Multi-User Layer-Aware Online Container Migration in Edge-Assisted Vehicular Networks

Zhiqing Tang, *Member, IEEE*, Fangyi Mou, Jiong Lou, Weijia Jia, *Fellow, IEEE*, Yuan Wu, *Senior Member, IEEE*, and Wei Zhao, *Fellow, IEEE*

***Abstract*— In edge-assisted vehicular networks, containers are very suitable for deploying applications and providing services due to their lightweight and rapid deployment. To provide high-quality services, many existing studies show that the containers need to be migrated to follow the vehicles' trajectory. However, it has been conspicuously neglected by existing work that making full use of the complex layer-sharing information of containers among multiple users can significantly reduce migration latency. In this paper, we propose a novel online container migration algorithm to reduce the overall task latency. Specifically: 1) we model the multi-user layer-aware online container migration problem in edge-assisted vehicular networks, comprehensively considering the initialization latency, computation latency, and migration latency. 2) A feature extraction method based on attention and long short-term memory is proposed to fully extract the multi-user layer-sharing information. Then, a policy gradient-based reinforcement learning algorithm is proposed to make the online migration decisions. 3) The experiments are conducted with real-world data traces. Compared with the baselines, our algorithms effectively reduce the total latency by 8% to 30% on average.**

Manuscript received 19 February 2023; revised 25 September 2023; accepted 1 November 2023; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor J.-W. Lee. Date of publication 10 November 2023; date of current version 18 April 2024. This work was supported in part by the Chinese National Research Fund (NSFC) under Grant 62302048 and Grant 62272050; in part by the Science and Technology Development Fund (FDCT) of Macau SAR, under Grant 0015/2019/AKP; in part by the Open Foundation of Yunnan Key Laboratory of Software Engineering under Grant 2023SE207; in part by the Guangdong Key Laboratory of AI and Multi-modal Data Processing, Beijing Normal University-Hong Kong Baptist University United International College (UIC), under Grant 2020KSYS007; and in part by the Interdisciplinary Intelligence SuperComputer Center, Beijing Normal University (Zhuhai). *(Zhiqing Tang and Fangyi Mou contributed equally to this work.) (Corresponding authors: Weijia Jia; Yuan Wu.)*

Zhiqing Tang is with the Institute of Artificial Intelligence and Future Networks, Beijing Normal University, Zhuhai 519087, China, also with the State Key Laboratory of Internet of Things for Smart City, University of Macau, Macau, SAR, China, and also with the Yunnan Key Laboratory of Software Engineering, Kunming, Yunnan 650091, China (e-mail: zhiqingtang@bnu.edu.cn).

Fangyi Mou is with the Beijing Normal University–Hong Kong Baptist University, Zhuhai 519087, China (e-mail: moufangyi@uic.edu.cn).

Jiong Lou is with the Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai 200240, China (e-mail: lj1994@sjtu.edu.cn).

Weijia Jia is with the Institute of Artificial Intelligence and Future Networks, Beijing Normal University, Zhuhai 519087, China, and also with the Guangdong Key Laboratory of AI and Multi-Modal Data Processing, BNU–HKBU United International College, Zhuhai 519087, China (e-mail: jiawj@bnu.edu.cn).

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

Wei Zhao is with the CAS Shenzhen Institute of Advanced Technology, Shenzhen 518055, China (e-mail: zhao.wei@siat.ac.cn).

Digital Object Identifier 10.1109/TNET.2023.3330255

***Index Terms*— Layer-aware scheduling, container migration, edge computing, vehicular networks.**

I. INTRODUCTION

WITH the rapid development of fifth-generation (5G) and Internet of Vehicles, vehicles generate a large amount of raw data every day with more intelligent sensors equipped [1]. These raw data require local real-time processing, fusion, and feature extraction for target detection [2], path planning [3], computation offloading [4], [5], etc. Therefore, a low latency and reliable edge computing platform is significant. The edge-assisted vehicular network relies on edge-cloud collaboration and communication infrastructure provided by LTE/5G [6]. In addition, each roadside unit (RSU) is equipped with an edge node to provide more computing capability.

In edge-assisted vehicular networks, using containers to deploy applications can fully utilize the lightweight and easy deployment characteristics to achieve rapid and scalable Internet of Vehicles services. When the vehicle moves, the services requested by the vehicular user need to migrate with the user. With the maturity of Cellular Vehicle to Everything (C-V2X) communications, fast handover can be achieved [7]. However, a large amount of vehicular data in the edge node must also be migrated following the movement. Therefore, efficiently and timely migrating containers and vehicular data has been regarded as a critical issue.

The container migration mainly includes the migration of the writable container layer and the read-only container image file.[1] The image can be shared among users since many users request the same image, e.g., the task of road information processing [9]. When multiple vehicles move simultaneously, many repeated downloads during migration can be reduced if the sharing information of the requests is considered. Moreover, the image comprises multiple layers, and common layers can be shared by different images [8]. By optimizing which layers need to be migrated and which ones need to be downloaded, we expect to significantly reduce the latency for container migration [10].

It is very promising to make online container migration decisions considering the multi-user layer-sharing information to reduce the migration time and further reduce the

[1]The container image file contains the code, binaries, system tools, configuration files, etc., required to run the container [8]. We use 'image' in this paper to refer to the container image file for simplicity.

total latency. Nevertheless, the following challenges require to be addressed. First, how to fully extract the multi-user layer-sharing information. Different users may request the same set of layers, i.e., the same image, while some layers can be shared among multiple users. Considering the geographic location information of layers and node heterogeneity, extracting layer-sharing features and making wise migration decisions efficiently is technically challenging. Moreover, when multiple users are moving, considering the moving targets, the overlap of paths, and the same requested layers, the users requiring the same set of layers and having similar paths can be migrated to the same node to reduce the migration latency further. To solve these issues, a self-attention-based encoder network is designed to extract interdependence features of layers [11]. And a long short-term memory (LSTM [12])-based neural network is designed to input each migration decision to the next time slot to extract the multi-user features further.

The second challenge focuses on making appropriate online scheduling decisions based on the extracted multi-user layer-sharing information to gain long-term benefits in less total latency, i.e., the sum of initial latency, computation latency, and migration latency. Compared with heuristic algorithms, Reinforcement Learning (RL) algorithms can fully consider the impact of continuous decisions [13]. The long-term benefits and the impact of layer sharing can be fully considered with a reward function. Moreover, the strategy of RL can be learned and updated through the loss function, adapting to complex environments without any human intervention. Thus, RL-based algorithms are suitable for online decision-making, and a policy gradient-based RL algorithm [14] is further utilized to reduce the total latency.

To the best of our knowledge, there exist few studies investigating the multi-user online container migration problem considering the layer sharing in edge-assisted vehicular networks. An Online Container Migration (OCM) algorithm is proposed based on the proximal policy optimization (PPO) algorithm [14]. The resources of heterogeneous edge nodes, the features of tasks, and the layer dependencies are fully considered in the input state. The self-attention method is used to extract the layer-sharing features. The policy network and value function of the RL agent are also carefully designed to make migration decisions based on feature embeddings and combinations. Finally, experiments are conducted based on real-world mobility traces of taxi cabs [15] to verify the performance of the algorithm. The image and layer information is crawled from Docker Hub [16]. The proposed algorithm is compared with the default scheduling algorithm of Kubernetes [17] and the state-of-the-art layer-based heuristic algorithms [8]. Experimental results show that the proposed algorithm performs better than all baseline algorithms.

The contributions of our work are summarized as follows.

1) We model the multi-user layer-aware online container migration problem in edge-assisted vehicular networks for the first time. Our objective is to minimize the initial latency, computation latency, and migration latency of vehicular tasks.
2) To fully consider the layer-sharing information during migration, a self-attention-based encoding network is designed. Then, a feature extraction network is proposed to capture the multi-user trajectory features. An OCM algorithm is further proposed based on policy gradient RL to make online container migration decisions.
3) The experiments are conducted based on large-scale real-world taxi traces and image traces. The results validate the effectiveness of our proposed OCM algorithm. The total latency is reduced by up to 53%.

The remainder of the paper is organized as follows. In Section II, the related work and motivation examples are illustrated. In Section III, the system model and problem formulation are described. OCM algorithm is proposed in Section IV. Performance is evaluated in Section V. Finally, Section VI gives some discussions, and Section VII concludes the paper.

II. Related Work and Motivation

A. Container and Service Migration

Some researchers have modified the container architecture or designed the migration strategy to reduce the migration cost. Shi et al. [18] propose a system across data centers to improve the migration performance by reducing the amount of dirty data in the migration process. Fu et al. [19] propose a runtime system that effectively deploys microservice-based services in the cloud-edge continuum to minimize the required computational resources. CloudHopper [20] provides the design and implementation of live migration of containers across cloud providers.

To further meet the mobility requirements in edge computing, Tang et al. [21] and Wang et al. [22] model the container and service migration strategy as a Markov Decision Process (MDP) and propose migration algorithms to reduce the migration delay and power consumption. Wang et al. [23] propose a novel learning-driven method to make effective online migration decisions. To address unpredictable user mobility, Ouyang et al. [24] study the mobile edge service performance optimization problem under long-term cost budget constraints and design an approximation algorithm based on Markov approximation to seek a near-optimal solution. Ma et al. [25] consider the user mobility and service delay requirements and formulate two optimization problems of user service request admissions to maximize the accumulative network utility. Zhang et al. [26] propose an online lazy-migration adaptive interference-aware algorithm for real-time virtual network function (VNF) deployment and cost-efficient VNF migration in a 5G network slice to maximize the total throughput. Besides, there are many studies on virtual machine migration. For example, Han et al. [27] propose an approximate MDP-based dynamic virtual machine management method.

B. Layer-Aware Scheduling

Layer-aware scheduling research is currently in its infancy. Ma et al. [10] propose an edge computing platform architecture that supports seamless migration of offloaded services, which uses the layered features of the storage system to reduce the synchronization cost of the file system. Rong et al. [28] pull 3735 representative images from Docker Hub and find

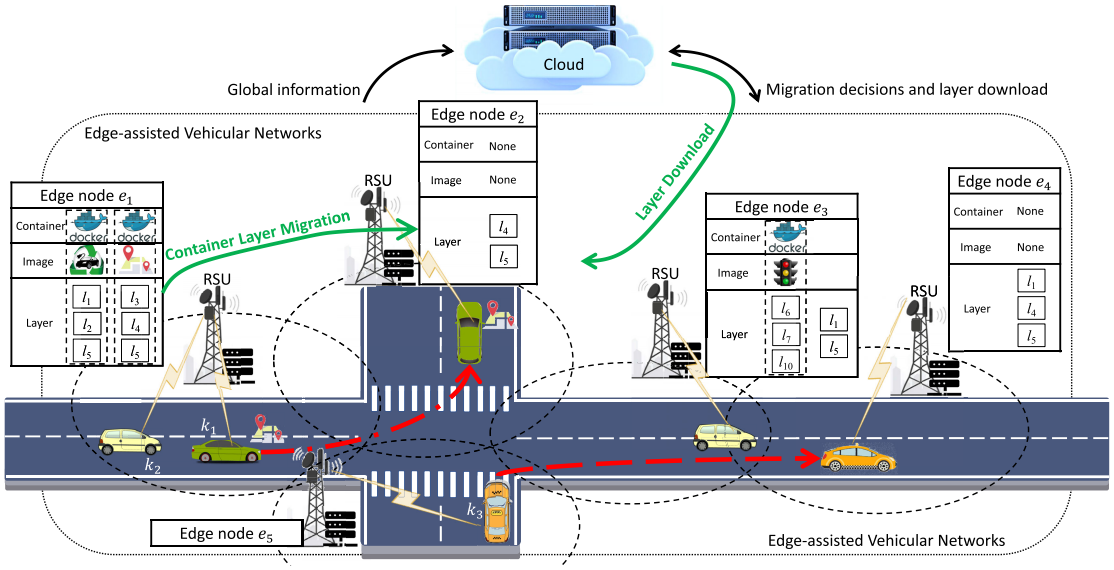


Fig. 1. An example of multi-user layer-aware online container migration. Task k_2 requests layers l_1, l_2, l_3, l_4, and l_5. When k_2 moves, if layer sharing is not considered, the container requested by k_2 should be migrated to e_3, and layers l_2, l_3, and l_4 need to be downloaded. When considering vanilla layer sharing, the container should be migrated to e_4 because there are more required layers on it. However, when considering multi-user layer sharing, the container should be migrated to e_3 to minimize the overall download cost for multiple users. More detailed explanations are in Section II-C and TABLE IV.

that the image layers can be cached in the destination servers to reduce the migration time.

Gu et al. [29] propose a layer-sharing microservice deployment and image-pulling strategy that explores the advantage of layer-sharing to speed up microservice startup and lower image storage consumption. Besides, they also study a layer aware microservice placement and request scheduling at the edge [30]. Lou et al. [31] formulate the container assignment and layer sequencing problem and prove its NP-hardness. A layer-aware scheduling algorithm is further proposed. Dolati et al. [32] address essential aspects of orchestrating services such as downloading and sharing container layers and steering traffic among network functions. Liu et al. [33] study the optimal deployment strategy to balance layer sharing and chain sharing of microservices to minimize image pull delay and communication overhead.

However, the above studies are all based on offline decision-making. Tang et al. [8] model the online layer-aware scheduling problem and propose an online scheduling algorithm. To our knowledge, this paper is the first study to make online migration decisions based on layer-sharing information.

TABLE I
EXAMPLE LAYERS

Layer	l_1	l_2	l_3
Size	1 MB	2 MB	3 MB
Layer	l_4	l_5	l_6
Size	6 MB	20 MB	40 MB

TABLE II
EXAMPLE TASKS

Task	Requested Image	Layers
k_1	i_1	l_1, l_2, l_5
k_2	i_2	l_1, l_2, l_3, l_4, l_5
k_3	i_3	$l_1, l_2, l_3, l_4, l_5, l_6$

TABLE III
EXAMPLE EDGE NODES

Edge Node	Layers
e_1	l_1, l_2, l_3, l_4, l_5
e_2	l_4, l_5
e_3	$l_1, l_5, l_6, l_7, l_{10}$
e_4	l_1, l_4, l_5

C. Motivation Example

When migrating, it is important to consider the trade-off between the migration cost and communication latency [21]. To illustrate the importance of multi-user layer sharing, in this example, we only consider the download cost incurred when migrating. As shown in Fig. 1, in the edge-assisted vehicular networks, different services are deployed on the edge nodes associated with the RSUs through containers. The vehicles k_1, k_2 and k_3 are moving. TABLE I shows six different layers of different sizes. The tasks k_1, k_2 and k_3 request the image i_1, i_2, and i_3, respectively. The corresponding requested layers are shown in TABLE II. The layers stored on each node are shown in TABLE III.

As shown in Fig. 1, three vehicles start from node e_1 or e_5 and move along the road, so the containers they request should be migrated to nodes e_2, e_3 or e_4. As shown in TABLE IV, when no layer sharing is considered, as k_1 moves, the containers it requests will be migrated from edge node e_1 to node e_2. Therefore, the layers that need to be downloaded are l_1 and l_2, and the download size is 3 MB. On the other hand, for tasks k_2 and k_3, the download sizes are 11 MB and 45 MB, respectively. Therefore, the total download size is 59 MB when layer sharing is not considered.

TABLE IV
EXAMPLE OF MULTI-USER LAYER SHARING

Sharing Method	Task	Migration Path	Download Layers	Download Size	Total Size
None Layer Sharing	k_1	$e_1 \rightarrow e_2$	l_1, l_2	3 MB	59 MB
	k_2	$e_1 \rightarrow e_3$	l_2, l_3, l_4	11 MB	
	k_3	$e_5 \rightarrow e_4$	l_2, l_3, l_6	45 MB	
Vanilla Layer Sharing	k_1	$e_1 \rightarrow e_3$	l_2	2 MB	16 MB
	k_2	$e_1 \rightarrow e_4$	l_2, l_3	5 MB	
	k_3	$e_5 \rightarrow e_3$	l_3, l_4	9 MB	
Multi-User Layer Sharing	k_1	$e_1 \rightarrow e_3$	l_2	2 MB	11 MB
	k_2	$e_1 \rightarrow e_3$	l_3, l_4	9 MB	
	k_3	$e_5 \rightarrow e_3$	None	0	

Vanilla layer sharing refers to single-user layer sharing without considering the layer sharing among users, which only considers the layer sharing of a single user on different nodes [8], [31]. Since there are more layers on e_3 than e_2, k_1 can be migrated to e_3 so that only l_2 needs to be downloaded, and the corresponding download size is 2 MB. Similarly, k_2 can be directly migrated to e_4, and k_3 can be migrated to e_3. As a result, the total download size should only be 16 MB. Therefore, after considering vanilla layer sharing, the download size can be reduced from 59 MB to 16 MB, effectively reducing the migration cost.

As shown in TABLE IV, such a migration decision of vanilla layer sharing is not problematic when the layers cannot be shared among users. However, in reality, layers requested by different users can also be shared, so multi-user layer sharing needs to be considered. The task k_2 and k_3 both request the layers i_1, i_2, i_3, i_4, and i_5. Their paths from e_2 are the same. Therefore, if the container of k_2 is directly migrated to e_3, the download size can be further reduced to 11 MB.

From the above example, the migration cost can be effectively reduced if multi-user layer sharing can be fully considered when making migration decisions. Furthermore, this requires extracting the layer-sharing information and multi-user information fully. In this paper, in addition to considering the download cost, we also consider the initialization cost and computation cost during migration so that the problem will be much more complicated than the example. The online container migration algorithm is proposed to achieve better long-term benefits, and the details are as follows.

TABLE V
NOTATIONS

$\mathbf{U}$	Mobile user set
u	u^{th} mobile user ($u \in \mathbf{U}$)
$\mathbf{K}$	Task set
k	k^{th} task ($k \in \mathbf{K}$)
f_k	CPU request of task k
m_k	Memory request of task k
d_k	Size of task k
$o_k(t)$	Location of task k at time t
$\mathbf{E}$	Edge node set
e	e^{th} edge node ($e \in \mathbf{E}$)
o_e	Location of edge node k
$o_{k,e}(t)$	Distance between task k and node e at time t
$\mathbf{C}$	Container set
b_c	Bandwidth of container
$\mathbf{L}$	Layer set
$\mathbf{L}_k$	Requested layers of task k
d_l	l^{th} layer size
b_e	Bandwidth of edge node server
d_e	Storage capacity of edge node server
n_c^e	Number of containers c on edge node e
T_k^{comm}	Communication latency for task k
T_k^{down}	Download latency for task k
T_k^{init}	Initialization latency for task k
T_k^{comp}	Computation latency for task k
T_k^{mig}	Migration latency for task k
T_k	Total latency for task k

III. SYSTEM MODEL AND PROBLEM FORMULATION

A. System Model

We consider a dynamic edge-assisted vehicular network architecture. As shown in Fig. 1, edge nodes are deployed on the roadside. The number and capacity of edge nodes can be upgraded when needed. New tasks from new vehicles (i.e., users) arrive at any time, e.g., real-time navigation information, real-time road condition information processing, etc. These tasks have stringent latency requirements. Hence, they must be offloaded to the edge-assisted RSU for real-time processing. The required containers should be initialized on edge nodes before executing tasks, which incur startup latency. Each task runs in one corresponding container [8]. During the execution of tasks, as the vehicles move, the containers they request must also be migrated along the moving trajectory, thereby reducing the task communication latency. Migration will bring a certain amount of migration cost, which can be effectively reduced when considering multi-user layer sharing. The main components are defined as follows.

Task: The set of mobile users, denoted as $u \in \mathbf{U} = \{u_1, u_2, \ldots, u_{|\mathbf{U}|}\}$, are connected to different RSUs through wireless access technology, i.e., 5G or WiFi, where $|\cdot|$ is used to indicate the number elements in the set, e.g., $|\mathbf{U}|$ is the number of users. The task $k \in \mathbf{K} = \{k_1, k_2, \ldots, k_{|\mathbf{K}|}\}$ is generated from different users and offloaded to the edge nodes for processing. The data size for task k is d_k. The location of task k at time t is denoted as $o_k(t)$, a two-dimensional coordinate value.

Edge node: A set of RSUs are deployed at the roadside. Each RSU is equipped with an edge node to provide computing capability. The edge nodes, denoted as $e \in \mathbf{E} = \{e_1, e_2, \ldots, e_{|\mathbf{E}|}\}$, provide the services to users through containers. The bandwidth of the edge node e is b_e. Moreover, the

storage capacity is d_e. The maximum number of running containers is C_e. The resource capacity of each node can be adjusted. Besides, the remote cloud can be considered an edge node with unlimited computing capacity, denoted as $e_{|\mathbf{E}|+1}$.

Container: A group of containers, denoted as $\mathbf{C} = \{c_1, c_2, \ldots, c_{|\mathbf{C}|}\}$, are deployed on the edge nodes. The bandwidth requirement of the container $c \in \mathbf{C}$ is b_c. Each container needs an image file. The set of images are denoted as $\mathbf{I} = \{i_1, i_2, \ldots, i_{|\mathbf{I}|}\}$. Requesting a container is equivalent to requesting the corresponding image. The difference between a container and an image is only a writable container layer, i.e., i_0. Specifically, when a new container is created, a new writable layer on top of the underlying layers is added, which is often called the "container layer" [34]. All changes to the running container, e.g., writing new files or modifying existing files, are written to this writable container layer. Thus, multiple containers can share access to the same underlying image and yet have their own data. This means that only the container layer needs to be transferred when migrating. Each image is composed of several layers, and the set of layers is denoted as $\mathbf{L} = \{l_1, l_2, \ldots, l_{|\mathbf{L}|}\}$. The size of each layer $l \in \mathbf{L}$ is denoted as d_l. After scheduling, the node assigned by this task is represented as e_k. The requested layers of task k can be denoted as $\mathbf{L}_k$. Generally, we assume that only one task runs in each container [8], [31].

B. Cost

The scheduler should carefully decide whether to migrate the associated containers with the user if the user u moves, comprehensively considering the initialization latency, computation latency, and migration latency.

Initialization latency: The initialization latency includes the communication latency between the mobile user and the RSU and the container startup latency in the edge node. Each mobile user u is associated with its nearby RSU to offload the task, e.g., the real-time traffic information processing task.

The mobile users connected to the RSU equally share the bandwidth resources. The wireless uplink transmission rate $\xi_{k,e}^{comm}(t)$ for the task k with data size d_k to edge node e is defined as follows [35] and [36].

$$\xi_{k,e}^{comm}(t) = b_{k,e}(t) \times \log\left(1 + \frac{p_k \times |h_{k,e}(t)|^2}{b_{k,e}(t) \times \sigma}\right), \quad (1)$$

where $b_{k,e}(t)$ is the allocated bandwidth resource of the edge node e for the task k at time slot t and p_k is the transmission power of the task k. $h_{k,e}(t)$ is the channel gain between the mobile device and the corresponding edge node, which is calculated as $h_{k,e}(t) = 10\beta \log(o_{k,e}(t)) + 5$ [35], where β is the path loss coefficient and $o_{k,e}(t)$ is the distance between task k and edge node e at time t. σ is the power spectral density of the Gaussian white noise.

Then, the communication latency for the task k on edge node e can be calculated as:

$$T_k^{comm} = \int \frac{d_k}{\xi_{k,e}^{comm}(t)} dt. \quad (2)$$

When the task k is scheduled to edge node e, the requested container c needs to be started. We define the binary variable $x_{c,k} \in \{0, 1\}$. If $x_{c,k} = 1$, the container c is requested by the task k. Otherwise, the container c is not requested. The binary variable $y_{c,l} \in \{0, 1\}$ is defined. If $y_{c,l} = 1$, the layer l is contained in the container c. Otherwise, the layer is not contained in the container. Besides, the binary variable $z_{l,e}(t) \in \{0, 1\}$ is defined. If $z_{l,e}(t) = 1$, the layer l is located on edge node e at time t. Otherwise, it is not.

Then, the download size required to initialize task k can be calculated as follows.

$$D_{k,e} = \sum_{c \in \mathbf{C}} \sum_{l \in \mathbf{L}} x_{c,k} \times y_{c,l} \times (1 - z_{l,e}(t)) \times d_l. \quad (3)$$

Compared with the significant transmission delay, the propagation delay and queuing delay can be ignored [30], [37]. Thus, the download latency can be obtained as follows:

$$T_k^{down} = \int \frac{D_{k,e}}{b_{k,e}(t)} dt. \quad (4)$$

The data communication and layer download can proceed simultaneously so that the initialization time can be calculated as follows:

$$T_k^{init} = \max\left(T_k^{comm}, T_k^{down}\right). \quad (5)$$

Computation latency: The tasks are offloaded to the RSU associated with an edge node. Multiple mobile users share the computing resources of edge nodes to process their applications. At time slot t, the processing density of the offloaded tasks is denoted as κ. Thus, the required CPU cycles for processing the tasks can be calculated as $f_k = d_k \times \kappa$. Moreover, the workload of the serving edge node is defined as $w_e(t)$, and the total computing capacity of the serving edge node is $F_e(t)$. A weighted resource allocation strategy is considered on each edge node, where tasks are allocated with computation resources proportional to their required CPU cycles [23]. Therefore, the computation delay of executing the task can be calculated as

$$T_k^{comp} = \int \frac{f_k}{\frac{f_k}{w_e(t)+f_k} \times F_e(t)} dt. \quad (6)$$

Migration latency: Let th_o denote the maximum communication distance between the task and edge node. If $o_{k,k_e}(t) > th_o$, then the task k needs to be migrated. As explained in [30], the container contains a list of read-only layers $\mathbf{L}_k \in \mathbf{L}$ and a writable layer l_k. The writable layer is on the top of a container that records the changes in the running container. The other underlying read-only layers can be downloaded from the cloud registry. Hence, only the writable layer needs to be migrated. Define the binary variable $x_{c,k,e}(t)$, when $x_{c,k,e}(t) = 1$, it means that the container c needs to be migrated to the edge node e at time slot t. The download size from the cloud can be calculated as follows:

$$D_{c,k,e} = \sum_{e \in \mathbf{E}} \sum_{c \in \mathbf{C}} \sum_{l \in \mathbf{L}} x_{c,k,e}(t) \times y_{c,l} \times (1 - z_{l,e}(t)) \times d_l. \quad (7)$$

Then, the download time can be calculated as follows:

$$T_k^{down} = \int \frac{D_{c,k,e}}{b_{k,e}(t)} dt. \quad (8)$$

Moreover, the writable container layer must be migrated from the original edge node to the destination edge node. The migration latency can be obtained as follows:

$$T_k^{mig} = \max\left(T_{c,k,e}^{down}, \frac{d_{l_k}}{b_v} + \alpha_k \times h_{e',e}\right), \tag{9}$$

where b_v is the wire transfer rate between edge nodes. α_k is a positive coefficient, indicating the experience value of one hop. $h_{e',e}$ is the two-dimensional state mapping of the shortest path from the original edge node to the target edge node and measures the number of hops [22]. During the migration, in addition to the transmission latency caused by migration, there will also be handover latency and startup latency. However, this latency is minimal compared to the transmission latency [21], so we only consider the transmission delay in this paper.

Finally, the total latency of container migration is denoted as:

$$T_k = T_k^{init} + T_k^{comp} + T_k^{mig}. \tag{10}$$

C. Problem Formulation and Analysis

Constraints: It is assumed that the scheduler is located at the remote cloud or a master RSU [21]. When the vehicle moves, the decision must meet the resource capacity and the storage resource limit of the associated edge node. The container number limit is used to denote the resource constraint of the edge nodes, which can be described as:

$$|\mathbf{C}_e(t)| \leq C_e, \quad \forall t, \forall e. \tag{11}$$

The storage resource limit of each node is defined as follows:

$$\sum_{l \in \mathbf{L}} (1 - z_{l,e}(t)) \times d_l \leq d_e, \quad \forall t, \forall n. \tag{12}$$

Moreover, each task should be scheduled to only one node or the cloud. We denote the scheduling results by $\{u_{k,e} | e \in \mathbf{E} \cup \{e_{|\mathbf{E}|+1}\}\}$, where $u_{k,e} = 1$ if the task k is scheduled to edge node e, otherwise, $u_{k,e} = 0$. Then, this constraint can be represented as:

$$\sum_{e \in \mathbf{E} \cup \{e_{|\mathbf{E}|+1}\}} u_{k,e} = 1, \quad \forall k. \tag{13}$$

Problem Formulation: We aim to minimize the overall total latency from a long-term perspective, which is defined in Eq. (10). The target is to find the best strategy to minimize the overall time while obeying the constraints. The Online Container Migration (OCM) problem is defined as follows:

$$\begin{aligned} \textbf{Problem OCM.} \ & \min T = \sum_{k \in \mathbf{K}} T_k, \\ & \text{s.t. Eqs. (11), (12), (13),} \\ & x_{c,k} \in \{0,1\}, \forall c \in \mathbf{C}, \forall k \in \mathbf{K}, \\ & y_{c,l} \in \{0,1\}, \forall c \in \mathbf{C}, \forall l \in \mathbf{L}, \\ & z_{l,e}(t) \in \{0,1\}, \forall l \in \mathbf{L}, \forall e \in \mathbf{E}. \end{aligned}$$

Problem OCM is an advanced bin-packing problem, which is NP-hard and can only be solved heuristically. The goal is to make online migration decisions in a dynamic vehicular network and obtain long-term benefits. However, decisions are made according to a deterministic strategy for most heuristic algorithms, which cannot consider the dynamic environment and the impact of continuous decisions. For meta-heuristic algorithms, all future information needs to be known if used to solve this problem from a long-term perspective. Nevertheless, the tasks arriving in the future are unknown. As a result, most of the existing heuristic and meta-heuristic algorithms are unstable in a real vehicular network environment.

In this problem, the first-order transition probability of the tasks' resource demand is quasi-static for an extended period and not uniform distribution by adequately choosing the time slice duration [38]. Moreover, the arrival of tasks and the environment update have the memoryless property [21]. Therefore, this problem can be modeled as an MDP. RL algorithms are suitable for solving MDP problems [39]. In RL algorithms, at each time t, the RL agent collects system state s_t and calculates the reward during the last time slice r_{t-1}. Then, the agent selects action a_t according to a pre-defined strategy. After performing the action, the system transits to the new state s_{t+1} in the next time slice. Based on the collected state, action, reward, and a proper discount factor, a value can be calculated to denote the expected long-term return. The reward is an immediate signal received in a given state, while the value is a long-term expectation. The RL agent might receive a low, immediate reward even as it selects an action with great potential for long-term value. By value function, the RL agent can optimize the policy and make decisions from a long-term perspective.

IV. Algorithms

A. Algorithm Settings

The state, action, and reward are defined to train the agent.

State: A state s_t is a complete description of the vehicular networks. As shown in Fig. 2, the state contains three aspects: the task, nodes, and the action embedding. The state s_t^k of task k is described as follows. First, the location $o_k(t)$ of the task at time t, the size d_k, the CPU request f_k, and the memory request m_k are all features that affect decision-making. The requested layer set $\mathbf{L}_k$ is also essential. The total size of layers that need to be downloaded is calculated as $d_{\mathbf{L}_k} = \sum_{l \in \mathbf{L}_k} d_l$. Besides, the distance between task k and all edge nodes are also significant, denoted as $\mathbf{O}_k(t) = \{o_{k,1}(t), \ldots, o_{k,|\mathbf{E}|}(t)\}$. $\mathbf{O}_k(t)$ is different for each task, which has a great impact on the initialization latency. In short, the state of task k is $s_t^k = \{o_k(t), d_k, f_k, m_k, \mathbf{L}_k, d_{\mathbf{L}_k}, \mathbf{O}_k(t)\}$.

The location of each edge node $\mathbf{O}_e = \{o_1, o_2, \ldots, o_{|\mathbf{E}|}\}$, the available storage capacity $\mathbf{D}_e(t) = \{d_1(t), d_2(t), \ldots, d_{|\mathbf{E}|}(t)\}$, available CPU capacity $\mathbf{F}_e(t) = \{f_1(t), f_2(t), \ldots, f_{|\mathbf{E}|}(t)\}$, and the available memory capacity $\mathbf{M}_e(t) = \{m_1(t), m_2(t), \ldots, m_{|\mathbf{E}|}(t)\}$ are used as part of the state. $\mathbf{O}_e$ can be used to calculate the distance between nodes and has an impact on initialization latency and migration latency. Besides, $\mathbf{O}_e$ may vary with the node location, e.g., mobile edge nodes like drones. Therefore, it is also added to the state. The layer status $\mathbf{L}_e(t) = \{z_{1,e}(t), \ldots, z_{|\mathbf{L}|,e}(t)\}$ are also significant. Then, the state of the node e can be denoted

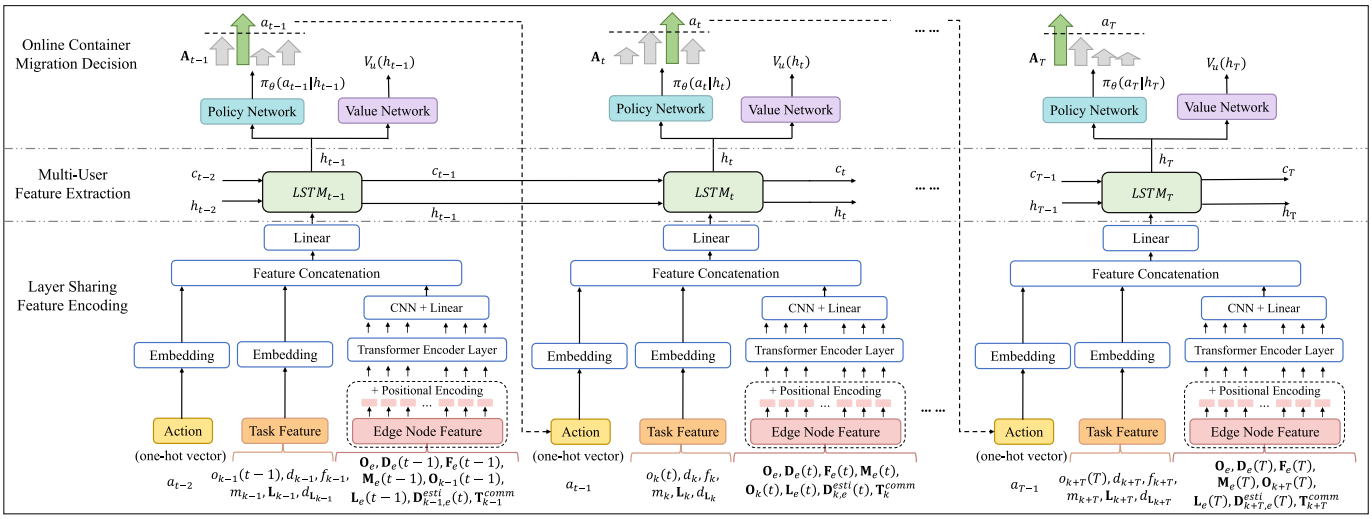


Fig. 2. Algorithm overview.

as $s_t^e = \{\mathbf{O}_e, \mathbf{D}_e(t), \mathbf{F}_e(t), \mathbf{M}_e(t), \mathbf{L}_e\}$. In this way, the resource capacity adjustment of the node will only affect the remaining resources of the node, and will not affect the state representation and feature extraction.

Moreover, to fully explore the layer sharing on each node and consider the computation resources, the estimated download size $\mathbf{D}_{k,e}^{esti}(t)$ can be calculated by Eq. (3). The estimated communication latency from the task k to all edge nodes are denoted as $\mathbf{T}_k^{comm} = \{T_{k,1}^{comm}, \ldots, T_{k,|\mathbf{E}|}^{comm}\}$. Due to the influence of continuous actions, the action at the last moment is also used as part of the state. To improve the representation ability of the action a_{t-1}, we convert it into embeddings by looking up a trainable embedding matrix, i.e., $\mathcal{A}_{t-1}$.

To sum up, the state at time t can be denoted as:

$$s_t = \{s_t^k, s_t^e, \mathbf{D}_{k,e}^{esti}(t), \mathbf{T}_k^{comm}, \mathcal{A}_{t-1}\} \tag{14}$$

Action space: A task can be offloaded to any available edge server. Meanwhile, with the mobility of the agent, the task can be migrated to any valid node or the remote cloud. Therefore, the action at time t can be defined as $a_t \in \mathbf{E} \cup \{e_{|\mathbf{E}|+1}\}$. When the number of nodes changes, the action space must also be modified accordingly.

Reward: The reward function r_t is critically important for long-term revenue. The goal of the agent is to maximize the reward, while in vehicular networks, the goal is to minimize the total latency. So, the reward can be obtained as $r_t = -T_k$.

B. Online Container Migration

In this section, the Online Container Migration (OCM) algorithm is introduced, including layer-sharing feature encoding, multi-user feature extraction, and the training of the algorithm.

Overview: The OCM algorithm is shown in Algorithm 1. To facilitate traversal of tasks, all tasks are added to a priority queue $\mathcal{Q}_1$, and use the start time of the task as the key [8]. In other words, $\mathcal{Q}_1$ stores all the tasks that need to be processed, and we take the tasks out of $\mathcal{Q}_1$ according to the time order for scheduling. We can select the next task to be scheduled according to the priority, task interaction, or other indicators, but this is not the focus of this paper. Besides, an empty priority queue $\mathcal{Q}_2$ is initialized to record running tasks. Through $\mathcal{Q}_2$, it can be judged whether a task is still running or has finished running. In practice, reading the task process can usually achieve this goal. As shown in lines 1 - 4, in the beginning, the first task k_0 is obtained from $\mathcal{Q}_1$. Then the location $o_k(t_0)$ and the filtered set of edge nodes $\mathbf{A}_{t_0}$ are obtained, which is a set of all edge nodes whose distance from task k_0 is within the range of communication distance. The hidden parameters of the policy network are initialized.

Each time t, the agent observes the state s_t and then selects the action a_t according to the policy π_t and filtered action set $\mathbf{A}_t$. The key-value tuple $(t + T_k^{comp}, k)$ is put into $\mathcal{Q}_2$ to be processed. Then, as shown in lines 10 - 22, when the task queue $\mathcal{Q}_2$ is not empty, the agent processes tasks in $\mathcal{Q}_2$ according to time order sorted by the key t_k. If $t_k < t$, the task k is finished, then the resources are released. Otherwise, the latest location $o_k(t)$ is obtained. Then, the distance $o_{k,k_e}(t)$ between the task k and the corresponding edge node k_e is calculated. If the task needs to be migrated, it is put into the task queue $\mathcal{Q}_1$. Otherwise, the task is returned to $\mathcal{Q}_2$.

After that, the first task in the queue $\mathcal{Q}_1$ is obtained. The environment is updated, and the new filtered set $\mathcal{A}_{t+1}$ is obtained. The layer-sharing features are extracted through an encoder network based on the attention mechanism and an LSTM-based recurrent neural network (RNN). The extracted features are fused into the policy network. Finally, the policy π is trained and updated through the policy gradient method. The details will be introduced in the following.

Layer sharing feature encoding: To fully extract the layer-sharing information, a self-attention-based encoder network is designed [11]. As shown in Fig. 2, the layer-sharing feature encoding mainly includes three parts: action embedding, task feature embedding, and edge node feature encoding. The action embedding and task feature embedding map the last action and task features to two embedding vectors.

For the node features, a self-attention-based network is designed based on Transformer [11], which receives as input

Algorithm 1 The OCM Algorithm

Input : $\mathcal{Q}_1, \mathcal{Q}_2 = \emptyset, \pi_\theta, \mathbf{N}$
Output: a_t
Get the first task k_0 from $\mathcal{Q}_1$;
Get the location $o_k(t_0)$ of task k_0 at time t_0 ;
Get the filtered set of edge nodes $\mathbf{A}_{t_0}$;
Initialize the hidden parameters;
for $t = 1, 2, \ldots$ **do**
  Get the state s_t by Eq. (14);
  Select the node $e = a_t$ according to π_t and $\mathbf{A}_t$;
  Calculate the computation time T_k^{comp} by Eq. (6);
  Put $(t + T_k^{comp}, k)$ to $\mathcal{Q}_2$;
  while $\mathcal{Q}_2 \neq \emptyset$ **do**
    Get the first task (t_k, k) from $\mathcal{Q}_2$;
    if $t_k < t$ **then**
      Release the resource by task k;
    else
      Get the location $o_k(t)$;
      if $o_{k,k_e}(t) > th_o$ **then**
        $\mathcal{Q}_1 \leftarrow$ put(k) ;
      else
        $\mathcal{Q}_2 \leftarrow$ put(t_k, k).
      end if
    end if
  end while
  Get the first task k from $\mathcal{Q}_1$;
  Get the location $o_k(t)$;
  Update the environment ;
  Get the filtered set of edge nodes $\mathbf{A}_{t+1}$;
end for

a 1D feature sequence $Z \in \mathbb{R}^{L \times |\mathbf{E}|}$, where $|\mathbf{E}|$ is the number of edge nodes, L is the length of a sequence. By vectoring each state feature into a latent $|\mathbf{E}|$-dimensional space, we obtain a sequence of input embedding z for state s_t at time slot t. To encode the state spacial information, we add position embedding p_i for each node to all state embedding z_i that form the final sequence input $Z = \{z_1 + p_1, z_2 + p_2, \cdots, z_{Len} + p_{Len}\}$.

The inputs are fed to an encoder to learn feature representations. The encoder is made of L network layers with alternating multi-headed self-attention and feed-forward blocks. Dropout, Layernorm, and residual connections are applied after each block [11]. The first network layer expands the dimension from D_{model} to $D_{mlp} = 4 \cdot D_{model}$ and applies the non-linearity. The second network layer reduces the dimension from D_{mlp} to D_{model}. At each network layer $\mathcal{L}$, the input to self-attention is in a triplet of $(query, key, value)$ computed from the input $Z^{\mathcal{L}-1} \in \mathbb{R}^{L \times |\mathbf{E}|}$ as:

$$query = Z^{\mathcal{L}-1}\mathbf{W}_Q, key = Z^{\mathcal{L}-1}\mathbf{W}_K, value = Z^{\mathcal{L}-1}\mathbf{W}_V, \tag{15}$$

where $\mathbf{W}_Q/\mathbf{W}_K/\mathbf{W}_V \in \mathbb{R}^{|\mathbf{E}| \times d}$ are the learnable parameters of three linear projection layers and d is the dimension of $(query, key, value)$. Self-Attention (SA) is then formulated as follows:

$$SA\left(Z^{\mathcal{L}-1}\right) = Z^{\mathcal{L}-1} + \text{softmax}\left(\frac{Z^{\mathcal{L}-1}\mathbf{W}_Q\left(Z\mathbf{W}_K\right)^\top}{\sqrt{\mathcal{M}}}\right)\left(Z^{\mathcal{L}-1}\mathbf{W}_V\right). \tag{16}$$

MSA is an extension with m independent SA operations and projects their concatenated outputs:

$$MSA\left(Z^{\mathcal{L}-1}\right) = \left[SA_1\left(Z^{\mathcal{L}-1}\right); S\ A_2\left(Z^{\mathcal{L}-1}\right); \cdots; SA_y\left(Z^{\mathcal{L}-1}\right)\right]\mathbf{W}_O, \tag{17}$$

where $\mathbf{W}_O \in \mathbb{R}^{\mathcal{M}d \times |\mathbf{E}|}$. d is typically set to $|\mathbf{E}|/\mathcal{M}$. An MLP block then transforms the output of MSA with residual skip as the layer output as:

$$Z^{\mathcal{L}} = MSA\left(Z^{\mathcal{L}-1}\right) + MLP\left(MSA\left(Z^{\mathcal{L}-1}\right)\right) \in \mathbb{R}^{L \times |\mathbf{E}|}. \tag{18}$$

We denote $\left\{Z^1, Z^2, \cdots, Z^{L_e}\right\}$ as the features of transformer layers. After that, a CNN and a linear layer are used to reduce the dimensionality of the encoder features.

Multi-user feature extraction: To extract the multi-user feature, we study an RNN structure called LSTM cells [12], that can learn the time interdependence between the columns of the state. In vehicular networks, task scheduling decisions are made sequentially, forming a sequence. Through LSTM, the decision sequence of the tasks in the past can be embedded into the current state, thus taking into account the influence of multiple users at different times. Moreover, LSTM can better capture state changes, including the mobility of multiple vehicle locations, layer distribution changes, etc., which cannot be obtained from a single state at the current time slot [40]. Specifically, the LSTM network introduces a new internal state $\boldsymbol{c}_t$ for linear recurrent information transmission, and at the same time, non-linearly outputs information to the external state of the hidden layer $\boldsymbol{h}_t$, which are calculated as follows:

$$\boldsymbol{c}_t = \boldsymbol{f}_t \odot \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \odot \tilde{\boldsymbol{c}}_t, \quad \boldsymbol{h}_t = \boldsymbol{o}_t \odot \tanh\left(\boldsymbol{c}_t\right), \tag{19}$$

where $\boldsymbol{f}_t$ is the forget gate, which receives the internal state and learns how much it should memorize or forget from the past. $\boldsymbol{i}_t$ is the input gate that aggregates the output of past steps and the current input and passes it through an activation function as done in a conventional RNN. $\boldsymbol{o}_t$ is the output gate that combines the current cell state and the output of the input gate and generates the LSTM output. $\odot$ represents element-wise multiplication. $\tilde{\boldsymbol{c}}_t$ is the candidate state obtained by the nonlinear function. The calculation process is as follows:

$$\begin{bmatrix} \tilde{\boldsymbol{c}}_t \\ \boldsymbol{o}_t \\ \boldsymbol{i}_t \\ \boldsymbol{f}_t \end{bmatrix} = \begin{bmatrix} \tanh \\ \sigma \\ \sigma \\ \sigma \end{bmatrix} \left(\boldsymbol{W}\begin{bmatrix} \boldsymbol{s}_t \\ \boldsymbol{h}_{t-1} \end{bmatrix} + \boldsymbol{b}\right), \tag{20}$$

where $\boldsymbol{s}_t$ is the layer sharing feature encoding as shown in Fig. 2. $\boldsymbol{W}$ and $\boldsymbol{b}$ are neural network weight matrices.

The LSTM blocks can map the state s_t to a vector with a fixed size. Every task in the sequence from the task queue

is fed to the LSTM iteratively. Then the internal state $\boldsymbol{c}_t$ and recurrent output $\boldsymbol{h}_t$ are concatenated into a vector, $\tilde{\boldsymbol{s}}_t$. Then, $\boldsymbol{c}_t$ and $\boldsymbol{h}_t$ are fed into the second LSTM as the initial internal state and recurrent inputs. Now, the input sequence to the second LSTM will be the sequence of the tasks. During the training of this model, the goal is to find optimal values for the weights and biases of the LSTMs. We use the same concept to learn a fixed-size representation of our state space. Through the above steps, the node and task information contained in state s_t form a sequence through $\boldsymbol{c}_t$ and $\boldsymbol{h}_t$. This enables the agent to fully consider the impact of historical tasks, i.e., multiple tasks when making decisions.

Training: The OCM algorithm is based on policy optimization. A policy is a rule used by the agent to decide what actions to take, which is usually denoted by π, i.e., $a(t) \sim \pi(\cdot|s(t))$. The probability of the scheduling process in vehicular network scenarios is defined as:

$$P(\tau|\pi) = \rho_0(s_0)\Pi_{t=0}^{T-1} P(s_{t+1}|s_t, a_t)\pi(a_t|s_t), \tag{21}$$

where $\rho_0(s_0)$ is the start-state distribution. The expected return denoted by $J(\pi)$ is obtained as:

$$J(\pi) = \int_{\tau} P(\tau|\pi)R(\tau) = \mathbb{E}_{\tau\sim\pi}[R(\tau)]. \tag{22}$$

The optimal policy problem aims to obtain the optimal policy π^*:

$$\pi^* = \arg\max_{\pi} J(\pi). \tag{23}$$

Policy gradient methods compute an estimator of the policy gradient and plug it into a stochastic gradient ascent algorithm, where the advantage function is crucially important [41]. The advantage function indicates the relative advantage of each action, which is denoted as:

$$A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s), \tag{24}$$

where the value function $V^{\pi}(s)$ gives the expected return if the agent starts in state s and acts according to policy π, which is defined as:

$$V^{\pi}(s) = \mathbb{E}_{\tau\sim\pi}[R(\tau)|s_0 = s]. \tag{25}$$

The most commonly used gradient estimator $\hat{g}$ has the form of:

$$\hat{g} = \hat{\mathbb{E}}_t \left[\nabla_\theta \log \pi_\theta(a_t|s_t)\hat{A}_t\right], \tag{26}$$

which is obtained by differentiating the loss function $L_{PG}(\theta)$ of policy gradient:

$$L_{PG}(\theta) = \hat{\mathbb{E}}_t \left[\log \pi_\theta(a_t|s_t)\hat{A}_t\right], \tag{27}$$

where π_θ is a stochastic policy and $\hat{A}_t$ is an estimator of the advantage function at time step t. However, it is appealing to perform multiple optimization steps on the loss $L_{PG}(\theta)$ using the same scheduling trace. Moreover, it often leads to destructively large policy updates. To solve these problems, the loss function can be customized as [42]:

$$L_{TRPO}(\theta) = \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}\hat{A}_t\right] = \hat{\mathbb{E}}_t \left[r_t(\theta)\hat{A}_t\right], \tag{28}$$

where $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$, and $r_t(\theta_{\text{old}}) = 1$. Besides, the estimated advantage $\hat{A}_t$ is calculated as:

$$\hat{A}_t = -V(s_t) + \gamma^{T-t}V(s_T) + \sum_{\tau=t}^{T-1}\gamma^{\tau-t}r_\tau. \tag{29}$$

Without a constraint, maximization of $L_{TRPO}(\theta)$ would lead to a huge policy update. Hence, the loss is further customized into [14]:

$$L(\theta) = \hat{\mathbb{E}}_t \left[\min\left(r_t(\theta)\hat{A}_t, \text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}_t\right)\right], \tag{30}$$

where ϵ is a hyperparameter, e.g., $\epsilon = 0.2$. Furthermore, clip $(r_t(\theta), 1-\epsilon, 1+\epsilon)$ is used to clip the probability, i.e., removes the incentive for moving r_t outside of the interval $[1-\epsilon, 1+\epsilon]$. Then, the minimum of the clipped and unclipped objectives is taken.

Algorithm 2 Training of the OCM Algorithm

Input : $\mathcal{Q}_1, \mathcal{Q}_2 = \emptyset, \pi_\theta, \mathbf{N}, \mathbf{D}$
Output: a_t
Initialize policy network π_θ;
Initialize replay memory $\mathbf{D} = \emptyset$;
for *episode* $\leftarrow 1, 2, \ldots$ **do**
  while $\mathcal{Q}_1 \neq \emptyset$ *or* $\mathcal{Q}_2 \neq \emptyset$ **do**
    Sample a set of trajectories τ on policy π_θ;
    Call Algorithm 1 and store $\{s_t, a_t, r_t\}$ in $\mathbf{D}$;
    Compute target value $\hat{v}_\pi(s_t)$;
  end while
  for t=$1, 2, \ldots, T$ **do**
    Compute $\hat{A}_1, \ldots, \hat{A}_T$ by Eq. (29) ;
    Compute $L(\theta)$ by Eq. (30);
    Optimize the network by mini-batch SGD with $Adam$;
  end for
  Update weights from time to time $\pi_{old} \leftarrow \pi_\theta$;
end for

The training of the OCM algorithm is shown in Algorithm 2. The policy network π_θ and replay memory $\mathbf{D}$ are initialized for each episode. Then, the Algorithm 1 is called to make the online container migration decisions. The results are stored, and the target value is calculated. After that, the estimated advantages $\hat{A}_t$ and the loss are calculated. Then, the network is optimized and updated.

C. Computational Complexity Analysis

The analysis of computational complexity is as follows. First, as shown in Algorithm 1, the state is obtained by Eq. (14). The numbers of containers and layers are $|\mathbf{C}|$ and $|\mathbf{L}|$, respectively. The complexity of Eq. (14) is $O(|\mathbf{C}||\mathbf{L}|)$. Then, the action is selected according to π_t and $\mathbf{A}_t$. The time complexity of the policy is only related to the network size, which can be considered a constant time O_t. So, the complexity of the action selection is $O(O_t)$. The complexity for the

while loop is $O(|\mathbf{K}|)$. These steps are executed sequentially so they can be completed in polynomial time.

Second, for the training of the OCM algorithm as shown in Algorithm 2, it mainly updates the network weights in Fig. 2. To evaluate the complexity of the network update, a theoretical analysis of the computational complexity of the policy and value networks based on floating point operations (FLOPs) is performed, which is widely used to measure the computational complexity of deep learning models [43], [44].

As shown in Fig. 2, we analyze the FLOPs in order from bottom to top. In the layer-sharing feature encoding part, the embedding layers are dictionary lookups with 0 FLOPs [45]. For the Transformer encoder layer, the FLOPs are $4L|\mathbf{E}|d + 2L^2d$, where L is the length of the input sequence, $|\mathbf{E}|$ is the number of edge nodes, and d is the dimension of $(query, key, value)$ [46]. For the convolutional kernels, the FLOPs is $2HW(C_{in}K^2+1)C_{out}$, where H, W, and C_{in} are height, width, and number of channels of the input feature map, respectively. K is the kernel width, and C_{out} is the number of output channels [47]. Denote the input and output dimensions of the j-th linear layer (from bottom to top) by D_j^i and D_j^o, respectively. The FLOPs of the two linear layers in the layer sharing feature encoding part are $2(D_1^i - 1)D_1^o$ and $2(D_2^i - 1)D_2^o$, respectively [47].

Then, for the multi-user feature extraction part, the FLOPs of the LSTM cell are $4D_h(D_i + D_h + 3)$, where D_i and D_h are the dimensions of the input and hidden layers, respectively [48]. Finally, for the online container migration decision part, the policy network and value network are composed of one linear layer, whose FLOPs are $2(D_3^i - 1)D_3^o$ and $2(D_4^i - 1)D_4^o$, respectively. The FLOPs of the activation functions can be ignored compared with matrix multiplies and inner products [49].

V. Evaluation

A. Experimental Settings

This subsection introduces the data preprocessing, parameter settings, baseline algorithms, and simulator setup.

Data preprocessing: The data-trace used in the experiments are from the mobility traces of taxi cabs in Rome, Italy [15]. The area with relatively high data density in the data set is selected, and the moving path is limited within $7km \times 7km$. To better compare the performance, different numbers of nodes and tasks are set. When comparing different numbers of nodes, the number of tasks is fixed at 500. Furthermore, the numbers of nodes are set to 16, 36, 64, 100, and 144, respectively [22]. When comparing the number of different tasks, the number of nodes is fixed at 36, the node interval is 1km, and the communication range between base stations is fixed at 1.2km. In addition, for heterogeneous nodes, the CPU frequency ranges within [64, 96] GHz, and the CPU clock speed of the task is (0, 1] GHz [23].

Parameter settings: The transmission power p_k is set to $0.25W$. The power spectral density of the Gaussian white noise σ is set to $-174dBm/Hz$. The path loss coefficient β is set to 6 [35]. The communication bandwidth $b_{k,e}(t)$ between the task k and edge node e is set to [70, 90] MB/s.

TABLE VI
Hyperparameter Settings

Type	Hyperparameter	Value
Encoder	Encoder layer dimension	512
	CNN kernel size	4
	CNN Stride	1
	CNN dense hidden dimension	256
Actor	Input dimension	256
	Output dimension	128
LSTM	Hidden dimension	128
Critic	Dimensions for the 1st layer	512 and 256
	Dimensions for the 2nd layer	256 and 128
Others	Learning rate	0.0003
	Optimizer	Adam
	Discount factor	0.99
	GAE parameter	0.95
	Clipping parameter	0.2
	Policy entropy coefficient	0.01
	Batch size	256

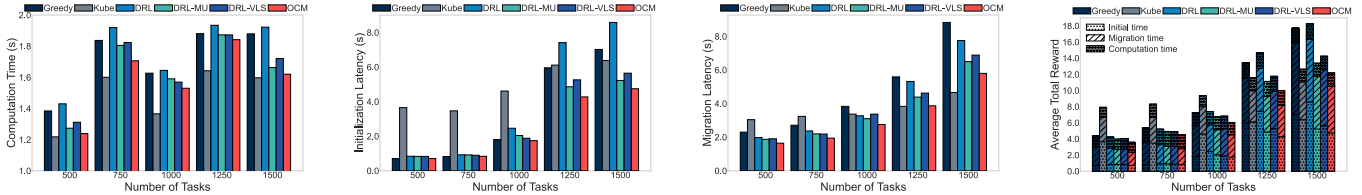
The distance between edge nodes is calculated similarly as [22]. The positive coefficient of migration latency is set to [1, 3]. The task sizes are set from 5kb to 5Mb. During a training epoch, tasks arrive randomly, and the image requested by each task ranges in size from 1.84MB to 2.03GB [16]. Among them, the size of a layer ranges from 7B to 880.58MB. Each node is initialized with a random number of layers assigned to it. For the input of the neural network, we use the min-max normalization method to scale the layer state into $[-1, 1]$ and the resource capacity into [0, 1]. The detailed hyperparameter settings of the deep neural network are shown in TABLE VI.

Baselines: To compare the performance, several baselines are conducted. The details are as follows.

1) Kube [17]. Kube is an image-based scheduling algorithm. It is one of the default algorithms of Kubernetes. The scheduling decision is made according to the score calculated by the distribution of requested images.
2) Greedy. A greedy algorithm selects the edge node with a minimal layer download size.
3) DRL. It is a traditional actor-critic-based DRL algorithm with several fully connected layers.
4) DRL-MU. It is a DRL algorithm with only the LSTM network to extract the Multi-User (DRL-MU) features.
5) DRL-VLS. It is a DRL algorithm with a self-attention-based encoder to extract the Vanilla Layer-Sharing (DRL-VLS) features.

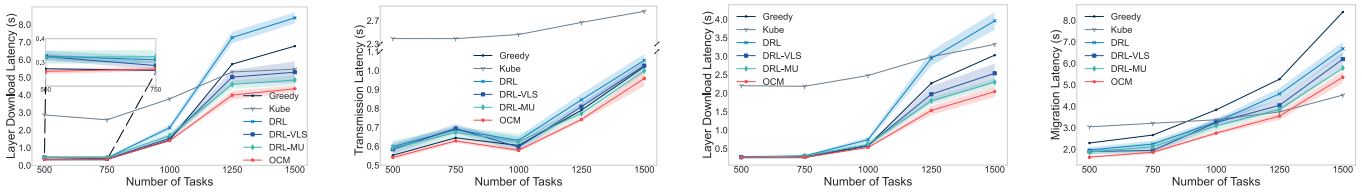
Simulator setup: The vehicular network simulation environment is implemented with Python, which mainly includes the classes of edge node, container, image, layer, task, scheduler, etc. The details are as follows.

1) Cloud. The remote cloud data center mainly includes two attributes: CPU and bandwidth.
2) Image. The image class includes the image ID and image name. Besides, the list of layers in the image, the number of layers, and the image size are also included.
3) Layer. The layer class includes layer ID, layer name, image list containing the layer, and layer size.



(a) Computation Time (b) Initial Time (c) Migration Time (d) Total Reward

Fig. 3. Performance with different numbers of tasks.



(a) Layer Download Time while Offloading Tasks (b) Transmission Time while Offloading Tasks (c) Layer Download Time while Migrating Tasks (d) Container Layer Migration Time while Migrating Tasks

Fig. 4. Performance with different numbers of tasks.

4) Node. Each node includes node ID, CPU capacity, memory capacity, bandwidth, storage capacity, geographic location coordinates, running container list, and existing layers on this node.
5) Task. Each task contains the task ID, arrival time, requested CPU, requested memory, requested bandwidth, requested image, and the layer list in the image.

The vehicular network environment is created based on these classes to return the reward, state, etc. Moreover, the environment is updated online according to the action the agent selects. In real deployment, we can deploy socket servers on the decision-making controller nodes and edge nodes and connect and transmit decision results, data, etc., through sockets. The vehicle and the RSU can be communicated through C-V2X, such as LTE-V2X or 5G-V2X [7]. More deployment details are discussed in the discussion.

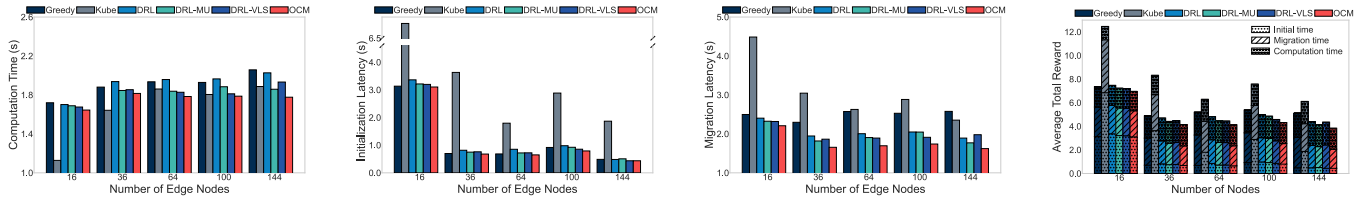
B. Experimental Results

To illustrate the performance of the proposed OCM algorithm, the experiments are conducted in a heterogeneous vehicular network scenario. Firstly, an example of the OCM algorithm is illustrated. Then, we conduct a detailed analysis of the experimental results. Each experimental result has been repeated 5 - 6 times to mitigate the influence of randomness. We draw the mean and variance of the experimental results in the figures. For example, the solid line in Fig. 4 is the mean, and the shaded area represents the standard variance of multiple round experiments. Besides, the rewards in the figures are absolute values to compare the total delay better.

Performance with the different numbers of tasks: Figs. 3 and 4 show the performance of each algorithm under the different numbers of tasks. Fig. 3 shows the computation time, initial time, migration time, and total reward performance of the algorithms. As shown in Fig. 3(a), the number of tasks is set to 500, 750, 1000, 1250, and 1500, respectively. As the number of tasks increases, the effect of the image-based Kube algorithm worsens because as the number of tasks increases, the required image also increases, and the Kube algorithm cannot reduce the download of unnecessary layers from the perspective of layers. Furthermore, the greedy algorithm is unstable under different task numbers because it cannot consider long-term benefits. In addition, as the number of tasks increases, the computation time of the OCM algorithm is consistently better than that of the DRL-MU and DRL-VLS algorithms.

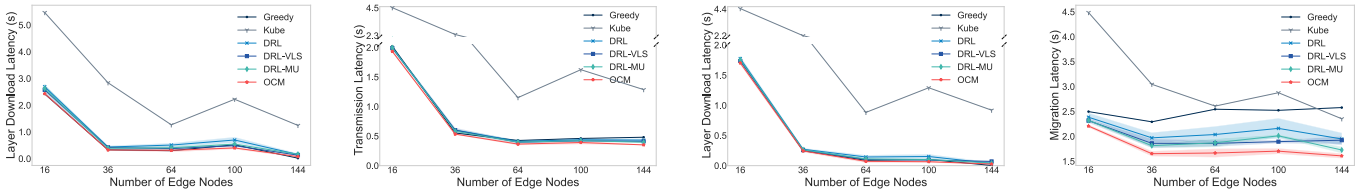
As shown in Fig. 3(b), the initial time of the OCM algorithm is always lower than that of other algorithms. Overall, the initial time order of these algorithms is OCM $<$ DRL-VLS $<$ DRL-MU $<$ Greedy $<$ DRL $<$ Kube. As described in the system model in Section III, the initial time includes the download time of layers and the transmission time of tasks, as shown in Figs. 4(a) and 4(b), respectively. As the number of tasks increases, the layer download time and transfer time of the OCM algorithm are consistently lower than other algorithms. Compared with the DRL-VLS algorithm, the OCM algorithm can consider the multi-user features, so tasks can be scheduled to the same node as much as possible to save downloads. Compared with the DRL-MU algorithm, OCM can fully consider sharing layers, effectively reducing the layer download overhead.

The migration times of different algorithms are shown in Fig. 3(c). As the number of tasks increases, the overall migration time relationship is OCM $<$ DRL-VLS $<$ DRL-MU $<$ DRL $<$ Kube $<$ Greedy. This is because the OCM algorithm can fully consider sharing layers, effectively reducing migration time. Figs. 4(c) and 4(d) show the layer download time and container layer migration time during container migration, respectively. It can be seen from the figure that the effect of the Greedy algorithm is worse than that of Kube because the Greedy algorithm only considers that the target node needs to download the least layers but does not consider the container transmission time caused by the distance, so it may choose to exist layer A more significant number of nodes with a



(a) Computation Time (b) Initial Time (c) Migration Time (d) Total Reward

Fig. 5. Performance with different numbers of edge nodes.



(a) Layer Download Time while Offloading Tasks (b) Transmission Time while Offloading Tasks (c) Layer Download Time while Migrating Tasks (d) Container Layer Migration Time while Migrating Tasks

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

longer distance leads to a longer overall migration time of the algorithm.

Finally, the overall reward function under different task numbers is shown in Fig. 3(d). The rewards in the figure are absolute values. It can be seen from the figure that the OCM algorithm can reduce up to 53% of the total latency against the baseline algorithms. Specifically, the total latency with the different numbers of tasks is reduced by 30%, 21%, 21%, 8%, and 10% on average compared with Kube, Greedy, DRL, DRL-VLS, and DRL-MU algorithms, respectively. DRL-VLS and DRL-MU algorithms only consider vanilla layer sharing or multi-user features. As the number of tasks increases, many popular layers have been downloaded, so the number of layer downloads required by popular nodes can be reduced. In other words, our proposed OCM algorithm can adapt well to different numbers of tasks and maintain better performance than the baselines.

Performance with the different numbers of edge nodes: Figs. 5 and 6 show the performance of different algorithms in the case of different numbers of nodes. As shown in Fig. 5(a), as the number of edge nodes increases, the computing resources are more sufficient, so the overall computing time becomes less and less. The initialization time of different algorithms is shown in Fig. 5(b). The initialization time mainly includes layer download and transmission time, as shown in Figs. 6(a) and 6(b), respectively. Fig. 6(a) shows that the OCM algorithm can fully consider the sharing of layers, thus effectively reducing the layer download time. From Fig. 6(b), we can see that the transmission time decreases as the number of nodes increases. This is because the distance between nodes becomes shorter as the number of nodes increases, so the required transmission time is further reduced.

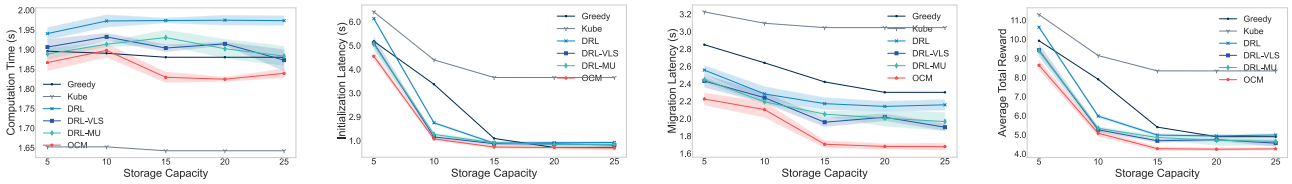
As shown in Fig. 5(c), when the number of nodes changes, the migration time of containers does not change. This is because the user's movement track does not change, and the user's task volume also does not change. Overall, in terms of container migration time, OCM $<$ DRL-VLS $<$ DRL-MU $<$ DRL $<$ Greedy $<$ Kube. The container migration time mainly includes the time to download the layer from the cloud and the time to migrate the container layer, as shown in Figs. 6(c) and 6(d), respectively. As the number of nodes increases, the number and types of layers existing on each node are also more, so the layer download time decreases, as shown in Fig. 6(c). In addition, with the number of nodes, the distance between nodes becomes closer, so the migration time is shortened, as shown in Fig. 6(d).

As shown in Fig. 5(d), different numbers of nodes have little effect on reward, which fully demonstrates the robustness of the algorithm. Besides, this also illustrates that our algorithm can adapt to different vehicular networks, e.g., different numbers of nodes. In terms of overall reward, the OCM algorithm reduces the total latency than Greedy, Kube, DRL, DRL-MU, and DRL-VLS algorithms by 23%, 55%, 14%, 10%, and 10% on average, respectively.

Performance with different storage capacities: As shown in Fig. 7, to compare the performance under different edge node capacities, the capacity of edge nodes is set from 5 to 25, respectively. As the capacity of the node increases, the initialization time and migration time of different algorithms decrease. This is because the node can store more layers when the capacity is more extensive, reducing the layers that need to be downloaded during initialization and migration. Our OCM algorithm can maintain the best performance when the node capacity changes.

Performance with different running container numbers: To compare the impact of different computing capabilities of nodes, the maximum number of running containers of the nodes is set from 5 to 25, and the experimental results are shown in Fig. 8. It can be seen from the figure that as the computing capability of the nodes increases, the total task running time decreases. This is because each node can run more containers, resulting in an effective reduction in computation time.

Performance with different bandwidth: Since the bandwidth affects the download speed, affecting the total latency, Fig. 9 shows the performance under different bandwidths.



(a) Computation Time (b) Initial Time (c) Migration Time (d) Total Reward

Fig. 7. Performance with different storage capacities.

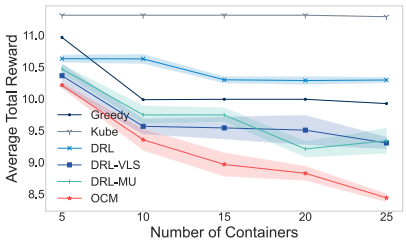


Fig. 8. Performance with different numbers of containers.

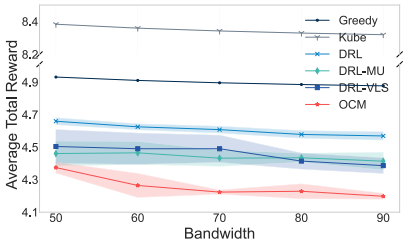


Fig. 9. Performance with different bandwidth.

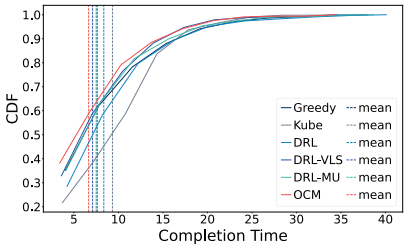


Fig. 10. CDF of total latency.

As the bandwidth increases, the time required to download the layer decreases, so the overall reward decreases. No matter how the bandwidth changes, the performance of the OCM algorithm is better than other algorithms.

CDF of total latency: It can be seen from Fig. 10 that the proportion of tasks with a shorter total latency of the OCM algorithm is more than baseline algorithms. Moreover, the average total latency of the OCM algorithm is shorter than that of baseline algorithms. This proves that the OCM algorithm can optimize container migration decisions from a long-term perspective.

Performance of the OCM algorithm: In order to illustrate the convergence of the algorithm, the policy loss, critic loss, and average total reward of the OCM algorithm are shown in Figure 8. As shown in Figs. 8(a) and 8(b), the loss of the policy network and the critic network reached convergence after about 200 epochs, showing the OCM algorithm's convergence. The reward of the first 1000 epochs is shown in Figure 8(c). It can be seen from the figure that the reward of the OCM algorithm also converges to a higher value quickly. Since the Kube and Greedy algorithms adopt a fixed strategy, the reward of each epoch is the same. Therefore, after multiple rounds of epoch training, the reward of the OCM algorithm is better than other algorithms.

TABLE VII
COMPUTATION RESOURCES FOR DIFFERENT ALGORITHMS

Algorithm	RAM	VRAM	Execution Time
Greedy	-	-	0.26ms
Kube	-	-	0.17ms
DRL	554Kb	559Kb	15ms
DRL_MU	562Kb	564Kb	14ms
DRL_VLS	562Kb	1653Kb	15ms
OCM	562Kb	1653Kb	14ms

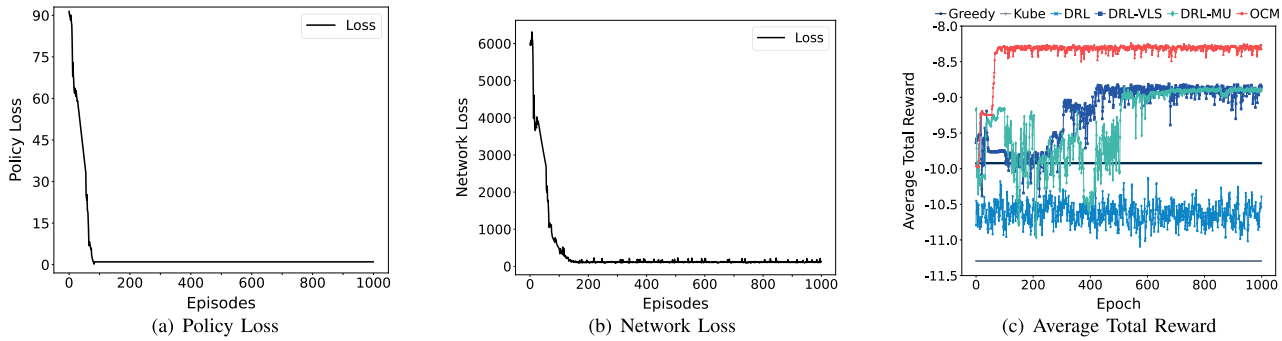
Performance for the migration decisions: In order to further illustrate the effect of the algorithm, the average migration frequency is shown in Figure 9. It can be seen from the figure that the average migration frequency of the OCM algorithm is the lowest among all algorithms. This fully demonstrates that the OCM algorithm can effectively reduce the number of migrations while maintaining a high reward, which further illustrates the effect of the algorithm. On the other hand, the migration frequency of the Kube algorithm is the highest. This is because the Kube algorithm only judges based on the image and cannot take into account the distribution of the layer, so it will bring a lot of wasted migration times.

Computation resources for different algorithms: As shown in TABLE VII, we use *torch.profiler* [45] to record the Random Access Memory (RAM), Video RAM (VRAM), and execution time for different algorithms. Greedy and Kube algorithms require the shortest execution time because they are simple judgments and comparisons. The computation resources and execution time required by our proposed OCM algorithm and the RL-based baseline algorithms are close, indicating that our improvements do not bring additional execution overhead. Moreover, the computation resources and execution time required by the algorithms are within acceptable limits, demonstrating that our algorithm has low complexity and can be applied in reality.

VI. DISCUSSION

From the experimental results, we can see the effectiveness of our algorithms. However, the following issues deserve further investigation.

Migration or routing: When a task is processed on the origin edge node through routing, while it could potentially reduce migration latency, it could lead to other issues:



(a) Policy Loss (b) Network Loss (c) Average Total Reward

Fig. 11. Policy loss, network loss, and average total reward of the OCM algorithm.

1) Resource allocation: If the task is resource-intensive, it might overburden the origin edge node, leading to performance degradation for other tasks.
2) Load balancing: Keeping the task on the origin node may result in an unbalanced load among the edge nodes. Migration can help balance the load.
3) Latency: If the users are closer to the target edge node, it might be faster to migrate the task and process it there despite the migration latency.
4) Energy efficiency: Different edge nodes may have varying energy efficiency levels. If the target node can perform the task more energy-efficiently, it might make sense to migrate the task there.
5) Reliability and robustness: If the origin edge node already operates at high capacity, migrating the task to a more reliable or less utilized node can reduce the risk.

In summary, although avoiding migration latency is an important issue, it is just one of the important factors we should consider. The overall performance, efficiency, and reliability of the edge computing system often require the capability to migrate tasks between nodes as conditions change.

Ping-pong phenomenon: The ping-pong phenomenon, i.e., rapid and unnecessary migrations between two adjacent RSUs due to small fluctuations in received signal strength, can introduce high cost and degrade performance. Several strategies can be implemented to avoid this: 1) Hysteresis threshold: It sets a threshold that must be exceeded to trigger a migration. 2) Dwell timer: It is a time delay before a migration is performed. 3) Predictive algorithms: If the algorithm predicts that the user will return to the original RSU quickly, it might decide to delay or avoid migration. 4) Cost-benefit analysis: It considers migration costs against the benefits.

In this paper, we avoid the ping-pong phenomenon through the following aspects:

1) System model: We have set a hysteresis threshold th_o as described in Section III-B. Moreover, the cost-benefit analysis is conducted in Eq. (10).
2) Algorithms: Our OCM algorithm has considered the global state at the current time and the continuous rewards at the previous and current time slots. Frequent migration will cause the long-term cumulative rewards to decrease, so the RL-based algorithm we designed can naturally prevent the ping-pong phenomenon.

It can be seen from Fig. 12 that the migration frequency of our OCM algorithm is the smallest, which further illustrates

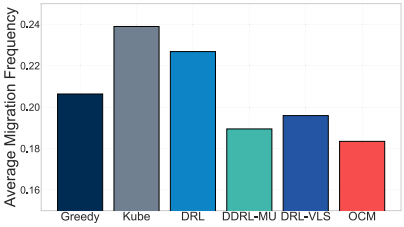


Fig. 12. Average migration frequency.

that our algorithm can effectively avoid the ping-pong phenomenon. Future work will further consider dwell timers and predictive algorithms.

Implementation method: The OCM algorithm can be implemented on RSUs through the Kubernetes scheduling framework [50]. The status of the vehicles can be sent to the RSUs via C-V2X [7]. Prometheus collects and stores the real-time resource load data, subsequently accessed via the Kubernetes API [51]. The Kubernetes scheduling framework allows for complete scheduling cycle customization, entailing two steps: the Scheduling Cycle and the Binding Cycle. The former selects an edge node for the container, while the latter applies that decision to the RSUs. These two cycles collectively form a "scheduling context" featuring multiple extension points, including Filter, Score, Reserve, Bind, etc. A scheduler plugin must be implemented to enact a custom scheduling algorithm, with several extension points registered [50]. Once the scheduler plugin is completed, it can be deployed through Kubernetes, with the application scheduler specified as our custom scheduler.

Nevertheless, this comprehensive deployment process necessitates a substantial amount of engineering code. Additionally, using RL demands interaction between the training and decision-making processes, implicating RL and the Kubernetes API, which makes system implementation more complex. As this paper primarily focuses on the scheduling algorithm, the performance of the algorithm has been evaluated through large-scale simulations rather than implementation in Kubernetes. Currently, our team is working on deploying a custom scheduler into the Kubernetes system. This endeavor has surfaced many new challenges, guiding our future work.

VII. Conclusion

We proposed a multi-user layer-aware online container migration algorithm in edge-assisted vehicular networks.

Firstly, we modeled the OCM problem comprehensively, considering the initialization latency, computation latency, and migration latency. Then, a feature extraction method based on self-attention and LSTM was proposed to extract the multi-user layer-sharing information. Finally, an OCM algorithm based on policy gradient RL was proposed for migration decisions. The experiments were conducted with real-world data traces, and the experimental results demonstrated that our proposed algorithm can outperform the baseline algorithms up to 53% in total latency. Future work will consider the joint optimization problem of layer-based container placement and migration and implement it in the Kubernetes system.

REFERENCES

[1] F. Yang, S. Wang, J. Li, Z. Liu, and Q. Sun, "An overview of Internet of Vehicles," *China Commun.*, vol. 11, no. 10, pp. 1–15, Oct. 2014.

[2] P. Sun and A. Boukerche, "Modeling and analysis of coverage degree and target detection for autonomous underwater vehicle-based system," *IEEE Trans. Veh. Technol.*, vol. 67, no. 10, pp. 9959–9971, Oct. 2018.

[3] S. Wan, J. Lu, P. Fan, and K. B. Letaief, "Toward big data processing in IoT: Path planning and resource management of UAV base stations in mobile-edge computing system," *IEEE Internet Things J.*, vol. 7, no. 7, pp. 5995–6009, Jul. 2020.

[4] Y. Li, Y. Wu, M. Dai, B. Lin, W. Jia, and X. Shen, "Hybrid NOMA-FDMA assisted dual computation offloading: A latency minimization approach," *IEEE Trans. Netw. Sci. Eng.*, vol. 9, no. 5, pp. 3345–3360, Sep. 2022.

[5] Z. Wei, B. Li, R. Zhang, X. Cheng, and L. Yang, "Many-to-many task offloading in vehicular fog computing: A multi-agent deep reinforcement learning approach," *IEEE Trans. Mobile Comput.*, early access, Feb. 28, 2023, doi: 10.1109/TMC.2023.3250495.

[6] Z. Ning, J. Huang, X. Wang, J. J. P. C. Rodrigues, and L. Guo, "Mobile edge computing-enabled Internet of Vehicles: Toward energy-efficient scheduling," *IEEE Netw.*, vol. 33, no. 5, pp. 198–205, Sep. 2019.

[7] S. Chen, J. Hu, Y. Shi, L. Zhao, and W. Li, "A vision of C-V2X: Technologies, field testing, and challenges with Chinese development," *IEEE Internet Things J.*, vol. 7, no. 5, pp. 3872–3881, May 2020.

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

[9] X. Chen et al., "Age of information aware radio resource management in vehicular networks: A proactive deep reinforcement learning perspective," *IEEE Trans. Wireless Commun.*, vol. 19, no. 4, pp. 2268–2281, Apr. 2020.

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

[11] A. Vaswani et al., "Attention is all you need," *Adv. neural Inf. Process. Syst.*, vol. 30, 2017, pp. 1–11.

[12] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997.

[13] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: MIT Press, 2018.

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

[15] L. Bracciale, M. Bonola, P. Loreti, G. Bianchi, R. Amici, and A. Rabuffi. (Jul. 17, 2014). *CRAWDAD Dataset Roma/Taxi*. [Online]. Available: https://crawdad.org/roma/taxi/20140717

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

[17] *Kubernetes*. Accessed: Sep. 10, 2023. [Online]. Available: https://kubernetes.io

[18] B. Shi, H. Shen, B. Dong, and Q. Zheng, "Memory/disk operation aware lightweight VM live migration," *IEEE/ACM Trans. Netw.*, vol. 30, no. 4, pp. 1895–1910, Aug. 2022.

[19] K. Fu, W. Zhang, Q. Chen, D. Zeng, and M. Guo, "Adaptive resource efficient microservice deployment in cloud-edge continuum," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 8, pp. 1825–1840, Aug. 2022.

[20] T. Benjaponpitak, M. Karakate, and K. Sripanidkulchai, "Enabling live migration of containerized applications across clouds," in *Proc. IEEE INFOCOM Conf. Comput. Commun.*, Jul. 2020, pp. 2529–2538.

[21] Z. Tang, X. Zhou, F. Zhang, W. Jia, and W. Zhao, "Migration modeling and learning algorithms for containers in fog computing," *IEEE Trans. Services Comput.*, vol. 12, no. 5, pp. 712–725, Sep. 2019.

[22] S. Wang, R. Urgaonkar, M. Zafer, T. He, K. Chan, and K. K. Leung, "Dynamic service migration in mobile edge computing based on Markov decision process," *IEEE/ACM Trans. Netw.*, vol. 27, no. 3, pp. 1272–1288, Jun. 2019.

[23] J. Wang, J. Hu, G. Min, Q. Ni, and T. El-Ghazawi, "Online service migration in mobile edge with incomplete system information: A deep recurrent actor-critic learning approach," *IEEE Trans. Mobile Comput.*, vol. 22, no. 11, pp. 6663–6675, Nov. 2023, doi: 10.1109/TMC.2022.3197706.

[24] T. Ouyang, Z. Zhou, and X. Chen, "Follow me at the edge: Mobility-aware dynamic service placement for mobile edge computing," *IEEE J. Sel. Areas Commun.*, vol. 36, no. 10, pp. 2333–2345, Oct. 2018.

[25] Y. Ma, W. Liang, J. Li, X. Jia, and S. Guo, "Mobility-aware and delay-sensitive service provisioning in mobile edge-cloud networks," *IEEE Trans. Mobile Comput.*, vol. 21, no. 1, pp. 196–210, Jan. 2022.

[26] Q. Zhang, F. Liu, and C. Zeng, "Online adaptive interference-aware VNF deployment and migration for 5G network slice," *IEEE/ACM Trans. Netw.*, vol. 29, no. 5, pp. 2115–2128, Oct. 2021.

[27] Z. Han, H. Tan, R. Wang, G. Chen, Y. Li, and F. C. M. Lau, "Energy-efficient dynamic virtual machine management in data centers," *IEEE/ACM Trans. Netw.*, vol. 27, no. 1, pp. 344–360, Feb. 2019.

[28] C. Rong, J. H. Wang, J. Liu, T. Yu, and J. Wang, "Exploring the layered structure of containers for design of video analytics application migration," in *Proc. IEEE Wireless Commun. Netw. Conf. (WCNC)*, Apr. 2022, pp. 842–847.

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

[30] L. Gu, D. Zeng, J. Hu, B. Li, and H. Jin, "Layer aware microservice placement and request scheduling at the edge," in *Proc. IEEE INFOCOM Conf. Comput. Commun.*, May 2021, pp. 1–9.

[31] 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. 2023, doi: 10.1109/TSC.2022.3159728.

[32] M. Dolati, S. H. Rastegar, A. Khonsari, and M. Ghaderi, "Layer-aware containerized service orchestration in edge networks," *IEEE Trans. Netw. Service Manag.*, vol. 20, no. 2, pp. 1830–1846, Jun. 2023, doi: 10.1109/TNSM.2022.3217134.

[33] Y. Liu, B. Yang, Y. Wu, C. Chen, and X. Guan, "How to share: Balancing layer and chain sharing in industrial microservice deployment," *IEEE Trans. Services Comput.*, vol. 16, no. 4, pp. 2685–2698, Jul./Aug. 2023, doi: 10.1109/TSC.2022.3230699.

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

[35] Y. Wang, X. Tao, X. Zhang, P. Zhang, and Y. T. Hou, "Cooperative task offloading in three-tier mobile computing networks: An ADMM framework," *IEEE Trans. Veh. Technol.*, vol. 68, no. 3, pp. 2763–2776, Mar. 2019.

[36] Y. Wu, K. Ni, C. Zhang, L. P. Qian, and D. H. K. Tsang, "NOMA-assisted multi-access mobile edge computing: A joint optimization of computation offloading and time allocation," *IEEE Trans. Veh. Technol.*, vol. 67, no. 12, pp. 12244–12258, Dec. 2018.

[37] R. Li, Z. Zhou, X. Zhang, and X. Chen, "Joint application placement and request routing optimization for dynamic edge computing service management," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 12, pp. 4581–4596, Dec. 2022.

[38] Z. Han, H. Tan, G. Chen, R. Wang, Y. Chen, and F. C. M. Lau, "Dynamic virtual machine management via approximate Markov decision process," in *Proc. IEEE INFOCOM 35th Annu. IEEE Int. Conf. Comput. Commun.*, Apr. 2016, pp. 1–9.

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

[40] M. Hausknecht and P. Stone, "Deep recurrent Q-learning for partially observable MDPs," in *Proc. AAAI Fall Symp. Ser.*, 2015, pp. 1–9.

[41] R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Proc. 12th Int. Conf. Neural Inf. Process. Syst. (NIPS)*, 1999, pp. 1057–1063.

[42] J. Schulman, S. Levine, P. Moritz, M. Jordan, and P. Abbeel, "Trust region policy optimization," in *Proc. 32nd Int. Conf. Int. Conf. Mach. Learn. (ICML)*, 2015, pp. 1889–1897.

[43] Q. Liu, T. Xia, L. Cheng, M. van Eijk, T. Ozcelebi, and Y. Mao, "Deep reinforcement learning for load-balancing aware network control in IoT edge systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 6, pp. 1491–1502, Jun. 2022.

[44] J. Wang, J. Hu, G. Min, A. Y. Zomaya, and N. Georgalas, "Fast adaptive task offloading in edge computing based on meta reinforcement learning," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 1, pp. 242–253, Jan. 2021.

[45] *PyTorch Documentation*. Accessed: Sep. 10, 2023. [Online]. Available: https://pytorch.org/docs/

[46] K. Han, A. Xiao, E. Wu, J. Guo, C. Xu, and Y. Wang, "Transformer in transformer," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 34, 2021, pp. 15908–15919.

[47] P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz, "Pruning convolutional neural networks for resource efficient inference," in *Proc. 5th Int. Conf. Learn. Represent.*, 2019, pp. 1–17.

[48] A. Shahkarami, M. I. Yousefi, and Y. Jaouen, "Efficient deep learning of Kerr nonlinearity in fiber-optic channels using a convolutional recurrent neural network," in *Deep Learning Applications, Volume 4*. Berlin, Germany: Springer, 2022, pp. 317–338.

[49] V. Nair and G. E. Hinton, "Rectified linear units improve restricted Boltzmann machines," in *Proc. 27th Int. Conf. Mach. Learn. (ICML)*, 2010, pp. 807–814.

[50] *Scheduling Framework*. Accessed: Feb. 15, 2023. [Online]. Available: https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/

[51] J. Turnbull, *Monitoring With Prometheus*. James Turnbull, 2018. [Online]. Available: https://books.google.com/books?id=EtlfDwAAQBAJ

Zhiqing Tang (Member, IEEE) received the B.S. degree from the School of Communication and Information Engineering, University of Electronic Science and Technology of China, China, in 2015, and the Ph.D. degree from the Department of Computer Science and Engineering, Shanghai Jiao Tong University, China, in 2022. He is currently an Assistant Professor with the Advanced Institute of Natural Sciences, Beijing Normal University, China. His current research interests include edge computing, resource scheduling, and reinforcement learning.

Fangyi Mou received the B.S. degree from the Department of Electronic, Communication and Physics, Shandong University of Science and Technology, China, in 2017, and the M.Sc. degree from the Department of Computer Science and Engineering, University of Macau, China, in 2020. She is currently a Research Assistant with the Institute of Artificial Intelligence and Networks, Beijing Normal University, China. Her current research interests include mobile edge computing, resource allocation, and reinforcement learning.

Jiong Lou received the B.S. and Ph.D. degrees from the Department of Computer Science and Engineering, Shanghai Jiao Tong University, China, in 2016 and 2023, respectively. Since 2023, he has been a Research Assistant Professor with the Department of Computer Science and Engineering, Shanghai Jiao Tong University. He has published more than ten papers in leading journals and conferences (e.g., IEEE TRANSACTIONS ON MOBILE COMPUTING, IEEE TRANSACTIONS ON SERVICES COMPUTING, and *Computer Networks*). His current research interests include edge computing, task scheduling, and container management. He has served as a reviewer for *Computer Networks*, *Journal of Parallel and Distributed Computing*, IEEE INTERNET OF THINGS JOURNAL, and ICDCS.

Weijia Jia (Fellow, IEEE) received the B.Sc. and M.Sc. degrees in computer science from Center South University, China, in 1982 and 1984, respectively, and the M.A.S. and Ph.D. degrees in computer science from the Polytechnic Faculty of Mons, Belgium, in 1992 and 1993, respectively. He was the Chair Professor and the Deputy Director of the State Kay Laboratory of Internet of Things for Smart City, University of Macau. From 1993 to 1995, he joined the German National Research Center for Information Science (GMD), Bonn (St. Augustine), as a Research Fellow. From 1995 to 2013, he was with the City University of Hong Kong as a Professor. He is currently a Chair Professor and the Director of the BNU-UIC Institute of Artificial Intelligence and Future Networks, Beijing Normal University (Zhuhai), the VP for Research of the BNU-HKBU United International College (UIC), and has been the Zhiyuan Chair Professor of Shanghai Jiao Tong University, China. His contributions have been recognized as optimal network routing and deployment, anycast and QoS routing, sensors networking, AI (knowledge relation extractions; NLP), and edge computing. He has more than 600 publications in prestigious international journals/conferences, research books, and book chapters. He has received the best product awards from the International Science and Tech. Expo (Shenzhen) in 2011 and 2012, respectively, and the First Prize of Scientific Research Awards from the Ministry of Education of China in 2017 (List 2). He has served as an area editor for various prestige international journals, the chair, and a PC member/keynote speaker for many top international conferences. He is a Distinguished Member of CCF.

Yuan Wu (Senior Member, IEEE) received the Ph.D. degree in electronic and computer engineering from The Hong Kong University of Science and Technology in 2010. From 2016 to 2017, he was a Visiting Scholar with the Department of Electrical and Computer Engineering, University of Waterloo. His current research interests include resource management for wireless networks, green communications and computing, mobile edge computing, and edge intelligence. He was a recipient of the Best Paper Award from the IEEE International Conference on Communications in 2016 and the Best Paper Award from the IEEE Technical Committee on Green Communications and Computing in 2017. He is currently on the editorial boards of IEEE TRANSACTIONS ON WIRELESS COMMUNICATIONS, IEEE TRANSACTIONS ON VEHICULAR TECHNOLOGY, IEEE TRANSACTIONS ON NETWORK SCIENCE AND ENGINEERING, and IEEE INTERNET OF THINGS JOURNAL.

Wei Zhao (Fellow, IEEE) received the degree in physics from Shaanxi Normal University, China, in 1977, and the M.Sc. and Ph.D. degrees in computer and information sciences from the University of Massachusetts at Amherst in 1983 and 1986, respectively. He has served in important leadership roles in academics, including the Chief Research Officer with the American University of Sharjah; the Chair of the Academic Council; CAS Shenzhen Institute of Advanced Technology; the eighth rector of the University of Macau; the Dean of Science with the Rensselaer Polytechnic Institute; the Director for the Division of Computer and Network Systems, U.S. National Science Foundation; and the Senior Associate Vice President for Research at Texas A&M University. He has made significant contributions to cyber-physical systems, distributed computing, real-time systems, and computer networks. He led the effort to define the research agenda and to create the very first funding program for cyber-physical systems in 2006. His research results have been adopted in the standard of survivable adaptable fiber optic embedded networks. He was awarded the Lifelong Achievement Award by the Chinese Association of Science and Technology in 2005.