RESEARCH PAPER
Future Research in AI & IoT, 2025, 1(1)
Homepage: https://frai.journals.umz.ac.ir/
DOI: 10.22080/frai.2025.29341.1018

MDEU-A2C: A Mobility, Deadline, Energy and Utilization-Aware Multi-Agent A2C Scheduling Approach to Support Fog and Edge Computing in IoT Applications

Armin Mohammadi Ghaleh[1], Sayed Gholam Hassan Tabatabaei[2]*

[1] *K. N. Toosi University of Technology, Tehran, Iran, armin.m.ghaleh@gmail.com*

[2] *Department of Electrical and Computer Engineering, Malek-e-Ashtar University of Technology, Tehran, Iran, tabatabaei@mut.ac.ir*

Article Info
Received 26 May 2025
Accepted 01 June 2025
Available online 01 June 2025

Keywords:
Mobile Edge Computing (MEC);
Fog and Edge Computing (FEC);
Multi-Agent Reinforcement Learning;
Advantage Actor-Critic (A2C);
Decentralized Scheduling.

Abstract:
Mobile Edge Computing reduces latency and response time by bringing computational resources closer to end-user. However, user mobility poses a significant challenge, as users continuously move between coverage areas of different edge nodes with limited range. This dynamic environment demands efficient scheduling mechanisms that can adapt to user movement while meeting application deadlines and optimizing edge resource utilization. This paper proposes an approach for scheduling based on Deep Reinforcement Learning, specifically using an Advantage Actor-Critic architecture within a Fog and Edge computing framework for IoT applications. The method enables distributed decision-making by deploying actor agents at edge nodes and a centralized critic at the fog node, facilitating continuous adaptation through system-wide feedback. User mobility is addressed using location prediction via RNN models embedded at each edge node, allowing proactive and informed offloading decisions. Experimental results demonstrate the proposed approach significantly improves task completion rate by 50%, failure rate by 26%, and response latency by 60%, while also adapting well to dynamic environments, outperforming state-of-the-art methods in real-world-inspired scenarios.

© 2025 University of Mazandaran

***Corresponding Author:** tabatabaei@mut.ac.ir

Supplementary information: Supplementary information for this article is available at https://frai.journals.umz.ac.ir/

Please cite this paper as: Mohammadi Ghaleh, A., & Tabatabaei, S. G. H. (2025). MDEU-A2C: A Mobility, Deadline, Energy and Utilization Aware Multi-Agent A2C Scheduling Approach to Support Fog and Edge Computing in IoT Applications. Future Research on AI and IoT, 1(1), 42-56. DOI: 10.22080/frai.2025.29341.1018

1. Introduction

The rise of data-intensive applications in areas such as autonomous driving, augmented reality, and mobile health has led to the development of Mobile Edge Computing (MEC), which brings data processing and services closer to end-users and IoT devices [1]. By shifting computation from centralized cloud data centers to distributed edge nodes, MEC can significantly reduce latency, improve real-time responsiveness, and decrease network load [2]. However, MEC systems face considerable challenges due to the limited computational resources of edge nodes, which calls for efficient, adaptive scheduling to manage workloads in real-time.

To address these challenges, Fog and Edge Computing (FEC), a layered model linking resource-limited edge devices with more powerful intermediate fog nodes, has emerged as a promising framework [3]. This architecture enables dynamic task offloading and resource sharing between edge and fog, effectively extending the system's capabilities [4]. Yet, scheduling in MEC remains complex, as it must account for device heterogeneity, limited resources, and client mobility, which may require frequent task migrations to keep up with moving clients [5]. Conventional scheduling methods, often based on heuristics or rules, struggle to adapt to these dynamic and diverse environments [6].

Deep Reinforcement Learning (DRL) offers a flexible solution, with its ability to adapt continuously to changing system states and autonomously develop optimized scheduling strategies [7]. DRL-based scheduling has proven effective in MEC, supporting goals like reduced latency, energy efficiency, and intelligent task offloading [8]. However, existing DRL approaches frequently assume static conditions or overlook complexities such as mobile clients and edge node diversity, which limits their applicability in real-world scenarios [9].

To fill this gap, we propose a distributed Advantage Actor-Critic (A2C) scheduling approach designed to handle MEC's resource constraints, client mobility, and heterogeneous environment. By dynamically distributing tasks based on real-time system states, our approach

© 2025 by the authors. Licensee FRAI, Babolsar, Mazandaran. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC-BY) license (https://creativecommons.org/licenses/by/4.0/deed.en)

ISSN 3092-7552

optimizes task placement according to each node's capabilities, current load, and mobility patterns, thereby achieving efficient task management across the MEC landscape.

Our main contributions are as follows:

1. **Decentralized Decision-Making**: Each edge node operates independently with an actor module, deciding whether to process tasks locally or offload them to other nodes, minimizing communication delays and improving scalability.
2. **Centralized Feedback from Fog Node**: Acting as a central critic, the fog node gathers system-wide information and provides feedback to edge nodes, enhancing task scheduling coherence.
3. **Mobility Prediction**: Using an RNN model to predict the future location of client devices to make task scheduling more efficient.
4. **Offload/Migration Destination**: Find the best edge by creating a 3-dimensial Pareto Set and offloading or migrating a task to it.
5. **Real-Time Simulation Testing**: We validate our approach with real-time simulations, using Docker containers to emulate a realistic FEC environment and assess performance.

The rest of the paper is organized as follows: Section II reviews related works; Section III presents the system model; Section IV details the proposed scheduling approach; Section V evaluates our approach; and Section VI concludes the paper.

2. Related Work

Mobile and Vehicular Edge Computing (MEC and VEC) have gained significant traction as frameworks to enable real-time data processing closer to end-users, addressing the latency and resource constraints of traditional cloud computing. Given the dynamic and resource-limited nature of edge environments, task scheduling and resource allocation have been extensively studied, with recent approaches focusing on leveraging Deep Reinforcement Learning (DRL) for intelligent, adaptive scheduling. Each of the reviewed methods below addresses specific gaps in MEC and VEC task scheduling but falls short of providing a comprehensive solution that addresses all the complexities of highly dynamic, multi-user edge environments.

The [10] scheduling model adopts a Multi-action Environment-adaptive Proximal Policy Optimization (MEPPO) algorithm to tackle energy efficiency and priority awareness in VEC, especially under fluctuating vehicular traffic and variable resource availability. MEPPO addresses these gaps by jointly scheduling tasks and optimizing resource allocation, focusing on dynamic priority assignment and power control for efficient energy consumption. Although it effectively manages time-sensitive tasks in mobile environments, it does not explicitly address the complexities introduced by task dependencies or large-scale, multi-user systems.

In parallel, MARINA [11], a mobility and deadline-aware scheduling mechanism, focuses on real-time task scheduling in VEC environments with high vehicular mobility. MARINA utilizes LSTM-based mobility prediction and Pareto optimization to prioritize tasks by deadline while leveraging a Bin Covering Problem (BCP)-based heuristic for efficient task distribution across edge resources. While it improves deadline adherence and resource utilization, its heuristic-based design may struggle to adapt rapidly in highly dynamic and heterogeneous environments, particularly as the number of users and tasks grows.

Addressing decentralized decision-making, DOSA [12] employs a Double Deep Q-Network (Double-DQN) combined with Dueling DQN and Prioritized Experience Replay, allowing each edge device to make independent task scheduling decisions without centralized control. This decentralized model reduces communication overhead and enables concurrency by processing multiple tasks simultaneously across edge nodes. However, DOSA's focus on concurrency does not account for task dependencies, which are critical in applications requiring sequential task execution and optimal resource sharing.

In scenarios requiring freshness of information, an Age-Based DRL approach [13] incorporates Post-Decision States (PDS) with Deep Deterministic Policy Gradient (DDPG) to directly optimize the Age of Information (AoI). This method redefines AoI for event-driven data updates, making it suitable for applications with strict real-time data requirements. However, its focus on AoI does not translate well to multi-task, multi-user MEC environments where task interdependencies and varying user demands complicate scheduling dynamically.

For edge-cloud systems with significant variability, the A2C-DRL [8] framework utilizes Advantage Actor-Critic (A2C) DRL to balance task loads across edge resources dynamically. Its decentralized scheduling allows for distributed load balancing and rapid task assignment, leveraging prioritized experience replay to enhance learning speed. While effective in improving resource utilization in edge-cloud systems, it does not sufficiently address task dependencies or the rapid changes in resource demand common in dense, multi-user edge environments.

A distinct solution for nonstationary environments is Meta-PPO [14], a multiagent meta-reinforcement learning approach designed for noncooperative, multi-user MEC systems. By integrating Model-Agnostic Meta-Learning (MAML) within a Proximal Policy Optimization (PPO) framework, Meta-PPO enables each agent (user) to learn adaptive scheduling policies based on prior knowledge, allowing rapid adjustment in nonstationary edge environments. Though powerful in handling competitive environments, Meta-PPO's multiagent design assumes static task requirements, lacking explicit mechanisms to handle interdependent tasks or dynamic priority shifts.

For collaborative VEC applications, an asynchronous A3C-based DRL approach combines V2V and V2I offloading for cross-layer resource orchestration [15]. By

integrating hybrid offloading strategies, the model enables multi-resource orchestration across vehicle, edge, and cloud layers. However, its design does not fully capture the requirements of non-cooperative, large-scale edge networks where independent scheduling agents must balance dependencies and dynamically adapt to varying task priorities.

In addressing task dependencies, a Graph Attention Network (GAT) integrated with Proximal Policy Optimization (PPO) models dependent task offloading in edge environments by encoding tasks as Directed Acyclic Graphs (DAGs) [16]. This approach uses the GAT to capture task dependencies, allowing the DRL scheduler to manage complex offloading in multi-user settings. Despite effectively handling task dependencies, this GNN-based approach is limited in addressing resource variability and user mobility common in real-time MEC environments.

The reviewed methods address various critical aspects of MEC and VEC scheduling, including mobility prediction, deadline sensitivity, dependency management, and resource adaptability. However, a gap remains in efficiently integrating decentralized decision-making, adaptability to dynamic multi-user environments, and dependency-aware scheduling in highly heterogeneous edge systems. Many current approaches focus on specific elements, such as mobility or energy efficiency, without fully addressing the combined complexities of heterogeneous environments, task dependencies, and real-time client mobility. This gap highlights the need for a more comprehensive framework that can dynamically adapt to the unique demands of MEC environments while ensuring efficient resource usage across diverse edge devices. Table 1 summarizes the reviewed research.

Table 1. Summary of Related Works

Work	Method	Offload	Migration	Energy	Bandwidth	Computation Resource	Mobility
[1]	PPO	x		x	x	x	x
[2]	Pareto Set, BCP	x	x			x	x
[3]	Double-DQN, Dueling DQN	x				x	
[4]	DDPG			x			x
[5]	A2C	x		x		x	x
[6]	Meta-RL					x	x
[7]	A3C	x		x	x	x	x
[8]	PPO	x				x	
Proposed	A2C	x	x	x	x	x	x

3. System Model and Problem Definition

This section presents the system model, including the edge device and task representations, network and communication models, and computation models. We also define the optimization objective to minimize task execution time, energy consumption, and resource utilization across the network.

3.1. Edge Device Model

Let $E_n = \{e_1, e_2, \dots, e_n\}$ denote the set of edge devices in the system, where n represents the total number of edge devices. Each edge device e_i is defined as:

$$e_i =< ID_{e_i}, C_{e_i}, P_{e_i}, B_{e_i}, E_{e_i} > \quad (1)$$

where:

- ID_{e_i}: Unique identifier of the edge device e_i.
- C_{e_i}: Available CPU resources of e_i , represented as the number of CPU cycles per second.
- P_{e_i}: RAM resources of e_i, representing the memory capacity in GB.
- B_{e_i}: Bandwidth resources available to e_i, representing the maximum data transmission capacity in Mbps.
- E_{e_i}: Current energy level in watt, relevant for battery-powered edge devices

3.2. Task Model

Let $R_k = \{r_1, r_2, \dots, r_k\}$ represent the set of tasks, where k is the total number of tasks in the system. Each task r_i is defined as:

$$r_i =< ID_{r_i}, I_{r_i}, D_{r_i}, Y_{r_i} > \quad (2)$$

where:

- ID_{r_i}: Unique identifier of the task r_i.
- I_{r_i}: Data size of r_i, representing the input data size required for processing in bytes.
- D_{r_i}: Deadline by which the task (r_i) must be completed.
- Y_{r_i}: CPU cycles required for r_i, representing the total computation demand.

3.3. Decision Model

The action A taken for a task r_i can be represented as:

$$A = \{0,1\} \quad (3)$$

where:

- $A = 0$: The task continues running on the local edge device without migration.

- $A = 1$: Task is either offloaded to another edge or migrated, depending on whether it is actively executing.

3.4. Network Model

The network model includes edge-to-edge (E2E) and edge-to-fog (E2F) communication links, with different data transmission rates calculated based on the specific channel characteristics. This work, does not consider communication links between client and edge devices (C2E) and keeps track of everything from the moment a task arrives at an edge device until it finishes its execution, as illustrated in Figure 1. We use and modify the network model of [1].

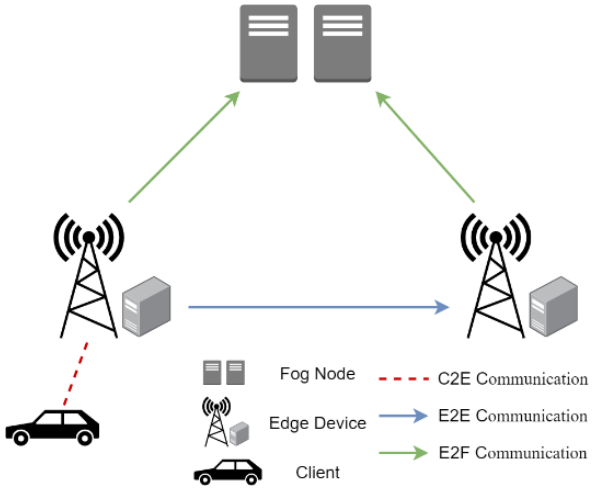

Figure 1. Network Model

3.4.1.Edge-to-Edge (E2E)

Data transmission rate R between two edge devices i and j is given by:

$$R_{i,j} = \eta_j . B_i . log_2(1 + SNR_{i,j}) \quad 4$$

where:

- η: Proportion of bandwidth allocated to this transmission, with $\sum \eta_j \leq 1$. By utilizing the full available bandwidth, more data can be transmitted simultaneously, resulting in higher data transmission rates.
- B: Total bandwidth resource.
- SNR (Signal-to-Noise Ratio) for E2E is defined as:

$$SNR_{i,j} = \frac{P_{i,j} . G_{i,j}}{\varepsilon_{i,j} . d_{i,j} + {\sigma_{i,j}}^2} \quad 5$$

where $P_{i,j}$ is the transmit power, $G_{i,j}$ is the channel gain, $\varepsilon_{i,j}$ represents path loss, $d_{i,j}$ is the transmission distance, and ${\sigma_{i,j}}^2$ is the Gaussian noise.

3.4.2.Edge-to-Fog (E2F)

Data transmission rate R from an edge device i to the fog node g is given by:

$$R_{i,g} = \eta_i . B_g . log_2(1 + SNR_{i,g}) \quad 6$$

where the SNR for E2F communication is defined as:

$$SNR_{i,g} = \frac{P_{i,g} . G_i}{I_{i,g} + {\sigma^2}_{i,g}} \quad 7$$

where $I = \sum P_{i,g} . G_g$ represents the interference noise from other transmissions, with other parameters as defined above.

3.5. Computation Model

The task execution time T and energy consumption E are calculated based on the selected action A:

3.5.1.Local Execution ($A = 0$)

Execution time T includes only the local computation time T^{exe} and queuing time T^{queue}, given by:

$$T = T^{exe} + T^{queue} \quad 8$$

where $T^{exe} = \frac{Y}{C}$ with Y as the CPU cycles required and C as the CPU resource of the edge device.

Energy consumption E for local execution is:

$$E = \varepsilon . (f)^{\gamma} . Y \quad 9$$

where f is the CPU frequency and γ is a constant reflecting the energy coefficient of the edge device.

3.5.2.Migration/Offload ($A = 1$)

Execution time T includes memory dump of executing task T^{dump}, transmission time T^{trans}, execution time at destination edge device T^{exe}, and any potential registry and queue delays:

$$T = T^{dump} + T^{trans} + T^{exe} + \delta . T^{registry} + T^{queue} \quad 10$$

where δ is a binary indicator for whether registry time applies. If source code for executing the task cannot be provided by edge device, fog should provide it and send the source code to the edge node, hence $T^{registry}$ defined as follows:

$$T^{registry} = 2 * T^{fog-trans} + T^{fog-exec} \quad 11$$

where $T^{fog-trans}$ is transmission time to fog and $T^{fog-exec}$ is source code packaging time.

The T^{dump} is pre-migration process's time, this procedure includes stop task's process (SIGSTOP), dump any information exists in edge's memory and package all files, including any intermediate files that task may create.

Energy consumption E for migration is calculated as:

$$E = \varepsilon . (f)^{\gamma} . Y^{left} + p . \frac{dump}{R} \quad 12$$

For simple offloading, T and E are modified as:

$$T = T^{trans} + T^{exe} + \delta . T^{registry} + T^{queue} \quad 13$$

$$E = \varepsilon . (f)^{\gamma} . Y + p . \frac{I}{R} \quad 14$$

3.6. Resource Utilization Model

The resource utilization U is calculated as a weighted sum of CPU (C), RAM (P), and bandwidth (B) usage:

$$U = \alpha_1 . C_{Usage} + \alpha_2 . P_{Usage} + \alpha_3 . B_{Usage} \quad 15$$

where $\alpha_1 + \alpha_2 + \alpha_3 = 1$ are the weighting factors and C is usage as a percentage of available cycles, P is usage as a percentage of used GB capacity, and B as a percentage of Mbps capacity.

3.7. Problem Formulation

The objective function for the system model is to minimize a weighted combination of task execution time T, energy consumption E, and resource utilization U:

$$obj \rightarrow min\{\omega_1.T + \omega_2.E + \omega_3.U\} \quad 16$$

with ω_1, ω_2, and ω_3 reflecting their relative importance.

4. MDEU-A2C

The A2C algorithm is a powerful DRL approach that combines two primary components: an actor, which selects actions based on a policy, and a critic, which evaluates these actions by estimating the value of each state-action pair. A2C works synchronously, where multiple agents interact with the environment and update the network synchronously, stabilizing the training process.

The actor learns a policy $\pi_\theta(a|s)$, which maps the current state s to an action a based on parameters θ.

The critic learns a value function $V(s;\omega)$, which estimates the expected reward of state s under the current policy, with parameters ω.

The key feature in A2C is the use of an advantage function, which quantifies how much better an action a is compared to the average action taken in state s. The advantage function $A(s,a)$ is defined as:

$$A(s,a) = Q(s,a) - V(s) \quad 17$$

where $Q(s,a)$ is the action-value function. In A2C, we approximate the advantage function using the temporal difference (TD) error:

$$A(s,a) \approx r + \gamma V(s') - V(s) \quad 18$$

where r is the reward, γ is the discount factor, and s' is the next state.

The total loss $\mathcal{L}$ in A2C combines both policy loss (actor) and value loss (critic) as well as an entropy term to encourage exploration:

$$\mathcal{L} = \mathcal{L}_{policy} + \alpha\mathcal{L}_{value} + \beta\mathcal{L}_{entropy} \quad 19$$

The policy loss $\mathcal{L}_{policy}$ is computed by maximizing the expected advantage:

$$\mathcal{L}_{policy} = -\mathbb{E}_{\pi_\theta}[A(s,a)\, log\, \pi_\theta(a|s)] \quad 20$$

The value loss $\mathcal{L}_{value}$ minimizes the error between the estimated value $V(s)$ and the expected return $r + \gamma V(s')$:

$$\mathcal{L}_{value} = \frac{1}{2}\big(r + \gamma V(s') - V(s)\big)^2 \quad 21$$

The entropy loss $\mathcal{L}_{entropy} = -\sum_\alpha \pi(a|s) \log \pi(a|s)$ encourages exploration by penalizing highly confident action probabilities.

Our proposed scheduling method in an MEC environment utilizes a multi-agent A2C framework with distinct actor and critic roles distributed across edge devices and a central fog node. Actors reside on each edge device and make real-time, local decisions regarding task scheduling and offloading based on a limited set of observed states. Each actor is trained to optimize task execution locally while balancing network and resource constraints. A centralized critic is located at the fog node, where it has a global view of the system, seeing the combined state across all edges. The critic evaluates actions taken by each actor, providing feedback to improve decision-making across the system. This structure enables each actor to focus on optimizing its own resource allocation based on its immediate environment, while the critic guides these decisions toward an overall system optimization.

The actor's state S^{actor} represents the local observations at each edge device:

$$S_{e_j}^{actor} = \left\{I_{r_i}, D_{r_i}, T_{r_i,e_j}^{exec}, d_{r_i,e_j}^{predict}, U_{e_j}, E_{e_j}, k_{e_j}\right\} \quad 22$$

where $T_{r_i,e_j}^{exec} = \frac{Y_{r_i}}{C_{e_i}}$ is execution time of task r_i in edge e_j, $d_{r_i,e_j}^{predict}$ is predicted distance between the edge device and the client device, based on mobility forecasts and k_{e_j} is count of tasks currently being processed on the edge device. This information allows each actor to make real-time decisions based on immediate, local conditions, optimizing for both task completion and resource utilization.

The critic's state S^{critic} provides a global view of the system, encompassing aggregated data across all edges:

$$S^{critic} = \left\{\bar{I}, \bar{D}, \bar{Y}, \overline{d^{predict}}, \bar{U}, \bar{E}, \bar{C}\right\} \quad 23$$

where $\bar{D}$ is the average deadline across all tasks in the system, $\bar{I}$ is the average size of tasks, $\bar{Y}$ represents the average CPU cycle requirement for tasks execution, $\overline{d^{predict}}$ is the average predicted distance from edge devices to clients and lastly, $\bar{U}, \bar{E}$ and $\bar{C}$ represents the average resource information of all edges. By using the global state S^{critic}, the critic can calculate a more holistic advantage value that considers the system-wide impact of each decision, helping to guide actors toward actions that benefit the entire network.

The reward function is carefully designed to encourage optimal scheduling by balancing time efficiency, resource utilization, energy efficiency, and task migration costs. The reward function R is defined as:

$$R = \alpha_0.Time + \alpha_1.Utilization + \alpha_2.Energy - \alpha_3.Migration \quad 24$$

where $Time = |D_{r_i} - T_{r_i,e_j}^{exec}|$, $Utilization = |\bar{U}_{next} - \bar{U}|$, $Energy = |\bar{E}_{next} - \bar{E}|$, $Migration$ is number of migrations until task finish execution, $\alpha_0, \alpha_1, \alpha_2$ and α_3 are weight coefficients representing the importance of each term in the reward. These components help balance the competing objectives of latency, efficiency, energy conservation, and stability (reduced migrations).

There are two possible actions (A) available for task execution:

$$\begin{cases} A = 0 & Local\ Execution \\ A = 1 & Offload/Migartion \end{cases} \quad 25$$

Figure 2 shows the architecture consists of separate networks for the actor (policy network) and the critic (value network). In the actor network, the Input Layer takes the actor's local state S^{actor}, Hidden Layers are three dense layers with 256 neurons each, using the mish activation function [9], which promotes gradient flow and learning efficiency, and the Output Layer is a softmax layer that outputs a probability distribution over possible actions. In the critic network, the Input Layer receives the global state S^{critic}, Hidden Layers are dense layers with 256, 128, and 64 neurons sequentially, using the mish activation, and the Output Layer is a single linear output providing the state-value $V(s)$, which estimates the expected return from the global state.

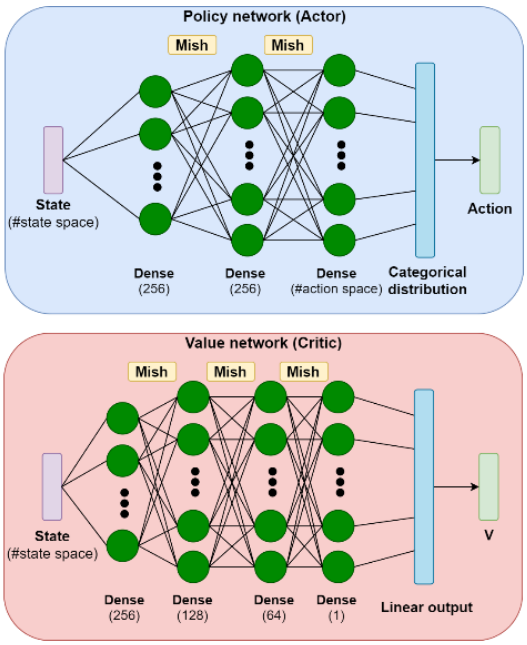

Figure 2. Policy and Value Network Architecture

Neural networks have gained increasing attention from academia and industrial groups for the accurate predictions offered by their various models. In this scenario, a Recurrent Neural Network (RNN) is a deep learning approach that extends the traditional feed-forward networks with internal cycles [10]. To handle client mobility in our environment, we employ an RNN-based model to predict the future location of a device. Specifically, we aim to forecast the device's location at the exact time when the task's deadline will be reached, simplifying the scheduling process and reducing resource overhead at the edge nodes. Instead of using a detailed time-series of predicted locations over the entire task duration, we focus only on the estimated final location at the deadline, minimizing computational demands. The RNN model, shown in Figure 3, is structured as follows:

- **Input Sequence**: This includes previous coordinates (X, Y) of a device, combined with the task's deadline. These inputs enable the model to account for both spatial position and temporal constraints.
- **Recurrent Layers**: The model consists of two RNN layers, each with 50 units, to learn temporal dependencies in the movement pattern of the device. These recurrent layers enable the RNN to capture patterns in the device's movement over time, making it well-suited for location prediction.
- **Dense Layers**: Following the RNN layers, we incorporate fully connected (dense) layers with ReLU activation. These layers refine the high-level features extracted by the RNN layers, ensuring a smooth transition from sequence information to coordinate prediction.
- **Output Layer**: The final output is a linear layer that provides the predicted future coordinates (X', Y'), which represent the device's expected location when the task reaches its deadline. This prediction allows the scheduling mechanism to consider where the client device will likely be when deciding where to execute the task, thus improving scheduling accuracy in a dynamic environment.

In a standard RNN layer, each unit maintains a hidden state that is updated at each time step based on the input sequence and the previous hidden state. Let's denote the input sequence at time t as $x_t = [X, Y, D]$, where D represents the task deadline. The hidden state h_t is computed as:

$$h_t = \sigma(W_h h_{t-1} + W_x x_t + b_h) \quad 26$$

where W_h and W_x are the weight matrices for the hidden state and the input, respectively, b_h is the bias term and σ is an activation function, typically tanh or ReLU.

The RNN then produces an output at each step, and the final output is passed through dense layers to compute the predicted location (X', Y'). The dense layers refine the representation and ultimately generate the coordinates by:

$$(X', Y') = W_{out} h_T + b_{out} \quad 27$$

where W_{out} and b_{out} are the weights and biases for the output layer.

By using RNNs in this way, we leverage their strengths in capturing temporal dependencies, making them a powerful choice for predicting future positions in dynamic, mobile environments. This allows the scheduling system to anticipate each device's location, providing more accurate and efficient task scheduling across edge and fog nodes.

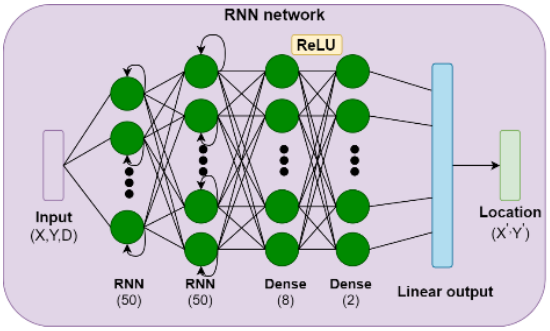

Figure 3. RNN Network Architecture

This comprehensive approach integrates multi-agent reinforcement learning, system-wide critic guidance, mobility prediction, and resource-aware reward design to enable efficient task scheduling in a dynamic, resource-constrained MEC environment. Each component works synergistically to achieve a balance between real-time task execution, resource utilization, energy efficiency, and reduced migration costs.

As it has been said, two possible actions are available for task execution: local execution on the current edge device or offload/migration to another edge device. If the decision is made to offload or migrate a task, the fog node selects the optimal destination edge for task processing. This decision is made by evaluating multiple factors within a 3D Pareto Set, consisting of the following dimensions:

- **Edge-to-Client distance**: The physical distance between the edge node and the client device (since fog knows the whole system state, it knows where the client is). Minimizing this distance can reduce communication latency, which is crucial for real-time applications.
- **Edge-to-PredictedLocation distance**: As client mobility is anticipated, the RNN-based prediction model forecasts the client's future location. The fog node considers the distance from the edge to this predicted location to ensure that the offloaded task remains within a reasonable range of the client's future position, thereby minimizing latency even as the client moves.
- **Estimated processing time**: The computation time estimated for the task at each potential destination edge device. This estimate considers the current load and computational capacity of each edge, aiming to choose a destination that can meet the task's deadline requirements.

The Pareto Set approach is widely respected in decision-making frameworks for its ability to handle conflicting criteria and allow for trade-offs. Instead of a single optimal solution, the Pareto front provides a set of "efficient" solutions, each representing a different balance between criteria. For example, a task scheduling decision could balance proximity to the client, energy consumption, and expected processing delay, depending on real-time system priorities. This technique is particularly beneficial in environments with heterogeneous and dynamic resource availability [11] [12].

The metrics chosen for the Pareto Set are critical because they collectively balance communication latency, adaptability to client mobility, and computational efficiency. By considering these metrics, the system optimizes task scheduling decisions that accommodate both current resource constraints and anticipated changes in client location, ensuring low-latency, energy-efficient task processing in dynamic edge environments. The fog node evaluates each candidate edge in terms of these three criteria and identifies the optimal offload destination using Pareto optimization. A Pareto set identifies the set of edge nodes that are not outperformed across all three criteria, ensuring a balanced compromise between latency, mobility support, and processing efficiency. This approach aligns with the methodologies highlighted in the paper [2], which discusses the use of Pareto optimization for scheduling in mobile and deadline-sensitive edge environments, ensuring that the offloaded tasks are processed at the most suitable edge device, enhancing both performance and adaptability to client mobility. By leveraging Pareto-based selection, the fog node can effectively balance the trade-offs between proximity to clients, latency, and processing time, thereby supporting high-quality service in vehicular edge computing applications. Figure 4 shows the workflow of the proposed method and, Figure 5 shows the workflow of the offloading process.

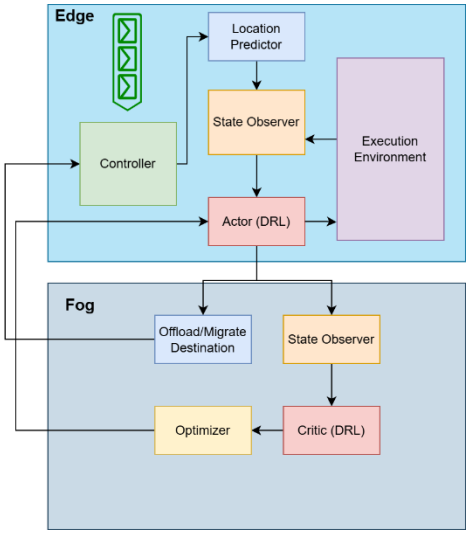

Figure 4. MDEU-A2C Workflow

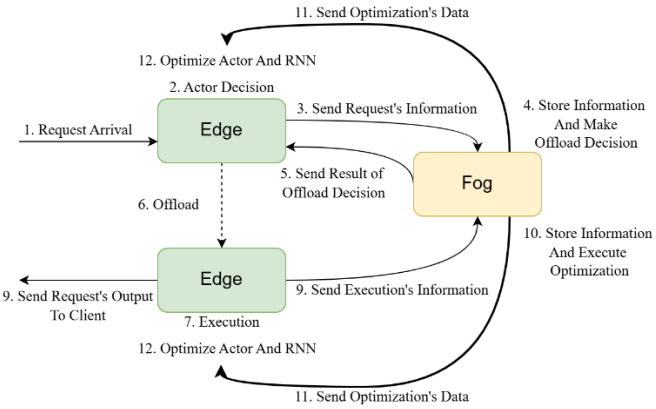

Figure 5. Offload Workflow of MDEU-A2C

Algorithm 1 demonstrates the main process inside an edge node and Algorithm 2 shows the main process of the fog node.

5. Results & Discussion

In this section, we first describe implementation details and experimental setups, then present evaluation results and discussion.

5.1. Implementation and Experimental Setting

In the evaluation environment, a Docker [13] container is created for each server (Edge, Fog, or Cloud), with all designated modules inside. Since Docker allows resource limitations, these containers can operate with resources similar to real-world devices. As a result, all components mimic real-world conditions, except for task execution. In real scenarios, tasks run inside a Docker container on an edge node. Here, tasks are executed as separate processes, and edge containers manage these processes just as real edge devices manage containers.

Algorithm1: Scheduling process in Edge

```
Initialize paramters and policies
While true:
    requests=Collect tasks from clients;
    requests=requests+(Running Tasks);
    locations=RNN(requests)
    edge_info=Collect Edge nodes information
    decisions=Actor(requests,locations,edge_info)
    for d in decisions
        if d==local
            Execute(d.task) / Continue(d.task)
        else
            destination=Get_destination(d.task)
            Offload(d, destination) /
            Migration(d,destination)
    Send(decisions,logs)
    optimize_info=Receive optimization data
    from fog
    Actor_Optimize(optimize_info.a2c)
    RNN_Optimize(optimize_info.rnn)
```

Algorithm2: Main process in Fog

```
Initialize paramters and policies
for timestep t = 1,2,...
    logs=Collect decisions and logs
    requests=Collect requests for
    offload/migration
    for r in requests
        P=ParetoSet(r)
        Send(requests.edge,P[0])
    for l in logs
        if task in l is done
            optimize_info=Optimize(l,all_edges)
            Send(task.edge,optimize_info)
```

For evaluation, one container is designated as the client, and a separate process is created for each considered client. These processes are responsible for task generation, sending requests, and executing tasks. Parameters such as geographical location, transmission power, speed, and client movement direction are set as fixed and simulated.

This environment enables more realistic and accurate evaluations of scheduling methods and provides a foundation for future research in this area. For evaluation purposes, each container must include a module for collecting logs and data to analyze and compare different approaches.

Comparative methods were evaluated in different environments; implementing them in this environment may lead to slight changes in the proposed algorithms (but no major changes are expected).

All communication between fog and edge is conducted using the MQTT protocol, one of the most commonly used protocols in such scenarios. The Mosquitto [14] broker an open-source and high-performance tool placed in the Fog node, is used for implementation. Each edge has a dedicated channel that both the fog and the respective edge subscribe to and publish relevant messages. The sequence diagram of communication is shown in Figure 6.

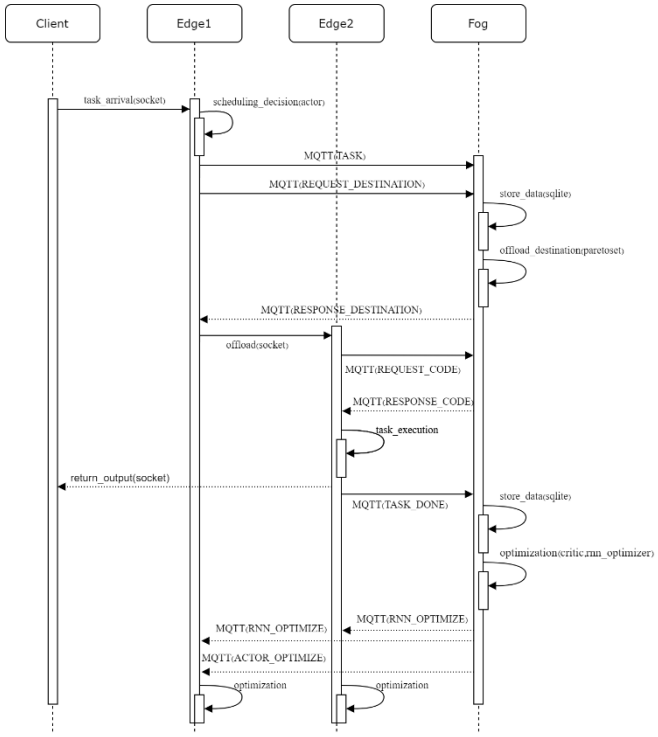

Figure 6. Communication's Sequence Diagram

All services are implemented using Python 3.11 [15]. As mentioned, Docker containers are used to run FEC services. For simulation, edges are randomly positioned in a 2D space with maximum coordinates of (-200, -200) to (200, 200). Each edge also receives a random coverage radius. A client must connect to an edge if it falls within its coverage (i.e., a ground station). To simulate client movement, 50 random paths are generated, each assigned to a different client. Clients move along these paths at random speeds and connect to edges when within their coverage. Clients are aware of the edge positions and their coverage radii, and use this information to determine the appropriate edge to connect to.

Client-Edge communication takes place via TCP/IP sockets. As soon as a client enters an edge's coverage area, it sends a keep-alive message containing its location and ID. This message is periodically sent to maintain the connection. On the edge side, a separate thread is allocated for each connected client to handle communication and messages. Clients periodically select a task randomly from a list of predefined tasks and send it to the edge. Upon receiving a task message, the edge moves the request information to a PENDING_TASKS list. The Scheduler service, implemented as a separate thread on the edge, periodically checks this list. After predicting the client's future location using an RNN model, the Actor module in the service makes the final decision.

If the decision is to execute locally, the task is moved to the EXECUTION_TASK list. If offloading or migration is chosen, the edge requests the target destination from the Fog, and once it responds, the task is processed accordingly. A separate thread is responsible for managing the execution of tasks, checking periodically and executing them as needed. If the client is still connected to the edge, the response is sent back upon completion. When the Fog receives a TASK_DONE message indicating task completion, it runs the optimization process. It uses the device's initial and final locations during the task to optimize the RNN model and sends updated data to all edges for further optimization. Additionally, the Critic model in the Fog performs optimization and sends updated information to the corresponding edge for updating its local Actor model. The architecture of the Actor, Critic, and RNN networks is provided in the Table 2. During offloading, the source edge sends the task to the target edge over a TCP/IP socket. The target edge receives and adds it to its EXECUTION_TASK list. For migration, CRIU is used. The currently running process is paused (using SIGSTOP), and a memory dump of the process is created using CRIU. This dump, along with task metadata, is transferred via a TCP/IP socket to the destination edge, which restores the task and resumes execution. Tasks can be moved between multiple edges based on scheduler decisions until execution is complete.

Table 2. Model's Parameters

Model	Layers	Parameters
Actor	Dense(64,ReLU)- Dense(64,ReLU)- Dense(2,Softmax)	Learning-Rate= 0.000001 Gamma=0.9 Epsilon=0.0001
Critic	Dense(64,ReLU)- Dense(64,ReLU)- Dense(1,Softmax)	
RNN	RNN(50)-Dropout(0.3)- RNN(50)-Dense(8,ReLU)- Dropout(0.3)-Dense(2,Linear)	Learning- Rate=0.001

As stated, Docker enables resource limitations for containers. Hence, the defined edge nodes have constrained resources to better reflect real-world conditions. Resource specifications for edge nodes are provided in Table 3. Each edge is assigned a random combination of these resources.

Table 3. Edge's Resources

Container	Resources		
	CPU	RAM	Bandwidth
Edge	1 Core	1 Gb	4 Mbps
	2 Core	2 Gb	6 Mbps
	3 Core	3 Gb	8 Mbps
Fog	3 Core	8 Gb	16 Mbps

Table 4 provides the list of processing services; we have implemented eleven functions, as detailed in the reference [16]. Their source codes reside in the Fog. If an edge receives a task without the necessary service code, it requests it from the Fog.

Table 4. Execution Services

Function Name	CPU	RAM	Disk	Network	Definition
Cipher	high	high	-	-	Creates random numbers of cipher messages with random input settings
Sin	high	high	-	-	Calculate sine of random number
Cos	high	high	-	-	Calculate cosine of random number
JSON	low	low	-	high	JSON dumps a large JSON file from the network
LINPACK	high	high	-	-	LINPACK benchmarks
Matrix	high	high	-	-	Multiply 2 matrixes with random dimensions
DD	low	low	high	-	Convert and copy a file
Feature Extraction	high	high	-	-	Extracting features from a dataset
Image	high	high	high	-	Apply multiple change to an image like rotation
ML Prediction	high	low	-	-	Prediction using a pre-trained ML model
ML Training	high	high	-	-	Training a ML model

Edge states and conditions are periodically saved in the Fog. These logs can be used to analyze resource usage, energy consumption, network traffic, and load balancing. Additionally, task logs can be analyzed to evaluate response delay, error rates, and success rates. All this data is stored in an SQLite database in the Fog for further analysis. These metrics are among the most important and widely used for evaluating such methods.

This study uses Node-Red [17] as the cloud simulation platform. It connects to the Fog via MQTT, receiving all data for real-time display and later analysis. The received data is visualized in three sections: resources, tasks, and maps. Once the simulation ends and MQTT is disconnected from the Fog, final evaluations are performed using the collected data. Figure 7 shows the simulation's environment architecture.

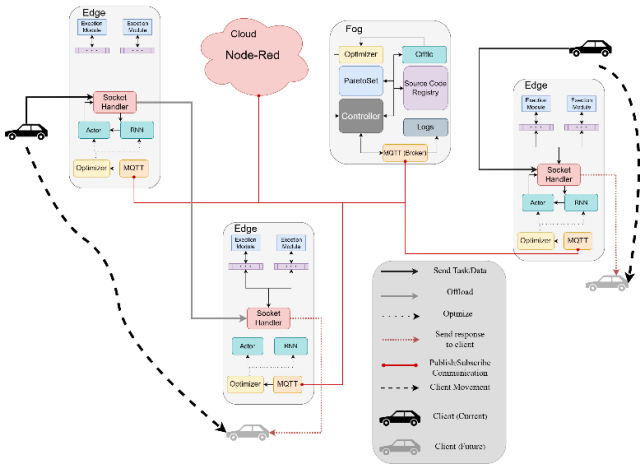

Figure 7. Simulation's Environment Architecture

5.2. Evaluation and Discussion

In the evaluation section, the methods were examined using a variable number of edges and clients. Specifically, the number of edges ranged from 2 to 10, and the number of clients ranged from 5 to 20. Details of the scenarios are presented in Table 5. All tests were conducted for a duration of 10 minutes. The positions of the edges and their coverage areas were determined randomly. Since the focus of this study is not on maximizing edge coverage, we set the overlap ratio of servers to 0.8 in order to minimize the overlap of edge coverage and achieve maximum effective coverage. The movement paths of clients were also selected randomly from a set of 100 predefined paths. This randomness brings the evaluation closer to real-world conditions, as in practice, client behavior and edge coverage can vary due to various factors. Therefore, introducing randomness to these variables lends more credibility to the evaluation. To enhance analysis, each test was repeated twice, and the data used in the analysis is the average of these two runs.

One advantage of this evaluation over prior work is the variation in the number of edges. Previous studies used a fixed number of edges, whereas this variation provides a more accurate evaluation and considers the scalability factor.

Table 5. Evaluation's Details

Parameter	Value
Edge	2-10
Client	5-10-15-20
SeV (MEPPO)	0.2*Number of Clients
VEC (MARINA)	0.2*Number of Edges

Figure 8 shows the distance between the predicted positions of clients using the proposed RNN model and their actual positions at the end of a task or its deadline.

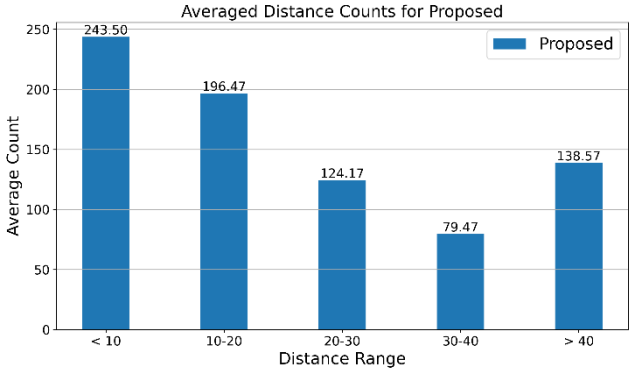

Figure 8. Difference Between Predicted and Actual Location of Clients

Most of the predicted positions fall within a distance of less than 20 units. Our goal is not to predict the exact position of the client but rather to correctly predict which edge will cover the client at the deadline. Based on the RNN's performance and the prediction results, this objective has been largely met. Although some predictions are off by more than 40 units, which may lead to selecting the wrong edge, the model achieves around 82% reasonably accurate predictions. This accuracy justifies the computational overhead introduced by the RNN, especially since its output can guide decision-making in the proposed method.

One key reason for the RNN's strong performance is continuous improvement: over time, the fog node, which communicates with all edges and is aware of both client locations and RNN decisions, continuously optimizes the model and sends updates to the edges. This gradual improvement allows the model to make better decisions as time progresses. If the simulation ran for more than 10 minutes, the RNN's performance would likely improve further. As shown in Table 2, the RNN model includes two dropout layers that help prevent overfitting and dependence on training-time data. These layers, combined with continuous optimization, make the model adaptive to different environments and help improve predictions over time.

Figures 9–15 present the evaluation results for various methods as the number of clients increases.

One of the most important parameters in scheduling algorithm evaluation is delay (latency), which reflects how well the algorithm enhances user experience. As the number of clients increases, the number of tasks and arrival rates also increase, leading to higher computational loads and longer waiting times, which increase total delay. Despite this, as can be seen in Figure 9, the proposed method outperformed the others. MARINA showed the highest delay, likely due to ignoring mobility. In MARINA, mobile nodes are servers and clients are stationary, making it unsuitable for environments with mobile clients. MEPPO, although mobility-aware, did not perform well—possibly because it does not consider the randomness of many parameters in the simulated environment, reducing its effectiveness.

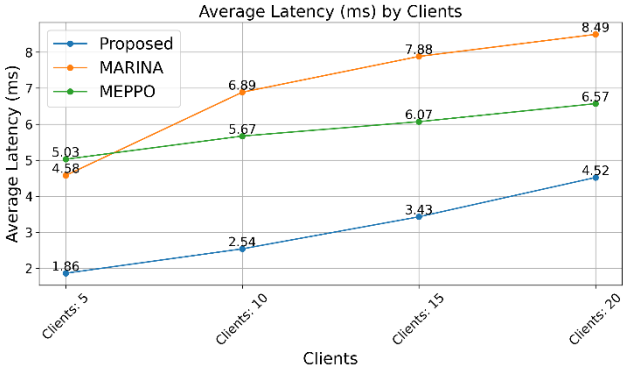

Figure 9. Average Latency as Number of Clients Increases

Figure 10 shows the average reward received by agents in the proposed method and MEPPO (MARINA does not use reinforcement learning). Both used the same reward function, but the proposed method performed better, indicating superior decision-making without disrupting service or overloading the system.

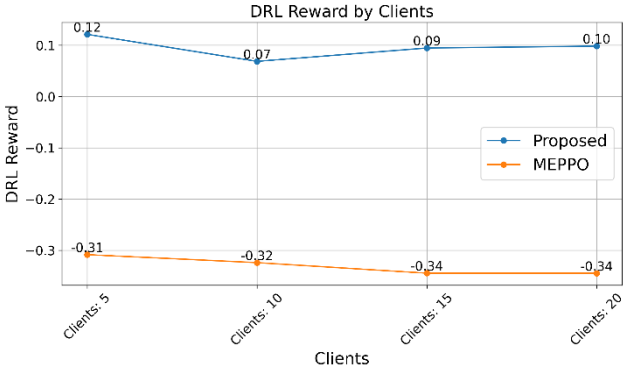

Figure 10. Average Reward as Number of Clients Increases

As client numbers increase, more tasks enter the system. Processed tasks include those handled by the system (even if dropped later due to deadline misses), but not those dropped due to full queues. A higher number of processed tasks indicates better scheduling decisions and the system's ability to handle load. The proposed method had the highest number of processed tasks as presented in Figure 11.

Figure 12 shows the total number of tasks sent into the system. All methods processed nearly all incoming tasks. However, the number of successful versus dropped tasks is more important.

Figure 13 indicates the percentage of failed tasks. Tasks can fail for two reasons: 1) missed deadlines, and 2) client not being within the edge coverage at completion. Both reasons depend on the scheduling algorithm. The failure rate increases with more clients due to increased load and reduced processing speed. The proposed method had the lowest failure rate, averaging below 10%. MARINA had the highest failure rate, and MEPPO also performed poorly—partly due to its Edge-Cloud architecture, which incurs long delays when offloading to the cloud.

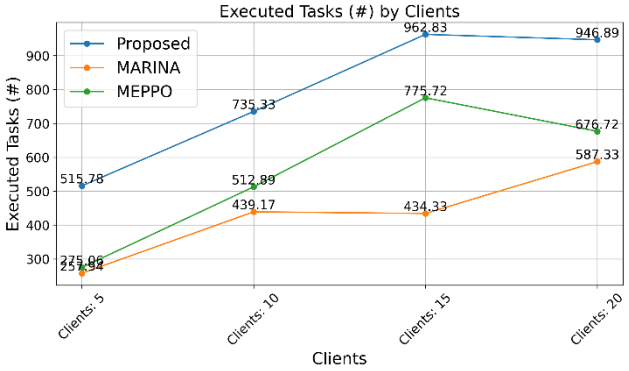

Figure 11. Average Executed Tasks as Number of Clients Increases

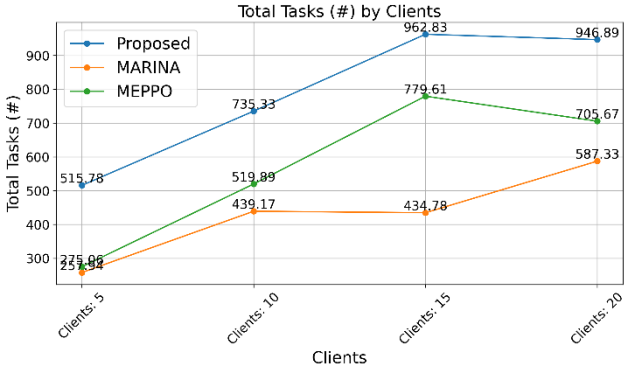

Figure 12. Total Number of Tasks as Number of Clients Increases

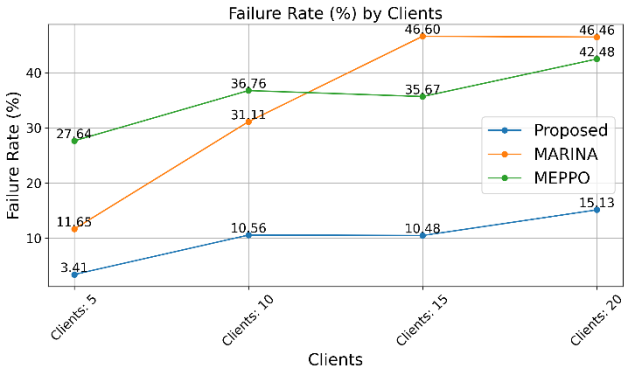

Figure 13. Average Failure Rate as Number of Clients Increases

Figure 14 shows the network traffic generated. MEPPO generated the highest traffic, due to offloading to the cloud. The proposed method, using FEC architecture and supporting migration, also produced significant traffic. MARINA, which limits communication between edges, had the lowest traffic.

Only the proposed method supports both migration and offloading, contributing to its better performance. Including migration allows the correction of initial poor scheduling decisions by moving tasks to better-suited edges, the results can be seen in Figure 15. However, this also adds computational and network overhead, so it may not be ideal in latency- or bandwidth-sensitive environments.

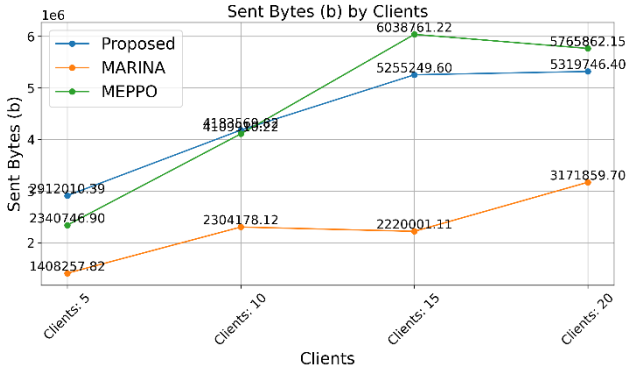

Figure 14. Average Sent Bytes as Number of Clients Increases

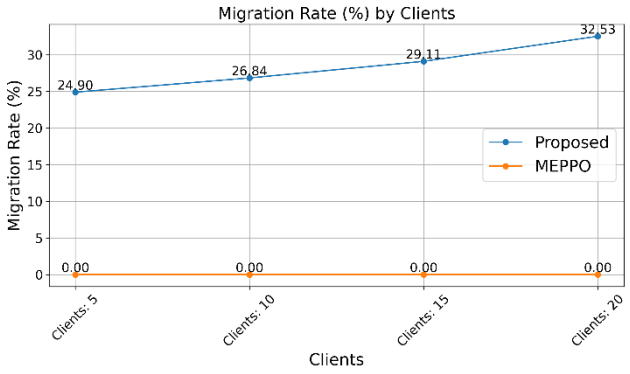

Figure 15. Average Migration Rate as Number of Clients Increases

Figure 16 illustrates the average task delay as the number of edge nodes increases. The proposed method demonstrates the best performance, while MARINA shows the worst in this metric. A noteworthy observation is the increase in delay with more edges, which contradicts the expectation of reduced delay due to higher computational resources. Interestingly, only MEPPO avoids this increasing trend, maintaining relatively stable results. However, both the proposed method and MARINA show noticeable increases in delay. This behavior stems from two key reasons: (1) randomness in the simulation parameters and (2) the design and functioning of the algorithm.

Although increasing the number of edges boosts both available resources and coverage, it also increases the number of environmental parameters the algorithm must handle. For the proposed method, which relies on DRL, this results in a longer adaptation period to the new environment and a higher probability of suboptimal decisions, as well as an increase in the scheduling algorithm's execution time. While the delay increase is modest in the proposed method, it is quite pronounced in MARINA.

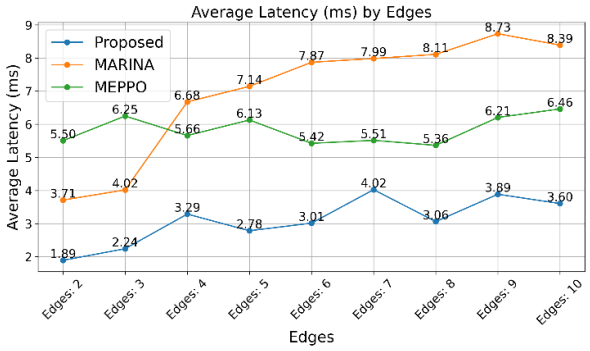

Figure 16. Average Latency as Number of Edges Increases

Figure 17 presents the average reward received by the Agents using the proposed method and MEPPO. The results show that the proposed method consistently makes better decisions that neither disrupt client service nor cause excessive system overhead. Even though the delay increased with more edges in the proposed method, the number of correct decisions remained higher than those in MEPPO.

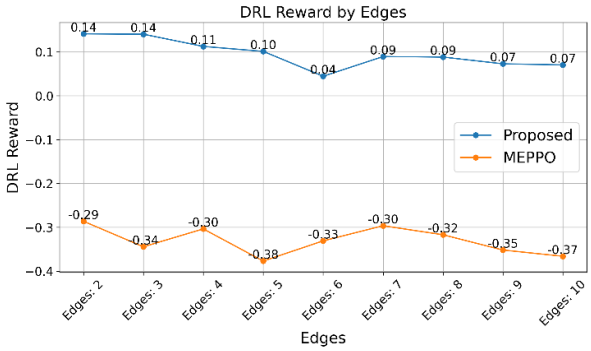

Figure 17. Average Reward as Number of Edges Increases

Figures 18 and 19 show the average number of executed and total generated tasks, respectively. As expected, increasing the number of edge nodes leads to wider coverage and more task processing opportunities. The proposed method outperforms all others by executing the highest number of tasks. Additionally, as mentioned earlier, all methods successfully process nearly all tasks generated by clients.

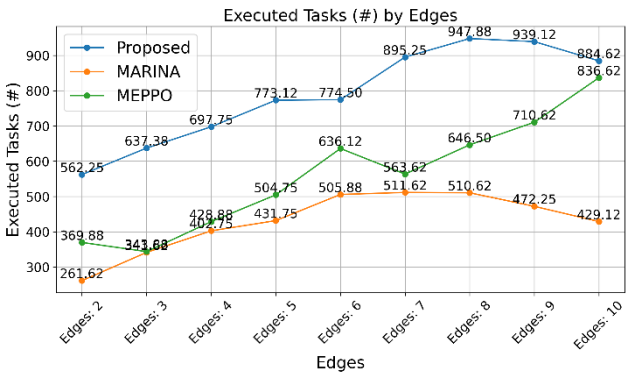

Figure 18. Average Executed Tasks as Number of Edges Increases

Figure 20 shows the average percentage of failed tasks. As previously explained, more edges lead to more generated tasks, which in turn increases computational overhead and the likelihood of failure. The proposed method performs best in this metric, while MARINA performs the worst. A significant insight from this figure (and others) is that adding edges has the most negative impact on MARINA. Increased resources lead to higher failure rates and delays, highlighting its unsuitability for large-scale or resource-rich environments.

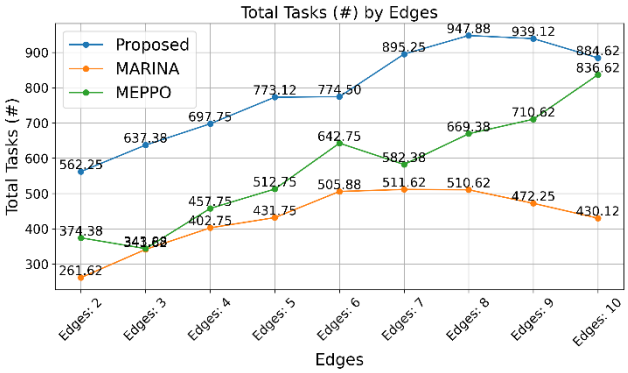

Figure 19. Total Number of Tasks as Number of Edges Increases

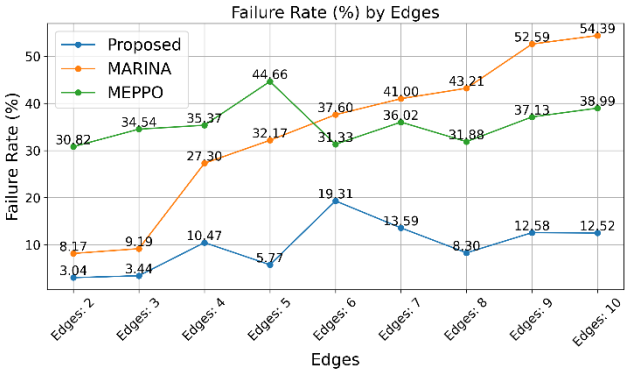

Figure 20. Average Failure as Number of Edges Increases

Figure 21 shows the average number of bytes received and sent across the network. The results align with Figure 14. MARINA generates the lowest network traffic, while both the proposed method and MEPPO generate significantly higher traffic.

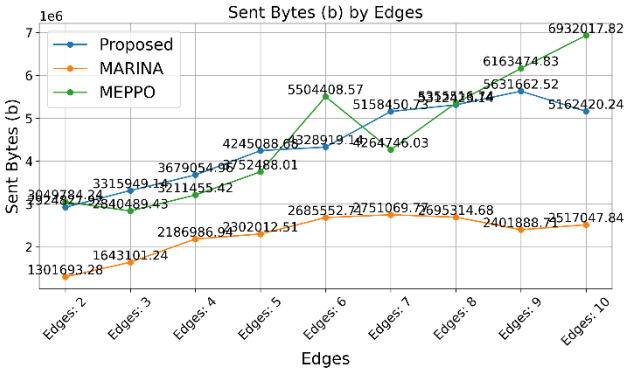

Figure 21. Average Sent Bytes as Number of Edges Increases

Figure 22 displays the average number of task migrations for the proposed method. Among the evaluated methods, only the proposed method supports migration in addition to offloading. This capability contributes directly to its superior performance.

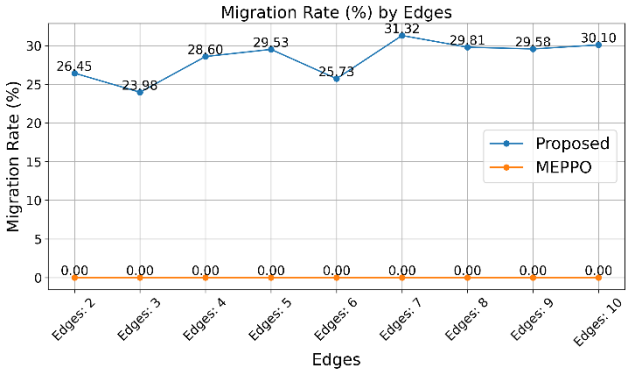

Figure 22. Average Migration Rate as Number of Edges Increases

Overall, the proposed method demonstrates superior performance compared to MEPPO and MARINA, followed by MEPPO and MARINA. One key insight from this evaluation is the role of the environment in algorithm performance. MEPPO and MARINA were originally developed for environments that differ greatly from the environment considered in this research. Although both methods performed well in their original settings, they showed suboptimal results in our more realistic and dynamic environment. This work aims to simulate a highly randomized and real-world-like scenario, enhancing the credibility of the findings.

MARINA, by neglecting client mobility and location changes, shows weak results in dynamic environments, making it unsuitable for such scenarios. Moreover, as resource availability increases (i.e., more edges), MARINA experiences significant performance degradation, making it ineffective for large-scale deployments.

Although MEPPO is mobility-aware, it relies on an Edge-Cloud architecture. This leads to high delays and failure rates due to the inherent latency of offloading to the Cloud. Additionally, MEPPO includes SeVs (Service Vehicles) that act as mobile processors, which adds another layer of complexity and overhead. Managing both mobile clients and SeVs introduces a significant burden, making the approach less practical in densely populated environments.

In contrast, the proposed method benefits from its simplicity and continuous optimization. Edge-based Agents, in conjunction with RNN-based location prediction models, are consistently optimized by the Fog node, enabling adaptive decision-making and improved performance over time.

However, a noteworthy trade-off is the higher network traffic. Due to its FEC architecture and continuous interactions between edge nodes and the Fog, the proposed method results in substantial data transmission. This limits its applicability in environments with strict bandwidth or network resource constraints.

Table 6 summarizes the proposed method's overall performance improvements over MARINA and MEPPO in each evaluation metric.

Table 6. Evaluation's Summary

Parameter	Compare to MEPPO	Compare to MARINA
Failure Rate	20.27%	33.94%
Total Tasks	25.98%	110.79%
Executed Tasks	30.84%	86.84%
Sent Bytes	6.46%	-90.10%
Latency	60.11%	60.11%
Reward	128.13%	-

While the proposed method is designed to support large-scale deployments, our experimental evaluation was conducted using a local PC environment, which imposed practical constraints on resource availability. As a result, the simulations were limited to a maximum of 10 edge nodes and 20 clients. This setup, however, aligns with the evaluation scope adopted in previous works such as MEPPO and MARINA, which were also tested in similarly constrained environments. Despite the limited scale, our results clearly demonstrate the effectiveness and adaptability of the proposed method in dynamic, heterogeneous edge environments. In future work, we aim to validate the system's performance in larger-scale deployments using high-performance computing infrastructure or cloud-based simulation platforms.

6. Conclusion

Mobile Edge Computing has emerged to meet the demands of modern data-driven applications. By bringing data processing and computation closer to end-users, MEC reduces latency, improves real-time responsiveness, and alleviates network congestion—making it essential for delay-sensitive applications. Unlike centralized cloud architectures, MEC relies on distributed edge nodes, each with limited computational resources. While this decentralized approach enables faster data processing, it also introduces challenges in workload management, particularly under conditions of resource heterogeneity, limited capacity, and user mobility.

This research proposes a scheduling solution that considers user mobility, resource heterogeneity, energy consumption, and resource utilization. The proposed approach is based on the Advantage Actor-Critic (A2C) method, a deep reinforcement learning technique, and is used for decision-making and task scheduling. In this distributed method, actors deployed on each edge node decide whether to execute a user request locally or offload it to another edge. If offloading is chosen, the fog node, having a global view of the system, selects the optimal destination. Additionally, each edge node is equipped with an RNN model alongside its actor to predict user location, which is factored into the scheduling decisions.

One of the strengths of the proposed method is the placement of the critic in the fog node, which enables continuous optimization of both the edge actors and their RNN models, enhancing the system's adaptability.

Evaluations comparing the proposed method with baseline approaches show that it achieves excellent performance in terms of response delay, failure rate, user location prediction accuracy, and the number of successfully served requests. However, due to the FEC architecture, the method results in higher network traffic compared to others, which may make it less suitable for bandwidth-constrained environments.

For future work, we suggest continuing to useother DRL method in MECs and using the whole system's state for optimization (something like shared experienced approaches), as we presented in this research, combined methods (like ours) have shown promising results, combining RL and DRL methods together or with other heuristic methods could lead to better results. Despite scheduling methods, importing more and more real-world parameters in these types of fields makes future research more applicable, parameters like network details, communication noises, and client anomaly behaviors are good choices to start.

7. References

[1] P. Li, Z. Xiao, X. Wang, K. Huang, Y. Huang and H. Gao, "EPtask: Deep Reinforcement Learning Based Energy-Efficient and Priority-Aware Task Scheduling for Dynamic Vehicular Edge Computing," *IEEE Transactions on Intelligent Vehicles,* vol. 9, no. 1, pp. 1830-1846, 2023.

[2] J. B. D. da Costa, A. M. de Souza, R. I. Meneguette, E. Cerqueira, D. Rosário, C. Sommer and L. Villas, "Mobility and Deadline-Aware Task Scheduling Mechanism for Vehicular Edge Computing," *IEEE Transactions on Intelligent Transportation Systems,* vol. 24, no. 10, pp. 11345-11359, 2023.

[3] Y. Fan, J. Ge, S. Zhang, J. Wu and B. Luo, "Decentralized Scheduling for Concurrent Tasks in Mobile Edge Computing via Deep Reinforcement Learning," *IEEE Transactions on Mobile Computing,* pp. 1-15, 2023.

[4] X. He, C. You and T. Q. S. Quek, "Age-Based Scheduling for Mobile Edge Computing: A Deep Reinforcement Learning Approach," *IEEE Transactions on Mobile Computing (Early Access),* pp. 1-16, 2024.

[5] J. Lu, J. Yang, S. Li, Y. Li, W. Jiang and J. Dai, "A2C-DRL: Dynamic Scheduling for Stochastic Edge–Cloud Environments Using A2C and Deep Reinforcement Learning," *IEEE Internet of Things Journal,* vol. 11, no. 9, pp. 16915-16927, 2024.

[6] L. Niu, X. Chen, N. Zhang, Y. Zhu, R. Yin and C. Wu, "Multiagent Meta-Reinforcement Learning for Optimized Task Scheduling in Heterogeneous Edge Computing Systems," *IEEE Internet of Things Journal,* vol. 10, no. 12, pp. 10519-10531, 2023.

[7] L. Liu, J. Feng, X. Mu, Q. Pei, D. Lan and M. Xiao, "Asynchronous Deep Reinforcement Learning for Collaborative Task Computing and On-Demand Resource Allocation in Vehicular Edge Computing," *IEEE Transactions on Intelligent Transportation Systems,* vol. 24, no. 12, pp. 15513-15526, 2023.

[8] Z. Cao, X. Deng, S. Yue, P. Jiang, J. Ren and J. Gui, "Dependent Task Offloading in Edge Computing Using GNN and Deep Reinforcement Learning," *IEEE Internet of Things Journal (Early Access),* 2024.

[9] D. Misra, "Mish: A Self Regularized Non-Monotonic Activation Function," *Arxiv,* 2019.

[10] Y. LeCun, Y. Bengio and G. Hinton, "Deep learning," *Nature,* vol. 521, p. 436–444, 2015.

[11] N. A. Rashed, Y. H. Ali, T. A. Rashid and A. Salih, "Unraveling the Versatility and Impact of Multi-Objective Optimization: Algorithms, Applications, and Trends for Solving Complex Real-World Problems," *Arxiv,* 2024.

[12] H. Anysz, A. Nicał, Ž. Stević, M. Grzegorzewski and K. Sikora, "Pareto Optimal Decisions in Multi-Criteria Decision Making Explained with Construction Cost Cases," *Symmetry* , vol. 13, no. 1, 2021.

[13] "Docker," Docker, [Online]. Available: https://www.docker.com/.

[14] "Mosquitto," Eclipse, [Online]. Available: https://mosquitto.org/.

[15] "Python," Python, [Online]. Available: https://www.python.org/.

[16] J. Kim and K. Lee, "Function Bench : A Suite of Workloads for Serverless Cloud Function Service," in *IEEE International Conference on Cloud Computing,* Milan, Italy, 2019.

[17] "Node-Red," IBM, [Online]. Available: https://nodered.org/.

[18] A. Biswas and H.-C. Wang, "Autonomous Vehicles Enabled by the Integration of IoT, Edge Intelligence, 5G, and Blockchain," *Sensors,* vol. 23, no. 4, 2023.

[19] S. Liu, L. Liu, J. Tang, B. Yu, Y. Wang and W. Shi, "Edge Computing for Autonomous Driving: Opportunities and Challenges," *Proceedings of the IEEE,* vol. 107, no. 8, pp. 1697-1716, 2019.

[20] A. Hazra , P. Rana , M. Adhikari and T. Amgoth , "Fog computing for next-generation Internet of Things: Fundamental, state-of-the-art and research challenges," *Computer Science Review,* vol. 48, 2023.

[21] S. N. Srirama, "Distributed Edge Analytics in Edge-Fog-Cloud Continuum," *Arxiv,* 2023.

[22] W. Qin , H. Chen , L. Wang , Y. Xia , A. Nascita and A. Pescapè , "MCOTM: Mobility-aware computation offloading and task migration

for edge computing in industrial IoT," *Future Generation Computer Systems*, vol. 151, 2024.

[23] M. Ferens, D. Hortelano, I. de Miguel, R. J. Durán Barroso, J. C. Aguado and L. Ruiz, "Deep Reinforcement Learning Applied to Computation Offloading of Vehicular Applications: A Comparison," in *International Balkan Conference on Communications and Networking*, Sarajevo, Bosnia and Herzegovina, 2022.

[24] N. Yang, J. Wen, M. Zhang and M. Tang, "Multi-objective Deep Reinforcement Learning for Mobile Edge Computing," in *International Symposium on Modeling and Optimization in Mobile, Ad Hoc, and Wireless Networks*, Singapore, Singapore, 2023.

[25] B. Xie and H. Cui, "Deep reinforcement learning-based dynamical task offloading for mobile edge computing," *The Journal of Supercomputing*, vol. 81, 2024.