Les Cahiers du GERAD ISSN: 0711–2440

Transformer-based representation for chained DAG task offloading under service constraints in edge computing networks

M. R. Golzari Oskoui, B. Sansò

G–2026–33

June 2026

GERAD HEC Montréal
3000, chemin de la Côte-Sainte-Catherine
Montréal (Québec) Canada H3T 2A7

Tél. : 514 340-6053
Téléc. : 514 340-5665
info@gerad.ca
www.gerad.ca

Transformer-based representation for chained DAG task offloading under service constraints in edge computing networks

Mohammad Reza Golzari Oskoui

Brunilde Sansò

GERAD & Department of Electrical Engineering, Polytechnique Montréal, Montréal (Qc), Canada, H3T 1J4

reza.golzari@polymtl.ca
brunilde.sanso@polymtl.ca

June 2026
Les Cahiers du GERAD
G–2026–33

Abstract : Task offloading in edge computing becomes significantly more challenging when applications are modeled as Directed Acyclic Graphs (DAGs), where task dependencies exist within each instance and extend across consecutive instances, chaining them over time in a mobile environment. In addition, each task requires a specific service, but edge servers can provide only a limited subset of services. As the spatial distribution of service demand changes with user movement, the cached services must be updated accordingly, which in turn affects the offloading decisions that depend on service availability. This paper proposes a dependency-aware representation learning framework for task offloading in such environments. A Transformer encoder is pretrained offline using structural supervision signals derived from DAG properties, such as critical-path workload and slack, to capture task importance and dependency structure. The learned embeddings are integrated with system-level features to construct the state of a Double Deep Q-Network (DDQN) agent deployed at each access gateway. Ablation studies show that task embedding and service availability are the most influential components in offloading decisions, and that their effectiveness depends on complementary system information. Simulation results demonstrate that the proposed method consistently reduces application completion time across varying CPU frequencies, task workloads, network scales, and background server loads, outperforming DDQN baselines with handcrafted features and heuristic methods.

Keywords : Edge computing; task offloading; service caching; reinforcement learning; transformer

Acknowledgements: We kindly acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC) under Grant DG 05734. The authors used a generative AI assistant for language editing and to improve the clarity of the manuscript; all technical content and conclusions are the authors' own, and the authors reviewed and take full responsibility for the final text.

1 Introduction

Edge computing supports latency-sensitive applications by processing data closer to users [17], which significantly reduces delay compared to centralized cloud infrastructures and makes it well suited for real-time applications [5]. Emerging applications such as augmented reality, automated vehicles, face recognition, virtual reality, and industrial IoT require substantial computation with low latency [8]. Many are no longer composed of independent tasks but structured as Directed Acyclic Graphs (DAGs), where tasks are interdependent and one task's output feeds another [19], so execution must respect the DAG's precedence constraints. Moreover, the output of one DAG instance often feeds the next, which chains consecutive instances over time. We address this chained DAG setting, where offloading an instance must account for its dependency on the previous one.

Beyond dependencies, each task may require a specific software service for execution. To illustrate the interaction between task dependencies, offloading decisions, and service requirements, Fig. 1 presents a logistics management application composed of six interdependent tasks arranged as a DAG. Each task requires a service that must be present at its executing cloudlet, and since each cloudlet hosts only a subset of services, task placement is constrained by where its service is cached. When dependent tasks run on different cloudlets, their intermediate outputs are transferred across the backhaul, as shown by the data-transfer flow. Task placement, service availability, and the resulting inter-cloudlet communication jointly determine the application completion time.

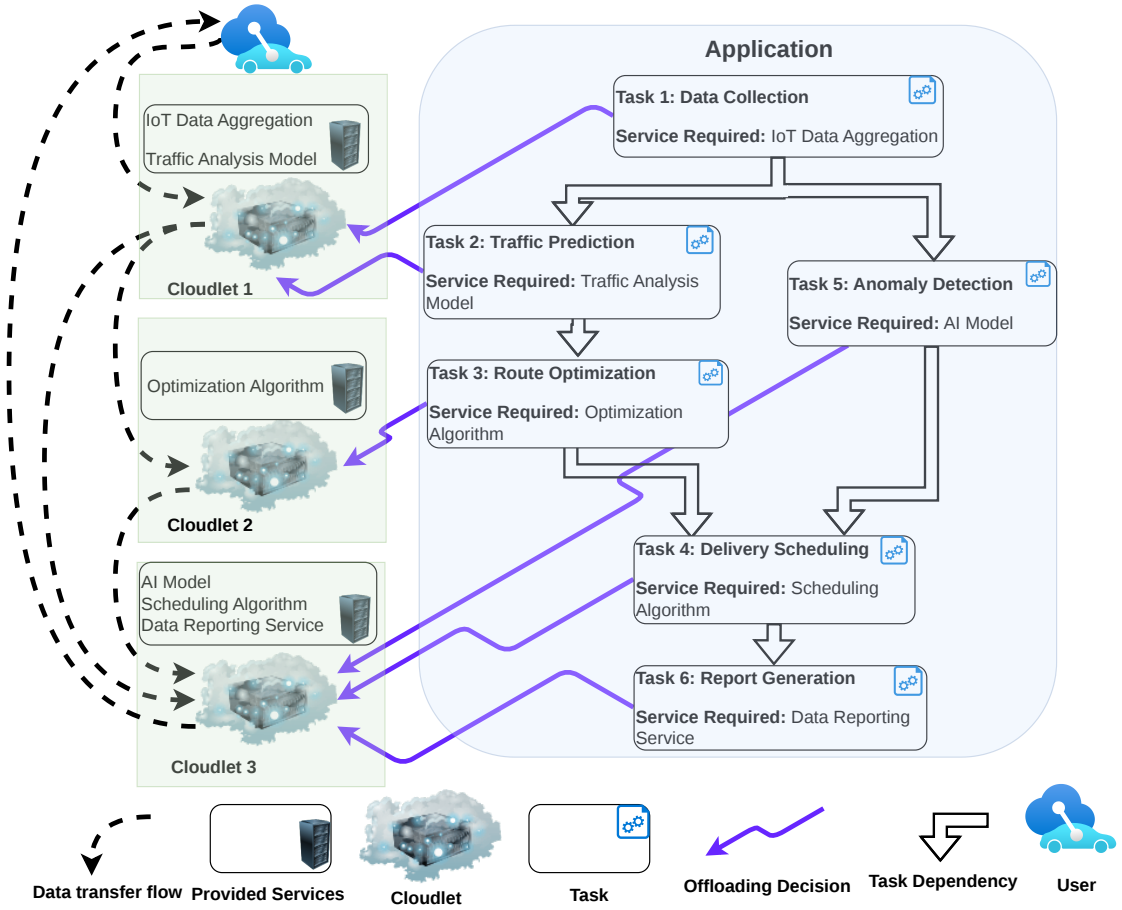


Figure 1: Illustration of a DAG-based application in an edge computing environment. Solid arrows represent task offloading decisions; dashed arrows indicate the resulting data transfer flow between cloudlets.

The coupling between task placement and service availability gives rise to two decisions that must be aligned: offloading (where each task executes) and caching (which services are activated at each cloudlet). Since services cached at one cloudlet can be retrieved by another over the backhaul with different delays, caching is non-trivial and depends strongly on service demand patterns and inter-cloudlet retrieval delays.

User mobility further complicates these decisions, since as users move, the cloudlet they connect to, the corresponding communication latencies, and the spatial distribution of service demand continuously change. Evaluating all of these factors for multiple mobile users with inter-cloudlet interactions results in a high-dimensional, combinatorial optimization problem. To address this challenge, we propose a task offloading framework with coordinated service caching for DAG-based applications in cloudlet networks. Because service caching requires coordination among cloudlets and depends on the service demand the offloading policy induces, it is handled as a separate centralized decision, decoupled from the per-task offloading problem. For the offloading decision, given its high dimensionality and dynamic nature, we adopt a deep reinforcement learning (DRL) approach that learns offloading policies directly from system observations. A central question for the DRL agent is how to represent the DAG. Per-task features such as workload, input size, and service requirements capture local properties but not the global structural role of each task, a key factor in completion time. A Transformer encoder suits this setting: self-attention provides direct pairwise interactions between all tasks, while depth- and height-based positional encodings inject DAG-specific directional information about each task's position relative to the source and sink. The Transformer is pretrained offline on structural targets computed from the DAG, such as critical-path workload and slack, producing task embeddings that capture each task's role. These embeddings are used by the DDQN agent, which combines them with observable system features to make sequential offloading decisions.

The objective of this paper is to minimize the average completion time of repeated DAG-based applications in dynamic edge networks. Unlike prior work that considers dependency-aware offloading within a single DAG instance [4], we model chained DAG execution across consecutive instances and learn dependency-aware task representations. The main contributions are summarized as follows:

- We study task offloading with coordinated service caching for repeated DAG-based applications in cloudlet networks that account for intra- and inter-DAG dependencies, mobility, backhaul transfer, service retrieval, and dynamic cloudlet conditions.
- We develop a Transformer-based task representation model pretrained offline using structural DAG supervision signals, including workload accumulation, ancestor/descendant information, and critical-path slack.
- We design a gateway-level DDQN offloading framework that combines learned task embeddings with service availability, predecessor-data locations, mobility, queueing state, and cloudlet processing capacity.
- We incorporate coordinated service caching as a supporting system component and solve the resulting cache-placement subproblem using linear relaxation followed by rounding.
- We evaluate the framework through ablation and sensitivity analyses, showing the impact of learned task representations and system-state features on application completion time.

2 Literature review

Existing studies adopt different modeling assumptions, which we organize by task modeling, network configuration, mobility, and decision-making mechanism to highlight key differences and research gaps.

Regarding the task model, many studies simplify the decision process by considering independent service requests or content-level caching rather than DAG-structured applications [7, 10–12, 14, 21]. Tang et al. [20] consider a sequence of dependent tasks in vehicular edge computing, but do not model general DAG structures or service caching. To capture richer task dependencies, [3, 6, 19] adopt

DAG-based models. However, [3, 6] focus mainly on computational dependencies without explicitly modeling service requirements and their interaction with execution. Although [19] jointly considers DAG-based offloading and service caching, it treats DAG instances independently and does not model inter-DAG dependencies across consecutive time slots. In practice, the output of one DAG instance may be required for subsequent ones, forming a chained DAG that couples instances over time. This inter-DAG coupling, together with service constraints, is not adequately captured in existing studies.

Regarding network configuration and mobility, [21] considers a vehicular fog framework with road-side unit (RSU) coordination and road-based mobility, while [20] studies an RSU-assisted architecture with exponential sojourn times. [10] assumes a single base-station edge server with traffic-flow mobility. Other studies incorporate mobility-aware network or caching models, including random-waypoint mobility in collaborative edge computing [3], mobility-aware online DAG offloading in vehicular edge computing [6], Markov-based mobility in FL-assisted map caching [12], discrete-jump mobility in content caching [14], and trace-driven vehicular fog resource allocation [11]. Although topology and mobility are modeled, prior studies do not capture how mobility affects inter-DAG data transfer and service availability across cloudlets, leaving the interaction between network dynamics, task dependencies, and service caching underexplored.

Regarding decision-making, DRL-based approaches address offloading under dynamic conditions, targeting delay, energy, and joint resource allocation, including cost-aware Bayesian frameworks [20], multi-agent resource allocation [10], and digital twin-assisted offloading [3]. Graph-based neural models such as Graph Attention Networks (GATs) and heterogeneous GATs have also been used to derive dependency-aware task representations in DAG applications [2, 25]. These approaches encode task dependencies through graph message passing and integrate the resulting representations into DRL-based offloading policies. However, existing learning-based methods mainly optimize offloading within a single application instance, while the role of explicit structural supervision for task representation learning, and its interaction with service availability and chained DAG execution, remains insufficiently explored.

These gaps motivate a unified treatment of learned dependency-aware task representations, service-aware offloading, coordinated service caching, and chained DAG execution across consecutive instances, which is the focus of this paper.

3 System model

Table 1: Summary of key notations.

Symbol	Description
System Parameters	
$\mathcal{M}, \mathcal{N}, \mathcal{Q}$	Sets of M users, N cloudlets, Q services
$\mathcal{V}_m$	Tasks of user m
$\mathcal{V}_{m,i}^{pred}, \mathcal{V}_{m,i}^{succ}$	Predecessor/successor sets of $v_{m,i}$
$w_{m,i}, l_{m,i}, o_{m,i}, z_{m,i}$	CPU cycles, input, output, service of $v_{m,i}$
$W^{\max}, L^{\max}, O^{\max}$	Max CPU cycles, input, output
$g_{m,t}$	Access gateway of user m, slot t
$f_{n,t}$	Effective CPU frequency of cloudlet n
$w_{n,t}^{\text{queue}}$	Queued workload of cloudlet n
$\delta_{n,n'}(l)$	Backhaul delay for size l, $n \to n'$
b_q, r_n	Service q size; cloud rate of cloudlet n
$C_n^{\max}$	Cache capacity of cloudlet n
$T_{m,i,n,t}$	Completion time of $v_{m,i}$ on cloudlet n
Decision Variables	
$x_{m,i,n,t}$	1 if $v_{m,i}$ offloaded to cloudlet n
$c_{q,n,t}$	1 if service q cached on cloudlet n

The system consists of M mobile users, such as smartphones or vehicles, represented by the set $\mathcal{M} = \{1, \ldots, M\}$. These users move dynamically within the environment while continuously executing applications. Due to their limited computational resources, user devices cannot efficiently handle the processing demands of these applications. To ensure low-latency execution, the system leverages N cloudlet servers, denoted by the set $\mathcal{N} = \{1, \ldots, N\}$, which are statically deployed at fixed locations across the environment. Task execution may require software services drawn from a set of Q available services $\mathcal{Q} = \{1, \ldots, Q\}$. Time is divided into discrete slots of duration Δ, where each slot corresponds to the execution interval of one DAG application instance. Each user m connects wirelessly to one cloudlet in each slot, referred to as its access gateway $g_{m,t}$, and this association changes between slots through handover events as the user moves. Movement is represented by the sequence of gateway associations rather than exact coordinates. Each user knows its destination, and the mobility management layer informs each gateway of upcoming handovers. A user stays on one gateway within a slot; handovers occur between slots.

3.1 Task model

Each user $m \in \mathcal{M}$ continuously executes a sequence of identical DAG applications that form a repeated DAG chain. Each DAG instance is represented by $G_m = (\mathcal{V}_m, E_m)$, where $\mathcal{V}_m = \{v_{m,1}, \ldots, v_{m,|\mathcal{V}_m|}\}$ is the set of tasks indexed in topological order, with $v_{m,1}$ as the entry task and $v_{m,|\mathcal{V}_m|}$ as the sink task. The set $E_m \subseteq \mathcal{V}_m \times \mathcal{V}_m$ represents the directed edges encoding task dependencies. Each DAG instance has a fixed structure but is executed with new input data at each time slot t.

Each task $v_{m,i} \in \mathcal{V}_m$ is described by the tuple

$$v_{m,i} = (w_{m,i}, l_{m,i}, o_{m,i}, z_{m,i}), \tag{1}$$

where $w_{m,i}$ is the required CPU cycles, $l_{m,i}$ is the input data size required from the user device (which may be zero for tasks that do not require user input), $o_{m,i}$ is the output data size generated by the task, and $z_{m,i}$ denotes the required service.

The first task $v_{m,1}$ (entry task) receives input $l_{m,1}$ from the user device as well as state data $o_{m,|\mathcal{V}_m|}$ from the sink task of the previous DAG instance. The input data is transmitted through the user's access gateway $g_{m,t}$. The sink task $v_{m,|\mathcal{V}_m|}$ is the final task in the DAG, whose output is transferred to the cloudlet executing the entry task of the next instance in slot $t+1$, which forms an inter-DAG dependency across consecutive time slots. It also generates a lightweight control signal for the user; however, its downlink latency is negligible and is therefore ignored.

Intermediate tasks $v_{m,i}$ receive the outputs of their predecessor tasks and, when applicable, input data from the user device. Each edge $(v_{m,j}, v_{m,i}) \in E_m$ indicates that task $v_{m,i}$ cannot begin execution until task $v_{m,j}$ has completed and its output is available at the hosting cloudlet of $v_{m,i}$. The predecessor and successor sets of task $v_{m,i}$ are

$$\mathcal{V}_{m,i}^{pred} = \{v_{m,j} \mid (v_{m,j}, v_{m,i}) \in E_m\}, \tag{2}$$

$$\mathcal{V}_{m,i}^{succ} = \{v_{m,j} \mid (v_{m,i}, v_{m,j}) \in E_m\}. \tag{3}$$

3.2 Cloudlet server model

The cloudlet servers provide computational resources at the network edge. Unlike static edge models, each cloudlet's processing capability and backlog vary over time due to resource contention and background activity, and are therefore modeled as time-varying. For each cloudlet $n \in \mathcal{N}$, let f_n^{base} and w_n^{base} denote its nominal CPU frequency and nominal background workload in CPU cycles. At time slot t, the effective CPU frequency and background workload are given by

$$f_{n,t} = f_n^{\text{base}}\, \phi_{n,t}, \tag{4}$$

$$w_{n,t}^{\text{queue}} = w_n^{\text{base}}\,\psi_{n,t}, \tag{5}$$

where $\phi_{n,t}$ and $\psi_{n,t}$ are bounded stochastic state variables representing temporal fluctuations in available processing speed and background workload, respectively. These variables evolve according to first-order stochastic updates:

$$\phi_{n,t+1} = \text{clip}\Big(\rho_f \phi_{n,t} + (1-\rho_f) + \varepsilon_{n,t}^{f},\, \phi_{\min},\, \phi_{\max}\Big)\,, \tag{6}$$

$$\psi_{n,t+1} = \text{clip}\big(\rho_w \psi_{n,t} + (1-\rho_w) + \varepsilon_{n,t}^{w},\, \psi_{\min},\, \psi_{\max}\big)\,, \tag{7}$$

where $\rho_f, \rho_w \in [0,1)$ control temporal correlation, $\varepsilon_{n,t}^{f} \sim \mathcal{N}(0,\sigma_f^2)$ and $\varepsilon_{n,t}^{w} \sim \mathcal{N}(0,\sigma_w^2)$ are zero-mean Gaussian perturbations, and clip$(\cdot)$ constrains the values to predefined intervals, so that cloudlet conditions change gradually across slots.

Since each cloudlet has limited storage, server n can host only up to $C_n^{\max} < Q$ of the available services. We define a binary variable $c_{q,n,t}$ to indicate whether service q is cached on cloudlet n at time slot t:

$$c_{q,n,t} = \begin{cases} 1, & \text{if service } q \text{ is cached on cloudlet } n, \\ 0, & \text{otherwise.} \end{cases} \tag{8}$$

The access gateway $g_{m,t}$ is the entry point for the user's input data, but task execution is not restricted to it. The binary variable $x_{m,i,n,t}$ indicates whether task $v_{m,i}$ is offloaded to cloudlet n in slot t; this decision is made by the user's access gateway each slot.

3.3 Communication model

Communications are of two types: wireless access between users and gateways, and wired backhaul between cloudlets. The wireless uplink rate is $r^u_{m,g_{m,t}} = B\log_2(1+\text{SNR}_{m,g_{m,t}})$, where B is the channel bandwidth and $\text{SNR}_{m,g_{m,t}}$ is modeled as a truncated Gaussian with mean μ_{SNR} and variance σ^2_{SNR}. The uplink carries only the task input to the gateway, and this delay is common to all offloading choices since the gateway is fixed by the user's location.

For the backhaul, the cloudlets are interconnected by a wired backbone modeled as a graph $G_{\mathcal{N}} = (\mathcal{N}, \mathcal{E})$, where $\mathcal{E}$ is the set of physical links. The latency for transferring data of size l from cloudlet n to n' is computed along the minimum-delay path; letting $\Pi_{n,n'}$ denote its links, the backhaul delay is

$$\delta_{n,n'}(l) = \begin{cases} 0, & \text{if } n = n', \\ \sum_{(i,j)\in\Pi_{n,n'}} \dfrac{l}{r^b_{i,j}}, & \text{if a path exists,} \\ \infty, & \text{otherwise.} \end{cases} \tag{9}$$

where $r^b_{i,j}$ is the rate of the backbone link between i and j. Propagation and processing delays at intermediate nodes are neglected as small relative to transmission. Note $\delta_{n,n'}(l)$ is linear in l, i.e., $\delta_{n,n'}(l) = l \cdot \delta_{n,n'}(1)$.

3.4 Application execution model

We measure execution time relative to the start of each slot t. At the start of slot t, all data transfers are initiated, including input and state data from the previous DAG instance. The arrival time of the input data at cloudlet n for task $v_{m,i}$ is

$$\tau^{\text{input}}_{m,i,n,t} = \frac{l_{m,i}}{r^u_{m,g_{m,t}}} + \delta_{g_{m,t},n}(l_{m,i})\,. \tag{10}$$

If task $v_{m,i}$ requires no input data ($l_{m,i} = 0$), then $\tau^{\text{input}}_{m,i,n,t} = 0$. Due to the chained DAG structure, we distinguish the entry task $v_{m,1}$ from subsequent tasks.

The entry task executed on cloudlet n requires two inputs: the input data and the state data from the previous slot. The state data is the output of the sink task $v_{m,|\mathcal{V}_m|}$ from slot $t-1$; letting n' be the cloudlet that executed it, its arrival time is $\tau^{\text{state}}_{m,n,t} = \delta_{n',n}(o_{m,|\mathcal{V}_m|})$. The ready time of the entry task is then

$$T^{\text{ready}}_{m,1,n,t} = \max\Big(\tau^{\text{input}}_{m,1,n,t}, \tau^{\text{state}}_{m,n,t}\Big). \tag{11}$$

For any task $v_{m,i}$ with $i > 1$ executed on cloudlet n, execution begins only after all predecessor outputs and any input data arrive at cloudlet n. Letting $n^*_{m,j}$ denote the cloudlet executing predecessor $v_{m,j}$ (i.e., $x_{m,j,n^*_{m,j},t} = 1$), the arrival time of predecessor outputs at cloudlet n is

$$T^{\text{pred}}_{m,i,n,t} = \max_{v_{m,j} \in \mathcal{V}^{pred}_{m,i}} \Big(T_{m,j,n^*_{m,j},t} + \delta_{n^*_{m,j},n}(o_{m,j})\Big), \tag{12}$$

where $T_{m,j,n^*_{m,j},t}$ is the completion time of predecessor task $v_{m,j}$.

The ready time of task $v_{m,i}$ is therefore

$$T^{\text{ready}}_{m,i,n,t} = \max\Big(T^{\text{pred}}_{m,i,n,t},\ \tau^{\text{input}}_{m,i,n,t}\Big), \quad i > 1. \tag{13}$$

After all required data for task $v_{m,i}$ arrive at cloudlet n, the task experiences three processing-related delays. First, the task incurs a queueing delay $\tau^{\text{queue}}_{m,i,n,t} = w^{\text{queue}}_{n,t}/f_{n,t}$ from the background workload $w^{\text{queue}}_{n,t}$ [19], which we model as queued CPU cycles at the cloudlet. Second, it incurs a computation delay $\tau^{\text{computing}}_{m,i,n,t} = w_{m,i}/f_{n,t}$, determined by its CPU cycles and the cloudlet's effective capacity. Third, execution requires the task's service to be active at cloudlet n. We model each service q as a deployable software package (e.g., a container image) of size b_q; if it is not cached locally, the cloudlet fetches it either from the cloud at rate r_n or from another cloudlet that caches it. The resulting service loading delay is

$$\tau^{\text{service}}_{m,i,n,t} = \underbrace{\big(1 - c_{z_{m,i},n,t}\big)}_{\substack{\text{equals 0 if service} \\ \text{is cached locally}}} \cdot \tau^{\text{fetch}}_{m,i,n,t}, \tag{14}$$

where

$$\tau^{\text{fetch}}_{m,i,n,t} = \min\left(\frac{b_{z_{m,i}}}{r_n},\ \min_{\bar{n}: c_{z_{m,i},\bar{n},t}=1} \delta_{\bar{n},n}\big(b_{z_{m,i}}\big)\right). \tag{15}$$

The completion time of task $v_{m,i}$ on cloudlet n is the sum of the ready time, queueing delay, service loading time, and computation delay:

$$T_{m,i,n,t} = T^{\text{ready}}_{m,i,n,t} + \tau^{\text{queue}}_{m,i,n,t} + \tau^{\text{service}}_{m,i,n,t} + \tau^{\text{computing}}_{m,i,n,t}. \tag{16}$$

The ordering reflects the execution sequence: the task first waits for all predecessor outputs and input data, then queues behind the background workload. Since loading a service before the task is ready would occupy a cache slot and block others, the service is loaded only when the task reaches the front of the queue, after which computation begins. The application completes once the sink task finishes at its selected cloudlet. For each user m at time slot t, the application completion time is defined as

$$\tilde{T}_{m,t} \triangleq \sum_{n \in \mathcal{N}} x_{m,|\mathcal{V}_m|,n,t}\, T_{m,|\mathcal{V}_m|,n,t}. \tag{17}$$

4 Problem formulation

The objective of the system is to minimize the long-term average application completion time experienced by all users over a finite horizon of $T_{\max}$ time slots. The optimization problem is formulated as

$$\min_{\mathbf{X},\mathbf{C}} \quad \frac{1}{T_{\max}} \sum_{t=1}^{T_{\max}} \sum_{m=1}^{M} \tilde{T}_{m,t} \tag{18a}$$

$$\text{s.t.} \quad \sum_{n\in\mathcal{N}} x_{m,i,n,t} = 1 \qquad \forall m \in \mathcal{M},\ \forall i \in \mathcal{V}_m,\ \forall t \tag{18b}$$

$$\sum_{q\in\mathcal{Q}} c_{q,n,t} \le C_n^{\max} \qquad \forall n \in \mathcal{N},\ \forall t \tag{18c}$$

$$x_{m,i,n,t} \in \{0,1\} \qquad \forall m,i,n,t \tag{18d}$$

$$c_{q,n,t} \in \{0,1\} \qquad \forall q,n,t \tag{18e}$$

Here $\mathbf{X} \triangleq \{x_{m,i,n,t}\}$ and $\mathbf{C} \triangleq \{c_{q,n,t}\}$ are the offloading and caching decisions. The objective (18a) minimizes the average completion time across all users and slots. Constraint (18b) assigns each task to exactly one cloudlet, (18c) caps the cached services per cloudlet, and (18d)–(18e) impose binary variables.

Problem (18) is a Mixed-Integer Non-Linear Programming (MINLP) problem and is NP-hard, as assigning dependent tasks to heterogeneous servers is itself NP-hard [22]. The binary variables $x_{m,i,n,t}$ and $c_{q,n,t}$ impose combinatorial structure, and the non-linearity arises from the expressions in (11), (12), (13), (14), and (15).

To see the scale of the problem, note that each task is assigned to one of the N cloudlets, and each cloudlet independently caches any of the Q services, giving a per-slot solution space of $|\Omega| = N^{\sum_{m=1}^{M}|\mathcal{V}_m|} \times 2^{N\cdot Q}$. The search space thus grows exponentially with users, tasks, cloudlets, and services. Combined with the system's dynamics from user mobility and time-varying server conditions, this renders exact methods infeasible. We therefore adopt a DRL approach that learns adaptive policies from observable system parameters without a complete model of the environment.

5 Solution method

To solve the problem, the decision maker needs information about the full DAG structure and the current network state, since the completion time depends on both task dependencies and dynamic network conditions. Moreover, since service hosting is shared across cloudlets, service caching is handled centrally to account for inter-server cooperation. The solution method therefore has three parts: a Transformer encoder is pretrained offline to learn dependency-aware task representations from the DAG; task offloading decisions are then made by DRL using the encoder outputs and network state; and service caching is solved as a decoupled problem via linear relaxation and rounding.

5.1 Transformer-based task representation

We employ a Transformer model [24] to learn task representations that capture structural interactions within the DAG. The Transformer relies on a self-attention mechanism. Self-attention lets each task token attend to all other tasks and incorporate information beyond its immediate neighbors [9], and attention has been used to encode DAG structure directly [13]. The encoder is trained offline on structural supervision signals derived from DAG properties. These signals make the embeddings reflect task importance and dependency-aware execution characteristics. Since the encoder is trained on generic DAG structures independently of any user, we drop the user index m in this subsection. Fig. 2 shows the offline training architecture.

5.1.1 Input representation

We extend prior methods that represent each task by basic attributes such as workload and data sizes [15] with structural features of the DAG. Let $W^{\max}$, $L^{\max}$, and $O^{\max}$ denote the maximum CPU

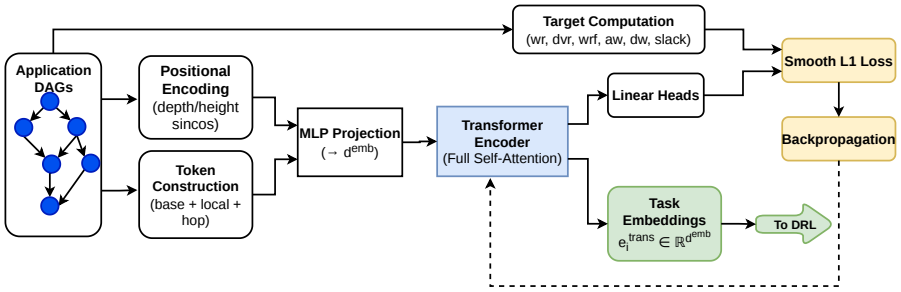


Figure 2: Offline training framework of the Transformer-based task representation. The encoder is trained on six structural supervision signals derived from the DAG. After training, the linear heads are discarded and the encoder produces task embeddings for the DRL offloading agent.

cycles, input size, and output size across all tasks. The token $\mathbf{e}_i^{\text{task}} \in \mathbb{R}^{15}$ is constructed from the feature groups listed in Table 2, where $\mathbb{1}\{\cdot\}$ denotes the indicator function. To encode task positions

Table 2: Token feature groups for task v_i.

Group	Feature	Dim	Description
Base	$w_i/W^{\max}$	3	Norm. CPU cycles
	$l_i/L^{\max}$		Norm. input size
	$o_i/O^{\max}$		Norm. output size
Local	$\lvert\mathcal{V}_i^{pred}\rvert/\lvert\mathcal{V}\rvert$	4	Norm. in-degree
	$\lvert\mathcal{V}_i^{succ}\rvert/\lvert\mathcal{V}\rvert$		Norm. out-degree
	$\mathbb{1}\{i=1\}$		Source indicator
	$\mathbb{1}\{i=\lvert\mathcal{V}\rvert\}$		Sink indicator
Hop data	$\sum_{j\in\mathcal{V}_i^{pred}} o_j/O^{\max}$	4	Total pred. output
	$\max_{j\in\mathcal{V}_i^{pred}} o_j/O^{\max}$		Max pred. output
	$\sum_{j\in\mathcal{V}_i^{succ}} l_j/L^{\max}$		Total succ. input
	$\max_{j\in\mathcal{V}_i^{succ}} l_j/L^{\max}$		Max succ. input
Hop work	$\overline{w}_i^{pred}/W^{\max}$	4	Mean pred. workload
	$\max_{j\in\mathcal{V}_i^{pred}} w_j/W^{\max}$		Max pred. workload
	$\overline{w}_i^{succ}/W^{\max}$		Mean succ. workload
	$\max_{j\in\mathcal{V}_i^{succ}} w_j/W^{\max}$		Max succ. workload
Total		**15**	

within the DAG, we define the depth and height of each task based on longest-path distances. The depth of v_i is defined recursively as

$$\text{depth}_i = \begin{cases} 0, & \text{if } \mathcal{V}_i^{pred} = \emptyset, \\ 1 + \max_{v_j \in \mathcal{V}_i^{pred}} \text{depth}_j, & \text{otherwise}, \end{cases} \tag{19}$$

which corresponds to the longest-path distance from a source task to v_i. Similarly, the height of v_i is defined as

$$\text{height}_i = \begin{cases} 0, & \text{if } \mathcal{V}_i^{succ} = \emptyset, \\ 1 + \max_{v_j \in \mathcal{V}_i^{succ}} \text{height}_j, & \text{otherwise}, \end{cases} \tag{20}$$

representing the longest-path distance from v_i to a sink task. For a positional scalar $p_i \in \{\text{depth}_i, \text{height}_i\}$, we normalize it by the maximum value within the DAG and scale by $\alpha = 10$, yielding $\tilde{p}_i = \alpha \cdot p_i / \max_j p_j$. The positional encoding vector $\mathbf{e}^{\text{pos}}(p_i) \in \mathbb{R}^{d^{\text{pe}}/2}$, where d^{pe} denotes the total positional encoding dimension, is computed using sinusoidal functions [16, 24]:

$$\mathbf{e}^{\text{pos}}(p_i)[2k] = \sin\left(\frac{\tilde{p}_i}{10000^{\frac{2k}{d^{\text{pe}}/2}}}\right), \tag{21}$$

$$\mathbf{e}^{\text{pos}}(p_i)[2k+1] = \cos\left(\frac{\tilde{p}_i}{10000^{\frac{2k}{d^{\text{pe}}/2}}}\right), \tag{22}$$

where $k = 0, \ldots, \frac{d^{\text{pe}}}{4} - 1$ is the dimension index. The final positional encoding is $\mathbf{e}_i^{\text{pos}} = [\mathbf{e}^{\text{pos}}(\text{depth}_i) \| \mathbf{e}^{\text{pos}}(\text{height}_i)] \in \mathbb{R}^{d^{\text{pe}}}$, and the encoder input is $\mathbf{e}_i^{\text{in}} = [\mathbf{e}_i^{\text{task}} \| \mathbf{e}_i^{\text{pos}}] \in \mathbb{R}^{15+d^{\text{pe}}}$.

5.1.2 Encoder architecture

The input tokens are stacked into a matrix $\mathbf{E}^{\text{in}} = [\mathbf{e}_1^{\text{in}}, \ldots, \mathbf{e}_{|\mathcal{V}|}^{\text{in}}]^\top \in \mathbb{R}^{|\mathcal{V}| \times (15+d^{\text{pe}})}$ and projected to d^{emb} dimensions by a multi-layer perceptron (MLP):

$$\mathbf{H}^{(0)} = \text{MLP}(\mathbf{E}^{\text{in}}) = \sigma(\mathbf{E}^{\text{in}}\,\mathbf{W}_1 + \mathbf{b}_1)\,\mathbf{W}_2 + \mathbf{b}_2 \in \mathbb{R}^{|\mathcal{V}| \times d^{\text{emb}}}, \tag{23}$$

where $\sigma(\cdot)$ is a nonlinear activation function specified in Section 6. The projected representations are then processed by pre-norm Transformer encoder layers with multi-head self-attention [24]. At layer λ:

$$\hat{\mathbf{H}}^{(\lambda)} = \mathbf{H}^{(\lambda-1)} + \text{MultiHead}\Big(\text{LN}\Big(\mathbf{H}^{(\lambda-1)}\Big)\Big), \tag{24}$$

$$\mathbf{H}^{(\lambda)} = \hat{\mathbf{H}}^{(\lambda)} + \text{FFN}\Big(\text{LN}\Big(\hat{\mathbf{H}}^{(\lambda)}\Big)\Big), \tag{25}$$

where $\text{LN}(\cdot)$ is layer normalization and $\text{FFN}(\cdot)$ is a two-layer feedforward network with inner dimension $d^{\text{ff}} = 4 \cdot d^{\text{emb}}$. The multi-head self-attention with $\mathbb{H}$ heads is defined as

$$\text{MultiHead}(\mathbf{H}) = \text{Concat}(\text{head}_1, \ldots, \text{head}_{\mathbb{H}})\,\mathbf{W}^O, \tag{26}$$

$$\text{head}_h = \text{softmax}\left(\frac{\mathbf{H}\mathbf{W}_h^Q(\mathbf{H}\mathbf{W}_h^K)^\top}{\sqrt{d^{\text{emb}}/\mathbb{H}}}\right)\mathbf{H}\mathbf{W}_h^V, \tag{27}$$

where $h = 1, \ldots, \mathbb{H}$ is the head index. After all Transformer layers, a final layer normalization is applied to produce $\mathbf{H}^{\text{out}} = \text{LN}(\mathbf{H}^{(\Lambda)}) \in \mathbb{R}^{|\mathcal{V}| \times d^{\text{emb}}}$, where Λ is the number of encoder layers. The per-task embedding $\mathbf{e}_i^{\text{trans}} \in \mathbb{R}^{d^{\text{emb}}}$ is the i-th row of $\mathbf{H}^{\text{out}}$.

5.1.3 Supervision signals and training

The encoder is pretrained offline using structural supervision signals derived from the DAG. All targets are computed under a simplified setting with uniform server capacity ($f_n = 1$) and no queueing. Six targets are defined for each task, capturing workload accumulation, output-data accumulation, ancestor/descendant workload, and structural position within the DAG. In particular, the remaining workload wr_i and forward workload wrf_i, related to the upward and downward ranks in classical DAG scheduling [22], are used as supervision signals. The complete set of targets is summarized in Table 3. Each target is normalized to $[0, 1]$ by dividing by its maximum value across the DAG. For each target $k \in \mathcal{H} = \{\text{wr}, \text{dvr}, \text{wrf}, \text{aw}, \text{dw}, \text{slack}\}$, the prediction is obtained through a linear projection followed by a sigmoid activation:

$$\hat{y}_{k,i} = \text{sigmoid}(\mathbf{p}_k^\top \mathbf{e}_i^{\text{trans}} + \eta_k), \tag{28}$$

where $\mathbf{p}_k \in \mathbb{R}^{d^{\text{emb}}}$ and $\eta_k \in \mathbb{R}$ are learnable parameters. Each prediction is trained using the Smooth L1 loss defined as

$$\ell_{k,i} = \begin{cases} \frac{1}{2}(y_{k,i} - \hat{y}_{k,i})^2, & \text{if } |y_{k,i} - \hat{y}_{k,i}| < 1, \\ |y_{k,i} - \hat{y}_{k,i}| - \frac{1}{2}, & \text{otherwise}, \end{cases} \tag{29}$$

where $y_{k,i}$ denotes the normalized target value computed analytically. The total loss of an application is obtained by summing over all targets and all tasks in the application. The prediction heads are intentionally lightweight so that the encoder is forced to capture the structural properties of the DAG. After training, the prediction heads are discarded, and each cloudlet uses the pretrained encoder to generate task embeddings $\mathbf{e}_i^{\text{trans}} \in \mathbb{R}^{d^{\text{emb}}}$ for offloading decisions.

Table 3: Supervision signals for Transformer pretraining. All targets are normalized to $[0, 1]$ by dividing by their maximum value.

Target	Definition
wr_i	$w_i + \max_{v_j \in \mathcal{V}_i^{succ}} \mathrm{wr}_j$ (w_i at sink). Max cumulative workload from v_i to sink.
dvr_i	$o_i + \max_{v_j \in \mathcal{V}_i^{succ}} \mathrm{dvr}_j$ (o_i at sink). Max cumulative output data from v_i to sink.
wrf_i	$w_i + \max_{v_j \in \mathcal{V}_i^{pred}} \mathrm{wrf}_j$ (w_i at source). Max cumulative workload from source to v_i.
aw_i	$\sum_{v_j \in \mathrm{anc}(i)} w_j$. Ancestor workload, defined as the total workload of all transitive ancestors.
dw_i	$\sum_{v_j \in \mathrm{desc}(i)} w_j$. Descendant workload, defined as the total workload of all transitive descendants.
slack_i	$\max(0,\ \mathrm{wr}^{\max} - (\mathrm{wrf}_i + \mathrm{wr}_i - w_i))$, where $\mathrm{wr}^{\max} = \max_j \mathrm{wr}_j$. Distance from the critical path.

$\mathrm{anc}(i)$, $\mathrm{desc}(i)$: transitive ancestors and descendants of v_i.

5.2 DRL-based task offloading framework

The offloading decision for each task affects its dependent tasks. This creates a tightly coupled and sequential decision process within a time slot and across consecutive time slots. Moreover, server conditions change across time slots. This requires adaptive policies that can generalize across different network states. We therefore model the task offloading process at each access gateway as a Markov Decision Process (MDP), characterized by a state space, an action space, a reward function, and a discount factor γ. A DRL agent at each access gateway determines the execution cloudlet for each task based on the current state observation. For each user m in slot t, the agent at the current gateway $g_{m,t}$ makes all offloading decisions for the DAG instance executed in that slot; since handovers occur only between consecutive slots, a DAG instance is not split across gateways. Each gateway trains its own agent independently from the interactions between its associated users' tasks and the network, while all gateways share the same network architecture and hyperparameters. The components of the MDP are defined as follows.

5.2.1 State space

When task $v_{m,i}$ of user m arrives at its current access gateway $g_{m,t}$ in time slot t, the agent observes the state and makes an offloading decision. The state captures task-level information and observable network-related features. Accordingly, the state at each decision step is defined as

$$\mathbf{s}_{m,i,t} = \big[\mathbf{e}_{m,i}^{\mathrm{trans}} \,\big|\, \mathbf{w}_{m,i} \,\big|\, \mathbf{serv}_{m,i,t} \,\big|\, \mathbf{data}_{m,i,t} \,\big|\, \mathbf{g}_{m,t}^{\mathrm{cur}} \,\big|\, \mathbf{g}_{m,t}^{\mathrm{next}} \,\big|\, \mathbf{queue}_t \,\big|\, \mathbf{freq}_t\big]. \tag{30}$$

Task Embedding: $\mathbf{e}_{m,i}^{\mathrm{trans}} \in \mathbb{R}^{d^{\mathrm{emb}}}$ is the Transformer-based embedding of task $v_{m,i}$ that encodes structural dependencies within the application.

Task Workload: $\mathbf{w}_{m,i} = [w_{m,i}/W^{\max},\ l_{m,i}/L^{\max}] \in \mathbb{R}^2$ encodes the normalized CPU cycles and input size of the current task.

Service Availability: $\mathbf{serv}_{m,i,t} \in \{0,1\}^N$ indicates which cloudlets cache the service required by $v_{m,i}$, with $\mathbf{serv}_{m,i,t}(n) \triangleq c_{z_{m,i},n,t}$, $\forall n \in \mathcal{N}$.

Input-Data Location: $\mathbf{data}_{m,i,t} \in \mathbb{R}^N$ encodes the normalized input data at each cloudlet, originating from already-executed predecessors, input at the current gateway, and (for entry tasks) the previous sink output:

$$\begin{aligned}\mathbf{data}_{m,i,t}(n) \triangleq & \sum_{v_{m,j} \in \mathcal{V}_{m,i}^{pred}} x_{m,j,n,t} \frac{o_{m,j}}{O^{\max}} \\ & + \mathbb{1}\{n = g_{m,t}\} \frac{l_{m,i}}{L^{\max}} \\ & + \mathbb{1}\{i = 1\}\, x_{m,|\mathcal{V}_m|,n,t-1} \frac{o_{m,|\mathcal{V}_m|}}{O^{\max}}.\end{aligned} \tag{31}$$

Mobility Features: $\mathbf{g}^{\text{cur}}_{m,t}(n) \triangleq \mathbb{1}\{n = g_{m,t}\}$ and $\mathbf{g}^{\text{next}}_{m,t}(n) \triangleq \mathbb{1}\{n = g_{m,t+1}\}$ are one-hot vectors for the current and next gateways.

Queue Load: $\mathbf{queue}_t \in \mathbb{R}^N$ encodes the queueing delay at each cloudlet, $\mathbf{queue}_t(n) \triangleq w^{\text{queue}}_{n,t}/f_{n,t}$, $\forall n \in \mathcal{N}$.

Server Frequency: $\mathbf{freq}_t \in \mathbb{R}^N$ encodes the normalized CPU frequency, $\mathbf{freq}_t(n) \triangleq f_{n,t}/\max_{n'} f_{n',t}$, $\forall n \in \mathcal{N}$.

Considering all components, the state dimension is $d^{\text{emb}} + 6N + 2$. The server-side features $\mathbf{queue}_t$, $\mathbf{freq}_t$, and $\mathbf{serv}_{m,i,t}$ are disseminated to all access gateways via lightweight control signals over the backhaul network and are updated at the start of each time slot, so that each gateway has an up-to-date view of the global server state before it makes offloading decisions.

5.2.2 Action space

The action space is defined as the selection of a cloudlet for task execution, with the required data routed to the chosen cloudlet. At each decision step, the agent selects one cloudlet from the set $\mathcal{N}$. Let $a_{m,i,t} \in \mathcal{N}$ denote the cloudlet selected for executing task $v_{m,i}$ at time slot t. The offloading decision variable is defined as

$$x_{m,i,n,t} = \begin{cases} 1, & \text{if } a_{m,i,t} = n, \\ 0, & \text{otherwise.} \end{cases} \tag{32}$$

5.2.3 Reward function

After task $v_{m,i}$ is executed, the agent receives the reward $r_{m,i,t} = -T_{m,i,n^*_{m,i},t}$. This per-task reward provides dense feedback after every task rather than only at the sink and improves credit assignment along the DAG. Two elements prevent locally greedy behavior. First, the pretrained embedding encodes structural targets such as ancestor/descendant workload and critical-path slack, so the agent can distinguish structurally important tasks. Second, the completion time $T_{m,i,n^*_{m,i},t}$ already includes dependency-induced waiting, predecessor transfer, queueing, service loading, and computation, so reducing intermediate times aligns with reducing the final sink-task time. The effect on later tasks is further captured through the discounted future Q-value.

5.2.4 Learning algorithm

To learn the offloading policy, each access gateway employs a DDQN [23] with one-step Temporal-Difference (TD) learning. The agent maintains two networks: an online network $Q(\mathbf{s}, a; \theta)$ that selects actions, and a target network $Q(\mathbf{s}, a; \theta^-)$ that provides stable TD targets. At each decision step, the agent observes state $\mathbf{s}_{m,i,t}$, selects action

$$a_{m,i,t} = \arg\max_{a \in \mathcal{N}} Q(\mathbf{s}_{m,i,t}, a; \theta), \tag{33}$$

with ϵ-greedy exploration, receives reward $r_{m,i,t}$, and transitions to the next state. The TD target is computed using the double Q-learning update:

$$y_{m,i,t} = r_{m,i,t} + \gamma\, Q\Big(\mathbf{s}_{m,i+1,t},\, \arg\max_{a} Q(\mathbf{s}_{m,i+1,t}, a; \theta);\, \theta^-\Big), \tag{34}$$

where γ is the discount factor. The action is selected by the online network θ but evaluated by the target network θ^-, which reduces overestimation bias compared to a standard Deep Q-Network. The online network is updated by minimizing

$$\mathcal{L} = \mathbb{E}\left[\left(y_{m,i,t} - Q(\mathbf{s}_{m,i,t}, a_{m,i,t}; \theta)\right)^2\right], \tag{35}$$

and the target network parameters θ^- are periodically copied from θ. Transitions $(\mathbf{s}_{m,i,t}, a_{m,i,t}, r_{m,i,t}, \mathbf{s}_{m,i+1,t})$ are stored in a replay buffer and randomly sampled for mini-batch updates to reduce correlation among training samples [18].

5.3 Service cache placement subproblem

The joint optimization in (18) couples offloading and caching through the service loading delay in (14), and the large number of coupled binary variables makes solving both jointly impractical. We therefore decompose the two decisions by treating caching as a coordinated system-level decision conditioned on the service demand induced by the offloading policy.

Given the trained DDQN offloading policy, the service demand at each cloudlet can be predicted per slot from the anticipated user movement. The caching problem is then solved separately to coordinate service placement across cloudlets in response to this predicted demand.

Let $d_{q,n,t} \geq 0$ denote the number of tasks assigned to cloudlet n that require service q in time slot t. Since $d_{q,n,t}$ depends on offloading decisions in slot t, it is predicted before the slot begins: each gateway runs a deterministic rollout of its trained DDQN agent over its associated users, using the previous slot's caching variables $c_{q,n,t-1}$ and the anticipated gateway associations for slot t, to predict the task-to-cloudlet assignments $\hat{x}_{m,i,n,t}$. The predicted demand is then

$$d_{q,n,t} = \sum_{m \in \mathcal{M}} \sum_{i \in \mathcal{V}_m} \hat{x}_{m,i,n,t} \, \mathbb{1}\{z_{m,i} = q\}. \tag{36}$$

Services are provided cooperatively across cloudlets. If service q is not cached locally at cloudlet n, it must be fetched either from the cloud at rate r_n or from another cloudlet $\bar{n}$ that currently caches it, incurring backhaul delay $\delta_{\bar{n},n}(b_q)$. Since the backhaul delay in (9) is linear in the data size, the delay for fetching a service of size b_q can be written as $b_q \cdot \delta_{\bar{n},n}(1)$. Similarly, cloud retrieval incurs delay b_q / r_n.

We define $\mathcal{N}^+ = \mathcal{N} \cup \{\text{cloud}\}$, where the cloud stores all services (i.e., $c_{q,\text{cloud},t} = 1, \ \forall q, t$). For notational consistency, we define $\delta_{\text{cloud},n}(1) \triangleq 1/r_n$ for all $n \in \mathcal{N}$, so that the per-unit retrieval delay from any provider in $\mathcal{N}^+$ is expressed uniformly as $\delta_{\bar{n},n}(1)$. The service caching subproblem in time slot t is then formulated as

$$\min_{\{c_{q,n,t}\}} \quad \sum_{q \in \mathcal{Q}} \sum_{n \in \mathcal{N}} d_{q,n,t} \, b_q \min_{\bar{n} \in \mathcal{N}^+ : \, c_{q,\bar{n},t} = 1} \delta_{\bar{n},n}(1) \tag{37a}$$

$$\text{s.t.} \quad \sum_{q \in \mathcal{Q}} c_{q,n,t} \leq C_n^{\max} \quad \forall n \in \mathcal{N}, \tag{37b}$$

$$c_{q,n,t} \in \{0,1\} \quad \forall q \in \mathcal{Q}, \ \forall n \in \mathcal{N}. \tag{37c}$$

The objective minimizes the total weighted service fetching delay, where the weight of each service-cloudlet pair reflects both the demand $d_{q,n,t}$ and the service size b_q. The constraint in (37b) limits the number of cached services at each cloudlet.

This is an integer program due to the binary caching variables. To obtain a computationally efficient solution, we solve a linear relaxation. The caching variables are relaxed to continuous values $0 \leq \tilde{c}_{q,n,t} \leq 1$, and continuous provider-assignment variables $u_{q,\bar{n},n,t} \in [0,1]$ are introduced to represent the selection of provider $\bar{n} \in \mathcal{N}^+$ for delivering service q to cloudlet n. Let $\tilde{\mathbf{C}}_t \triangleq \{\tilde{c}_{q,n,t}\}_{q \in \mathcal{Q}, \, n \in \mathcal{N}}$ and $\mathbf{U}_t \triangleq \{u_{q,\bar{n},n,t}\}_{q \in \mathcal{Q}, \, \bar{n} \in \mathcal{N}^+, \, n \in \mathcal{N}}$ denote the relaxed caching and provider-assignment variables in slot t, respectively. The resulting linear program (LP) is

$$\min_{\tilde{\mathbf{C}}_t, \, \mathbf{U}_t} \quad \sum_{q \in \mathcal{Q}} \sum_{n \in \mathcal{N}} d_{q,n,t} \, b_q \sum_{\bar{n} \in \mathcal{N}^+} \delta_{\bar{n},n}(1) \, u_{q,\bar{n},n,t} \tag{38a}$$

$$\text{s.t.} \quad \sum_{\bar{n} \in \mathcal{N}^+} u_{q,\bar{n},n,t} = 1 \qquad \forall q \in \mathcal{Q}, \ n \in \mathcal{N} \tag{38b}$$

$$0 \leq u_{q,\bar{n},n,t} \leq \tilde{c}_{q,\bar{n},t} \quad \forall q \in \mathcal{Q}, \ n, \bar{n} \in \mathcal{N} \tag{38c}$$

$$0 \leq u_{q,\text{cloud},n,t} \leq 1 \qquad \forall q \in \mathcal{Q}, \ n \in \mathcal{N} \tag{38d}$$

$$\sum_{q\in\mathcal{Q}} \tilde{c}_{q,n,t} \le C_n^{\max} \quad \forall n \in \mathcal{N} \tag{38e}$$

$$0 \le \tilde{c}_{q,n,t} \le 1 \qquad \forall q \in \mathcal{Q},\ n \in \mathcal{N}. \tag{38f}$$

Constraint (38b) ensures that each service at each cloudlet is fully allocated to a provider. Constraint (38c) ensures that a cloudlet can serve as a provider only if it caches the service. Constraint (38d) allows the cloud to always serve as a fallback provider. Due to the linear objective, the optimal solution favors assigning each service request to providers with lower transfer costs, subject to caching and capacity constraints.

Finally, the relaxed variables $\tilde{c}_{q,n,t}$ are rounded to obtain the integer caching variables $c_{q,n,t}$ by selecting, for each cloudlet n, the $C_n^{\max}$ services with the largest fractional values. This rounding heuristic prioritizes services with larger relaxed cache-placement values, which reflect demand, service size, and retrieval cost in the LP objective. The resulting caching variables $c_{q,n,t}$ are broadcast to all gateways, which use them as part of $\mathbf{serv}_{m,i,t}$ when making the actual offloading decisions during slot t.

5.4 Computational complexity

For a DAG with $|\mathcal{V}|$ tasks, each Transformer layer has complexity $O(|\mathcal{V}|^2 d^{\text{emb}} + |\mathcal{V}|(d^{\text{emb}})^2)$ due to self-attention and feedforward operations. With Λ layers, the per-DAG encoder cost is $O(\Lambda(|\mathcal{V}|^2 d^{\text{emb}} + |\mathcal{V}|(d^{\text{emb}})^2))$. This cost is incurred offline during pretraining.

Since each user executes a repeated DAG with fixed structure, task embeddings are computed once and reused across time slots. Online, each DDQN decision uses a state of dimension $d^{\text{emb}} + 6N + 2$, so the per-task inference cost scales as $O(d^{\text{emb}} + N)$ for fixed hidden-layer sizes. Across all users and tasks, the per-slot offloading complexity is $O(M|\mathcal{V}_m|(d^{\text{emb}} + N))$. The caching LP in (38) is solved once per slot and contains $O(N^2 Q)$ provider-selection variables. In contrast, exact joint optimization requires searching over a per-slot space of $|\Omega| = N^{\sum_{m=1}^{M}|\mathcal{V}_m|} \times 2^{NQ}$. The proposed framework therefore replaces exponential joint optimization with sequential DDQN decisions and polynomial-time LP-based caching.

6 Simulation results

This section evaluates the proposed framework through an offline encoder study, a state-representation ablation, convergence and latency-breakdown analyses, and a sensitivity study.

6.1 Evaluation of transformer-based task representation

The Transformer encoder is evaluated through an offline ablation study. It is trained on randomly generated DAG applications and evaluated on DAGs from real Alibaba traces [1]. The encoder uses $\Lambda = 4$ layers, $\mathbb{H} = 4$ attention heads, embedding dimension $d^{\text{emb}} = 128$, feedforward dimension $d^{\text{ff}} = 512$, positional encoding dimension $d^{\text{pe}} = 16$, and GELU activation, trained with AdamW at learning rate 3×10^{-4} for 500 epochs.

To evaluate each design component, we incrementally build the final configuration from a minimal baseline. Fig. 3 shows the convergence of the mean absolute error (MAE), averaged over the six targets, for each configuration. The base token uses only the three raw features ($w_i/W^{\max}$, $l_i/L^{\max}$, $o_i/O^{\max}$) and carries no dependency information, plateauing at an MAE of 0.3075. Adding local structural features (normalized in/out-degree and source/sink indicators) reduces the MAE to 0.1648, so the encoder can distinguish tasks at different DAG positions. Adding 1-hop neighborhood statistics, which expose aggregated workload and data volume from immediate predecessors and successors, further reduces it to 0.0878. Finally, adding depth/height sinusoidal positional encoding yields the lowest

MAE of 0.0339, the largest single reduction, since explicit positional information lets the Transformer distinguish tasks at different levels of the dependency chain, which is critical for critical-path quantities such as slack.

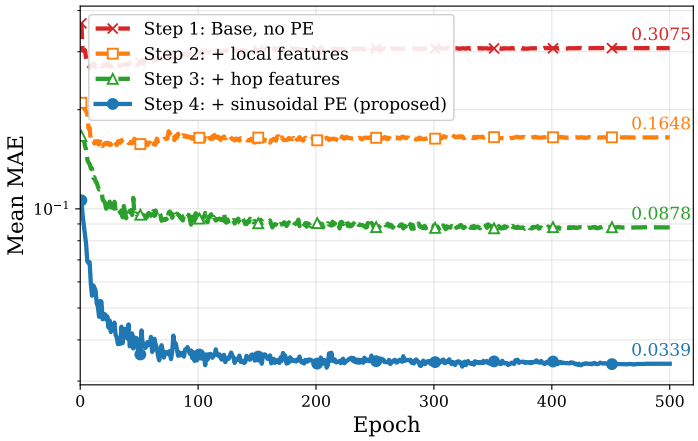


Figure 3: Convergence of mean MAE over 500 training epochs. Each step adds one component to the previous configuration.

Table 4 reports the MAE for each configuration and target. Slack is the hardest target, as it depends on the global critical-path structure, yet the proposed configuration still reduces its MAE from 0.1913 to 0.0972. Combining the base, local, and hop features with sinusoidal positional encoding gives the lowest MAE across all targets, and this configuration is adopted to generate the task embeddings.

Table 4: Ablation results at epoch 500. Each column adds one component to the previous step.

	Step 1 base	**Step 2** + local	**Step 3** + hop	**Step 4** + sincos PE
wr	0.2763	0.1305	0.0680	**0.0149**
dvr	0.2758	0.1353	0.0733	**0.0201**
wrf	0.2759	0.1430	0.0591	**0.0160**
aw	0.3246	0.1578	0.0678	**0.0254**
dw	0.3087	0.1186	0.0676	**0.0296**
slack	0.3840	0.3038	0.1913	**0.0972**
Mean	0.3075	0.1648	0.0878	**0.0339**

6.2 Simulation setup and baselines

The simulation parameters used in our experiments are summarized in Table 5. Unless otherwise stated, these values are used as the default setting. We compare the proposed framework with several learning-based and heuristic baselines. All results are averaged over 10 independent random seeds. Each seed fixes a distinct network topology, user mobility pattern, and task generation, so all methods are evaluated under the same set of conditions.

6.2.1 Learning-based approaches

All learning-based methods share the same DDQN backbone and training procedure described in Section 5.2, differing only in their task representation and state design. This isolates the effect of representation choice on offloading performance.

TA-DDQN (Task-Aware DDQN): A baseline that relies only on task-level features, without incorporating service information or task dependencies.

Table 5: Main simulation parameters.

Parameter	Value / Setting
# cloudlets, users, services	$N = 10,\ M = 100,\ Q = 10$
Tasks per user (DAG size)	$\lvert\mathcal{V}_m\rvert = 10$
Base CPU frequency	$f_n^{\text{base}} = 5$ GHz
Frequency: range, ρ_f, σ_f	$[0.4, 1.3],\ 0.87,\ 0.13$
Base background workload	$w_n^{\text{base}} = 200$ Mcycles
Workload: range, ρ_w, σ_w	$[0.4, 1.3],\ 0.87,\ 0.13$
Cache capacity	$C_n^{\max} = 2$ per cloudlet
Wireless bandwidth	$B = 20$ MHz
Backhaul / cloud rate	$[30, 150]$ / $[3, 10]$ Mbps
Task CPU cycles	$w_{m,i} \leq 500$ Mcycles
Input / interm. / inter-DAG size	≤ 4 / ≤ 2 / ≤ 10 MB
Service size	$b_q \in [0.1, 20]$ MB
User velocity	40 km/h
Discount factor / batch	$\gamma = 0.95$ / 2048
Learning rate / replay buffer	3×10^{-4} / 8192
Hidden layers	[256, 256]

SA-DDQN (Service-Aware DDQN): Extends TA-DDQN by incorporating service availability information, enabling awareness of service requirements during offloading decisions.

PSA-DDQN (Predecessor and Service-Aware DDQN): Adapted from [19], this baseline uses handcrafted task and dependency features, including predecessor execution locations and service requirements, to represent the offloading state.

GAT-DDQN: Uses a two-layer GAT as a graph-based alternative to the proposed Transformer encoder. The GAT is trained end-to-end with the DDQN agent, motivated by prior GAT-augmented DDQN work on DAG subtask assignment [15].

Proposed (Transformer-DDQN): The proposed method, which incorporates learned dependency-aware task embeddings together with system state information to guide offloading decisions.

6.2.2 Heuristic baselines

- **Nearest:** Each task is offloaded to the user's current access gateway, the cloudlet to which the user is directly connected.
- **Fastest:** Each task is greedily assigned to the cloudlet that minimizes the estimated immediate task latency, considering the current processing delay, queueing delay, and data-transfer delay.
- **Random:** Each task is assigned to a randomly selected cloudlet.

6.3 Results and discussion

We first analyze the contribution of each state component through an ablation study, then compare the proposed method against all baselines.

6.3.1 Ablation study of state representation

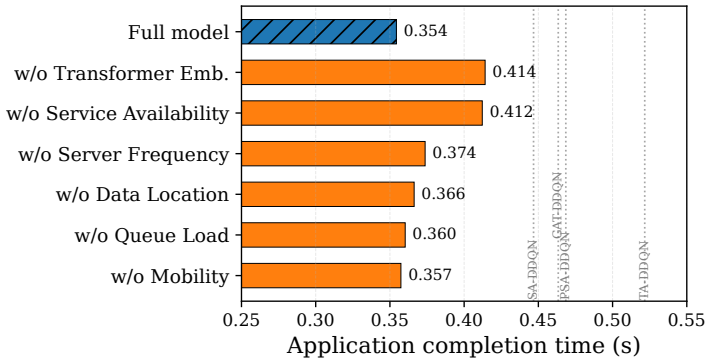
To analyze the contribution of each state component, we conduct an ablation study from two complementary directions, shown in Fig. 4. The leave-one-out analysis in Fig. 4(a) removes one component at a time from the full model. The single-group analysis in Fig. 4(b) adds one component at a time to a base model that contains only task workload features.

Transformer embedding and service availability play the leading role. Removing either one causes the largest degradation in Fig. 4(a). Without the Transformer embedding, latency increases by 16.9%. Without service availability, latency increases by 16.3%. Both removals lead to degradation comparable to or worse than baseline methods, indicating that dependency-aware representations and service information are key factors for effective offloading decisions.

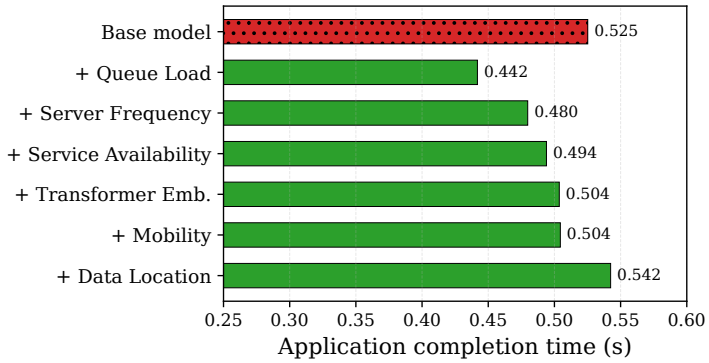
Server frequency and data location provide moderate gains. Removing server frequency increases latency by 5.5%. Removing data location increases it by 3.4%. These components capture server capacity and data locality, enabling more informed offloading decisions.

Queue load and mobility contribute less in this setting. Queue load reflects the instantaneous congestion level at each server, so the agent can account for queueing delays in offloading decisions. Mobility captures changes in gateway association during handover events, which affect communication paths. Their contribution is expected to increase under higher server congestion or more frequent handovers, where dynamic system conditions play a larger role in offloading decisions.

The single-group analysis provides complementary insight. In Fig. 4(b), queue load provides the largest single-component improvement (15.9% over base). This reflects the absence of server-side information in the base model, where queue load provides the first signal of server congestion. Server frequency provides the second largest gain (8.6%). The Transformer embedding alone provides only 4.1% improvement when added in isolation. The embedding captures task structure but cannot be effectively utilized if the agent lacks basic server information.



(a) Leave-one-out: removing each component from the full model.



(b) Single-group: adding each component to the base model.

Figure 4: Ablation study of the state representation. Dotted lines indicate baseline methods.

6.3.2 Convergence of total latency

Fig. 5 presents the convergence of total latency for all methods, where the shaded regions denote one standard deviation across the 10 seeds. The proposed method converges to the lowest latency. Among the learning-based methods, TA-DDQN yields the highest latency, followed by SA-DDQN, PSA-DDQN, and GAT-DDQN. This ordering highlights the importance of progressively richer state representations, from task-level features to service awareness, predecessor information, and graph-based DAG encoding.

Notably, the Fastest heuristic outperforms TA-DDQN and SA-DDQN. This aligns with the ablation results, which show that server capacity information has a strong impact on offloading quality. The Fastest heuristic explicitly uses immediate latency estimates based on server and transfer conditions, while TA-DDQN and SA-DDQN do not include sufficient server-side and dependency-aware information in their state.

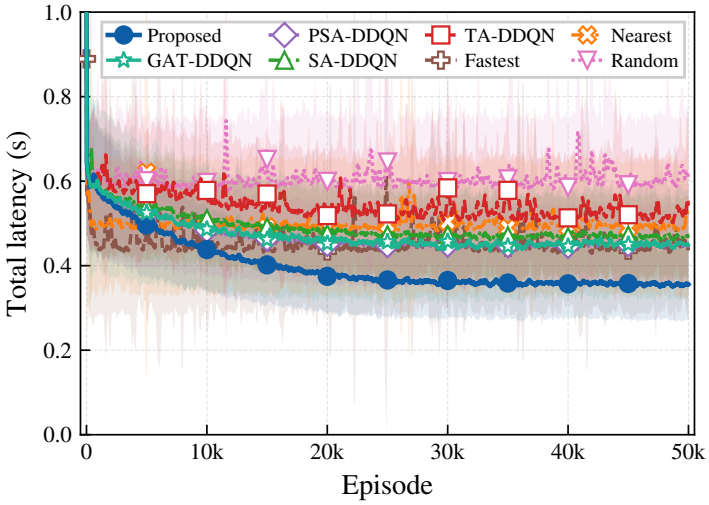


Figure 5: Convergence of total latency for the proposed method and benchmark schemes.

A key comparison is the proposed method versus GAT-DDQN, since both encode the DAG but differ in how the representation is learned. GAT-DDQN trains the graph encoder end-to-end with the DDQN agent, so representation and policy learning are optimized simultaneously from the same reinforcement signal, which can make it harder for the encoder to learn explicit DAG-level structural properties compared with offline pretraining on analytically computed targets; it also uses only two attention layers to keep complexity comparable. The proposed method instead pretrains the encoder offline with explicit structural supervision, so its embeddings already capture DAG-level structure before policy training, letting the agent focus on offloading under dynamic conditions. GAT-DDQN therefore converges to higher latency. The proposed method surpasses all baselines, including PSA-DDQN, GAT-DDQN, and Fastest, indicating that pretrained embeddings provide dependency-aware information beyond handcrafted DAG features.

6.3.3 Analysis of latency components

Fig. 6 breaks down the seed-averaged total latency into its main components. The results reveal clear trade-offs across different methods, as no single approach minimizes all components simultaneously.

The Fastest heuristic achieves low computing latency by favoring faster, less congested servers, but greedily optimizing each task without accounting for DAG dependencies and service placement prevents it from reaching the lowest completion time. The Nearest heuristic achieves the lowest data transfer and inter-DAG latency by keeping tasks on one cloudlet, which avoids inter-cloudlet transfers except at handovers. Both incur high service latency, since neither considers service availability.

The learning-based baselines show similar trade-offs. SA-DDQN reduces service latency via service availability but may raise data transfer latency for lack of dependency awareness. PSA-DDQN mitigates this with predecessor information yet still lacks a complete DAG representation. GAT-DDQN reduces computing latency faster early in training; by the end both reach comparable computing latency, but GAT-DDQN's service, data transfer, and inter-DAG components remain higher, so it learns short-term execution well but does not match the full trade-off across dependency-aware offloading,

Figure 6: Convergence of the main latency components: (a) computing latency, (b) service latency, (c) data transfer latency, and (d) inter-DAG latency.

service availability, and data locality. The proposed method benefits from offline structural pretraining, which yields a more informative DAG-level representation of task dependencies and critical-path structure. Combined with service availability and data-location features in the state, this lets the agent balance service placement, data locality, and computational load, so it reaches the lowest overall latency rather than optimizing any single component.

6.3.4 Sensitivity analysis

Fig. 7 evaluates the sensitivity of all methods to four system parameters, where the error bars denote one standard deviation across the 10 seeds.

CPU frequency (Fig. 7(a)). Higher CPU frequencies reduce latency for all methods as computation becomes faster; the proposed method's advantage here comes from also accounting for service availability and data transfer, not computation alone.

CPU cycles per task (Fig. 7(b)). As the maximum CPU cycles per task increases, computation dominates latency, so the Fastest heuristic, which directly optimizes per-task latency, narrows its gap with the proposed method; the residual gap reflects the proposed method's joint awareness of computational capacity, service availability, and data locality.

Number of cloudlets (Fig. 7(c)). More cloudlets have competing effects: they add computational capacity and service availability, but a larger network increases handovers and inter-cloudlet transfers, which raises latency within and across DAG instances. The Random baseline degrades clearly, as spreading tasks across more servers increases inter-cloudlet transfer. The proposed method stays stable as the network grows by making consistent offloading decisions.

Background server load (Fig. 7(d)). Rising background server load increases latency for all methods through longer queueing delays. Under heavy load, TA-DDQN, SA-DDQN, and Nearest converge to similar levels, since none account for server congestion, while Fastest and the proposed method stay lower by selecting less congested servers. The proposed method consistently beats Fastest, as its state captures richer information beyond server-side conditions.

In all four sweeps, the proposed method gives the lowest latency at every operating point. Its advantage is smallest when servers are fast and largest under heavy load, where avoiding congested cloudlets matters most, showing that it adapts to the dominant latency factor in each regime.

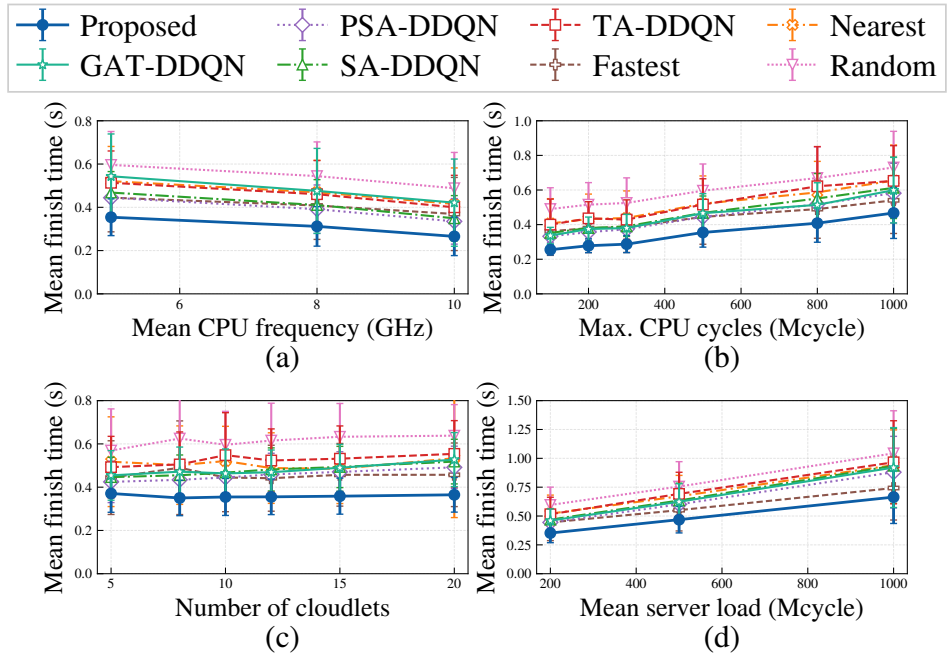


Figure 7: Total latency under varying system parameters: (a) mean CPU frequency, (b) maximum CPU cycles per task, (c) number of cloudlets, and (d) background server load.

7 Conclusion

This paper addressed task offloading for DAG-based applications in edge networks under task dependencies, chaining across consecutive DAG instances, and user mobility. We proposed a framework that combines pretrained dependency-aware task representations with DDQN-based offloading. A Transformer encoder was trained offline using structural supervision signals derived from DAG properties to generate task embeddings that capture dependency structure and task importance. These embeddings were incorporated into the state of distributed DDQN agents alongside system-level features such as service availability, server load, and data location. Coordinated service caching was decoupled and solved efficiently via LP relaxation as a supporting system component. The proposed approach consistently outperforms DDQN baselines with handcrafted state representations and heuristic methods across varying CPU frequencies, task workloads, network scales, and background server loads, with the ablation studies identifying task embeddings and service availability as the most influential components. Future work will explore multi-agent coordination across gateways via federated learning, and extend the framework to cloudlet failures during DAG execution through robustness mechanisms such as redundant data placement and service replication.

References

[1] Alibaba Cloud. Alibaba cluster trace dataset. https://github.com/alibaba/clusterdata/tree/master/cluster-trace-v2018, 2018.

[2] Zequn Cao, Xiaoheng Deng, Sheng Yue, Ping Jiang, Ju Ren, and Jinsong Gui. Dependent task offloading in edge computing using gnn and deep reinforcement learning. IEEE Internet of Things Journal, 11(12):21632–21646, 2024.

[3] Xiangchun Chen, Jiannong Cao, Yuvraj Sahni, Mingjin Zhang, Zhixuan Liang, and Lei Yang. Mobility-aware dependent task offloading in edge computing: a digital twin-assisted reinforcement learning approach. IEEE Transactions on Mobile Computing, 2024.

[4] Mohammad Reza Golzari Oskoui and Brunilde Sansò. Online dependency-aware task offloading in cloudlet-based edge computing networks. In Proceedings of the Int'l ACM Symposium on Mobility Management and Wireless Access, pages 91–97, 2023.

[5] Lina A Haibeh, Mustapha CE Yagoub, and Abdallah Jarray. A survey on mobile edge computing infrastructure: Design, resource management, and optimization approaches. IEEE Access, 10:27591–27610, 2022.

[6] Xiao He, Shanchen Pang, Haiyuan Gui, Kuijie Zhang, Nuanlai Wang, and Shihang Yu. Online offloading and mobility awareness of dag tasks for vehicle edge computing. IEEE Transactions on Network and Service Management, 2024.

[7] Ke Hongchang, Wang Hui, Sun Hongbin, and Halvin Yang. Deep reinforcement learning-based task offloading and service migrating policies in service caching-assisted mobile edge computing. China Communications, 21(4):88–103, 2024.

[8] Akhirul Islam, Arindam Debnath, Manojit Ghose, and Suchetana Chakraborty. A survey on task offloading in multi-access edge computing. Journal of Systems Architecture, 118:102225, 2021.

[9] Chaitanya K Joshi. Transformers are graph neural networks. arXiv preprint arXiv:2506.22084, 2025.

[10] Ying Ju, Yuchao Chen, Zhiwei Cao, Lei Liu, Qingqi Pei, Ming Xiao, Kaoru Ota, Mianxiong Dong, and Victor CM Leung. Joint secure offloading and resource allocation for vehicular edge computing network: A multi-agent deep reinforcement learning approach. IEEE Transactions on Intelligent Transportation Systems, 24(5):5555–5569, 2023.

[11] Seung-seob Lee and SuKyoung Lee. Resource allocation for vehicular fog computing using reinforcement learning combined with heuristic information. IEEE Internet of Things Journal, 7(10):10450–10464, 2020.

[12] Chunlin Li, Yong Zhang, and Youlong Luo. A federated learning-based edge caching approach for mobile edge computing-enabled intelligent connected vehicles. IEEE Transactions on Intelligent Transportation Systems, 24(3):3360–3369, 2022.

[13] Manqing Liu, David R Bellamy, and Andrew L Beam. Dag-aware transformer for causal effect estimation. arXiv preprint arXiv:2410.10044, 2024.

[14] Xinwei Liu, Jiaxin Zhang, Xing Zhang, and Wenbo Wang. Mobility-aware coded probabilistic caching scheme for mec-enabled small cell networks. IEEE Access, 5:17824–17833, 2017.

[15] Zhang Liu, Lianfen Huang, Zhibin Gao, Manman Luo, Seyyedali Hosseinalipour, and Huaiyu Dai. Ga-drl: Graph neural network-augmented deep reinforcement learning for dag task scheduling over dynamic vehicular clouds. IEEE Transactions on Network and Service Management, 21(4):4226–4242, 2024.

[16] Yuankai Luo, Veronika Thost, and Lei Shi. Transformers over directed acyclic graphs. Advances in Neural Information Processing Systems, 36:47764–47782, 2023.

[17] Yuyi Mao, Changsheng You, Jun Zhang, Kaibin Huang, and Khaled B Letaief. A survey on mobile edge computing: The communication perspective. IEEE communications surveys & tutorials, 19(4):2322–2358, 2017.

[18] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. Nature, 518(7540):529–533, 2015.

[19] Mohammad Reza Golzari Oskoui and Brunilde Sansò. Distributed dependency-aware task offloading and service caching in cloudlet-based edge computing networks. IEEE Transactions on Services Computing, pages 1–14, 2026.

[20] Dian Tang, Xuefei Zhang, Meng Li, and Xiaofeng Tao. Adaptive inference reinforcement learning for task offloading in vehicular edge computing systems. In 2020 IEEE International Conference on Communications Workshops (ICC Workshops), pages 1–6. IEEE, 2020.

[21] Shujuan Tian, Xianghong Deng, Pengpeng Chen, Tingrui Pei, Sangyoon Oh, and Weiping Xue. A dynamic task offloading algorithm based on greedy matching in vehicle network. Ad Hoc Networks, 123:102639, 2021.

[22] Haluk Topcuoglu, Salim Hariri, and Min-You Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. IEEE transactions on parallel and distributed systems, 13(3):260–274, 2002.

[23] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. Proceedings of the AAAI Conference on Artificial Intelligence, 30(1), Mar. 2016.

[24] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. Advances in neural information processing systems, 30, 2017.

[25] Jinming Wu, Yuan Zou, Xudong Zhang, Jiahui Liu, Wunjing Sun, and Guodong Du. Dependency-aware task offloading strategy via heterogeneous graph neural network and deep reinforcement learning. IEEE Internet of Things Journal, 2025.