Container Scheduling Strategy Based on Image Layer Reuse and Sequential Arrangement in Mobile Edge Computing

Haijie Wu, Weiwei Lin, *Senior Member, IEEE*, Haotong Zhang, Fang Shi, Wangbo Shen, Keqin Li, *Fellow, IEEE*, and Albert Y. Zomaya, *Fellow, IEEE*

***Abstract*—In Mobile Edge Computing (MEC) scenarios, computational tasks are popularly deployed using containerization to isolate the runtime environment. To complete the execution of the task, the edge server first pulls the image, then instantiates and runs the container. Since it takes a lot of time for the edge server to download the image from the cloud, image reuse reduces the pulling latency significantly. However, the limited storage capacity of edge servers hinders image reuse. Recent works have enhanced reuse efficiency by leveraging the hierarchical structure of images and caching high-value layers. However, their efficiency remains limited due to the lack of multi-container collaboration. This paper proposes a novel container scheduling strategy based on image layer reuse and sequence arrangement (ILR-SA) for MEC scenarios, which achieves efficient scheduling by collaborating multiple containers. First, containers are greedily deployed into the edge cluster. Then, the execution sequence of containers is modeled as an optimal Hamiltonian path problem, efficiently solved by our proposed decomposition algorithm. Finally, an efficient image layer update strategy is used to achieve layer reuse. We conduct rigorous experiments to demonstrate that our proposed container scheduling strategy reduces the computational task completion time by up to 91.3% compared to existing approaches.**

***Index Terms*—Container scheduling, hierarchical structure of image, image layer reuse, mobile edge computing.**

Received 24 December 2024; revised 24 February 2025; accepted 27 March 2025. Date of publication 2 April 2025; date of current version 6 August 2025. This work was supported in part by Guangxi Key Research and Development Project under Grant 2024AB02018, in part by the Guangdong Provincial Natural Science Foundation Project under Grant 2025A1515010113, in part by Guangzhou Development Zone Science and Technology Project under Grant 2023GH02, and in part by the Major Key Project of PCL, China under Grant PCL2023A09. Recommended for acceptance by S. Abadal. *(Corresponding author: Weiwei Lin.)*

Haijie Wu and Wangbo Shen are with the School of Computer Science and Engineering, South China University of Technology, Guangzhou 510006, China (e-mail: cswuhaijie@mail.scut.edu.cn; 202010107337@mail.scut.edu.cn).

Weiwei Lin is with the School of Computer Science and Engineering, South China University of Technology, Guangzhou 510006, China, and also with Pengcheng Laboratory, Shenzhen 518066, China (e-mail: linww@scut.edu.cn).

Haotong Zhang is with the School of Software Engineering, South China University of Technology, Guangzhou 510006, China (e-mail: sezhanght@mail.scut.edu.cn).

Fang Shi is with the College of Mathematics and Informatics, South China Agricultural University, Guangzhou 510642, China (e-mail: csshifang@scau.edu.cn).

Keqin Li is with the Department of Computer Science, State University of New York, New Paltz, NY 12561 USA (e-mail: lik@newpaltz.edu).

Albert Y. Zomaya is with the School of Computer Science, The University of Sydney, Sydney, NSW 2006, Australia (e-mail: albert.zomaya@sydney.edu.au).

Digital Object Identifier 10.1109/TMC.2025.3557160

I. Introduction

In Mobile Edge Computing (MEC) scenarios [1], containerization has become a key technology for deploying applications [2]. By encapsulating applications and their dependent environments in containers, containerization ensures the consistency and portability of applications across different computing environments, thus improving application development efficiency. Due to the distributed characteristics of MEC and its need for low latency, applications need to be managed efficiently and deployed and launched quickly, and these requirements are met by containerization, thus containerization plays a crucial role in MEC.

As one of the most popular containerization tools, Docker is widely used to provide lightweight deployment of applications or computing tasks. In the review [3], the authors revealed that containerization is considered a substitute for virtualization, and Docker has become the most popular tool among container vendors for deploying software applications or services, indicating that Docker containers have gained a huge market share. Furthermore, the Docker containers are orchestrated through some tools to achieve efficient management [4]. For example, Kubernetes has the ability to manage and automate the deployment of large-scale Docker container clusters and provides a high availability and scalability solution [5]. Different container orchestration tools have different security, stability, and scalability [6], but they all provide unified management of containers and achieve distributed application management. These orchestration tools have been widely applied in different fields. For example, Kubernetes is used to build a blockchain [7] that can predict health diseases. Due to Kubernetes' ability to automate the deployment, scaling, and management of containerized applications, it will facilitate container orchestration and efficient blockchain deployment. A cloud-edge collaborative architecture-based industrial robot platform was proposed in [8], and container orchestration technology was used to facilitate the deployment of cloud-edge collaborative services. The paper [9] implemented a testing platform in a Kubernetes environment to generate a dataset of 5G network resources that is closer to real-world data for deep learning model training, and evaluated the best performance model for predicting resource usage. These studies fully demonstrate the high application value of containers and their orchestration tools.

The container depends on an image, a static file containing the initial file system, which is the foundation for the container to run. Before starting the container, the device needs to pull the corresponding image file for the container. When the image file is missing, the device will pull the image from a remote image repository, such as DockerHub. The speed of pulling is affected by network bandwidth, so storing the image on the local disk for reuse can avoid the latency caused by pulling the image. However, in the MEC scenario, edge devices' storage capacity and network bandwidth are limited, resulting in the image-pulling delay becoming a challenge that cannot be ignored. Kubernetes provides a default scheduler that takes into account the latency of image pulls. This default scheduler assigns a higher score to nodes that already have the corresponding image in their local image repository when selecting nodes for containers. Thus, the node's image repository is used as a metric for scheduling consideration along with computational resources. Some works reduce the pulling latency of images by pre-placing them on edge devices. For example, Sun et al. proposed an image-based microservice placement algorithm that places microservice images on edge devices based on the frequency of microservice requests [10]. As images with higher usage frequency are more likely to be used in the future, this method will effectively reduce the image retrieval time. Some works are optimized from a caching perspective, which means that during the scheduling process, the images on edge devices will be updated based on their value. Limited storage capacity will prioritize retaining the most valuable images, while the least valuable images will be evicted because they have the lowest possibility of being reused. Mou et al. considered the image cache on edge devices in the process of using reinforcement learning for task scheduling optimization, and the value of the image was defined as the product of the frequency of use and the size of the image [11]. These works have fundamental significance for MEC scheduling optimization.

However, the method of reusing the entire image has its limitations, as once there are too many different containers, their images are considered entirely different, and the efficiency of reuse will be significantly degraded. In fact, images often have a hierarchical structure, forming a tree structure if these images all develop from the same basic image [12]. Therefore, the reuse of the image layer will bring greater pull latency optimization when compared to the entire image, as two nonidentical images may have the same image layers. Many works utilize the hierarchical structure of images to optimize scheduling [13], [14], where containers tend to be scheduled to devices with more corresponding image layers. The caching of the image layer is also a popular research topic [12], [15], [16]. When the storage space of edge devices is insufficient, managing the image layer correctly can help improve the scheduling efficiency of containers.

While these works alleviate the image-pulling latency, the lack of multi-container collaboration results in limited container scheduling performance. When co-scheduling a batch of containers, the scheduler can obtain more prior knowledge [17], [18], [19], with greater optimization space, and the corresponding solution becomes the NP-hard problem [20], [21]. Since batch scheduling is a significant challenge in MEC [18], [19], [22], it is significant to study how to achieve efficient collaborative scheduling of multiple containers by using the hierarchical structure of images. However, existing works on container scheduling in MEC scenarios do not work well by image layer reuse to synergize the scheduling of multiple containers, resulting in excessive image pulling latency and container completion time, and aggravating the burden on the edge network.

In this paper, we innovatively propose an efficient container scheduling algorithm based on Image Layer Reuse and Sequential Arrangement (ILR-SA). We collaborate with multiple containers and reduce the pulling latency of images and the total completion time of containers by reusing image layers. ILR-SA considers the constraints of computational resources and disk space, heuristically deploys containers to edge devices, and reuses the image layer by caching and updating the image layer dynamically. By sequential arrangement of containers, the reusing rate of the image layer is greatly enhanced, which alleviates the bandwidth burden and reduces the response time in MEC scenarios. ILR-SA can quickly obtain scheduling strategies with a small computational burden. The main contributions of this paper are summarized as follows:

1) We address the container scheduling problem from the perspective of multiple containers collaborating and reducing image-pulling latency through image layer reuse. Since multiple containers can collaborate, we focus on maximizing the reuse rate of the containers' image layers, thereby reducing the completion time of the containers.
2) We design a scheduling algorithm for containers, named ILR-SA. First, containers are greedily deployed to edge nodes to wait for further processing. Then, we model the execution sequence of multiple containers on a single edge node as the optimal Hamiltonian path, since containers with similar image layers should be run adjacent to each other. We design a fast and efficient method based on the idea of decomposition and justify the method theoretically. Finally, the optimal image layer caching strategy is proposed to improve the reuse efficiency of containers running on the edge device.
3) We have conducted rigorous comparative experiments to fully validate that ILR-SA can significantly reduce the container completion time and improve the image layer reuse rate within a very short algorithm execution time.

The rest of this paper is organized as follows: Section II introduces the background and related work. Section III formally describes the scheduling problem of this paper. Section IV introduces the design and analysis of the proposed method. Section V conducts the experiments and analyzes the results. Section VI concludes this paper.

II. Background and Related Work

A. Container Scheduling

Containers, with their independent runtime environment, are very popular tools for deploying applications. However, the scheduling performance of containers determines whether applications can be deployed quickly or computational tasks can be completed promptly. When the image pulling time is not considered as the bottleneck of scheduling performance, the device's performance becomes a key indicator for container scheduling. Lai et al. proposed a delay-aware container scheduling (DACS) algorithm to achieve container scheduling on heterogeneous devices in edge clusters [23]. Due to the heterogeneity of edge devices, worker nodes exhibit varying performance, resulting in different computational resources and latency for different worker nodes. DACS integrates these indicators into scheduling algorithms to achieve container scheduling on heterogeneous clusters by predicting remaining resources and latency. Tang et al. proposed a priority-aware greedy container scheduling strategy (PGT) to optimize the container scheduling performance in cloud-edge collaborative environments, using response time, energy consumption, and task execution cost as indicators to be optimized [24]. A novel network-aware framework for Kubernetes was proposed by [25] to determine the placement of microservices, focusing on reducing end-to-end latency of applications and saving bandwidth. Wang et al. proposed a new load prediction model CNN-BiGRU-Attention to predict the future load of containers and generate scheduling strategies to cope with changes in load [26]. These tasks are often based on the assumption that the image has already been pre-pulled to the worker node, without considering the network burden and latency caused by image pulling, and only targeting the data transmission and resource requirements of the container itself.

B. Batch Task Scheduling for Edge Servers

In the MEC scenario, tasks are required to have a small latency, resulting in tasks often only being scheduled with online processing, as the tasks cannot tolerate a period of latency to accumulate a batch of tasks for simultaneous scheduling. However, this tolerance may decrease in scenarios where computing resources or bandwidth are very limited, as scheduling multiple tasks collaboratively can bring better scheduling performance, making online processing secondary. Miao et al. pointed out that most existing schedulers are queue-based, meaning that tasks are scheduled sequentially and lack tightly coupled joint processing, and thus, time-saving batch scheduling methods are needed [17]. Due to the collaborative processing of multiple tasks, scheduling problems can be modeled as a min-cost max-flow problem, which can achieve more efficient scheduling results at a relatively low deployment cost. A similar viewpoint is illustrated in [19]. The authors specify that collaborative scheduling of multiple containers is superior to independent scheduling of containers, which has been overlooked in existing work. Therefore, they used multiple collaborating containers and proposed methods for container allocation and image layer sequence sorting, significantly reducing the startup delay of containers. Xu et al. proposed an adaptive mechanism for dynamic collaborative computing capability and task scheduling (ADCS) in edge environments to reduce deadline loss rate and task completion time [27]. ADCS processes the tasks generated by mobile devices and finds better scheduling strategies than sequential scheduling by adjusting the execution order between tasks, an effect that online scheduling cannot achieve. Teng et al. proposed a multi-server multi-task allocation and scheduling (MMAS) problem for MEC scenarios, scheduling a batch of tasks offloaded from the edge layer, and proposed the Game based Distributed Task Allocation and Scheduling (GDTAS) scheme and the Centralized Greedy Task Allocation and Scheduling (CGTAS) scheme to maximize system profits [18]. These works fully demonstrate the importance of batch task scheduling in MEC scenarios. Through the collaborative scheduling of multiple tasks, the scheduling performance of tasks can be significantly improved.

C. Container Reuse and Image Layer Reuse

To reduce the latency of deploying containers on edge devices, some works reuse containers to reduce their startup latency, including image pulling. Chen et al. proposed an optimized request distribution algorithm and context-aware probabilistic container caching strategy, which can significantly reduce the cost of container deployment by preventing frequent creation and destruction of containers [28]. Pan et al. presented a similar idea, investigating the problem of retention-aware container caching in serverless edge computing and reducing the overall cost of the system [29]. Some works are based on reinforcement learning to design container caching and reuse algorithms [30], [31], and some use heuristic algorithms [10], [32]. These works inspired the idea of reuse and showed the optimizations that come with reuse. However, they focused more on the startup latency of the container itself rather than just the image pull time. When bandwidth resources are more scarce, considering how to reduce the image pull time will lead to more significant optimization.

To alleviate the shortage of bandwidth resources as much as possible, the image is divided into layers to improve the reuse rate since two similar images will not be considered as the same image but have many identical reusable image layers. Yin et al. proposed a two-stage optimization storage strategy to reduce the download time of images [12]. The value of the image layer is first defined, and then the pre-placed image layers are determined through the knapsack algorithm. Subsequently, during the container scheduling process, the image layers are continuously updated based on their usage frequency. A layer-wise container pre-arming and keep-alive technique, RainbowCake, was proposed by [33]. Through structured container layering and shared-aware modeling, RainbowCake has robustness and tolerance for call bursts, thereby reducing container startup latency. In [34], the service request scheduling and container reuse problem with layer sharing and container caching are studied to reduce latency in vehicular services and ensure responsiveness. Zhao et al. also studied the loading strategy of the image layer in storage constraints to reduce the total computation completion time of tasks [35]. Much work has shown that reusing the image layer can bring significant optimization, especially in edge environments with limited storage capacity and bandwidth.

III. Problem Formulation

To clearly state the impact of image pull latency on container scheduling, we model container scheduling in edge scenarios

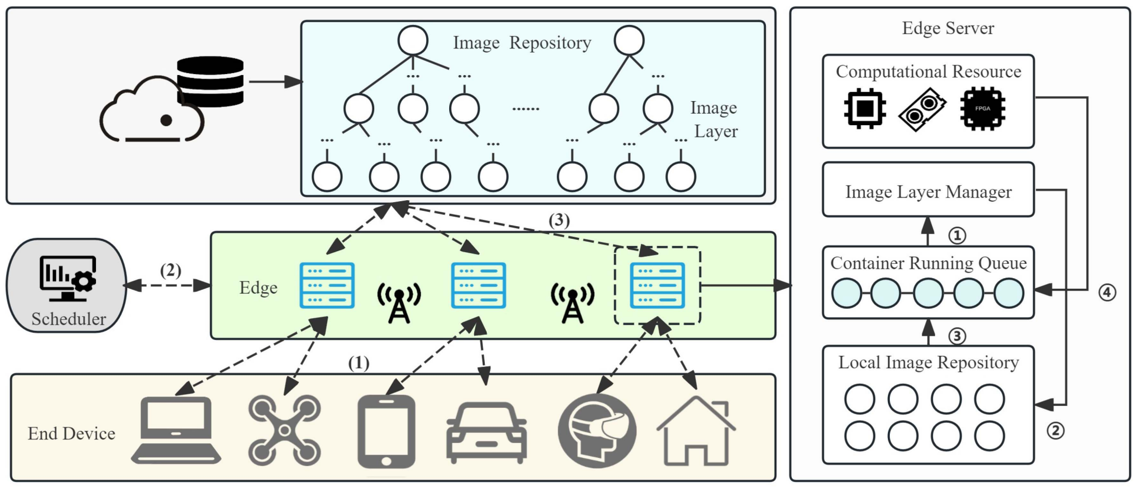

Fig. 1. Illustration of the container scheduling process in the MEC scenarios. Process (1): End devices send task requests and communicate with edge devices during container execution. Process (2): The scheduler collects task information from edge devices and sends scheduling strategies. Process (3): Edge devices pull image layers from the cloud. In edge devices, ① container queue information is sent to the image layer manager, ② the manager caches layers or pulls image layers, and ③ the local image repository acquires all necessary image layers, ④ enabling container execution using edge device computational power.

TABLE I
KEY NOTATIONS DEFINITION

Notations	Descriptions
N, M, R, W	Number of tasks, edge devices, resource types, and all image layers
C	Set of all containers
E	Set of all edge devices
Q_i^r	Resource requirements of containers
A_j^r	Resources of edge devices
L	Set of all image layers
$l_{i,k}$	Binary variable representing the image layer required by the container
s_k	Size of image layer
d_j	Disk size in the edge device for storing the image layer
$\Upsilon_{j,k}$	Binary variable representing the image layer pulled locally
γ_j	Bandwidth of edge devices
$\mathcal{H}$	Scheduling strategy
h_i^j	Container running on the specified edge device
$t^{\triangle}_{h_i^j}$	Completion time of a container
$t^w_{h_i^j}, t^{\downarrow}_{h_i^j}, t^r_{h_i^j}$	Times for waiting, downloading images, and executing tasks
ACT, AMS	Average total completion time and average makespan
α	Weight factor for ACT and AMS

and formally describe the scheduling model using mathematical symbols.

The overall container scheduling process is shown in Fig. 1, and the key notations are defined as shown in Table I. First, the scheduler collects requests for tasks from the end devices, which are packaged as containers. We assume that the scheduler receives N tasks at a certain time slot, so there are N containers that need to be scheduled for the execution of these tasks, denoted as $C = \{c_1, c_2, \ldots, c_N\}$. The execution of these tasks depends on computational resources such as CPU, memory, etc. We assume that a total of R types of computational resources are considered, where Q_i^r denotes the number of requests from the ith container for the rth computational resource, where $r = 1, 2, \ldots, R$ and $i = 1, 2, \ldots, N$. We assume that these containers involve W different image layers, represented as $L = \{L_1, L_2, \ldots, L_W\}$. The binary variable $l_{i,k} = 1$ indicates that for the ith container, its image contains the kth image layer, while $l_{i,k} = 0$ indicates that it does not. The size of these W image layers is represented as $\{s_1, s_2, \ldots, s_W\}$, so the ith container has an image size of $\sum_{k=1}^{W} l_{i,k} s_k$. The scheduler calculates the scheduling strategy and deploys containers to M edge devices in the edge cluster, denoted as $E = \{e_1, e_2, \ldots, e_M\}$. A_j^r represents the rth type of computational resource quantity for the jth edge device. Thus, for container c_i executing on e_j, the resource constraint is represented as:

$$Q_i^r \leq A_j^r, r = 1, 2, \ldots, R \tag{1}$$

Specifically, the disk size used for storing the image layers is separately noted as $\{d_1, d_2, \ldots, d_M\}$, as the disks of edge devices cannot be used entirely for storing the image layer. The binary variable $\Upsilon_{j,k} = 1$ indicates that the jth edge device has already stored the kth image layer, while $\Upsilon_{j,k} = 0$ indicates that it has not been pulled locally. The bandwidth of edge devices is denoted as $\{\gamma_1, \gamma_2, \ldots, \gamma_M\}$. The local image repository capacity of edge devices is limited. Therefore, the following restrictions are met:

$$\sum_{k=1}^{W} \Upsilon_{j,k} \times s_k \leq d_j, j = 1, 2, \ldots, M \tag{2}$$

We define the scheduling strategy $\mathcal{H} = \{\mathcal{H}_1, \mathcal{H}_2, \ldots, \mathcal{H}_M\}$, where $\mathcal{H}_j$ denotes the containers of the tasks allocated to the jth edge device, thus $\mathcal{H}_j = \{h_1^j, h_2^j, \ldots, h_{|\mathcal{H}_j|}^j\}$, where $h_i^j \in C, j = 1, 2, \ldots, M, i = 1, 2, \ldots, |\mathcal{H}_j|, \mathcal{H}_j \cap \mathcal{H}_k = \varnothing, \sum_{j=1}^{M} |\mathcal{H}_j| = N$. The scheduling strategy contains not only the mapping of tasks to edge devices but also the startup sequence of multiple containers on the same edge device since the startup sequence will greatly affect the reuse efficiency of images. Then, the edge device receives the scheduling strategy sent by the scheduler and will start executing these tasks sequentially. In jth edge device, an image layer manager is deployed to implement the update strategy for the image layer.

The completion time of container h_i^j is defined as $t^{\triangle}_{h_i^j}$, which includes a waiting time $t^{w}_{h_i^j}$ for the container, a downloading time $t^{\downarrow}_{h_i^j}$ for the image, and a time $t^{r}_{h_i^j}$ for the task to run, i.e.,

$$t^{\triangle}_{h_i^j} = t^{w}_{h_i^j} + t^{\downarrow}_{h_i^j} + t^{r}_{h_i^j} \tag{3}$$

where $t^{w}_{h_i^j}$ is the waiting delay caused by the queuing of the containers on the edge device according to the sequence on $\mathcal{H}_j$, which depends on the resource limitation of the edge device, and the container stops waiting only if the image is available or the edge device does not pull other images. When the image pull is complete, it will continue to wait if the computational resources are insufficient. $t^{\downarrow}_{h_i^j}$ is calculated by the following equation:

$$t^{\downarrow}_{h_i^j} = \frac{\sum_{k=1}^{W} l_{h_i^j,k} s_k (1 - \Upsilon_{j,k})}{\gamma_j} \tag{4}$$

It is worth noting that the image layer manager updates the image layer of the edge device. When the capacity of the image repository is insufficient, some of $\Upsilon_{j,k}$ is varied since the capacity of the image repository is limited. The image layers not being used will be deleted [12], [15]. $t^{r}_{h_i^j}$ is determined by the contents of the task and contains the time of computation on the edge servers, data transfer with the end devices, and so on. Other delays, such as the transmission delay of the scheduling strategy from the scheduler to the edge device, are to be omitted since they are much smaller than the above three delays. We define the average total completion time ACT of a set of containers as the average of the completion times of all N containers:

$$ACT = \sum_{i=1}^{N} t_i^{\triangle}/N = \sum_{j=1}^{M} \sum_{i=1}^{|\mathcal{H}_j|} t^{\triangle}_{h_i^j}/N \tag{5}$$

Since each edge device handles a set of containers individually, we define the average makespan AMS of the containers to be the average of the makespan of M devices:

$$AMS = \sum_{j=1}^{M} \max_{h_i^j \in \mathcal{H}_j} t^{\triangle}_{h_i^j}/M \tag{6}$$

Our goal is to optimize ACT and AMS by improving the reuse efficiency of images, formalized as:

$$\begin{aligned} \min \quad & \alpha \times ACT + (1-\alpha) \times AMS \\ \text{s.t.} \quad & \text{Eqs. (1), (2).} \end{aligned} \tag{7}$$

where α is a weighting factor to weigh ACT and AMS. ACT represents the average execution efficiency of each task, while AMS represents the execution efficiency of the overall task queue. In general, ACT and AMS are positively correlated, as lower ACT tends to result in lower AMS. These two metrics are the main basis for evaluating our scheduling strategy, and we weigh them to achieve a joint optimization.

IV. Design of ILR-SA

In this section, we detail the proposed ILR-SA. ILR-SA focuses on optimizing the image pull time before the container runs. By analyzing the hierarchical structure of images and arranging the execution sequence of containers appropriately, the image layer reuse efficiency for multiple containers executing on the same device can be improved, thereby reducing container completion time. As shown in Fig. 2, we divide the proposed scheduling strategy into three phases. In the first phase, the containers are properly deployed to a specific edge device. In the second phase, the execution sequence of the containers deployed to the same device is arranged, as containers with more identical image layers are more suitable for adjacent execution. In the third phase, we run these containers on devices based on the calculated execution sequence and design an update strategy for the image layer to improve the image layer reuse efficiency, thereby reducing the completion time of the containers.

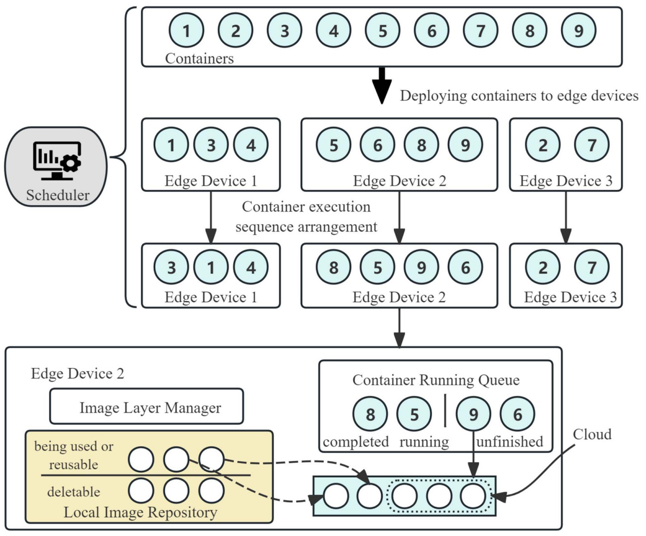

Fig. 2. Schematic diagram of the ILR-SA. Containers are first deployed to the edge devices, then the execution sequence is arranged, and finally, the image layer reuse is realized using the update strategy.

A. Deploying Containers to Nodes

To minimize the completion time, containers tend to be deployed to the devices with the most remaining resources. At the same time, the image layer repository also determines the time a container spends on pulling images. Based on these two points, we design a scheduling algorithm for containers to achieve the optimization objective of (7). In this phase, the mapping of containers to edge devices is determined, which means that the elements of each H_j are determined, but not the sequence of these elements. The completion time of the containers on the edge device can be computed in advance because both the resource consumption of the container and the resources of the edge device are known. Therefore, we define the function $f(\cdot)$ that calculates the average completion time $\widehat{ACT}$ and makespan $\widehat{AMS}$ of an edge device when a set of containers is deployed on it. During the deployment process, the containers are deployed to the edge cluster E one by one in the sequence of C. The random eviction strategy is adopted to update image layers

Algorithm 1: Scheduling Algorithm for Deploying Containers to Edge Devices.

```
Data: Containers to be deployed $C$ and edge devices $E$.
Result: $\mathcal{H}$.
$\mathcal{H} = \{\mathcal{H}_1, \mathcal{H}_2, ..., \mathcal{H}_M\}, \mathcal{H}_j \leftarrow \{\}, \forall e_j \in E$;
for $c_i \in C$ do
    $best\_id \leftarrow 0, v_{best} \leftarrow +\infty$;
    for $e_j \in E$ do
        $\widehat{ACT}, \widehat{AMS} = f(e_j, \mathcal{H}_j \cup \{c_i\})$;
        if $v_{best} > \alpha \times \widehat{ACT} + (1-\alpha) \times \widehat{AMS}$ then
            $v_{best} \leftarrow \alpha \times \widehat{ACT} + (1-\alpha) \times \widehat{AMS}$;
            $best\_id \leftarrow j$;
    $\mathcal{H}_{best\_id} \leftarrow \mathcal{H}_{best\_id} \cup \{c_i\}$;
return $\mathcal{H}$
```

when the capacity of the repository is insufficient. The designed scheduling algorithm is shown in Algorithm 1.

In Algorithm 1, each container is sequentially deployed to edge devices, and the average completion time and makespan of the edge device are calculated when the container is deployed to the edge device. Based on this, the optimal edge device is selected. For Algorithm 1, we have the following two notes: 1) The scheduling algorithm for containers is coarse-grained, and the process of calculating scheduling strategies is seen as simulated scheduling, which means that containers do not actually run on edge clusters but only calculate the total completion time and makespan. When deploying c_i to the edge cluster, the deployment of $c_{i+1}, \ldots, c_N$ is unknown, so the execution sequence of the container will not be adjusted, and the eviction of the image layer is also random. However, since containers always choose the minimum average completion time and makespan, they will select edge devices with more abundant resources and higher image reuse rates. This decision helps with the subsequent sequence arrangement and image layer updates. 2) The design of scheduling is necessary. To improve the reuse efficiency of the image layer, similar containers will be deployed to the same edge device as much as possible. However, due to the resource limitations of edge devices, this deployment may not be effective because the idle resources of other edge devices also need to be considered. Additionally, deploying multiple containers on edge devices first simplifies the problem. The subsequent design focuses on efficiently executing a set of containers on only an edge device.

B. Container Execution Sequence Arrangement

In this phase, the execution sequence of containers deployed to the same edge device is arranged. For $\forall e_j \in E$, $\mathcal{H}_j = \{h_1^j, h_2^j, \ldots, h_{|\mathcal{H}_j|}^j\}$ will be processed using the same algorithm. It is computationally infeasible to find the optimal sequence by traversing all of $|\mathcal{H}_j|!$ different sequences. Therefore, we propose the container execution sequence arrangement algorithm based on the image layer reuse, which reduces the image pulling time and achieves the goal of (7).

Specifically, denoting ψ_{c_i,c_q} as the amount of layer data duplicated by containers c_i and c_q, thus

$$\psi_{c_i,c_q} = \sum_{k=1}^{W} l_{i,k} \times l_{q,k} \times s_k \tag{8}$$

If c_i and c_q are adjacently executed, the duplicated data will be reused because when c_i is executed, e_j owns all the image layers of c_i. Therefore, the image layers that can be reused by c_q will not be evicted when the image layers are updated. For the container queue $\mathcal{H}_j$, we define the sequence value V to be calculated by the following equation:

$$V = \sum_{i=1}^{|\mathcal{H}_j|-1} \psi_{h_i^j, h_{i+1}^j} \tag{9}$$

where V denotes the minimum amount of image layer reuse data for that container queue. An optimal container execution sequence arrangement will have a maximum V as it will be able to increase the amount of reused data. To compute the container queue with maximum V, we model the execution sequence of containers as the optimal Hamiltonian of a graph. Specifically, we define a complete undirected graph G with $\mathcal{N} = |\mathcal{H}_j|$ nodes, each node represents a container, and the edge of the ith node and the qth node is $\psi_{h_i^j, h_q^j}$, which means that if h_i^j and h_q^j are executed adjacent to each other, the amount of data that can be reused is $\psi_{h_i^j, h_q^j}$. Therefore, solving for a queue of containers with maximum V is to find a Hamiltonian path P with the maximum sum of edge weights in G. Since the Hamiltonian path requires that the path passes through all the nodes only once, the sequence in which this path passes through the nodes is the sequence in which the containers are executed.

This problem is similar to the Traveling Salesman Problem (TSP), which is an NP-hard problem. Dynamic programming can solve for the optimal Hamiltonian path according to the following equation:

$$D[S][h_t] = \max_{h_o \in S \setminus \{h_t\}} \left(D\left[S \setminus \{h_t\}\right]\left[h_o\right] + \psi_{h_o,h_t}\right) \tag{10}$$

where $h_t \in S$, and $D[S][h_t]$ represents the maximum V of the path, which passes through node set S and terminates at node h_t. The state space of S comprises $2^{\mathcal{N}}$ distinct states, thus the time complexity of solving the Hamiltonian path with the maximum V is $\mathcal{O}(\mathcal{N}^2 \times 2^{\mathcal{N}})$. Although the Hamiltonian path solved by this method is optimal, the computation time cannot be tolerated with an increasing number of nodes. Therefore, we propose a fast algorithm for finding a suboptimal solution to search for the Hamiltonian path. Then, we will explain that our design is based on the tree structure of the image layer and has good performance.

The container execution sequence arrangement algorithm we designed is based on (10). $\mathcal{N}$ determines the speed of finding the optimal Hamiltonian path. When $\mathcal{N}$ is small, the algorithm can calculate the optimal Hamiltonian path within an acceptable time. Therefore, as shown in Fig. 3, our method utilizes the idea of decomposition to divide the $\mathcal{N}$ nodes into multiple groups,

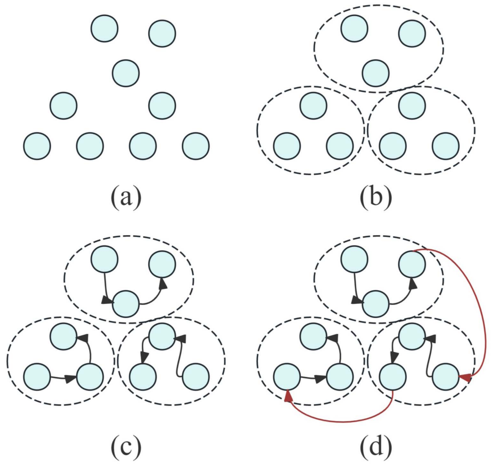

Fig. 3. Schematic diagram of the decomposition algorithm we proposed. Each node represents a container, and the path represents the execution sequence of these containers. From (a) to (b), nodes are first grouped. From (b) to (c), the optimal Hamiltonian path for each group is determined. From (c) to (d), these paths are connected via the optimal Hamiltonian path.

each with $\mathcal{N}_o$ nodes, resulting in a total of $\lfloor \frac{\mathcal{N}}{\mathcal{N}_o} \rfloor$ groups. Each group can use (10) to calculate the optimal Hamiltonian path, then each path is viewed as a point again, with its starting and ending points connected to other paths. Therefore, (10) can be used again to solve the connection of these $\lfloor \frac{\mathcal{N}}{\mathcal{N}_o} \rfloor$ paths. The time complexity of this method is $\mathcal{O}(\mathcal{N} \times \mathcal{N}_o \times 2^{\mathcal{N}_o} + \frac{\mathcal{N}^2}{\mathcal{N}_o^2} \times 2^{\frac{\mathcal{N}}{\mathcal{N}_o}})$. Intuitively, to minimize time complexity, $\mathcal{N}_o$ need to take $\sqrt{\mathcal{N}}$, since the exponential term can become a bottleneck in time complexity, and making $\frac{\mathcal{N}}{\mathcal{N}_o} = \mathcal{N}_o$ is optimal. Therefore, the time complexity is optimized to $\mathcal{O}(\mathcal{N}^{\frac{3}{2}} \times 2^{\sqrt{\mathcal{N}}})$. Although the algorithm is still not non-polynomial in complexity, it can handle more containers compared to (10). Moreover, the effectiveness of this algorithm will be explained in the following content, and the extended algorithm will be introduced to cope with larger container queues. The algorithm for arranging the container sequence proposed is shown in Algorithm 2.

Algorithm 2 processes the containers $\mathcal{H}_j$ deployed on the edge device e_j and outputs the sorted execution sequence of these containers. Containers are first grouped, selecting the first container from the ungrouped ones. Then, the container is grouped with the $\lfloor\sqrt{\mathcal{N}}\rfloor - 1$ containers having the largest edge weights among the ungrouped containers. The effectiveness of this grouping method will also be demonstrated in subsequent content. Then, the containers in the same group find the optimal Hamiltonian path. The two endpoints of this path are determined as the starting and ending points, respectively. After grouping all containers of $\mathcal{H}_j$, g paths are obtained, and these g paths form a complete directed graph by treating them as nodes. In the end, these g nodes obtained the optimal Hamiltonian path, thereby connecting the g paths and obtaining the final path P^*. For our proposed algorithm, there are the following issues that need to be explained: 1) Is there a significant discrepancy between the path calculated by the proposed method and the optimal solution? 2) What is the basis for grouping?

Algorithm 2: Algorithm for Container Execution Sequence Arrangement.

```
Data: $\mathcal{H}_j$.
Result: $\mathcal{H}_j^*$.
$\mathcal{N} \leftarrow |\mathcal{H}_j|, \mathcal{N}_o \leftarrow \lfloor\sqrt{\mathcal{N}}\rfloor$;
Calculate $\Psi = \{\psi_{h_i^j, h_q^j} | h_i, h_q \in \mathcal{H}_j\}$ and then
  construct graph $G = (\mathcal{H}_j, \Psi)$;
$g \leftarrow 1$;
while $\mathcal{H}_j$ is not empty do
    $\mathcal{G}_g \leftarrow \varnothing$;
    Select the first element $h_o$ in $\mathcal{H}_j$;
    $\mathcal{G}_g \leftarrow \mathcal{G}_g \cup \{h_o\}$;
    $\mathcal{H}_j \leftarrow \mathcal{H}_j \setminus \{h_o\}$;
    for $i \leftarrow 1$ to $\min(\mathcal{N}_o - 1, |\mathcal{H}_j|)$ do
        Select the node $h_e$ with the highest edge
          weight connected to node $h_o$ in $\mathcal{H}_j$;
        $\mathcal{G}_g \leftarrow \mathcal{G}_g \cup \{h_e\}$;
        $\mathcal{H}_j \leftarrow \mathcal{H}_j \setminus \{h_e\}$;
    Using the nodes and corresponding edges in $\mathcal{G}$,
      calculate the Hamiltonian path $P_g$ that maximizes
      V according to Eq. (10);
    $g \leftarrow g + 1$;
Regard $P_1, P_2, ..., P_g$ as $g$ nodes, and the edge weight
  from $P_i$ to $P_q$ is the edge weight between the
  endpoint of $P_i$ and the starting point of $P_q$;
Calculate the Hamiltonian path $P^*$ with the maximum
  $V$ for this newly constructed complete directed graph
  according to Eq. (10);
Obtain the execution sequence of the containers $\mathcal{H}_j^*$
  according to $P^*$;
return $\mathcal{H}_j^*$
```

First, we explain why the proposed decomposition algorithm works well. In the complete undirected graph G, there are a total of $\mathcal{N}$ nodes, which are divided into g clusters, denoted as $G = \{G_1, G_2, \ldots, G_g\}$, and G_i represents the node set of the ith cluster, denoted as $\{p_1^i, p_2^i, \ldots, p_{|G_i|}^i\}$, where $\sum_{i=1}^{g} |G_i| = \mathcal{N}$. We assume that the images of containers represented by nodes in the same cluster are similar, while the images between clusters are not as similar as within clusters. This assumption is formalized as

$$\psi_{p_a^i, p_b^i} \geq \psi_{p_a^i, p_c^j} \tag{11}$$

where $i, j = 1, 2, \ldots, g$, and $a, b = 1, 2, \ldots, |G_i|$ and $c = 1, 2, \ldots, |G_j|$. We prove that if graph G can be decomposed into these subgraphs satisfying (11) and the optimal solution is unique, then the Hamiltonian path P^* calculated using the proposed algorithm is consistent with the optimal Hamiltonian path P calculated using (10). For the convenience of subsequent proof, the following two lemmas are proposed and proven.

Lemma 1: For any three nodes p_i, p_j, p_k, the weights $\psi_{p_i,p_j}, \psi_{p_j,p_k}, \psi_{p_i,p_k}$ of the three edges they connect must satisfy that at least two edge weights are equal, and the remaining edge weight must be greater than these two equal edge weights. Assuming $\psi_{p_i,p_j} = \max(\psi_{p_i,p_j}, \psi_{p_j,p_k}, \psi_{p_i,p_k})$, then $\psi_{p_i,p_j} \geq \psi_{p_j,p_k} = \psi_{p_i,p_k}$.

Proof: First, we note that the sets of image layers for containers p_i, p_j, and p_k are A_i, A_j, and A_k. The tree structure of the image layers determines that image layers near the root can be reused, while leaves are generally not reusable. Considering the intersections of image layer sets $A_i \cap A_j$, $A_j \cap A_k$, and $A_i \cap A_k$, they have an inclusion relationship between each pair and represent $\psi_{p_i,p_j}, \psi_{p_j,p_k}, \psi_{p_i,p_k}$. For example, $A_j \cap A_k \subseteq A_i \cap A_j$, since both $A_j \cap A_k$ and $A_i \cap A_j$ are part of A_j and they are the first few image layers of A_j near the root. Similarly, either $A_i \cap A_k \subseteq A_i \cap A_j$ or $A_i \cap A_k \supseteq A_i \cap A_j$. If the latter holds, then $A_i \cap A_k \supseteq A_j \cap A_k$. We assume that $A_i \cap A_j$ is the largest among the three intersections, which implies that $A_j \cap A_k \subseteq A_i \cap A_j$ and $A_i \cap A_k \subseteq A_i \cap A_j$. Next, we prove that $A_j \cap A_k = A_i \cap A_k$.

For $A_j \cap A_k \subseteq A_i \cap A_j$, assuming the image layer $\vartheta \in A_j \cap A_k$, then $\vartheta \in A_i \cap A_j$. Therefore, $\vartheta \in A_i$ and $\vartheta \in A_j$ and $\vartheta \in A_k$, that is, $\vartheta \in A_i \cap A_j \cap A_k \subseteq A_i \cap A_k$. Therefore, for any image layer $\vartheta \in A_j \cap A_k$, $\vartheta \in A_i \cap A_k$ is satisfied, hence $A_j \cap A_k \subseteq A_i \cap A_k$. Similarly, for $A_i \cap A_k \subseteq A_i \cap A_j$, it can be inferred that $A_i \cap A_k \subseteq A_j \cap A_k$. Therefore, $A_j \cap A_k = A_i \cap A_k$. Thus, we have proven Lemma 1. □

Lemma 2: The edge weights from any node in ith cluster to a node p_c^j in another cluster are the same, that is, $\psi_{p_a^i,p_c^j} = \psi_{p_b^i,p_c^j}$, $i, j = 1, 2, \ldots, g$ and $i \neq j$ and $a, b = 1, 2, \ldots, |G_i|$ and $c = 1, 2, \ldots, |G_j|$.

Proof: According to the assumption of (11), $\psi_{p_a^i,p_b^i} \geq \psi_{p_a^i,p_c^j}$. According to Lemma 1, $\psi_{p_a^i,p_c^j} = \psi_{p_b^i,p_c^j}$. Lemma 2 is proved by the arbitrariness of a and b. □

Theorem 1: Subject to the fulfillment of (11), P^* is optimal.

Proof: We divide all edges of the path P into g sets based on the clusters in which the starting points of the $\mathcal{N} - 1$ edges are located, and edges that are in the same set have their starting points in the same cluster. Consider the ith set, i.e., all edges in that set whose starting points belong to the ith cluster, and 2 cases need to be discussed:

1) The ith set does not contain the last edge of P. In this case, the ith set has a total of $|G_i|$ edges, and at least one edge has an endpoint that does not belong to the ith cluster. According to (11), the edges whose endpoints do not belong to the ith cluster are not greater than those within the cluster. Therefore, we keep one edge whose endpoint does not belong to the ith cluster, which will be elaborated on below regarding how this edge is selected. Then, all other edges are amplified to the edges within the cluster, as shown in Fig. 4 from Fig. 4(a) to (b). The sum of the weights of the original edges does not exceed the sum of the weights of the amplified edges, and they can form a Hamiltonian path. Furthermore, this path is not superior to the Hamiltonian path P_i calculated using (10), and we perform the corresponding amplification as shown in Fig. 4(b) to (c). According to Lemma 2, the starting point of the reserved edge can be selected from any point in the ith cluster, so the endpoint of P_i is used as the starting point of the reserved edge as shown in Fig. 4(c) to (d). Finally, we amplify all edges of the ith set to path P_i and add a non-cluster edge.

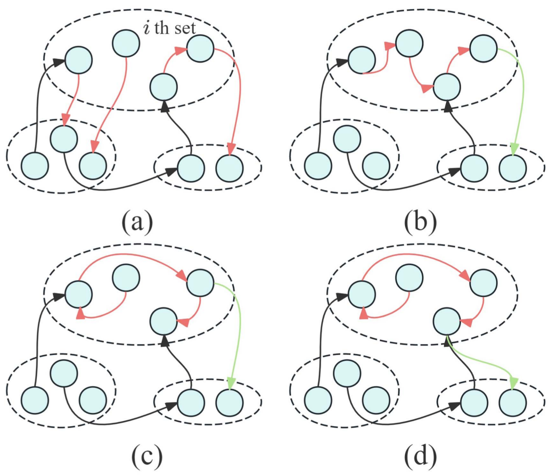

Fig. 4. Example of amplifying the edges of the ith set to the optimal Hamiltonian path of the ith cluster. (a) The initial case, where the red edges denote they belong to the ith set. (b) Amplify the edges to the cluster, where the green edge denotes it is kept. (c) Solving the optimal Hamiltonian path and amplifying the edges. (d) Adjust the kept edge.

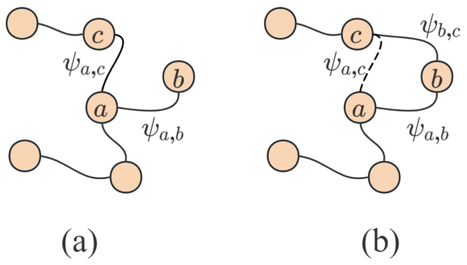

Fig. 5. Example of degree transfer. Nodes with degrees more than 2 are able to transfer degrees to neighboring nodes with larger edge weights. (a) The initial case. (b) $\psi_{a,c}$ is amplified to $\psi_{b,c}$ with $\psi_{a,b} \geq \psi_{a,c}$, thus transferring the degree of node a.

2) The ith set contains the last edge of P. This case leads to the ith set having $|G_i| - 1$ edges, but the number of nodes is $|G_i|$, and thus the edge set amplifies to path P_i in the same way as 1), with no other non-cluster edge.

By 1) and 2), we amplify the path P into g subpaths $P_1, P_2, \ldots, P_g$, and $g - 1$ non-cluster edges, since only one set satisfies the case of 2). The path P is connected, and the graph in which each set before amplification is considered as a node is still connected, so there exists a selection strategy for the edges reserved by each set that enables the graph G^* in which each set after amplification is considered as a node to be connected as well. Next, we show that the sum of the weights of $g - 1$ edges on the graph G^* does not exceed the sum of the edge weights of the paths computed using (10). The graph G^* is an undirected connected acyclic graph, and there exist some nodes whose degree exceeds 2 and whose degree can be transferred to the neighboring nodes, as shown in Fig. 5. Node a neighbors two points b and c with edge weights $\psi_{a,b}, \psi_{a,c}$. From Lemma 1, if $\psi_{b,c}$ is the largest edge, then one of $\psi_{a,b}$ and $\psi_{a,c}$ can be amplified into $\psi_{b,c}$. If either $\psi_{a,b}$ or $\psi_{a,c}$ is the largest edge, assuming that $\psi_{a,b} \geq \psi_{a,c}$, then $\psi_{a,c} = \psi_{b,c}$, and thus $\psi_{a,c}$ is transformed into $\psi_{b,c}$. As a result, the degree of node a can be transferred to the neighboring nodes with greater edge weights. Through this transformation, the $g-1$ edges are eventually able to form a Hamiltonian path that will not outperform the path computed in (10).

So far, we have amplified the path P step by step to become P^*, showing that P^* is not inferior to P, but the path P is the optimal path, and hence P^* is optimal. □

The above proof shows that as long as graph G satisfies (11), the optimal container execution sequence can be found by the proposed method in the time complexity of $\mathcal{O}(\mathcal{N}^{\frac{3}{2}} \times 2^{\sqrt{\mathcal{N}}})$. However, this assumption cannot always be satisfied. To satisfy this assumption as much as possible, we make the ungrouped node p_a^i select the first few edges with the largest weight when grouping, thus forming ith cluster. Therefore, the edge weight $\psi_{p_a^i,p_c^j}$ formed by p_a^i and the ungrouped node p_c^j is certainly less than the weights of these selected edges $\psi_{p_a^i,p_b^i}$. According to Lemma 1, $\psi_{p_b^i,p_c^j} = \psi_{p_a^i,p_c^j} \le \psi_{p_a^i,p_b^i}$, indicating that non-cluster edges must be smaller than the selected edge, and again by Lemma 1 all edges in a cluster are not smaller than the minimum of the selected edge. Therefore, we greedily group the nodes, thus making them satisfy the assumption as much as possible and making the results computed by the proposed algorithm close to optimal. Up to this point, we have elucidated the rationale for the proposed container sequence arrangement algorithm.

In the proposed decomposition algorithm, it is obvious that the decomposition can proceed further when the number of nodes in the graph G is particularly high. This means that when the nodes of graph G are grouped, each cluster G_i can be decomposed using the same method when solving the optimal Hamiltonian path using (10). Suppose the number of nodes in each cluster is $\mathcal{N}_o$ for the first division and the cluster is divided into smaller clusters with $\mathcal{N}_u$ nodes. Then the time complexity is $\mathcal{O}(\mathcal{N} \times \mathcal{N}_u \times 2^{\mathcal{N}_u} + \frac{\mathcal{N}_o \times \mathcal{N}}{\mathcal{N}_u^2} \times 2^{\frac{\mathcal{N}_o}{\mathcal{N}_u}} + \frac{\mathcal{N}^2}{\mathcal{N}_o^2} \times 2^{\frac{\mathcal{N}}{\mathcal{N}_o}})$. When $\mathcal{N}_u = \mathcal{N}^{\frac{1}{3}}$ and $\mathcal{N}_o = \mathcal{N}^{\frac{2}{3}}$, the exponent is minimized, and the time complexity is $\mathcal{O}(\mathcal{N}^{\frac{4}{3}} \times 2^{\sqrt[3]{\mathcal{N}}})$.

C. Image Layer Update Strategy

After determining the container queue and sequence, we focus on further improving the image layer reuse rate by caching the image layer while the containers are running. We denote the sequentially arranged container queue as $\mathcal{H}_j^* = \{h_1^j, h_2^j, \ldots, h_{|\mathcal{H}_j^*|}^j\}$, which also represents the container indices. Thus, $l_{h_i^j,k}$ indicates whether the container h_i^j consists of the kth layer of images. When the container h_i^j starts executing, all the image layers of h_i^j need to be pulled. If the capacity of the image layer repository is insufficient, some unused and low-value image layers will be evicted.

Intuitively, the limited image layer repository space stores the image layers with the highest potential for future reuse as much as possible. Therefore, the retention of the most valuable image layers is transformed into the Knapsack problem. When the container h_i^j is about to run, we divide the image layers stored in the image layer repository into three types: 1) those that can be reused by container h_i^j, 2) those that are being used by other containers but cannot be reused by h_i^j, and 3) the remaining image layers. The set of image layers for the first two types is defined as $L^{\triangle}$. Insufficient space in the image layer repository means that h_i^j in e_j does not have enough remaining disk to pull in the image layers needed except for those that can be reused. Therefore, the third type of image layer needs to be appropriately evicted. In other words, the most valuable of the third type of image layer will be retained, and the space $\hat{B}$ that can be used to retain these image layers will be calculated by the following equation:

Algorithm 3: Image Layer Update Strategy.

```
Data: $\mathcal{H}_j^*$.
for $i \leftarrow 1$ to $|\mathcal{H}_j^*|$ do
    if All computational resources required for $h_i^j$ on $e_j$ are insufficient then
        Waiting for other containers to be released until computational resources are sufficient;
    Calculating $\hat{B}$ using Eq. (12);
    if $\hat{B} < 0$ then
        Waiting for other containers to be released until $\hat{B} \ge 0$;
    Calculating $v_k^L$ using Eq. (13);
    Calculating the image layers that can be retained by solving Knapsack, and the remaining image layers are evicted;
    Pulling the required image layer for $h_i^j$;
    Running container $h_i^j$;
```

$$\hat{B} = d_j - \sum_{L_k \in L^{\triangle}} s_k - \sum_{L_k \notin L^{\triangle}} l_{h_i^j,k} \times s_k \tag{12}$$

which means that except for $L^{\triangle}$ and layers of h_i^j, the remaining space will determine how many free image layers are reserved. Since the container queue to be run by e_j is known, the value of the image layer is defined as v_k^L, and

$$v_k^L = \sum_{w=i+1}^{|\mathcal{H}_j|} l_{h_w^j,k} \times s_k \tag{13}$$

Thus, the update strategy of the image layer is transformed into the Knapsack problem. The optimal solution can be quickly obtained in polynomial time by the approximation algorithm, representing the best image layers that can be preserved in the finite disk space. The image layer update strategy is shown in Algorithm 3. It demonstrates the control of the image layer by the image layer manager, with no data to be returned.

D. Complexity Analysis of ILR-SA

In the first phase of ILR-SA, each container is deployed to the most suitable edge device. For an edge device e_j, the computational time complexity of $f(\cdot)$ is up to $\mathcal{O}(|\mathcal{H}_j|)$, i.e., the size of all containers deployed on e_j. The time complexity of each container is $\mathcal{O}(\sum_{j=1}^{M} |\mathcal{H}_j|)$=$\mathcal{O}(N)$, thus the time complexity of Algorithm 1 is $\mathcal{O}(N^2)$. In the second

TABLE II
DETAILED INFORMATION FOR EDGE DEVICES

Type	Computational Resources(Cores+Memory)			Disk	Number
	CPU	GPU	NPU		
1	64 cores+12GB	2 cores+15GB	NULL	20GB	1
2	2 cores+7.6GB	NULL	1 core+8GB	12GB	3
3	4 cores+7.5GB	NULL	NULL	15GB	4

NULL indicates that no such chip is available.

TABLE III
DETAILED INFORMATION FOR TASKS

Type	Main Resource Requests	Image Size	Number
1	0.495 - 0.98 CPU units + 900 - 1770MB	10.83 - 11.86 GB	5
2	0.1 CPU units + 1GB and 0.3 GPU units + 2GB	10.83 - 11.86 GB	5*
3	0.1 CPU units + 1GB and 0.3 NPU units + 1GB	8.14 - 9.17 GB	5
4	0.105 - 0.413 CPU units + 1900 - 2000MB	1.82 - 4.48 GB	10
5	0.113 - 0.339 CPU units + disk 2GB	2.71 - 5.67GB	10

Resource requests vary with different devices. The chip utilization rate of the device is normalized to 1 unit. Images of type 2 (marked with *) are shared with type 1.

phase, the containers deployed on e_j perform sequential arrangement with time complexity $\mathcal{O}(|\mathcal{H}_j|^{\frac{3}{2}} \times 2^{\sqrt{|\mathcal{H}_j|}})$. Since $\sum_{j=1}^{M} |\mathcal{H}_j|^{\frac{3}{2}} \times 2^{\sqrt{|\mathcal{H}_j|}} < N^{\frac{3}{2}} \times 2^{\sqrt{N}}$, the time complexity of Algorithm 2 is $\mathcal{O}(N^{\frac{3}{2}} \times 2^{\sqrt{N}})$. The third phase executes one Knapsack per container, which has an upper time complexity of $\mathcal{O}(Wd_j)$, so the time complexity of Algorithm 3 is $\mathcal{O}(\sum_{j=1}^{M} \mathcal{H}_j W d_j)$. The total time complexity of ILR-SA is $\mathcal{O}(N^2 + N^{\frac{3}{2}} \times 2^{\sqrt{N}} + \sum_{j=1}^{M} \mathcal{H}_j W d_j)$.

The space complexity of ILR-SA also consists of three parts. Algorithm 1 requires a data structure of size N to store $\mathcal{H}$. Algorithm 2 requires a space of $\sqrt{|\mathcal{H}_j|} \times 2^{\sqrt{|\mathcal{H}_j|}}$ to store the two-dimensional array $D[][]$. Since $\sum_{j=1}^{M} \sqrt{|\mathcal{H}_j|} \times 2^{\sqrt{|\mathcal{H}_j|}} < \sqrt{N} \times 2^{\sqrt{N}}$, the space complexity is $\mathcal{O}(\sqrt{N} \times 2^{\sqrt{N}})$. Algorithm 3 stores data of size Wd_j for Knapsack and hence the space required is $\sum_{j=1}^{M} Wd_j$. The total space complexity of ILR-SA is $\mathcal{O}(N + \sqrt{N} \times 2^{\sqrt{N}} + \sum_{j=1}^{M} Wd_j)$.

V. EXPERIMENTS

We validate the effectiveness of ILR-SA through simulation experiments. The simulation experiment environment and the proposed ILR-SA are implemented using Python 3.10, and the simulation environment is run on a server equipped with 2 Intel (R) Xeon (R) Gold 5218 2.30 GHz CPUs. In the simulation experiments, we set up 8 edge devices whose resource information is shown in Table II. Some devices have heterogeneous chips, such as GPUs and NPUs. We normalize the utilization of these chips to 1 unit, which means that the chip usage of the task will be represented using a number less than 1 instead of the number of cores. We build 30 images for tasks with different focuses on server requirements and we divide them into 5 types. The details of the tasks are shown in Table III. Some tasks contain deep learning-related operations, such as inference of models, so GPUs and NPUs are used. Meanwhile, deep learning frameworks result in relatively large images. These images have a total of 50 different image layers. 82% of the image layers are 600 MB - 2000 MB in size, with a few over 7 GB and a few closer to 1 MB. The bandwidth and image layer repository capacity of these devices are set as experimental variables. α is set to 0.5.

The experiments are conducted from several perspectives. First, the comprehensive performance of ILR-SA is verified by comparing it with the baseline scheduling algorithm. Then, since the three steps of ILR-SA are decoupled, we perform ablation experiments to evaluate the image layer update strategy and the container execution sequence arrangement algorithm. Finally, we evaluate the execution speed of ILR-SA to verify its feasibility. There are 9 baseline algorithms in the experiments, of which the first 4 are scheduling algorithms and the rest are caching algorithms: 1) Multi-Constraint Layer Locality (MCLL) [14]: MCLL improves on the Kubernetes scheduling algorithm. Containers will be scheduled to the device with the most image layer reusables in the repository. To ensure balanced utilization of multiple devices, MCLL uses fair multi-constraints. However, MCLL does not cache the image layers and the layers will be deleted after the container has finished. 2) ADCS [27]: ADCS is an adaptive mechanism for dynamically collaborative computing power and task scheduling. It collaboratively handles a batch of tasks, and the execution sequence of the tasks is adapted to enhance the scheduling effectiveness. In this experiment, the image layer reuse mechanism of ADCS is set to be consistent with that of MCLL. 3)RCCO [15]: RCCO solves the chained containerized virtual network functions orchestration problem. It models the problem as an integer linear program and uses a limited backtracking mechanism, scaling link capacity, and layer eviction to achieve efficient container orchestration. In this experiment, we select edge devices for containers by backtracking mechanism ($\Gamma = 1$) and improve layer reuse rate by layer eviction ($\epsilon = 0.9$). 4) ILR-SA-1&3: The core of ILR-SA lies in the sequence arrangement of the second phase, and this baseline algorithm removes the processing of the second phase while the other phases remain unchanged. 5) LFU: The LFU algorithm counts the frequency of use of the image layer, and when the device's image layer repository is undercapacitated, the least frequently used image layer is evicted. 6) LFU-M [11]: LFU-M is based on LFU, and the value of the image layer is defined as the product of frequency and size. 7) FaasCache [32]: FaasCache is a container caching algorithm that considers the container's survival time, frequency of use, size, and resource requirements to define the container's value. In our experiment, we consider cached objects as image layers. 8) Optimal Image Storage Strategy (OISS) [12]: OISS has two phases. In the first phase, the image layer is pre-placed on the device based on the number of child nodes of the tree structure using the Knapsack algorithm, and in the second phase, the image layer is updated based on the frequency and size. 9) Entire: Entire is the same as LFU, but the image can be reused only if the whole image is identical.

A. ILR-SA Scheduling Performance Evaluation

To evaluate the scheduling performance of ILR-SA comprehensively, MCLL, ADCS, RCCO, and ILR-SA-1&3 will be compared with it. Since the size of the image layer repository

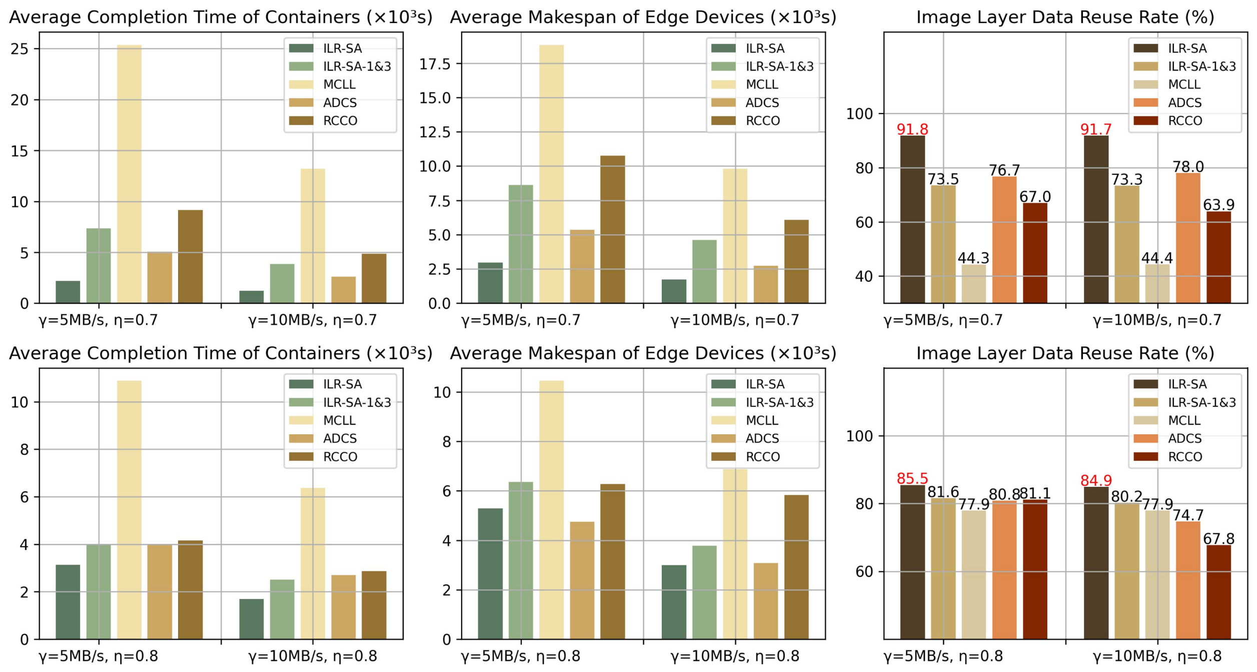

Fig. 6. Performance comparison between ILR-SA and baseline algorithms with different image layer repository capacities and bandwidths.

greatly affects the reuse rate of the image layer, we divide a portion of the disk space from the device as an image layer repository according to the ratio η. We set $\eta = 0.7$ and $\eta = 0.8$ to simulate the image layer repository in the case of tight and sufficient capacity, respectively. In addition, the edge device's bandwidth determines the image layer's download speed. The tighter bandwidth leads to more critical image-pulling time. We set the bandwidth to 5 MB/s and 10 MB/s, denoted as $\gamma = 5$ MB/s and $\gamma = 10$ MB/s, respectively. The experiments measure the ACT, AMS, and image layer reuse rate of these algorithms, defined as the percentage of reused image layer data out of the total required by all containers. The queue size of containers to be scheduled is 200, uniformly generated from 30 containers.

The experimental results are shown in Fig. 6. Clearly, ILR-SA exhibits the most outstanding optimization effect, demonstrating its ability to effectively reduce ACT and AMS through image layer reuse. The experimental data reveal the following three points: 1) ILR-SA has a greater optimization in bandwidth-scarce scenarios. In $\gamma = $5MB/s, $\eta = 0.7$, the AMS of ILR-SA is optimized by 44.71%, 72.47% and 65.70% compared with that of ADCS, RCCO and ILR-SA-1&3, while the improvements become 36.75%, 71.61% and 62.64% in $\gamma = 10$ MB/s, $\eta = 0.7$. This is because, in bandwidth-limited scenarios, image pulling latency becomes a critical performance bottleneck. ILR-SA is designed to address this by leveraging image layer reuse for efficient container scheduling. 2) ILR-SA also has good scheduling optimization in storage-constrained scenarios. In $\gamma = $ 5MB/s, $\eta = 0.7$, the ACT of ILR-SA is optimized by 56.54%, 75.92% and 70.00% compared to ADCS, RCCO and ILR-SA-1&3, while in $\gamma = $ 5MB/s, $\eta = 0.8$, the improvements are 21.78%, 24.81% and 21.81%. It is worth noting that with more image layer repository capacity ($\eta = 0.8$), the ACT, AMS of ILR-SA is longer than $\eta = 0.7$, while the data reuse rate is smaller. This shows that for IRL-SA, $\eta = 0.7$ is already sufficient for the image layer reuse mechanism. At the same time, a larger repository means less disk space for containers to run, resulting in some containers being unable to execute in parallel on the device due to lack of disk space. 3) MCLL performs poorly in the experiment because it schedules containers online without coordinating multiple containers. This highlights the importance of multi-container coordination for better scheduling performance since ILR-SA reduces up to 91.3% of ACT and 84.3% of AMS compared to MCLL in $\gamma = $5MB/s, $\eta = 0.7$. Container completion time includes not only image pull latency but also resource wait time and execution time, which MCLL does not consider. Consequently, even with $\eta = 0.8$, MCLL has a similar data reuse rate to other algorithms but much longer ACT and AMS. 4) The container execution sequence plays a significant role, as ILR-SA and ADCS both outperform ILR-SA-1&3.

B. Image Layer Update Strategy Evaluation

In this section, we verify the effectiveness of the image layer update strategy in ILR-SA. The scheduling algorithm uses the first phase of ILR-SA, while the second phase is not executed. Then, LFU, LFU-M, FaasCache, OISS, and Entire are compared with the proposed image layer update strategy, denoted as ILR-SA-1&3. We have set up two container queues of different lengths, with 200 and 500 containers, respectively, and $\gamma = $ 10 MB/s, $\eta = 0.7$. The experimental results are shown in Fig. 7. Intuitively, ILR-SA-1&3, LFU, LFU-M, and FaasCache have similar data reuse rates, with ILR-SA-1&3 being the highest, while the method of reusing the entire image is inferior to all layered methods. This indicates that image layering can significantly reduce the pulling time of images and decrease the completion time of containers. LFU, LFU-M, and FaasCache

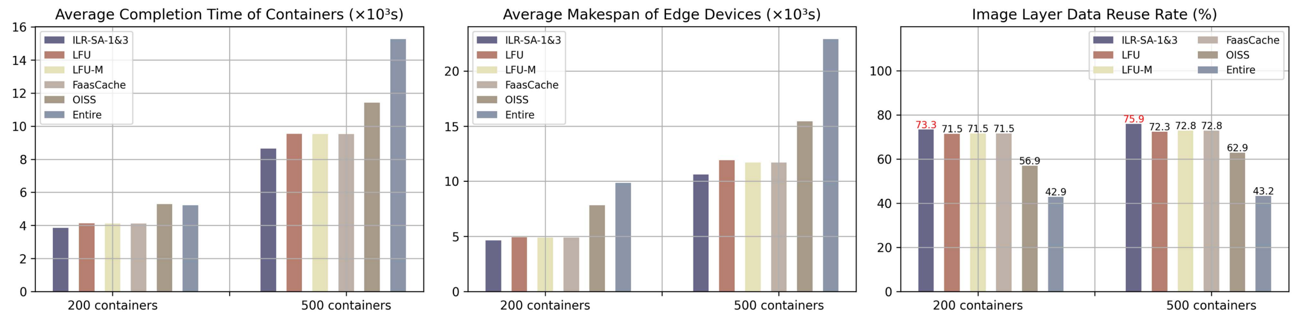
Fig. 7. Comparison of ILR-SA's image layer update strategy and baseline algorithm with different container queues.

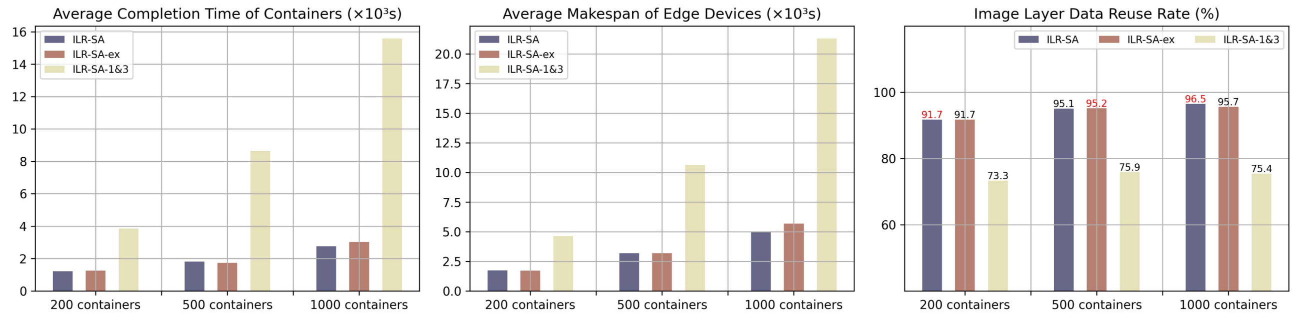
Fig. 8. Experimental results of ILR-SA's container execution sequence arrangement algorithm and its extended algorithm with different container queues.

TABLE IV
EXECUTION SPEED OF ILR-SA AT VARIOUS PHASES

Containers	ILR-SA-1&3				ILR-SA				ILR-SA-ex			
	Reuse Rate	T_1	T_2	T_3	Reuse Rate	T_1	T_2	T_3	Reuse Rate	T_1	T_2	T_3
200	73.35%	5.24s	0s	0.13s	91.73%	5.18s	0.05s	0.09s	91.73%	5.29s	0.04s	0.09s
500	75.86%	12.72s	0s	0.48s	95.10%	13.13s	0.96s	0.28s	95.15%	12.68s	0.23s	0.35s
1000	75.40%	27.54s	0s	1.55s	96.52%	28.02s	17.41s	0.79s	95.65%	27.65s	0.87s	0.84s

all determine the update of the image layer based on frequency, while ILR-SA-1&3 uses an optimized algorithm to determine the update based on the future frequency of the image layers. Thus, it has better performance.

C. Container Execution Sequence Arrangement Performance Evaluation and Algorithm Speed Evaluation

As the core method of ILR-SA, we conduct an ablation experiment on the proposed container execution sequence arrangement algorithm. We set $\gamma = 10$ MB/s, $\eta = 0.7$, and use three different container queue lengths of 200, 500, and 1000. The experiment compares ILR-SA-1&3, which does not perform the second phase of ILR-SA. In addition, the extended container execution sequence arrangement algorithm mentioned at the end of Section IV is evaluated and denoted as ILR-SA-ex. The experimental results are shown in Fig. 8. It can be observed that the container execution sequence arrangement algorithm significantly improves the data reuse rate and reduces ACT and AMS, indicating that adjusting the execution sequence of containers is crucial for improving the efficiency of image reuse. The performance of ILR-SA and ILR-SA-ex is similar. However, ILR-SA-ex has lower time complexity. This occurs due to the execution of many identical containers on the same device, leading to increased cluster similarity. Therefore, even after two rounds of grouping, the calculated solution is still close to optimal.

To verify the time complexity of deploying ILR-SA, the running time of ILR-SA is evaluated. We record the data reuse rate of the image layer, and the running times of the three phases denoted as T_1, T_2, and T_3, respectively. These times are recorded on the device on which the simulator is running. The experimental results are shown in Table IV. It indicates that the container execution sequence arrangement and the image layer update strategy both have a fast execution speed, which in most cases does not exceed 1 s. When the container queue length is 1000, the execution time of the second phase of ILR-SA is 17.41 s, while ILR-SA-ex is 0.87 s, and has similar data reuse rates. Therefore, the extended ILR-SA can efficiently address an excessive number of containers. The first phase of ILR-SA has the most execution time, as each container's scheduling decision requires calculating the total completion time and makespan to determine the most suitable edge device. However, these execution times are particularly small relative to the container's completion time, meaning that ILR-SA has a fast enough execution speed to deploy on edge scenarios.

VI. Conclusion

This paper proposes ILR-SA, a container scheduling strategy based on image layer reuse and sequence arrangement for the MEC scenarios. By improving the reuse rate of the image layer, the time for containers to pull images is significantly shortened, which reduces the completion time of computational tasks. The proposed container scheduling strategy first deploys containers sequentially to appropriate edge devices. Then, the execution sequence of containers on the same device is adjusted so that containers with similar image layers are executed adjacent to each other. Finally, the containers keep the most valuable image layers in the limited repository through the image layer update strategy at runtime, thus improving the image layer reuse rate. The experiments show that ILR-SA is efficient and can effectively reduce the completion time of containers, thus improving the quality of service in the MEC scenarios.

However, the proposed strategy has some limitations. ILR-SA focuses on reducing the image pulling time, which means that for containers with short pulling time and long execution time, ILR-SA may not work very well. In addition, the design of ILR-SA makes it only capable of scheduling independent containers, but not containers with complex dependencies and priorities. In the future, we will further address these challenges.

References

[1] Y. C. Hu, M. Patel, D. Sabella, N. Sprecher, and V. Young, "Mobile edge computing–A key technology towards 5G," *ETSI White Paper*, vol. 11, no. 11, pp. 1–16, 2015.

[2] L. Urblik, E. Kajati, P. Papcun, and I. Zolotová, "Containerization in edge intelligence: A review," *Electronics*, vol. 13, no. 7, 2024, Art. no. 1335.

[3] W. Kithulwatta, W. U. Wickramaarachchi, K. Jayasena, B. Kumara, and R. Rathnayaka, "Adoption of docker containers as an infrastructure for deploying software applications: A review," in *Proc. Adv. Smart Soft Comput.*, 2021, pp. 247–259.

[4] J. N. Acharya and A. C. Suthar, "Docker container orchestration management: A review," in *Proc. Int. Conf. Intell. Vis. Comput.*, Springer, 2021, pp. 140–153.

[5] S. Telenyk, O. Sopov, E. Zharikov, and G. Nowakowski, "A comparison of kubernetes and kubernetes-compatible platforms," in *Proc. 11th IEEE Int. Conf. Intell. Data Acquisition Adv. Comput. Syst.: Technol. Appl.*, 2021, pp. 313–317.

[6] A. Malviya and R. K. Dwivedi, "A comparative analysis of container orchestration tools in cloud computing," in *Proc. 9th Int. Conf. Comput. Sustain. Glob. Develop.*, 2022, pp. 698–703.

[7] A. Algude, N. Ranjan, and M. Panpaliya, "A detailed review on blockchain-enabled deep learning on kubernetes for disease prediction," *Grenze Int. J. Eng. Technol.*, vol. 10, pp. 1413–1420, 2024.

[8] J. Yan and K. Zhang, "An industrial internet platform for industrial robots based on cloud-edge-end service collaboration," in *Proc. Int. Conf. Smart Manuf. Ind. Logistics Eng.*, Springer, 2023, pp. 467–473.

[9] M. P. J. Kuranage, L. Nuaymi, A. Bouabdallah, T. Ferrandiz, and P. Bertin, "Deep learning based resource forecasting for 5G core network scaling in kubernetes environment," in *Proc. IEEE 8th Int. Conf. Netw. Softwarization*, 2022, pp. 139–144.

[10] X. Sun, D. Wang, W. Zhang, G. Lou, J. Wang, and R. Yadav, "Minimizing service latency through image-based microservice caching and randomized request routing in mobile edge computing," *IEEE Internet Things J.*, vol. 11, no. 18, pp. 30054–30068, Sep. 2024.

[11] F. Mou, Z. Tang, J. Lou, J. Guo, W. Wang, and T. Wang, "Joint task scheduling and container image caching in edge computing," 2023, *arXiv:2310.00560*.

[12] L. Yin, J. Luo, and K. Li, "An optimal image storage strategy for container-based edge computing in smart factory," *IEEE Internet Things J.*, vol. 10, no. 8, pp. 7204–7214, Apr. 2023.

[13] J. Gu, Z. Liu, D. Zhang, C. Chen, and Y. Cheng, "A container scheduling strategy based on node image layer cache," in *Proc. 35th Chin. Control Decis. Conf.*, 2023, pp. 263–268.

[14] L. Funari, L. Petrucci, and A. Detti, "Storage-saving scheduling policies for clusters running containers," *IEEE Trans. Cloud Comput.*, vol. 11, no. 1, pp. 595–607, First Quarter 2023.

[15] M. Dolati, S. H. Rastegar, A. Khonsari, and M. Ghaderi, "Layer-aware containerized service orchestration in edge networks," *IEEE Trans. Netw. Service Manag.*, vol. 20, no. 2, pp. 1830–1846, Jun. 2023.

[16] Z. Tang, J. Lou, and W. Jia, "Layer dependency-aware learning scheduling algorithms for containers in mobile edge computing," *IEEE Trans. Mobile Comput.*, vol. 22, no. 6, pp. 3444–3459, Jun. 2023.

[17] Z. Miao, P. Yong, Z. Jiancheng, and Y. Quanjun, "Efficient flow-based scheduling for geo-distributed simulation tasks in collaborative edge and cloud environments," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 12, pp. 3442–3459, Dec. 2022.

[18] H. Teng, Z. Li, K. Cao, S. Long, S. Guo, and A. Liu, "Game theoretical task offloading for profit maximization in mobile edge computing," *IEEE Trans. Mobile Comput.*, vol. 22, no. 9, pp. 5313–5329, Sep. 2023.

[19] 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.

[20] S. Dauzère-Pérès, J. Ding, L. Shen, and K. Tamssaouet, "The flexible job shop scheduling problem: A review," *Eur. J. Oper. Res.*, vol. 314, no. 2, pp. 409–432, 2024.

[21] P. Perez-Gonzalez and J. M. Framinan, "A review and classification on distributed permutation flowshop scheduling problems," *Eur. J. Oper. Res.*, vol. 312, no. 1, pp. 1–21, 2024.

[22] Z. Liu, Q. Lan, and K. Huang, "Resource allocation for multiuser edge inference with batching and early exiting," *IEEE J. Sel. Areas Commun.*, vol. 41, no. 4, pp. 1186–1200, Apr. 2023.

[23] W.-K. Lai, Y.-C. Wang, and S.-C. Wei, "Delay-aware container scheduling in kubernetes," *IEEE Internet Things J.*, vol. 10, no. 13, pp. 11813–11824, Jul. 2023.

[24] B. Tang, J. Luo, M. S. Obaidat, and P. Vijayakumar, "Container-based task scheduling in cloud-edge collaborative environment using priority-aware greedy strategy," *Cluster Comput.*, vol. 26, no. 6, pp. 3689–3705, 2023.

[25] J. Santos, C. Wang, T. Wauters, and F. De Turck, "Diktyo: Network-aware scheduling in container-based clouds," *IEEE Trans. Netw. Service Manag.*, vol. 20, no. 4, pp. 4461–4477, Dec. 2023.

[26] L. Wang et al., "An efficient load prediction-driven scheduling strategy model in container cloud," *Int. J. Intell. Syst.*, vol. 2023, no. 1, 2023, Art. no. 5959223.

[27] Y. Xu, L. Chen, Z. Lu, X. Du, J. Wu, and P. C. K. Hung, "An adaptive mechanism for dynamically collaborative computing power and task scheduling in edge environment," *IEEE Internet Things J.*, vol. 10, no. 4, pp. 3118–3129, Feb. 2023.

[28] C. Chen, M. Herrera, G. Zheng, L. Xia, Z. Ling, and J. Wang, "Cross-edge orchestration of serverless functions with probabilistic caching," *IEEE Trans. Services Comput.*, vol. 17, no. 5, pp. 2139–2150, Sep./Oct. 2024.

[29] L. Pan, L. Wang, S. Chen, and F. Liu, "Retention-aware container caching for serverless edge computing," in *Proc. IEEE Conf. Comput. Commun.*, 2022, pp. 1069–1078.

[30] A. C. Zhou, R. Huang, Z. Ke, Y. Li, Y. Wang, and R. Mao, "Tackling cold start in serverless computing with multi-level container reuse," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2024, pp. 89–99.

[31] D. Jayaram, S. Jeelani, and G. Ishigaki, "Container caching optimization based on explainable deep reinforcement learning," in *Proc. IEEE Glob. Commun. Conf.*, 2023, pp. 7127–7132.

[32] A. Fuerst and P. Sharma, "FaasCache: Keeping serverless computing alive with greedy-dual caching," in *Proc. 26th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2021, pp. 386–400.

[33] H. Yu et al., "RainbowCake: Mitigating cold-starts in serverless with layer-wise container caching and sharing," in *Proc. 29th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2024, pp. 335–350.

[34] S. Hu, Z. Qu, B. Tang, B. Ye, G. Li, and W. Shi, "Joint service request scheduling and container retention in serverless edge computing for vehicle-infrastructure collaboration," *IEEE Trans. Mobile Comput.*, vol. 23, no. 6, pp. 6508–6521, Jun. 2024.

[35] M. Zhao, X. Zhang, Z. He, Y. Chen, and Y. Zhang, "Dependency-aware task scheduling and layer loading for mobile edge computing networks," *IEEE Internet Things J.*, vol. 11, no. 21, pp. 34364–34381, Nov. 2024.

Haijie Wu is currently working toward the MS degree with the School of Computer Science and Engineering, South China University of Technology, Guangzhou, China, supervised by Dr. Weiwei Lin. His research interests mainly include cloud edge collaboration, edge computing, and AI algorithms.

Weiwei Lin (Senior Member, IEEE) received the BS and MS degrees from Nanchang University, in 2001 and 2004, respectively, and the PhD degree in computer application from the South China University of Technology, in 2007. Currently, he is a professor with the School of Computer Science and Engineering, South China University of Technology. His research interests include distributed systems, cloud computing, and AI application technologies. He has published more than 200 papers in refereed journals and conference proceedings. He has been a reviewer for many international journals, including *IEEE Transactions on Parallel and Distributed Systems*, *IEEE Transactions on Services Computing*, *IEEE Transactions on Cloud Computing*, *IEEE Transactions on Computers*, *IEEE Transactions on Cybernetics*, etc. He is a distinguished member of CCF.

Haotong Zhang received the bachelor's and master's degrees from the South China University of Technology, in 2016 and 2019, respectively. Currently, he is working toward the PhD degree with the South China University of Technology. His research interests mainly include edge computing, edge cloud collaboration, and Internet of Things.

Fang Shi received the MS degree from Guangzhou University, in 2019, and the PhD degree from the South China University of Technology, in 2023. She is now an associate professor with the College of Mathematics and Informatics, South China Agricultural University, Guangzhou, China. Her research interests focus on federated learning, distributed systems, edge intelligence and wireless cooperative communications.

Wangbo Shen received the BS degree from Changsha University, Changsha, China, in 2012 and 2016, respectively, and the MS degrees from Central South University, Changsha, in 2016 and 2019 respectively. Currently, he is working toward the PhD degree with the School of Computer Science and Engineering, South China University of Technology, Guangzhou, China, supervised by Dr. Weiwei Lin. His research interests mainly include Kernel learning, AutoML, and edge computing.

Keqin Li (Fellow, IEEE) is a SUNY distinguished professor of computer science with the State University of New York. He is also a national distinguished professor with Hunan University, China. His current research interests include, fog computing and mobile edge computing, energy-efficient computing and communication, embedded systems and cyber-physical systems, heterogeneous computing systems, Big Data computing, high performance computing, computer architectures and systems, computer networking, ML, intelligent and soft computing. He is currently an associate editor of the *ACM Computing Surveys* and the *CCF Transactions on High Performance Computing*. He is an AAIA fellow. He is also a member of Academia Europaea (Academician of the Academy of Europe).

Albert Y. Zomaya (Fellow, IEEE) received the PhD degree in control engineering from Sheffield University, Sheffield, U.K., in 1990. He is currently the Peter Nicol Russell chair professor of computer science and the director of the Centre for Distributed and High-Performance Computing, School of Computer Science, University of Sydney, Sydney, NSW, Australia. Prior to that, he was the chair professor of High-Performance Computing & Networking from 2008 to 2021. He was an australian research council professorial fellow from 2010 to 2014 and held the CISCO Systems chair professor of Internetworking from 2002 to 2007. He also served as the head of the School of Computer Science from 2006 to 2007.