return $\mathcal{A}_t$, $\mathcal{RW}_t$

difference (**TD**) learning algorithm [43], the loss is denoted as

$$L_\theta = (y_t - Q_\theta(\mathcal{S}_t, \mathcal{A}_t))^2, \tag{35}$$

where y_t is the TD target. A target network $Q_{\theta'}(\mathcal{S}_t, \mathcal{A}_t)$ is used to stabilize the training by providing a fixed reference for calculating target values. Then, the target can be denoted as

$$y_t = \mathcal{RW}_t + \gamma \max Q_{\theta'}(\mathcal{S}_{t+1}, \mathcal{A}_{t+1}), \tag{36}$$

where θ' is regularly synchronized from θ. However, directly performing action selection and value estimation by $Q_{\theta'}(\mathcal{S}_k, \mathcal{A}_k)$ may lead to overestimation bias of the target, which is mainly due to the max operation in Equation (36). The max operation will greedily choose positively biased actions due to the noise and uncertainty of $Q_{\theta'}(\mathcal{S}_k, \mathcal{A}_k)$ during training. This positive feedback leads to a continuous overestimation of Q value and deviation from the true expectation with iterations. Therefore, double DQN is employed to separate them [44]. Explicitly, the main network $Q_\theta(\mathcal{S}_t, \mathcal{A}_t)$ is used for action selection and the target network $Q_{\theta'}(\mathcal{S}_t, \mathcal{A}_t)$ is used for value estimation. The target is adjusted to

$$y_t = \mathcal{RW}_t + \gamma \max Q_{\theta'}(\mathcal{S}_{t+1}, \arg\max_{\mathcal{A}} Q_\theta(\mathcal{S}_{t+1}, \mathcal{A})). \tag{37}$$

As shown in Algorithm 2, the agent selects an action based on ϵ-greedy to balance exploration and exploitation, where ϵ is the probability of exploration with an initial value of 1 and a range of $[\epsilon_{min}, 1]$. If the action is to dispatch the queue head task to a node immediately, the chosen node needs to be further checked if it meets the execution conditions, which can be expressed as

$$condition = 1 \cdot \{Equation\ (3)\} \times 1 \cdot \{Equation\ (5)\}, \tag{38}$$

where $condition = 1$ indicates that the task can be executed immediately, 0 otherwise. If the condition is not met or the action is to actively wait, the task is inserted into the queue tail.

As shown in Algorithm 3, the experience replay is adopted in the training process. After the interactions between agent and environment, the transition $(\mathcal{S}_t, \mathcal{A}_t, \mathcal{RW}_t, \mathcal{S}_{t+1})$ is stored in a replay buffer D, and ϵ is decay with rate ϵ_{decay}. If enough transitions have been collected, a batch of them is randomly sampled every few steps. For each sampled transition, the loss is calculated by Equation (35) and the θ is optimized by gradient descent. If all episodes are finished, the optimal action-value function $Q^*_\pi(\mathcal{S}_t, \mathcal{A}_t)$ is obtained.

ALGORITHM 3: AW mechanism-integrated DQN Trainning Algorithm

Input: $\mathcal{S}_0$
Output: $Q_\theta(\mathcal{S}_t, \mathcal{A}_t)$
Initialize replay buffer D, main network $Q_\theta(\mathcal{S}_t, \mathcal{A}_t)$ and target network $Q_{\theta'}(\mathcal{S}_t, \mathcal{A}_t)$
for $episode = 1, 2, \ldots$ **do**
  Get state S_0 **for** $t = 1, 2, \ldots$ **do**
    Observe state $\mathcal{S}_t$
    Perform scheduling and get reward by Algorithm 2
  Observe state $\mathcal{S}_{t+1}$
  Store transition $(\mathcal{S}_t, \mathcal{A}_t, \mathcal{RW}_t, \mathcal{S}_{t+1})$ in D
  $\epsilon \leftarrow \max(\epsilon_{min}, \epsilon \times \epsilon_{decay})$
  if $|D| \mod BatchSize = 0$ **then**
    Sample a batch of transitions D' from D
    Perform gradient descent to update θ by Equation (35)
  if $t \mod TargetUpdateSlots = 0$ **then**
    $\theta' \leftarrow \theta$
Return $Q_\theta(\mathcal{S}_t, \mathcal{A}_t)$

5 Experiment and Evaluation

In this section, we compare the proposed LRCS with the state-of-the-art baselines using a testbed based on real device data. Since LRCS employs a weakly coupled collaborative manner between execution and scheduling nodes, evaluation involves ablation experiments. Specifically, the layer reuse effect of EWI and DVALC is verified on the execution node, while the completion time optimization effect of AW-based DQN is examined on the scheduling node. Besides, to validate the applicability and scalability of our framework in real industrial environments, we conducted a case study by gradually increasing the cluster scale and the task number.

5.1 Experiment Setup

5.1.1 Task Queue. We use 70 containers consisting of 391 layers as the dependence of the tasks. The image sizes range from 1.32 GB to 10.81 GB, and the layer sizes range from 212 MB to 971 MB. These containers have different resource preferences, including CPU-intensive, memory-intensive, and disk-intensive. We collected their resource requirements and run time from CPU host, Atlas 200 DK, and Raspberry Pi 4B, as shown in Figure 5. To simulate the type distribution and arrival time of tasks in real-time industrial systems, we obtained the tasks by multiple Gaussian distributions, and determined the arrival time by uniform or Poisson distributions.

5.1.2 Edge Cluster. In the experiment on layer reuse effect and completion time optimization effect, an edge cluster with 12 heterogeneous nodes was built based on the resource capacity of the three device types mentioned above. Specifically, the CPU capacity ranges from 1.2 GHz to 2.4 GHz, the memory capacity from 1 GB to 4 GB, the disk capacity from 16 GB to 32 GB, and the bandwidth is 20 MB/s.

5.2 Layer Reuse Effect

To verify the LRCS framework's layer reuse effect, we first compared EWI and DVLAC with the baseline algorithm at the execution node. Based on the summary of related work, we selected one node registry-based layer reuse approach [27], and two node cache-based layer reuse approaches [14], [35]. We refer to these three baselines as LIR, FC, and TSIC, respectively. It is worth mentioning

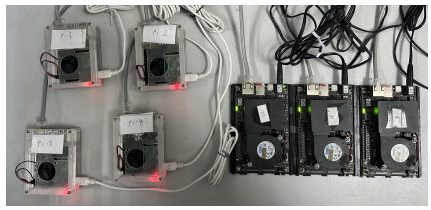

Fig. 5. Atlas 200 DK and Raspberry Pi 4B Cluster.

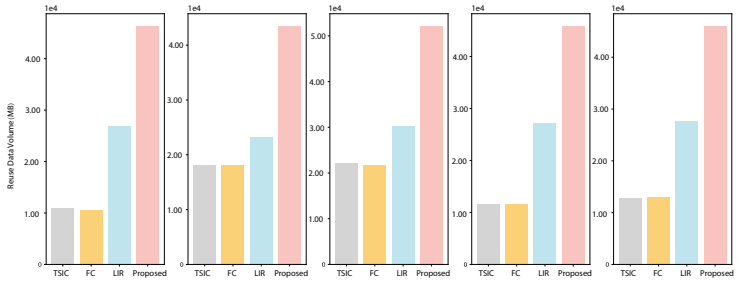

Fig. 6. Comparision of reuse data volume.

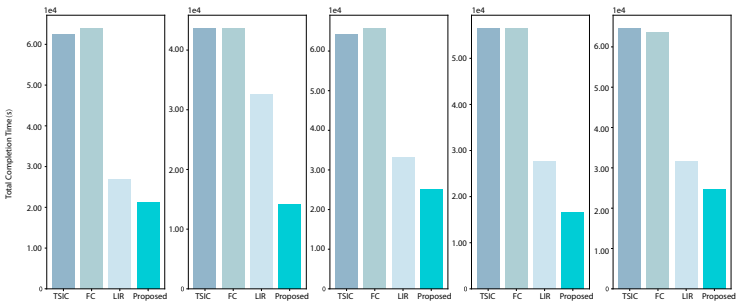

Fig. 7. Comparison of total completion time.

that since the mounted container-based layer reuse approach does not involve caching mechanisms on execution nodes, we compared them in the next subsection. To ensure a fair comparison, we uniformly use the greedy algorithm for scheduling tasks.

We use five queues with 100 slow-arriving tasks to reduce the randomness caused by task types and scheduling strategy. The comparison metrics include total reuse data volume (in MB) and total task completion time (in seconds). As shown in Figure 6, the proposed framework outperforms the baseline algorithms on all task queues in terms of the reused data volume. It is worth noting that TSIC and FC fluctuate to varying degrees due to the randomness of task sampling, which is essentially due to the fact that the directly applied LFU algorithm is not fully adapted to all task distributions.

Meanwhile, as shown in Figure 7, benefiting from the excellent data reuse effect of EWI and DVLAC, the LRCS still has a significant advantage in total task completion time under the same scheduling strategy.

Furthermore, to more intuitively demonstrate the significant impact of layer reuse effects on optimizing task completion time, we show their correlation in Figure 8. As shown in Figure 8, there

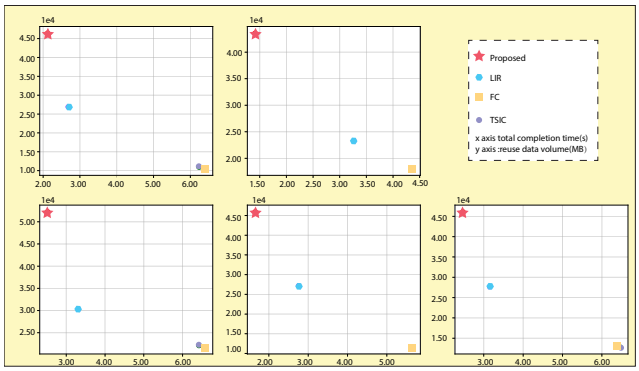

Fig. 8. Correlation of reuse data volume and total completion time.

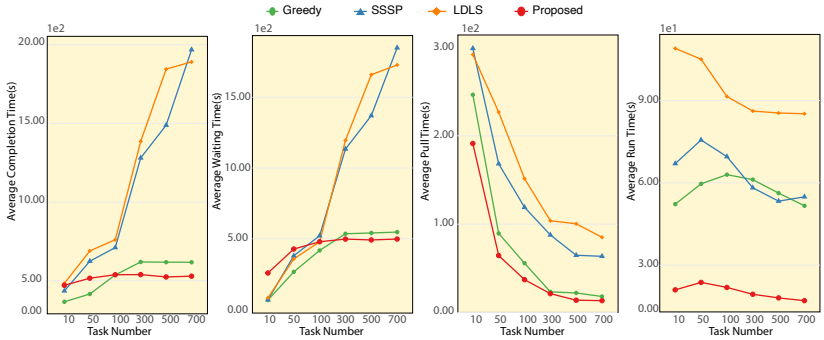

Fig. 9. Average completion time under poisson distribution.

is a negative correlation, indicating that layer reuse can significantly shrink the completion time of the container-dependent task.

5.3 Completion Time Optimization Effect

We validate the complete LRCS using three task queue setups with different arrival times, which is very common in industrial real-time systems. The execution nodes manage the layer caches using EWI and DVALC, and the scheduling node assigns tasks through AW-based DQNs. The baseline algorithms include the greedy strategy and two mounted container-based layer reuse methods, namely, SSSP [20] and LDLS [30]. To clearly analyze the completion time, we show waiting time, pull time, and run time separately. Additionally, the results are presented as averages to account for differences in the number of tasks.

First, we conducted experiments with poisson distributed task arrival times, which is used to simulate the periodic sharp increase in the number of tasks that may occur in real-time industrial systems. The number of tasks varies from 10 to 700, and the arrival time of the last task is five times the number of tasks. As shown in Figure 9, when the number of tasks is small, the task completion time of the proposed framework does not outperform all baseline algorithms because the AW mechanism increases the waiting time to gather more information about the task queue. However, the advantage of LRCS begins to appear when the number of tasks reaches 100 within 500 seconds. As the number of tasks increases further, the optimization effect of LRCS becomes significant.

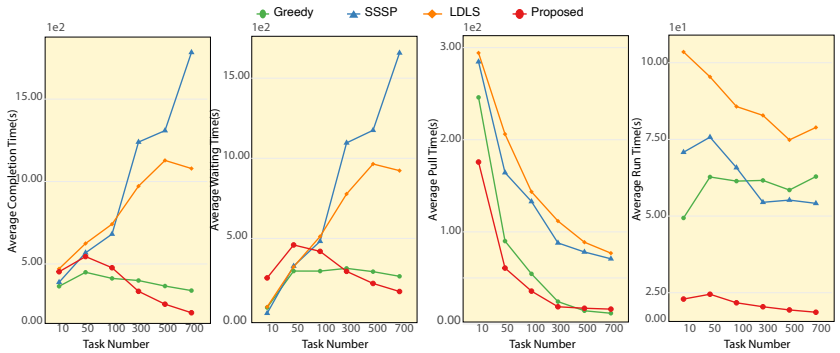

Fig. 10. Average completion time under uniform distribution.

This phenomenon relates to two reasons. First, LRCS's waiting time is consistently higher than the baseline until the number of tasks reaches 300, but the run time remains well below the baseline. This demonstrates the effectiveness of the AW mechanism in obtaining task queue information and adjusting the task execution order. When the number of tasks exceeds a certain threshold, the cost of AW becomes worthwhile. This is not only due to the tradeoff for optimized run time, but also because the tasks were already encountering a waiting situation. Second, it can be observed that the completion time of SSSP and LDLS cannot be better than the greedy strategy across all task queues. This is because the mounted container-based layer reuse approach does not maximize disk utilization, whereas EWI and DVALC are still used in the greedy strategy. Therefore, it also confirms the impact of layer reuse in LRCS on execution nodes. Comparing with the other three algorithms its average performance improvement reaches 1.79%, 108.73%, and 126.29%, respectively.

Then, we adjust the task queue arrival times to follow a uniform distribution, which helps simulate the stabilization of task numbers that can occur in real-time industrial systems. As shown in Figure 10, LRCS still outperforms the baseline algorithm when the number of tasks exceeds a certain threshold. There is even a trend for waiting times to decrease as task volume increases. This is partly because the tasks are less intensive than those modeled by the Poisson distribution and partly due to the effective performance of the AW mechanism in the online bin-packing problem. Specifically, the AW-based DQN not only optimizes the run time of the current task but also enhances overall resource utilization. It reserves more free resources for future tasks, reducing passive waiting time caused by insufficient resources. Compared with the other three algorithms, its average performance improvement reaches 2.72%, 163.05%, and 121.04%, respectively.

To thoroughly demonstrate LRCS's strong performance under resource constraints, we set the maximum task arrival time to 800 seconds and gradually increase the number of tasks from 300 to 550. As shown in Figure 11, as the number of tasks rises, the average completion time increases owing to resource intensification. However, LRCS consistently maintains the lowest increase in completion time and growth rate. This demonstrates the favorable robustness of LRCS under resource constraints. Compared to Greedy, SSSP, and LDLS, the improvements are 27.44%, 103.72%, and 132.22%, respectively.

5.4 Case Studies of Large-scale Clusters

To validate the applicability and scalability of LRCS in real industrial environments, we conducted a case study by gradually increasing the cluster scale and the task number. The cluster scale was expanded from 24 to 60 nodes, and the task number was increased from 20 to 3,500.

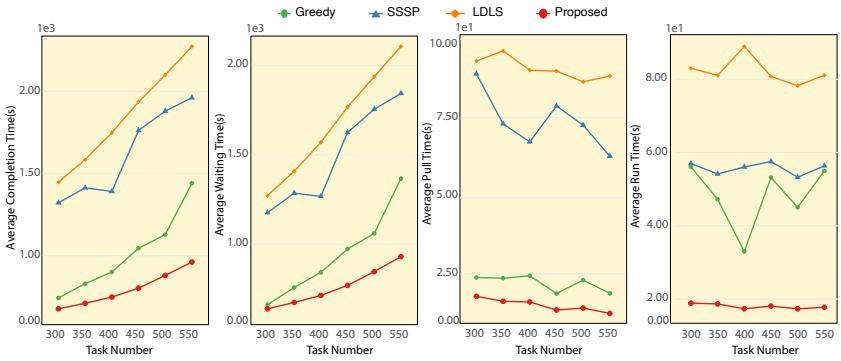

Fig. 11. Total completion time under resource constraints.

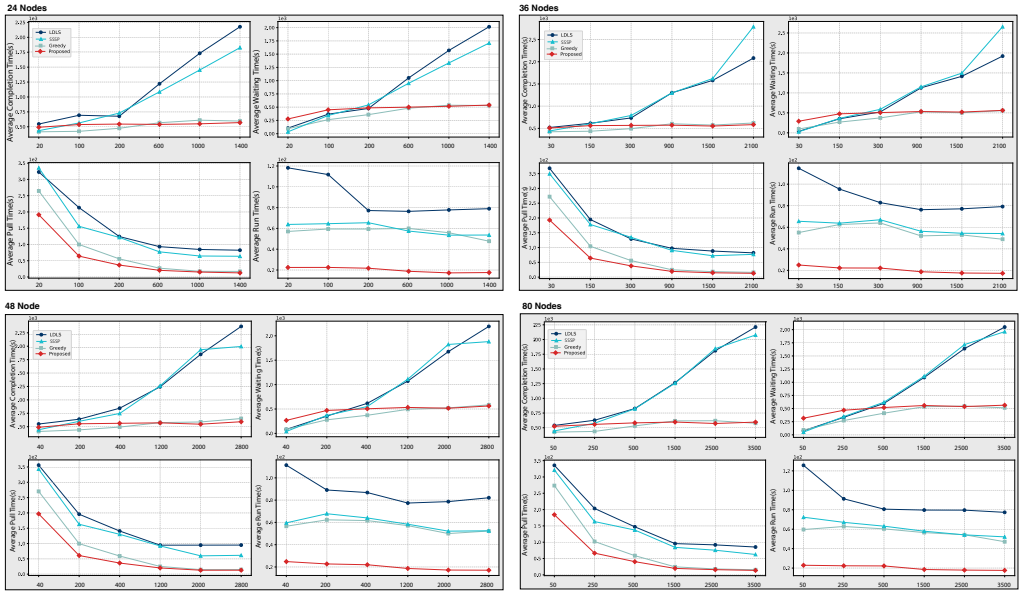

Fig. 12. Average task completion time in large-scale clusters.

As shown in Figure 12, LRCS has an advantage at all cluster scales once the number of tasks exceeds a certain point. This is because, in large-scale clusters with sufficient resources, competition for resources becomes evident only when the number of tasks exceeds this threshold, enabling the benefits of the AW mechanism to emerge. This long-term strategy is valuable in industrial environments, where task queues can grow infinitely long as the system runs nonstop. Consequently, LRCS is likely to offer significant practical benefits in real-timess industrial systems.

Meanwhile, the average completion time of LRCS remains stable despite the increasing number of tasks in clusters of different sizes. It demonstrates that LRCS has better robustness, which is essential in industrial systems.

Moreover, the LRCS waiting time remains stable across different task numbers. This demonstrates that the penalty term based on waiting times effectively ensures that the AW-based DQN balances executing the current task with exploring more information about the queue.

Furthermore, the run time of LRCS is significantly lower than that of other algorithms. This shows that AW-based DQN pays more attention to the resource utilization efficiency, which helps ensure long-term task execution efficiency.

We also observe that the average pulling time decreases and stabilizes across all algorithms, but LRCS remains lower compared to LDLS and SSSP. This illustrates that EWI and DVALC are effective at preserving the most valuable layers. The greedy method is similar to LRCS because it also employs EWI and DVALC.

Combining the case studies above, we believe that LRCS has strong applicability and scalability in real industrial systems.

6 Conclusion and Future Direction

In this article, we propose a collaborative scheduling framework based on layer reuse for the problem of optimizing the completion time of container-dependent tasks in edge computing-assisted industrial real-time systems. The framework improves the effectiveness of layer reuse by maximizing disk utilization and layer reuse rate at the execution node, and adopts an AW mechanism at the scheduling node to optimize the long-term execution efficiency under resource constraints. By experimentally comparing with the state-of-the-art algorithms, our proposed framework can effectively improve the execution efficiency.

This is our first attempt to trade disk space for image pull time in a cache management manner. Since we have adopted a low-coupling synergistic approach to solve these two problems, there is a lot of future work that can be continued on this basis. For example, further consideration of optimization objectives such as cost and energy consumption of edge cluster. The form of container-dependent tasks can also be further extended from independent tasks to the form of the dependent task.

References

[1] Megha Sharma, Abhinav Tomar, and Abhishek Hazra. 2024. Edge computing for industry 5.0: Fundamental, applications, and research challenges. *IEEE Internet of Things Journal* 11, 11 (2024), 19070–19093. DOI: http://doi.org/10.1109/JIOT.2024.3359297

[2] Jie Wang, Yulin Hu, Yao Zhu, Tong Wang, and Anke Schmeink. 2024. Reliability-optimal offloading for mobile edge-computing in low-latency industrial IoT networks. *IEEE Transactions on Wireless Communications* 23, 10 (2024), 12765–12781. DOI: http://doi.org/10.1109/TWC.2024.3396161

[3] Kai Peng, Peiyun Xiao, Shangguang Wang, and Victor C. M. Leung. 2024. SCOF: Security-aware computation offloading using federated reinforcement learning in industrial internet of things with edge computing. *IEEE Transactions on Services Computing* 17, 4 (2024), 1780–1792. DOI: http://doi.org/10.1109/TSC.2024.3377899

[4] Jing Li, Song Guo, Weifa Liang, Jianping Wang, Quan Chen, Zichuan Xu, and Wenzheng Xu. 2024. AoI-aware user service satisfaction enhancement in digital twin-empowered edge computing. *IEEE/ACM Transactions on Networking* 32, 2 (2024), 1677–1690. DOI: http://doi.org/10.1109/TNET.2023.3324704

[5] Lei Xie, Zihao Chu, Yi Li, Tao Gu, Yanling Bu, Chuyu Wang, and Sanglu Lu. 2023. Industrial vision: Rectifying millimeter-level edge deviation in Industrial Internet of Things With Camera-Based Edge Device. *IEEE Transactions on Mobile Computing* 23, 2 (2023), 1–17. DOI: http://doi.org/10.1109/TMC.2023.3246176

[6] Subhramoy Mohanti, Debashri Roy, Mark Eisen, Dave Cavalcanti, and Kaushik Chowdhury. 2024. L-NORM: L earning and n etwork o rchestration at the edge for r obot connectivity and m obility in factory floor environments. *IEEE Transactions on Mobile Computing* 23, 4 (2024), 2898–2914. DOI: http://doi.org/10.1109/TMC.2023.3266643

[7] Hui Li, Xiuhua Li, Qilin Fan, Qingyu Xiong, Xiaofei Wang, and Victor C. M. Leung. 2024. Transfer learning for real-time surface defect detection with multi-access edge-cloud computing networks. *IEEE Transactions on Network and Service Management* 21, 1 (2024), 310–323. DOI: http://doi.org/10.1109/TNSM.2023.3301718

[8] Wissal Attaoui, Essaid Sabir, Halima Elbiaze, and Mohsen Guizani. 2023. VNF and CNF placement in 5G: Recent advances and future trends. *IEEE Transactions on Network and Service Management* 20, 4 (2023), 4698–4733. DOI: http://doi.org/10.1109/TNSM.2023.3264005

[9] Wenbin Dai, Yingyue Zhang, Lingbo Kong, James H. Christensen, and Dan Huang. 2023. Design of industrial edge applications based on IEC 61499 microservices and containers. *IEEE Transactions on Industrial Informatics* 19, 7 (2023), 7925–7935. DOI: http://doi.org/10.1109/TII.2022.3214199

[10] Weiwen Zhang, Jinzhou Luo, Lei Chen, and Jianqi Liu. 2023. A trajectory prediction-based and dependency-aware container migration for mobile edge computing. *IEEE Transactions on Services Computing* 16, 5 (2023), 3168–3181. DOI: http://doi.org/10.1109/TSC.2023.3290023

[11] Chenghao Rong, Jessie Hui Wang, Jilong Wang, Yipeng Zhou, and Jun Zhang. 2024. Live migration of video analytics applications in edge computing. *IEEE Transactions on Mobile Computing* 23, 3 (2024), 2078–2092. DOI: http://doi.org/10.1109/TMC.2023.3246539

[12] Lionel Nkenyereye, Kang-Jun Baeg, and Wan-Young Chung. 2023. Deep reinforcement learning for containerized edge intelligence inference request processing in IoT edge computing. *IEEE Transactions on Services Computing* 16, 6 (2023), 4328–4344. DOI: http://doi.org/10.1109/TSC.2023.3320752

[13] Syed Danial Ali Shah, Mark A. Gregory, Shuo Li, Ramon Dos Reis Fontes, and Ling Hou. 2022. SDN-based service mobility management in MEC-enabled 5G and beyond vehicular networks. *IEEE Internet of Things Journal* 9, 15 (2022), 13425–13442. DOI: http://doi.org/10.1109/JIOT.2022.3142157

[14] Alexander Fuerst and Prateek Sharma. 2021. FaasCache: Keeping serverless computing alive with greedy-dual caching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, Virtual USA, 386–400. DOI: http://doi.org/10.1145/3445814.3446757

[15] Yangchuan Xu, Lulu Chen, Zhihui Lu, Xin Du, Jie Wu, and Patrick C. K. Hung. 2023. An adaptive mechanism for dynamically collaborative computing power and task scheduling in edge environment. *IEEE Internet of Things Journal* 10, 4 (2023), 3118–3129. DOI: http://doi.org/10.1109/JIOT.2021.3119181

[16] Chen Chen, Manuel Herrera, Ge Zheng, Liqiao Xia, Zhengyang Ling, and Jiangtao Wang. 2024. Cross-edge orchestration of serverless functions with probabilistic caching. *IEEE Transactions on Services Computing* 17, 5 (2024), 2139–2150. DOI: http://doi.org/10.1109/TSC.2024.3399651

[17] Xianzhong Tian, Huixiao Meng, Yifan Shen, Junxian Zhang, Yuzhe Chen, and Yanjun Li. 2024. Dynamic microservice deployment and offloading for things–edge–cloud computing. *IEEE Internet of Things Journal* 11, 11 (2024), 19537–19548.

[18] Zhenzheng Li, Jiong Lou, Jianfei Wu, Jianxiong Guo, Zhiqing Tang, Ping Shen, Weijia Jia, and Wei Zhao. 2024. Online container scheduling with fast function startup and low memory cost in edge computing. *IEEE Transactions on Computers* 73, 12 (2024), 2747–2760. DOI: http://doi.org/10.1109/TC.2024.3441836

[19] Tyler Harter, Brandon Salmon, Rose Liu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. Slacker: Fast distribution with lazy docker containers. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST 16)*. 181–195.

[20] Ludovico Funari, Luca Petrucci, and Andrea Detti. 2023. Storage-saving scheduling policies for clusters running containers. *IEEE Transactions on Cloud Computing* 11, 1 (2023), 595–607. DOI: http://doi.org/10.1109/TCC.2021.3104662

[21] Wei-Kuang Lai. 2023. Delay-aware container scheduling in kubernetes. *IEEE Internet of Things Journal* 10, 13 (2023), 11813–11824.

[22] Chiao-Cheng Chen, Yao Chiang, Yu-Chieh Lee, and Hung-Yu Wei. 2024. Edge computing management with collaborative lazy pulling for accelerated container startup. *IEEE Transactions on Network and Service Management* 21, 6 (2024), 1–1. DOI: http://doi.org/10.1109/TNSM.2024.3462408

[23] Dragi Kimovski, Narges Mehran, Christopher Emanuel Kerth, and Radu Prodan. 2022. Mobility-aware IoT application placement in the cloud - edge continuum. *IEEE Transactions on Services Computing* 15, 6 (2022), 3358–3371. DOI: http://doi.org/10.1109/TSC.2021.3094322

[24] Jiawei Zhang, Xiaochen Zhou, Tianyi Ge, Xudong Wang, and Taewon Hwang. 2021. Joint task scheduling and containerizing for efficient edge computing. *IEEE Transactions on Parallel and Distributed Systems* 32, 8 (2021), 2086–2100. DOI: http://doi.org/10.1109/TPDS.2021.3059447

[25] Muhammad Mudassar, Yanlong Zhai, and Liao Lejian. 2022. Adaptive fault-tolerant strategy for latency-aware IoT application executing in edge computing environment. *IEEE Internet of Things Journal* 9, 15 (2022), 13250–13262. DOI: http://doi.org/10.1109/JIOT.2022.3144026

[26] Sisi Li, Ao Zhou, Xiao Ma, Mengwei Xu, and Shangguang Wang. 2022. Commutativity-guaranteed docker image reconstruction towards effective layer sharing. In *Proceedings of the ACM Web Conference 2022*. ACM, Virtual Event, Lyon France, 3358–3366. DOI: http://doi.org/10.1145/3485447.3512154

[27] Luxiu Yin, Juan Luo, and Keqin Li. 2023. An optimal image storage strategy for container-based edge computing in smart factory. *IEEE Internet of Things Journal* 10, 8 (2023), 7204–7214. DOI: http://doi.org/10.1109/JIOT.2022.3229206

[28] Mingxiong Zhao, Xianqi Zhang, Zhenli He, Yu Chen, and Yunchun Zhang. 2024. Dependency-aware task scheduling and layer loading for mobile edge computing networks. *IEEE Internet of Things Journal* 11, 21 (2024), 34364–34381. DOI: http://doi.org/10.1109/JIOT.2024.3382682

[29] You Shi, Yuye Yang, Changyan Yi, Bing Chen, and Jun Cai. 2024. Toward online reliability-enhanced microservice deployment with layer sharing in edge computing. *IEEE Internet of Things Journal* 11, 13 (2024), 23370–23383.

[30] Zhiqing Tang, Jiong Lou, and Weijia Jia. 2023. Layer dependency-aware learning scheduling algorithms for containers in mobile edge computing. *IEEE Transactions on Mobile Computing* 22, 6 (2023), 3444–3459. DOI: http://doi.org/10.1109/TMC.2021.3139995

[31] Lele Ma, Shanhe Yi, Nancy Carter, and Qun Li. 2019. Efficient live migration of edge services leveraging container layered storage. *IEEE Transactions on Mobile Computing* 18, 9 (2019), 2020–2033. DOI: http://doi.org/10.1109/TMC.2018.2871842

[32] Mahdi Dolati, Seyed Hamed Rastegar, Ahmad Khonsari, and Majid Ghaderi. 2023. Layer-aware containerized service orchestration in edge networks. *IEEE Transactions on Network and Service Management* 20, 2 (2023), 1830–1846. DOI: http://doi.org/10.1109/TNSM.2022.3217134

[33] Shihong Hu, Zhihao Qu, Bin Tang, Baoliu Ye, Guanghui Li, and Weisong Shi. 2023. Joint service request scheduling and container retention in serverless edge computing for vehicle-infrastructure collaboration. *IEEE Transactions on Mobile Computing* 23, 6 (2023), 1–14. DOI: http://doi.org/10.1109/TMC.2023.3323524

[34] Ke Xiao, Song Yang, Fan Li, Liehuang Zhu, Xu Chen, and Xiaoming Fu. 2024. Making serverless not so cold in edge clouds: A cost-effective online approach. *IEEE Transactions on Mobile Computing* 23 (2024), 1–14. DOI: http://doi.org/10.1109/TMC.2024.3355118

[35] Fangyi Mou, Zhiqing Tang, Jiong Lou, Jianxiong Guo, Wenhua Wang, and Tian Wang. 2023. Joint Task scheduling and container image caching in edge computing. In *Proceedings of the 19th International Conference on Mobility, Sensing and Networking (MSN'23)*, 230–237.

[36] Alan Jay Smith. 1982. Cache memories. *ACM Computing Surveys* 14, 3 (1982), 473–530. DOI: http://doi.org/10.1145/356887.356892

[37] Jiong Lou, Hao Luo, Zhiqing Tang, Weijia Jia, and Wei Zhao. 2023. Efficient container assignment and layer sequencing in edge computing. *IEEE Transactions on Services Computing* 16, 2 (2023), 1118–1131. DOI: http://doi.org/10.1109/TSC.2022.3159728

[38] Yongquan Zhou, Yan Shi, Yuanfei Wei, Qifang Luo, and Zhonghua Tang. 2023. Nature-inspired algorithms for 0-1 knapsack problem: A survey. *Neurocomputing* 554, 5 (2023), 126630. DOI: http://doi.org/10.1016/j.neucom.2023.126630

[39] Hanshuai Cui, Zhiqing Tang, Jiong Lou, Weijia Jia, and Wei Zhao. 2024. Latency-aware container scheduling in edge cluster upgrades: A deep reinforcement learning approach. *IEEE Transactions on Services Computing* 17, 5 (2024), 2530–2543. DOI: http://doi.org/10.1109/TSC.2024.3394689

[40] Henrik I. Christensen, Arindam Khan, Sebastian Pokutta, and Prasad Tetali. 2017. Approximation and online algorithms for multidimensional bin packing: A survey. *Computer Science Review* 24, 4 (2017), 63–79. DOI: http://doi.org/10.1016/j.cosrev.2016.12.001

[41] Sara Ali, António Galrão Ramos, Maria Antónia Carravilla, and José Fernando Oliveira. 2024. Heuristics for online three-dimensional packing problems and algorithm selection framework for semi-online with full look-ahead. *Applied Soft Computing* 151, 41 (2024), 111168. DOI: http://doi.org/10.1016/j.asoc.2023.111168

[42] Mahmud Parvez, Pratik J. Parikh, Faisal Aqlan, and Md. Noor-E-Alam. 2024. An online dynamic dual bin packing with lookahead approach for server-to-cell assignment in computer server industry. *Computers and Industrial Engineering* 196, 24 (2024), 110404. DOI: http://doi.org/10.1016/j.cie.2024.110404

[43] Christopher J. C. H. Watkins and Peter Dayan. 1992. Q-learning. *Machine Learning* 8, 3 (1992), 279–292. DOI: http://doi.org/10.1007/BF00992698

[44] Hado Van Hasselt, Arthur Guez, and David Silver. 2016. Deep reinforcement learning with double q-learning. *Proceedings of the AAAI Conference on Artificial Intelligence* 30, 1 (2016), 2094–2100. DOI: http://doi.org/10.1609/aaai.v30i1.10295

Received 31 January 2025; revised 19 October 2025; accepted 13 November 2025