Les Cahiers du GERAD ISSN: 0711–2440

Distributed dependency-aware task offloading and service caching in cloudlet-based edge computing networks

M. R. Golzari Oskoui, B. Sansò

G–2026–32

June 2026

CITATION ORIGINALE / ORIGINAL CITATION

M. R. Golzari Oskoui and B. Sansò, Distributed dependency-aware task offloading and service caching in cloudlet-based edge computing networks, in *IEEE Transactions on Services Computing*, vol. 19, no. 2, pp. 1120–1133, March-April 2026, doi: 10.1109/TSC.2026.3664339. https://ieeexplore.ieee.org/abstract/document/11395623.

La publication de ces rapports de recherche est rendue possible grâce au soutien de HEC Montréal, Polytechnique Montréal, Université McGill, Université du Québec à Montréal, ainsi que du Fonds de recherche du Québec – Nature et technologies.

The publication of these research reports is made possible thanks to the support of HEC Montréal, Polytechnique Montréal, McGill University, Université du Québec à Montréal, as well as the Fonds de recherche du Québec – Nature et technologies.

Dépôt légal – Bibliothèque et Archives nationales du Québec, 2026 – Bibliothèque et Archives Canada, 2026

Legal deposit – Bibliothèque et Archives nationales du Québec, 2026 – Library and Archives Canada, 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

Distributed dependency-aware task offloading and service caching in cloudlet-based 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–2026–32

Abstract : To meet the increasing demand for low-latency processing in mobile and IoT devices, edge computing offloads user application tasks to nearby cloudlets (edge servers). Modern applications comprise multiple interdependent tasks with diverse service requirements that must be preloaded on cloudlets. This creates a coupled optimization challenge: jointly deciding task offloading locations and service caching across cloudlets. Inter-task dependencies further complicate the problem by introducing additional communication delays through data transfers and connectivity between cloudlets. Unlike existing studies that rely on a centralized controller with global knowledge, this work introduces a distributed framework for dependency-aware task offloading and service caching. This decentralized design reduces communication overhead and improves scalability. In the proposed framework, each cloudlet observes user task arrivals and network conditions, then applies a deep reinforcement learning (DRL) model enhanced with guided action shaping to determine task-offloading decisions. In parallel, service caching is performed independently by each cloudlet based on its observed service-demand statistics. Simulation results demonstrate that the proposed algorithm outperforms existing benchmarks for dependent task offloading and service caching. Moreover, it exhibits strong adaptability to dynamic environments, enabling more efficient and scalable edge computing.

Keywords : Edge computing; Internet of Things; task offloading; service caching, reinforcement learning; action shaping

Acknowledgements: We kindly acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC) under Grant DG 05734.

1 Introduction

Due to limited energy, storage, and computing power, mobile and Internet of Things (IoT) devices may not be able to handle time-sensitive tasks locally [15]. Cloud computing emerged as a solution by offloading such tasks to powerful remote servers [10,13]. Yet, the massive and rapidly increasing data generated by IoT devices strains cloud infrastructure, leading to higher transmission delays between devices and distant servers [9]. This makes cloud computing inadequate for many latency-critical applications [12]. To overcome these limitations, edge computing has been introduced, placing computing resources closer to users [2,7]. This proximity reduces end-to-end latency between devices and compute resources.

Applications in areas such as artificial intelligence, augmented reality, and autonomous systems often involve complex interdependent tasks that are modeled using Directed Acyclic Graphs (DAGs) [28]. Task dependencies complicate offloading because the system must account for connections between edge servers and the latency of transferring data between them. In Fig. 1, we illustrate a self-driving car application with dependent tasks and their corresponding offloading and service caching decisions. In this example, the initial task gathers and prepares data from multiple sensors such as GPS, cameras, RADAR, and LIDAR on the vehicle. Subsequent pre-processing and navigation tasks are executed on cloudlets, and each task requires a specific service that must be provided by the selected server. A given cloudlet may cache some services but not others, in which case tasks must either be offloaded to other cloudlets that host the required services or incur additional delay due to service loading [28]. Given the limited capacity of cloudlets, it is not feasible for all servers to offer every service [11]. Consequently, decisions must be made about the distribution of services across servers to attain optimal performance in application execution. In other words, decisions need to be made regarding service caching. This issue poses a greater challenge for cloudlet edge computing networks, characterized by their lower storage capacity and more widespread distribution.

In this diverse environment where servers exhibit varying characteristics due to diverse hardware configurations and dynamic caching decisions, it becomes critical for offloading strategies to adjust to changing environmental conditions. Meanwhile, the increasing number of users in the network makes centralized control impractical for real-time offloading. Addressing the offloading challenge and determining the optimal server for each task is better approached through a distributed methodology. However, distributed solutions require decision-makers to exchange network information, which may introduce considerable communication overhead. To mitigate this, decision-makers can leverage reinforcement learning to autonomously learn the environment and make more effective server assignment decisions for task offloading.

Task offloading, service caching, and task dependency management can be addressed independently; however, jointly optimizing them enables more effective latency reduction by capturing the interactions between task execution locations, service availability, and inter-task dependencies. In particular, data transfers between dependent tasks make coordinated offloading and caching decisions essential. To the best of our knowledge, existing studies that jointly consider task dependencies and service caching mainly rely on centralized decision-making. In contrast, we adopt a distributed approach in which cloudlets make local decisions with low overhead and improved scalability. Building on these gaps, the main contributions of this paper are summarized as follows:

- We formulate a joint task offloading and service caching problem in cloudlet edge computing. The formulation considers task dependencies, service requirements, and configurations. Our formulation includes various latency types, such as processing, queuing, service loading, and data transfer between servers handling consecutive tasks. Additionally, to address the service caching problem, we derive it as a subproblem of the main problem.
- We introduce a novel distributed DRL framework designed to address the challenges of offloading dependent tasks in a diverse cloudlet-based edge computing network. Recognizing server heterogeneity, our approach accounts for servers having varying capabilities and possibly not

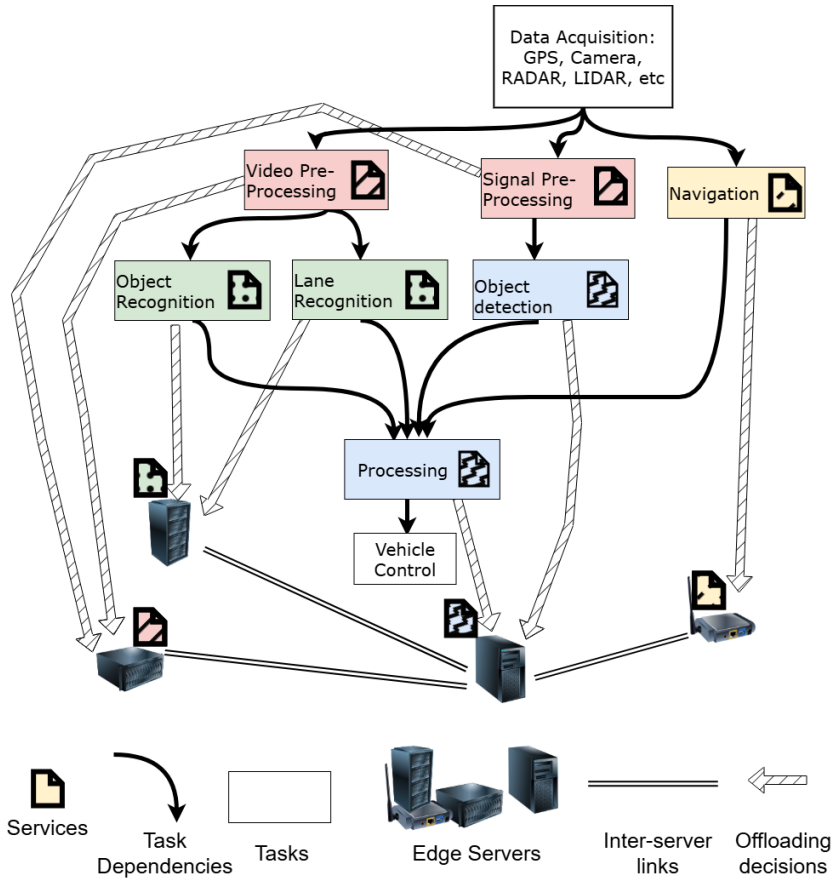

Figure 1: Illustration of task dependencies, offloading decisions, and service caching decisions in an autonomous vehicle application, where each task may be executed on different cloudlets according to its requirements.

supporting all services. Through reinforcement learning, our solution reduces communication overhead by enabling autonomous learning and adaptation to the edge environment.

- Diverging from prior studies, our approach distinguishes itself by integrating service requirements and task dependencies into the RL-based decision-making process. Previous RL-based methodologies often overlooked the role of DAG structures in decision-making. In our approach, DAG structures are reflected in the state representation by vectors indicating predecessor task destination servers and the service needs of successor tasks. This incorporation enhances the overall efficiency of the task offloading technique.
- We propose a decentralized mechanism in which cloudlets independently make service caching decisions based on local statistical analysis. To enhance offloading efficiency, we employ guided action shaping, which steers offloading decisions toward servers that are more likely to host the required services, thereby accelerating convergence and improving performance.
- We conduct a comprehensive performance evaluation of our proposed approach by comparing it with five benchmark methods, including both standard and guided versions of DRL-based approaches. The proposed method demonstrates strong adaptability to environmental changes and consistently outperforms all compared methods under various conditions.

The remainder of this paper is organized as follows: Section 2 reviews related work and compares existing studies. Section 3 outlines the system model, while Section 5 details the proposed approach. Section 6 presents and analyzes simulation results. Finally, Section 7 summarizes the study and suggests directions for future research.

2 Related work

Based on the system models adopted in existing studies, we categorize prior work on task offloading and service caching in edge computing into five main categories, which are reviewed in the following. This classification is based on fundamental system modeling assumptions, including the number of edge servers, task dependency structures, and whether service requirements and caching decisions are explicitly considered. Table 1 summarizes representative studies according to these criteria and further distinguishes decision architectures as centralized, distributed, or hybrid.

Table 1: Comparison of edge-computing studies by server setting, task dependency, caching, and decision architecture.

Ref.	Servers	Tasks	Caching	Arch.
[8]	Single	Independent	✗	Centralized
[14]	Single	Dependent (DAG)	✗	Distributed
[16]	Single	Independent	✓	Centralized
[4, 20, 24]	Multiple	Independent	✗	Distributed
[25, 26]	Multiple	Independent	✓	Distributed
[23, 27]	Multiple	Independent	✓	Centralized
[19]	Multiple	Independent	✓	Hybrid
[3, 5, 6, 22, 29]	Multiple	Dependent (DAG)	✗	Distributed
[18]	Multiple	Dependent (DAG)	✗	Hybrid
[17, 28]	Multiple	Dependent (DAG)	✓	Centralized
Proposed	Multiple	Dependent (DAG)	✓	Distributed

Single-edge offloading. Several studies considered a single-edge server system model, often complemented by a remote cloud server, where the offloading decision was limited to executing tasks locally, at the edge, or in the cloud [8, 14, 16]. These approaches employed learning-based methods, such as centralized DRL or per-user Q-learning, to optimize performance metrics including latency, energy consumption, or service loading delay. However, these single-edge systems did not capture multi-server settings, such as edge server selection or inter-edge communication latencies, which are essential for modeling realistic large-scale edge computing networks.

Multiple-edge servers without caching and dependencies. A number of studies investigated task offloading in multi-edge server environments, where the main challenge was selecting an appropriate edge server for task execution. In [24], a UAV-enabled edge computing network was considered, and learning-based methods were employed to optimize transmission power of users. In [4], server selection in a multi-edge setting was studied using learning-based approaches. Tang et al. [20] employed Long Short-Term Memory (LSTM) and Deep Q-Network models to make decentralized offloading decisions based on task characteristics and queue states of edge servers. However, these studies assumed independent tasks and did not consider service caching, which limited their applicability to modern service-oriented applications.

Service-aware caching (independent tasks). There are studies that considered the service requirements of individual tasks [19, 23, 25–27]. In [25] and [23], the joint optimization of service caching and task offloading was investigated using learning-based and optimization-based methods, respectively. In [27], this setting was extended by incorporating federated learning to preserve service request privacy. In [26], a multi-agent reinforcement learning framework enhanced with graph attention mechanisms was adopted. Sun et al. [19] proposed a hierarchical DRL architecture in which caching decisions were made locally at edge servers and offloading decisions were coordinated centrally. These studies jointly optimized service caching and task offloading but assumed independent tasks, and therefore did not capture the impact of task dependencies on caching efficiency or application latency.

Dependency-aware offloading (no caching). Several studies explicitly modeled task dependencies using DAGs to capture execution order in edge computing systems. In [22], scheduling and learning-based offloading strategies are proposed to reduce execution latency for dependent tasks in dynamic environments. Goudarzi et al. [6] developed heuristic methods for clustering, placement, and migration in fog computing systems, focusing on managing dependent tasks to improve response time and energy efficiency. In [3], a dependency-aware reinforcement learning framework was introduced that incorporated task dependencies into the reward design to optimize offloading and resource allocation. Shu et al. [18] studied dependency-aware offloading using both centralized heuristic methods and decentralized game-theoretic approaches in multi-user, multi-server environments. In [5, 29], dependency-aware offloading was further investigated using reinforcement learning, but task-level execution was considered without explicitly modeling service caching. However, these studies captured task dependencies but did not consider service-oriented caching, leaving open how dependency structures interact with service availability and caching efficiency.

Joint dependency-aware offloading and service caching. In [28] and [17], task offloading and service caching were jointly considered for DAG-based applications in multi-edge server environments. In [28], centralized algorithms based on convex programming were proposed to minimize the overall makespan under the assumption of full knowledge of users, tasks, and servers. In [17], a centralized heuristic approach was adopted that incorporated DAG structures and service requirements. However, these methods required complete global information, and their reliance on centralized decision-making resulted in scalability limitations and increased communication overhead.

In conclusion, existing studies either assumed independent or purely computational tasks or relied on centralized decision-making with complete global knowledge, leaving a gap for scalable distributed solutions that jointly address task dependencies and service caching.

3 System model

The system model comprises four main components: task, network, communication, and computation. The main notations are summarized in Table 2.

3.1 Tasks model

The application of user m is modeled by utilizing a DAG denoted as $G_m = (\mathcal{V}_m, \mathcal{E}_m)$, where $\mathcal{V}_m = \{v_{m,i} | 1 \leq i \leq |\mathcal{V}_m|\}$ represents the set of tasks corresponding to the application executed by user m, and $\mathcal{E}_m = \{e_{m,i,j} | 1 \leq i, j \leq |\mathcal{V}_m|\}$ indicates the set of data flows that need to be transferred between these tasks. $e_{m,i,j}$ is a binary indicator representing whether task $v_{m,j}$ depends on the output generated by task $v_{m,i}$.

Each computational task requires input data provided by the user, along with output data from its predecessor tasks. In addition to the input data requirements explained above, each task also needs a processor to execute its operations. The computational workload is defined by the number of CPU cycles required. However, these elements alone are not sufficient to process the task. Each task also needs a specific service that must be preloaded on the server to facilitate the processing. After processing, the task produces its own output data as a result of its execution, where the output size may differ from the input size. Considering these aspects, task $v_{m,i}$ can be formally defined as a tuple $v_{m,i} =< d_{m,i}, l_{m,i}, b_{m,i}, P_{m,i}, \eta_{m,i} >$, where $d_{m,i}$ denotes the input data size, $l_{m,i}$ denotes the output data size, $b_{m,i}$ indicates the number of CPU cycles necessary for task execution, $P_{m,i}$ is the set of immediate predecessors of task $v_{m,i}$, and $\eta_{m,i} \in \mathcal{Q}$ represents the corresponding service containing essential libraries for task execution. Here, $\mathcal{Q} = \{1, 2, \ldots, Q\}$ denotes the set of all services, and Q is the number of services.

Table 2: Main notation definition

Notations	Definition
Decision and System Variables	
$x_{m,i,s}$	Offloading decision: 1 if task $v_{m,i}$ of user m executes on cloudlet s, 0 otherwise
$c_{q,s}$	Caching decision: 1 if service q is cached on cloudlet s, 0 otherwise
M, S, Q; $\mathcal{M}, \mathcal{S}, \mathcal{Q}$	Numbers and sets of users, cloudlets, and services, respectively
G_m	Application DAG of user m
$v_{m,i}$	Task i of user m
$(d_{m,i}, l_{m,i}, b_{m,i}, \eta_{m,i})$	Input size, output size, CPU cycles, and required service of task $v_{m,i}$
$P_{m,i}$	Predecessor set of task $v_{m,i}$
f_s	CPU frequency of cloudlet s
K_s	Maximum number of services cached at cloudlet s
Delay and Timing Quantities	
$\tau_{m,s}(\delta)$	End-to-end delay to transmit data size δ from user m to cloudlet s
$\tau^{\text{Uplink}}_{m,s_m}(\delta)$	Uplink delay from user m to its nearest cloudlet s_m
$\tau^{\text{Trans}}_{s_m,s}(\delta)$	Inter-cloudlet transmission delay from s_m to s
$T_{m,i,s}$	Completion time of task $v_{m,i}$ on cloudlet s
$T^{*}_{m,i,s}$	Latest arrival time of predecessor outputs at cloudlet s
$\tau^{\text{Exe}}_{m,i,s}$	Execution time of task $v_{m,i}$ on cloudlet s
W_s	Background processing workload on cloudlet s
τ^{Load}_{q}	Time to load service q on cloudlet s (if not cached)
DRL State and Parameters	
$S_{m,i}$	State of task $v_{m,i}$ of user m used by the DRL agent
$\mathcal{A}^{Pred}_{m,i}$	Cloudlet assignment vector of predecessor tasks of $v_{m,i}$
$\mathcal{Z}_{m,i}, \mathcal{Z}^{\text{Succ}}_{m,i}$	Service one-hot vector of $v_{m,i}$ and aggregated service vector of its successors
$\mathcal{C}$	Global service–cloudlet caching configuration vector
$\bar{D}_m$	Estimated completion deadline of user m's application
G_t	Discounted return of task t aggregated from its successor tasks
θ	Neural network parameters
ϵ, γ	Exploration rate and discount factor
β	Guidance probability for action shaping

3.2 Network model

Our network model includes M edge users, S cloudlets. The edge users, denoted by $\mathcal{M} = \{1, 2, \ldots, m, \ldots, M\}$, each operate an application composed of dependent tasks, where each task requires a specific service to be executed. The cloudlets are denoted by $\mathcal{S} = \{1, 2, \ldots, s, \ldots, S\}$. Each cloudlet s is situated at a specific location and is equipped with a CPU operating at a frequency f_s for task processing. Furthermore, due to storage capacity limitations, each cloudlet is unable to offer the full spectrum of services and can host only a limited number of services—up to K_s services.

In this work, we assume that users remain stationary relative to their nearest cloudlet during the execution of a single application. This assumption is appropriate for several practical scenarios: (i) fixed IoT deployments such as smart home devices, industrial sensors, or surveillance systems; and (ii) applications with short execution times (on the order of milliseconds) during which user displacement is negligible compared to cloudlet coverage areas. Furthermore, even in mobile scenarios, once an application is submitted to a cloudlet, the offloading decisions and task processing are executed rapidly, and the results are returned before significant user movement occurs. However, service demand

patterns may still vary over time as different users with diverse application requirements change their location, requiring the caching decisions to adapt accordingly. Extending the framework to fully mobile users with service requirement prediction is discussed as a direction for future work in Section 7.

Given this setting, cloudlets are responsible for assigning tasks from nearby users to themselves or neighboring servers. In other words, each user initially offloads tasks to its nearest cloudlet, which then handles the assignment process based on resource requirements and service placement. In addition, service caching decisions are made independently by cloudlets using local statistical analysis of task and service demand pattern.

3.3 Communication model

The communication in our system is categorized into two main types: interactions between edge users and cloudlets, and communications between cloudlets.

When offloading a task to a cloudlet, the data is first transmitted to the nearest cloudlet s_m and then relayed to the target cloudlet s via a wired network. As a result, the total delay experienced by a user m when transmitting data of size δ to cloudlet s, denoted by $\tau_{m,s}(\delta)$, consists of two components:

$$\tau_{m,s}(\delta) = \tau^{\text{Uplink}}_{m,s_m}(\delta) + \tau^{\text{Trans}}_{s_m,s}(\delta), \tag{1}$$

where $\tau^{\text{Uplink}}_{m,s_m}(\delta)$ is the uplink transmission delay from the user to the nearest cloudlet s_m, and $\tau^{\text{Trans}}_{s_m,s}(\delta)$ represents the additional delay from cloudlet s_m to the target cloudlet s.

The uplink transmission delay $\tau^{\text{Uplink}}_{m,s_m}(\delta)$ is calculated as:

$$\tau^{\text{Uplink}}_{m,s_m}(\delta) = \frac{\delta}{R^u_{m,s_m}},$$

where R^u_{m,s_m} is the data transmission rate from user m to cloudlet s_m, given by

$$R^u_{m,s_m} = BW \log_2(1 + \text{SNR}_{m,s_m}), \tag{2}$$

where BW stands for the channel bandwidth, and SNR_{m,s_m} is the signal-to-noise ratio at the nearest cloudlet from user m. SNR_{m,s_m} is influenced by path loss and shadowing, modeled as a log-normal distribution with a mean of $\mu_{\text{SNR}} - 10\rho \log_{10}(d^{\text{UL}}_{m,s_m})$ and variance σ^2_{SNR}. Here, μ_{SNR} is a constant representing the average SNR, d^{UL}_{m,s_m} is the distance from user m to cloudlet s_m, and ρ is the path loss exponent.

The additional delay $\tau^{\text{Trans}}_{s_m,s}(\delta)$ when transferring data of size δ from cloudlet s_m to cloudlet s is given by:

$$\tau^{\text{Trans}}_{s_m,s}(\delta) = \begin{cases} \frac{\delta}{R^{\text{Link}}_{s_m,s}} + \tau^p_{s_m,s} & \text{if } s_m \neq s \\ 0 & \text{if } s = s_m \end{cases} \tag{3}$$

Here, $R^{\text{Link}}_{s_m,s}$ is the data transmission rate between cloudlets s_m and s, and $\tau^p_{s_m,s}$ is the processing time for data packets. Since each cloudlet makes caching decisions locally based on its own observations and statistical analysis, the control communication overhead is minimized and managed within cloudlets. Given that the final application result has a small data size, the downlink transmission from the final cloudlet to the user incurs minimal delay. As a result, we consider the downlink delay negligible and do not address the downlink model in our work.

3.4 Computation model

Before a task is processed, it must collect all required input data at the destination cloudlet, including the user's initial input and the outputs of its predecessor tasks. Since the user's input data needs to

be up-to-date, the cloudlet first receives the outputs from preceding tasks, and the user's input is sent to the cloudlet afterward.

Once the task has received all necessary data, it is placed in a processing queue and must wait before execution. The length of this queue depends on how busy the server is with background demands beyond what is modeled in our system. To capture this variability, we assign each cloudlet a uniformly distributed queue load value. This approach allows queue loads to differ across servers, enabling a more realistic representation of heterogeneous background processing demands. We denote this value by W_s, which represents the number of CPU cycles waiting in the processing queue of cloudlet s. For queue load modeling, we consider only background processing demand from external sources and exclude tasks generated by the users in our system. This simplification is justified by assuming that servers have multiple processing cores, making it unlikely that several of our tasks share the same core on one server.

Before execution, the cloudlet checks whether the required service for the task is already preloaded. If not, the time needed to load the service is added to the task total processing time. Hence, the completion time of task $v_{m,i}$ at cloudlet s is given by:

$$T_{m,i,s} = \underbrace{\tau_{m,s}(d_{m,i})}_{\text{User Input Delay}} + \underbrace{\max_{v_{m,j} \in P_{m,i}} \left(T_{m,j,s_j} + \tau^{\text{Trans}}_{s_j,s}(l_{m,j}) \right)}_{\text{Predecessor Task Delay}} + \underbrace{\tau^{\text{Exe}}_{m,i,s}}_{\text{Execution Time}} \tag{4}$$

This formula consists of three terms: the first, as derived from (1), represents the delay in receiving the user's input data. The second term measures the delay in obtaining predecessor outputs by selecting the maximum of their completion times plus transmission to cloudlet s, accounting for concurrent execution across cloudlets. The final term indicates the task execution time, which is detailed as follows:

$$\tau^{\text{Exe}}_{m,i,s} = \underbrace{\frac{W_s}{f_s}}_{\text{Queuing Delay}} + \underbrace{\sum_{q \in \mathcal{Q}} \mathbb{I}(q = \eta_{m,i}) \cdot (1 - c_{q,s}) \cdot \tau^{\text{Load}}_q}_{\text{Service Loading Time}} + \underbrace{\frac{b_{m,i}}{f_s}}_{\text{Task Processing Time}} \tag{5}$$

This expression consists of three distinct terms: the first term estimates the queuing delay, calculated by dividing W_s by the CPU frequency of server s. The second term represents the waiting time required to load the libraries associated with the service needed by task $v_{m,i}$, which is incurred only if the corresponding service is not already cached on the cloudlet. The caching status of service q on server s is indicated by the binary variable $c_{q,s}$, where a value of one indicates that the service is cached. This variable is one of the decision variables defined in the problem formulation. In addition, we use the indicator function $\mathbb{I}(q = \eta_{m,i})$, which equals 1 if task $v_{m,i}$ requires service q and 0 otherwise. This ensures that the service loading time τ^{Load}_q is only associated with the specific service needed by the task. The third term corresponds to the processing time required for the task.

4 Problem formulation

The main objective is to minimize the average completion time of user applications. The problem involves two types of binary decision variables: $x_{m,i,s}$ for task offloading and $c_{q,s}$ for service caching. The variable $x_{m,i,s} = 1$ if the i-th task of user m is offloaded to cloudlet s, and 0 otherwise. Similarly, $c_{q,s} = 1$ if service q is cached on cloudlet s, and 0 otherwise. The optimization problem is formulated as follows:

$$\min \quad \frac{1}{M} \sum_{m \in \mathcal{M}} \sum_{s \in \mathcal{S}} x_{m,|\mathcal{V}_m|,s} T_{m,|\mathcal{V}_m|,s} \tag{6a}$$

$$\text{s.t.} \quad T_{m,i,s} = T^*_{m,i,s} + \tau_{m,s}(d_{m,i}) + \tau^{\text{Exe}}_{m,i,s} \qquad \forall s, m, v_{m,i} \in \mathcal{S}, \mathcal{M}, \mathcal{V}_m \tag{6b}$$

$$T^*_{m,i,s} \geq \sum_{s' \in \mathcal{S}} x_{m,j,s'} \left(T_{m,j,s'} + \tau^{\text{Trans}}_{s',s}(l_{m,j}) \right) \quad \forall v_{m,j} \in P_{m,i} \tag{6c}$$

$$\sum_{s\in\mathcal{S}} x_{m,i,s} = 1 \qquad \forall m \in \mathcal{M},\ \forall v_{m,i} \in \mathcal{V}_m \tag{6d}$$

$$\sum_{q\in\mathcal{Q}} c_{q,s} \le K_s \qquad \forall s \in \mathcal{S} \tag{6e}$$

$$x_{m,i,s} \in \{0,1\} \qquad \forall s \in \mathcal{S}, \forall m \in \mathcal{M},\ \forall v_{m,i} \in \mathcal{V}_m \tag{6f}$$

$$c_{q,s} \in \{0,1\} \qquad \forall q \in \mathcal{Q},\ \forall s \in \mathcal{S} \tag{6g}$$

As described by (6), the objective is to minimize the average finishing time of the applications across all users, where the finishing time of each application is determined by the completion of its final task, indexed as $|\mathcal{V}_m|$ for user m.

Constraint (6b) defines the calculation of the completion time $T_{m,i,s}$ for task $v_{m,i}$, based on Equation (4). In this constraint, $T^*_{m,i,s}$ denotes the maximum time required for all data from the immediate predecessor tasks of $v_{m,i}$ to arrive at cloudlet s. The requirement related to this variable is specified by Constraint (6c), which ensures that task $v_{m,i}$ cannot begin execution until all its predecessor tasks $v_{m,j}$ have completed and their outputs have been transmitted to cloudlet s. The assignment of each task to exactly one cloudlet is enforced by Constraint (6d), while the caching capacity of each cloudlet is limited by Constraint (6e). Finally, the binary nature of the decision variables $x_{m,i,s}$ for task offloading and $c_{q,s}$ for service caching is specified by Constraints (6f) and (6g), respectively.

The computational complexity for achieving an optimal solution is exponential in the number of users M, the average number of tasks V per application, the number of servers S, and the number of services Q. Specifically, it is:

$$\mathcal{O}\big(S^{MV}2^{QS}\big) \tag{7}$$

While an ILP solver can provide the optimal solution, solving the problem becomes more difficult as the size of the problem increases. To overcome this difficulty, we introduce a method that integrates DQN for task offloading with an action shaping strategy, and applies a heuristic approach to the caching problem. We provide a detailed explanation of this method in the following section.

5 Dependent task offloading and service caching solution

The introduced problem is divided into two sub-problems. The first involves the caching of services on cloudlets, with each cloudlet making decisions locally. The second pertains to task offloading, where cloudlets decide on the assignment of tasks for their surrounding users. In the following two subsections, we address the solution of these two sub-problems.

5.1 Caching decision

As shown in (5), a part of each task execution time is the service loading latency. Since this component directly impacts the overall finishing time of applications, reducing the service loading time for the tasks offloaded to each cloudlet can help improve application completion times. To support this in a distributed manner, each cloudlet can independently make caching decisions aimed at minimizing the loading time of the services requested by users. The caching problem for each cloudlet $s \in \mathcal{S}$ is formulated as follows:

$$\text{Min} \sum_{m\in\mathcal{M}} \sum_{v_{m,i}\in\mathcal{V}_m} x_{m,i,s} \sum_{q\in\mathcal{Q}} \mathbb{I}(q=\eta_{m,i})(1-c_{q,s})\,\tau_q^{\text{Load}} \tag{8a}$$

$$\text{s.t.} \quad \sum_{q\in\mathcal{Q}} c_{q,s} \le K_s \tag{8b}$$

$$c_{q,s} \in \{0,1\} \qquad \forall q \in \mathcal{Q} \tag{8c}$$

The objective (8a) is to minimize the total service loading time for all tasks offloaded to cloudlet s, assuming that the offloading decisions are known. Constraint (8b) ensures that the number of services cached on cloudlet s does not exceed its capacity K_s. Constraint (8c) enforces that the caching decision variable $c_{q,s}$ is binary. By rearranging the summation, the objective function can be reformulated as follows:

$$\text{Min:} \sum_{q\in\mathcal{Q}} \left(\sum_{m\in\mathcal{M}} \sum_{v_{m,i}\in\mathcal{V}_m} x_{m,i,s}\, \mathbb{I}(q=\eta_{m,i}) \right) (1-c_{q,s})\, \tau_q^{\text{Load}} \tag{9}$$

We define $N_{q,s}$ as the total number of tasks assigned to cloudlet s that request service q, computed as:

$$N_{q,s} = \sum_{m\in\mathcal{M}} \sum_{v_{m,i}\in\mathcal{V}_m} x_{m,i,s}\, \mathbb{I}(\eta_{m,i}=q) \tag{10}$$

Using this notation, the reformulated problem can be defined as follows:

$$\text{Min:} \quad \sum_{q\in\mathcal{Q}} N_{q,s}(1-c_{q,s})\, \tau_q^{\text{Load}} \tag{11}$$

subject to constraints (8b) and (8c).

The solution to this problem for cloudlet s is to select the K_s services with the highest values of $N_{q,s} \times \tau_q^{\text{Load}}$. These values reflect service popularity combined with the cost of loading it. For the selected services, $c_{q,s}$ is set to 1, and for all others, it is set to 0.

Since the exact values of $N_{q,s}$ are not available at decision time, we first note that they could, in principle, be computed recursively. Let $v_{m,i}$ be the task offloaded to cloudlet s, and define $\mathbb{I}(\eta_{m,i}=q)$ as an indicator that equals 1 if the task requires service q and 0 otherwise. The cumulative popularity-cost quantity

$$Y_{q,s}(L) = N_{q,s}(L)\, \tau_q^{\text{Load}}$$

represents the number of times service q has been requested at cloudlet s after L offloaded tasks, multiplied by its loading cost. It can be updated recursively as

$$Y_{q,s}(L) = Y_{q,s}(L-1) + \mathbb{I}(\eta_{m,i}=q)\, \tau_q^{\text{Load}}, \qquad Y_{q,s}(0) = 0. \tag{12}$$

The average of this quantity after L offloaded tasks is given by $\bar{Y}_{q,s}(L) = Y_{q,s}(L)/L$, which can also be expressed in recursive form as

$$\bar{Y}_{q,s}(L) = \bar{Y}_{q,s}(L-1) + \frac{1}{L}\big(\mathbb{I}(\eta_{m,i}=q)\, \tau_q^{\text{Load}} - \bar{Y}_{q,s}(L-1)\big) \tag{13}$$

However, this exact recursion uses a diminishing step size $1/L$, which adapts slowly to changes in offloading behavior. Since task offloading patterns and the resulting service demand at each cloudlet vary over time, we maintain an online ranking variable $H_{q,s}$ that approximates this popularity–cost term and update it using an exponential moving average (EMA). EMA replaces the diminishing step size with a constant α, giving recent observations more weight while still smoothing out noise. This leads to the following update rule:

$$H_{q,s} = H_{q,s} + \alpha \left(\mathbb{I}(\eta_{m,i}=q)\, \tau_q^{\text{Load}} - H_{q,s}\right), \qquad \forall q \in \mathcal{Q}. \tag{14}$$

This EMA-based estimate allows $H_{q,s}$ to remain responsive to the current offloading pattern while providing a stable approximation of the underlying popularity-cost metric used for caching decisions. After performing N updates, the caching decision is made by selecting the top K_s services with the highest rankings in $H_{q,s}$ for each cloudlet s. Initially, all values of $H_{q,s}$ are set to zero. The algorithm is outlined in Algorithm 1.

It is important to note that the update operation for $H_{q,s}$ is performed for every task offloaded to cloudlet s, whereas the caching decision is made after every N updates. Whenever a cloudlet updates its caching decision, it broadcasts the changes to all other cloudlets to ensure consistent service awareness across the network.

Algorithm 1 Caching Decision Algorithm for Cloudlet s

Initialization:
Set initial values for all $H_{q,s}$ to zero.

Update Averaged Values:
for each task $v_{m,i}$ offloaded to cloudlet s **do**
  for each service $q \in \mathcal{Q}$ **do**
    Update $H_{q,s}$ using:
    $H_{q,s} = H_{q,s} + \alpha \left(\mathbb{I}(\eta_{m,i} = q)\tau_q^{\text{Load}} - H_{q,s}\right)$
  end for
end for

Caching Decision:
After N updates, select the top K_s services with the highest rankings in $H_{q,s}$.

5.2 Offloading decision

Our aim in this paper is to introduce a distributed approach for offloading decision-making, considering the dynamic nature of the environment influenced by service caching decisions. Our proposed offloading strategy differs from the previous approach [5] in four key aspects. First, the cloudlet nearest to each user makes the offloading decisions. Second, the state representation in the reinforcement learning model has been modified to include both the global caching status and the destination cloudlets of predecessor tasks. Third, neural network training is based on returns computed from complete application trajectories. Fourth, we incorporate guided reinforcement learning to shape action selection, accelerating the learning process and distinguishing our approach from previous methods.

We model the offloading decision process as a Markov Decision Process and employ DQN to learn optimal policies. To formulate the problem in the context of DQN, the state space, action space, and reward function are specified as follows:

State Space: Let $S_{m,i}$ denote the state associated with task $v_{m,i}$ of user m, as observed by the cloudlet responsible for making the offloading decision. This state consists of the following six components:

$$S_{m,i} = \{b_{m,i}, d_{m,i}, \mathcal{A}_{m,i}^{Pred}, \mathcal{C}, \mathcal{Z}_{m,i}, \mathcal{Z}_{m,i}^{\text{Succ}}\} \tag{15}$$

where $b_{m,i}$ denotes the number of required CPU cycles, and $d_{m,i}$ is the input data size of task $v_{m,i}$. The vector $\mathcal{A}_{m,i}^{\text{Pred}} = [a_{m,i,1}, \ldots, a_{m,i,S}]$ indicates the cloudlet destinations of predecessor tasks of $v_{m,i}$, and each $a_{m,i,s}$ is defined as follows.

$$a_{m,i,s} = \begin{cases} 1 & \text{if at least one predecessor of task } v_{m,i} \\ & \text{is offloaded to cloudlet } s \\ 0 & \text{otherwise} \end{cases} \tag{16}$$

The vector $\mathcal{C} = [c_{q,s}]_{QS\times 1}$ represents the global caching configuration broadcast by cloudlets. The vector $\mathcal{Z}_{m,i} = [z_1^{m,i}, \ldots, z_Q^{m,i}]$ represents the service vector of the current task i of user m, where $z_q^{m,i}$ is defined as follows:

$$z_q^{m,i} = \begin{cases} 1 & \text{if} \quad q = \eta_{m,i} \\ 0 & \text{otherwise.} \end{cases} \tag{17}$$

To account for the impact of task dependencies on offloading decisions, we incorporate the services needed by subsequent tasks into the user state. This is denoted as $\mathcal{Z}_{m,i}^{\text{Succ}} = [z_1^{\text{succ}}, \ldots, z_Q^{\text{succ}}]$ and

is defined as follows:

$$z_q^{\text{succ}} = \begin{cases} 1 & \text{if at least one successor task of } v_{m,i} \\ & \text{requires service } q \\ 0 & \text{otherwise} \end{cases} \tag{18}$$

Action Space: The agent needs to determine which cloudlet to offload its task to. Therefore, the action space is a discrete set $A = \{1, \ldots, S\}$ with S actions. In each state, the agent selects one of the actions.

Reward Function: The aim of this work is to minimize the total execution time of an application, with particular focus on completing its final task as quickly as possible. To capture this objective, the reward at time step t is defined as:

$$r_t = \begin{cases} 0 & \text{if the current task is not the final task,} \\ 1 & \text{if it is the final task and } T_{m,|\mathcal{V}_m|} \leq \bar{D}_m, \\ -1 & \text{if it is the final task and } T_{m,|\mathcal{V}_m|} > \bar{D}_m \end{cases} \tag{19}$$

where $\bar{D}_m$ represents the estimated application deadline for user m and it is updated using the following equation

$$\bar{D}_m = \bar{D}_m + \alpha_{bs}(T_{m,|\mathcal{V}_m|} - \bar{D}_m) \tag{20}$$

where α_{bs} is the updating step size of the estimated deadline [5]. The reward for the final task is set to either $+1$ or -1, depending on whether the application completion time is shorter or longer than the deadline. The deadline is continuously updated to reflect the average application completion time. Since the algorithm consistently aims for positive rewards, training the agent reduces the deadline, as outlined in (20). This suggests that the learning agent is persistently striving to reduce the application's completion time.

As shown in (19), intermediate task rewards are intentionally set to zero because, in DAG-based applications, task completion times are conditionally coupled. The delay of one task may or may not affect its successors, depending on whether it lies on the critical path and on the data-transfer delays between dependent tasks. Since the overall goal is to minimize the completion time of the final task, intermediate rewards could mislead the agent by rewarding locally beneficial but globally harmful actions. Although intermediate rewards are omitted, the effect of each decision on the final outcome is still captured through the return computation, where the terminal reward is propagated backward according to (22).

The proposed algorithm for offloading decisions is outlined in Algorithm 2. The algorithm starts by initializing the neural network parameters in line 1 and setting the initial deadline to zero in line 2. The learning process is executed over multiple iterations, with each iteration beginning by clearing the trajectory in line 4 and selecting the first task, initializing its state in lines 5 and 6. The trajectory, which records states, actions, and rewards throughout the iteration, is crucial for calculating returns and updating the neural network.

Algorithm 2 The proposed algorithm for each user

Initialize neural network parameter θ
Initialize deadline $\bar{D}_m \leftarrow 0$
for each iteration **do**
  Initialize Trajectory $E \leftarrow \{\}$
  Select the first task of the application: t
  Initialize state:
  $s \leftarrow \{b_{m,i}, d_{m,i}, \mathcal{A}_{m,i}^{Pred}, \mathcal{C}, \mathcal{Z}_{m,i}, \mathcal{Z}_{m,i}^{\text{Succ}}\}$
  repeat
    Let $q = \eta_{m,i}$ be the service required by task $v_{m,i}$
    Define $\mathcal{A}_{\text{guided}} = \{s \in \mathcal{S} \mid c_{q,s} = 1\}$
    Select action a using guided RL:

$$a \leftarrow \begin{cases} \text{random from } \mathcal{A}_{\text{guided}} & \text{with prob. } \beta \\ \arg\max_a Q(s,a;\theta) & \text{with prob. } (1-\beta)(1-\epsilon) \\ \text{random action} & \text{with prob. } (1-\beta)\epsilon \end{cases}$$

    Offload the task based on action a and get the finishing time of the task, T_t
    if t has any successor **then**
      No immediate reward: $r \leftarrow 0$
      Pick a task with no remaining predecessors: t'
      Build the state tuple s' based on t'
    else
      Get immediate reward based on the Deadline:

$$r \leftarrow \begin{cases} 1 & \text{if } T_t \le \bar{D}_m \\ -1 & \text{if } T_t > \bar{D}_m \end{cases}$$

      Terminal state: $s' \leftarrow 0$
      Update Deadline $\bar{D}_m$
    end if
    Store transition (s, a, r, t) in E
  until $s' \leftarrow 0$
  for each transition (s, a, r, t) in trajectory E in reverse order **do**
    Compute return value $G_t = r + \gamma \sum_{t' \in \text{Succ}(t)} G_{t'}$
    Store the transition (s, a, G_t) in replay buffer $\mathcal{B}$
  end for
  if iteration mod $N_{\text{update}} == 0$ **then**
    if $|\mathcal{B}| > B$ **then**
      Sample a batch of transitions (s_i, a_i, G_i) from $\mathcal{B}$
      Update the neural network parameters θ by minimizing the loss:

$$L(\theta) = \sum_{i=1}^{\text{batch\_size}} (G_i - Q(s_i, a_i; \theta))^2$$

    end if
  end if
end for

The *guided action selection mechanism* incorporates domain knowledge into the decision-making process. With probability β, the agent selects an action from the set of cloudlets that cache the required service, denoted by $\mathcal{A}_{\text{guided}}$. With the remaining probability $1 - \beta$, the agent follows an ϵ-greedy policy that balances exploitation and exploration. This hybrid strategy enables both informed decision-making and efficient exploration, thereby accelerating the learning process and improving convergence.

The motivation behind using the guided action mechanism lies in the limited environmental knowledge available during the early stages of training. At this point, the only known information is which cloudlets cache the required service—making them more likely to minimize application finishing time. Consequently, the agent initially favors these cloudlets with higher probability. As learning progresses and the agent gains more experience about processing delays, queueing times, and communication latencies, it increasingly relies on its learned policy.

To enable this transition, the guidance probability β is decayed exponentially over time according to:

$$\beta = \max(\beta_{\min}, \beta_0 \cdot \delta_\beta^i), \tag{21}$$

where β_0 is the initial guidance strength, δ_β is the decay factor ($0 < \delta_\beta < 1$), and i is the training episode index.

In line 11, the task is offloaded based on the selected action, and the reward is determined based on whether the task is terminal or has successors, as detailed in lines 12 to 19. If the task is terminal, the reward is set based on the task finishing time relative to the deadline, and the deadline is updated in line 20. The transition, which includes the state, action, reward and task information, is recorded in the trajectory in line 22.

After the iteration concludes, the return for each state is computed based on the reward returns of successor tasks using the following equation:

$$G_t = r + \gamma \sum_{t' \in \mathrm{Succ}(t)} G_{t'} \tag{22}$$

where G_t is the return for task t, γ is the discount factor, r is the immediate reward, and $G_{t'}$ represents the returns of the successor tasks t'. This dependency-based return calculation ensures that the value of each decision accounts for its impact on subsequent tasks in the application workflow.

The computed returns, along with state-action pairs, are then stored as transitions in a replay buffer. After a sufficient number of transitions have been gathered in the replay buffer, a batch is sampled every N_{update} iterations, and the neural network parameters θ are updated. This update involves minimizing the loss function defined as:

$$L(\theta) = \sum_{i=1}^{\text{batch\_size}} \left(G_i - Q(s_i, a_i; \theta)\right)^2 \tag{23}$$

To reduce this loss, the Adam optimization algorithm adjusts the neural network parameters θ based on the gradient of the loss function.

5.3 Complexity analysis

The complexity of the caching decision algorithm can be described as follows. The initialization phase, where values are set for all services and cloudlets, results in a complexity of $\mathcal{O}(SQ)$. During the update phase, where averaged values are recalculated for each task assigned to a cloudlet and each service, the complexity remains $\mathcal{O}(SQ)$. For the caching decision process, selecting the top K_s services based on their rankings involves sorting, which has a complexity of $\mathcal{O}(Q \log Q)$ for each cloudlet. Given that this sorting must be done for all S cloudlets, the complexity for this step is $\mathcal{O}(SQ \log Q)$. Consequently, the overall complexity of the caching decision algorithm is $\mathcal{O}(SQ + SQ \log Q)$, which encompasses the costs of initialization, updates, and decision-making processes.

The complexity of our DRL-based algorithm for each user encompasses both decision-making and training phases, largely influenced by the neural network's architecture. The input layer of the network is determined by the state dimension, given by $SQ + S + 2Q + 3$. For a neural network with two hidden layers, each having n neurons, the computational cost for a single forward pass is $\mathcal{O}((SQ + S + 2Q + 3)n + n^2 + nS)$. Training the network involves processing data in batches of size B, with updates performed every N_{update} iterations, resulting in an amortized complexity of $\mathcal{O}\left(\frac{B}{N_{\text{update}}}\left((SQ + S + 2Q + 3)n + n^2 + nS\right)\right)$. Taking all these factors into account, the complexity of the algorithm for each task decision is

$$\mathcal{O}\left(\left(1 + \frac{B}{N_{\text{update}}}\right)\left((SQ + S + 2Q + 3)n + n^2 + nS\right)\right) \tag{24}$$

When comparing the computational complexities of the DRL-based algorithm and the caching decision algorithm with the complexity of obtaining an optimal solution, the contrast is substantial. For example, with sample parameters $S = 10$, $Q = 10$, $M = 10$, $n = 64$, $N_{\text{update}} = 100$, and $B = 1024$, the complexities are approximately $\mathcal{O}(1.5 \times 10^5)$ operations for the DRL-based algorithm per decision per task and $\mathcal{O}(200)$ operations for the caching decision algorithm. On a 1 GHz CPU (i.e., 10^9 operations per second), these complexities correspond to an offloading decision time on the order of $150\,\mu$s, with negligible additional cost for the caching decision.

In contrast, for the above parameters and assuming 10 tasks per application, the complexity of finding an optimal solution based on Equation (7) is approximately $\mathcal{O}(10^{130})$. This significant difference shows that, even for relatively small parameter values, the computational complexity of the proposed methods is many orders of magnitude lower than that of computing an optimal solution.

5.4 Scalability analysis

Since the proposed framework is designed as a distributed system, it naturally scales with the number of cloudlets and users. In the following, we examine these scalability properties in detail.

Computational scalability with respect to cloudlets. Each cloudlet operates its own DRL agent and caching module independently. According to the complexity analysis, the caching decision at each cloudlet has a complexity of $\mathcal{O}(Q \log Q)$, leading to an aggregate complexity of $\mathcal{O}(SQ \log Q)$ across all S cloudlets. The per-decision complexity of the DRL-based offloading scales with the per-decision cost given in Equation (24), which increases with S and Q through the state dimension SQ. Because each cloudlet makes its decisions independently, there is no centralized bottleneck, and the overall computational load increases linearly with the number of cloudlets S.

User scalability. Offloading decisions are made locally at the nearest serving cloudlet of each user. As the number of users M increases, each cloudlet processes more offloading decisions over time (proportional to the task arrival rate), but the computational cost of a single decision remains bounded by Equation (24). Because the computational cost of each offloading decision does not depend on the total number of users M, increasing the number of users only increases how often decisions are made, not how expensive each decision is. As a result, a cloudlet can serve many users, each running DAG-based applications, while keeping the latency of individual offloading decisions low.

Communication overhead. Caching decisions and their broadcasts occur at a much lower frequency compared to task offloading operations. Furthermore, the overhead associated with caching updates is significantly smaller than the data traffic volume generated by task offloading and inter-cloudlet data transfers. As a result, the overhead from caching broadcasts remains negligible compared to the overall system traffic.

5.5 Convergence discussion

While Q-learning is proven to converge to the optimal policy in finite state spaces [21], using deep neural networks introduces practical stability challenges. Our DRL framework mitigates these through several mechanisms. First, experience replay breaks the temporal correlation between consecutive samples, reducing instability caused by highly correlated updates. Second, the reward function is simple and bounded, which keeps value estimates numerically stable. Third, the adaptive deadline mechanism encourages steady improvement: whenever the agent succeeds, the target becomes slightly tighter, pushing the policy toward faster completion times. Finally, guided action shaping accelerates training by steering early exploration toward service provider cloudlets.

The stability of the overall system also benefits from its interaction with service caching. A key observation is the timescale separation between the two components: offloading decisions evolve rapidly,

while caching updates adjust slowly through an exponential moving average. As the offloading policy stabilizes, the resulting service demand patterns become more predictable, allowing the caching mechanism to converge toward a consistent set of frequently requested services.

Although providing a formal convergence proof for this coupled DRL-caching system is beyond current theoretical tools, the mechanisms described above contribute to its stable behavior in practice. Our simulation results (Section 6) support this observation empirically: the average application completion time decreases and then stabilizes over training iterations.

6 Simulation results

In this section, we begin by describing the simulation configuration, followed by an overview of the baseline methods used for comparison. Finally, we present and discuss the results.

6.1 Simulation setting

Fig. 2 illustrates the simulation environment, where red circles represent cloudlet servers and blue squares denote users. Users and cloudlet servers are distributed throughout the area. The black lines indicate the connections between cloudlets in the server network.

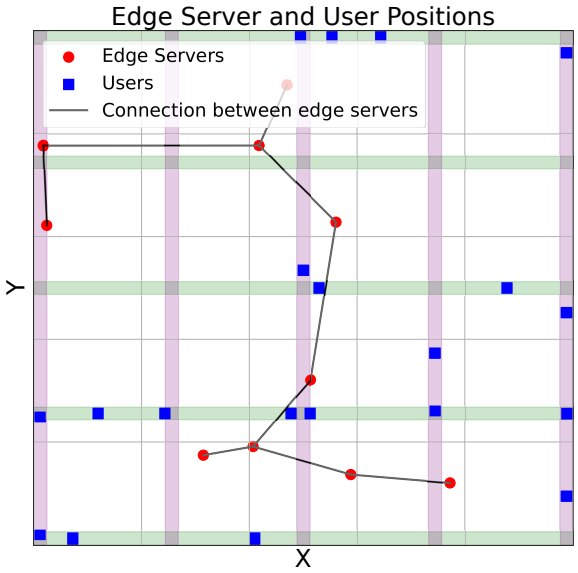

Figure 2: Topology of the simulated environment with 20 users and 10 cloudlet servers.

In our simulation, each user executes a DAG-based application. The structure of these application DAGs is obtained from the Alibaba Trace 2018 dataset [1]. Two representative examples of DAG-based applications are depicted in Fig. 3. Due to the lack of a comprehensive dataset containing all the necessary parameters for our simulation, certain values have been generated based on realistic assumptions.

We assume that users do not change their applications over time. Upon completion, each application restarts, reflecting the behavior of IoT devices with fixed functional roles, or scenarios where a continuously running application is required on mobile devices or autonomous vehicles. Our evaluation focuses on application completion time, which measures the total time required for the average user to complete their entire DAG-based application, from the first task to the final one. This metric reflects the cumulative impact of offloading decisions and service caching.

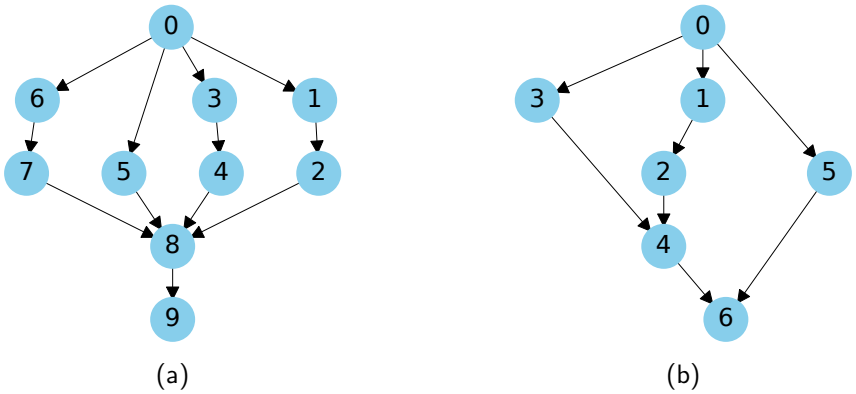

Figure 3: Examples of DAG-based applications from the Alibaba Cluster Trace dataset.

The performance of neural network-based methods is highly influenced by the choice of hyperparameters. In our implementation, we use a neural network with 2 hidden layers, each containing 64 nodes, and a learning rate of 0.001. These values were selected based on preliminary tuning. Network, simulation, and neural network parameters are listed in Table 3.

Table 3: Network and simulation parameters

Category	Parameter	Value / Range
Network Parameters	Number of users	20
	Cloudlet servers	10
	Number of services	10
	Area size	$1\,\text{km} \times 1\,\text{km}$
	Cloudlet–cloudlet rate	10 − 40 Mbps
	Load on cloudlets	0.2 − 100 Mcycles
	CPU frequency	0.2 − 30 Gcycles/s
	Service data length	10 − 1000 MB
	Task data size	up to 1.5 MB
	Task CPU requirement	up to 200 Mcycles
	Bandwidth	15 kHz
	Caching capacity K_s	2
Simulation Parameters	Simulation duration	30,000 steps
	Discount factor γ	0.9
	Learning rate α	0.001
	Exploration rate ϵ	0.01
	β_0	0.9
	$\beta_{\min}$	0.1
	δ_β	0.995
	Batch size B	1024
	N_{update}	100
	Hidden layers	2
	Hidden nodes	64
	Activation function	tanh
	Optimizer	Adam

6.2 Performance evaluation

In our system model, servers operate under realistic constraints and do not know the exact communication latency, queueing delay, or computational capacity of other servers. Since these limitations stem from privacy, security, and signaling overhead, comparing our algorithm with methods that assume full global knowledge would be unfair. We therefore select baselines that follow similar information assumptions and include both guided and unguided reinforcement learning variants, where guidance means informing the agent about service availability on different servers.

Random: Each task is offloaded to a randomly selected cloudlet, regardless of proximity, load, or service availability.

Nearest Server: Each task is always offloaded to the geographically closest cloudlet, aiming to reduce transmission delay but ignoring service availability and server capabilities.

Nearest Server with Service (Greedy): Tasks are offloaded to the nearest cloudlet that provides the required service; if none exist, the task is assigned to the closest cloudlet. This greedy policy prioritizes immediate service access and ignores other factors.

DQN without Dependency and Service Awareness (DQN-WDSA): A Deep Q-Network baseline with a simplified state that omits task dependencies and required services, used to assess the benefit of including this information.

Actor–Critic (Unguided): The dependency-aware Actor–Critic offloading method from [22], used to compare our approach with an existing RL solution that models task dependencies.

Actor–Critic (Guided): The same Actor–Critic framework as above, augmented with the guided action selection strategy used in our method to isolate the effect of guidance in a different architecture.

Unguided Proposed Algorithm: A variant of our algorithm without guided action selection. The agent uses DAG and service information in the state but chooses actions solely from learned Q-values via ϵ-greedy, highlighting the impact of guidance in the full method.

Fig. 4 illustrates the convergence behavior of all compared algorithms throughout training. During the early iterations, when the proposed algorithm is still in its exploration phase and has not yet converged to an effective policy, the *Nearest with Service (Greedy)* baseline initially achieves lower average application finishing time. This is expected, as it always offloads tasks to cloudlets that already host the required services, effectively avoiding service loading latency from the beginning. However, as the training progresses, the proposed algorithm gradually outperforms all baselines. Through continuous exploration, it gradually learns the structure and resource distribution of the environment. It then identifies more capable cloudlet servers and decides where to offload tasks based on the learned environment and the service requirements of incoming tasks.

In Fig. 5, we present the ablation analysis, where the contribution of each component of the algorithm is examined. All variants are compared against the *Nearest Server* baseline, which does not employ any learning mechanism; each added component incrementally improves performance. Introducing a simple decision-maker that uses basic task features—CPU cycles and input data size—reduces the average finishing time from about $0.37s$ to $0.23s$ (roughly a 40% reduction). Adding *service-awareness*, which includes both the service requirement of the current task and the services needed by upcoming tasks, further decreases the finishing time to approximately $0.21s$. Incorporating *dependency-awareness*, by embedding the assigned servers of previously executed tasks, reduces it further to around $0.15s$. Finally, integrating the *guiding mechanism*, which leverages broadcast information about cached services at each server, yields the lowest finishing time of about $0.075s$, corresponding to an overall reduction of nearly 80% compared to the *Nearest Server* baseline. The figure also shows that, for the full proposed method, replacing static caching (average finishing time around $0.25s$) with the proposed dynamic caching strategy reduces the finishing time to about $0.075s$, confirming the substantial impact of the caching design.

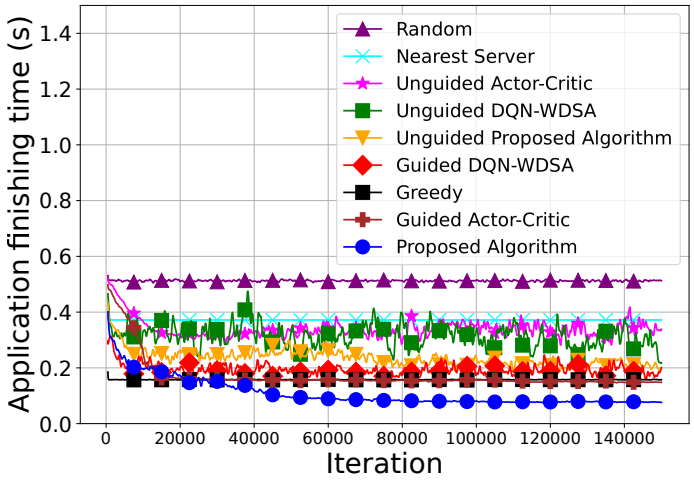

Figure 4: Convergence behavior of different algorithms over training iterations. The proposed method initially explores the environment, resulting in higher application finishing time. Over time, it learns the environment structure and resource distribution, enabling more effective offloading decisions based on server capabilities and service requirements.

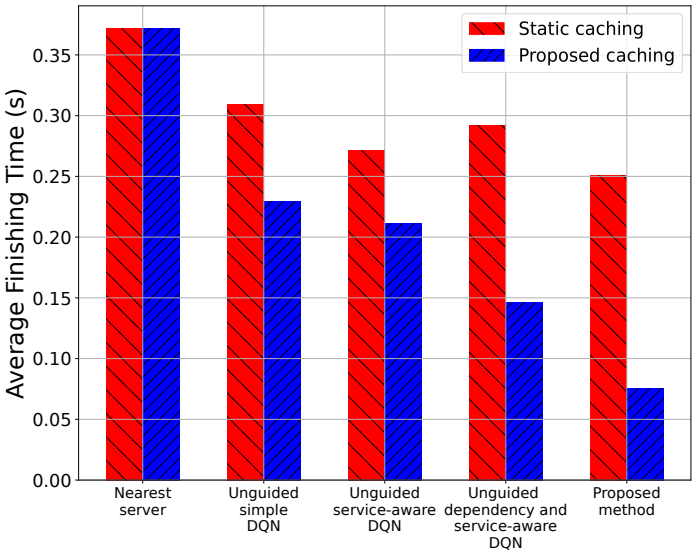

Figure 5: Ablation study showing the contribution of each component in the proposed framework. Caching, service-awareness, Dependency-awareness, and guiding progressively reduce the application finishing time, with their combination in the proposed method achieving the best overall performance.

Fig. 6 shows the components that contribute to the overall application execution time, including computation latency, service loading latency, and waiting latency. Each algorithm focuses on reducing specific components of the total latency. For example, the greedy algorithm achieves zero service loading latency by always selecting a cloudlet that already hosts the required service. However, it performs poorly in other components such as waiting time and computation latency. In contrast, the proposed algorithm does not focus on reducing a single component but instead balances all latency components. By maintaining this balance, it achieves the lowest total execution time among all methods.

For all figures, results are averaged over 10 independent simulation runs with different random seeds. In each run, the network topology and system parameters are randomly generated to reflect diverse deployment scenarios. Error bars denote 95% confidence intervals computed using the standard

error of the mean and the Student t-distribution. Unless stated otherwise, the number of services is fixed to 10.

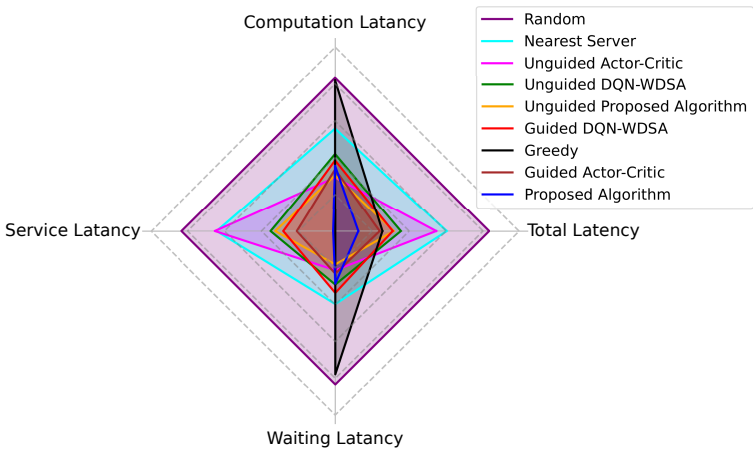

Figure 6: Breakdown of application finishing time into its main components: computation latency, service loading latency, and waiting latency. While some baselines, such as the greedy algorithm, reduce specific components like service loading, the proposed method balances all latency types, resulting in the lowest total execution time.

Fig. 7 shows the average application finishing time as a function of the number of cloudlets. As the number of cloudlets increases, the finishing time consistently decreases due to shorter average user–cloudlet distances and greater server diversity, which enables more flexible offloading and service placement. When the number of cloudlets is small, caching capacity is limited and greedy strategies that select the nearest cloudlet with the required service perform well. As the system scales, the larger offloading and caching decision space allows learning-based methods—particularly the proposed algorithm—to better exploit variations in load, latency, and service availability by jointly considering task characteristics and DAG dependencies.

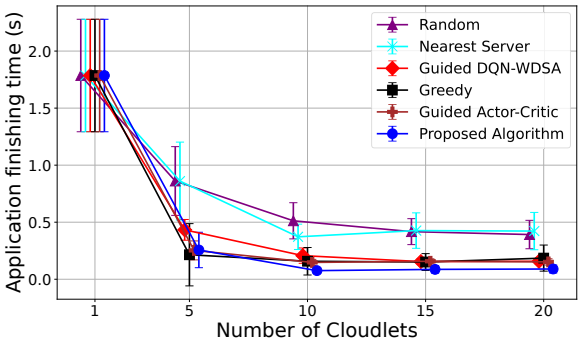

Figure 7: Impact of the number of cloudlets on the average application finishing time. As the number of cloudlets increases, the proposed method outperforms other approaches by leveraging greater flexibility in offloading and service caching.

Fig. 8 shows how the average application finishing time varies with the number of distinct services in the system. The number of cloudlet servers was fixed across all configurations at 10, with each cloudlet capable of caching at most two services. As the number of services increases, the application finishing time rises across all methods. This trend is expected, as a larger service set reduces the probability that a required service is already cached at a nearby cloudlet. Consequently, tasks are more likely to experience service loading delays, either due to fetching services from other cloudlets or from the remote cloud. When the total number of services is relatively small (i.e., less than or equal

to the network-wide caching capacity of 20), the proposed algorithm can fully exploit the available caching space. It learns to coordinate offloading and caching decisions to minimize both service loading and processing delays. However, once the number of services exceeds the total caching capacity of the network, the algorithm faces fundamental limitations: some services cannot be cached anywhere, and loading delays become inevitable. In this worst-case scenario, the proposed method continues to perform competitively, achieving an application finishing time comparable to the greedy algorithm. This robustness highlights the algorithm adaptability and its ability to make efficient offloading decisions even in service-constrained environments.

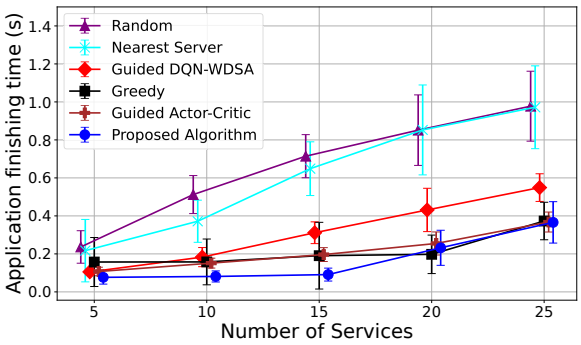

Figure 8: Impact of the number of distinct services on average application finishing time. The proposed method exhibits slower growth before reaching the caching capacity limit.

Fig. 9 illustrates how the average application finishing time changes as the combined size of task input and output data increases, ranging from 0.5 MB to 3.0 MB. The number of cloudlet servers and services is fixed across all configurations to 10. As expected, the finishing time increases as data size grows. This rise is mainly due to longer transmission delays. The proposed algorithm performs best across all data sizes and shows the smallest increase in latency as data volume grows. This is because it offloads tasks more intelligently by taking the size of the data into account. In contrast, algorithms like Random or Nearest Server select cloudlets without considering data volume, which leads to inefficient decisions when communication costs are high.

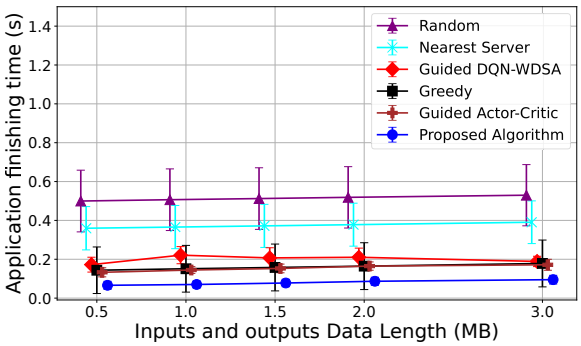

Figure 9: Impact of varying input and output data size on average application finishing time. The proposed method consistently achieves lower latency and is less sensitive to data size increases, due to its data-aware offloading strategy.

Fig. 10 illustrates how the average application finishing time varies with the size of service data, ranging from 100 MB to 5000 MB. As expected, most algorithms experience an increase in finishing time as service size grows. However, this trend is less severe for the greedy baseline and for the proposed method, both of which explicitly consider service availability in the offloading process. In particular, the proposed algorithm ensures that services are cached at the selected cloudlet whenever possible and optimizes task placement to reduce both queuing and processing delays.

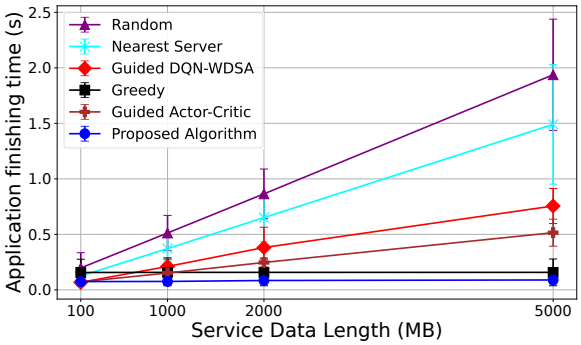

Figure 10: Impact of service data size on application finishing time. All methods experience increasing latency as service data size grows, while the proposed method and greedy baseline show slower growth due to service awareness.

Fig. 11 illustrates how the average application finishing time is affected by variations in the maximum data rate between cloudlet servers, ranging from 10 Mbps to 80 Mbps. In all configurations, the number of cloudlet servers and services is fixed. As shown in the figure, the relative performance of the algorithms varies across different bandwidth conditions. When the inter-cloudlet bandwidth is high, even reinforcement learning methods such as Guided DQN and Guided A2C perform well—sometimes matching or exceeding the performance of greedy algorithms. This is because the service-loading delay becomes negligible, reducing the advantage of strictly offloading to servers that already cache the required services.

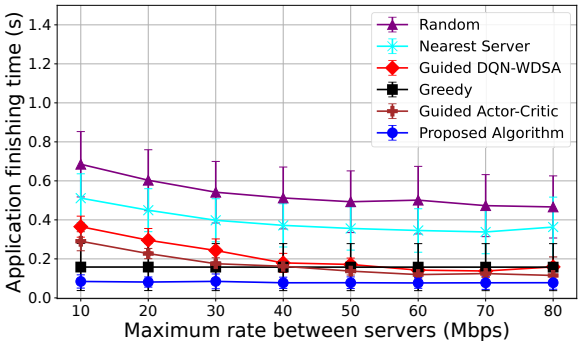

Figure 11: Impact of maximum data rate between cloudlet servers on average application finishing time. Higher inter-cloudlet bandwidth reduces service-loading delays and decreases the performance gap between methods.

What distinguishes the proposed algorithm is its ability to adapt to these conditions automatically. It learns to detect situations where remote service fetching is no longer a major bottleneck and adjusts its offloading policy accordingly. As a result, the proposed method consistently maintains the best performance across all bandwidth levels.

7 Conclusion and future work

In this paper, we proposed a distributed framework for joint task offloading and service caching in cloudlet-based edge computing systems with DAG-based applications. Unlike existing studies that address task dependencies or service caching in isolation, our approach jointly considers task dependencies and service caching decisions within a fully distributed learning framework. In the proposed system, each cloudlet independently makes offloading decisions for nearby users and manages its own service caching based on local observations. Task offloading is learned using deep reinforcement learning with a state representation that explicitly captures task dependencies, service requirements, and

caching status. To improve learning efficiency, action selection is guided during early training stages using service availability information, which significantly accelerates convergence. Simulation results show that the proposed method consistently outperforms baseline approaches across a wide range of system configurations. The results also confirm that jointly integrating dependency awareness, service caching, and guided learning is key to achieving lower application completion times in distributed edge environments. As future work, we plan to extend the framework to scenarios with mobile users. User mobility introduces additional dynamics in connectivity and service demand, where caching decisions may not adapt quickly enough to frequent location changes. Incorporating mobility prediction to proactively adjust caching strategies is a promising direction for further improvement.

References

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

[2] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing, MCC '12, page 13–16, New York, NY, USA, 2012. Association for Computing Machinery.

[3] Xiangchun Chen, Jiannong Cao, Yuvraj Sahni, Shan Jiang, and Zhixuan Liang. Dynamic task offloading in edge computing based on dependency-aware reinforcement learning. IEEE Transactions on Cloud Computing, 12(2):594–608, 2024.

[4] Zhaoyu Gan, Rongheng Lin, and Hua Zou. A multi-agent deep reinforcement learning approach for computation offloading in 5g mobile edge computing. In 2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid), pages 645–648, a, 2022. IEEE.

[5] 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, MobiWac '23, page 91–97, New York, NY, USA, 2023. Association for Computing Machinery.

[6] Mohammad Goudarzi, Marimuthu Palaniswami, and Rajkumar Buyya. A distributed application placement and migration management techniques for edge and fog computing environments. In 2021 16th Conference on Computer Science and Intelligence Systems (FedCSIS), pages 37–56, a, 2021. IEEE.

[7] Yun Chao Hu, Milan Patel, Dario Sabella, Nurit Sprecher, and Valerie Young. Mobile edge computing—a key technology towards 5g. ETSI white paper, 11(11):1–16, 2015.

[8] Liang Huang, Suzhi Bi, and Ying-Jun Angela Zhang. Deep reinforcement learning for online computation offloading in wireless powered mobile-edge computing networks. IEEE Transactions on Mobile Computing, 19(11):2581–2593, 2020.

[9] Sundas Iftikhar, Sukhpal Singh Gill, Chenghao Song, Minxian Xu, Mohammad Sadegh Aslanpour, Adel N. Toosi, Junhui Du, Huaming Wu, Shreya Ghosh, Deepraj Chowdhury, Muhammed Golec, Mohit Kumar, Ahmed M. Abdelmoniem, Felix Cuadrado, Blesson Varghese, Omer Rana, Schahram Dustdar, and Steve Uhlig. Ai-based fog and edge computing: A systematic review, taxonomy and future directions. Internet of Things, 21:100674, 2023.

[10] 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.

[11] Zhongqiang Luo and Xiang Dai. Reinforcement learning-based computation offloading in edge computing: Principles, methods, challenges. Alexandria Engineering Journal, 108:89–107, 2024.

[12] 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.

[13] Mazliza Othman, Sajjad Ahmad Madani, Samee Ullah Khan, et al. A survey of mobile cloud computing application models. IEEE communications surveys & tutorials, 16(1):393–413, 2013.

[14] Shengli Pan, Zhiyong Zhang, Zongwang Zhang, and Deze Zeng. Dependency-aware computation offloading in mobile edge computing: A reinforcement learning approach. IEEE Access, 7:134742–134753, 2019.

[15] Pawani Porambage, Jude Okwuibe, Madhusanka Liyanage, Mika Ylianttila, and Tarik Taleb. Survey on multi-access edge computing for internet of things realization. IEEE Communications Surveys & Tutorials, 20(4):2961–2991, 2018.

[16] Ce Shang, Youliang Huang, Yan Sun, and Mohsen Guizani. Joint computation offloading and service caching in mobile edge-cloud computing via deep reinforcement learning. IEEE Internet of Things Journal, 2024.

[17] Qiaoqiao Shen, Bin-Jie Hu, and Enjun Xia. Dependency-aware task offloading and service caching in vehicular edge computing. IEEE Transactions on Vehicular Technology, 71(12):13182–13197, 2022.

[18] Chang Shu, Zhiwei Zhao, Yunpeng Han, and Geyong Min. Dependency-aware and latency-optimal computation offloading for multi-user edge computing networks. In 2019 16th Annual IEEE International Conference on Sensing, Communication, and Networking (SECON), pages 1–9, a, 2019. IEEE, IEEE.

[19] Chuan Sun, Xiuhua Li, Chenyang Wang, Qiang He, Xiaofei Wang, and Victor CM Leung. Hierarchical deep reinforcement learning for joint service caching and computation offloading in mobile edge-cloud computing. IEEE Transactions on Services Computing, 17(4):1548–1564, 2024.

[20] Ming Tang and Vincent W.S. Wong. Deep reinforcement learning for task offloading in mobile edge computing systems. IEEE Transactions on Mobile Computing, 21(6):1985–1997, 2022.

[21] Christopher JCH Watkins and Peter Dayan. Q-learning. Machine learning, 8(3):279–292, 1992.

[22] Han Xiao, Changqiao Xu, Yunxiao Ma, Shujie Yang, Lujie Zhong, and Gabriel-Miro Muntean. Edge intelligence: A computational task offloading scheme for dependent iot application. IEEE Transactions on Wireless Communications, 21(9):7222–7237, 2022.

[23] Jie Xu, Lixing Chen, and Pan Zhou. Joint service caching and task offloading for mobile edge computing in dense networks. In IEEE INFOCOM 2018 - IEEE Conference on Computer Communications, pages 207–215, 2018.

[24] Jianbin Xue, Qingqing Wu, and Haijun Zhang. Cost optimization of uav-mec network calculation offloading: A multi-agent reinforcement learning method. Ad Hoc Networks, 136:102981, 2022.

[25] Zheng Xue, Chang Liu, Canliang Liao, Guojun Han, and Zhengguo Sheng. Joint service caching and computation offloading scheme based on deep reinforcement learning in vehicular edge computing systems. IEEE Transactions on Vehicular Technology, 72(5):6709–6722, 2023.

[26] Zhixiu Yao, Shichao Xia, Yun Li, and Guangfu Wu. Cooperative task offloading and service caching for digital twin edge networks: A graph attention multi-agent reinforcement learning approach. IEEE Journal on Selected Areas in Communications, 41(11):3401–3413, 2023.

[27] Shuai Yu, Xu Chen, Zhi Zhou, Xiaowen Gong, and Di Wu. When deep reinforcement learning meets federated learning: Intelligent multitimescale resource management for multiaccess edge computing in 5g ultradense network. IEEE Internet of Things Journal, 8(4):2238–2251, 2021.

[28] Gongming Zhao, Hongli Xu, Yangming Zhao, Chunming Qiao, and Liusheng Huang. Offloading tasks with dependency and service caching in mobile edge computing. IEEE Transactions on Parallel and Distributed Systems, 32(11):2777–2792, 2021.

[29] Liang Zhao, Zijia Zhao, Ammar Hawbani, Zhi Liu, Zhiyuan Tan, and Keping Yu. Dynamic caching dependency-aware task offloading in mobile edge computing. IEEE Transactions on Computers, 74(5):1510–1523, 2025.