CCF Transactions on Pervasive Computing and Interaction
https://doi.org/10.1007/s42486-025-00186-z

REGULAR PAPER

LR²Scheduler: layer-aware, resource-balanced, and request-adaptive container scheduling for edge computing

Wentao Peng[1] · Zhiqing Tang[2] · Jianxiong Guo[1,2] · Jiong Lou[3] · Tian Wang[2] · Weijia Jia[1,2]

Received: 25 January 2025 / Accepted: 2 February 2025

Abstract

Lightweight containers provide an efficient approach for deploying computation-intensive applications in network edge. The layered storage structure of container images can further reduce the deployment cost and container startup time. Existing research has rarely considered the dynamic adjustment of different metrics in schedulers, and layer-aware scheduling is still in the theoretical stage. Moreover, current schedulers fail to utilize system resources efficiently. To address this gap, a Layer-aware, Resource-balanced, and Request-adaptive container Scheduler (LR²Scheduler) has been proposed and implemented in edge computing. Specifically, we first utilize container image layer information to design and implement a node scoring and container scheduling mechanism. This mechanism effectively lowers download costs for container deployment, which is crucial for edge computing with limited bandwidth. Then, we design a scoring system that adapts to resource demands based on user requirements and the remaining resource information to optimize idle resource utilization. Finally, based on the aforementioned multifaceted scoring mechanism, the scheduler can dynamically adjust scheduling weights to select appropriate strategies to meet user demands while also ensuring load balancing within the edge cluster. Our LR²Scheduler is built on the scheduling framework of Kubernetes, enabling full process automation from task information acquisition to container deployment. Testing on a real system has demonstrated that our LR²Scheduler effectively reduces load imbalance among cluster nodes, enhances resource utilization, and significantly optimizes the efficiency and performance of container deployment compared to the default scheduler.

Keywords Dynamic weight · Layer-aware scheduling · Container scheduler · User requirements · Edge computing

1 Introduction

Emerging as a prominent computing paradigm, edge computing enhances resource availability by deploying applications on edge servers closer to users (Shi et al. 2016). Containers have emerged as the preferred method for deploying services and applications in edge computing, thanks to their lightweight nature and ease of deployment, which greatly facilitates the flexible allocation and efficient utilization of resources (Tang et al. 2023; Ma et al. 2018; Fu et al. 2020). By utilizing containers on edge servers, applications can significantly reduce the response time and enhance the Quality of Service (QoS). Kubernetes has become the leading tool for container cluster orchestration in cloud data centers(Carrión 2022), managing the entire lifecycle of containers including deployment (Tang et al. 2023), migration (Tang et al. 2024), updates (Cui et al. 2024), and elastic scaling (Brooker et al. 2023). Kubernetes offers various scheduling strategies, such as `ImageLocality` and

✉ Zhiqing Tang
zhiqingtang@bnu.edu.cn

Wentao Peng
wentaopeng@uic.edu.cn

Jianxiong Guo
jianxiongguo@bnu.edu.cn

Jiong Lou
lj1994@sjtu.edu.cn

Tian Wang
tianwang@bnu.edu.cn

Weijia Jia
jiawj@bnu.edu.cn

1 Guangdong Key Lab of AI & Multi-Modal Data Processing, Beijing Normal University-Hong Kong Baptist University United International College, Zhuhai 519087, China

2 Institute of Artificial Intelligence and Future Networks, Beijing Normal University, Zhuhai 519087, China

3 Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai 200240, China

Published online: 25 March 2025

`LeastAllocated`, to achieve different goals like selecting nodes with pre-existing container images or those with balanced resource usage (Rejiba and Chamanara 2022). However, few default scheduling strategies in edge computing take into account the limited bandwidth and historical user request data, which is crucial for latency-sensitive edge users and resource-constrained servers.

Existing research shows that default Kubernetes scheduling algorithms are poorly suited for edge computing environments due to their limited resources, geographic dispersion, and network instability (Zhu et al. 2021; Xing et al. 2022; Carrión 2022). To address this, container management tools like KubeEdge (Xiong et al. 2018), K3s (2024), and Akraino (2024) extend Kubernetes to the edge by adding features such as robust management and MQTT support (Xiong et al. 2018). Additionally, tools like Koordinator (2024), Volcano (2024), and Katalyst (2024) enhance Kubernetes for distributed scenarios by improving QoS support. However, these tools not only neglect the issue of limited bandwidth in edge computing, but also fail to adequately address the problems of resource fragmentation and load imbalance as well. This results in degraded system stability and response speed, making the downloading of container images time-consuming (Fu et al. 2020). Container images are stored in layers, and repeated downloads can be reduced by sharing these layers (Gu et al. 2023). Existing researches have explored layer sharing and proposed algorithms for container placement (Tang et al. 2023; Gu et al. 2021), migration (Tang et al. 2024), and image downloads (Gu et al. 2023; Lou et al. 2022) based on layer sharing. Despite this, a systematic implementation of a layer sharing scheduler is still necessary. Implementing this scheduler in edge environments is crucial to reduce deployment cost for many edge clusters managed by Kubernetes.

Implementing the layer-aware, resource-balanced, and request-adaptive scheduler in edge clusters is highly challenging. Using the scheduling framework of Kubernetes (Scheduling Framework 2024), we can create various extension points like Filter, Score, and Bind. The Filter extension point eliminates nodes that cannot run the container. The Score then ranks the remaining nodes. The scheduler calls each scoring extension point for every node. Finally, the Bind extension point binds a container to a node. *However, the first challenge remains on how to automatically obtain and score layer information for nodes*. Currently, most existing work lacks systematic implementation, with some basic schedulers requiring prior knowledge of layer information (Fu et al. 2020). To fill in such gaps, we develop a custom layer-aware scheduler within the Kubernetes scheduling framework that automatically retrieves and updates layer information from the Docker registry, integrating seamlessly with Kubernetes deployments (2024). Layer information is periodically retrieved from the registry and cached locally. The scheduler analyzes the required layer information for new container deployment tasks, and gathers the existing image layer information from each edge node, scores and rates the nodes, finally deploys containers accordingly.

However, using only the layer-aware scheduler will make Kubernetes tend to schedule containers on edge nodes with more layers, leading to higher load on these nodes with others remaining underutilized and generating a large amount of resource fragmentation. *This brings up a second challenge, i.e., how to make container scheduling decisions that meet user needs while ensuring efficient utilization of node resources.* Existing research has considered the resource utilization when scheduling containers (Gunasekaran et al. 2020), including the default scheduling policy `NodeResourcesBalancedAllocation` (Scheduler Configuration 2024). However, these studies cannot dynamically focus on different scheduling strategies based on user needs, nor can they effectively combine layer sharing to further reduce deployment costs while maintaining load balancing. To address these issues, we propose a resource-balanced, user request-adaptive strategy combined with layer-aware approaches and Kubernetes scheduling plugins to derive new scores through weighted calculations. Moreover, static weights for various metrics do not effectively adapt to load changes and cannot fine-tune scheduling parameters for different network environments, which is essential for ensuring QoS for various services (Li et al. 2012). Therefore, we further design a Layer-aware, Resource-balanced, and Request-adaptive container Scheduler (LR^{2}Scheduler) for edge computing. The LR^{2}Scheduler dynamically adjusts the layer score weight, lowering it during high load to minimize impact and raising it during low load to decrease download costs and shorten container startup time.

In this paper, we propose and implement the LR^2 Scheduler within the Kubernetes scheduling framework for edge computing. As shown in Fig. 1, LR^{2}Scheduler employs a layer-sharing scoring mechanism that dynamically adapts to resource utilization and user request using scoring extension points, the Kubernetes API, etcd, and Kubelet. When a new container request is sent from the user, LR^{2}Scheduler first retrieves the required resource information and layer information from the user, and obtains the remaining resource information and locally stored layer information from each node. It then scores the nodes using all the information and combines the score with the score of default Kubernetes scheduler to minimize container deployment costs while maintaining efficient resource utilization. Finally, the scheduler selects the highest-scoring node for task deployment. The scheduler dynamically adjusts the weights of all scoring mechanisms and integrates well with various scheduling plugins, providing good scalability. We have implemented this custom scheduler in Kubernetes and verified it in a

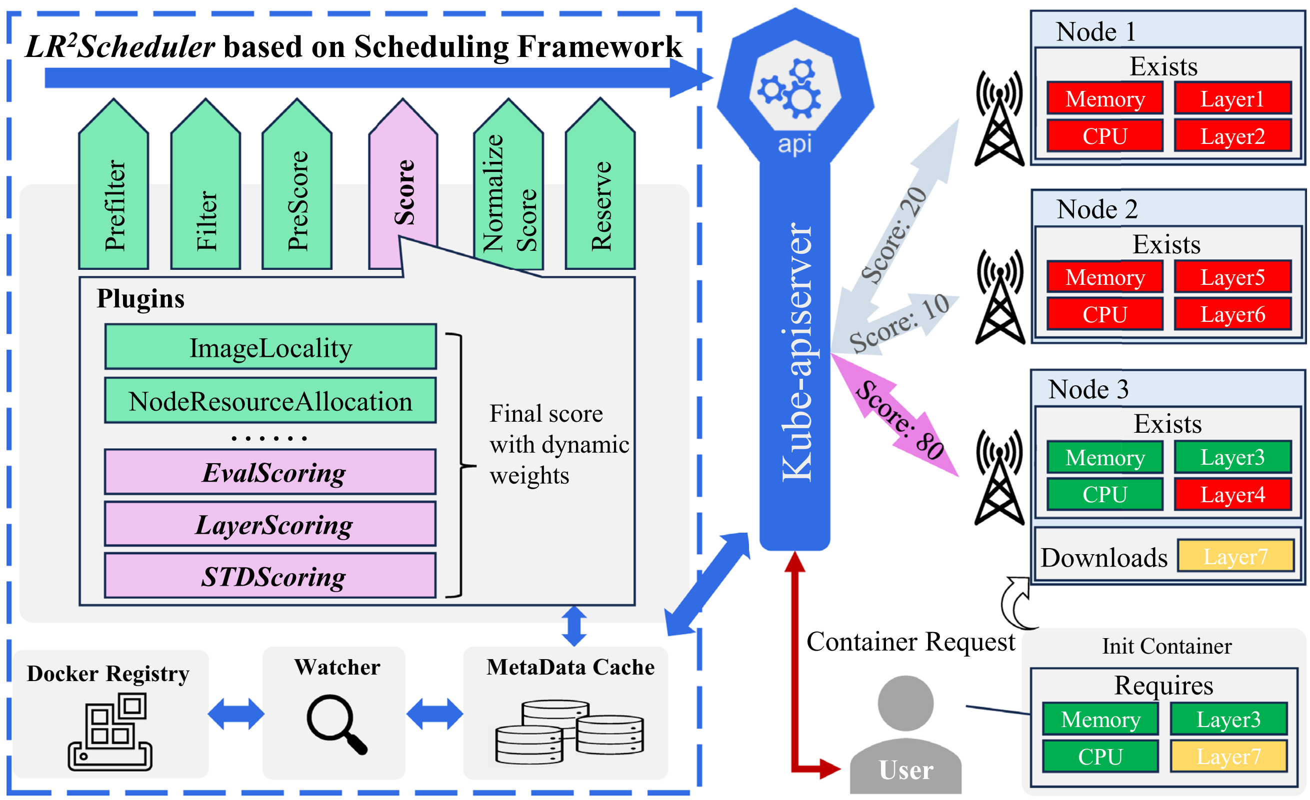


Fig. 1 Overview of LR²Scheduler

real cluster environment. Experimental results show that our LR²Scheduler lowers deployment costs while considering load balancing.

In this extended version of our previous work (Tang et al. 2024), we aim to minimize resource fragmentation and optimize the dynamic weight algorithm to improve adaptability and effectiveness in dynamic environments. First, we improve the problem modeling by incorporating a resource demand adaptive strategy and refining the final score of scheduler based on real-world adjustments to better capture features of user demand. The strategy is used to determine whether there is similarity in the history of user requests, reducing resource fragmentation, enabling the cluster to deploy more containers. Second, we improve the dynamic weight mechanism by changing its values from discrete to continuous and implement real-time weight adjustments for the three scheduling strategies that we used, leading to better decision-making and minimized adverse effects compared to previous work (Tang et al. 2024). Additionally, we refine the resource balance scheduling mechanism by reducing the occurrence of high-load nodes. We further validate the effectiveness and adaptability of our LR²Scheduler through experimental testing.

In summary, the contributions of this paper are as follows:

1. We propose and implement a layer-aware, resource-balanced, and request-adaptive container scheduler, which autonomously calculates scores using the existing resource information on nodes, user requirements, and layer information. This scheduler can effectively reduce resource fragmentation and lower deployment costs when deploying containers.
2. We present a resource-adaptive weight adjustment algorithm that enhances load balancing and optimizes resource utilization. This method reduces layer download costs by combining resource demand adaptive scheduling plugins with layer scheduling plugins and the official scheduler. This approach reduces layer download costs during low load periods while balancing container distribution among nodes during high load.
3. We implement our LR²Scheduler in a real Kubernetes-based edge system. The experimental results show that our LR²Scheduler has good scalability. It can effectively reduce the deployment cost of containers and balance the resource load of different nodes.

The rest of this paper is organized as follows. In Sect. 2, the related work is introduced; Sect. 3 describes the system model and problem statement; Sect. 4 presents the dynamic adaptation layer scheduling algorithm based on resource demand; Sect. 5 details the system implementation; Sect. 6 evaluates performance; Sect. 7 concludes the paper.

2 Related work

2.1 Resource allocation in edge computing

Significant advancements have been made in resource allocation research for edge computing (Wang et al. 2020, 2021). For example, Xing et al. (2023) model the computing resources of the edge nodes uniformly and introduce methods for heterogeneous task classification and recognition. Cai et al. (2024) present an explainable online approximation algorithm to optimize resource allocation, balancing model training and inference accuracy. Ouyang et al. (2023) propose a reactive provisioning approach for hybrid resource provisioning without prior knowledge of future system dynamics. Xu et al. (2024) formulate the dynamic parallel multi-server selection and allocation problem to minimize task computing and transmission times. Chen et al. (2024) develop an algorithm to minimize system energy consumption while meeting performance requirements for dynamic task offloading and resource allocation. Xu et al. (2023) explore joint channel estimation and resource allocation in Intelligent Reflecting Surface-aided edge computing systems.

In large-scale task scheduling within cloud environments, existing mechanisms do not adequately address the specific characteristics of user tasks, limiting Kubernetes's ability to optimize performance (Dong et al. 2024). Analyzing users' historical deployment tasks can reveal patterns in their needs, enabling better resource allocation, minimizing fragmentation, and predicting future resource requirements (Xie et al. 2019).

2.2 Layer-aware container scheduling

Layer-aware scheduling research is in its early stages. Rong et al. (2022) analyze 3735 images from Docker Hub and find that caching image layers on destination servers reduces migration time. Ma et al. (2018) propose an edge computing platform that uses the layered features of the storage system to reduce the synchronization cost of the file system. Lou et al. (2022) address the container assignment and layer sequencing problem, proving its NP-hardness, and proposing a layer-aware scheduling algorithm. Gu et al. (2021) study a layer aware microservice placement and request scheduling at the edge. Dolati et al. (2022) address essential aspects of orchestrating services such as downloading and sharing container layers and steering traffic among network functions. Liu et al. (2022) study the optimal deployment strategy to balance layer sharing and chain sharing of microservices to minimize image pull delay and communication overhead.

However, existing research on layer-aware container scheduling and resource allocation has not effectively integrated layer sharing information with load balancing and user demands. Although some studies have made initial considerations (Tang et al. 2023; Gu et al. 2021; Tang et al. 2024), they lack focus on real system implementation or information retrieval. This paper presents LR^{2}Scheduler, an efficient and scalable solution that addresses this gap in current research.

3 System model and problem formulation

3.1 System model

In edge computing, services are created on specific edge nodes, requiring containers to run. These containers rely on images, which are built from multiple layers.

Overview: A set of tasks $\mathbf{K} = \{k_1, k_2, ..., k_{|\mathbf{K}|}\}$ is offloaded from users to edge nodes for processing, where $|\cdot|$ is used to indicate the number of elements in the set, e.g., $|\mathbf{K}|$ is the number of tasks. To handle these tasks, a set of containers $\mathbf{C} = \{c_1, c_2, ..., c_{|\mathbf{C}|}\}$ is deployed on the nodes. Each container requires an image file from the set $\mathbf{M} = \{m_1, m_2, ..., m_{|\mathbf{M}|}\}$. Since requesting a container is equivalent to requesting its corresponding image, and the only difference is a writable container layer, these concepts are unified (Zhao et al. 2020; Tang et al. 2023). Essentially, a task requests a container, which in turn requires specific layers from the set $\mathbf{L} = \{l_1, l_2, ..., l_{|\mathbf{L}|}\}$.

Edge node: The set of edge nodes, $\mathbf{N} = \{n_1, n_2, \dots, n_{|\mathbf{N}|}\}$ is deployed at the edge of the core network. Each node $n \in \mathbf{N}$ maintains three sets: running containers $\mathbf{C}_n(t) \subseteq \mathbf{C}$, local images $\mathbf{M}_n(t) \subseteq \mathbf{M}$, and local layers $\mathbf{L}_n(t) \subseteq \mathbf{L}$. Additionally, each node has a CPU core number p_n, memory capacity e_n, bandwidth b_n, and storage capacity d_n. A node can run a maximum of C_n containers simultaneously.

Layer: The set of layers in container $c \in \mathbf{C}$ is $\mathbf{L}_c = \{x_c^l \mid l \in \mathbf{L}\}$, where $x_c^l = 1$ if container c contains layer l, and $x_c^l = 0$ otherwise. The size of layer $l \in \mathbf{L}$ is d_l.

Task: For each task $k \in \mathbf{K}$ generated by a user at time t, the requested CPU resource is p_k and the requested container is c_k. After scheduling, the node assigned to this task is $n_k = \{u_k^n \mid n \in \mathbf{N}\}$, where $u_k^n = 1$ if task k is scheduled to node n, otherwise $u_k^n = 0$.

3.2 Modeling of cost and score

In edge computing, limited bandwidth and large image sizes result in significant download cost when deploying containers. Compared to this, container startup cost is minimal (Tang et al. 2023). Therefore, our paper focuses on download cost .

For task k requesting container c, the requested layers are $\mathbf{L}_c$. At time t, the layers stored on edge node n are $\mathbf{L}_n(t)$. The layers from $\mathbf{L}_c$ found on node n are $\mathbf{L}_c \cap \mathbf{L}_n(t)$. The download cost $\mathcal{C}_c^n(t)$ for deploying container c on node n is:

$$\mathcal{C}_c^n(t) = \sum_{l \in \mathbf{L}_c \setminus \mathbf{L}_n(t)} d_l. \tag{1}$$

The download time for node n can be obtained as

$$\mathcal{T}^{k,n} = \frac{\mathcal{C}_k^n(t)}{b_n}. \tag{2}$$

Moreover, the total size $\mathcal{D}_c^n(t)$ of local layers for node n is:

$$\mathcal{D}_c^n(t) = \sum_{l \in \mathbf{L}_c \cap \mathbf{L}_n(t)} d_l.$$

Assume that the maximum score before weighting for each node is denoted as MaxScore (Carrión 2022). Then, the layer sharing score $\mathcal{S}_{\text{Layer}}^{k,n}(t)$ of node n at time t is calculated as follows:

$$\mathcal{S}_{\text{Layer}}^{k,n}(t) = \frac{\mathcal{D}_c^n(t)}{\sum_{l \in \mathbf{L}_c} d_l} \times \text{MaxScore}. \tag{4}$$

The layer reuse index r is used to measure the utilization of image layer resources:

$$r = \left(A_{l \in \mathbf{L}_c \cap \mathbf{L}_n(t)} \times 0.3\right) + \left[\mathcal{D}_c^n(t) \times 0.7\right]. \tag{5}$$

To calculate the optimal deployment mechanism score, we first consider the set of the five most recent tasks, including the current one, deployed in the cluster. For each task, the ratio of requested memory e_i to requested CPU p_i is calculated for $i \in [1, 5]$. The ratio $\frac{e_6}{p_6}$ is calculated for the node's remaining resources. Then, the standard deviation $\text{STD}_{\text{Eval}}^{k,n}(t)$ of the resource demands and node resources is calculated to measure their correlation:

$$\text{STD}_{\text{Eval}}^{k,n}(t) = \sqrt{\frac{1}{6} \sum_{i=1}^{6} (x_i - \mu)^2}, \tag{6}$$

where μ is the average ratio of local resources and historical task resource demands, obtained as:

$$\mu = \frac{1}{6} \sum_{i=1}^{6} \frac{e_i}{p_i}. \tag{7}$$

Using the standard deviation $\text{STD}_{\text{Eval}}^{k,n}(t)$, the optimal deployment mechanism score $\mathcal{S}_{\text{Eval}}^{k,n}(t)$ of node n is calculated as follows:

$$\mathcal{S}_{\text{Eval}}^{k,n}(t) = \text{MaxScore} \times \left(1 - \text{STD}_{\text{Eval}}^{k,n}(t)\right). \tag{8}$$

To reduce the occurrence of load imbalance, the resource balancing score $\mathcal{S}_{\text{Bal}}^{k,n}(t)$ is modified as follows:

$$\begin{aligned} \mathcal{S}_{\text{Bal}}^{k,n}(t) = &\text{MaxScore} \times \Bigg(1 - \text{STD}_{\text{Node}}^{k,n}(t) - \frac{p_n(t)}{p_n} \\ &- \frac{e_n(t)}{e_n} - 0.5 \times \frac{q_n(t)}{q_n}\Bigg), \end{aligned} \tag{9}$$

where $\text{STD}_{\text{Node}}^{k,n}(t)$ is the system resource standard deviation:

$$\text{STD}_{\text{Node}}^{k,n}(t) = 0.5 \times \left| \frac{p_n(t)}{p_n} - \frac{e_n(t)}{e_n} \right|. \tag{10}$$

The above scores of the scheduler are then combined using dynamic weights. The weight of the layer sharing score is denoted as $\mathcal{W}_{\text{Layer}}^{k,n}(t)$,

$$\mathcal{W}_{\text{Layer}}^{k,n}(t) = \frac{\mathcal{D}_c^n(t)}{\frac{2}{\text{STD}_S(t)}} = \frac{\mathcal{D}_c^n(t) \times \text{STD}_S(t)}{2}, \tag{11}$$

where $\text{STD}_S(t) = \min(\text{STD}_{\text{Node}}^{k,n}(t), \text{STD}_{\text{Eval}}^{k,n}(t))$.

The weight of the optimal deployment mechanism score $\mathcal{W}_{\text{Eval}}^{k,n}(t)$ is:

$$\mathcal{W}_{\text{Eval}}^{k,n}(t) = \frac{\frac{2}{\text{STD}_{\text{Eval}}^{k,n}(t)}}{\mathcal{D}_c^n(t)} = \frac{2}{\text{STD}_{\text{Eval}}^{k,n}(t) \times \mathcal{D}_c^n(t)}. \tag{12}$$

The weight of the resource balancing mechanism score $\mathcal{W}_{\text{Bal}}^{k,n}(t)$ is:

$$\mathcal{W}_{\text{Bal}}^{k,n}(t) = \frac{\frac{2}{\text{STD}_{\text{Node}}^{k,n}(t)}}{\mathcal{D}_c^n(t)} = \frac{2}{\text{STD}_{\text{Node}}^{k,n}(t) \times \mathcal{D}_c^n(t)}. \tag{13}$$

Moreover, the evaluation score of the default Kubernetes scheduler is denoted as $\mathcal{S}_{\text{K8s}}^{k,n}(t)$. The weighted score $\mathcal{S}^{k,n}(t)$ (with weights satisfying $w \in [0, 5]$) can be calculated as:

$$\begin{aligned} \mathcal{S}^{k,n}(t) = &\mathcal{W}_{\text{Layer}}^{k,n}(t) \times \mathcal{S}_{\text{Layer}}^{k,n}(t) + \mathcal{W}_{\text{Eval}}^{k,n}(t) \times \mathcal{S}_{\text{Eval}}^{k,n}(t) \\ &+ \mathcal{W}_{\text{Bal}}^{k,n}(t) \times \mathcal{S}_{\text{Bal}}^{k,n}(t) + \mathcal{S}_{\text{K8s}}^{k,n}(t). \end{aligned} \tag{14}$$

The node n_k for task k is selected as the scheduling node with the highest score:

$$n_k = \arg\max_n \mathcal{S}^{k,n}(t). \tag{15}$$

3.3 Layer-aware and request-aware problem

Constraints: During the scheduling process, constraints are used for prefiltering and filtering plugins. The storage capacity of each node must satisfy:

$$C_c^n(t) + \sum_{l \in L_n(t)} d_l \le d_n, \quad \forall t, \forall n. \tag{16}$$

Moreover, the running container number limit is as follows:

$$|\mathbf{C}_n(t)| \le C_n. \tag{17}$$

And each task should only be scheduled to one node:

$$\sum_{n \in \mathbf{N}} u_k^n = 1, \quad \forall k. \tag{18}$$

Problem statement: The goal of the layer-aware scheduler is to minimize the download cost, i.e., to maximize the layer sharing score $\mathcal{S}_{\text{Layer}}^{k,n}(t)$. The problem can be defined as follows:

$$\begin{aligned} &\max \mathcal{S}_{\text{Layer}} = \sum_{k \in \mathbf{K}} \mathcal{S}_{\text{Layer}}^{k,n}(t), \\ &\quad \text{s.t. Eqs.}(16), (17), (18). \end{aligned} \tag{19}$$

Similarly, the goal of resource demand adaptation is to maximize the deployable task volume, i.e., to maximize the evaluation score $\mathcal{S}_{\text{Eval}}^{k,n}(t)$. The problem can be defined as follows:

$$\begin{aligned} &\max \mathcal{S}_{\text{Eval}} = \sum_{k \in \mathbf{K}} \mathcal{S}_{\text{Eval}}^{k,n}(t), \\ &\text{s.t. Eqs.} \quad (16), (17), (18). \end{aligned} \tag{20}$$

By integrating resource demand adaptation, layer sharing scores, and other scheduling plugins, this problem can adapt to different forms. For example, when combined with the default Kubernetes scheduler:

$$\begin{aligned} &\max \mathcal{S} = \sum_{k \in \mathbf{K}} \mathcal{S}^{k,n}(t), \\ &\text{s.t. Eqs.} \quad (16), (17), (18). \end{aligned} \tag{21}$$

4 Proposed design of LR²Scheduler

4.1 LR²Scheduler

The LR²Scheduler algorithm, as shown in Algorithm 1, takes task k and a set of edge nodes $\mathbf{N}$ as input and outputs the selected node n_k for container deployment. First, the scores are initialized to 0. Then, the scores for the three scheduling strategies-layer sharing, resource demand adaptation, and node resource balancing-are calculated based on Eqs. (4), (8), (9), respectively. Next, the weights of the above three strategies are calculated based on Eqs. (11), (12), (13), dynamically balancing the system's existing resources with user demands. Finally, the weighted scores of these three strategies are combined with the evaluation scores of the Default Scheduler plugin (as in Eq. (14)). Each task is deployed to the node with the highest score.

Algorithm 1 LR²Scheduler

Input : k, $\mathbf{N}$
Output: n_k
Initialize $\mathcal{S}^{k,n}(t) \leftarrow 0, \forall k, \forall n$;
Update layer information from Registry;
for $n \leftarrow n_1, n_2, ..., n_{|\mathbf{N}|}$ **do**

```
    // Layer sharing score
    Calculate layer sharing score by Eq. (4);
    // Request evaluation score
    Calculate request evaluate score by Eq. (8);
    // Resource balance score
    Calculate resource balance score by Eq. (9);
    // Layer sharing weight
    Calculate layer sharing weight by Eq. (11);
    // Request evaluation weight
    Calculate request evaluation weight by Eq. (12);
    // Resource balance weight
    Calculate resource balance weight by Eq. (13);
    Get S_K8s^{k,n}(t) from Kubernetes default scheduler;
    // LR2Scheduler score
    Calculate the final score S^{k,n}(t) by Eq. (14);
```

Select and return the node n_k by Eq. (15);
end

4.2 Scalability of LR²Scheduler

Next, we discuss the extensibility of LR²Scheduler. As shown in Algorithm 1, LR²Scheduler first evaluates the scores for resource demand adaptation and layer sharing. Then, it calculates dynamic weights and combines the scores with other scheduling plugins to obtain the final weighted score. The method of adjusting dynamic weights can be easily extended to allow LR²Scheduler to work with any scheduling plugin, ensuring the performance of other schedulers while minimizing container deployment costs. The extensibility of LR²Scheduler is mainly reflected in three aspects: the conditions for dynamic weight adjustment, the values of dynamic weights, and the combination of schedulers. Details are as follows:

Conditions for dynamic weight adjustment: The weights of the three strategies in Algorithm 1 consider the node's resource demand adaptation, resource balancing, and layer sharing scores. In fact, other factors can also be taken into account. For example, storage space, memory, GPU resources, and node availability labels can be further analyzed to enhance dynamic weight adjustment methods.

Values for dynamic weight: This method can also be used to adjust dynamic weights to extend LR²Scheduler. For example, it can set other weights that better match individual needs. Moreover, we can add more conditions or piecewise functions, like a function $\omega = f(S^{k,n}_{\mathrm{Weight}}(t))$ or a neural network to adjust the weight.

Combining schedulers: LR²Scheduler can also be integrated with other Kubernetes schedulers. In the next section, we will discuss the implementation of LR²Scheduler. We have combined LR²Scheduler with some default plugins, as shown below:

1. `ImageLocality` that prefers nodes with the container images already present.
2. `TaintToleration` that implements taints and tolerations, reducing deployment priority for tainted nodes.
3. `NodeAffinity` that implements node selectors and affinity, scoring nodes higher that meet more affinity conditions. Preference is given to nodes that satisfy the specified rules.
4. `PodTopologySpread` that implements container topology spread by selecting the node with the highest score for each topology pair.
5. `NodeResourcesFit` that verifies if the node has all the resources requested by the container. The default strategy is `LeastAllocated`.
6. `VolumeBinding` that verifies if the node can bind the requested volumes, prioritizing the smallest volume that meets the required size.
7. `InterPodAffinity` that implements inter-Pod affinity and anti-affinity similar to `NodeAffinity`.

Notably, the plugins mentioned above can be enabled or disabled individually, and they can also be combined in various ways to achieve different effects. The main extension point of LR²Scheduler is the score; by integrating resource allocation strategies and layer sharing into the final differentiation, it can adapt to various scheduling requirements while minimizing container deployment costs. Overall, LR²Scheduler has distinctive extensibility.

5 System implementation

As shown in Fig. 2, LR²Scheduler is implemented within the Kubernetes system using the scheduling framework (Scheduling Framework 2024). LR²Scheduler is deployed to the system using `deployment` (Deployments 2024). First, the user sends a container deployment request, specifying the container and resource limits, and sets the scheduler to LR² Scheduler. Upon receiving the request, the Kubernetes API Server invokes LR²Scheduler for scheduling. LR²Scheduler first updates the layer information from the registry, then performs layer matching and scoring. Next, LR²Scheduler starts resource matching and evaluation. After the evaluation is completed, it calculates the dynamic weights and final scores, as detailed in Algorithm 1. Once the score is obtained, the Kubernetes API selects the node with the highest container deployment score to complete the entire scheduling process. Here are some key details in the implementation process of the LR²Scheduler as shown in Fig. 2.

① *Update layer information from Registry*. Existing methods cannot automatically retrieve layer information due to challenges in real-time reading and parsing, unstable bandwidth causing connection interruptions in edge computing, and read permission issues from container isolation (Fu et al. 2020). Currently, there is no automatic way to query mirror layer information. We address these issues by creating a `goroutine` to periodically fetch all images and their tags from the Docker registry's /v2/_catalog endpoint. At service start, the `Registry` class initializes. The method Registry Watcher is called, and it waits for 10 s by default to access the registration interface. It filters layer IDs and sizes, stores the data keyed by image name and tag in a JSON file as shown in Listing 1, and uses this cached file as the metadata to compare image sizes through layer information lookup. The retrieved data is formatted into a map[string]ImageMetadata structure and saved in the cache.json file.

② *Match and score layers*.Determining the size of the layers and aligning them is challenging. Due to the storage structure, we cannot directly obtain layer size from the

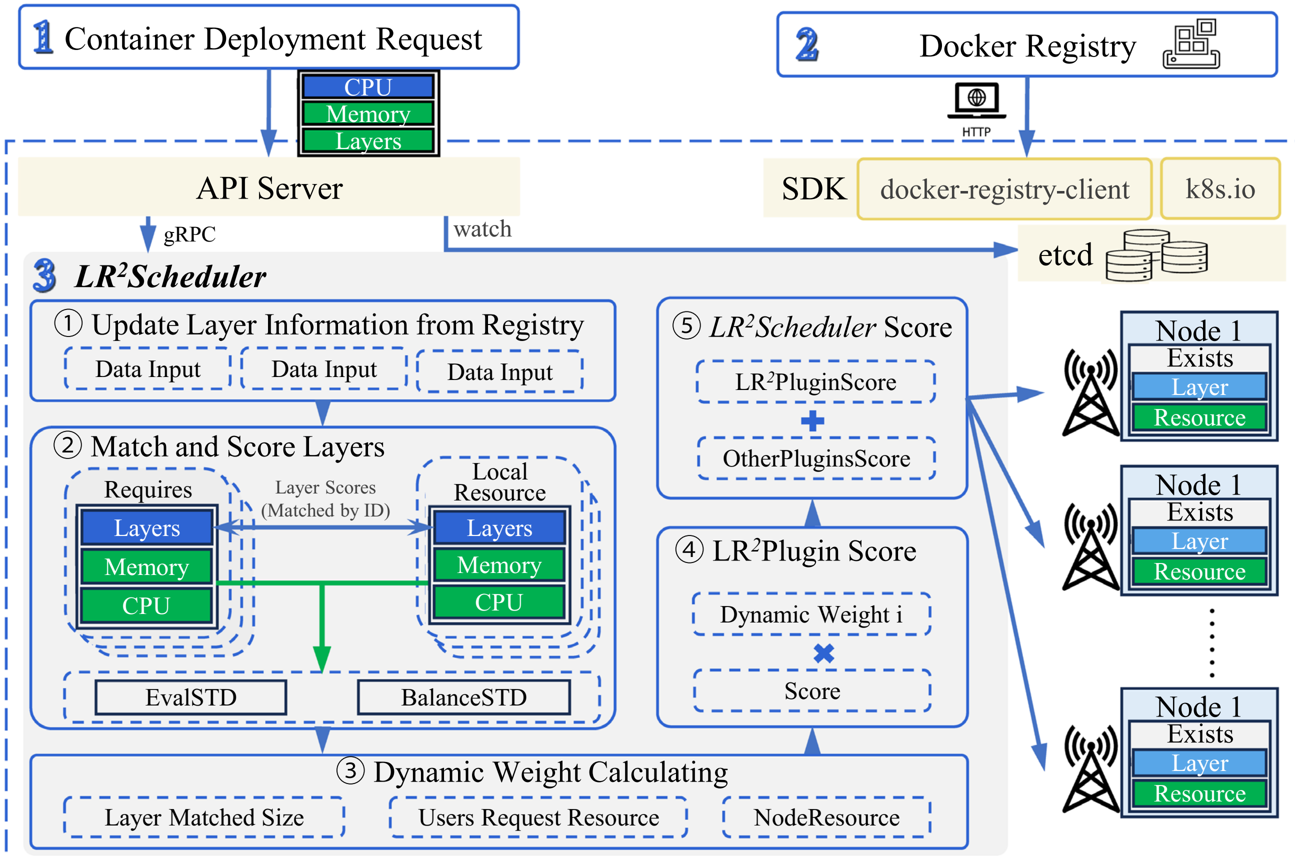


Fig. 2 LR²Scheduler implementation in Kubernetes system

image ID. Therefore, we utilize the `cache.json` file as follows:

1. The scheduler retrieves scheduled container information from *k8s.io/api/core/v1.Pod. The image name and tag are accessible via pod.spec.Containers[].Image.
2. To obtain the layer sizes from the Registry metadata, we use the image and tag as keys to search the `cache.json` file, returning the layer information `ImageMetadata` for that image.
3. Extract layer information from cached image names and tags in the `cache.json` file.
4. To calculate the node score, the node information including available resource and local images is obtained using the `Handle` method from the base class (framework.ScorePlugin), specifically k8s.io/kubernetes/pkg/scheduler/framework.Handle. This includes the node's IP address. By calling the Docker API at http://IP:2375, all cached images can be retrieved.
5. Compare the container layers from step 2 with the cached layers from step 4, extract the matching cached data, and calculate the total cached layer size.

③ *Dynamic Weight Calculation*. The challenge is how to determine the suitable weights and adjustments based on different needs as discussed in Sect. 3.2. LR²Scheduler calculates dynamic weights through the following steps:

1. Using node information (*k8s.io/kubernetes/pkg/scheduler/framework.Nod eInfo), we can access detailed information about all available resources on the current node and all running containers. This includes: the usage percentage of resources (CPU, memory, storage), the number of running Pods, and the resource consumption of each Pod.
2. Calculate the available CPU and memory percentages by dividing the total requested resources of all containers by the node's available resources. Then, compute the standard deviation (STD).
3. Return different weights based on Eqs. (11), (12), (13).

```
// Request Queue
type PodQueue struct{
items [] string itemsCpu [] float64
itemsMem [] float64
size int
}
// Single Layer
type LayerMetadata struct {
    Size  int64  `json:"size"`
    Layer string `json:"layer"`
}
// Single Image
type ImageMetadata struct {
    Id              string    `json:"id"`
    Name            string    `json:"name"`
    NameWithoutRepo string    `json:"name_without_repo"`
    Tag             string    `json:"tag"`
    TotalSize       int64     `json:"total_size"`
    LayerMetadata   []LayerMetadata   `json:"l_meta"`
}
// All Images
type ImageMetadataLists struct {
    CatchFile string
    Lists     map[string]ImageMetadata
}
```

Listing 1 Data Structure

6 Experiments

6.1 Experimental settings

To verify LR^{2}Scheduler, we set up a Kubernetes cluster with 1 master node and 4 worker nodes. All the nodes have Linux CentOS 7 installed. The Kubernetes version used is $v1.23.8$. The container runtime is Docker with version 20.10.8. The Kubelet and Kube-proxy versions are both $v1.23.8$. All nodes have 4-core CPUs. The master node has 8GB of memory and a 60GB hard drive. Worker node 1 has 4GB of memory and a 30GB hard drive. Worker node 2 has 2GB of memory and a 30GB hard drive. Worker nodes 3 and 4 each have 4GB of memory and a 20GB hard drive. The custom scheduler is implemented in Go language, version $go1.18 linux/amd64$.

We have deployed a private repository using Docker registry. We select some images from Docker Hub and upload them to our private repository, including WordPress, Ghost, GCC, Redis, Tomcat, MySQL, etc. During the experiments, we randomly request these images, setting random CPU and memory limits for each request. Each image consists of several layers, and the information about these layers can be retrieved from the registry. We conduct multiple experiments by deploying different numbers of workers and setting various bandwidth limits.

The experiments compare LR^{2}Scheduler with the Default Scheduler and the Static Layer scheduler. The Default Scheduler enables scheduling plugins as described in Sect. 3.2. The Static Layer Scheduler uses the layer-aware scheduling plugin as a baseline, with a weight setting of 2 while weights of other plugins are 1. The maximum score MaxScore for the node before weighted is set to 100.

6.2 Experimental results

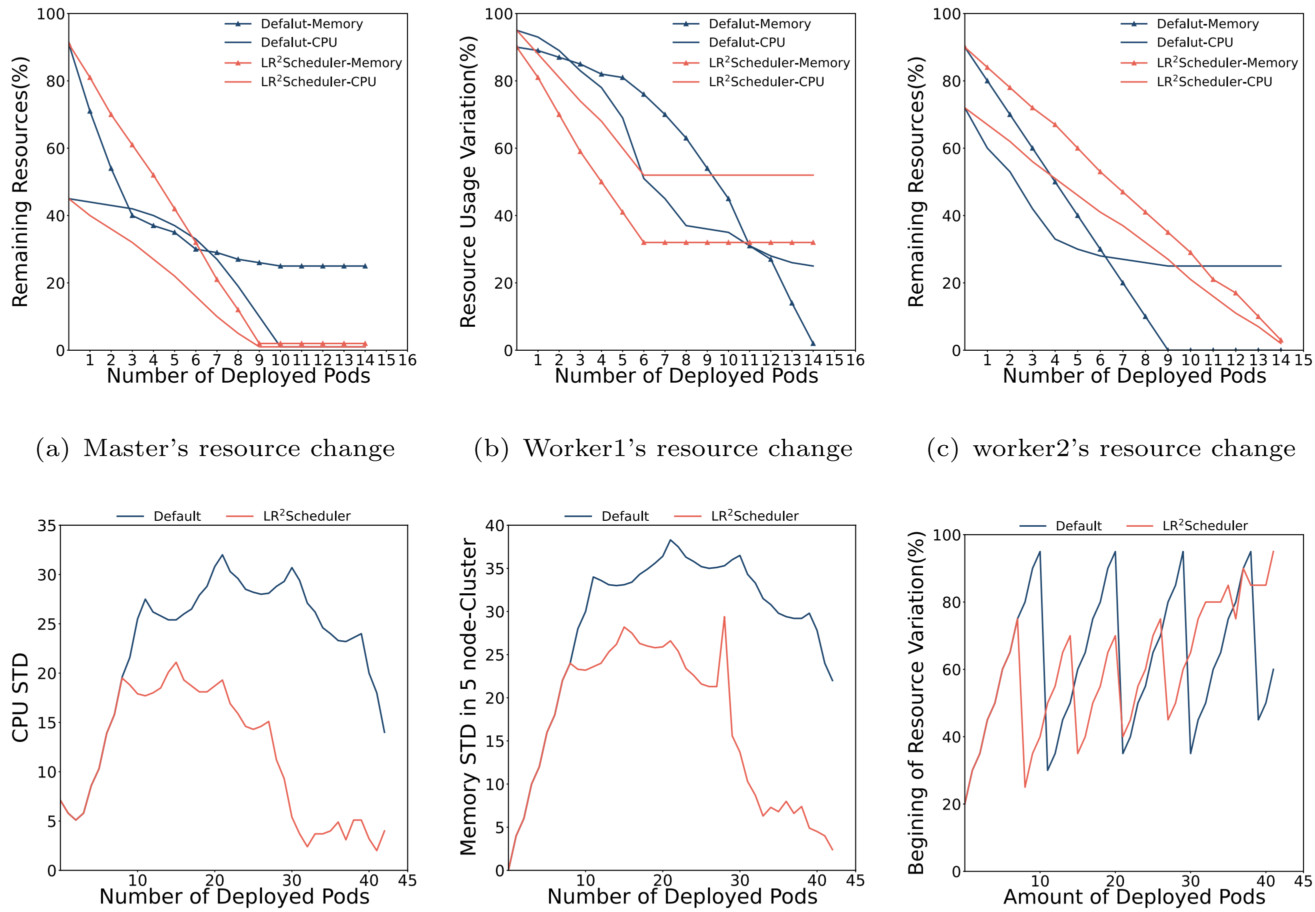
Performance with different number of pods. Kubernetes operates on Pods, which in our case are equivalent to single-container Pods. Figure 3a–c show that due to the Default Scheduler being a local optimization algorithm, it easily generates resource fragments. When some resources have been completely consumed, it will leave other resources unusable, while using LR^{2}Scheduler to schedule nodes can generate less resource fragmentation. Figure 3d and e demonstrate that in the scheduling process, compared to the Default Scheduler, LR^{2}Scheduler can maintain a lower level of standard deviation, indicating that LR^{2}Scheduler can help clusters achieve better balance. Additionally, the resource balance of the nodes after scheduling is significantly better than that achieved with the Default Scheduler, reducing the occurrence of load imbalance. Figure 3f illustrates that using the LR^{2}Scheduler decreases the number of nodes with resource usage over 80% by 50% compared to the Default Scheduler for the same tasks.

Performance with different number of nodes. To evaluate the performance under different number of nodes, experiments are conducted using 3, 4, and 5 edge nodes. With the help of resource request history evaluation strategy, the total number of tasks that can be deployed in the cluster has increased. Figure 4a indicates that the LR^{2}Scheduler can deploy the most containers, averaging 24% and 50% more than the official Default Scheduler and the Layer scheduler, respectively. Figure 4b shows that, for tasks with identical configurations, the Layer Scheduler and LR^{2}Scheduler significantly reduce download volume compared to the Default Scheduler. The average reduction is respectively 39% and 35%. As shown in Fig. 4c, the Layer Scheduler reduces the average disk usage by 45%, while LR^{2}Scheduler reduces the average disk usage by 41%. Although the Layer Scheduler performs a bit better in download size, the LR^{2}Scheduler can dynamically adjust the weights of different scheduling strategies, effectively balancing resource allocation. This is particularly evident in cluster resource balance. As shown in Fig. 4d, the slight advantage of the Layer Scheduler in metrics such as download volume comes at a significant cost to cluster resource balance (according to Eq. (10)), resulting in an average reduction of 33% in the number of deployable tasks compared to the LR^{2}Scheduler.

Performance with different bandwidth. Fig. 5 shows the download time at various bandwidths. It is clear that LR^2

Table 1 Performance analysis for 20 containers

#	Scheduler	Size (MB)	Reusage	STD	#	Scheduler	Size (MB)	Reusage	STD
1	Default	3	22.6	0.02	11	Default	3	26.42	0.11
	Layer	1	0.9	0.07		Layer	1	3.6	0.27
	LR^{2}Scheduler	3	0	0.02		LR^{2}Scheduler	4	2.4	0.15
2	Default	490	177.76	0.03	12	Default	474	141	0.14
	Layer	434	202.4	0.05		Layer	141	275.9	0.31
	LR^{2}Scheduler	434	72.2	0.05		LR^{2}Scheduler	141	233.6	0.18
3	Default	380	122.38	0.03	13	Default	164	110.71	0.19
	Layer	201	187.9	0.02		Layer	164	167.9	0.26
	LR^{2}Scheduler	201	127.6	0.02		LR^{2}Scheduler	164	118.7	0.21
4	Default	160	77.17	0.04	14	Default	52	42.4	0.23
	Layer	111	90.3	0.1		Layer	22	16.5	0.29
	LR^{2}Scheduler	111	71.58	0.06		LR^{2}Scheduler	55	0	0.24
5	Default	15	28.55	0.08	15	Default	37	36.65	0.17
	Layer	15	24	0.12		Layer	28	32.1	0.32
	LR^{2}Scheduler	15	19.5	0.08		LR^{2}Scheduler	29	19.5	0.21
6	Default	6	29.32	0.12	16	Default	356	75.48	0.19
	Layer	6	5.1	0.16		Layer	6	1.8	0.36
	LR^{2}Scheduler	9	2.4	0.09		LR^{2}Scheduler	6	251.6	0.25
7	Default	416	162.29	0.15	17	Default	518	77.59	0.21
	Layer	416	154.1	0.2		Layer	99	80.6	0.35
	LR^{2}Scheduler	416	29.3	0.13		LR^{2}Scheduler	189	47.3	0.21
8	Default	285	90.74	0.12	18	Default	238	64.6	0.17
	Layer	66	174.6	0.24		Layer	208	107	0.3
	LR^{2}Scheduler	66	154.8	0.18		LR^{2}Scheduler	228	99.8	0.26
9	Default	54	34.59	0.14	19	Default	113	32.81	0.19
	Layer	24	27.8	0.27		Layer	28	15.8	0.33
	LR^{2}Scheduler	24	14.3	0.16		LR^{2}Scheduler	22	1.7	0.23
10	Default	49	21.93	0.13	20	Default	46	48.19	0.24
	Layer	21	33.9	0.3		Layer	2	16.7	0.34
	LR^{2}Scheduler	49	24.12	0.15		LR^{2}Scheduler	50	33.63	0.22



(a) Master's resource change
(b) Worker1's resource change
(c) worker2's resource change
(d) CPU balance variation
(e) Memory balance variation
(f) Resource change's baseline

Fig. 3 Performance with different numbers of deployed pods

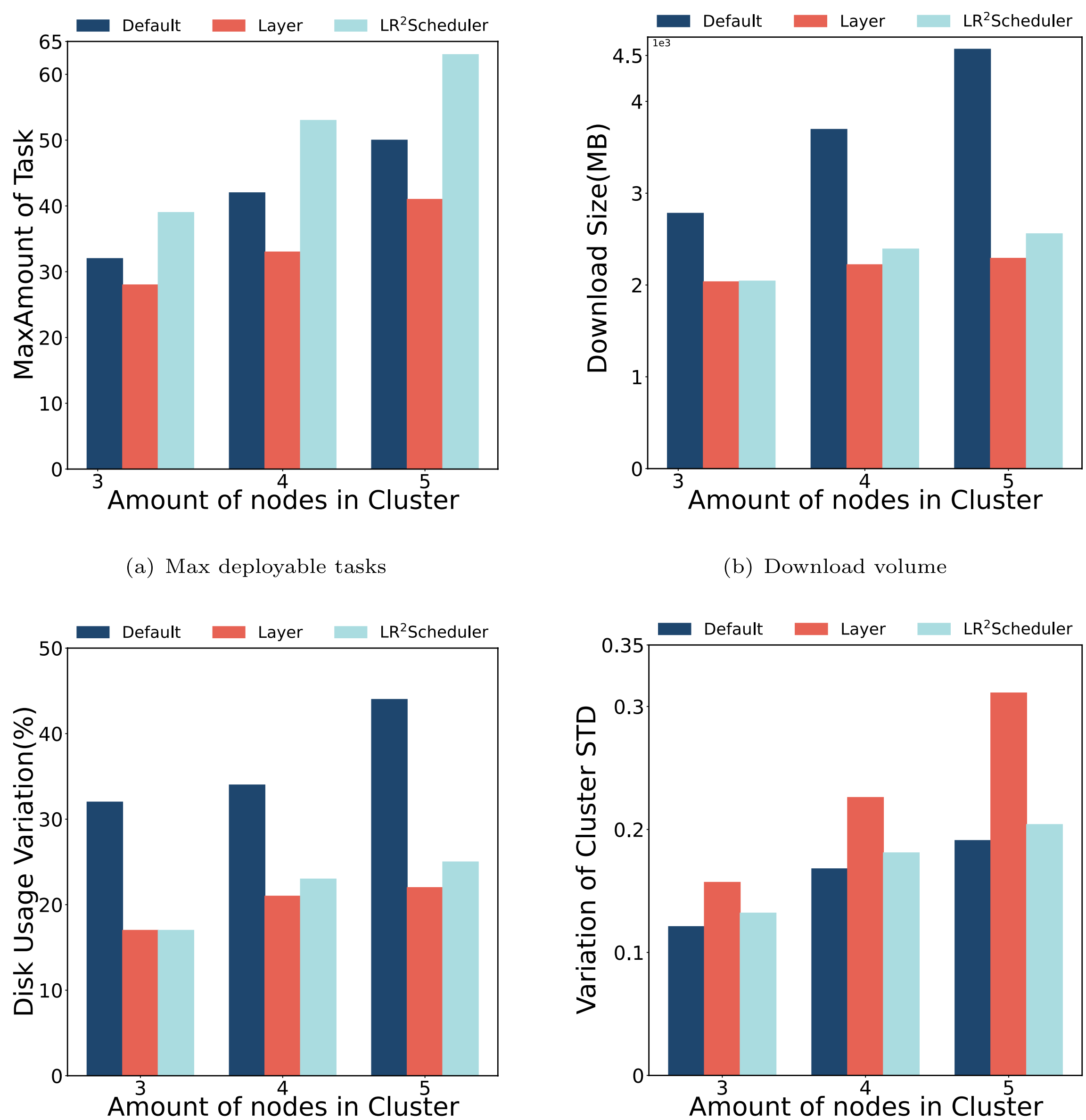
Scheduler has a more pronounced advantage when the edge network bandwidth is low. Overall, compared to the Default Scheduler, LR²Scheduler reduces the average download time by 47%. Due to the combination of layer scheduling plugins, LR²Scheduler shows a significant improvement over the Default Scheduler. Figure 6 shows that both Layer Scheduler and LR²Scheduler demonstrate significantly higher cumulative reuse index compared to the default scheduler as the number of deployed containers increases. LR²Scheduler's effectiveness is further demonstrated by its ability to consider additional metrics, such as resource balancing.

Moreover, as shown in Table 1, we have detailed the download size, reusage, and resource balancing (STD) for deploying 20 containers. While LR²Scheduler may not have the smallest download size at each step, it ultimately results in almost the lowest total download cost and reusage while considering resource balancing, demonstrating its long-term effectiveness despite room for improvement. Besides the scalability discussed in Sect. 4.2, reinforcement learning algorithms can also be considered to optimize container deployment costs by accounting for long-term benefits.

In summary, the LR²Scheduler effectively reduces download costs while maintaining efficient resource utilization. Additionally, it allows for the selection of different scheduling strategies or adjustment of weights based on specific needs. The effectiveness of the LR²Scheduler is further reflected in its ability to consider additional metrics.

7 Conclusion

In this paper, we proposed and implemented a layer-aware, resource-balanced, and request-adaptive container scheduler for edge computing. First, we designed a user request evaluation plugin, and then integrated it with a layer-aware mechanism to form a Kubernetes scheduling scheduler



(a) Max deployable tasks

(b) Download volume

(c) Disk variation

(d) Cluster Resource Balance

Fig. 4 Performance with different number of nodes

that can effectively reduce the network transmission cost between container deployments, minimize resource fragmentation, and meet resource balance and other indicators. Finally, by using the Kubernetes scheduling framework, the LR^{2}Scheduler was implemented. The experimental results in the Kubernetes system show that this scheduler improved resource utilization rates, optimized deployment costs, and enhanced system performance. This study demonstrates that in real systems, the LR^{2}Scheduler can achieve the flexibility of shared scheduling among layers based on resource requirements, while also highlighting further optimization opportunities. In future work, we will design scheduling algorithms using reinforcement learning and other long-term optimization strategies, and implement them in Kubernetes. Moreover, we will explore cloud-edge and edge-edge collaborative layer sharing to

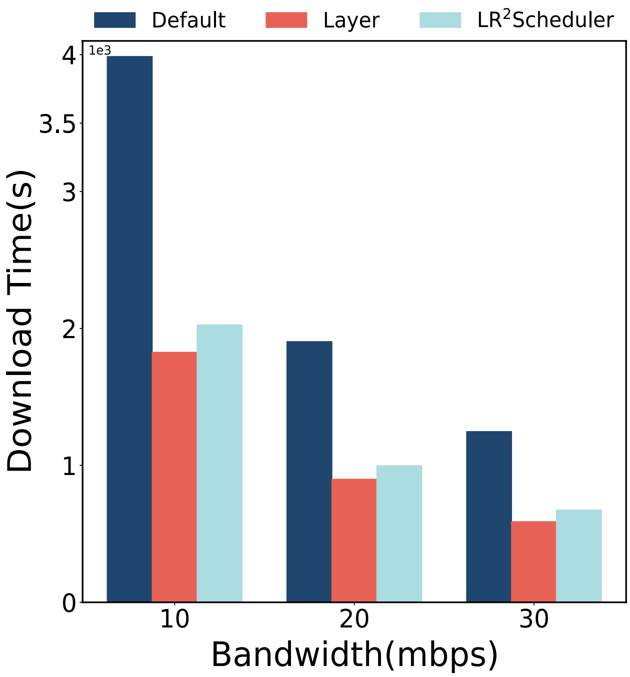


Fig. 5 Performance with different bandwidth

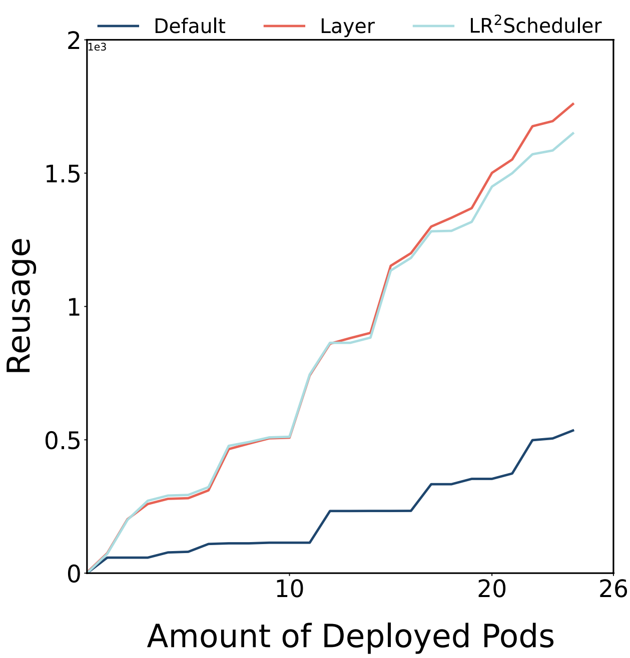


Fig. 6 Accumulated Reusage

reduce container startup time by transferring layers from other edge nodes.

Funding This work was supported in part by the Chinese National Research Fund (NSFC) under Grant 62272050 and Grant 62302048; in part by the Guangdong Key Lab of AI and Multi-modal Data Processing, United International College (UIC), Zhuhai under 2023–2024 Grants sponsored by Guangdong Provincial Department of Education; in part by the Institute of Artificial Intelligence and Future Networks (BNU-Zhuhai) and Engineering Center of AI and Future Education, Guangdong Provincial Department of Science and Technology, China; Zhuhai Science-Tech Innovation Bureau under Grant No. 2320004002772; and in part by the Interdisciplinary Intelligence SuperComputer Center of Beijing Normal University (Zhuhai).

Declarations

Conflict of interest The authors have no relevant financial or non-financial interests to disclose.

References

Akraino.: https://www.lfedge.org/projects/akraino/ (2024)

Brooker, M., Danilov, M., Greenwood, C., Piwonka, P.: On-demand container loading in {AWS$\}$ lambda. In: Proceedings of 2023 USENIX Annual Technical Conference (USENIX ATC 23), pp. 315–328 (2023)

Cai, H., Zhou, Z., Huang, Q.: Online resource allocation for edge intelligence with colocated model retraining and inference. In: Proceedings of 2024 IEEE Conference on Computer Communications (INFOCOM), pp. 1–9 (2024). IEEE

Carrión, C.: Kubernetes scheduling: taxonomy, ongoing issues and challenges. ACM Comput Surv **55**(7), 1–37 (2022)

Chen, Y., Xu, J., Wu, Y., Gao, J., Zhao, L.: Dynamic task offloading and resource allocation for noma-aided mobile edge computing: an energy efficient design. IEEE Trans Serv Comput **17**(4), 1492–1503 (2024)

Cui, H., Tang, Z., Lou, J., Jia, W., Zhao, W.: Latency-aware container scheduling in edge cluster upgrades: A deep reinforcement learning approach. IEEE Transactions on Services Computing (2024)

Deployments.: https://kubernetes.io/docs/concepts/workloads/controllers/deployment/ (2024)

Dolati, M., Rastegar, S.H., Khonsari, A., Ghaderi, M.: Layer-aware containerized service orchestration in edge networks. IEEE Trans Netw Serv Manag **20**(2), 1830–1846 (2022)

Dong, B., Yang, Q., Li, M.: M-rsf: a multilevel feedback queue task scheduling mechanism for unikernel. J Commun **45**(05), 54–69 (2024). https://doi.org/10.11959/j.issn.1000-436x.2024061

Fu, S., Mittal, R., Zhang, L., Ratnasamy, S.: Fast and efficient container startup at the edge via dependency scheduling. In: Proceedings of 3rd USENIX Workshop on Hot Topics in Edge Computing (HotEdge) (2020)

Gu, L., Zeng, D., Hu, J., Li, B., Jin, H.: Layer aware microservice placement and request scheduling at the edge. In: Proceedings

of 2021 IEEE Conference on Computer Communications (INFOCOM), pp. 1–9 (2021). IEEE
Gu, L., Huang, J., Huang, S., Zeng, D., Li, B., Jin, H.: Lopo: An out-of-order layer pulling orchestration strategy for fast microservice startup. In: Proceedings of 2023 IEEE Conference on Computer Communications (INFOCOM), pp. 1–9 (2023). IEEE
Gunasekaran, J.R., Thinakaran, P., Nachiappan, N.C., Kandemir, M.T., Das, C.R.: Fifer: Tackling resource underutilization in the serverless era. In: Proceedings of the 21st International Middleware Conference (Middleware), pp. 280–295 (2020)
K3s.: Lightweight Kubernetes. https://k3s.io (2024)
Katalyst.: https://gokatalyst.io (2024)
Li, F.-W., Wang, K., Zhu, J., et al.: Novel adapting packet scheduling of td-hsupa. J Commun **33**(5), 177–182 (2012)
Liu, Y., Yang, B., Wu, Y., Chen, C., Guan, X.: How to share: balancing layer and chain sharing in industrial microservice deployment. IEEE Trans Serv Comput **16**(4), 2685–2698 (2022)
Lou, J., Luo, H., Tang, Z., Jia, W., Zhao, W.: Efficient container assignment and layer sequencing in edge computing. IEEE Trans Serv Comput **16**(2), 1118–1131 (2022)
Ma, L., Yi, S., Carter, N., Li, Q.: Efficient live migration of edge services leveraging container layered storage. IEEE Trans Mob Comput **18**(9), 2020–2033 (2018)
Ouyang, T., Zhao, K., Zhang, X., Zhou, Z., Chen, X.: Dynamic edge-centric resource provisioning for online and offline services colocation. In: Proceedings of 2023 IEEE Conference on Computer Communications (INFOCOM), pp. 1–10 (2023). IEEE
QoS Based Scheduling System Koordinator.: https://koordinator.sh (2024)
Rejiba, Z., Chamanara, J.: Custom scheduling in Kubernetes: a survey on common problems and solution approaches. ACM Comput Surv **55**(7), 1–37 (2022)
Rong, C., Wang, J.H., Liu, J., Yu, T., Wang, J.: Exploring the layered structure of containers for design of video analytics application migration. In: 2022 IEEE Wireless Communications and Networking Conference (WCNC), pp. 842–847 (2022). IEEE
Scheduler Configuration.: https://kubernetes.io/docs/reference/scheduling/config/ (2024)
Scheduling Framework.: https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/ (2024)
Shi, W., Cao, J., Zhang, Q., Li, Y., Xu, L.: Edge computing: vision and challenges. IEEE Internet Things J **3**(5), 637–646 (2016)
Tang, Z., Lou, J., Jia, W.: Layer dependency-aware learning scheduling algorithms for containers in mobile edge computing. IEEE Trans Mob Comput **22**(6), 3444–3459 (2023)
Tang, Z., Peng, W., Guo, J., Lou, J., Cui, H., Wang, T., Wu, Y., Jia, W.: Lrscheduler: A layer-aware and resource-adaptive container scheduler in edge computing. In: Proceedings of the 2024 20th International Conference on Mobility, Sensing and Networking (MSN), pp. 244–251 (2024). IEEE
Tang, Z., Mou, F., Lou, J., Jia, W., Wu, Y., Zhao, W.: Multi-user layer-aware online container migration in edge-assisted vehicular networks. IEEE/ACM Trans Netw **32**(2), 1807–1822 (2024)
Volcano.: https://volcano.sh (2024)
Wang, T., Liang, Y., Zhang, Y., Zheng, X., Arif, M., Wang, J., Jin, Q.: An intelligent dynamic offloading from cloud to edge for smart iot systems with big data. IEEE Trans Netw Sci Eng **7**(4), 2598–2607 (2020)
Wang, T., Liu, Y., Zheng, X., Dai, H.-N., Jia, W., Xie, M.: Edge-based communication optimization for distributed federated learning. IEEE Trans Netw Sci Eng **9**(4), 2015–2024 (2021)
Xie, X., Zhang, Z., Wang, J., et al.: Cloud resource prediction model based on triple exponential smoothing method and temporal convolutional network. J Commun **40**(8), 143–150 (2019). https://doi.org/10.11959/j.issn.1000-436x.2019172
Xing, T., Barbalace, A., Olivier, P., Karaoui, M.L., Wang, W., Ravindran, B.: H-container: enabling heterogeneous-isa container migration in edge computing. ACM Trans Comput Syst **39**(1–4), 1–36 (2022)
Xing, T., Cui, H., Chen, Y., Luo, Z., Guo, B., Yu, Z., Guo, X., Ma, Y.: Harnessing edge computing resources for accelerating industrial tasks. In: Proceedings of 2023 19th International Conference on Mobility, Sensing and Networking (MSN), pp. 652–659 (2023). IEEE
Xiong, Y., Sun, Y., Xing, L., Huang, Y.: Extend cloud to edge with kubeedge. In: Proceedings of 2018 IEEE/ACM Symposium On Edge Computing (SEC), pp. 373–377 (2018). IEEE
Xu, W., Yu, J., Wu, Y., Tsang, D.H.-K.: Joint channel estimation and reinforcement learning-based resource allocation of intelligent reflecting surface-aided multicell mobile edge computing. IEEE Internet Things J **11**(7), 11862–75 (2023)
Xu, C., Guo, J., Li, Y., Zou, H., Jia, W., Wang, T.: Dynamic parallel multi-server selection and allocation in collaborative edge computing. IEEE Trans Mob Comput **23**(11), 10523–10537 (2024)
Zhao, N., Tarasov, V., Albahar, H., Anwar, A., Rupprecht, L., Skourtis, D., Paul, A.K., Chen, K., Butt, A.R.: Large-scale analysis of docker images and performance implications for container storage systems. IEEE Trans Parallel Distrib Syst **32**(4), 918–930 (2020)
Zhu, Z., Liu, Q., Liu, D., et al.: Research progress of mimic multi-execution scheduling algorithm. J Commun **42**(5), 179–190 (2021). https://doi.org/10.11959/j.issn.1000-436x.2021072

Wentao Peng received the B.S. degree in Information Engineering from Guangdong University of Technology, China, in 2023. He is currently pursuing an M.Phil. degree with the Department of Computer Science, BNU-HKBU United International College, Zhuhai, China. He is supervised by Prof. Weijia Jia, and his research interests include mobile edge computing, Large Language Models, and Embodied Intelligence.

Zhiqing Tang received the B.S. degree from School of Communication and Information Engineering, University of Electronic Science and Technology of China, China, in 2015 and the Ph.D. degree from Department of Computer Science and Engineering, Shanghai Jiao Tong University, China, in 2022. He is currently an Assistant Professor with the Advanced Institute of Natural Sciences, Beijing Normal University, China. He is a member of CCF/IEEE/ACM. He has published more than 30 peer-reviewed papers and been the reviewer for many famous international journals/conferences. His current research interests include edge computing, resource scheduling, and reinforcement.

Jianxiong Guo received his Ph.D. degree from the Department of Computer Science, University of Texas at Dallas, Richardson, TX, USA, in 2021, and his B.E. degree from the School of Chemistry and Chemical Engineering, South China University of Technology, Guangzhou, China, in 2015. He is currently an Associate Professor with the Advanced Institute of Natural Sciences, Beijing Normal University, and also with the Guangdong Key Lab of AI and Multi-Modal Data Processing, BNU-HKBU United International College, Zhuhai, China. He is a member of IEEE/ACM/CCF. He has published more than 80 peer-reviewed papers and been the reviewer for many famous international journals/conferences. His research interests include social networks, wireless sensor networks, combinatorial optimization, and machine learning.

Jiong Lou received the B.S. degree and Ph.D. degree in the Department of Computer Science and Engineering, Shanghai Jiao Tong University, China, in 2016 and 2023. Since 2023, he has held the position of Research Assistant Professor in the Department of Computer Science and Engineering, Shanghai Jiao Tong University, China. He has published more than ten papers in leading journals and conferences (e.g., ToN, TMC and TSC). His current research interests include edge computing, task scheduling and container management.

Tian Wang received his BSc and MSc degrees in Computer Science from the Central South University in 2004 and 2007, respectively. He received his PhD degree in City University of Hong Kong in Computer Science in 2011. Currently, he is a professor in the Institute of Artificial Intelligence and Future Networks, Beijing Normal University. His research interests include internet of things, edge computing and mobile computing. He has 27 patents and has published more than 200 papers in high-level journals and conferences. He has more than 16000 citations, according to Google Scholar. His H-index is 73.

Weijia Jia (Fellow, IEEE) is currently a Chair Professor, Director of BNU-UIC Institute of Artificial Intelligence and Future Networks, Beijing Normal University (Zhuhai) and VP for Research of BNUHKBU United International College (UIC) and has been the Zhiyuan Chair Professor of Shanghai Jiao Tong University, China. He was the Chair Professor and the Deputy Director of State Kay Laboratory of Internet of Things for Smart City at the University of Macau. He received BSc/MSc from Center South University, China, in 82/84 and Master of Applied Sci./PhD from Polytechnic Faculty of Mons, Belgium in 92/93, respectively, all in computer science. From 93-95, he joined German National Research Center for Information Science (GMD) in Bonn (St. Augustine) as a research fellow. From 95-13, he worked at City University of Hong Kong as a professor. His contributions have been recognized as optimal network routing and deployment, anycast and QoS routing, sensors networking, AI (knowledge relation extractions; NLP, etc.), and edge computing. He has over 600 publications in the prestige international journals/conferences and research books, and book chapters. He has received the best product awards from the International Science & Tech. Expo (Shenzhen) in 2011/2012 and the 1st Prize of Scientific Research Awards from the Ministry of Education of China in 2017 (list 2). He has served as area editor for various prestige international journals, chair and PC member/skeynote speaker for many top international conferences. He is the Fellow of IEEE and the Distinguished Member of CCF.