

Dual Dependency-Aware Collaborative Service Caching and Task Offloading in Vehicular Edge Computing

Liang Zhao¹, Member, IEEE, Lu Sun¹, Ammar Hawbani², Zhi Liu³, Senior Member, IEEE, Xiongyan Tang⁴, Senior Member, IEEE, and Lexi Xu⁵, Senior Member, IEEE

Abstract—Although some studies in recent years have focused on the coexistence of service and task dependencies in the collaborative optimization of service caching and task offloading in Vehicle Edge Computing (VEC), the challenges brought by dual dependencies have not been fully addressed. Therefore, this paper proposes a more comprehensive joint optimization method for service caching and task offloading under dual dependencies. First, this paper proposes a service criticality prediction method based on the Gated Graph Recurrent Network (GGRN) to perceive complex task dependencies and accurately capture the service requirements of critical task types. Based on this, a hierarchical active-passive hybrid caching strategy is designed, which aims to satisfy diverse service demands while reducing the additional overhead caused by remote service requests. Second, a global task priority computation method based on application heterogeneity has been developed to prevent cascading delays in task chains. Finally, this paper formulates a joint optimization problem for service caching and task offloading in a three-layer VEC system, models it as a markov decision process, and applies a proximal policy optimization-driven collaborative optimization algorithm named COHCTO. Simulation results show that COHCTO achieves multi-objective optimization across metrics such as delay, energy consumption, caching hit rate, and application success rate under conditions different from those of other algorithms.

Index Terms—Task dependency, service caching, task offloading, deep reinforcement learning.

Received 10 January 2025; revised 9 April 2025; accepted 21 May 2025. Date of publication 23 May 2025; date of current version 3 September 2025. This work was supported in part by the National Natural Science Foundation of China under Grant 62372310 and Grant W2421018, in part by the Liaoning Province Applied Basic Research Program under Grant 2023JH2/101300194, in part by the LiaoNing Revitalization Talents Program under Grant XLYC2203151, and in part by the Royal Society-International Exchanges 2021 Cost Share (NSFC) under Grant IEC\NSFC\211034. LLM was used to proofread this paper. Recommended for acceptance by Dr. S. Wang. (Corresponding authors: Lu Sun; Ammar Hawbani.)

Liang Zhao, Lu Sun, and Ammar Hawbani are with the School of Computer Science, Shenyang Aerospace University, Shenyang 110136, China (e-mail: lzhao@sau.edu.cn; sunlu3@stu.sau.edu.cn; ammande@ustc.edu.cn).

Zhi Liu is with the Department of Computer and Network Engineering, University of Electro Communications, Tokyo 182-8585, Japan (e-mail: liu@ieee.org).

Xiongyan Tang and Lexi Xu are with the Research Institute, China United Network Communications Corporation, Beijing 100048, China (e-mail: tangxy@chinaunicom.cn; davidlexi@hotmail.com).

This article has supplementary downloadable material available at <https://doi.org/10.1109/TMC.2025.3573379>, provided by the authors.

Digital Object Identifier 10.1109/TMC.2025.3573379

I. INTRODUCTION

WITH the rapid development of wireless communication technology and intelligent transportation systems [1], various computationally intensive and delay-sensitive vehicle applications have emerged, placing higher demands on the computing power of vehicles [2]. However, given the limitations of vehicle resources [3], it is quite challenging to meet such computing power requirements. Vehicle Edge Computing (VEC) is an effective way to overcome these challenges, among which service caching and task offloading are two key research areas.

Vehicles will generate task requests during driving, and services refer to the functional modules required to perform these tasks. For example, when performing obstacle detection tasks, it is necessary to deploy target recognition models on computing nodes. Service caching refers to pre-storing these services on edge nodes, such as Road Side Units (RSUs), while task offloading is to assign tasks to these edge nodes or the cloud for processing. The emergence of intelligent connected vehicles has resulted in a significant increase in data traffic and computational demands for in-vehicle applications. The growing diversity of these applications imposes increasingly stringent requirements on network latency and reliability. In this context, existing approaches that focus solely on optimizing either service caching or task offloading exhibit inherent limitations. Optimizing service caching alone may not meet the dynamic needs of task offloading, resulting in inflexible caching strategies. Similarly, optimizing task offloading alone may lead to service overload or resource waste [4]. Therefore, coordinated optimization of service caching and task offloading has become a key issue to improve system efficiency in VEC environments [5].

Several works have focused on service caching and task offloading, generally considering either task dependency or service dependency as the key factor [6]. Service dependency refers to the requirement for specific services to support the execution of certain tasks, meaning that tasks can only be offloaded to computational nodes with the corresponding services available. Task dependencies are interrelationships between tasks, where the execution of one task may depend on the results of other tasks. Such dependencies can be linear and sequential or more complex and topological structured relationships, such as Directed Acyclic Graphs (DAGs), where multiple dependency paths and mutual constraints may exist between tasks [7]. Due to the complexity

of implementation, existing works rarely consider service and task dependencies simultaneously. However, in practical VEC scenarios, service and task dependencies often coexist [8]. Therefore, designing a collaborative optimization strategy for service caching and task offloading in dual-dependency scenarios is of great practical significance and is crucial to ensuring system stability and efficiency.

In scenarios considering service dependencies, existing task offloading studies often assume that RSUs or the cloud processes all tasks, provided these nodes have cached the required services. If RSUs don't have the corresponding service cached, the task must request the service from the cloud or be directly processed by the cloud, which is not ideal. On the one hand, data transmission may cause significant delays under poor network conditions [9], [10]. On the other hand, if the RSU does not cache the required service in advance, loading and initializing the service from the cloud will further increase the delay [11]. Therefore, allowing some tasks to be executed locally on vehicles when the RSU does not cache the required service may be more efficient [12]. With the diversification of VEC scenarios, user-generated customized services at the edge have become common [11], [12], which is not only an essential condition for enabling local computation but also a key means to reduce reliance on the cloud, thereby improving system flexibility and responsiveness.

In scenarios considering task dependencies, service caching faces several challenges. First, some studies rely solely on RSUs for caching [13], neglecting that local mobile vehicles also have certain storage capabilities. This can lead to unnecessary communication delays, especially when the required service is unavailable at a nearby RSU, potentially requiring service transmission across multiple RSUs or even requesting the service from the cloud [11]. Second, existing studies generally assume that frequent services are more worthy of being cached [14], [15], [16]. However, this strategy has limitations in task dependency scenarios. For instance, an application may include traffic data aggregation, high-definition map provisioning, route optimization, and traffic signal coordination tasks. Among these, traffic data aggregation is a critical task. Suppose the service required for traffic data aggregation is not cached correctly. In that case, the subsequent tasks may experience interruptions due to the high delay in retrieving data from remote cloud services, thereby diminishing system performance. In addition, high-definition map provisioning is a task that is frequently accessed during navigation, and its required services also need to be prioritized in the caching strategy. Therefore, when designing a service caching policy, it is crucial to ensure the availability of critical services while also caching frequent services appropriately to prevent misleading decisions in extreme cases.

In response to the actual application requirements in VEC scenarios, this paper proposes a collaborative optimization strategy of service caching and task offloading for dual dependency VEC scenarios. First, a service criticality prediction method is proposed to meet complex task dependencies, and a hybrid service caching model is designed. Second, a task priority scheduling method is proposed to avoid delays caused by task chain interruptions. Finally, to solve the joint optimization problem of

service caching and task offloading in dynamic and complex VEC scenarios, a collaborative optimization algorithm based on Deep Reinforcement Learning (DRL) is proposed. The main contributions of this paper are as follows:

- Frequent task types can be easily obtained through access statistics, but the complex DAG dependencies make it difficult to directly identify the critical task types. Therefore, this paper proposes a prediction method based on Gated Graph Recurrent Network (GGRN) to perceive complex task dependencies and accurately capture the service requirements of critical task types. Based on this, a hierarchical active-passive hybrid caching strategy is designed. To the best of our knowledge, this is the first research work that takes into consider both frequent and critical task service requirements.
- A heterogeneous-aware topology priority allocation algorithm is proposed to achieve more efficient task scheduling and avoid cascading delays within task chains. The algorithm proactively perceives the characteristics of different applications and tasks, assigning a global priority to each task. This enables the establishment of a hierarchical advancement mechanism within complex DAG dependency structures, thereby reducing potential delays and improving system response efficiency.
- We formulate a joint service caching and task offloading optimization problem in a three-layers VEC system for dual-dependency scenarios, and then model the optimization problem as a Markov Decision Process (MDP) and propose a Proximal Policy Optimization (PPO)-driven hierarchical hybrid service caching and task offloading collaborative optimization algorithm to solve it, called the COHCTO algorithm. The objective is to maximize weighted composite rewards by covering key metrics such as caching hit rate, application success rate, delay, and energy consumption.

The remainder of this paper is organized as follows. Section II introduces related work. Section III presents the system model and problem formulation. Section IV describes the main design of the algorithm. Section V presents the experimental simulation results. Finally, Section VI concludes the paper.

II. RELATED WORK

We review the literature on task offloading and service caching, with a focus on their limitations in handling dependent tasks in Vehicular Edge Computing (VEC) scenarios. More details are provided in the supplementary file for the related work.

A. Task Computing Offloading

Offloading of dependent tasks in Mobile Edge Computing (MEC) has received growing attention due to the execution constraints imposed by task dependencies. These constraints often result in cascading delays that negatively affect overall system performance. Existing studies, including those based on reinforcement learning [17], priority scheduling [18], and

heuristic algorithms [19], [20], [21], primarily optimize individual task execution in terms of delay or energy consumption. However, they generally lack explicit modeling of inter-task dependencies and are therefore ineffective in mitigating delay propagation across task chains. Furthermore, many of these methods adopt static scheduling mechanisms or oversimplified assumptions, which reduce their adaptability in dynamic VEC scenarios.

B. Joint Optimization of Service Caching and Task Offloading

In VEC environments, tasks frequently require specific service components for execution. Due to limited caching capacity at edge nodes and onboard units, it is often infeasible to store all services in advance. Consequently, service caching has become a critical factor influencing offloading efficiency. Several works have investigated joint optimization strategies for service caching and task offloading [5], [22], [23]. Nevertheless, most of these methods rely on static service placement policies or assume task independence, thereby overlooking the interplay between service availability and task execution order. These limitations restrict their effectiveness in real-world scenarios, where task-service dependencies evolve dynamically.

C. Summary

Although existing research has made progress in service caching and task offloading in VEC environments, few studies address the dual constraints of task and service dependencies simultaneously. Many methods fail to account for the dynamic service requirements of different tasks and the hierarchical nature of cache management, limiting their effectiveness in complex scenarios. Even when service caching is considered in task offloading, strategies often rely on predefined policies that cannot adapt to evolving task execution patterns, leading to inefficient resource utilization. Additionally, most approaches lack fine-grained forecasting models for service demand and focus primarily on optimizing latency or energy consumption in isolation.

In contrast, we integrate both task and service dependencies into a unified service caching and task offloading framework. First, it addresses cascading delays caused by task dependencies by designing an adaptive scheduling strategy that alleviates execution bottlenecks and enhances system responsiveness in dynamic environments. Second, the proposed caching strategy considers not only the resource constraints on the vehicle side but also introduces a task-aware service caching mechanism at the edge, supported by predictive modeling of service demand under dependency constraints. Finally, a collaborative optimization approach is developed to jointly address task dependency, service availability, and service diversity, aiming to improve system performance across multiple dimensions, including cache hit rate, application success rate, delay, and energy consumption.

III. SYSTEM MODEL AND PROBLEM FORMULATION

This section first describes the research scenario and then introduces the critical components of this study, including the

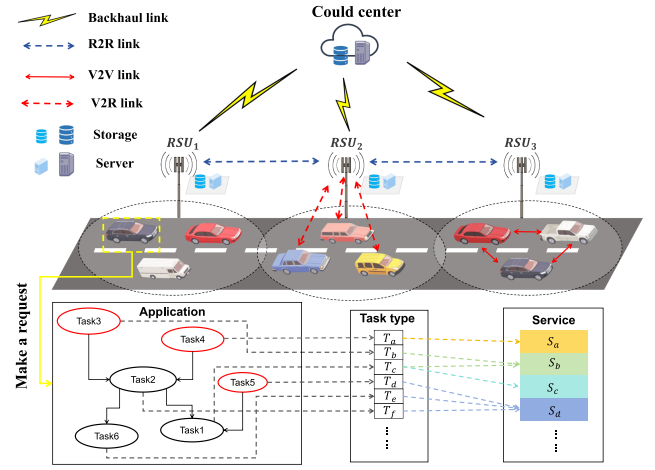


Fig. 1. System model.

service caching model and task computation offloading model. Finally, a multi-objective optimization problem is proposed. Table I outlines the key symbols mentioned in the text. More details are provided in the supplementary file for related analyses of the application model and communication model.

A. System Model

The system consists of a cloud server, RSUs, and multiple mobile vehicles. The system architecture is shown in Fig. 1. The RSU set is denoted as $\tilde{R} = \{r_1, r_2, \dots, r_M\}$, where M is the number of RSUs. The vehicle set is denoted as $\tilde{V} = \{v_1, v_2, \dots, v_N\}$, where N is the number of vehicles. Each RSU can communicate with vehicles within its coverage area to collaborate on task processing. The system operation time is divided into multiple equal-length discrete time slots, represented by the set $\tilde{T} = \{1, 2, \dots, T\}$. At each time slot $\tau \in \tilde{T}$, each vehicle $v_i \in \tilde{V}$ generates an application request consisting of a set of tasks dependent on each other in the form of DAG. The set of applications is denoted as $\tilde{A} = \{a_1, a_2, a_3, \dots, a_Q\}$, where Q represents the types of applications. Each application consists of a set of tasks, denoted as $T = \{t_1, t_2, \dots, t_K\}$, where K is the number of tasks within the application. Each task t_k is described by a tuple $(i(t_k), \varphi(t_k), \delta(t_k), \theta(t_k))$, where $i(t_k)$ represents the input data size of task t_k ; $\varphi(t_k)$ represents the number of CPU cycles required for computing task t_k ; $\delta(t_k)$ represents the type of task t_k ; and $\theta(t_k)$ represents the local time constraint of task t_k . Tasks can be executed concurrently or sequentially, depending on the task dependencies, and each task execution relies on the availability of specific services. The system defines a set of services, denoted as $\tilde{S} = \{s_1, s_2, \dots, s_L\}$. For each t_k , the required service $s_{\tilde{k}}$ must be available to complete the computation. The dependencies between tasks and services are crucial for supporting computation.

As shown in Fig. 1, the relationship between tasks and services may fall into one of the following cases: 1) One-to-One Mapping: Each task type relies on only one specific service module. For example, task type T_a depends on service module S_a , and task type T_b depends on service module S_b . 2) One-to-Many

TABLE I
DEFINITION OF KEY SYMBOLS

Symbol	Description	Symbol	Description
$i(t_k)$	Input data size of the k -th task	$\varphi(t_k)$	CPU frequency required for the k -th task
$\delta(t_k)$	Type of the k -th task	$\theta(t_k)$	The local time constraint of the k -th task
$\hat{\phi}(s_k)$	The size of service data required by task t_k	$\hat{\eta}(s_k)$	The service size required by task t_k
$h_{v,r}^{up}$	Uplink channel gain between vehicle and RSU	$h_{r,v}^{down}$	Downlink channel gain between vehicle and RSU
$R_{r,v}^{down}$	Downlink transmission rate between vehicle and RSU	R_{v2v}	Transmission rate between vehicles
$R_{v,r}^{up}$	Uplink transmission rate between vehicle and RSU	p_{tr}^v	Transmission power of the vehicle
p_{co}^v	Computation power of the vehicle	p_{tr}^r	Transmission power of the RSU
p_{co}^r	Computation power of the RSU	p_{tr}^c	Transmission power of the cloud
R_{r2c}^{link}	Transmission rate for R2C communication	R_{r2r}^{link}	Transmission rate for R2R communication

Mapping: Some task types may simultaneously rely on multiple service modules. For example, task type T_c depends on service module S_b and S_c . 3) Many-to-One Mapping: Multiple task types can share the same service module. For example, task type T_e and T_f depend on service module S_d . Unlike the existing study [23], this paper assumes that the cloud server pre-configures all services. At the same time, RSUs and vehicles are constrained by storage limitations and can only cache some services. The system adopts a distributed partial offloading strategy, allowing different applications' tasks to be offloaded to the cloud server, RSU, or local vehicles. Considering the small size of task results, we ignore the download delay of task results.

B. Service Caching Model

This paper designs a layered active-reactive hybrid caching model. In this model, RSUs proactively decide whether and where to cache based on the agent policy. Meanwhile, when the agent selects a vehicle as a computational node, the vehicle makes a caching decision based on a predefined response policy after completing the task computation. More details are provided in the supplementary file for the overhead analysis of the caching strategy.

During time slot τ , the caching status of service s_k on the m -th RSU and the n -th vehicle is represented by the binary control variables $C_{r_m}^k(\tau)$ and $C_{v_n}^k(\tau)$, respectively. $C_{r_m}^k(\tau) = \{0, 1\}$ indicates whether the service s_k is cached in the m -th RSU. $C_{v_n}^k(\tau) = \{0, 1\}$ indicates whether the service s_k is cached in the n -th vehicle.

Constraints must be imposed on each RSU and vehicle to ensure the rational utilization of caching resources. Therefore, for each time slot $\tau \in \tilde{T}$, the caching usage in each RSU and vehicle should not exceed their maximum caching capacities $M_{r_m}^{total}$ and $M_{v_n}^{total}$. The caching service occupancy of RSU and vehicle is defined as (1) and (2), respectively.

$$M_{r_m}^\tau = \sum_{\forall s_k \in \tilde{S}} C_{r_m}^k(\tau) \cdot \hat{\eta}(s_k), \quad m \in \{1, M\} \quad (1)$$

$$M_{v_n}^\tau = \sum_{\forall s_k \in \tilde{S}} C_{v_n}^k(\tau) \cdot \hat{\eta}(s_k), \quad n \in \{1, N\} \quad (2)$$

where $\hat{\eta}(s_k)$ represents the service size.

We define a threshold γ to distinguish small-scale services. Although the constraint in our model is imposed on the service data size $\hat{\phi}(s_k)$, the design of γ is informed by several underlying system-level factors, including the computational demand of tasks, the associated service generation cost at RSUs, and the overhead of transmitting services over V2I links. In our implementation, each task is associated with a difficulty level that reflects the required number of CPU cycles. Services supporting low-difficulty tasks tend to have smaller data size $\hat{\phi}(s_k)$, as simpler computation generally implies lighter models or configuration files. Thus, by restricting vehicle-side caching to services whose data size does not exceed γ , we implicitly filter for lightweight services that can be efficiently customized by RSUs and rapidly delivered to vehicles. This design ensures that vehicle caching remains communication-efficient and responsive under bandwidth and storage constraints.

Specifically, if the service data size $\hat{\phi}(s_k)$ required for RSU customization is at most γ , we classify s_k as a small-scale service. In our model, vehicles are only permitted to cache these small-scale services that can be efficiently generated by RSUs. Therefore, we impose the following constraint:

$$C_{v_n}^k(\tau) \in \begin{cases} \{0\}, & \text{if } \hat{\phi}(s_k) > \gamma \\ \{0, 1\}, & \text{if } \hat{\phi}(s_k) \leq \gamma \end{cases} \quad (3)$$

where $\hat{\phi}(s_k)$ represents the service data size. This formula ensures that high transmission delays will not be generated in the V2I link due to excessive service data. In addition, this paper assumes that the $\hat{\eta}(s_k)$ is positively correlated with $\hat{\phi}(s_k)$. If the data size required to generate a service is larger, the resulting service entity (such as model, configuration file, etc.) will also be larger.

When the caching space is full, RSUs and vehicles can update the contents of their caching. Let the caching state set within time slot τ be denoted as (4).

$$C_j^k(\tau) = \{C_{r_m}^k(\tau) \cup C_{v_n}^k(\tau), \forall \tau \in \tilde{T}\} \quad (4)$$

To maximize the caching hit rate and minimize delays caused by fetching services from remote servers. During time slot τ , the reward of caching hits of computing nodes is defined as (5).

$$I_j(\tau) = \sum_{j \in R \cup \tilde{V}} C_j^k(\tau) \cdot \Omega_j(\tau), \quad \forall \tau \in \tilde{T} \quad (5)$$

where $\tilde{R} \cup \tilde{V}$ represents the set of caching nodes, including all RSUs and vehicles, and $\Omega_j(\tau)$ represents the priority weight of cached service s_k in node j during time slot τ .

C. Task Computation Offloading Model

This section first defines the decision variables for task offloading and then provides a detailed description of the cost incurred by each computation method.

The decision vector $x_{j,k}^i(\tau) = \{x_{n,k}^v(\tau), x_{m,k}^r(\tau), x_{d,k}^c(\tau)\}$ is used to represent the offloading decision of task t_k to device j during time slot τ . Specifically, $x_{n,k}^v(\tau) \in \{0, 1\}$ indicates whether task t_k is computed locally, $x_{m,k}^r(\tau) \in \{0, 1\}$ indicates whether task t_k is offloaded to the RSU, and $x_{d,k}^c(\tau) \in \{0, 1\}$ indicates whether task t_k is offloaded to the cloud. Since tasks are inseparable, each task can only choose one computation model at a time slot. Let $i \in \{v, r, c\}$ represent the type of computing node and $j \in m + n + 1$ represent the computing node index. The total offloading decision within the time slot τ is defined as (6).

$$X(\tau) = [x_{j,1}^i(\tau), x_{j,2}^i(\tau), \dots, x_{j,k}^i(\tau)] \quad (6)$$

In addition, tasks have DAG dependencies, meaning task t_k can only start executing if all preceding tasks t_p in the set $F_{\hat{p}}$ have been completed. Therefore, to avoid task conflicts, it is necessary to ensure that the offloading decision for task t_k at time slot τ is not made earlier than any preceding tasks. Finally, we set the total computing resources of each vehicle and RSU to be f_v^{max} and f_r^{max} , respectively. For each time slot τ , it is necessary to ensure that the computing resources used by each node do not exceed the preset maximum computing resources, thereby ensuring the stability and efficiency of the system. The computing resources used by the local and RSU can be computed using (7) and (8), respectively.

$$f_{v_n}^\tau = \sum_{n=1}^N x_{n,k}^v(\tau) \cdot \varphi(t_k) \quad (7)$$

$$f_{r_m}^\tau = \sum_{m=1}^M x_{m,k}^r(\tau) \cdot \varphi(t_k) \quad (8)$$

Next, we discuss the costs that may be incurred during the execution of the three computing models.

1) *Local Computation*: If the required service has been cached in the task vehicles that made the request, the cost of local computing only includes computing delay and computing energy consumption. The total delay and energy consumption are defined as (9) and (10), respectively.

$$\tilde{T}_{t_k}^v = \frac{\varphi(t_k)}{f_v} \quad (9)$$

$$\tilde{E}_{t_k}^v = \frac{\varphi(t_k)}{f_v} \cdot p_{co}^v \quad (10)$$

where f_v represents the available computing resources of the task vehicles, $\varphi(t_k)$ represents the CPU frequency required to execute task t_k , p_{co}^v is the computed power of the vehicle.

In our scenario, cached contents are shared among vehicles. If a task vehicle does not cache the required service, it can retrieve the corresponding service from nearby service vehicles that have cached it. The service is transmitted from the service vehicle to the task vehicle, and the task is then completed locally. This means that once the task vehicle successfully obtains the required service, it can independently complete subsequent task processing without maintaining continuous connections to specific service vehicles. The total delay and energy consumption are given by (11) and (12), respectively.

$$\tilde{T}_{t_k}^v = \frac{\hat{\eta}(s_k)}{R_{v2v}} + \frac{\varphi(t_k)}{f_v} \quad (11)$$

$$\tilde{E}_{t_k}^{v2v} = \frac{\hat{\eta}(s_k)}{R_{v2v}} \cdot p_{tr}^{v'} + \frac{\varphi(t_k)}{f_v} \cdot p_{co}^v \quad (12)$$

where R_{v2v} is the V2V transmission rate, $p_{tr}^{v'}$ is the transmit power of the service vehicles, and $\hat{\eta}(s_k)$ represents the service size.

If neither task vehicles nor service vehicles have the required service cached, when the service data size does not exceed the threshold, the customized service of task vehicles can be obtained from RSU without making a remote request to the cloud. At this time, the total delay includes service uplink delay, service generation delay at RSU, service downlink delay, and local computing delay. The delay and energy consumption of local computing are defined as (13) and (14), respectively.

$$\tilde{T}_{t_k}^v = \frac{\hat{\phi}(s_k)}{R_{v,r}^{up}} + \alpha \cdot \varphi(t_k) + \frac{\hat{\eta}(s_k)}{R_{r,v}^{down}} + \frac{\varphi(t_k)}{f_v} \quad (13)$$

$$\begin{aligned} \tilde{E}_{t_k}^v = & \frac{\varphi(t_k)}{f_v} \cdot p_{co}^v + \alpha \cdot \varphi(t_k) \cdot p_{co}^r \\ & + \frac{\hat{\phi}(s_k)}{R_{v,r}^{up}} \cdot p_{tr}^v + \frac{\hat{\eta}(s_k)}{R_{r,v}^{down}} \cdot p_{tr}^r \end{aligned} \quad (14)$$

where $\hat{\phi}(s_k)$ represents the service data size, p_{tr}^v is the transmit power of the task vehicles, p_{tr}^r is the transmit power of the RSU. To simplify the model, we consider the service generation time proportional to the CPU frequency required for task execution, defined as $\alpha \cdot \varphi(t_k)$, where α is the scaling factor.

2) *RSU Computation*: When t_k is offloaded to the RSU, the task will experience queuing delay when it is transmitted between the vehicle and the RSU, which is jointly affected by the resource contention caused by the task priority in the queue and the dependencies between tasks. The queuing delay of t_k during transmission is the maximum of the global priority delay and the DAG dependency delay, computed by (15).

$$\tilde{T}_q^{v2r}(t_k) = \max(\tilde{T}_q^{gp}(t_k), \tilde{T}_q^{dag}(t_k)) \quad (15)$$

The global task priority of each task $t_i \in \mathcal{Q}$ is expressed as $p(t_i^\tau)$, where \mathcal{Q} represents the task queue on the RSU. Task t_k can only start computing after task t_j is completed, provided that $p(t_j^\tau) > p(t_k^\tau)$. The queuing delay caused by the global priority of tasks can be computed by (16).

$$\tilde{T}_q^{gp}(t_k) = \frac{\sum_{t_j \in \mathcal{Q}} \varphi(t_j)}{f_r} \quad (16)$$

If task t_k depends on a set of predecessor tasks $F_{\hat{p}}$, then task t_k must wait until all its predecessor tasks $t_p \in F_{\hat{p}}$ have been completed before it can be executed. The queuing delay of task t_k caused by task dependencies is the maximum total delay among all predecessor tasks, computed using (17).

$$\tilde{T}_q^{dag}(t_k) = \max_{\forall t_p \in F_{\hat{p}}} \tilde{T}_{t_p}^j \quad (17)$$

If the services required to execute task t_k are already cached in RSU, the total delay of RSU processing the task includes the uplink delay of task input data, the computation delay at RSU, and the queuing delay of task t_k at RSU. At this time, the total delay and energy consumption are defined as (18) and (19), respectively.

$$\tilde{T}_{t_k}^r = \frac{i(t_k)}{R_{v,r}^{up}} + \frac{\varphi(t_k)}{f_r} + \tilde{T}_q^{v2r}(t_k) \quad (18)$$

$$\tilde{E}_{t_k}^r = \frac{\varphi(t_k)}{f_r} \cdot p_{co}^r + \frac{i(t_k)}{R_{v,r}^{up}} \cdot p_{tr}^v \quad (19)$$

where f_r represents the remaining computing resources of RSU.

If the service required to perform task t_k is not cached in the RSU, the total delay of the RSU in processing the task includes the transmission delay of the service from the cloud to the RSU in addition to the several delays included in (18). Since the cloud computing resources are sufficient and include all services, the service generation delay and service data uplink delay are approximately 0 at this time. The total delay and energy consumption are defined as (20) and (21), respectively.

$$\tilde{T}_{t_k}^r = \frac{i(t_k)}{R_{v,r}^{up}} + \frac{\varphi(t_k)}{f_r} + \frac{\hat{\eta}(s_{\hat{k}})}{R_{c,r}^{link}} + \tilde{T}_q^{v2r}(t_k) \quad (20)$$

$$\tilde{E}_{t_k}^r = \frac{\varphi(t_k)}{f_r} \cdot p_{co}^r + \frac{\hat{\eta}(s_{\hat{k}})}{R_{c,r}^{link}} \cdot p_{tr}^c + \frac{i(t_k)}{R_{v,r}^{up}} \cdot p_{tr}^v \quad (21)$$

3) *Cloud Computation*: When the RSU cannot provide sufficient computing resources, the vehicle can choose to offload the task to the cloud. The cloud server has all the services required to perform the task. In addition, due to its rich computing resources, the computing time of the task offloaded to the cloud is usually negligible. We mainly focus on the transmission delay. Specifically, the task offloaded from the vehicle to the cloud first needs to transmit the task's input data and service data to the RSU through V2R communication, and then the RSU forwards it to the cloud through the core network. Currently, the total delay and energy consumption are defined by (22) and (23), respectively.

$$\tilde{T}_{t_k}^c = \frac{(i(t_k) + \hat{\phi}(s_{\hat{k}}))}{R_{v,r}^{up}} + \frac{(i(t_k) + \hat{\phi}(s_{\hat{k}}))}{R_{r2c}^{link}} \quad (22)$$

$$\tilde{E}_{t_k}^c = \frac{(i(t_k) + \hat{\phi}(s_{\hat{k}}))}{R_{v,r}^{up}} \cdot p_{tr}^v + \frac{(i(t_k) + \hat{\phi}(s_{\hat{k}}))}{R_{r2c}^{link}} \cdot p_{tr}^r \quad (23)$$

Therefore, at time slot τ , the total cost for computing node j to execute application $a_q \in \tilde{A}$ can be expressed as follows:

$$Z_j(\tau) = \sum_{\forall t_k \in a_q} \underbrace{x_{i,k}^j(\tau) \cdot \tilde{T}_{t_k}^j}_I + \underbrace{x_{i,k}^j(\tau) \cdot \tilde{E}_{t_k}^j}_II, \quad j \in \{v, r, c\}, \quad i \in m + n + 1 \quad (24)$$

D. Problem Formulation

The application success rate is an essential indicator for measuring the effectiveness of collaborative optimization of caching and offloading decisions. For each application a_q , if the maximum completion delay of all tasks in the task list does not exceed the deadline of applications, the application at time slot τ is considered completed. The reward of the application success rate at time slot τ can be expressed by (25).

$$D_j(\tau) = \begin{cases} \alpha, & \text{if } \max_{t_k \in a_q} \left(\tilde{T}_{t_k}^j \right) \leq \phi(a_q) \\ -\alpha, & \text{if } \max_{t_k \in a_q} \left(\tilde{T}_{t_k}^j \right) > \phi(a_q) \end{cases}, \quad \alpha > 0 \quad (25)$$

where $\phi(a_q)$ represents the deadline of application a_q , α represents the benefit coefficient of the application being completed on time, and D_j represents the benefit of the application being completed on time on the j -th computing node. A positive value indicates successful execution of the application, and a negative value indicates failure.

We aim to jointly optimize the service caching decision $C(\tau)$ (4) and computation offloading decision $X(\tau)$ (6) to maximize a weighted reward that considers the total cost, caching hit rate, and application success rate. The formal description is defined as (26). The coefficients ω_1 , ω_2 , and ω_3 control the balance between caching hit rate, total cost, and application success rate, respectively.

$$\max_{\{C(\tau), X(\tau)\}} \sum_{j=1}^{M+N} (\omega_1 I_j(\tau) - \omega_2 Z_j(\tau) + \omega_3 D_j(\tau))$$

s.t.

$$\begin{aligned} C1 : & \quad C_{\hat{k}}^j(\tau) \in \{0, 1\}, \quad \forall s_{\hat{k}} \in \tilde{S}, \quad \forall j \in \{v, r\} \\ C2 : & \quad M_j^\tau \leq M_j^{total}, \quad \forall j \in \{v, r\} \\ C3 : & \quad x_{j,k}^i(\tau) \in \{0, 1\}, \quad \forall t_k \in T, \quad i \in \{v, r, c\} \\ C4 : & \quad x_{j,k}^i(\tau) \leq \sum_{\tau' < \tau} x_{j,p}^i(\tau'), \quad \forall t_p \in F_{\hat{p}} \\ C5 : & \quad f_j^\tau \leq f_j^{max}, \quad \forall j \in \{v, r\} \end{aligned} \quad (26)$$

where C1 represents the caching location of service $s_{\hat{k}}$; C2 indicates that the caching occupancy of RSUs and vehicles cannot exceed the maximum caching capacity of RSUs and vehicles; C3 indicates that at time slot τ , each task t_k can only select one type of computational method; C4 shows that task t_k can be executed if the preceding task t_p has been completed, where the inequality on the left represents whether task t_k is being executed, and the inequality on the right indicates whether the preceding task t_p has already been completed in some previous

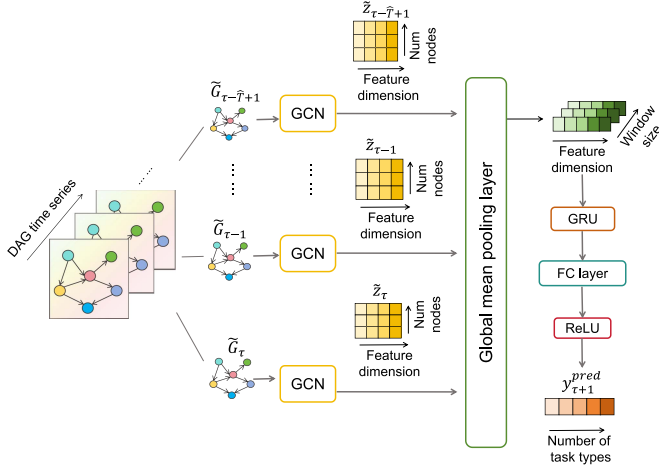


Fig. 2. GGRN-based service demand criticality prediction model.

time slot $\tau' < \tau$; $C5$ represents that the computational workload of each computing node during each time slot must not exceed its maximum computational capacity. The above problem is a dynamic multi-objective optimization problem, and due to the coupling relationship between task offloading and service caching, solving it is quite challenging. Therefore, we model this problem as MDP, and in the next section, we propose a DRL-based method to solve this problem.

IV. ALGORITHM DESIGN

This section first introduces a service demand criticality prediction method based on the GGRN model. Second, the heterogeneous-aware topology priority allocation algorithm proposed is introduced. Finally, the COHCTO algorithm proposed is introduced. It aims to maximize the weighted compound reward by covering key indicators such as caching hit rate, application success rate, delay, and energy consumption. COHCTO can optimize service caching and task offloading under different network conditions by learning the optimized strategy in a constantly changing environment. More details are provided in the supplementary file for the algorithm computational complexity analysis.

A. Service Demand Criticality Prediction Model

This section details the service demand criticality prediction method based on the GGRN model. This method can capture the structural relationships and timing changes between tasks in dual-dependency scenarios, thereby predicting key service requirements and laying the foundation for the proactive caching strategy of the RSU layer. It should be noted that the model is used to predict dependencies between task types rather than specific task-to-task dependencies, which simplifies the model's computation and improves its ability to generalize to various task combinations. The exact steps are outlined in Algorithm 1.

Fig. 2 shows the GGRN service demand criticality prediction process. First, load the task information from the history. Then, the task DAG dependency topological sorting of each time slot is converted into graph structure data that can be recognized

by the model and forms a time series. The window size is set to \hat{T} , and for each time slot τ , the graph sequence of the previous \hat{T} time slots is defined as $\mathcal{G}_\tau = \{\tilde{G}_{\tau-\hat{T}+1}, \dots, \tilde{G}_\tau\}$. The sequence is fed to the Graph Convolutional Network (GCN) layer for feature extraction, generating node-level embeddings $\mathcal{Z}_\tau = \{\tilde{Z}_{\tau-\hat{T}+1}, \dots, \tilde{Z}_\tau \mid \tilde{G}_i \in \mathcal{G}_\tau\}$. These node-level embeddings are then transformed into graph-level embeddings $\mathcal{Z}'_\tau = \{\tilde{Z}'_{\tau-\hat{T}+1}, \dots, \tilde{Z}'_\tau \mid \tilde{Z}_i \in \mathcal{Z}_\tau\}$ through global pooling.

The graph-level embeddings \mathcal{Z}'_τ are subsequently fed into the Gated Recurrent Unit (GRU) layer to capture the temporal sequence's dynamic changes. The goal is to predict the frequency of each task type dependent on other task types in the DAG dependency at time slot $\tau + 1$. The quantitative result is the criticality value of task types. Finally, the criticality of the service is evaluated through the mapping relationship between tasks and services. Since the mapping between tasks and services is not a simple one-to-one mapping, it is necessary to sum up the criticality values of all task types that depend on the service and then normalize them to measure the criticality of the service comprehensively. The specific process is as follows:

1) *Graph Convolution Layer Extracts DAG Features*: The GGRN model utilizes the GCN to extract features from the task dependency graph \tilde{G}_i at each time slot in the \hat{T} time windows. Through the propagation mechanism of GCN layer, each node can integrate its characteristics with those of its neighboring nodes, thereby progressively aggregating dependency information within the graph structure. The node-level embedding features \tilde{Z}_i are then obtained and transformed into graph-level embedding features \tilde{Z}'_i through global pooling. These features effectively represent the complex dependency relationships among tasks, providing rich information support for the subsequent time-series modeling. The core computation of GCN layer is defined as (27).

$$\tilde{Z}_i^{(l+1)} = \partial \left(\hat{\mathbf{A}} \tilde{Z}_i^{(l)} \mathbf{W}^{(l)} \right) \quad (27)$$

where $\hat{\mathbf{A}}$ is the normalized adjacency matrix representing task dependencies; $\tilde{Z}_i^{(l)}$ is the node feature matrix at the l -th layer; $\mathbf{W}^{(l)}$ is the weight matrix at the l -th layer; and ∂ is the activation function.

2) *GRU Layer Captures Temporal Relationships*: The GRU layer processes the graph-level embedding feature sequence \mathcal{Z}'_τ , extracted by the GCN layer, to capture the temporal dynamics of task dependencies. By sequentially modeling the input features at each time slot, the GRU layer updates its hidden state h_τ , thereby retaining dependency information across the time series and learning temporal variation patterns. The computation is expressed as follows:

$$h_\tau = GRU(h_{\tau-1}, \mathcal{Z}'_\tau) \quad (28)$$

where \mathcal{Z}'_τ represents the aggregated graph-level embedding features at time slot τ , derived from the GCN layer and global pooling feature extraction process. The hidden state h_τ of the GRU layer stores the sequential information from the past to the present time slot.

Algorithm 1: Service Demand Criticality Prediction Using GGRN.

Require:

1: **Input:** Historical task information from $\tau - \hat{T} + 1$ to τ

Ensure:

```

2: Output: Criticality values  $y_{\tau+1}^{pred}(\delta(t_k))$  for all task
   types  $\delta(t_k)$  at time slot  $\tau + 1$ 
3: Initialize: Window size  $\hat{T}$ , GRU hidden state  $h_0$ 
4: for each time slot  $\tau \in \hat{T}$  do
5:   Construct the graph sequence
    $\mathcal{G}_\tau = \{\tilde{G}_{\tau-\hat{T}+1}, \dots, \tilde{G}_\tau\}$ 
6:   Initialize the graph-level embedding set  $\mathcal{Z}'_\tau \leftarrow \emptyset$ 
7:   for each graph  $\tilde{G}_i \in \mathcal{G}_\tau$  do
8:     Initialize node features  $\tilde{Z}_i^{(0)} \leftarrow$  One-hot encoding
       of task types in  $\tilde{G}_i$ 
9:     for layer  $l = 1$  to  $L$  do
10:       $\tilde{Z}_i^{(l)} \leftarrow \partial(\hat{\mathbf{A}} \tilde{Z}_i^{(l-1)} \mathbf{W}^{(l-1)})$ 
11:    end for
12:     $\tilde{Z}'_i \leftarrow \text{GlobalPooling}(\tilde{Z}_i^{(L)})$ , add  $\tilde{Z}'_i$  to  $\mathcal{Z}'_\tau$ 
13:  end for
14:  Update GRU hidden state  $h_\tau \leftarrow \text{GRU}(h_{\tau-1}, \mathcal{Z}'_\tau)$ 
15:  for each task type  $\delta(t_k)$  do
16:     $y_{\tau+1}^{pred}(\delta(t_k)) \leftarrow \text{ReLU}(W_{fc} \cdot h_\tau + b_{fc})$ 
17:  end for
18:  for each service  $s_{\hat{k}} \in \tilde{\mathcal{S}}$  do
19:    Using (30) to compute the service criticality value
     $y_{\tau+1}^{pred}(s_{\hat{k}})$ 
20:  end for
21: end for

```

3) *Model Training and Prediction:* The prediction layer is formed by a feedforward neural network, using the final hidden state h_τ of the GRU layer as input. This hidden state encapsulates information from the previous \hat{T} time slots. Then, the GGRN model applies a fully connected layer and uses a ReLU activation function to output criticality values of task types $y_{\tau+1}^{pred}(\delta(t_k))$, as shown in (29).

$$y_{\tau+1}^{pred}(\delta(t_k)) = \text{ReLU}(W_{fc} h_\tau + b_{fc}) \quad (29)$$

where W_{fc} and b_{fc} are the weight and bias parameters of the fully connected layer.

Finally, the criticality value $y_{\tau+1}^{pred}(s_{\hat{k}})$ of each service $s_{\hat{k}} \in \tilde{\mathcal{S}}$ is determined by summing and normalizing the criticality values $y_{\tau+1}^{pred}(\delta(t_k))$ of all task types $\delta(t_k)$ associated with the service, as defined in (30).

$$y_{\tau+1}^{pred}(s_{\hat{k}}) = \frac{\sum_{\delta(t_k) \in T_{s_{\hat{k}}}} y_{\tau+1}^{pred}(\delta(t_k))}{\sum_{s_{\hat{k}} \in \tilde{\mathcal{S}}} \sum_{\delta(t_k) \in T_{s_{\hat{k}}}} y_{\tau+1}^{pred}(\delta(t_k))} \quad (30)$$

where $T_{s_{\hat{k}}}$ represents the set of task types associated with service $s_{\hat{k}}$, and $\tilde{\mathcal{S}}$ represents the set of all services.

B. Topology Priority Assignment Algorithm

In heterogeneous VEC environments, applications exhibit varying levels of time sensitivity and urgency. Effectively allocating limited computing resources to prioritize high-urgency tasks is a key challenge for scheduling algorithms, especially when compounded by DAG-based task dependencies. To address this, we propose a heterogeneous topology-aware priority allocation algorithm that dynamically assigns global priorities to tasks for unified and efficient scheduling. Additionally, local time constraints and penalty mechanisms are introduced to mitigate cascading delays, enabling flexible and intelligent task management. The algorithm is detailed in Algorithm 2.

At the beginning of each time slot, the application set $\tilde{\mathcal{A}} = \{a_1, a_2, \dots, a_Q\}$ is collected. For each application $a_q \in \tilde{\mathcal{A}}$, the following information is collected: application deadline $\phi(a_q)$, the set of tasks contained $T^q = \{t_1^q, t_2^q, \dots, t_K^q\}$, the attributes of each task, and the task dependencies represented by DAG. The global priority $p(t_k^q)$ of a task t_k^q is determined by the initial priority weight of the application to which it belongs, the computational complexity of the task, and whether the task is on the critical path. This ensures that high computational complexity and critical tasks within emergency applications are given higher priority when scheduling. The initial priority of each application is set to the inverse of its deadline, and the initial priority weight ω_{a_q} of the application is obtained after normalization. The global priority of the task is obtained by (31).

$$p(t_k^q) = \alpha_1 \cdot \omega_{a_q} + \alpha_2 \cdot \frac{\varphi(t_k^q)}{\sum_{q=1}^Q \sum_{k=1}^K \varphi(t_k^q)} + \alpha_3 \cdot \Gamma(t_k^q) \quad (31)$$

where α_1 , α_2 and α_3 are the coefficients controlling the three influencing factors; ω_{a_q} is the global initial normalized priority weight of the application computed based on the application deadline, reflecting the urgency of the application; $\varphi(t_k^q)$ denotes the computational complexity of task t_k^q ; and $\Gamma(t_k^q)$ is a binary indicator that equals 1 if the task is on the critical path and 0 otherwise.

Then, a reasonable local deadline is assigned to each task based on the computational complexity and global priority of each task, as defined by (32). In this formula, Part I represents the basic deadline assigned to the task t_k^q , which is positively correlated with the computational complexity of the task $\varphi(t_k^q)$. Tasks with high computational complexity will receive more basic deadlines to ensure that they have enough time to complete the computation; Part II adjusts the local deadline assigned to the task within each application based on the global priority of the task, and high-priority tasks will be assigned stricter local time constraints. This allocation strategy effectively balances the computational requirements and priority requirements of the task, improves scheduling efficiency, and ensures the timely completion of critical tasks.

$$\theta(t_k^q) = \underbrace{\phi(a_q) \times \frac{\varphi(t_k^q)}{\sum_{t_k^q \in T^q} \varphi(t_k^q)}}_I \times \underbrace{\left(1 - \frac{p(t_k^q)}{\sum_{t_k^q \in T^q} p(t_k^q)}\right)}_{II} \quad (32)$$

Algorithm 2: Topology Priority Allocation Algorithm.

```

1: Input: Application deadline  $\phi(a_q)$ , set of tasks
    $T^q = \{t_1^q, t_2^q, \dots, t_K^q\}$ , attributes of each task  $t_k^q$ , task
   dependency graph.
2: Output: local deadlines  $\theta(t_k^q)$  for each task
3: for each application  $a_q \in \bar{A}$  do
4:   Computing the initial unnormalized priority of the
   application  $\tilde{\omega}(a_q) \leftarrow \frac{1}{\phi(a_q)}$ 
5:   Normalizing application initial priority
    $\omega(a_q) \leftarrow \frac{\tilde{\omega}(a_q)}{\sum_{a_q \in \bar{A}} \tilde{\omega}(a_q)}$ 
6:   for each task  $t_k^q \in T^q$  do
7:     Using formulas (31) and (32) to compute the
     global priority  $p(t_k^q)$  and assign local deadlines
      $\theta(t_k^q)$ 
8:      $\theta(t_{child}^q) \leftarrow \max(\theta(t_{child}^q), \theta(t_{parent}^q))$ 
9:   end for
10: end for

```

Finally, the local deadlines are adjusted starting from the source node of the DAG to ensure that the local time constraints of the subtasks t_{child}^q are not earlier than those of the parent tasks t_{parent}^q .

C. Cooperative Service Caching and Task Offloading Algorithm

This section proposes a joint optimization problem for service caching and task offloading in a three-layers VEC system under dual-dependency scenarios. To solve this complex optimization problem effectively, we model it as MDP and propose a CO-HCTO algorithm for service caching and task offloading. The goal is to maximize a weighted composite reward of several key performance indicators, including caching hit rate, application success rate, delay, and energy consumption.

In this algorithm, we propose a hierarchical hybrid active-reactive service caching strategy. Specifically, an active caching mechanism is employed at the RSU layer, where the service caching priority is determined as a weighted sum of the service criticality value predicted by the GGRN model and the service invocation frequency. This approach ensures that critical and frequent tasks are able to access the required services in a timely manner and avoids making misleading decisions in extreme cases (i.e., caching only critical or frequent services). Moreover, when the agent decides to execute a task at the vehicle and if the RSU does not caching the service needed and sufficient resources are available, the vehicle can generate a customized service at the RSU instead of frequently requesting remote services from the cloud. Once the task is completed, the vehicle layer uses the Least Recently Used (LRU)-based reactive caching strategy to determine whether to caching the service generated by the RSU, thereby reducing the frequency of requesting remote cloud services and fully utilizing local resources.

1) *State Space:* The state space is defined as S . At time slot τ , the agent observes the state $S(\tau)$, which consists of seven

components, expressed as:

$$S(\tau) = \{L(\tau), U(\tau), D(\tau), G(\tau), K(\tau), V(\tau)\} \quad (33)$$

where $L(\tau)$ represents all applications' information and task attributes currently available at time slot τ . $U(\tau)$ is the list of service criticality values obtained from Algorithm 1. $D(\tau)$ and $G(\tau)$ represent the RSU caching state vector and the vehicle caching state vector, respectively. $K(\tau)$ represents the service request frequency list. $V(\tau)$ is the resource threshold vector of each computing node at the time slot τ .

2) *Action Space:* The action space is defined based on the current system state $S(\tau)$, where the agent must make decisions regarding task offloading and RSU caching strategies at each time slot τ . The action at time slot τ , denoted by $A(\tau)$, can be represented as:

$$A(\tau) = \{X_j(\tau), C_{r_m}^k(\tau)\} \quad (34)$$

where $X_j(\tau)$ represents the task offloading decision, where $j \in \{1, 2, \dots, M + N + 1\}$ denotes the offloading nodes, including M RSUs, N vehicles, and one cloud node. $C_{r_m}^k(\tau)$ represents the RSU service caching decision, where r_m denotes the m -th RSU, and \hat{k} denotes the service index.

3) *Reward Function:* The reward function $R(\tau)$ represents the immediate reward obtained for selecting action $A(\tau)$ given the state $S(\tau)$. In our model, the reward is a weighted sum that considers delay, energy consumption, caching hit rate, and application success rate. Specifically, $I_j(\tau)$ encourages caching decisions that improve the cache hit rate at RSUs and vehicles, reducing redundant service retrievals. $Z_j(\tau)$ imposes penalties for both execution delay and energy consumption, guiding the agent toward efficient offloading decisions. $D_j(\tau)$ promotes timely task completion by rewarding successful executions within local deadline constraints. The reward function is formulated as:

$$R(\tau) = \sum_{j=1}^{M+N} \omega_1 I_j(\tau) - \sum_{j=1}^{M+N+1} \omega_2 Z_j(\tau) + \sum_{j=1}^{M+N+1} \omega_3 D_j(\tau) \quad (35)$$

where $I_j(\tau)$ represents the reward associated with the caching hit rate of RSUs and vehicles, $Z_j(\tau)$ denotes the penalty related to delay and energy consumption, and $D_j(\tau)$ represents the reward for the application success rate. ω_1 , ω_2 , and ω_3 are weights used to balance the importance of these different metrics.

4) *Training Process:* At the beginning of each training episode, the algorithm initializes the environment to obtain the initial state S_0 , which is then normalized. The agent selects an action $A(\tau)$ based on the current state using the policy network and executes the action in the environment. Specifically, at the end of time slot $\tau - 1$, task execution data within time window \hat{T} is collected, and the service criticality value in time slot τ is predicted according to the steps in Algorithm 1. These criticality values form part of the state vector in the state space $S(\tau)$.

At the beginning of each time slot τ , the agent first executes the RSU-layer active caching strategy by weighting the service criticality value $y_{\tau}^{pred}(s_{\hat{k}})$ of each service $s_{\hat{k}} \in \hat{S}$, predicted in time slot $\tau - 1$, with the invocation frequency $\hat{f}_{\tau-1}(s_{\hat{k}})$ of each

service $s_k \in \tilde{S}$ recorded in time slot $\tau - 1$, to determine the service caching priority:

$$Prio(s_k) = \lambda \cdot y_{\tau}^{pred}(s_k) + (1 - \lambda) \cdot \tilde{f}_{\tau-1}(s_k) \quad (36)$$

where $\lambda \in [0, 1]$ is the weight factor balancing criticality and frequency. Then, based on this priority, the agent updates RSU caching state in two stages within each time slot τ . First, it checks for stale services by comparing the unique identifier of each newly predicted high-priority service against those already cached. If a matching identifier is detected, the existing version in the cache is deemed obsolete and removed regardless of the cache occupancy to prevent lingering outdated services. Next, if an RSU's caching storage is fully occupied, the service with the lowest caching priority is evicted to cache services with higher priorities, ensuring efficient caching resource utilization.

After updating the RSU caching state, the agent proceeds with the task offloading decision. For all tasks in all applications within time slot τ , assign global task priorities and set reasonable local time constraints according to the steps in Algorithm 2. During the task offloading process, all tasks are scheduled based on their global priorities. Additionally, during training, if the total delay exceeds its assigned local time constraint, a penalty is applied to guide the agent in making reasonable offloading decisions. When a task is offloaded to a vehicle, the required service can be obtained through the following methods: from the RSU caching, generated by RSU customized services, or by requesting from the cloud, depending on the current caching status, network status, and task requirements. If the agent selects a vehicle as the execution node and the required service is not cached in the RSU layer but sufficient resources are available, the vehicle requests the RSU to generate a customized service. Upon task computing completion, the vehicle layer employs a reactive caching strategy based on the LRU policy, deciding whether to cache the generated service locally. If the vehicle cache is already fully occupied, the least recently used service is evicted to accommodate the newly generated service.

This process generates the next state $S(\tau + 1)$, the reward $R(\tau)$, and the completion flag *done*. The new state $S(\tau + 1)$ is normalized, and the interaction data $(S(\tau), A(\tau), R(\tau), S(\tau + 1))$ is stored in the experience replay buffer. When the buffer reaches capacity, a batch $B = (S(\tau), A(\tau), R(\tau), S(\tau + 1))$ is sampled to update the policy and value networks accordingly. The detailed steps of the proposed COHCTO algorithm are outlined in Algorithm 3. More details are provided in the supplementary file for the relevant analyses of the policy network and the value network.

V. SIMULATION RESULTS

This section first provides an overview of the simulation environment setup. Second, three sets of ablation experiments are conducted to analyze the impact of each contribution on overall performance in depth. Subsequently, the performance trends of the COHCTO algorithm are evaluated under different parameter settings. Finally, a comparison between the COHCTO algorithm and other algorithms is made to verify the superiority

Algorithm 3: PPO-Driven Hierarchical Active and Reactive Service Caching and Task Offloading Cooperative Algorithm (COHCTO).

```

1: Input: Environment settings for RSUs, vehicles, cloud
2: Output: Trained policy and value networks ( $\theta$  and  $\Phi$ )
3: Initialize: Policy and value networks parameters  $\theta$  and  $\Phi$ , Replay buffer  $B$ , GGRN model with pre-trained weights
4: for each episode  $u = 1, 2, \dots, U$  do
5:   Initialize system state  $S_0$ 
6:   for each time slot  $\tau \in \tilde{T}$  do
7:     Normalize state  $S(\tau)$ 
8:     Generate action  $A(\tau)$  from policy  $\pi_{\theta}(A|S(\tau))$ 
9:     Execute the joint action  $A(\tau)$  to obtain the immediate reward  $R(\tau)$  according to (35)
10:    Observe next state  $S(\tau + 1)$ 
11:    Store transition  $(S(\tau), A(\tau), R(\tau), S(\tau + 1))$  in memory buffer  $B$ 
12:  end for
13:  for epoch  $e = 1, 2, \dots, \mathcal{E}$  do
14:    Shuffle data and sample mini-batches from  $B$ 
15:    Compute advantage estimates  $\hat{A}_t$  using GAE
16:    Update  $\theta$  and  $\Phi$ 
17:  end for
18:  Update old policy parameters  $\theta_{old} \leftarrow \theta$  and  $\Phi_{old} \leftarrow \Phi$ 
19:  Clear buffer  $B$ 
20: end for

```

TABLE II
ALGORITHM AND ENVIRONMENT PARAMETERS

Parameter	Value	Parameter	Value
Time slot T	[0,199]	Number of RSU	3
Actor learning rate	$1e - 4$	Road range	1000 m
Critic learning rate	$3e - 4$	Vehicle speed	[25,50] km/h
Batch size	2048	Task difficulty threshold	5 GHz
Mini batch size	256	Discount factor	0.99
GAE parameter	0.95	Average application deadline	15s
PPO clip coefficient	0.1	Vehicle computing capacity	[2,3] GHz
Entropy coefficient	0.1	Vehicle caching capacity	1 GB
GGRN model learning rate	$2e - 5$	RSU computing capacity	[3,6] GHz
GCN hidden layer dimension	32	RSU caching capacity	6 GB
GRU hidden layer dimension	64	Require CPU cycles $\varphi(t_k)$	[1,10] GHz

of the proposed method. More details are provided in the supplementary file for the related analyses of the impact of RSU failures and the error analysis of the GGRN model.

A. Simulation Setup

In the simulation experiments, this paper constructs a road state simulation environment based on OpenAI Gym using Python 3.9 to simulate a dynamic traffic network consisting of vehicles, RSUs, and a cloud. In addition, the GGRN model was subjected to 5-fold cross validation to ensure the generalization ability of the model, with evaluation metrics including MSE, MAE, and R^2 . Table II outlines details of the main parameters used in the simulation experiments. More details are provided in the supplementary file for the weights of each part of the reward function and key parameter settings of the GGRN model.

B. Evaluation Metrics

We use the following key evaluation metrics to evaluate the effectiveness and robustness of the proposed algorithm:

- 1) *Average delay*: Quantifies the average time from the launch of the application to completion, including transmission, computation, and queuing delays. As shown in Part I of (24).
- 2) *Average energy consumption*: Characterizes overall energy utilization by counting the total energy consumed by each device during task execution and normalizing it with the number of completed tasks. As shown in Part II of (24).
- 3) *Service caching hit rate*: Measures the effective utilization of local and edge caching, which is the proportion of successful caching responses to the total number of requests.
- 4) *Application success rate*: Assesses the proportion of applications completed within the specified period of the total number of applications.

C. Ablation Experiment

In this section, we conducted an ablation experiment to evaluate the impact and contribution of different modules in the COHCTO strategy on system performance by removing or modifying the core mechanisms in the COHCTO strategy proposed in this paper. Specifically, the COHCTO-POP strategy is used as a comparison scheme to evaluate the effectiveness of the service demand criticality prediction model, the COHCTO-NTP strategy is used as a comparison scheme to measure the effectiveness of the heterogeneous-aware topology priority scheduling algorithm, and the COHCTO-NSC strategy is used as a comparison scheme to evaluate the impact of the service cache strategy on the overall system performance. More details on the ablation experiments are given in the supplementary file.

Fig. 3 shows the performance comparison of several schemes under different numbers of vehicles. Fig. 3(a) and (b) show the average delay and energy consumption of various schemes. The proposed COHCTO strategy always maintains stable delay and energy consumption under different vehicle densities. In contrast, other strategies show greater volatility. The COHCTO-POP scheme only caches commonly used services and performs poorly when handling diverse service requirements, resulting in unstable system delay and energy consumption. As the number of vehicles increases, the average delay and energy consumption increase by 12.3% and 21.4%, respectively. The COHCTO-NTP scheme ignores the global task priority and weakens the effective task scheduling when the competition for computing resources intensifies, resulting in an increase of 43.3% and 8.3% in delay and energy consumption, respectively. The COHCTO-NSC scheme does not use service caching and requires a large number of remote cloud requests. As the number of vehicles increases, the average delay and energy consumption increase by 72.5% and 65.5%, respectively.

Fig. 3(c) reflects the changes in the application success rate of different strategies under the change of vehicle number. It can be seen that the COHCTO strategy maintains a stable application success rate throughout the process, while several

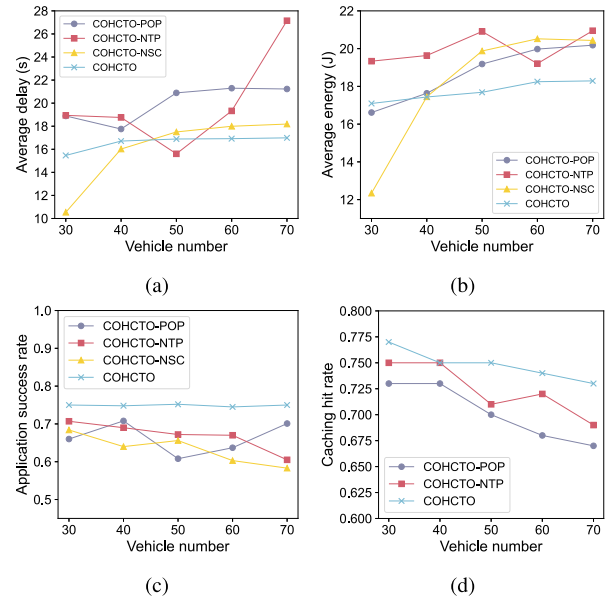


Fig. 3. Comparison of algorithm performance under different numbers of vehicles: (a) average delay; (b) average energy consumption; (c) application success rate; (d) service caching hit rate.

other strategies perform poorly. Since the COHCTO-POP scheme fails to consider the needs of task diversity, it performs poorly under individual conditions and cannot continuously optimize service caching decisions; the COHCTO-NTP scheme does not consider the global task priority, resulting in unbalanced resource allocation when the number of vehicles increases, further affecting the uninstallation success rate; the COHCTO-NSC scheme does not use service caching and relies too much on remote cloud service requests, resulting in communication delays and bandwidth competition, resulting in a low uninstallation success rate, especially when the number of vehicles is 70, the performance is the worst.

Fig. 3(d) shows the trend of caching hit rate with the increase of vehicle number under different strategies. As the number of vehicles increases, the service demand increases, and the caching hit rate of various strategies shows a downward trend. Among them, the caching hit rate for the COHCTO strategy remains high, decreasing from 0.77 to 0.73. However, the COHCTO-POP policy only caches frequent services and thus cannot effectively address dynamic service demands in heterogeneous environments, reducing the caching hit rate. The COHCTO-NTP strategy shows a performance decline under certain conditions, with the caching hit rate dropping from 0.75 to 0.69. This is because the COHCTO-NTP strategy directly schedules tasks based on the DAG sequence, allowing low-priority tasks to occupy system resources more easily. As a result, caching resources cannot be fully utilized, leading to decreased caching hit rate, reduced offloading efficiency, and diminished flexibility in task scheduling.

D. Parameter Analysis

In this section, we evaluate the average delay, average energy consumption, application success rate, and caching hit rate

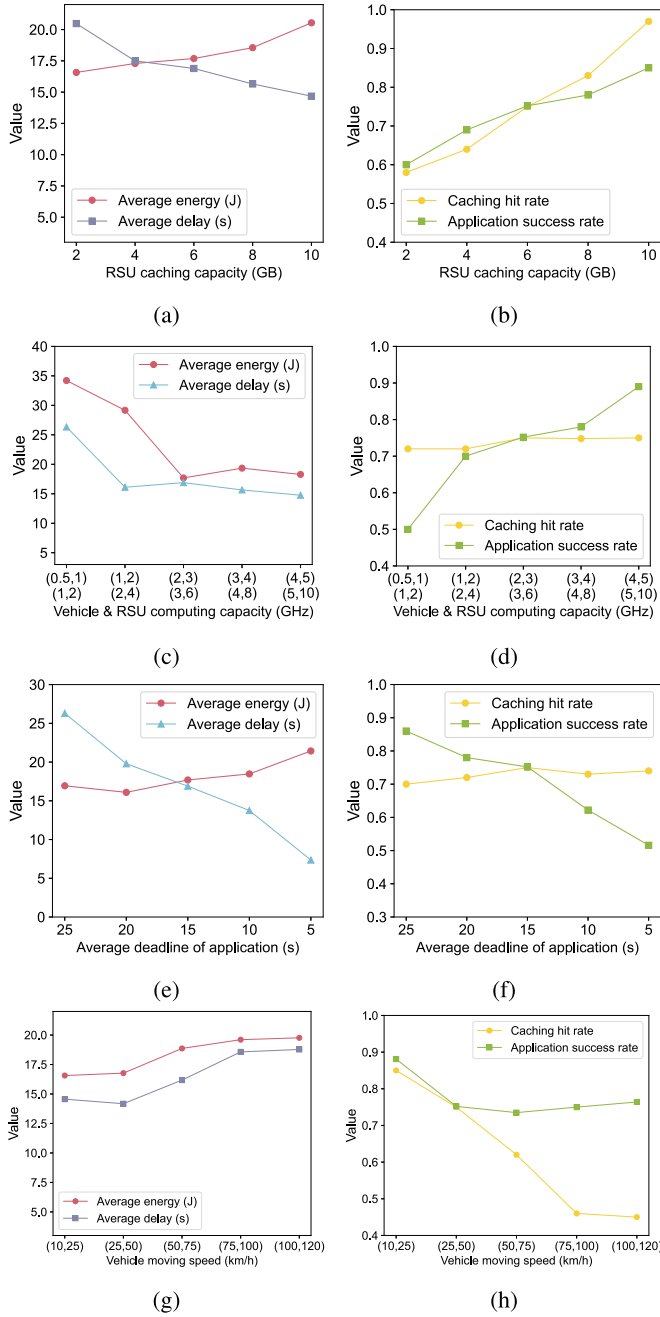


Fig. 4. Algorithm performance under different parameters.

performance of the proposed COHCTO algorithm under various parameters, fixing the number of vehicles to 50, as shown in Fig. 4. The evaluated parameters include the RSU's caching capacity, the computing power of the RSU and vehicles, and the application's average deadline.

1) *The Impact of RSU Caching Capacity:* Fig. 4(a) and (b) show the impact of RSU caching capacity on various indicators. As the RSU caching capacity increases from 2 GB to 10 GB, the average delay decreases to 14.678 s, a decrease of 28% ; the average energy consumption increases to 20.541 J, an increase of 23% ; the caching hit rate increases to 0.97, an increase of 67% ; the application success rate increased to 0.85,

an increase of 41% . It can be seen that with the increase of RSU caching capacity, the system performance significantly improves. This is because a larger caching capacity allows RSU to cache more services, effectively reducing the frequency of requests to the cloud and avoiding additional delays caused by frequent requests to remote cloud services. However, as shown in Fig. 4(a), as the caching capacity increases, more services will be cached in the RSU, which makes more and more tasks choose to be computed in the RSU. This resource competition and preemption reduce the delay while also paying a certain energy consumption.

2) *Impact of Computational Capability:* Fig. 4(c) and (d) show the impact of the computing capabilities of the vehicle and RSU on various indicators. When the computing capabilities of the vehicle and RSU increase from (0.5,1)GHz and (1,2)GHz to (4,5)GHz and (5,10)GHz, respectively, the overall system performance is significantly improved. The average delay is reduced to 14.785 s, a reduction of 43% ; the average energy consumption is reduced to 18.283 J, a reduction of 46% ; the caching hit rate is stable at around 0.75, with no obvious upward or downward trend; the application success rate is increased to 0.89, an increase of 78%, this is because higher computing power enables nodes to process tasks faster, thereby significantly reducing the average delay and improving the application success rate. The results in Fig. 4(c) further show that the algorithm proposed can consistently reduce the delay while ensuring stable optimization of energy consumption under different computing power.

3) *Impact of the Average Application Deadline:* Fig. 4(e) and (f) show the impact of average deadlines on system performance for different applications. As can be seen from the figure, the average delay stays within reasonable limits throughout the process, and the average energy consumption increases by 26% as the deadline gets tighter, generally optimizing the balance between delay and energy consumption over the course of the process. As can be seen in Fig. 4(f), as the average application deadline decreases, the application success rate decreases by 40% as the competition for resources increases and some applications are unable to complete on time. Nevertheless, when the average application deadline is reduced to a minimum value of 5 s, the average delay is still maintained at 7.355 s and the application success rate can reach 0.516, which proves that the algorithm proposed is still advantageous, even when facing the tightest deadlines.

4) *Impact of Vehicle Moving Speed:* Fig. 4(g) and (h) illustrate the impact of vehicle moving speed on the proposed COHCTO algorithm. As vehicle speed increases from (10,25) km/h to (100,120) km/h, both average delay and energy consumption rise, while the caching hit rate declines from 0.85 to 0.45. The application success rate first drops from 0.881 to 0.735 when speed increases to (50,75) km/h, then slightly rebounds to 0.764 at (100,120) km/h. This rebound is attributed to COHCTO's ability to adapt task scheduling via the proposed topology-priority algorithm, compensating for reduced caching performance. Overall, COHCTO performs best in urban and suburban settings with low-to-moderate mobility. In high-speed scenarios, rapidly changing topologies hinder the GGRN model's ability

TABLE III
PERFORMANCE OF DIFFERENT MODELS

Model	Average MSE	Average MAE	Average R^2
GGRN	0.1922	0.1552	0.5288
GRU	0.2033	0.1915	0.4737
LSTM	0.3445	0.3386	0.1114
SimpleRNN	0.3477	0.3407	0.1030

to extract temporal and structural patterns, reducing caching efficiency and increasing reliance on cloud services, which leads to higher delay and energy consumption.

E. Evaluation of Service Criticality Prediction Accuracy Based on GGRN Model

In this section, the proposed GGRN service criticality prediction model is compared with three models: GRU, LSTM, and SimpleRNN.

Table III outlines the performance of several models across three evaluation metrics: MSE, MAE, and R^2 . The GGRN model achieves MSE of 0.1922, MAE of 0.1552, and R^2 of 0.5288, outperforming all other models. These results highlight the GGRN model's significant advantages in capturing complex DAG dependencies in VEC scenarios and accurately predicting task dependencies. The GRU model ranks second to GGRN in capturing DAG dependencies for complex tasks, primarily due to its lack of an explicit mechanism for modeling graph structures. In comparison, the LSTM and SimpleRNN models exhibit larger errors, indicating their limited ability to capture task DAG dependencies effectively. While LSTM can retain long-term dependencies, its inherent complexity makes optimization challenging in intricate VEC scenarios, resulting in underwhelming performance. SimpleRNN struggles to manage long-sequence dependencies due to the gradient vanishing problem, making it the least effective model in this context.

F. Performance and Convergence Analysis of Different Algorithms

To evaluate the performance of COHCTO in the collaborative optimization of task offloading and service cache, we selected P-D3QN [24], SCRACO [25], and CoOR [6] as advanced comparison algorithms. Specifically, the P-D3QN algorithm combines double DQN and dueling DQN to explore the optimal task offloading strategy, a relatively innovative optimization algorithm. Applying it to the application scenario of this paper can evaluate the impact of introducing a heterogeneity-aware topology prioritization algorithm in the task offloading process. The SCRACO algorithm jointly optimizes task offloading, resource allocation, and service caching and has optimization objectives similar to those of this paper. The performance improvement of the proposed COHCTO under the same optimization objective can be evaluated through comparison. The CoOR algorithm involves task offloading and service cache replacement. It combines Gibbs sampling and DRL's iterative algorithm to find the optimal solution. By contrast, the proposed COHCTO's optimization ability under the same problem setting can be

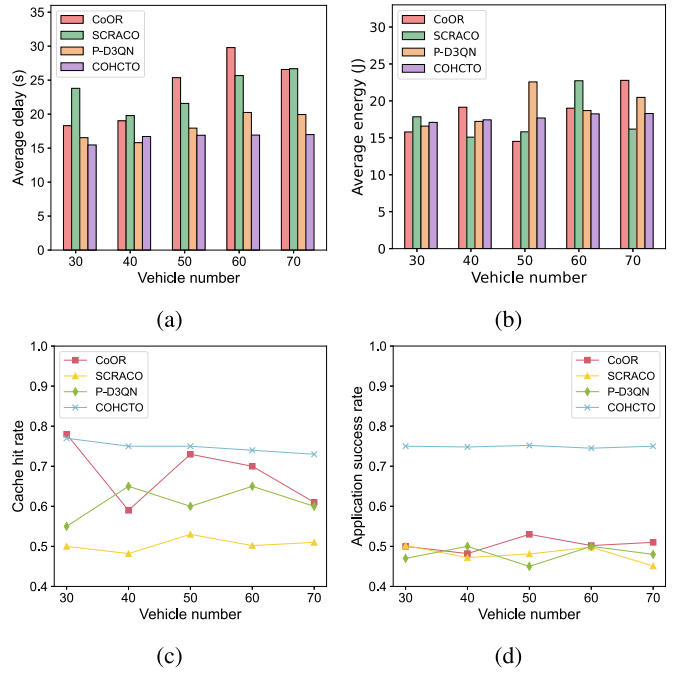


Fig. 5. Performance comparison of different algorithms under different numbers of vehicles: (a) average delay; (b) average energy consumption; (c) caching hit rate; (d) application success rate.

evaluated, especially in improving the impact of cache decisions on task execution efficiency.

Fig. 6 shows the convergence performance of several algorithms in 500 episodes when the number of vehicles is 50. The COHCTO algorithm can better improve the strategy in a dynamically changing environment by using the strategy gradient-based optimization, and its stable update mechanism makes the reward curve smoother and ultimately outperforms the value-function-based DQN variant algorithm. Specifically, SCRACO and CoOR show more obvious fluctuations and slow convergence when facing complex task dependencies and heterogeneous service demands because their updating mechanisms are based on a single Q-value function, which is prone to accumulating estimation bias and makes it difficult to achieve a stable and high level of rewards in complex environments. Although the P-D3QN algorithm shows faster convergence speeds and low volatility, its optimization mechanism is still based on the value function. The lack of global exploration and optimization ability of the strategy, especially in the service caching and task offloading collaborative optimization scenario, makes it difficult to deal with diversified task demands and complex service dependencies at the same time, which leads to a less effective convergence than COHCTO.

Fig. 5 shows the performance comparison of several algorithms under different numbers of vehicles. As shown in Fig. 5(a) and (b), when the average application deadline is 15 s, the COHCTO algorithm shows obvious advantages in average delay and energy consumption, and the changing trend is more stable. The COHCTO algorithm's average delay and energy consumption remain low. Especially when the number of vehicles is 70, the average delay and energy consumption of

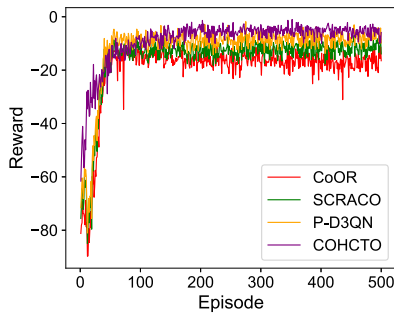


Fig. 6. Reward of different algorithms.

the COHCTO algorithm are 16.992 s and 18.291 J, respectively. It shows that the COHCTO algorithm can effectively cope with the complexity of task offloading and caching management in a dynamic environment, continuously reduce system delay, and optimize resource utilization. The other three algorithms show performance degradation when the number of vehicles reaches 50 and 60. In particular, the delay and energy consumption of the CoOR algorithm increase to 25 s and 14 J, respectively. This is because the DQN series algorithms perform single-step optimization based on local information and lack comprehensive modeling of global dependencies and task priorities, resulting in the inability to achieve optimized scheduling when faced with complex task dependencies.

As can be seen from Fig. 5(c), the caching hit rate of the COHCTO algorithm is relatively high when the number of vehicles increases and is stable at around 0.75 throughout the entire process, which shows that the global perspective and strategy optimization capabilities of the COHCTO algorithm are more effective in the face of dynamic and complex VEC environments. In contrast, the caching hit rate of the CoOR algorithm shows large fluctuations. The caching hit rate drops significantly from 0.78 to 0.59, especially when the number of vehicles reaches 40; this is mainly because the DQN strategy cannot comprehensively consider the dependencies and service requirements between tasks when facing a complex VEC environment, resulting in the low efficiency of the caching strategy. Although the SCRACO and P-D3QN algorithms have reduced the volatility of caching hit rates to a certain extent, they still have limitations. Especially for complex service dependency scenarios, their optimization effects are still insufficient.

As shown in Fig. 5(d), the COHCTO algorithm is significantly better than several other DQN-based strategies in terms of application success rate, and the success rate is stable at around 0.70 throughout the process. The reason is that the COHCTO algorithm can continuously learn the status and load of each node, achieve a reasonable allocation of resources through strategy optimization, avoid task offloading failure caused by improper dependency processing, and still effectively complete tasks in multi-vehicle scenarios. For the DQN series of strategies, although its application success rate shows a relatively stable trend when the number of vehicles increases, the overall level is low, only around 0.5. This is because these algorithms make policy choices based more on historical experience and lack in-depth modeling of global dependencies of tasks and dynamic

perception of service priorities, which leads to the inability to make full use of resources for efficient task offloading in complex environments.

VI. CONCLUSION

In this work, we first analyze the significant practical importance of considering the coexistence of service and task dependencies in VEC scenarios. Next, we conduct an in-depth analysis of the challenges posed by this scenario. In response to these challenges, we propose a service demand criticality prediction method based on the GGRN model and a heterogeneous-aware topology priority allocation algorithm. Finally, building on these two methods, we design a PPO-driven layered hybrid service caching and task offloading collaborative optimization algorithm. Simulation results demonstrate that the proposed algorithm exhibits excellent stability under different network conditions and achieves multi-objective collaborative optimization of delay, energy consumption, caching hit rate, and application success rate.

It is worth noting that both service caching and task offloading involve the transmission and storage of sensitive vehicular data, which may be exposed to potential threats when shared across untrusted edge nodes or communication links. In future work, we plan to incorporate lightweight privacy-preserving mechanisms to mitigate such risks. Additionally, we aim to develop adaptive fault-tolerance and reliability-aware caching strategies to address the performance degradation caused by RSU failures. These enhancements will further strengthen the robustness and practical deployability of the COHCTO framework in real-world, security-sensitive vehicular environments.

REFERENCES

- [1] H. Qian et al., "Collaborative overtaking strategy for enhancing overall effectiveness of mixed connected and connectionless vehicles," *IEEE Trans. Mobile Comput.*, vol. 23, no. 12, pp. 13556–13572, Dec. 2024.
- [2] Y. Chen, F. Zhao, X. Chen, and Y. Wu, "Efficient multi-vehicle task offloading for mobile edge computing in 6G networks," *IEEE Trans. Veh. Technol.*, vol. 71, no. 5, pp. 4584–4595, May 2022.
- [3] R. Meneguette, R. De Grande, J. Ueyama, G. P. R. Filho, and E. Madeira, "Vehicular edge computing: Architecture, resource management, security, and challenges," *ACM Comput. Surveys*, vol. 55, no. 1, pp. 1–46, 2021.
- [4] G. Zhang, S. Zhang, W. Zhang, Z. Shen, and L. Wang, "Joint service caching, computation offloading and resource allocation in mobile edge computing systems," *IEEE Trans. Wireless Commun.*, vol. 20, no. 8, pp. 5288–5300, Aug. 2021.
- [5] G. Zhao, H. Xu, Y. Zhao, C. Qiao, and L. Huang, "Offloading tasks with dependency and service caching in mobile edge computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 11, pp. 2777–2792, Nov. 2021.
- [6] Z. Li, C. Yang, X. Huang, W. Zeng, and S. Xie, "CoOR: Collaborative task offloading and service caching replacement for vehicular edge computing networks," *IEEE Trans. Veh. Technol.*, vol. 72, no. 7, pp. 9676–9681, Jul. 2023.
- [7] X. Fu, B. Tang, F. Guo, and L. Kang, "Priority and dependency-based DAG tasks offloading in fog/edge collaborative environment," in *Proc. IEEE 24th Int. Conf. Comput. Supported Cooperative Work Des.*, 2021, pp. 440–445.
- [8] G. Zhao, H. Xu, Y. Zhao, C. Qiao, and L. Huang, "Offloading dependent tasks in mobile edge computing with service caching," in *Proc. 2020 IEEE Conf. Comput. Commun.*, 2020, pp. 1997–2006.
- [9] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet Things J.*, vol. 3, no. 5, pp. 637–646, Oct. 2016.
- [10] M.-H. Chen, B. Liang, and M. Dong, "Joint offloading and resource allocation for computation and communication in mobile cloud with computing access point," in *Proc. 2017 IEEE Conf. Comput. Commun.*, 2017, pp. 1–9.

- [11] S. Bi, L. Huang, and Y.-J. A. Zhang, "Joint optimization of service caching placement and computation offloading in mobile edge computing systems," *IEEE Trans. Wireless Commun.*, vol. 19, no. 7, pp. 4947–4963, Jul. 2020.
- [12] S. S. Manvi and G. K. Shyam, "Resource management for infrastructure as a service (IaaS) in cloud computing: A survey," *J. Netw. Comput. Appl.*, vol. 41, pp. 424–440, 2014.
- [13] X. Xu, Z. Liu, M. Bilal, S. Vimal, and H. Song, "Computation offloading and service caching for intelligent transportation systems with digital twin," *IEEE Trans. Intell. Transp. Syst.*, vol. 23, no. 11, pp. 20757–20772, Nov. 2022.
- [14] C.-K. Huang and S.-H. Shen, "Enabling service cache in edge clouds," *ACM Trans. Internet Things*, vol. 2, no. 3, pp. 1–24, 2021.
- [15] S.-W. Ko, S. J. Kim, H. Jung, and S. W. Choi, "Computation offloading and service caching for mobile edge computing under personalized service preference," *IEEE Trans. Wireless Commun.*, vol. 21, no. 8, pp. 6568–6583, Aug. 2022.
- [16] T. He, H. Khamfroush, S. Wang, T. La Porta, and S. Stein, "It's hard to share: Joint service placement and request scheduling in edge clouds with sharable and non-sharable resources," in *Proc. IEEE 38th Int. Conf. Distrib. Comput. Syst.*, 2018, pp. 365–375.
- [17] J. Wang, J. Hu, G. Min, W. Zhan, A. Y. Zomaya, and N. Georgalas, "Dependent task offloading for edge computing based on deep reinforcement learning," *IEEE Trans. Comput.*, vol. 71, no. 10, pp. 2449–2461, Oct. 2022.
- [18] H. Xiao, C. Xu, Y. Ma, S. Yang, L. Zhong, and G.-M. Muntean, "Edge intelligence: A computational task offloading scheme for dependent IoT application," *IEEE Trans. Wireless Commun.*, vol. 21, no. 9, pp. 7222–7237, Sep. 2022.
- [19] P. Dass and S. Misra, "DeTTO: Dependency-aware trustworthy task offloading in vehicular IoT," *IEEE Trans. Intell. Transp. Syst.*, vol. 23, no. 12, pp. 24369–24378, Dec. 2022.
- [20] J. Liu, J. Ren, Y. Zhang, X. Peng, Y. Zhang, and Y. Yang, "Efficient dependent task offloading for multiple applications in MEC-cloud system," *IEEE Trans. Mobile Comput.*, vol. 22, no. 4, pp. 2147–2162, Apr. 2023.
- [21] J. Liu, Y. Zhang, J. Ren, and Y. Zhang, "Auction-based dependent task offloading for IoT users in edge clouds," *IEEE Internet Things J.*, vol. 10, no. 6, pp. 4907–4921, Mar. 2023.
- [22] X.-Q. Pham, T.-D. Nguyen, V. Nguyen, and E.-N. Huh, "Joint service caching and task offloading in multi-access edge computing: A QoE-based utility optimization approach," *IEEE Commun. Lett.*, vol. 25, no. 3, pp. 965–969, Mar. 2021.
- [23] Q. Shen, B.-J. Hu, and E. Xia, "Dependency-aware task offloading and service caching in vehicular edge computing," *IEEE Trans. Veh. Technol.*, vol. 71, no. 12, pp. 13182–13197, Dec. 2022.
- [24] J. Chi, X. Zhou, F. Xiao, Y. Lim, and T. Qiu, "Task offloading via prioritized experience-based double dueling DQN in edge-assisted IIoT," *IEEE Trans. Mobile Comput.*, vol. 23, no. 12, pp. 14575–14591, Dec. 2024.
- [25] L. Wang and G. Zhang, "Joint service caching, resource allocation and computation offloading in three-tier cooperative mobile edge computing system," *IEEE Trans. Netw. Sci. Eng.*, vol. 10, no. 6, pp. 3343–3353, Nov./Dec. 2023.



Liang Zhao (Member, IEEE) received the PhD degree from the School of Computing, Edinburgh Napier University, in 2011. He is a professor with Shenyang Aerospace University, China. Before joining Shenyang Aerospace University, he worked as associate senior researcher with Hitachi (China) Research and Development Corporation from 2012 to 2014. He is also a JSPS invitational fellow (2023) and a visiting professor with the University of Electro-Communications, Japan. He was listed as Top 2% of scientists in the world by Stanford University (2022 and 2023). His research interests include ITS, VANET, WMN and SDN. He has published more than 150 articles. He served as the chair of several international conferences and workshops, including 2022 IEEE BigDataSE (Steering co-chair), 2021 IEEE TrustCom (program co-chair), 2019 IEEE IUCC (program co-chair), and 2018–2022 NGDN workshop (founder). He is associate editor of the *Frontiers in Communications and Networking* and *Journal of Circuits Systems and Computers*. He is/has been a guest editor of the *IEEE Transactions on Network Science and Engineering*, *Springer Journal of Computing*, etc. He was the recipient of the Best/Outstanding Paper Awards at 2013 ACM MoMM, 2015 IEEE IUCC, 2020 IEEE ISPA, 2022 IEEE EUC, and 2023 IEEE SustainCom.



Lu Sun received the BS degree from the Liaoning Institute of Science and Technology. She is currently working toward the MS degree with Shenyang Aerospace University. Her research interests mainly include vehicle edge computing.



Ammar Hawbani received the BS degree in computer software and theory and the MS and PhD degrees from the University of Science and Technology of China (USTC), Hefei, China, in 2009, 2012, and 2016. Following his PhD completion, he served as a postdoctoral researcher with the School of Computer Science and Technology, USTC, from 2016 to 2019. He is a full professor with the School of Computer Science, Shenyang Aerospace University. Subsequently, he worked as an associate researcher with the School of Computer Science and Technology, USTC, from 2019 to 2023. His research interests include IoT, WSNs, WBANs, VANETs, and SDN.



Zhi Liu (Senior Member, IEEE) received the PhD degree in informatics from the National Institute of Informatics, Tokyo, Japan, in 2014. He is currently an associate professor with the University of Electro-Communications, Tokyo. His research interests include video network transmission and mobile-edge computing. He is currently an Editorial Board member of the *IEEE Network*.



Xiongyan Tang (Senior Member, IEEE) received the PhD degree in telecom engineering from the Beijing University of Posts and Telecommunications, in 1994. He is the chief scientist of China Unicom Research Institute, the vice dean of China Unicom Research Institute, the state candidate of Millions of Talents Project in the New Century, a member of the Telecom Technology Committee of Ministry of Industry and Information Technology. From 1994 to 1997, he conducted research of high-speed optical communications in Singapore and Germany. Since 1998, he has been working on technology management in telecom operators in China. He is also a professor with the Beijing University of Posts and Telecommunications. He has published more than 150 technical papers. His professional fields include broadband communications, optical transmission, IP networks, SDN/NFV, telecom Big Data, new generation networks, and Internet of Things.



Lexi Xu (Senior Member, IEEE) received the MS degree from the Beijing University of Posts and Telecommunications, Beijing, China, in 2009, and the PhD degree from the Queen Mary University of London, London, U.K., in 2013. He is a senior engineer with the Research Institute, China United Network Communications Corporation. He is also a China Unicom delegate in ITU, ETSI, 3GPP, and CCSA. His research interests include Big Data, self-organizing networks, satellite systems, and radio resource management in wireless systems.