Edge Computing Management with Collaborative Lazy Pulling for Accelerated Container Startup

Chiao-Cheng Chen, Yao Chiang, *Member, IEEE,* Yu-Chieh Lee, and Hung-Yu Wei, *Senior Member, IEEE*

***Abstract*—With the growing demand for latency-sensitive applications in 5G networks, edge computing has emerged as a promising solution. It enables instant response and dynamic resource allocation based on real-time network information by moving resources from the cloud to the network edge. Containers, known for their lightweight nature and ease of deployment, have been recognized as a valuable virtualization technology for service deployment. However, the prolonged startup time of containers can lead to long response time, particularly in edge computing scenarios characterized by long propagation time, frequent deployment, and migration. In this paper, we comprehensively consider image caching, container assignment, and registry selection problem in an edge system. To our best effort, there is no existing work that has taken all the above aspects into account. To address the problem, we propose a novel image caching strategy that employs partial caching, allowing local registries to cache either the least functional or complete version of application images. In addition, a container assignment and registry selection problem is solved by using an edge-based collaborative lazy pulling algorithm. To evaluate the performance of our proposed algorithms, we conduct experiments with real-world app usage data and popular images in a testbed environment. The experimental results demonstrate that our algorithms outperform traditional greedy algorithms in terms of average user response time and cache hit rate.**

***Index Terms*—Edge computing, 5G, Container, Caching, Resource allocation.**

I. Introduction

The emergence of latency-sensitive applications such as autonomous vehicles, virtual reality (VR), augmented reality (AR), and smart factories in the fifth-generation (5G) cellular networks has prompted the adoption of Multi-access Edge Computing (MEC) [1, 2] as a practical solution. MEC involves the relocation of processing and storage capabilities from the central node to the network edge. By deploying applications on edge nodes, MEC improves the quality of service (QoS) by processing tasks closer to the users [3]. Owing to the advantages of lightweight and easy deployment, containers have gained recognition as a valuable virtualization technology. For example, Google Search launches approximately 7,000 containers per second [4]. Recently, containers have been increasingly utilized in edge computing for frequent and rapid service deployment. Additionally, Kubernetes [5], an open-source platform for managing, scaling, and automating the deployment of containerized applications, is widely used in the industry.

Nevertheless, the deployment of containers in an edge computing environment faces multiple new challenges. First, it is impossible to maintain a pool of hot workers to accelerate container provisioning as in a cloud environment owing to the limited storage and computation resources [6]. Second, due to the extended propagation time, retrieving application images from a cloud-based registry to an edge node consumes a considerable amount of time and significantly slows down the service provisioning time, as evidenced by the fact that the download process constitutes 76% of the entire container startup time [7]. Consequently, the long container startup process violates the low latency demands of users. Furthermore, applications running on the edge have a substantial influence on user mobility, frequent deployment, and migration of containers within the dynamic edge computing system, thereby exacerbating the previously mentioned issue.

As a result of the constrained computational and storage capacity in edge computing, how to efficiently schedule services and allocate resources to ensure prompt service response has become one of the most challenging topics [8]. To cope with the slow service response time and resource allocation issue, several studies explore the service or package caching [9–14] approach and discuss various caching strategies. However, these studies overlook the possibility of applying a partial caching scheme, which is commonly utilized in the caching of video or data. Another approach investigates designing an efficient method to schedule tasks or assign containers to suitable edge nodes to enhance the overall service provisioning time or reduce the resource cost [15–20]. It is worth noting that the aforementioned research primarily focuses on the downloading of application packages or images from a cloud source, but does not jointly take advantage of the benefits offered by deploying multiple distributed local registries.

From another perspective, considering that a mere 6.4% of the image data transmitted during a pull operation is actually indispensable before a container can begin useful work according to the analysis of 57 images in [7]. Certain work utilizes the concept of exclusively downloading the requisite files before launching a container in order to avoid unnecessary waiting delay of retrieving data that will be used at a later stage, and thus expedite container startup time [7, 21–23]. However, the majority of these works concentrate on the cloud environment rather than in the context of edge computing, thereby neglecting the unique characteristics and is less applicable in resource-constrained and long propagation nodes.

Motivated by the aforementioned studies and the identified challenges, the objective of the work is to tackle the issue of image caching, container assignment, and registry selection that involves multiple registries and compute nodes across diverse edge areas. Recognizing that users within the same region often possess similar application preferences, we pro-

pose a system that deploys multiple local registries in each edge area to cache popular application images. Our caching scheme facilitates least functional-based partial caching, in which we can decide to cache either the least functional or complete function version for each application image. For those users who merely require basic application functionality, we can immediately execute tasks upon downloading the least functional portion of an image, denoted as the partial version. Subsequently, we provide services to the users who request advanced and complete functionality once the compute node downloads the entire image. Diverging from the traditional lazy pulling approach, we devise a novel concept that leverages local registries to collaboratively download images. To be more specific, our method can cache the least essential component of an image in local registries. We can retrieve the partial version from the local registry, while downloading the remaining files from the same local registry or the appstore.

The main contributions of this paper can be summarized as follows:

1) Combining lazy pulling techniques and partial caching capability on an edge computing architecture using local registries to accelerate the container startup time.

2) Formulating the optimization problem that aims to minimize the average user response time by considering image caching, container assignment, and registry selection, while taking into account an edge-based collaborative lazy pulling method.

3) Proposing least functional-based image caching decision algorithm and edge-based collaborative container assignment and registry selection algorithm to enhance space utilization efficiency and minimize average user response time.

4) Conducting comprehensive experiments using 20 widely-used applications from Docker Hub, along with a user application usage dataset on a real-world testbed to thoroughly evaluate the performance of our proposed algorithms. The results showcase a significant reduction in the average user response time, coupled with a substantial increase in terms of cache hit rate.

The rest of this work is structured as follows: Section II provides an overview of relevant literature, while Section III delves into our system model. Subsequently, our problem formulation and proposed algorithms are presented in Section IV. Finally, Section V assesses the performance of the proposed algorithms through real-world data and experiments and Section VI concludes this work.

II. Related Work

In this section, we discuss the existing literature on edge computing caching and the lazy pulling technique. We then present a comparative analysis between these existing works and our proposed work.

A. Caching in Edge Computing

Given the limited storage capacity of edge nodes, resource allocation emerges as a crucial concern in an edge computing system. Xiao et al. [24] focus on caching application data on edge servers by employing an online collaborative algorithm to minimize system cost from the app vendor's perspective, while the authors in [25] investigate the joint problem of video caching decision, computation, and radio resource allocation by using robust optimization to maximize the revenue of computation and radio resources. Moreover, [26] employs a reinforcement learning-based model to improve overall QoE and cache hit rate and an approach is proposed in [27] for shared node selection and cache replacement, aiming for lower user request latency and energy consumption in device-to-device assisted MEC networks. Additionally, video and data are fragmented into small segments when deciding the caching strategy to enhance space utilization, such as [28–30]. In these works, partial caching refers to the caching scheme in that only part of the video or data is cached.

For caching services or applications on edge nodes, The authors in [9] propose a proactive caching strategy that caches execution codes at the MEC server using a dynamic programming-based algorithm. The approach aims to reduce execution delay and energy consumption. Similarly, a task caching and offloading problem with the primary objective to minimize the energy cost is explored in [10] and the authors propose an alternating iterative algorithm as a solution. In [11, 12], deep reinforcement learning-based caching algorithms are employed to solve a package caching problem in serverless edge computing. Ma et al. [13] focus on addressing the optimization of edge service caching and workload scheduling to minimize service response time and outsourcing traffic based on Gibbs sampling and water filling techniques. Furthermore, the work [14] minimizes computation latency by determining service caching and task offloading strategies utilizing a Lyapunov-based online algorithm. However, none of the above studies consider a partial caching strategy when tackling application caching problem as data and video caching. Moreover, container assignment and registry selection problems are also not considered in the above studies.

B. Container Assignment and Registry Selection

In the field of edge computing, a series of work have been devoted to the scheduling problem of containers in order to achieve reduced response time. A three-step layer-aware scheduling algorithm is introduced in [15] that assigns multiple containers to edge nodes and decides the download sequence by considering the concept of layer sharing. In [16], a dependency scheduling method based on the dependencies between tasks is presented and implemented for improving the startup latency. Nevertheless, both of the studies overlook the scenario of multiple registries when it comes to downloading the required images onto the edge nodes. The work [17] addresses the optimization of service placement and network selection and introduces an iterative-based algorithm to minimize the access delay, communication delay, and switching delay. In [18], the authors propose an deployment algorithm that is based on Adam and weighted round-robin algorithm to minimize the cost of microservice deployment involving computing resources and storage resources. Sami et al. [19] present a DRL-based solution called IScaler to generate resource scaling and container placement decisions in MEC while taking the

TABLE I
A COMPARISON TABLE OF RELATED LITERATURE

#	Architecture	Objective	Container Assignment	Registry Selection	Caching Scheme	Lazy Pulling	Real Data
[9]	Edge	Energy consumption + execution delay	X	X	Whole	X	X
[10]	Edge	Energy consumption	X	X	Whole	X	X
[11]	2-tier edge	Cache hit rate + QoS violation number	X	X	Whole	X	X
[12]	2-tier edge	Service response time	X	X	Whole	X	X
[13]	Edge	Service response time	X	X	Whole	X	X
[14]	Edge	Computation latency	X	X	Whole	X	X
[15]	Edge	Container startup time	O	X	X	X	O
[16]	Edge	Container startup time	O	X	X	X	O
[17]	Edge	Quality of Service	O	X	X	X	O
[18]	Edge	Resource consumption cost	O	X	X	X	O
[19]	Edge	Application load + Costs	O	X	X	X	O
[20]	General	Container startup time	O	O	X	X	O
[7]	General	Container startup time	X	X	X	Traditional	O
[23]	2-tier edge	Container startup time	X	X	X	Traditional	O
LFPEC	3-tier edge	User response time (include container startup time)	O	O	LF Partial	EC	O

Hints: **LF Partial** = Least functional-based partial **EC** = Edge-based collaborative

demand of users and available resources into consideration. Additionally, the authors of [20] tackle the challenges of determining the appropriate deployment locations for microservices and selecting the registries to retrieve images. They suggest an accelerated distributed augmented lagrangian-based distributed algorithm exploiting the layer sharing concept. However, these studies fail to take into account of the characteristic that users within the same edge area tend to have similar application preferences. Accordingly, the existing studies mentioned above do not jointly consider the utilization of local registries to cache popular images, which could effectively reduce the image download time and further improve the user response time.

C. Container Startup Acceleration

All the aforementioned works are not highly aware of the fact that not all application files are required during the startup of a container, thus resulting in unnecessary waiting time and prolonged container startup duration. Another aspect of reducing the provisioning time involves launching a container before downloading all the files. While most of the existing research concentrates on cloud or general environments, they do not consider the specific characteristics of edge computing and cannot yield significant benefits. For instance, the authors in [7] conduct an evaluation on the deployment time of 57 different images and design a new Docker storage driver that speeds up container startup time by lazily fetching the required data based on the analysis. Similarly, a novel architecture is proposed in [23] that aims to accelerate container provisioning time on edge nodes by redesigning the deployment mechanisms. Moreover, [31] proposes a mechanism allowing less data transmission for image building by node-local duplicate data awareness, thus improving container update efficiency. A technique in [32] is introduced to effectively reduce container startup latency and memory wasting by its layer-wise sharing-aware container pre-warming and keep-alive mechanism.

However, the above studies primarily deal with reducing the container startup time, disregarding the potential collaboration between multiple registries that can further expedite provisioning time in edge computing scenarios.

D. Summary

A comparison of related work is listed in Table I. We take into consideration the features including system architecture, objective, registry selection, container assignment, lazy pulling, partial caching, and using real-world data or not. In summary, the analysis reveals that current research lacks a comprehensive approach that simultaneously incorporates partial caching and lazy pulling, which allows for caching the least functional version of images and involves cooperation between multiple registries in the decision-making process for caching, container assignment, and registry selection. Furthermore, by the combination of partial caching and lazy pulling, we aim to optimize not only container startup time but the total user response time and also make the most efficient use of our storage capacities.

III. SYSTEM MODEL

In this paper, we examine a 3-tier edge computing architecture including orchestrator-level, control-level, and computer-level defined in the IEEE 1935 Edge standard [33, 34] for resource and application management and orchestration [35, 36]. The system depicted in Fig. 1 is capable of caching application images in local registries for compute nodes to download necessary files and provide services to users. The five major components in the system will be described as follows:

1) Orchestrator: An orchestrator-level entity collecting a global view of the system including the cache list of local registries, resource information, and historical application request records, from each control node. After determining decisions, it sends control messages to control nodes.

2) Control Node: There are multiple edge areas within the whole system. In our scenario, users in the same edge area have similar application preferences and a majority of them tend to consistently request an identical application

service. For example, within the campus edge area, most of the users demonstrate a preference for accessing academic records through the application of the student information system (SIS), whereas payment services are prevalent in a shopping district. We denote the set of users within edge area m as $\mathbb{U}_m = \{1, 2, ..., U_m\}$. Every individual user initiates a request for an application service in a time slot, and p_u is the required number of CPU cycles for the given task. The control node is a control-level entity that is in charge of managing the cache inventory of a local registry, monitoring the resource utilization of each compute node, and documenting the historical request made by each user within the corresponding edge area.

3) Appstore: A computer-level entity regarded as a global registry with unlimited storage resources. It is responsible for operating a private registry service to store every application images that can be used throughout the entire system. Before uploading the images onto the appstore, each application image must undergo testing and authorization procedures to guarantee both security and functionality. The set of accessible applications in the whole system is represented as $\mathcal{A} = \{1, 2, ..., A\}$. Here we make a straightforward assumption that a request to instantiate an application corresponds to launching a container on a compute node. Additionally, an application image for application a is denoted as h_a.

4) Local Registry: Each edge area exists a local registry with limited cache space S_{b_m}. Besides caching a whole application image, a local registry is a computer-level entity that is able to partially cache a least functional application image. Therefore, a set of versions of an application image is denoted as $\mathcal{V} = \{partial, whole\}$ where respectively offer fundamental and complete functionality. In addition, we use $s_{comp}(h_{(a,v)})$ to represent the compressed size of image $h_{(a,v)}$ because local registries store compressed images. In order to determine which files are necessarily needed when a container launches, we record the order of file access during a testing process and select those files to form the content of the partial version of an image.

5) Compute Node: A computer-level entity that delivers application services to users. The set of compute nodes in edge area m is denoted as $\mathcal{W}_m = \{1, 2, ..., W_m\}$. Therefore, the set of all compute nodes can be represented as $\mathcal{W} = \{w \mid w \in \mathcal{W}_m, \forall m \in \mathcal{M}\}$. Each compute node w has limited resources with τ_w storage space and f_w CPU frequency. Additionally, there is a constraint on the maximum number of concurrent containers ρ_w that can be hosted on compute node [37]. Furthermore, before executing a container, a compute node must download and decompress the images. As a result, we utilize $s_{uncomp}(h_{(a,whole)})$ to signify the uncompressed size of whole image h_a.

In the following subsections, we will introduce storage model and delay model. For the sake of readability, the main notations involved in our system and their descriptions are summarized in Table II.

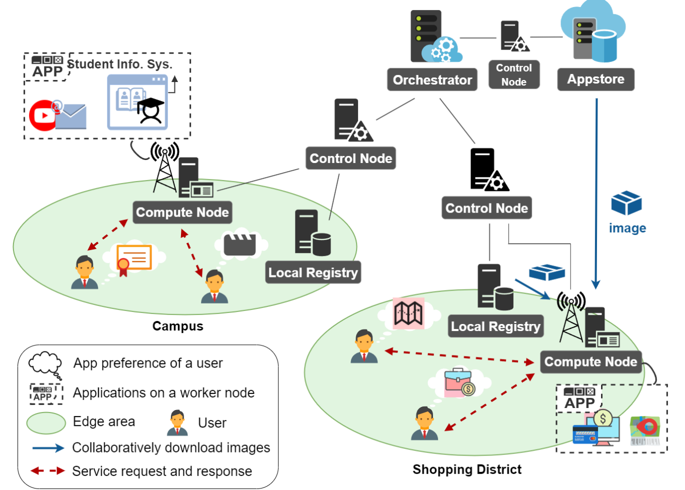


Fig. 1. The proposed 3-tier edge computing application image caching architecture.

TABLE II
LIST OF NOTATIONS

Notation	Definition
$\mathcal{M}$	The set of edge areas
$\mathcal{W}_m, \mathbb{U}_m$	The set of compute nodes and users in edge area m
τ_w, f_w ρ_w	Storage space and CPU frequency of compute node w Limited concurrent container number of compute node w
S_{b_m}	Cache space of local registry b_m
p_u	Required number of CPU cycles of the task of user u
$\mathcal{A}, \mathcal{V}$	The set of apps and versions in the system
$h_{(a,v)}$	The image of application a in version v
$\mathcal{L}_{h_a}$	The set of layers in an image h_a
s_{comp} s_{uncomp}	The compressed size. The uncompressed size.
$C_{b_m}^{h_{(a,v)}}$	Caching decision of registry b_m to $h_{(a,v)}$
E_w^a	Assignment decision of compute node w to app a
H_{b_m} H_w	The set of images cached in registry b_m The set of images used for apps in compute node w
$R_u^{a,v}$	Indicator denoting if user u requests for app a in version v
K_{w,b_m}^a	Decision of pulling app a from registry b_m to compute node w
t_{u,w^*}^{prop}	Propagation delay between user u and connected compute node w^*
$t_{w^*,w'}^{prop}$	Propagation delay between connected compute node w^* and processing compute node w'
$t_{u,w',a}^{pull}$	Time for compute node w' to pull files of application a for user u
$t_{w',b_{m'}}^{(a,pri.)}$, $t_{w',b_{m'}}^{(a,non_pri.)}$	Time for compute node w' to download the prioritized/non-prioritized portion of application a image from registry $b_{m'}$
t_a^{create}	Time to launch application a
$t_{w',u}^{proc}$	Processing time for compute node w' to complete task for user u
$D_{u,m}^a$	Response time of user u in edge area m toward app a
$\bar{D}_{u,m}^a$	Expected response time of user u in edge area m toward app a
$P_u^{a,v}$	Probability of user u requesting app a in version v

A. Storage Model

An image is composed of multiple compressed layers. We define the set of layers in an image h_a as $\mathcal{L}_{h_a} =$

$\{1, 2, ..., L_{h_a}\}$. The compressed size of layer l in version v is represented as $s_{comp}(l_v)$ and the uncompressed size of layer l is denoted as $s_{uncomp}(l_{whole})$. Here we introduce two binary variables to respectively represent image caching and container assignment decisions.

$$C_{b_m}^{h_{(a,v)}} = \begin{cases} 1, & \text{if local registry } b_m \text{ caches image} \\ & h_a \text{ in version } v \\ 0, & \text{otherwise} \end{cases} \quad (1)$$

$$E_w^a = \begin{cases} 1, & \text{if compute node } w \text{ instantiates app } a \\ 0, & \text{otherwise} \end{cases} \quad (2)$$

The storage model varies between local registries and compute nodes. In the case of a local registry, the set of cached images is symbolized as $H_{b_m} = \{h_{(a,v)} \mid C_{b_m}^{h_{(a,v)}} = 1, \forall a \in \mathcal{A}, \forall v \in \mathcal{V}\}$. For a compute node, the set of images utilized by all running application services is designated as $H_w = \{h_{(a,whole)} \mid E_w^a = 1, \forall a \in \mathcal{A}\}$. Assume that we have a collection of images denoted as $H_1 = \{h_{(a_1,v_1)}, h_{(a_2,v_2)}, ..., h_{(a_n,v_n)}\}$. Then the union of layers can be expressed as:

$$l_union(H_1) = \bigcup_{h \in H_1} \mathcal{L}_h \quad (3)$$

Due to the fact that many container images share common dependencies or libraries, layer sharing allows containers to share the same underlying layers. In other words, when multiple images share the same layers, only the layers that do not exist on the node need to be pulled. Accordingly, the total compressed size of a set of images H_1 involving layer sharing concept can be calculated as follows:

$$s_{comp}(H_1) = \sum\nolimits_{l_v \in l_union(H_1)} s_{comp}(l_v) \quad (4)$$

On the other hand, assuming we have another set of images $H_2 = \{h_{(a_1,whole)}, h_{(a_2,whole)}, ..., h_{(a_n,whole)}\}$ that has been decompressed and stored in a compute node while launching containers to provide services. We compute the total uncompressed size of a set of images H_2 involving layer sharing as follows:

$$s_{uncomp}(H_2) = \sum\nolimits_{l_{whole} \in l_union(H_2)} s_{uncomp}(l_{whole}) \quad (5)$$

B. Delay Model

We start by assuming that a user initiates the request for a single application within a time slot and introduce an auxiliary binary variable $R_u^{a,v}$ to represent the status of application requests received from users:

$$R_u^{a,v} = \begin{cases} 1, & \text{if user } u \text{ request for app } a \text{ and} \\ & \text{the task is requested for version } v \\ 0, & \text{otherwise} \end{cases} \quad (6)$$

Our focus lies in measuring the response time including the duration from when a user initiates a request for an application service until the task is fully executed and the response is returned to the user. It is assumed that users tend to connect to a compute node with minimal propagation time. To indicate which registry to download the required application packages for a compute node, we introduce an additional decision variable represented as follows:

$$K_{w,b_m}^a = \begin{cases} 1, & \text{if compute node } w \text{ downloads} \\ & \text{prioritized files of app } a \text{ from} \\ & \text{registry } b_m \\ 0, & \text{otherwise} \end{cases} \quad (7)$$

The delay model for the response time of a user comprises four distinct types of delays. In the subsequent subsections, we delve into an elaboration on the details of each component.

1) Propagation Time: t_{u,w^*}^{prop} is the propagation delay between user u and connected compute node w^*. Since the compute node a user connects to may not provide the target service that a user requests for, $t_{w^*,w'}^{prop}$ denotes the time it takes for forwarding the request from compute node w^* to the processing compute node w'. Additionally, it should be noted that $t_{w^*,w'}^{prop} = 0$ if w^* and w' refer to the same compute node, which means the request can be processed in the originally connected compute node, and do not have to be forwarded to another compute node.

2) Image Pulling Time: The time required for a compute node w' to pull the necessary files for application a in order to fulfill the request of user u is denoted as $t_{u,w',a}^{pull}$. As for the pulling time, we also consider the operation cost such as compression, extraction, and verification. In addition, we define $t_{w',b_{m'}}^{(a,pri.)}$ as the time taken by compute node w' to download the prioritized portion, i.e. all the files in partial version of an image, of application a image from registry $b_{m'}$, and $t_{w',b_{m'}}^{(a,non_pri.)}$ as the time needed to download the remaining non-prioritized files. The prioritized segment of an image can be downloaded either from the appstore or local registries. In the case where the prioritized part is obtained from the appstore, the non-prioritized part is also sourced from the appstore. However, if the prioritized portion is downloaded from a local registry, the non-prioritized section can be retrieved from the same local registry only if the registry has cached the whole version of the image. Otherwise, the non-prioritized files should be downloaded from the appstore. Note that if a user does not require the complete functionality of an application, the response time does not include the download time of non-prioritized files. As stated above, the image pulling time can be derived as follows:

$$t_{u,w',a}^{pull}(C, K, R) = K_{w',appstore}^a \cdot [\ t_{w',appstore}^{(a,pri.)} + R_u^{a,whole} \cdot t_{w',appstore}^{(a,non_pri.)}\ ] + \sum_{m' \in \mathcal{M}} K_{w',b_{m'}}^a \cdot [\ t_{w',b_{m'}}^{(a,pri.)} + C_{b_{m'}}^{h_{(a,partial)}} \cdot R_u^{a,whole} \cdot t_{w',appstore}^{(a,non_pri.)} + C_{b_{m'}}^{h_{(a,whole)}} \cdot R_u^{a,whole} \cdot t_{w',b_{m'}}^{(a,non_pri.)}\ ] \quad (8)$$

3) Container Creation Time: We introduce t_a^{create} as the time to launch application a after downloading the required files from registries, e.g. the time of executing $docker\ run$. Since creation time remains consistent, we set the time to constant for each application a [15].

4) Processing Time: To determine the processing time $t_{w',u}^{proc}$ for a compute node to complete the task assigned by user u. Refer to [38], the processing time can be given as:

$$t^{proc}_{w',u} = \frac{p_u}{f_{w'}} \tag{9}$$

where p_u represents the CPU cycles required for the given task, and $f_{w'}$ is the computation resource, expressed as the CPU frequency, of the processing compute node w'.

Finally, we formulate the response time function of user u in edge area m who requests the service of application a, taking into account all the aforementioned components:

$$D^a_{u,m}(C,E,K,R) = 2 \cdot t^{prop}_{u,w^*} + t^{pull}_{u,w',a}(C,K,R) + t^{create}_a + 2 \cdot t^{prop}_{w^*,w'} + t^{proc}_{w',u} \tag{10}$$

where

$$w^* = \arg\min_{w \in \mathcal{W}_m}(2 \cdot t^{prop}_{u,w})$$

and

$$w' \in \{E^a_w = 1 \mid \forall w \in \mathcal{W}\} \tag{11}$$

where w^* is the compute node a user originally connects to, and w' is the compute node that provides the requested service to a user.

IV. Problem Formulation

In this section, we will mathematically formulate the objective of our work, which aims to optimize the average response time of users in multiple edge areas by performing image caching, container assignment, and registry selection. Due to the heterogeneous time scale and complexity involved in image caching operation, as well as in real-time container assignment and registry selection, we further decompose the original problem into two subproblems to achieve our ultimate goal.

A. Main Problem Formulation

Our objective is to minimize the average response time of requests from all edge areas by determining the least functional-based image cache decision of local registries (i.e. C), container assignment decision of compute nodes (i.e. E), and which registry should the compute node download required application packages from (i.e. K).

We can formulate our optimization problem as defined in (12), where constraint C1 ensures that duplicate cache decisions are avoided in each local registry, allowing at most one version of an application image to be cached in each local registry. Constraint C2 limits the total size of all stored application packages not exceeding available cache space for each local registry. Constraint C3 describes the storage limitation of a compute node, stating that the total image size of all running applications should not exceed the available storage space. Then, constraint C4 restricts the maximum number of concurrent containers that can run on a compute node at a time. Furthermore, constraint C5 ensures that an application is deployed on a single compute node if there is any user requests for it. Constraint C6 indicates that the prioritized files of an application image should be downloaded from a single registry if a compute node w is assigned to instantiate an application a. Lastly, constraint C7 states that if a compute node w is assigned to download packages of application a from registry b_m, then the application packages should be cached in the registry.

$$\min_{C,E,K} \frac{1}{|\mathbb{U}|} \sum_{m \in \mathcal{M}} \sum_{u \in \mathbb{U}_m} \sum_{a \in \mathcal{A}} \sum_{v \in \mathcal{V}} R^{a,v}_u \cdot D^a_{u,m}(C,E,K,R) \tag{12}$$

s.t.

$$C1: \sum_{v \in \mathcal{V}} C^{h_{(a,v)}}_{b_m} \le 1, \forall a \in \mathcal{A}, \forall m \in \mathcal{M}$$

$$C2: s_{comp}(H_{b_m}) \le S_{b_m}, \forall m \in \mathcal{M}$$

$$C3: s_{uncomp}(H_w) \le \tau_w, \forall w \in \mathcal{W}$$

$$C4: \sum_{a \in \mathcal{A}} E^a_w \le \rho_w, \forall w \in \mathcal{W}$$

$$C5: \sum_{w \in \mathcal{W}} E^a_w = \min(\sum_{m \in \mathcal{M}} \sum_{u \in \mathbb{U}_m} \sum_{v \in \mathcal{V}} R^{a,v}_u, 1), \forall a \in \mathcal{A}$$

$$C6: K^a_{w,appstore} + \sum_{m \in \mathcal{M}} K^a_{w,b_m} = E^a_w, \forall w \in \mathcal{W}, \forall a \in \mathcal{A}$$

$$C7: K^a_{w,b_m} - \sum_{v \in \mathcal{V}} C^{h_{(a,v)}}_{b_m} \le 0, \forall m \in \mathcal{M}, \forall a \in \mathcal{A}$$

Given the complexity of the original problem and the requirement to prioritize the caching decision before determining the compute node for an application to launch on and selecting the appropriate package download location in real-time, we decompose problem (12) into two subproblems. The first subproblem optimizes the caching decision C for all local registries, taking into account their available cache space, application image information, and historical user information. The second subproblem optimizes the container assignment E and registry selection decision K based on the user requests, network conditions, and the results of the caching.

B. Least Functional-based Caching Decision

We begin by providing an extensive elaboration of the subproblem **P1**. Given the uncertainty of future user requests for different applications, we replace the indicator of the actual user request (i.e. $R^{a,v}_u$) with the historical probability of a request (i.e. $P^{a,v}_u$). We assume that a user's application preference remains relatively stable over time, and is highly influenced by historical usage patterns. Suppose the current time slot is denoted as T, we calculate the historical request probability of user u for application a in version a in time slot T according to historical information from previous t time slots as follows:

$$P^{a,v}_u = \frac{1}{t} \sum_{t'=T-t}^{T-1} R^{a,v}_u(t'), \ \forall a \in \mathcal{A}, \forall u \in \mathbb{U}, \forall v \in \mathcal{V} \tag{13}$$

Since adjusting caching decision only affects image pulling time, we substitute the actual response time of users (i.e. D) with the expected pulling time value (i.e. $\bar{t}^{pull}$). However, as the real download time of the application image is unknown in advance, we attempt to adapt the original image pulling time. To express the layer number that is necessary for downloading

the image of application a on compute node w, we take into account the existing application images on the node. After that, we proceed to normalize the layer number by dividing it by the maximum layer numbers found in all the application images using the following equation:

$$layer(w, a) = |\mathcal{L}_{h_a} \setminus \bigcup_{i \in H_w} \mathcal{L}_i| \tag{14}$$

Likewise, the size that a compute node needs to download for application a is shown in equation (15). Initially, a compute node downloads the prioritized files, namely the partial version, and offers the basic functionality to a portion of users, while providing complete functionality to the remaining users until the non-prioritized files are stored. Therefore, in the case where the input $type$ is equal to $pri.$, we compute the size required to download the partial version of the application image h_a on compute node w. On the other hand, if the input $type$ is $non_pri.$, it calculates the size of the remaining non-prioritized portion of the image.

$$\text{size}(w, a, type) = \begin{cases} s_{comp}(H_w \cup h_{(a,partial)})\text{-}s_{comp}(H_w), \\ \quad \text{if } type = pri. \\ s_{comp}(H_w \cup h_{(a,whole)})\text{-}s_{comp}(H_w \cup h_{(a,partial)}), \\ \quad \text{if } type = non_pri. \end{cases} \tag{15}$$

Because the actual download time of an application is unknown, we try to generate a new value that can serve as a substitute for the delay. Intuitively, there exists a strong correlation between image size and pulling time. Based on our experiment observations, we found that the number of layers also influences download time. An image with a higher layer number has a longer download time compared to one with a lower layer number even when the size is the same. This is because the Docker daemon in a compute node has an upper limit on the number of simultaneous downloads, and by default, it can concurrently fetch three layers of an image concurrently. Furthermore, during a standard Docker pull operation, the extraction and decompression process is executed in a sequential manner, starting from the first layer [39]. Accordingly, setting a higher concurrent layer download limit will prolong the completion time of the first layer due to the constrained bandwidth. It defers the decompression and extraction process and results in a longer pulling time. As a result, the delay is transformed into a new estimated value by means of the subsequent equation, which considers the measured RTT between compute node w and registry b_m, the download size of image files (15) and the cumulative operation delay costed by layers (14) as follows:

$$\Phi(w, b_m, a, type) = RTT_{w,b_m} \cdot size(w, a, type) + \chi(w, a) \tag{16}$$

Since after downloading a layer, the compute node needs to decompress and verify the checksum to ensure the integrity. We calculate the operation latency for processing decompression, extraction, and verification of checksum by referring to [40] as the following equation:

$$\chi(w, a) = \theta \cdot \frac{layer(w, a)(layer(w, a) + 1)}{2} + \lambda \cdot layer(w, a) \tag{17}$$

where $\theta = 0.001526$ and $\lambda = 0.03087$, which is same as [40].

In the default container assignment, the edge area m_a^* with the highest request probability is selected to run a specific application. Since edge area m_a^* may have multiple compute nodes, the expected image pulling time $\bar{t}^{pull}$ is determined as the average pulling time for all possible compute nodes in the selected edge area. Let $\bar{t}^{pull}_{u,a,appstore}(P)$ denote the expected image pulling time taken for all possible compute nodes to download the image of application a from the appstore. We change the original download time (e.g. $t^{(a,pri.)}_{w',b_{m'}}$) to the estimated value obtained through (16) and replace the actual user request with the historical probability of the request by using (13). The calculation can be performed as follows:

$$\bar{t}^{pull}_{u,a,appstore}(P) = \frac{1}{|W_{m_a^*}|} \sum_{w \in \mathcal{W}_{m_a^*}} [\ \Phi(w, appstore, a, pri.) + P_u^{a,whole} \cdot \Phi(w, appstore, a, non_pri.)] \tag{18}$$

We use $\bar{t}^{pull}_{u,a,local}(P)$ to represent the expected image pulling time required to download the image from local registries. Similarly, using the same principle as (18), we replace the original download time and the actual user request as well. Therefore, the equation can be expressed as follows:

$$\bar{t}^{pull}_{u,a,local}(C, P) = \frac{1}{|W_{m_a^*}|} \sum_{m' \in \mathcal{M}} \sum_{w \in \mathcal{W}_{m_a^*}} [C^{h_{(a,partial)}}_{b_{m'}} \cdot \Phi(w, b_{m'}, a, pri.) + C^{h_{(a,partial)}}_{b_{m'}} \cdot P_u^{a,whole} \cdot \Phi(w, appstore, a, non_pri.) + C^{h_{(a,whole)}}_{b_{m'}} \cdot P_u^{a,whole} \cdot \Phi(w, b_{m'}, a, non_pri.)] \tag{19}$$

If the expected image pulling time from local registries is non-zero, we set the expected image pulling time $\bar{t}^{pull}$ as the minimum value between the expected image pulling time from the appstore and the expected image pulling time from the local registries. The equation is presented as follows:

$$\bar{t}^{pull}_{u,a}(C, P) = \begin{cases} \bar{t}^{pull}_{u,a,appstore}(P), & \text{if } \bar{t}^{pull}_{u,a,local}(C, P) = 0 \\ \min(\bar{t}^{pull}_{u,a,appstore}(P), \bar{t}^{pull}_{u,a,local}(C, P)), & \text{otherwise} \end{cases} \tag{20}$$

where

$$m_a^* = \arg\max_{m' \in \mathcal{M}} \sum_{u \in \mathbb{U}_{m'}} \sum_{v \in \mathcal{V}} P_u^{a,v} \tag{21}$$

After adapting from equation (12), we formally define the subproblem **P1**, which focuses on minimizing the expected

Algorithm 1: Least Functional-based Image Caching Decision

Input: Historical request probability $P_u^{a,v}$, Registry storage space S_{b_m}
Output: Image caching decision C

```
1  Initialize C = {C_{b_m}^{h(a,v)} = 0 | ∀m ∈ M, ∀a ∈ A, ∀v ∈ V}
2  Initialize scheduled ← an empty list
3  Calculate popularity_score_m^{a,v} by (23), ∀a ∈ A, ∀v ∈ V
4  Sort popularity_score_m^{a,v} by value in descending order
5  foreach element in popularity_score_list do
6      if a in scheduled then
7          continue
8      end
9      if s_comp(H_{b_m} ∪ h_(a,v)) ≤ S_{b_m} then
10         C_{b_m}^{h(a,v)} ← 1
11         Append a to scheduled
12         continue
13     end
14     if v = whole then
15         if s_comp(H_{b_m} ∪ h_(a,partial)) ≤ S_{b_m} then
16             C_{b_m}^{h(a,partial)} ← 1
17             Append a to scheduled
18         end
19     end
20 end
```

image pulling time (i.e. $\bar{t}^{pull}$) for each potential application request by determining the collaborative cache decision of all local registries (i.e. C):

$$\mathbf{P1}: \min_{C} \frac{1}{|\mathbb{U}|} \sum_{m\in\mathcal{M}} \sum_{u\in\mathbb{U}_m} \sum_{a\in\mathcal{A}} \sum_{v\in\mathcal{V}} P_u^{a,v} \cdot \bar{t}_{u,m,a}^{pull}(C,P)$$
$$\text{s.t. C1, C2} \tag{22}$$

The subproblem is formulated as a 0-1 integer linear programming problem, which is considered NP-hard and will be proved in the later subsection.

To address this problem, we first define popularity score by the following equation (23) for version v of application a in edge area m based on the previous usage records and the information of application images.

$$popularity_score_m^{a,v} = \beta \cdot \frac{\sum_{u\in\mathbb{U}_m} P_u^{a,v}}{\max_{a'\in\mathcal{A},v'\in\mathcal{V},m'\in\mathcal{M}} \sum_{u\in\mathbb{U}_{m'}} P_u^{a',v'}} + \gamma \cdot \frac{|\mathcal{L}_{h_a}|}{\max_{a'\in\mathcal{A}} |\mathcal{L}_{h_{a'}}|} + (1-\beta-\gamma) \cdot \frac{s_{comp}(h_{(a,v)})}{\max_{a'\in\mathcal{A},v'\in\mathcal{V}} s_{comp}(h_{(a',v')})} \tag{23}$$

where β and γ are the weight coefficients between 0 and 1, which represents the importance of different factors. The computation can be performed by each control node to obtain the statistics in its edge area and they report the results to the orchestrator. Specifically, when downloading images with more layer numbers and larger sizes from different registries, there will be notable variations in the time required to pull the images. Conversely, when downloading images with fewer layer numbers and smaller sizes, the time difference will not be substantial. Consequently, in case of an equal number of requests, the act of placing an image with a greater magnitude in terms of size and layer number in a remote registry can significantly increase pull time and has a notable impact on the overall system, as opposed to an image with a smaller size and lower layer number. Therefore, the popularity score takes historical request probability, layer number, and the size of an image into consideration.

In the proposed Algorithm 1, we first calculate a popularity score for each application, version, and edge area (line 3). Afterward, the values are sorted in descending order (line 4), allowing higher-value applications and versions to be cached in the desired local registry, provided there is sufficient space available (lines 9-13). In addition, if the size of an application image exceeds the available space of a local registry and the image is in the whole version, we attempt to cache a smaller version of the image (lines 14-19), i.e. in a partial version.

C. Edge-based Collaborative Container Assignment and Registry Selection Decisions

Next, our attention shifts towards defining the subproblem **P2**, which entails the decisions of container assignment and registry selection. In order to focus on the specific issue at hand, we omit the consideration of the propagation time between the user and the connected worker. This is justified by the fact that the values of t_{u,w^*}^{prop} are determined solely from user location and there is no relevance to the decision-making process of container assignment and registry selection. As a result, we adapt the original delay model (10) to the following form:

$$\bar{D}_{u,m}^{a}(E,K,R) = \bar{t}_{u,w',a}^{pull}(C,K,R) + t_a^{create} + 2\cdot\bar{t}_{w^*,w'}^{prop} + t_{w',u}^{proc} \tag{24}$$

where w^* and w' maintain the same definitions as previously denoted in (11), t_a^{create} is the application creation time, $\bar{t}_{w^*,w'}^{prop}$ represents the measured propagation time in the current time slot, and the processing time $t_{w',u}^{proc}$ is calculated by equation (9). Drawing upon the concept developed in (20), the expected pulling time during the container assignment and registry selection phase is adapted from (8) by changing the actual pulling time to an estimated value and can be written as:

$$\begin{aligned}\bar{t}_{u,w',a}^{pull}(C,K,R) = {} & K_{w',appstore}^{a} \cdot [\Phi(w',appstore,a,\\ & pri.) + R_u^{a,whole} \cdot \Phi(w',appstore,a,non_pri.)] + \\ & \sum_{m'\in\mathcal{M}} K_{w',b_{m'}}^{a} \cdot [\Phi(w',b_{m'},a,pri.) + C_{b_{m'}}^{h_{(a,partial)}} \cdot \\ & R_u^{a,whole} \cdot \Phi(w',appstore,a,non_pri.) + C_{b_{m'}}^{h_{(a,whole)}} \cdot \\ & R_u^{a,whole} \cdot \Phi(w',b_{m'},a,non_pri.)]\end{aligned} \tag{25}$$

Ultimately, we define the objective of **P2** as minimizing the average expected user response time, accomplished by determining the container assignment E and registry selection

Algorithm 2: Edge-based Collaborative Container Assignment and Registry Selection

Input: limited number of concurrent containers ρ_w, storage space of compute nodes τ_w, computation resource of compute nodes f_w, network measurement RTT, user request $R_u^{a,v}$, caching status $C_b^{h_{(a,v)}}$
Output: container assignment decision E, registry selection decision K

```
1  Initialize H*_w ← ∅, ∀w ∈ W
2  Initialize E = {E^a_w = 0 | ∀w ∈ W, ∀a ∈ A}
3  Initialize K = {K^a_{w,b_m} = 0 | ∀a ∈ A, ∀w ∈ W, ∀m ∈ M} ∪ {K^a_{w,appstore} = 0 | ∀a ∈ A, ∀w ∈ W}
4  UR = {a | R^{a,v}_u = 1, ∀a ∈ A, ∀v ∈ V, ∀u ∈ U}
5  foreach a ∈ UR do
6      B_a = {b_m | C^{h_(a,v)}_{b_m} = 1, ∀m ∈ M, ∀v ∈ V} ∪ {appstore}
7      Calculate impact_a by (27)
8  end
9  Sort UR by impact_a in descending order
10 foreach a ∈ UR do
11     w' ← null, b' ← null, cost_a ← +∞
12     foreach w ∈ W do
13         if |H*_w| + 1 ≤ ρ_w and s_uncomp(H*_w ∪ {h_(a,whole)}) ≤ τ_w then
14             tmpE ← E, tmpE^a_w ← 1
15             cost, b ← CalCost (a, B_a, w, C, tmpE, K, R)
16             if cost < cost_a then
17                 cost_a ← cost, w' ← w, b' ← b
18             end
19         end
20     end
21     E^a_{w'} ← 1, K^a_{w',b'} ← 1
22     H*_{w'} ← H*_{w'} ∪ {h_(a,whole)}
23 end
24 Function CalCost (a, B_a, w, C, tmpE, K, R):
25     foreach b ∈ B_a do
26         if b ≠ appstore and Σ_{v∈V} C^{h_(a,v)}_b ≠ 1 then
27             continue
28         end
29         tmpK ← K, tmpK^a_{w,b} ← 1
30         cost_b = Σ_{m''∈M} Σ_{u∈U_{m''}} Σ_{v''∈V} R^{a,v''}_u · D̄^a_{u,m''}(C, tmpE, tmpK, R)
31     end
32     return minimal cost_b and b
```

K for all compute nodes:

$$\mathbf{P2}: \min_{E,K} \frac{1}{|\mathbb{U}|} \sum_{m\in\mathcal{M}} \sum_{u\in\mathbb{U}_m} \sum_{a\in\mathcal{A}} \sum_{v\in\mathcal{V}} R_u^{a,v} \cdot \bar{D}^a_{u,m}(C,E,K,R)$$
$$\text{s.t. C3, C4, C5, C6, C7} \tag{26}$$

which is classified as a 0-1 linear programming problem that is NP-hard and will be proved in the later subsection.

To solve the problem, we first define the impact value for application a by the following equation (27), which takes into account the request probability within the current time slot, the normalized layer number, and the normalized expected size. The normalized expected size is obtained based on the normalized image size and the request percentage of the corresponding version.

$$impact_a = \delta \cdot \frac{\sum_{v'\in\mathcal{V}} \sum_{u\in\mathbb{U}} R_u^{a,v'}}{|\mathbb{U}|} + \kappa \cdot \frac{|\mathcal{L}_{h_a}|}{\max_{a'\in\mathcal{A}} |\mathcal{L}_{h_{a'}}|} + (1-\delta-\kappa)\cdot \sum_{v\in\mathcal{V}} \left(\frac{s_{comp}(h_{(a,v)})}{\max_{a'\in\mathcal{A}, v'\in\mathcal{V}} s_{comp}(h_{(a',v')})} \cdot \frac{\sum_{u\in\mathbb{U}} R_u^{a,v}}{\sum_{v'\in\mathcal{V}} \sum_{u\in\mathbb{U}} R_u^{a,v'}} \right) \tag{27}$$

where δ and κ are the weight factors ranging from 0 to 1, which serves to quantify the significance of different factors.

Subsequently, we design novel Algorithm 2, in which UR is denoted as the set formed by applications requested by at least one user (line 4), and the impact value for each application in it is calculated (lines 5-8). We sort UR by impact values in descending order (line 9) so that applications having a greater impact on the overall system performance can be prioritized when scheduling. For each application a in UR, we iterate through all compute nodes that are capable of running application a to find out the compute node w' and registry b' that require the minimal cost $cost_a$ (lines 10-22) by using function $CalCost$. In function $CalCost$ (lines 24-32), we assume that the container is assigned to compute node w and we iterate through all the registers that contain the image of application a to find out the register b that causes the minimal cost when pulling the corresponding image. In the proposed algorithm, we run through three for loops, which leads to the complexity of $O(|UR||W||B_a|)$.

By running Algorithm 2, the system identifies the optimal pair of compute node and registry that incurs the minimal cost for assigning applications to a compute node and retrieving the corresponding images from the registry. Notably, we improve the download process of application images through collaboration between local registries and the appstore, which enables the separate download of the least functional part and the remaining part of an image from different registries. Additionally, the cost is computed by evaluating the cumulative user response time made by the current decision.

D. Proof of NP-hardness

We can prove the NP-hardness of subproblem P1 via the multiple knapsack problem (MKP): A set of knapsacks each with a specific capacity is given and a set of items each with a specific weight and value are given, in which we need to find a disjoint set of items for each knapsack under capacity limitation to optimize total values. We can think of local registries as knapsacks with limited storage spaces and application images as items with specific storage requirements and costs. Since MKP is considered an NP-hard problem, the NP-hardness of subproblem P1 is proved.

For subproblem P2, we can prove NP-hardness via multidimensional knapsack problem (MDKP): A set of knapsacks each with d-dimension capacity limits, and a set of items each with a d-dimension capacity and a specific value is given, in which we need to find a disjoint set of items for

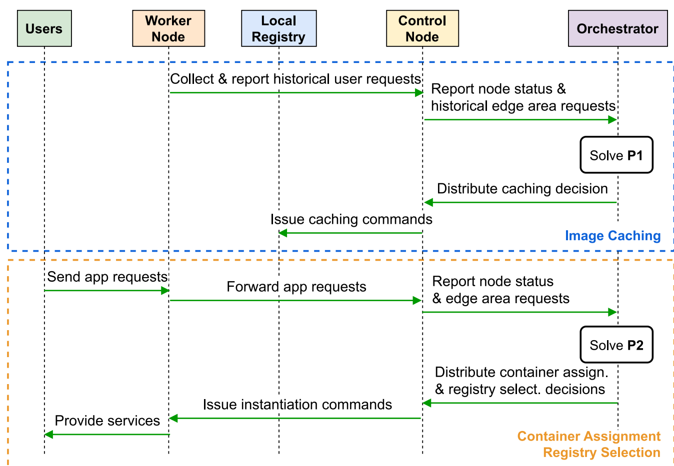


Fig. 2. The system workflow of proposed LFPEC.

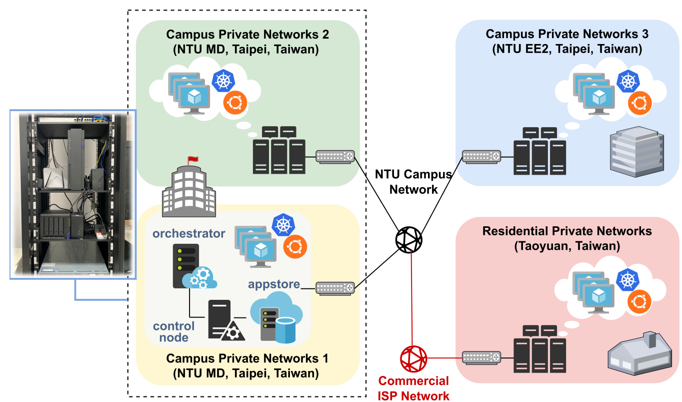


Fig. 3. An overview of our real-world testbed setup.

each knapsack under capacity constraints to optimize total values. We can think of compute nodes as knapsacks with multiple-dimension capacity limits and containers as items with multiple-dimension capacity and costs. Due to the NP-hardness of MDKP, subproblem P2 is also proved NP-hard.

E. System Workflow

The overall workflow of our proposed system LFPEC is visualized in Fig. 2. During the image caching phase, the orchestrator periodically updates the image caching decision by solving **P1**. The decision is based on the storage space of the local registries and historical user requests collected by the control nodes in each edge area. Subsequently, the orchestrator proactively distributes the caching decision back to the control nodes. Upon receiving the commands, each control node forwards them to the local registry, instructing it to perform caching operations. In the container assignment and registry selection phase, the orchestrator makes the decisions by solving **P2** based on the user requests and node information such as computation resources and network measurements obtained from the control nodes. The compute nodes then instantiate the corresponding applications from the target registry, following the control messages sent by the control node.

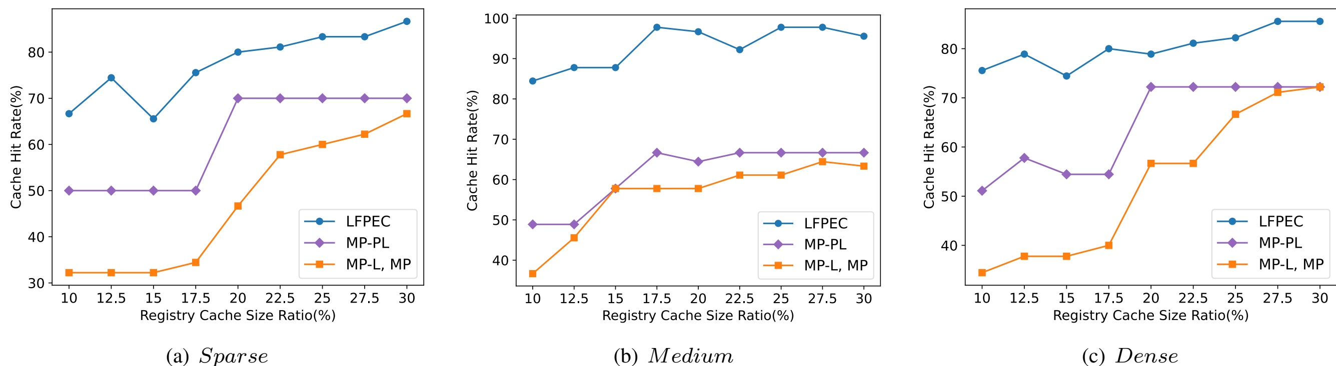
V. Experimental Results

In this section, we provide a comprehensive overview of our experimental setup, including the descriptions of the experimental environment, testbed implementation, utilization of real data, and system parameters. Afterward, we evaluate the performance of our proposed scheme with real-world datasets on the testbed.

A. Experiment Setup

1) Testbed Setting: Our testbed comprises multiple machines, containing two Intel i7-8700K 6-core CPU servers, one Intel i9-9900K 8-core GPU server, one Intel i7-11700K 8-core CPU server, and one i7-8559U 4-core CPU server. We create virtual machines equipped with 4096MB of RAM and 2 virtual CPU cores to represent nodes. All nodes in our testbed are operating on Ubuntu 18.04.6 LTS and are installed with Kubernetes v1.24.3 [5], which handles the management and deployment of containerized applications. Moreover, we have implemented the edge-based collaborative lazy pulling mechanism, which supports efficient and flexible image retrieval harnessing the collaboration between local registries and the appstore, by modifying a widely-used traditional lazy pulling open-source project called stargz snapshotter [41] in version v0.13.0. For our experiments, we consider a system architecture consisting of an orchestrator, an appstore, four control nodes, three local registries, and three compute nodes. The local registries and the appstore employ the Docker Registry 2.0 implementation [42] to store and distribute application images. The three edge areas are geographically distributed across Ming DA Hall and EE Building No.2 at National Taiwan University, and Taoyuan City in Taiwan, each with a control node, a local registry, and a compute node. In addition, the orchestrator, a control node, and the appstore are located in a separate edge area under Ming Da Hall, distinguished by a distinct public IP address. We measure the download throughput of each edge area using the NTU speed test website. The download throughput is 96.17Mb/s in the EE edge area, 113.16 Mb/s in the Taoyuan edge area, 92.3 Mb/s in the Ming Da 1 edge area, and 101.17 Mb/s in the Ming Da 2 edge area. Fig. 3 provides an overview of our testbed setup. Moving forward, we conduct performance evaluations using the aforementioned testbed configurations and real-world datasets.

2) Dataset Description: With an eye to further validating the performance of the proposed approach, we utilize the real-world Tsinghua App Usage Dataset [43] to represent the application usage behaviors of users in our experiment. The original dataset includes usage records of total 1,000 users and 2,000 applications which comprises user IDs, timestamps, and app IDs. We extract the data from a specific time range, preciously from 4/21 13:00:00 to 15:59:59, and then divide this time range into 30-minute time slots, resulting in a total of six time slots. From this data, we select the most 20 popular applications and filter out the records of other applications. We focus on active users who appeared in every time slot.



(a) *Sparse* (b) *Medium* (c) *Dense*

Fig. 4. Cache hit rate versus registry cache ratio in different application usage density.

Since a user may use multiple applications within a time slot, we simplify the scenario by setting the application with the highest number of requests from a user as his/her target application in that specific time slot. After all the operations above, we can obtain app usage records of 90 users, involving 20 applications, across the six time slots. We use the data of the last time slot as the current requests, while the data of other five time slots served as historical user requests.

To distribute the users among the edge areas, we assign 30 users to each edge area. We empirically set application usage density to 0.45, 0.5, and 0.55 for sparse, medium, and dense usage density scenarios. In the context of dense density, more users with same application preferences are allocated to the same edge area. While in the scenario of sparse density, users with same application preferences are distributed more thinly across the three edge areas. Moreover, we set the least functional application request ratio to 0.55, indicating that there are 55% of the tasks can be processed using a partial application image for each application.

For the convenience of the experiment, we use the most popular 20 images, which have been downloaded over a billion times or at least 500 million times in Docker Hub [44] instead of the actual applications from the Tsinghua App Usage Dataset. The layers of each image were defined when they were built in Docker. Before pushing all the images to the appstore, we launch the containers for each application image and record the file access order using stargz snapshotter [41] until they are ready to provide services. This process allows us to divide each image into prioritized files and non-prioritized files. Since different applications have various performances, we carefully determine the ready status for each application. Further details can be found in Table III. Additionally, we measure the container creation time by conducting 20 measurements and take the average value as the time for each application.

3) System Parameters: Refer to and modified from [45] for our scenario, the cache capacity of each local registry is configured to range from 10% to 30% of the total compressed size of application images. On the other hand, the storage space allocated to each compute node is set to 3GB. To determine the limited concurrent number of containers that can be executed on a compute node, we set the value to 7 by our real measurement. In the measurement, we attempt to deploy as many MySQL applications as possible on a compute node until the system crashes or the memory utilization is higher than 80%. The required number of CPU cycles for a task follows a uniform distribution within [1000, 1500]M, and the CPU frequency of each compute node is fixed at 1G CPU cycles/s [38]. The propagation time between users and compute nodes are uniformly distributed within [1,2]ms [46]. Furthermore, the parameters for equations (16), (23), and (27) are empirically set as $(\beta, \gamma, \delta, \kappa) = (0.6, 0.15, 0.6, 0.15)$ based on the importance of each terms.

TABLE III
LIST OF 20 SELECTED APP IMAGES

Image Name: tag	Ready Status
ubuntu: *focal-20220922*	/bin/bash -c "ls"
centos: *7.9.2009*	/bin/bash -c "ls"
debian: *bookworm*	/bin/bash -c "ls"
alpine: *3.16.2*	/bin/sh -c "ls"
redis: *7.0.5*	log: Ready to accept connections
postgres: *14.5*	log: database system is ready to accept connections
mysql: *8.0.31*	log: ready for connections
mongo: *6.0.2*	log: Waiting for connections
python: *3.10.7*	print("hello")
golang: *1.19.2*	fmt.Println("hello")
node: *19.0.0*	console.log("hello")
nginx: *1.22.0*	log: Configuration complete; ready for start up
httpd: *2.4.54*	log: httpd -D FOREGROUND
tomcat:*10.1-jdk17-temurin*	log: Server startup
openjdk: *20-jdk*	System.out.println("Hello")
eclipse-mosquitto: *2.0.15*	log: mosquitto version 2.0.15 running
registry: *2.8.1*	log: listening on [::]:5000
rabbitmq: *3.11.0*	log: Server startup complete
busybox: *1.35.0*	/bin/sh -c "ls"
influxdb: *1.8.10*	log: Listening for signals

B. Compared Algorithms

In our performance evaluation, we compare our proposed method LFPEC against the following five schemes:

1) Docker Hub: This scheme represents the typical scenario where systems without private registries download application images from remote cloud like [15] and [16]. In this scheme, we download application images directly from Docker Hub

without utilizing lazy pulling. That is, all the tasks of users can be processed until a compute node pulls a whole image from Docker Hub. As for the container assignment and registry selection, it calculates the total number of requests for each application in each edge area, and the application with the maximal value is assigned to the compute node of the target edge area.

2) Appstore with traditional lazy pulling (Appstore-L): In this scheme, all the compute nodes download application images from the appstore using the traditional lazy pulling method, which is also used in [7] and [23], without leveraging nearby local registries. The container assignment and registry selection scheme remains the same as the *DockerHub* scheme.

3) Most Popular (MP): This scheme starts caching process with the most popular application and caches an application image in the local registry of an edge area with the highest historical requests unless the cache size of the target registry is full. It is similar to one of the benchmarks compared in [10] and [13]. For the container assignment and registry selection, the total number of requests for each application in each edge area is calculated first. After that, it greedily selects an application with the highest value and chooses the pair of compute node and registry with the lowest cost and sufficient resources.

4) Most Popular with lazy pulling (MP-L): Similar to the *MP* scheme, this scheme utilizes edge-based collaborative lazy pulling but can only cache the whole image without partial caching. The container assignment and registry selection process remains the same as the *MP*.

5) Most Popular with partial caching and lazy pulling (MP-PL): The scheme, unlike *MP-L*, allows for the caching of partial versions of images and caches the most popular version of applications. The container assignment and registry selection process is identical to the *MP*.

C. Analysis of Cache Hit Rate Performance

The cache hit rate is calculated by dividing the total number of user requests that the processing compute nodes can download the required application packages from the local registry in the same edge area, by the total user requests.

1) The impact of cache size ratio: In terms of the impact of cache size ratio, Fig. 4 shows how the cache hit rate changes with varying cache size ratios of each registry. First, it is evident that the overall cache hit rate of our proposed algorithms is much higher than other compared algorithms. This can be attributed to our approach that better utilizes the storage space by partially caching the least functional images and flexibly selecting the version of images to cache. In addition, it can be observed that the cache hit rates of the *MP*, *MP-L*, and *MP-PL* schemes increase with the cache size ratio, as more images can be cached in the local registries.

2) The impact of edge-based collaborative partial caching: Regarding the impact of edge-based collaborative partial caching, schemes such as *MP* and *MP-L*, which can only cache the whole version of images, show decreased flexibility and leads to lower cache hit rates when the cache size is smaller. The *MP-PL* scheme consistently reaches a higher cache hit ratio than whole caching schemes when the registry cache size ratio is below 20% due to the adoption of partial caching. Furthermore, our proposed scheme achieves the highest cache hit ratio among all the other schemes and demonstrates a stronger inclination to fully utilize the available cache space.

To evaluate the stability, we examine the performance under different application usage scenarios. That is, adjusting the density of users with same application preferences in edge areas. Our proposed scheme reliably achieves the highest cache hit ratio in these diverse scenarios when compared with the baselines.

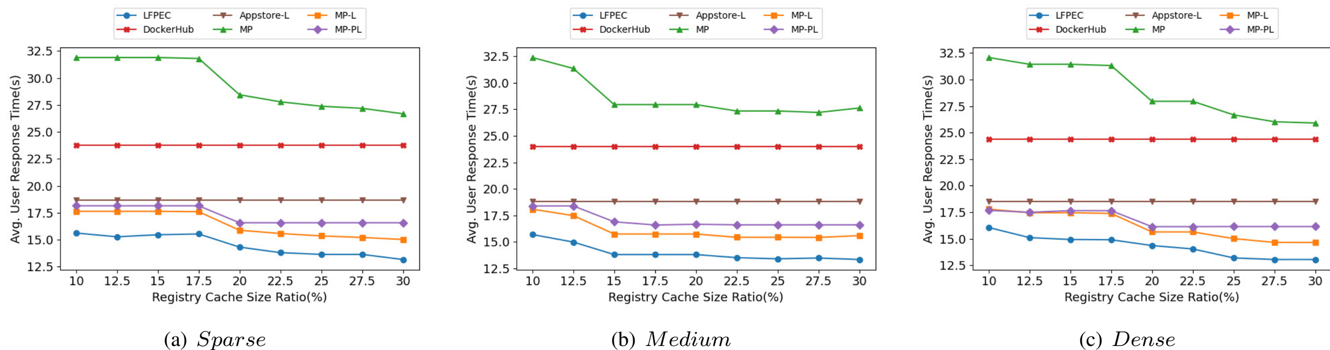
D. Analysis of Average User Response Time Performance

Fig. 5 illustrates the average response time of 20 iterations of different algorithms under different cache size ratios. The subfigures in different scenarios show that the average response time will be reduced when the registry cache size ratio increases. Moreover, with a denser application usage density, the value of the average response time will converge faster. The impact of distinct factors will be illustrated in detail below.

1) The impact of cache size ratio: As the cache size ratio increases, the average response time decreases because a larger number of images can be cached in the local registries. Therefore, compute nodes are given increased opportunities to retrieve application packages from a closer registry.

2) The impact of edge-based collaborative lazy pulling: The impact of edge-based collaborative lazy pulling is evident when comparing schemes that do not utilize lazy pulling, such as *DockerHub* and *MP*, with schemes that employ the lazy pulling method. It is obvious that the two schemes have a longer average user response time compared with other schemes. The schemes with lazy pulling demonstrate significantly reduced average response time, as they can respond to users by downloading only a portion of the application images, resulting in a shorter waiting time. Moreover, our proposed scheme further enhances the performance of lazy pulling by collaboratively utilizing local registries for downloading application images instead of solely relying on the same source as *Appstore-L* scheme. In summary, compared to *MP*, *DockerHub*, and *Appstore-L*, our proposed scheme outperforms the average user response time by an average of 50.8%, 40.7%, and 23.7%, respectively.

3) The impact of least functional-based partial caching: Additionally, the impact of least functional-based partial caching is examined. Our proposed algorithm outperforms *MP-L* 12.1% on average. We leverage space utilization by partially caching the least functional version of images, eliminating the need to store the whole images in local registries, which results in reduced average response time. However, we observe that *MP-PL* performs worse than *MP-L* as the registry cache size ratio increases. It is because an inadequately designed partial caching mechanism may degrade the response time owing to a longer pulling time for downloading the non-prioritized part of an image from the appstore. Moreover, when the target registry lacks sufficient space to store the most popular application packages, *MP-PL* wastes the remaining cache space and results in a longer average user response



(a) *Sparse* (b) *Medium* (c) *Dense*

Fig. 5. Average user response time versus registry cache ratio in different application usage density.

time. In our algorithm, we address this issue by alternatively caching the least functional version of the application image and making effective use of the remaining space.

An inherent limitation of the *MP-PL* is that it solely considers the popularity of applications and their versions when deciding where to cache the packages. However, it overlooks the fact that the time difference in downloading the least functional version of images from different registries is often shorter compared to retrieving the whole version of images. The advantages gained from deploying smaller-sized with fewer layers images may be compromised by the potential increase in pulling time caused by larger-sized images with a greater number of layers. We make up this by jointly considering the popularity, image size, and layer number of the images to prioritize the scheduling of versions and applications that have a higher impact on the system. The results demonstrate that, on average, our proposed algorithm can perform better than the *MP-PL* by 16.3%.

Similarly, we demonstrate the superior performance of our proposed algorithm against other baseline algorithms under different application usage density.

VI. Conclusions

In this paper, we formulate the problem of image caching, container assignment, and registry selection in edge computing where geographically distributed local registries cache application images, while compute nodes provide services to users. The primary objective of our study is to minimize the average response time experienced by users. To address the problem, we propose a novel least functional-based partial caching algorithm and an innovative edge-based collaborative lazy pulling algorithm leveraging the concept of cooperation between the appstore and local registries. In order to validate the effectiveness of our proposed algorithms, we conduct extensive experiments on a real-world testbed and using the real data of user application usage and popular images in Docker Hub. The results demonstrate that the proposed algorithms achieve the highest cache hit rate when compared to all the other baseline algorithms. Specifically, they reveal an average reduction in the user response time by 40.7%, 23.7%, 50.8%, 12.1%, and 16.3%, respectively, compared with *DockerHub*, *Appstore-L*, *MP*, *MP-L* and *MP-PL* algorithms.

In our future work, we decide to deploy an appropriate number of containers instead of only one container based on user requests, where a load balancing mechanism for reducing task completion time will also be incorporated. Additionally, besides only considering system scenario with no running applications on compute nodes, continuous-time requests can be explored and container migrations based on their popularity can be discussed in the future.

VII. Acknowledgement

Hung-Yu Wei is grateful for the funding support by National Science and Technology Council (NSTC) of Taiwan under Grant 113-2628-E-002-032-.

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, no. 11, pp. 1-16, Sep. 2015.

[2] D. Sabella, V. Sukhomlinov, L. Trang, S. Kekki, P. Paglierani, R. Rossbach, X. Li, Y. Fang, D. Druta, F. Giust *et al.*, “Developing software for multi-access edge computing,” *ETSI white paper*, vol. 20, pp. 1–38, 2019.

[3] Y. Chiang, et al., ”Management and Orchestration of Edge Computing for IoT: A Comprehensive Survey,” *IEEE Internet of Things Journal*, 2023.

[4] C. Babcock, ”Containers Explained: 9 Essentials You Need To Know”, InformationWeek. [Online]. Available: https://www.informationweek.com/it-strategy/containers-explained-9-essentials-you-need-to-know.

[5] T. L. Foundation. Kubernetes. [Online]. Available: https://kubernetes.io.

[6] O. Oleghe, “Container placement and migration in edge computing: Concept and scheduling models,” *IEEE Access*, vol. 9, pp. 68 028–68 043, 2021.

[7] T. Harter et al., ”Slacker: Fast Distribution with Lazy Docker Containers”, *Proc. USENIX Conf. File and Storage Tech.*, pp. 181-95, Feb. 2016.

[8] C.-C. Lin, Y. Chiang, and H.-Y. Wei, “Multi-service edge computing management with multi-stage coalition game task offloading,” *IEEE Transactions on Network and Service Management*, vol. 21, no. 3, pp. 3278–3291, 2024.

[9] Z. Chen, Z. Zhou and C. Chen, ”Code caching-assisted computation offloading and resource allocation for multi-user mobile edge computing”, *IEEE Trans. Netw. Serv. Manag.*, vol. 18, no. 4, pp. 4517-4530, Dec. 2021.

[10] Y. Hao, M. Chen, L. Hu, M. S. Hossain and A. Ghoneim, ”Energy efficient task caching and offloading for mobile edge computing”, *IEEE Access*, vol. 6, pp. 11365-11373, 2018.

[11] H. Jeon, S. Shin, C. Cho and S. Yoon, "Multi-Agent Learning-based Package Caching in Serverless Edge Computing," *Int. Conf. Inf. Commun. Technol. Converg. (ICTC)*, pp. 400-402, 2022.

[12] H. Jeon, S. Shin, C. Cho and S. Yoon, "Deep reinforcement learning for qos-aware package caching in serverless edge computing", *IEEE Global Commun. Conf. (GLOBECOM)*, pp. 1-6, 2021.

[13] X. Ma, A. Zhou, S. Zhang and S. Wang, "Cooperative service caching and workload scheduling in mobile edge computing", *IEEE Conf. Comput. Commun. (INFOCOM)*, pp. 2076-2085, Jul. 2020.

[14] J. Xu, L. Chen and P. Zhou, "Joint service caching and task offloading for mobile edge computing in dense networks", *Proc. IEEE INFOCOM Conf. Comput. Commun.*, pp. 207-215, 2018.

[15] 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. 2022.

[16] S. Fu, R. Mittal, L. Zhang and S. Ratnasamy, "Fast and efficient container startup at the edge via dependency scheduling", *Proc. 3rd USENIX Workshop Hot Top. Edge Comput.*, pp. 1-7, 2020.

[17] B. Gao, Z. Zhou, F. Liu and F. Xu, "Winning at the Starting Line: Joint Network Selection and Service Placement for Mobile Edge Computing," *IEEE Conf. Comput. Commun. (INFOCOM)*, pp. 1459-1467, 2019.

[18] B. Tang, F. Guo, B. Cao, M. Tang and K. Li, "Cost-aware Deployment of Microservices for IoT Applications in Mobile Edge Computing Environment," *IEEE Trans. Netw. Serv. Manag.*, 2022.

[19] H. Sami, H. Otrok, J. Bentahar and A. Mourad, "AI-Based Resource Provisioning of IoE Services in 6G: A Deep Reinforcement Learning Approach," *IEEE Trans. Netw. Serv. Manag.*, vol. 18, no. 3, pp. 3527-3540, Sept. 2021.

[20] L. Gu, D. Zeng, J. Hu, H. Jin, S. Guo and A. Y. Zomaya, "Exploring layered container structure for cost efficient microservice deployment", *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, pp. 1-9, 2021.

[21] H. Fan, S. Bian, S. Wu, S. Jiang, S. Ibrahim and H. Jin, "Gear: Enable efficient container storage and deployment with a new image format", *Proc. IEEE 41st Int. Conf. Distrib. Comput. Syst.*, pp. 115-125, 2021.

[22] H. Li, Y. Yuan, R. Du, K. Ma, L. Liu and W. Hsu, "DADI: Block-level image service for agile and elastic application deployment", *Proc. USENIX Annual Technical Conf. (ATC)*, pp. 727-740, 2020.

[23] J. L. Chen, D. Liaqat, M. Gabel, and E. de Lara, "Starlight: Fast container provisioning on the edge and over the WAN," *Proc. 19th USENIX Symp. Netw. Syst.e Des. Implementation (NSDI)*, pp. 35–50, 2022.

[24] X. Xia, F. Chen, Q. He, J. Grundy, M. Abdelrazek and H. Jin, "Online collaborative data caching in edge computing", *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 2, pp. 281-294, Feb. 2020.

[25] C. Wang, D. Feng, S. Zhang and Q. Chen, "Video caching and transcoding in wireless cellular networks with mobile edge computing: A robust approach", *IEEE Trans. Veh. Technol.*, vol. 69, no. 8, pp. 9234-9238, Aug. 2020.

[26] A. Lekharu, A. P. S. Chauhan, A. Sur, and M. Patra, "Reinforcement learning-based adaptive bitrate caching at mec server," *IEEE Transactions on Network and Service Management*, vol. 21, no. 3, pp. 3292–3304, 2024.

[27] Z. Teng, J. Fang, and Y. Liu, "Combining lyapunov optimization and deep reinforcement learning for d2d assisted heterogeneous collaborative edge caching," *IEEE Transactions on Network and Service Management*, vol. 21, no. 3, pp. 3236–3248, 2024.

[28] Q. Jia, R. Xie, H. Lu, W. Zheng and H. Luo, "Joint Optimization Scheme for Caching, Transcoding and Bandwidth in 5G Networks with Mobile Edge Computing," *Proc. IEEE 5th Int. Conf. Comput. Commun. (ICCC)*, pp. 999-1004, 2019.

[29] A. Mehrabi, M. Siekkinen and A. Ylä-Jaaski, "QoE-Traffic Optimization Through Collaborative Edge Caching in Adaptive Mobile Video Streaming," *IEEE Access*, vol. 6, pp. 52261-52276, 2018.

[30] Y. Chiang, C. -H. Hsu and H. -Y. Wei, "Collaborative Social-Aware and QoE-Driven Video Caching and Adaptation in Edge Network," *IEEE Trans. Multimedia*, vol. 23, pp. 4311-4325, 2021.

[31] H. Zhang, W. Lin, R. Xie, S. Li, Z. Dai, and J. Z. Wang, "An optimal container update method for edge-cloud collaboration," *Software: Practice and Experience*, vol. 54, no. 4, pp. 617–634, 2024.

[32] H. Yu, R. Basu Roy, C. Fontenot, D. Tiwari, J. Li, H. Zhang, H. Wang, and S.-J. Park, "Rainbowcake: Mitigating cold-starts in serverless with layer-wise container caching and sharing," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, 2024, pp. 335–350.

[33] "IEEE Standard for Edge/Fog Manageability and Orchestration," *IEEE Std 1935-2023*, pp.1-68, doi: 10.1109/IEEESTD.2023.10186301.

[34] T. Chen, Y. Chiang, J. Wu H. Chen, C. Chen and H. Wei, "IEEE P1935 Edge/Fog Manageability and Orchestration: Standard and Usage Example," *Proc. IEEE Int. Conf. Edge Comput. (EDGE)*, 2023.

[35] Y. Chiang, C.-H. Hsu, G.-H. Chen, and H.-Y. Wei, "Deep q-learning-based dynamic network slicing and task offloading in edge network," *IEEE Transactions on Network and Service Management*, vol. 20, no. 1, pp. 369–384, 2023.

[36] H.-T. Chen, Y. Chiang, and H.-Y. Wei, "Edge computing resource management for cross-camera video analytics: Workload and model adaptation," *IEEE Access*, vol. 12, pp. 12 098–12 109, 2024.

[37] Z. Tang, J. Lou and W. Jia, "Layer Dependency-Aware Learning Scheduling Algorithms for Containers in Mobile Edge Computing," *IEEE Trans. on Mob. Comput.*, vol. 22, no. 6, pp. 3444-3459, June 2023.

[38] A. Zhou, S. Li and S. Wang, "Task Offloading and Resource Allocation for Container-enabled Mobile Edge Computing," *IEEE Int. Conf. Services Comput. (SCC)*, pp. 222-232, 2021.

[39] A. Ahmed and G. Pierre, "Docker Container Deployment in Fog Computing Infrastructures," *2018 IEEE Int. Conf. Edge Comput. (EDGE)*, pp. 1-8, 2018.

[40] Li, Sisi, et al. "Commutativity-guaranteed docker image reconstruction towards effective layer sharing." *Proceedings of the ACM Web Conference*, pp. 3358-3366, 2022.

[41] Containerd. Stargz snapshotter. [Online]. Available: https://github.com/containerd/stargz-snapshotter.

[42] Docker Inc. Distribution. [Online]. Available: https://github.com/distribution/distribution.

[43] D. Yu, Y. Li, F. Xu, P. Zhang and V. Kostakos, "Smartphone app usage prediction using points of interest," *Proc. ACM Interact. Mobile Wearable and Ubiquitous Technol.*, vol. 1, no. 4, pp. 1-21, 2018.

[44] Docker Inc. Docker Hub container image library — app containerization. [Online]. Available: https://hub.docker.com/.

[45] L. Zhao, H. Li, N. Lin, M. Lin, C. Fan, and J. Shi, "Intelligent Content Caching Strategy in Autonomous Driving Toward 6G," *IEEE Trans. on Intell. Transport.*, vol. 23, no. 7, pp. 9786-9796, July 2022.

[46] A. Yousefpour, G. Ishigaki and J. P. Jue, "Fog Computing: Towards Minimizing Delay in the Internet of Things," *2017 IEEE Int. Conf. Edge Comput. (EDGE)*, pp. 17-24, 2017.

Chiao-Cheng Chen received the B.S. degree in computer science from National Yang Ming Chiao Tung University, Hsinchu, Taiwan, in 2021, and the M.S. degree in electrical engineering from National Taiwan University, Taipei, Taiwan, in 2023. Her research interests include edge computing, resource scheduling, 5G and container technology.

Yao Chiang received the B.S. degree and the M.S. degree in management information systems from National Pingtung University of Science and Technology (NPUST), Pingtung, Taiwan, in 2014 and 2016, respectively. He received the Ph.D. degree in electrical engineering at National Taiwan University (NTU) in 2021, and he is currently a postdoctoral research fellow in NTU. His research interests include data mining, machine learning and mobile communications design for multi-access edge computing systems.

Yu-Chieh Lee received the B.S. degree in electrical engineering from National Taiwan University (NTU), Taiwan, in 2023. She is currently pursuing the M.S. degree in electrical engineering at NTU. Her research interests include edge computing and resource allocation.

Hung-Yu Wei is a Professor in Department of Electrical Engineering and Graduate Institute of Communications Engineering, National Taiwan University. Currently, he is the Director of Graduate Institute of Communications Engineering and Chair of IoT Research Center. He served as Interim Department Chair and Associate Chair in Department of Electrical Engineering during 2019˜2022. He received the B.S. degree in electrical engineering from National Taiwan University in 1999. He received the M.S. and the Ph.D. degree in electrical engineering from Columbia University in 2001 and 2005 respectively. He was a summer intern at Telcordia Applied Research in 2000 and 2001. He was with NEC Labs America from 2003 to 2005. His research interests include next-generation wireless networks, IoT, and fog/edge computing.

He received the K.T. Li Distinguished Young Scholar Award 2009 from ACM Taipei/Taiwan Chapter in 2012, Excellent Young Engineer Award from the Chinese Institute of Electrical Engineering in 2014, Wu Ta You Memorial Award from MOST in 2015, and Outstanding Research Award from MOST in 2020. He was also recognized with the NTU Excellent Teaching Award three times. He serves as Chair of IEEE 1935 working group for edge/fog management and orchestration standard and Chair of IEEE 1934 Working Group for edge/fog computing and networking architecture. He is an Associate Editor for IEEE IoT Journal and IEEE IoT magazine. He was the Chair of IEEE VTS Taipei Chapter during 2016˜2017.