Optimising Service Migration in AC3: A Cache-Aided, Data-Mobility-Aware Framework

Jeffrey Redondo[a], Kostas Ramantas[a] and Christos Verikoukis[b]

[a]R&D Department, Carrer del Dr. Rizal, 10, Barcelona, 08006, Spain
[b]ISI/ATH and University of Patras, Campus Universitari de Patres, Patres, 265 04, Grècia

ARTICLE INFO

Keywords:
Edge Server
Reinforcement learning
cache memory
mobility
migration

ABSTRACT

The service migration mechanism is crucial for achieving low-latency performance in mobile edge computing (MEC). However, most existing approaches focus on optimizing migration, caching, or resource allocation separately. In stateless cloud-native environments, migration performance is heavily influenced by data locality. After relocation, services need to re-fetch their state, which adds extra latency. This work presents a cache-aided, mobility-aware service migration framework. It integrates cache occupancy and user–server proximity into the state space of a reinforcement learning (RL) agent. Rather than proposing a new RL algorithm. We formulates service migration as a joint latency–migration cost optimization problem. It explicitly incorporates cache occupancy and mobility-induced proximity into the decision process. We model heterogeneous mobility patterns (pedestrian, bicycle, vehicle, and IoT) and diverse MEC node resources in a unified 5G-compliant simulation. Results are evaluated against heuristic baselines such as Random and Greedy Knapsack, as well as standard deep RL methods including DQN, Double DQN, and Dueling DDQN. Our findings show that the cache-aided framework reduces total migration time by 54% and average service latency by 30%.

1. Introduction

Mobile Edge Computing (MEC) has emerged as a key architecture for enabling low-latency, high-quality services by moving computation and storage closer to end users Fan and Ansari (2019); Zhang, Zhou and Fortino (2018b); Pang, Chung, Chiu and Zhang (2017). This proximity reduces round-trip delay and improves responsiveness for applications such as augmented/virtual reality Yang, Liu, Zhang, Li, Chen and Fu (2022); Lee, Kim and Hwang (2019), real-time gaming Huang and Su (2023), and autonomous driving (e.g., HD map updates) Lee, Lee, Yoo and Moon (2020). However, sustaining consistent service quality in MEC environments is challenging. For instance, the user mobility continuously alters both the current location of users and the available resources at each edge node Zhang, Ni, Zhang, Zhang and Zhang (2023). These dynamics trigger frequent *service migrations* and dynamic resource reallocation across distributed edge servers Xu, He and Li (2024). Migration decisions must therefore balance latency reduction against the overhead introduced by relocating services.

To address mobility, existing works explore proactive migration strategies. These approaches predict a user's next attachment point and pre-move services to nearby nodes. This reduces handover delay and service disruption Zhang, Liu, Fu and Yahyapour (2018a); Singh, Sukapuram and Chakraborty (2023); He, Li, Lin, Dong, Qin and Li (2024); Liang, Liu, Lok and Huang (2021). While these methods are effective, they introduce significant prediction overhead due to the accurate user trajectories prediction. This includes increased computational cost and system-level adaptation complexity. Their performance also degrades under heterogeneous or highly dynamic mobility conditions Yuan, Li, Zhou, Lin, Luo and Shen (2020); Wang, Ouyang, Liao, Gong, Yu and Chen (2022).

In parallel, edge caching improves data availability by storing popular or recently used content close to users. When the users move, the services might be relocated, and cached data can reduce cold-start delays by avoiding retrieval of data from remote cloud servers. This is particularly important in cloud-native, stateless service architectures Sabella, Li, Lee, Cominardi, Huang, Pateromichelakis, Kashyap, Costa, Granelli, Featherstone et al. (2023), where the application instances do not retain persistent user state and must fetch needed data after each migration.

Despite their conceptual interdependence, service migration and edge caching have predominantly evolved as separate research directions Zhang et al. (2018a); Singh et al. (2023). Migration-centric frameworks typically focus on minimizing communication and computation overhead. It is often assumed that service state is retrieved from remote cloud infrastructure without explicitly incorporating cache occupancy or leveraging data locality information Peng, Tang, Zhou, Li, Qi, Liu and Lin (2024b); Liang et al. (2021). Conversely, caching-oriented studies optimize content placement or cache hit ratio using heuristic or learning-based approaches. But they do not integrate cache state into service migration decision-making Malektaji, Ebrahimzadeh, Elbiaze, Glitho and Kianpisheh (2021). This separation overlooks a critical interaction: cache availability that directly influences migration-induced latency and overall Quality of Service (QoS), particularly in stateless cloud-native service environments.

*Corresponding author
j.redondo@iquadrat.com (J. Redondo); kramantas@iquadrat.com (K. Ramantas); cveri@isi.gr (C. Verikoukis)
ORCID(s): 0000-0001-0000-0000 (J. Redondo)

While numerous studies apply deep RL (DRL) to MEC optimisation problems, most focus on algorithmic variations or isolated objectives (e.g., caching, offloading, or migration). In contrast, this work emphasizes a system-level contribution. Rather than proposing a new RL algorithm, we revisit the service migration problem by explicitly incorporating mobility dynamics, cache occupancy, and latency into the system formulation. This unified formulation enables systematic evaluation of learning-based migration strategies under heterogeneous mobility and resource conditions.

1.1. Limitations of Existing Approaches

Despite advances in both areas, **migration and caching have largely been optimized in isolation**:

- **Content-centric caching with RL:** Deep reinforcement learning (DRL) has been applied to content placement and migration in vehicular and edge networks (e.g., Malektaji *et al.* Malektaji et al. (2021)). While these methods reduce content delivery delay, they do not address service migration. In such works, the cache is treated as the optimization target rather than leveraged as a contextual signal for guiding migration decisions. However, a fundamental limitation in the **problem formulation** is that existing migration frameworks typically assume service data is always fetched from remote clouds, ignoring cache occupancy and data locality. This can lead to suboptimal migration decisions for stateless, cloud-native services.

- **Cost-aware service migration:** Studies, such as Peng et al. Peng et al. (2024b), incorporated communication and computing costs, considering migration overhead and CPU interference to improve service migration decisions. Nevertheless, these strategies usually assume that the data used by the services is always fetched from the cloud without considering cache state information.

- **Mobility modeling constraints:** Many proactive migration schemes rely on accurate trajectory prediction to anticipate user movement. While effective under stable mobility patterns, such approaches may incur high computational overhead and reduced robustness in heterogeneous environments. Other optimization-based methods (e.g., heuristic or cost-driven formulations) typically assume simplified mobility models or static conditions. In this context, RL offers an alternative data-driven approach that adapts decisions based on observed state transitions, without requiring explicit trajectory prediction.

1.2. Our Contribution

The contributions of this work are primarily methodological and system-oriented, rather than algorithmic. The contributions are:

- We formulate service migration as a joint optimization problem that explicitly incorporates latency, migration cost, mobility-induced distance variation, and cache occupancy.

- We integrate cache state as a contextual signal within the RL state space to guide migration decisions for stateless cloud-native services, shifting cache-awareness from content optimization to service placement.

- We design and implement a unified simulation framework based on a 5G-compliant environment that models heterogeneous mobility patterns (pedestrian, bicycle, vehicle, IoT) and resource-diverse MEC nodes.

We implement and evaluate multiple deep RL variants (DQN, DDQN, and Dueling DDQN) within the proposed framework. Simulation results in a 5G-compliant environment demonstrate that incorporating cache-awareness and mobility-aware state design significantly reduces total migration time and average service latency while preserving QoS under dynamic conditions.

1.3. Positioning

This work provides a new perspective on adaptive MEC orchestration. It leverages cache state to guide service migration decisions. Rather than treating caching as a separate optimisation goal or considering only CPU and backhaul costs. It is motivated by prior content-oriented caching Malektaji et al. (2021) and cost-aware migration Peng et al. (2024b) studies. These studies highlight the importance of data locality in improving service placement in highly mobile, heterogeneous edge networks. However, they do not exploit cache occupancy as a contextual signal for migration decisions. To address this gap, this work does not introduce a novel RL architecture. Instead, its novelty lies in (i) a cache- and mobility-aware problem formulation for service migration, (ii) the integration of data locality information into the RL state space, and (iii) a comprehensive evaluation under heterogeneous mobility and resource conditions. We demonstrate that these design choices alone can significantly influence the behavior and performance of standard deep RL agents.

2. Related Work

This section reviews prior work on service placement and migration in edge computing. We group existing approaches into (i) latency/workload-aware methods and (ii) network/preference-aware methods, highlighting limitations relevant to cache-aware migration. Finally, we position our work, which integrates multi-service support, dynamic user mobility, and cache-awareness into a unified migration framework.

2.1. Latency-aware and Workload-aware

Several studies address service migration by incorporating latency and workload information into their optimization frameworks Ray, Banerjee and Narendra (2024); Hui, Chen,

Zhou, He, Wu and Yang (2022); Tsourdinis, Makris, Fdida and Korakis (2023); Xiao, Ma, Xia, Zhou, Luo, Wang, Fu, Wei and Jiang (2022). For example, Tsourdinis et al. (2023) employs deep reinforcement learning (DRL) with workload and latency features in the state space. However, user mobility is not explicitly modeled, limiting the agent's ability to adapt to dynamic user–server associations. Similarly, Xiao et al. (2022) considers load imbalance caused by uneven user distribution and adjusts migration decisions accordingly. While this improves proximity-based placement, decisions remain primarily driven by workload distribution rather than long-term service performance or data locality considerations. Recent DRL-based caching research has focused on optimizing content placement rather than service migration. Malektaji et al. (2021) apply deep Q-learning to improve cache hit ratio and delivery latency in vehicular edge networks. Wu, Zhao, Fan, Fan, Wang and Zhang (2023) propose a cooperative caching strategy using asynchronous federated DRL to coordinate vehicles and roadside units. These approaches optimize caching efficiency but do not address the migration of running services. Other works rely on explicit trajectory prediction to enable proactive migration Yuan et al. (2020); Wang et al. (2022). These methods are effective under predictable mobility patterns but introduce additional computational and signaling overhead.

In contrast, our framework models mobility through dynamic user–server distance within the state representation and incorporates cache occupancy as a contextual signal for migration decisions. Rather than optimizing caching itself, we leverage cache state to guide service placement under dynamic workload and mobility conditions. This strategy enables the RL agent to learn mobility-aware policies without relying on heuristic or handcrafted mobility penalties, allowing for more adaptive and generalizable service placement decisions.

2.2. Network-aware and Preference-aware

Another line of work improves service migration using network state, typically bandwidth or connectivity conditions. For example, Diktyo et al. Santos, Wang, Wauters and De Turck (2023) and the authors of Addad, Dutra, Taleb and Flinck (2022) include bandwidth information in the state or action space. These methods can be affected by user mobility and the data types required by services. They typically assume that service data is retrieved from remote sources and do not explicitly account for cache locality effects. Peng *et al.* Peng et al. (2024b) address dynamic vehicular networks by proposing a transfer RL framework (fast-TRL) that models communication and computing costs. The authors include migration overhead and CPU interference to improve service placement. However, their approach assumes service data is always remote and does not exploit cache locality to reduce cold-start delays after migration. To cope with mobility, container-based migration methods such as Bellavista, Corradi, Foschini and Scotece (2019) introduce coarse- and fine-grained mobility prediction to trigger proactive handoff. Although effective in some cases, these methods may still suffer from ping-pong effects and hysteresis. Several other works attempt to optimise handoff in 5G using ML-based prediction Mbulwa, Tung Yew, Chekima and Dargham (2024); Elbatal, Maiwada, Danyaro and Sarlan (2025). Nonetheless, most assume trajectory prediction accuracy and do not incorporate cache or multi-service demands. Preference-aware caching strategies have also been studied. For example, the PACM scheme Lin, Ning, Zhang, Liu, Yu and Leung (2024) jointly optimizes content placement and migration for video streaming. Although effective for content delivery, such approaches do not address the migration of running services.

Traditional heuristic-based migration methods typically optimize single objectives such as latency or cost. For instance, (Peng, Liu, Zhang and Li, 2024a) propose a multi-resource trade-off heuristic for cost-aware migration in MEC systems. In contrast, our framework integrates network conditions, workload, mobility dynamics, and cache occupancy within a unified decision model to support adaptive service placement.

2.3. Positioning of Our Work

In contrast to prior studies, our framework focuses on service migration rather than content caching, incorporating cache occupancy and data locality as contextual features within the decision process. Unlike caching-oriented approaches that optimize content placement Malektaji et al. (2021); Wu et al. (2023) or cost-aware migration schemes that model only communication and computation overhead Peng et al. (2024b), our formulation explicitly integrates latency variation caused by user mobility, migration cost, resource availability, and cache state within a unified decision model. This enables migration of cloud-native stateless services under dynamic workload and mobility conditions.

Direct quantitative comparison with many recent learning-based frameworks is challenging due to substantial differences in system assumptions. These include variations in state and action representations, reward formulations, and runtime architectures. The primary objective of this work is to isolate and quantify the impact of cache-awareness and mobility-induced latency variation within a unified migration framework. Therefore, we conduct controlled comparisons under identical system assumptions rather than attempting cross-framework benchmarking, which would introduce architectural and modeling discrepancies.

Table 1 summarizes the main differences with our framework against the closest works. Unlike Peng et al. (2024b), which assumes service data is always fetched remotely and does not exploit cache locality, our formulation integrates cache occupancy directly into the RL state space. Tsourdinis et al. (2023) incorporates workload and latency features but omits explicit mobility modeling, limiting adaptability under heterogeneous movement patterns. Malektaji et al. (2021) focuses on cache hit optimization and treats the cache as the optimization target rather than a decision feature. Our framework is the only one that simultaneously integrates

Table 1
Comparison of features with existing frameworks.

Feature	Ours	Peng et al. (2024b)	Tsourdinis et al. (2023)	Malektaji et al. (2021)
Cache occupancy in state space	✓	×	×	×
Explicit mobility modeling	✓	Partial	×	Partial
Service migration focus	✓	✓	✓	×
5G-compliant simulation	✓	×	✓	×
Standard DRL evaluation	✓	✓	✓	✓
Cache as migration signal	✓	×	×	×

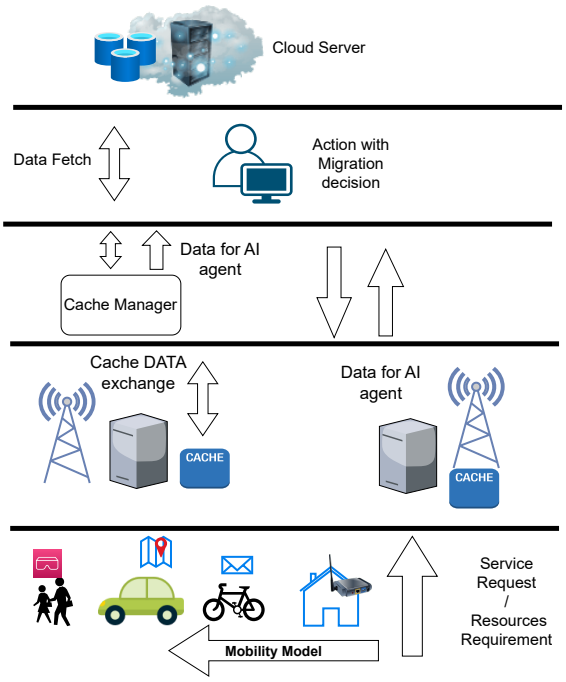

Figure 1: Migration Framework, it illustrates the system architecture showing MEC servers, cloud servers, and user devices, highlighting the service migration and cache management flow

cache-aware migration, heterogeneous mobility, and evaluation under unified 5G-compliant conditions.

3. Migration Framework and System Model

We propose a five-layer framework, in which two layers, the Cache Manager and the Migration Management, constitute novel extensions to the conventional 5G network architecture. These two components represent the core contributions of our work. In Section 3, we extend this framework with a formal mathematical description, including the system environment, problem formulation, and RL model definition. The complete framework and its constituent layers are illustrated in Fig. 1. The functionality of each layer is detailed as follows:

Mobility Layer: This layer encompasses the various types of mobile users in the network, including bicycles, pedestrians, vehicles, and IoT devices. Each user type generates service requests with specific resource demands. The supported service types include augmented reality, location-based services, alert/response messaging, and smart home sensor data. These service types are associated with distinct QoS and resource requirements that impact the system's behavior and decisions at higher layers.

Network & MEC Servers Layer: This layer consists of gNBs (next-generation NodeBs), MEC servers, and an extended cache memory component. It is responsible for hosting and managing service instances, allocating resources such as CPU, disk, RAM, and cache memory. During service placement or migration, the MEC servers handle the allocation of these resources. Additionally, this layer collects operational data that feeds into both the cache management and machine learning layers for further decision-making.

Cache Management Layer: This layer introduces a Cache Manager, which handles data fetching, allocation, and eviction within the MEC server's cache memory. Upon service placement, the cache manager determines whether the required data is already present or must be fetched from the cloud.

Migration Layer: This layer is responsible for enabling intelligent, data-driven service migration within the MEC environment. It incorporates a learning-based decision-making module that continuously observes the system state, including mobility patterns, resource availability, and cache memory status. By integrating cache-awareness into the migration logic, this layer introduces a novel enhancement over traditional approaches. It supports various migration strategies, ranging from random policies to machine learning-based decisions, allowing adaptive and context-aware migration across MEC nodes. The policies implemented within this layer are detailed in Table 2.

Cloud Server Layer: The last layer consists only of cloud servers that contain the data for the services that the cache manager fetched when required.

Table 2
Summary of migration strategies evaluated in the study.

Algorithm	Description
Random	This algorithm migrate services randomly.
Availability & number of applications	It focuses first on finding the edge server with the least number of applications running and availability.
Location	It focuses on allocating services closer to the user and availability.
Knapsack (Greedy Baseline)	A heuristic algorithm that selects the target MEC by maximizing a weighted gain–cost ratio, balancing latency reduction, migration overhead, and multi-resource pressure.
ML	Lastly, the RL algorithm learns mobility patterns and resource availability.

3.1. System Model

In our design, service migration is coordinated by a centralized controller that acts as the decision-making agent. All edge servers periodically report their resource states (CPU, RAM, Disk, Cache occupancy, etc.) to the controller. The centralized design is a deliberate choice that simplifies coordination logic and enables clean benchmarking of migration decision quality without confounding inter-agent communication overhead. This assumption is consistent with related MEC RL works Peng et al. (2024b); Tsourdinis et al. (2023), which adopt centralized control for the same reasons. Further discussion on complexity is provided in Section 3.2.1. Based on this global state, the RL agent selects the target server for migration. Edge servers themselves do not engage in direct communication or negotiation, which reduces overhead and simplifies coordination logic. After describing the migration framework, we must define the model of the system. We start by defining the environment in which the agent will interact.

Definition 1. *Given an environment E composed of the following elements:*

- gNB_n*: a set of n 5G base stations (gNBs),*
- S_m*: a set of m edge servers,*
- C_k*: a set of k cloud servers,*
- UE_l*: a set of l user equipments (UEs),*
- M_r*: a set of r distinct service types,*

we define the multi-service 5G cloud-native network $\mathbf{C}$ at a specific discrete time step $\tau \in \mathbb{N}$ as:

$$E_{C,\tau} = \left(X_\tau(S_m), X_\tau(C_k), P_\tau(S_m, m_r), P_\tau(C_k, S_m, m_l)\right) \quad (1)$$

where:

- $X_\tau(S_m)$*: the internal state of each edge server $s \in S_m$ at time τ, including CPU, memory, disk, and cache usage.*
- $X_\tau(C_k)$*: the internal state of each cloud server $c \in C_k$ at time τ, including CPU, memory, disk, and cache usage.*
- $P_\tau(S_m, m_r)$*: placement and execution status of service type $m_r \in M_r$ on edge servers at time τ.*
- $P_\tau(C_k, S_m, m_l)$*: migration placement involving edge server and service type $m_l \in M_r$ at time τ.*

We assume $m, n, k, l, r \in \mathbb{N}$, representing the number of edge servers, gNBs, cloud servers, user equipments, and service types, respectively.

The separation of $X_\tau(S_m)$ for edge servers allows for fine-grained control and analysis of edge-level resources, which are often constrained and more volatile compared to centralized cloud resources. This separation improves visibility into latency-sensitive service deployment and enables better optimization strategies.

Definition 2. ***(State Transition)*** *The environment is modelled as a Markov Decision Process (MDP), where the next state depends only on the current state and action:*

$$E_{C,\tau+1} \sim \mathcal{T}(E_{C,\tau}, a_\tau) \quad (2)$$

where $\mathcal{T}$ is the stochastic transition function that governs the evolution of the system. This satisfies the Markov property:

$$\begin{aligned}\mathbb{P}(E_{C,\tau+1} \mid & E_{C,\tau}, a_\tau, E_{C,\tau-1}, a_{\tau-1}, \dots) \\ &= \mathbb{P}(E_{C,\tau+1} \mid E_{C,\tau}, a_\tau)\end{aligned} \quad (3)$$

This assumption simplifies policy learning and allows RL algorithms to model and optimize sequential decisions efficiently.

3.2. Problem Statement

In our system, the latency experienced by users is directly influenced by their mobility patterns, which determine the time-varying network distance between a user and candidate edge servers. Let $\mathbf{x}_u(t)$ denote the position of user u at time t, and let $\mathbf{x}_e$ denote the position of edge server e. The user–server distance is defined as

$$d_{u,e}(t) = \|\mathbf{x}_u(t) - \mathbf{x}_e\|.$$

Since $\mathbf{x}_u(t)$ evolves over time according to the adopted mobility model, the distance $d_{u,e}(t)$ is inherently time-dependent. The network latency $\mathcal{L}(t)$ is modeled as a function of this distance, i.e.,

$$\mathcal{L}(t) = f(d_{u,e}(t)),$$

thereby capturing mobility-induced latency variations. At the same time, the migration time $\mathcal{M}_t$, defined as the duration required to reallocate a service, depends primarily on

the computational load and resource availability of the destination edge server. This includes CPU capacity, memory, disk space, cache occupancy, and the number of currently running applications.

Therefore, our objective is to minimise a weighted combination of network latency ($\mathcal{L}$) and the migration time ($\mathcal{M}_t$) for services, considering edge resource constraints. The optimisation problem selects the best placement or migration decision that adheres to each server's hardware resource limits and capacity constraints.

We aim to minimise the overall service latency and migration time by accounting for user mobility and edge server capacity. The optimisation problem is defined as follows:

$$\min \mathbb{E}_\tau \left[\mathcal{L}(\tau) + \mathcal{M}_t(\tau)\right] \tag{4}$$

subject to:

$$\text{CPU}_{m_r} \leq \text{CPU}_s, \quad \forall m_r \in M_r,\ s \in S_m, \tag{5}$$

$$\text{RAM}_{m_r} \leq \text{RAM}_s, \quad \forall m_r \in M_r,\ s \in S_m, \tag{6}$$

$$\text{Disk}_{m_r} \leq \text{Disk}_s, \quad \forall m_r \in M_r,\ s \in S_m, \tag{7}$$

$$\text{Cache}_{m_r} \leq \text{Cache}_s, \quad \forall m_r \in M_r,\ s \in S_m, \tag{8}$$

$$\sum_{m_r \in M_r} x_{m_r,s}(\tau) \leq \text{MaxApps}_s, \quad \forall s \in S_m, \tag{9}$$

$$x_{m_r,s}(\tau) \in \{0,1\}, \quad \forall m_r \in M_r,\ s \in S_m, \tag{10}$$

Variable Definitions:

- τ: discrete time index,
- $x_{m_r,s}(\tau)$: binary variable indicating whether service m_r is hosted on server s at time τ,
- $\text{CPU}_{m_r}, \text{RAM}_{m_r}, \text{Disk}_{m_r}, \text{Cache}_{m_r}$: resource demands of service m_r,
- $\text{CPU}_s, \text{RAM}_s, \text{Disk}_s, \text{Cache}_s$: available resources of server s,
- MaxApps_s: maximum number of applications allowed to run on server s.

3.2.1. Problem Complexity and Optimization Model Justification

While the objective function in Eq. (4) aims to jointly minimize network latency $\mathcal{L}(\tau)$ and migration time $\mathcal{M}_t(\tau)$, the underlying decision problem is both non-linear and combinatorial in nature. Specifically, the problem includes:

- **Binary decision variables** for service placement and migration: $x_{mr,s}(\tau) \in \{0,1\}$,
- **Dynamic state space** due to heterogeneous user mobility and service demands,
- **Non-linear effects** from cache hit/miss behavior and service reinitialization overhead.

The number of possible migration actions increases rapidly with the number of edge servers n, services m, and user devices l, leading to a solution space of approximate size $\mathcal{O}(n \times m \times l)$ at each time step.

Such characteristics make the problem intractable for classical optimization methods. In particular, the formulation resembles a **Mixed-Integer Non-Linear Programming (MINLP)** problem, which is known to be **NP-hard**.

Therefore, instead of solving it with MILP solvers, which scale poorly in highly dynamic environments, we adopt a **model-free RL** approach. RL enables efficient policy learning through interaction with the environment, without requiring a full system model or explicit enumeration of all possible states and actions. This allows our framework to remain scalable and adaptable to real-time variations in mobility, resource availability, and cache conditions. To improve readability and consistency, we summarize all key symbols, indices, and variables used throughout the paper in Table 3. This includes sets, state variables, decision variables, and RL-specific components such as observations, actions, and rewards. The table allows the reader to interpret equations, algorithms, and the RL framework without ambiguity.

In addition, in the system model the overhead grows linearly with the number of nodes and may become significant in large-scale deployments. The centralized architecture may also introduce scalability limitations and a single point of failure. These trade-offs are discussed further in Section 6.

3.3. RL definition

This is due to the fact that migration decisions must be made in a dynamic and uncertain environment where user mobility and resources change constantly over time. RL algorithm is well-suited for this type of environment as it could learn from unhidden patterns. It learns a policy π that minimizes long-term latency and migration time through interaction with the environment. It does not require a complete system model, enabling flexible adaptive service orchestration. Given the suitability of RL for this dynamic optimisation problem, we now define the observation space, action space, and reward function used to train the RL agent.

Definition 3. ***(Observation Space)*** *The observation space $\mathcal{O}$ defines what the agent perceives from the environment at time step τ. Each observation $o_\tau \in \mathcal{O}$ includes:*

$$o_\tau = \{(X_\tau(S_m), X_\tau(C_k), \textit{availability}, \textit{distance}, \textit{number of applications}\} \tag{11}$$

Formally, we define:

$$\mathcal{O} \subseteq \mathbb{R}^{d_o} \tag{12}$$

where d_o is the total number of features describing the state of all edge/cloud servers and service deployments. This allows the agent to observe current resource utilization,

Table 3
Notation Summary for Service Migration Framework

Symbol	Description
gNB_n	Set of 5G base stations
S_m	Set of edge servers
C_k	Set of cloud servers
UE_l	Set of user equipments
M_r	Set of service types
$E_{C,\tau}$	Environment state at time τ
$X_\tau(S_m)$	Edge server state
$X_\tau(C_k)$	Cloud server state
$P_\tau(S_m, m_r)$	Placement of service on edge
$P_\tau(C_k, S_m, m_l)$	Migration of service
$\mathcal{T}$	Stochastic state transition function
a_τ	Action by RL agent
$\mathcal{A}$	Action space
d_a	Action space dimensionality
o_τ	Observation
s_t	RL state (equivalent to o_τ)
$\mathcal{O}$	Observation space
d_o	Observation space dimensionality
r_τ	Reward
π	Policy
γ	Discount factor
T	Episode horizon
$\mathbf{x}_u(t)$	User u position at time t
$\mathbf{x}_e$	Edge server e position
$d_{u,e}(t)$	Euclidean distance between u and e at t
$\mathcal{L}(\tau)$	Network latency
$\mathcal{M}_t(\tau)$	Migration time
$x_{m_r,s}(\tau)$	Binary: 1 if m_r hosted on server s
$\text{CPU}_{m_r} \dots \text{Cache}_{m_r}$	Resource requirements of m_r
$\text{CPU}_s \dots \text{Cache}_s$	Server available resources
MaxApps_s	Max applications per server
$w_{1\dots4}$	Reward weighting factors
$\text{eff}_{CPU}, \text{eff}_{RAM}$	Utilisation efficiency ratios
$\text{pen}_{Cache}, \text{rew}_{Cache}$	Cache miss penalty / hit reward
pen_{avail}	Resource unavailability penalty
α, β, λ	Knapsack score weights
ε	Constant (prevent division by zero)
$\text{gain}(e), \text{cost}(e)$	Latency gain / Migration cost for MEC e
$\text{pressure}(e)$	Multi-resource scarcity for MEC e

data cache status, and location. It is important to note that the user mobility is captured in the observation space via distance measurements, which dynamically reflect changing user positions and influence the agent's latency-aware decisions.

Bandwidth is intentionally excluded from the state representation. The aim is to isolate the effect of cache-awareness. Thus by using the latency, included in the reward function, indirectly reflects network performance, including potential bandwidth bottlenecks.

Definition 4. ***(Action Space)*** *The action space $\mathcal{A}$ defines all possible actions the agent can take at each time step. The action $a_\tau \in \mathcal{A}$ defines where the service will be migrated:*

- *Migrating a service $m_r \in M_r$ on a server $s \in S_m$, that user $ue_l \in UE$ is utilizing.*

We formally define:

$$\mathcal{A} \subseteq \mathbb{R}^{d_a}$$

where d_a is the dimensionality of the action vector. Actions are discrete placement choices for the migration.

Definition 5. ***Reward Function*** *After each action, the agent receives a scalar reward $r_\tau \in \mathbb{R}$ defined as:*

$$r_\tau = w_1 \cdot efficiency_{CPU} + w_2 \cdot efficiency_{ram} + w_3 \cdot latency \quad (13)$$

if resources are not available for action a then:

$$r_\tau \mathrel{+}= \begin{cases} w_4 \cdot penalty_{Cache}, & \text{if cache condition is violated} \\ w_4 \cdot reward_{Cache}, & \text{otherwise} \end{cases} \quad (14)$$

Where: *w_1, w_2, w_3, w_4 are weighting factors balancing efficiency and penalty terms.*

if with the action $a_\tau \in \mathcal{A}$, which indicates the edge server to migrate the service, does not have available resources, the total reward is as defined in Eq. (15)

$$r_\tau \mathrel{+}= penalty_{availability} \quad (15)$$

The reward function $r(\tau)$ reflects system objectives such as minimising latency, balancing cache memory load and resources.

The RL agent seeks to learn a policy π that maximizes the expected cumulative reward:

$$\max_\pi \mathbb{E}_\pi \left[\sum_{\tau=0}^{T} \gamma^\tau \cdot r_\tau \right] \quad (16)$$

where $\gamma \in [0, 1]$ is the discount factor, determining the trade-off between short-term and long-term performance.

The Cache misses in the reward function are penalised by applying a negative weight when required data is not available in the corresponding Edge Server. This encourages the agent to favor servers with relevant cached content. This reflects the increased latency and resource usage associated with fetching data from the cloud.

3.4. Algorithm

Our main contribution and difference with the state of the art is the inclusion of cache memory information as part of the decision-making. For this study Least Frequently Used (LFU) is selected for cache policy. The rationale is its simplicity and determinism, which enables us to isolate and study the impact of cache state in the RL agent's migration

decisions. To investigate the interaction between migration policies and other cache strategies is left for future work. The RL algorithm considered the usage of cache memory and mobility information as part of the state space. The algorithm (1) and (2) describe the migration algorithm selection.

The first step in the algorithm (1) involves gathering the necessary information demanded by the service to migrate. This information serves as the input for the various migration algorithms listed in 2, as well as for the RL agent, where it forms part of the observation space defined in Eq. (11). For the different migration strategies to function, it is necessary to know the number of edge servers n, and which cache policy is used $cachePolicy$.

Subsequently, the migration algorithm is selected as $migType$. To enable the continuous checking of migration, a migration event starts. The migration process is triggered by a fixed interval timer implemented in the simulation environment. Once this timer reaches zero, a migration check event is raised, which evaluates whether the service should be migrated and where. As per comparison with the RL algorithm, the decision is based on the corresponding migration policy (random, availability-based, location-based, or RL-driven).

Since the simulation is implemented in OMNet++, an event is created to examine if a service migration is required, which triggers a flag. This migration event is defined in Line six in the algorithm 1, and the migration flag is evaluated in Line nine.

The event is scheduled to begin after the service placement is completed, that is, once the service is fully operational on the designated edge server. After service deployment, users begin to transmit data while the migration flag is concurrently monitored. If the flag is true, the migration process starts checking which migration algorithm was selected previously and executes it as described in algorithm 2. The different migration algorithms are outlined in Algorithm 2, with detailed descriptions provided in Table 2.

Migration-Aware Greedy Knapsack Baseline: As stated in table 2, we selected and implemented a Knapsack optimization algorithm for placement and migration events. This algorithm serves as the baseline optimization comparison with our RL design.

For each candidate MEC host e, we compute a scalar utility function:

$$\text{score}(e) = \frac{\alpha \cdot \text{gain}(e) - \beta \cdot \text{cost}(e)}{\max\left(\text{pressure}(e), \varepsilon\right)}. \tag{17}$$

This formulation balances latency (gain), migration overhead (cost), and multi-resource scarcity (pressure), providing a effective decision heuristic. The constants α, β, and λ correspond to the tunable parameters.

Gain Term: The **gain** term captures the latency benefit achieved by selecting a candidate MEC. For migration events, given the observed end-to-end latency before migration L_{before} and the estimated post-migration latency $L_{\text{after}}(e)$, we define:

$$\text{gain}(e) = \max\{0,\ L_{\text{before}} - L_{\text{after}}(e)\}. \tag{18}$$

Algorithm 1 Migration Algorithm Selection

1: **Input:**
2: - Resource requirements of service m_r: $(CPU_{m_r}, RAM_{m_r}, Disk_{m_r}, Cache_{m_r})$
3: - Set of edge servers S_m
4: - Selected migration policy $migType$
5: **Output:**
6: - Simulation performance metrics (latency, , active number of application, migration time, cache stats)
7: Finish = $False$
8: migration check event starts
9: n ← $numberofedgeservers$Finish ← $False$
10: $cachePolicy$ ← LFU
12: $n \leftarrow |S|$
13: Construct state $s_t = o_\tau$(resources, cache status, distance, active applications)
14: Trigger migration-check event
15: **while** !(Finish) **do**
16: User transmits data
17: Update system $state$
18: **if** migration **then**
19: **if** $migType$="random" **then**
20: edgeToMig ← $random(n)$
21: **else if** $migType$="availability" **then**
22: edgeToMig ← $availability(n)$
23: **else if** $migType$="location" **then**
24: edgeToMig ← $location(n)$
25: **else if** $migType$="agentAI" **then**
26: edgeToMig ← $agentAI(state)$
27: **end if**
28: allocateService($edgeToMig$, $cachePolicy$)
29: **end if**
30: **if** simulation termination condition is met **then**
31: $Finish \leftarrow True$
32: **end if**
33: **end while**
34: **return**

If L_{before} is unavailable (e.g., during the first allocation), a latency-aware proxy is used:

$$\text{gain}(e) = \frac{1}{L_{\text{after}}(e)}. \tag{19}$$

Latency estimation follows:

$$L_{\text{after}}(e) = L_0 + k \cdot d(e) + \text{loadPenalty}(e), \tag{20}$$

where $d(e)$ is the UE–MEC distance, k is a distance coefficient, and loadPenalty(e) increases with MEC CPU utilization.

Cost Term: The cost term reflects migration delay due to state transfer and service warmup:

$$\text{cost}(e) = \text{transferMs} + \text{warmupMs}.$$

The transfer time is estimated as:

$$\text{transferMs} = \frac{\text{stateMB}}{\text{throughputMBs}} \cdot 1000,$$

Algorithm 2 Migration Algorithms

Function: $random(n)$
Function: `random(`n`)`
Input: Number of edge servers n
Output: Selected edge server index $index$
$index \leftarrow \lfloor random(0,1) \cdot n \rfloor + 1$
return $index$

Function: `availability(`n`)`
Input:
    - Number of edge servers n
    - Resource availability per server
Output: Selected edge server index $selected$
$minApps \leftarrow \infty$
$selected \leftarrow -1$
for $i = 1$ to n **do**
  if resAvailable(i) AND numApps(i) $<$ $minApps$ **then**
    $minApps \leftarrow numApps(i)$
    $selected \leftarrow i$
  end if
end for
return $selected$

Function: $location(n)$
Input:
    - Number of edge servers n
    - User-to-server distance information
Output: Selected edge server index $selected$
$minDistance \leftarrow \infty$
$selected \leftarrow -1$
for $i = 1$ to n **do**
  $d \leftarrow compDistance(i)$
  if resAvailable(i) AND $d < minDistance$ **then**
    $minDistance \leftarrow d$
    $selected \leftarrow i$
  end if
end for
return $selected$

Function: $agentAI(state)$
Input: Current environment state $state$
Output: Selected edge server index $action$
$action \leftarrow policy(state)$
return $action$

stateMB is proportional to the application's RAM demand, and throughputMBs is the assumed inter-MEC throughput. Warmup delay aggregates container startup and cache reinitialization penalties. For first allocations, cost is set to zero. This uniform treatment keeps the baseline simple, deterministic, and reproducible.

Pressure Term: The pressure term models multi-resource scarcity through a knapsack-inspired expression:

Table 4
Parameter settings for the greedy knapsack baseline used for comparison with RL agents.

Symbol / Name	Default
α (`ALPHA_GAIN`)	1.0
β (`BETA_COST`)	1.0
λ (`LAMBDA_APPS`)	0.5
ε (`EPS`)	10^{-6}
`DEFAULT_STATE_MB`	$0.5 \cdot d_{\text{ram}}$
`DEFAULT_TPUT_MBPS`	100 MB/s
`CONTAINER_WARM_MS`	150 ms
`CACHE_WARM_MS`	50 ms
L_0	5 ms
k	0.005 ms/m

$$\text{pressure}(e) = \frac{d_{\text{cpu}}}{\max(f_{e,\text{cpu}}, \varepsilon)} + \frac{d_{\text{ram}}}{\max(f_{e,\text{ram}}, \varepsilon)} + \frac{d_{\text{disk}}}{\max(f_{e,\text{disk}}, \varepsilon)} + \mathbb{I}_{\text{cache}} \frac{d_{\text{cache}}}{\max(f_{e,\text{cache}}, \varepsilon)} + \lambda \cdot \frac{\#\text{apps}(e)}{\text{maxApps}(e)}. \tag{21}$$

Here, d_r and $f_{e,r}$ denote the resource demand and free capacity for resource r, respectively. The final term softly penalizes MECs already hosting many applications.

Algorithm knapsack: The algorithm is found in Appendix A.

Implementation and Parameters

The parameters in Table 4 control latency modelling, migration overhead, and resource balance. Neutral values ($\alpha = 1, \beta = 1, \lambda = 0.5$) are used to maintain transparency and reproducibility. Parameter sweeps over $\alpha \in \{0.5, 1, 2\}$, $\beta \in \{0.5, 1, 2\}$, and $\lambda \in \{0.1, 0.5, 1\}$ did not alter the qualitative outcome.

3.5. Discussion and Motivation

This baseline is deliberately designed to be simple and explainable. It (i) reacts to user mobility via latency L, (ii) accounts for migration overheads, and (iii) respects multi-resource scarcity. However, it remains a myopic heuristic: it lacks awareness of future mobility, temporal cache evolution. Hence, while valuable for benchmarking, it is expected that an RL policy, capable of learning and aiming for a long-term goal, can adapt to the interactions and dynamics of the environment. Thus, it will outperform this static greedy rule over extended horizons.

4. Setup & Simulation

OMNet++ was selected for its usefulness in simulating network protocols. The library for 5G network simu5G Nardini, Sabella, Stea, Thakkar and Virdis (2020) was also utilized as it emulated the 5G protocols necessary for building a 5G networks, including nodes such as gNB and Edge Servers. Following this methodology, we ensure that the RL

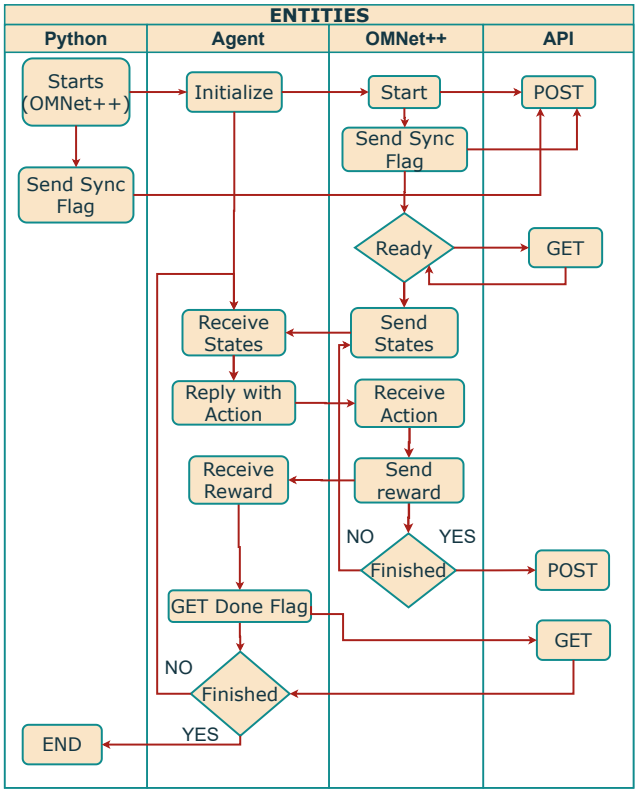

Figure 2: Diagram of the migration simulation, showing interactions between the entities (platforms).

agent learns from a simulated network that is closer to reality, which can be evaluated and analysed in more detail.

One of the contributions of this project was the development of a REST API that enables the communication between OMNet++ and the RL algorithm in Python, overcoming the compatibility issue in existing Schettler, Buse, Zubow and Dressler (2020). We replaced the ZeroMQ and protobuffer libraries with the REST API to allow utilization across machines and environments. This enables easier reproducibility. The integration of the different SW languages, platforms and frameworks is illustrated in Fig. 2, which outlines the different entities. It also illustrates both the initialization procedure and the agent-environment interaction loop. The simulation process is initiated from Python, which launches the OMNeT++ environment and establishes the communication. The system architecture comprises four main components: (i) the OMNeT++ network simulator, (ii) the Python environment responsible for coordination and control, (iii) a learning agent that interacts with the environment to optimize a decision policy, and (iv) an API interface facilitating bidirectional communication between OMNeT++ and Python.

The initialization phase requires synchronization between OMNeT++ and Python to ensure the proper exchange of observations, actions, and rewards. Once synchronization is achieved, the agent-environment loop begins. Each time a service migration is triggered within OMNeT++, the current environment state s_t is captured and formatted as

Table 5
Average mobility speeds for different user types considered in the simulation.

User Type	Average Speed
Pedestrian Forde and Daniel (2021)	1.45 (m/s)
Bicycle Selesnic and Kodsi (2016)	5.8 (m/s)
Vehicle (By Traffic Low)	13.8 (m/s)
Smart Home/Company (Static)	0 (m/s)

Table 6
Resource requirement per service.

Service Type	CPU	RAM	Disk	Cache
Augmented Reality	30MIPS	3GB	10GB	160MB
Smart Home	5000IPS	150MB	100MB	160MB
Localization GPS	5MIPS	500MB	500MB	160MB
Alert/Response Messages	500IPS	10MB	10MB	110MB

defined in Eq. (11). The agent then selects an action a_t according to its policy $\pi(a_t|s_t)$, which is stochastic. The selected action is sent back to OMNeT++, which executes the corresponding migration decision and returns a reward r_t, computed as specified in Eq. (14), to the agent.

This interaction continues until the simulation reaches a terminal state. Upon episode completion, OMNeT++ signals Python via a predefined flag, indicating that the current training episode has ended and that the environment can be reset for the next iteration.

4.1. Definition & Assumptions

As there are many types of users and services in a real-world scenario, we opted for using different types of end users to emulate different mobility patterns, as each of them has its own assigned speed. We simulate user mobility using the OMNeT++ framework. Each user is assigned a random destination. Users may represent pedestrians, vehicles, cyclists, or static IoT devices, and move within the simulation area at speeds determined by their mobility type. This methodology allows us to have dynamic user-to-server distances and gives the RL agent more stochastic environment to adapt to different mobilities without relying on explicit trajectory prediction or mobility models. The average speed for each of the user types was found in the literature and is stated in the Table 5.

Additionally, each user also has a different type of service. This diversity creates an environment where the RL agent could learn to take action in a dynamic environment and can adapt to different changes if other type of users or services is implemented. For those services, we have configured the resource requirements: CPU, RAM, and Disk. The requirements are described in the table 6.

The values presented in Table 6 were selected based on a thorough analysis of previous studies that evaluated similar application types, such as augmented reality, localization, and infotainment services Sonmez, Ozgovde and

Table 7
Simulation and RL Hyperparameter Settings

Parameter	Value
Simulation Settings	
Tile Dimension	1000×700 m^2
Wireless Network	5G
Number of Edge Servers	5
Number of Users	15–50
CPU Speed per MEC Server	1 GIPS
RAM per MEC Server	16 GB
Disk per MEC Server	500 GB
Cache Memory per MEC Server	1000 MB
RL Hyperparameters (shared)	
Training Episodes	1400
Discount Factor γ	0.99
Learning Rate α	0.001
Initial Exploration ϵ	1.0
Min. Exploration $\epsilon_{\min}$	0.01
Exploration Decay	0.995
Replay Buffer Size	10,000
Batch Size	64
Target Update Freq.	10 steps
NN Hidden Layers	2×32 (ReLU)
Optimizer	Adam
State Size d_o	40
Action Size	5
Convergence Criterion	100-episode rolling average reward within ±2% after episode 1000

Ersoy (2019); Myyara, Darif and Farchane (2024). Regarding memory cache size, different data files were assigned to each application, which are intended to be shared among multiple users. These files may represent databases, images, or videos. To ensure a realistic cache saturation scenario, we selected comparable cache size requirements across applications. This design choice allows the learning agent to develop a policy capable of managing increasingly demanding resource requirements.

The network configuration in OMNeT++ consists of a grid measuring 1000×700 meters, comprising five edge servers, between three and ten nodes per user type, and a single cloud server. Each edge server is equipped with cache memory, which stores data retrieved from the cloud in the event of a cache miss. As previously discussed, the cache-aided approach is designed to reduce latency by minimising the frequency of data retrievals from the cloud.

The values related to the MEC servers' CPU, RAM, and disk, presented in Table 7, were selected after a detailed review of the literature, particularly studies such as Sonmez et al. (2019); Sonmez, Ozgovde and Ersoy (2018). Initially, the CPU capacity was configured at 10 GIPS; however, it was intentionally reduced to 1 GIPS in order to constrain the server capacity and better evaluate system performance under limited resource conditions. This adjustment aligns with the service requirements outlined in Table 6. Regarding RAM and disk storage, although there is greater flexibility in their selection, we again consulted relevant literature Kaftantzis, Kogias and Patrikakis (2024) and adjusted the parameters accordingly. The final configuration sets the RAM at 16 GB and the disk capacity at 500 GB, reflecting realistic resource demands based on the targeted service profiles.

Regarding the hyperparameters, after several simulations, we concluded that this heterogeneous mobility and service environment benefits from a high discount factor of $\gamma = 0.99$. Since users move continuously across space, the agent's actions often have delayed consequences due to handovers. A high discount factor encourages the agent to plan for service continuity, focusing on long-term Quality of Service (QoS) objectives. Additionally, we selected a low learning rate ($\alpha = 0.001$) to ensure stable learning in a large and dynamic state space. The replay buffer of 10,000 transitions provides sufficient experience diversity to decorrelate sequential samples under heterogeneous mobility patterns. This choice helps prevent the agent from quickly forgetting important states, especially those influenced by varying mobility patterns observed in this study. All three DRL variants (DQN, DDQN, and DDDQN) share identical hyperparameters as listed in Table 7, ensuring that observed performance differences are attributable solely to architectural choices rather than training configuration. Convergence is assessed both visually and numerically in Section 5.3.2.

4.2. Baseline Evaluation Rationale

To ensure a fair and controlled evaluation, we adopted a set of baselines. These baselines represent the main categories of decision-making strategies in MEC service migration. The comparison between RL with Cache and RL without Cache is treated as an internal ablation study. With this approach, we isolate the contribution of cache-awareness to migration performance. The evaluated baselines include: Random migration, a Greedy Knapsack heuristic, and multiple deep reinforcement learning variants (DQN, DDQN, and Dueling DDQN). This selection enables coverage of three fundamental decision paradigms: (i) uninformed policies (Random), (ii) rule-based optimization without learning (Knapsack), and (iii) learning-based migration strategies (deep RL variants).

The Greedy Knapsack baseline approximates a cost-aware, deterministic optimization strategy by selecting migration targets based on a predefined utility function that balances resource availability and migration cost. This provides a strong non-learning benchmark for comparison against RL-based approaches. While additional RL-based migration frameworks exist in the literature, direct numerical re-implementation is not feasible due to inconsistencies in system assumptions and experimental environments, as discussed in Section 2.3. Specifically, differences in state/action space definitions and reward formulations in Peng et al. (2024b), as well as incompatible network stack assumptions in Tsourdinis et al. (2023), prevent fair and reproducible integration into our OMNeT++/Simu5G-based simulator.

To ensure experimental consistency, all evaluated methods are implemented under identical mobility, caching, and

resource configurations. This controlled setup enables us to isolate the impact of cache-awareness and RL design choices. We can eliminate confounding differences in simulation environments and system architectures.

5. Results

In this section, we present the results with the comparison of different migration policies to demonstrate the improvement the Cache-aided approach brought to the network. The analysis is based on latency, migration time, and Cache Hits and Misses. These Key Performance Indicators (KPIs) reflects the improvement the agent has over the other type of policies by learning how to migrate services, considering resources, cache memory, and mobility.

More importantly, the results were gathered using a Monte Carlo strategy. This approach ensures that the randomness inherent in our environment is properly captured and accounted for in the evaluation. Specifically, we performed 100 independent runs per policy (Random, availability, location, ML/RL, see Table 2). Each run introduces variability due to the stochastic nature of the environment: users are randomly allocated within a 1000m x 700m grid, their source and destination positions are randomly selected, the starting time of each data transmission is randomised, and the number of users varies between 3 and 10. The Monte Carlo method allows us to average performance across a wide range of possible scenarios, leading to more robust and generalizable conclusions. This approach mitigates the bias of evaluating policies on a single or limited set of conditions and highlights the policy's ability to perform consistently under uncertainty.

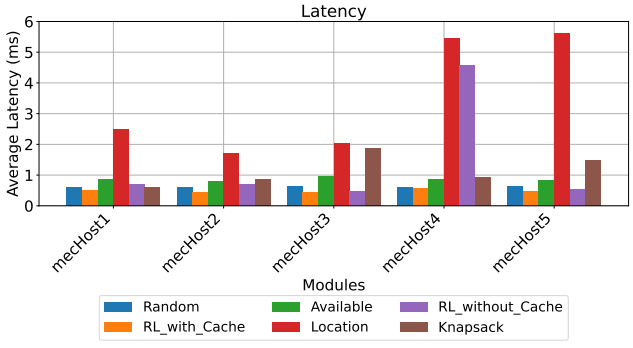

Figure 3: Average latency per MEC server comparing Random, Availability-based, Location-based, Knapsack, RL_without_Cache, and RL_with_Cache migration policies under heterogeneous mobility conditions.

5.1. Latency & Migration Time

To demonstrate the effectiveness of the proposed migration RL algorithm, we evaluated two KPIs: latency on the network and migration time taken from a service to be up and running in the designated Edge server. These KPIs represent the primary objectives that the agent seeks to optimise during the learning process.

In Fig. 3, we present the latency comparison among all migration policies: Random, Available, and RL_with_Cache.

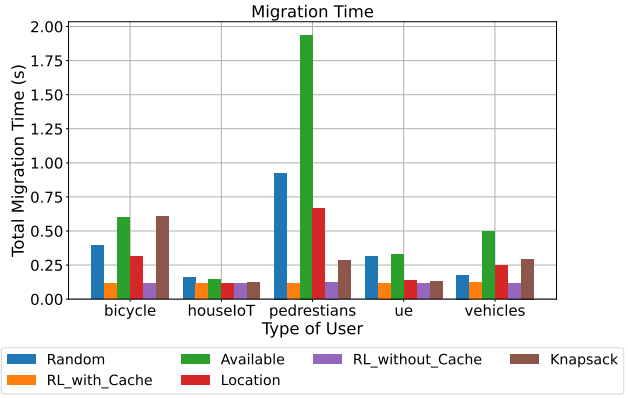

Figure 4: Total migration time per user type (bicycle, HouseIoT, pedestrian, UE, and vehicle) comparing Random, Availability-based, Location-based, Knapsack, RL_without_Cache, and RL_with_Cache migration policies.

Among them, the Random policy exhibits the most stable latency. However, in Fig. 4, it illustrates that Random results have a higher migration time on most MEC servers compared with the RL algorithm, indicating that random decisions do not optimise for migration efficiency. Furthermore, as shown in Fig. 5, random selection consistently supports a lower and nearly constant number of applications across all MEC servers compared to RL_with_Cache.

Because of its consistent service distribution, which tends to balance the load over time and produces averaged-out delay, the random selection appears to be stable. Nevertheless, this method is unable to prioritise service requirements and does not take into account current system conditions. Random, therefore, misses out on chances to optimise performance. These findings highlight that the Random strategy is unable to adjust to the ever-changing nature of network environments.

In contrast, the agent Rl_with_Cache adapts to the environment and achieves lower latency in 60% of the MEC servers, with an average latency reduction of approximately 20%. When compared with other policies, Available, Location, and RL_without_Cache in approximately 80%, 100%, and 60% of the MEC servers, respectively. These results confirm its effectiveness in reducing latency through informed, adaptive decision-making.

The random policy occasionally outperforms the RL-based approach in specific cases, especially under static or low-mobility user profiles. This is attributed to its uniform allocation behavior, which can inadvertently reduce server congestion. However, it lacks adaptability and context-awareness, and consistently underperforms in scenarios with dynamic resource demands or heterogeneous mobility. The RL agent, by contrast, learns to balance migration decisions across multiple factors (latency, resource availability, cache status), resulting in more reliable long-term performance across diverse scenarios.

For the migration time analysis, Fig. 4 displays the time per user type, which is the time experienced by the

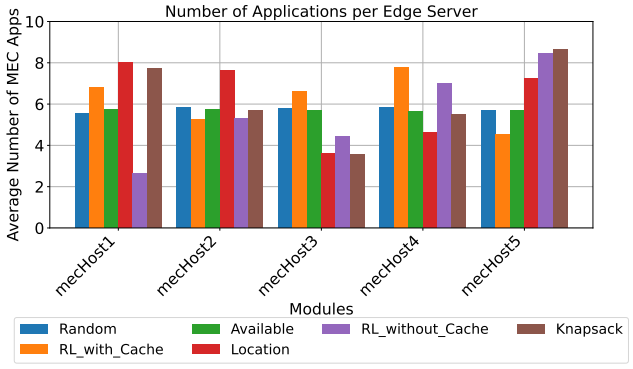

Figure 5: Average number of active applications per MEC server comparing Random, Availability-based, Location-based, Knapsack, RL_without_Cache, and RL_with_Cache migration policies.

users as the service must be migrated to another server. As observed, the RL_with_Cache demonstrated to perform with a lower value for 80% of the type of services. The other 20% corresponds to the UE type. It is a compromise for the agent. Another important observation is that for pedestrian mobility, the Random, Available, and Location-based algorithm migration result in higher migration times compared to the RL-based approach. Notably, in low-mobility scenarios, the Location-based policy often assigns the same MEC server for extended periods due to users remaining within the same area. This static assignment is not always optimal in terms of server load. Meanwhile, the Random and Available policies may assign MEC servers that are geographically farther from the user, leading to increased latency and inefficient migrations. These results highlight the advantage of our adaptive RL-cached aided strategy in a mobility heterogeneous environment.

In terms of migration time (Fig. 4), the method that performs slightly better is RL_without_Cache, showing a reduction of migration time for the bicycle and UE of approximately 8% and 74%, respectively, when compared to RL_with_Cache. In comparison with HouseIoT and Vehicles user type, both RL-based policies yield nearly identical results, with negligible differences approximately 1%. This similarity in their outcome is attributed to the static nature of HouseIoT devices, where the lack of mobility results in minimal variation between the agent with cache information and the one without. In the case of Vehicles, the high mobility may allow both policies to effectively learn user patterns, leading to comparable performance. Nevertheless, when compared with Pedestrian mobility type, the distinction between the two agents becomes more evident. Moderate mobility provides enough variability for the RL agent with cache information to benefit from additional information (cache). Therefore, incorporating cache data leads to better optimization, reducing migration time. For the pedestrian user type, RL_with_Cache clearly show a reduction of service migration time against RL_without_Cache by approximately 10%. Although the improvements in migration time alone may appear modest across some user types, a broader evaluation that considers additional metrics, such as latency and the number of concurrently running applications, consistently favors RL_with_Cache. This suggests that incorporating cache information enables more informed decision-making and provides better control across all user mobility profiles. Specifically, RL_with_Cache achieves up to 20% lower latency in 60% of MEC servers and maintains a higher number of active applications across 40% of them. This indicates that RL_with_Cache not only maintains competitive migration times but also ensures lower latency and better resource utilisation, ultimately reflecting more intelligent and adaptive decision-making under dynamic network conditions.

In comparison with the baseline Knapsack: As shown in Fig. 3, *RL_with_Cache* yields the lowest service latency across all MEC nodes, achieving an average improvement of 35%–50% relative to the Knapsack baseline and exceeding 60% compared to non-cache-aware policies. In Fig. 4. The RL-based policies achieve up to 70% lower migration delay than the Knapsack approach, particularly for pedestrian and vehicular users. This improvement stems from the RL agent's ability to anticipate mobility dynamics and perform proactive migrations, while the Knapsack heuristic reacts myopically to instantaneous latency fluctuations, leading to delayed or unnecessary migrations. The RL framework addresses this limitation, enabling it to predict the proper action due to its mobility awareness, cache-aided and resource states rather than optimizing only the current snapshot.

5.2. Cache Data

The analysis of cache hits (Fig. 6) and misses (Fig. 7) reveals that the RL_with_Cache policy achieves significantly more cache up to $2x$ higher than baseline approaches, due to its frequent migration decisions that actively seek to co-locate services with cached data. While this strategy also results in higher cache misses, it reflects a more proactive use of cached content, rather than inefficient cache behaviour. In contrast, the Location-based strategy shows the lowest number of cache misses, but this is largely due to the absence of migration triggers, which leads to fewer opportunities for cache utilisation. Therefore, when considering cache activity in conjunction with mobility behaviour, RL_with_Cache demonstrates a more dynamic and intelligent service placement approach that better leverages the cache to optimize performance.

While RL_without_Cache occasionally achieves comparable cache hits, it lacks the intelligent, cache-aware decision-making that characterises the RL_with_Cache agent. The latter demonstrates a more purposeful interaction with the cache system, achieving up to 43% more cache hits in some MEC hosts and enabling more dynamic service migrations, aligning well with its overall lower latency and improved decision control.

In conclusion, the cache-aided RL algorithm demonstrates superior decision-making capabilities by effectively

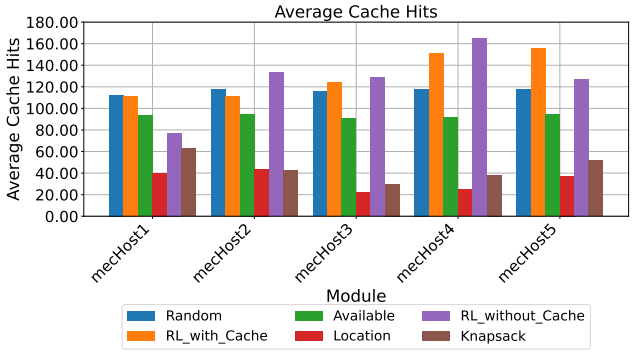

Figure 6: Average number of cache hits per MEC server comparing Random, Availability-based, Location-based, Knapsack, RL_without_Cache, and RL_with_Cache migration policies.

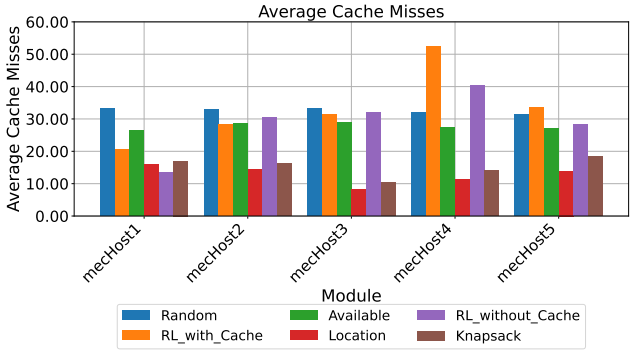

Figure 7: Average number of cache misses per MEC server comparing Random, Availability-based, Location-based, Knapsack, RL_without_Cache, and RL_with_Cache migration policies.

leveraging cached data to optimize service migration. This strategy not only increases cache hit rates but also significantly improves key performance metrics such as latency and migration time. By adapting to dynamic and data-intensive environments, the proposed approach aligns well with the performance and scalability requirements of 5G networks, where low-latency communication, efficient resource utilization, and seamless service continuity are critical.

In comparison with Knapsack: As depicted in Fig. 7, the *RL_with_Cache* policy consistently achieves the lowest number of cache misses, reducing them by approximately 40%–60% compared to the Knapsack heuristic across all MEC hosts. Correspondingly, Fig. 6 shows a 30%–50% increase in cache hits, confirming that the RL agent efficiently learns temporal data access patterns and proactively retains frequently requested content at the edge. In contrast, the Knapsack baseline lacks temporal awareness, treating cache occupancy as a static resource constraint; consequently, it exhibits frequent cache evictions and redundant data retrieval from the cloud.

Overall performance: These results confirm that the Knapsack baseline remains a myopic, suboptimal strategy under dynamic load and mobility conditions. In contrast, the RL frameworks demonstrate strong adaptability and long-term optimization capability. This is achieved due to the RL's ability to model temporal dependencies between user mobility, network state, cache and resource availability.

These results confirm that the Knapsack baseline remains a myopic, suboptimal strategy under dynamic load and mobility conditions. In contrast, the RL frameworks demonstrate strong adaptability and long-term optimization capability. This is achieved due to RL's ability to model temporal dependencies between user mobility, the network state, caching, and resource availability.

5.2.1. Computational Complexity of Compared Methods

To clarify the cost of each migration policy, we briefly analyze the computational complexity of those policies.

- The **Random** strategy selects a destination server randomly, resulting in O(1) time complexity per decision step. - The **Shortest-Distance** strategy computes distances to all available servers, requiring O(n) operations per decision, where n is the number of servers. - **Knapsack baseline** (Greedy heuristic): Has $O(n \cdot r)$ complexity per decision step, where r denotes the number of resource dimensions (e.g., CPU, RAM, disk). The algorithm computes latency gain, migration cost, and multi-resource pressure before selecting the highest scoring host. - The **DQN/DDQN** approach incurs O(k) complexity due to the forward pass through the neural network (with k being the number of parameters), and O(n) for selecting the best action.

5.3. RL Algorithms Comparison

In the previous section, we compared our RL-based migration approach against two baseline algorithms:

- **Random:** Migrates services to randomly selected MEC servers without considering user mobility, workload, or system state. - **Shortest Distance:** Always selects the closest MEC server to the user based on Euclidean distance, ignoring dynamic system conditions such as cache availability, congestion, or long-term performance trade-offs. - **Knapsack (Greedy Baseline):** It is a deterministic heuristic that evaluates all candidate MEC servers and selects the one that maximizes a weighted gain–cost ratio, refers to Table 2.

In this section, we extend the comparison to more advanced RL algorithms. **DQN:** A value-based RL agent that uses a single Q-network to estimate action values based on observed environment features. **DDQN:** An extension of DQN that reduces overestimation bias by decoupling action selection and value estimation using two separate neural networks (online and target). **DDDQN:** Builds on DDQN by incorporating a dueling network architecture, separating the estimation of state-value and advantage functions. This improves learning stability and enables more efficient policy optimization in highly dynamic environments.

These models are evaluated on multiple performance metrics including reward, latency, and migration time under varying cache-awareness and user mobility conditions.

5.3.1. RL algorithm

DQN algorithm selects the next action using the online network that evaluates it using the same network. This overestimates the Q-values. The Q-value formula Wang, Wang, Liang, Zhao, Huang, Xu, Dai and Miao (2024); Kiran, Sobh, Talpaert, Mannion, Sallab, Yogamani and Pérez (2022) is in Eq. (22).

$$Q(s,a) = r + \gamma \max_{a'} Q_{\text{online}}(s',a') \tag{22}$$

To overcome the problem of DQN, we have DDQN Wang et al. (2024) that uses the online network to select the best action, and uses the target network to evaluate it. The Q-value formula is in Eq. (23).

$$Q(s,a) = r + \gamma Q_{\text{target}}(s',a^*) \tag{23}$$

Additionally, we have the Dueling DDQN (DDDQN) that uses the same DDQN update. However, it also separates state-value and advantage. It is assumed that the value function $V(s')$ and advantage function $A(s',a)$ are estimated separately as outlined in Eq. (24).

$$Q_{\text{dueling}}(s',a) = V(s') + \left(A(s',a) - \frac{1}{|A|} \sum A(s',a') \right) \tag{24}$$

The agent chooses and evaluates actions using this adjusted Q-function instead of the raw Q-values.

5.3.2. Reward

In Fig. 8, it is observed that both learning processes with and without cache can learn an optimal policy after 1000 episodes. The values of the reward vary because the calculation for the RL algorithm without cache does not include the cache reward as stated in Eq. (13). Thus, no cache information was shared with the agent while deciding to migrate the service.

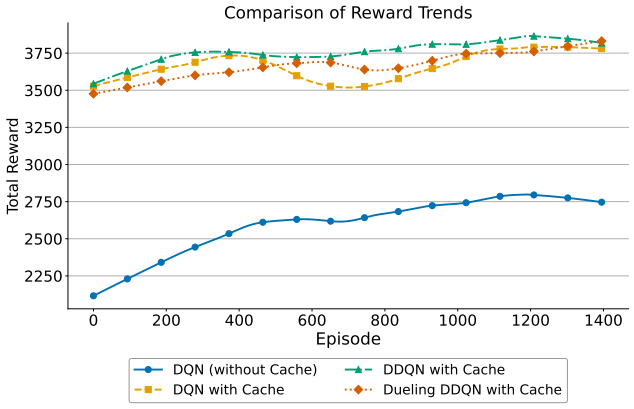

Figure 8: Cumulative reward per training episode for DQN without cache, DQN with cache, DDQN with cache, and Dueling DDQN with cache across 1400 training episodes.

In Fig. 8, one of the key limitations of the DQN approach, the overestimation of Q-values can be observed. To address this issue and enhance the stability of the learning process, we employed both DDQN and DDDQN. Furthermore, the integration of cache-related knowledge into the state representation and reward function contributed to a more stable and effective learning trajectory for the agent, as evidenced in the figure.

We examine the reward trajectories displayed in Fig. 8 to further demonstrate convergence performance. DQN without cache receives the lowest reward, since it lacks cache-related benefits. DDQN and DDDQN with cache, on the other hand, show quicker and more consistent convergence. Through the use of cache-aware state and duelling architecture, this stability reflects improved generalisation and less overestimation bias. These patterns held true throughout several runs, demonstrating the resilience of the learnt policies.

5.3.3. Latency and Migration

Concerning the latency and migration time, the DDDQN RL learning strategy was able to perform more effective than the DDQN and DQN. The calculations of the percentage for this analysis are made based on the average total migration time across all modules. The approach DDDQN achieved 19.9% reduction in latency compared to DQN, and 30% reduction compared to DDQN. Furthermore, whilst analysing migration time, the results showed a remarkable improvement of 36% over DQN, and 54% over DDQN when accounting for outliers. The enhancements are attributed to the structure of DDDQN, which combines both double Q-learning to mitigate overestimation bias and a duelling network architecture that separates value and advantage estimations. This strategy leads to more stable and focused learning. Even more, the results highlight the importance of using advanced RL architectures for optimising migration of services in dynamic 5G edge environments.

5.3.4. Implications for QoE

Although QoE is not explicitly modeled, the observed reductions in latency and migration time provide indirect insights into service continuity. Migration time directly determines the duration of service interruption during handover events. Therefore, the relative reduction observed in our framework (up to 54% compared to DDQN) implies a proportional decrease in service unavailability windows. For pedestrian users, where mobility is moderate, the observed improvements translate to shorter and less frequent service disruptions. For vehicular users, characterized by higher mobility and more frequent migrations, the cumulative effect is more pronounced, leading to improved perceived service continuity. These results suggest that cache-aware migration policies can contribute to more stable user experience, even without explicitly optimizing QoE metrics.

6. Limitations and Future Work

While this study presents a cache-aware and mobility-integrated RL framework for service migration, several limitations provide directions for future work:

Bandwidth Modeling: This work does not explicitly model bandwidth constraints in the state space. Although

Table 8
Average latency and migration time across all servers for different RL algorithms.

Metric	DQN	DDQN	DDDQN
LatencyCacheMec (avg)	0.8250	0.9461	**0.6613**
TotalMigrationTime (avg)	0.1856	0.2605	**0.1178**

latency captures some of its effects, future iterations will include explicit bandwidth-aware migration decisions.

Cache Policy Variety: The evaluation employs LFU exclusively. While it ensures stability, comparative studies with adaptive or LRU strategies would further validate the robustness of cache-awareness.

Service Interruption and Downtime: Migration success was evaluated via latency and cache metrics. However, direct service availability and interruption time were not modeled and will be considered in subsequent phases.

Real-world MEC Deployments: While simulation provides a cost-effective and flexible environment for testing RL-based migration under dynamic conditions, future work will include real-world implementation using physical edge servers with caching capabilities to validate latency measurements and migration behavior.

Mobility Prediction: Though mobility patterns were integrated, no trajectory prediction models were employed. Future work could assess the trade-off between prediction overhead and RL adaptability.

Reward Design Complexity: Current reward formulation focuses on resource efficiency and latency. Multi-objective reward tuning (e.g., fairness, user priority, energy) will be explored.

Service Interruption, Downtime, and QoE: The current evaluation focuses on system-level metrics such as latency, migration time, and resource utilization, and does not explicitly quantify service interruption, downtime, or user-perceived Quality of Experience (QoE). As discussed in Section 5.3.4, reductions in migration time (up to 54% compared to baseline methods) provide indirect evidence of shorter service unavailability windows, particularly under high-mobility scenarios. However, these metrics do not fully capture user experience. Future work will extend the framework to explicitly measure service interruption duration and downtime, and to incorporate QoE-aware reward components into the DRL formulation.

Centralized Control and Scalability: The framework assumes a centralized migration controller with global state visibility, which simplifies coordination but introduces scalability constraints and a potential single point of failure. As the number of edge servers increases, the signaling overhead per decision step grows linearly and may become a practical limitation in large-scale deployments. Future work will investigate distributed alternatives, including hierarchical control and multi-agent reinforcement learning (MARL), where edge servers operate as independent agents with local coordination. These approaches can reduce communication overhead and improve scalability and fault tolerance.

Comparison with External RL Frameworks: Several RL-based migration methods exist. However, direct quantitative comparison with external approaches (e.g., Peng et al. (2024b), Tsourdinis et al. (2023)) was not performed due to reproducibility barriers described in Section 2.3. Future work will explore standardized benchmarking scenarios to enable cross-framework comparison under unified experimental conditions.

7. Conclusion

The results demonstrate that the cache-aided RL algorithm achieves superior performance compared to other strategies, including the RL agent without cache awareness. Despite allocating similar or even fewer computational resources (CPU, RAM, and disk), the proposed RL agent consistently supports a higher number of MEC applications while maintaining lower latency and shorter migration times. This reflects not only improved resource efficiency but also more effective and intelligent service placement decisions. By incorporating cache awareness into both the state space and the reward function, the agent is empowered to make more informed and proactive migration decisions. This enhancement yields several important benefits for 5G networks. First, cache-awareness supports low-latency service delivery by ensuring content remains close to end users. Second, the adaptive migration behavior improves service continuity during user mobility, particularly in dynamic scenarios such as pedestrian and vehicular environments. Third, by optimizing cache utilization, the approach minimizes unnecessary data transfers and enhances the efficiency of edge resource usage. Finally, the RL-based framework exhibits scalability and generalizability, making it suitable for handling the increasing heterogeneity and service demands anticipated in future 5G and beyond networks.

Jeffrey Redondo: Conceptualization, Methodology, Software, Investigation, Data curation, Formal analysis, Validation, Writing – original draft, Writing – review & editing. **Kostas Ramantas**: Conceptualization, Methodology, Resources, Validation, Writing – review & editing. **Christos Verikouskis**: Resources, project management, Writing – review & editing.

8. Declaration of competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

9. Acknowledgements

This research was funded by the AC3 (Agile and Cognitive Cloud-edge Continuum management) project (Grant No. 101093129)

References

Addad, R.A., Dutra, D.L.C., Taleb, T., Flinck, H., 2022. Ai-based network-aware service function chain migration in 5g and beyond networks.

IEEE Transactions on Network and Service Management 19, 472–484. doi:10.1109/TNSM.2021.3074618.

Bellavista, P., Corradi, A., Foschini, L., Scotece, D., 2019. Differentiated service/data migration for edge services leveraging container characteristics. IEEE Access 7, 139746–139758. doi:10.1109/ACCESS.2019.2943848.

Elbatal, I., Maiwada, U.D., Danyaro, K.U., Sarlan, A.B., 2025. Dynamic handover optimization in 5g heterogeneous networks. Journal of Radiation Research and Applied Sciences 18, 101411. URL: https://www.sciencedirect.com/science/article/pii/S1687850725001232, doi:https://doi.org/10.1016/j.jrras.2025.101411.

Fan, Q., Ansari, N., 2019. On cost aware cloudlet placement for mobile edge computing. IEEE/CAA Journal of Automatica Sinica 6, 926–937.

Forde, A., Daniel, J., 2021. Pedestrian walking speed at un-signalized midblock crosswalk and its impact on urban street segment performance. Journal of Traffic and Transportation Engineering (English Edition) 8, 57–69. doi:https://doi.org/10.1016/j.jtte.2019.03.007.

He, Z., Li, L., Lin, Z., Dong, Y., Qin, J., Li, K., 2024. Joint optimization of service migration and resource allocation in mobile edge–cloud computing. Algorithms 17. doi:10.3390/a17080370.

Huang, H.S., Su, Y.S., 2023. A practical study of qoe on cloud gaming in 5g networks, in: 2023 International Wireless Communications and Mobile Computing (IWCMC), pp. 638–643. doi:10.1109/IWCMC58020.2023.10182439.

Hui, M., Chen, J., Zhou, Y., He, B., Wu, K., Yang, L., 2022. Server deployment and load balancing in stochastic mobile edge computing networks. IEEE Communications Letters 26, 1194–1198. doi:10.1109/LCOMM.2022.3151467.

Kaftantzis, N., Kogias, D.G., Patrikakis, C.Z., 2024. Exploring the impact of resource management strategies on simulated edge cloud performance: An experimental study. Network 4, 498–522. URL: https://www.mdpi.com/2673-8732/4/4/25, doi:10.3390/network4040025.

Kiran, B.R., Sobh, I., Talpaert, V., Mannion, P., Sallab, A.A.A., Yogamani, S., Pérez, P., 2022. Deep reinforcement learning for autonomous driving: A survey. IEEE Transactions on Intelligent Transportation Systems 23, 4909–4926. doi:10.1109/TITS.2021.3054625.

Lee, J., Lee, K., Yoo, A., Moon, C., 2020. Design and implementation of edge-fog-cloud system through hd map generation from lidar data of autonomous vehicles. Electronics 9. URL: https://www.mdpi.com/2079-9292/9/12/2084, doi:10.3390/electronics9122084.

Lee, L.N., Kim, M.J., Hwang, W.J., 2019. Potential of augmented reality and virtual reality technologies to promote wellbeing in older adults. Applied Sciences 9. URL: https://www.mdpi.com/2076-3417/9/17/3556, doi:10.3390/app9173556.

Liang, Z., Liu, Y., Lok, T.M., Huang, K., 2021. Multi-cell mobile edge computing: Joint service migration and resource allocation. IEEE Transactions on Wireless Communications 20, 5898–5912. doi:10.1109/TWC.2021.3070974.

Lin, P., Ning, Z., Zhang, Z., Liu, Y., Yu, F.R., Leung, V.C.M., 2024. Joint optimization of preference-aware caching and content migration in cost-efficient mobile edge networks. IEEE Transactions on Wireless Communications 23, 4918–4931. doi:10.1109/TWC.2023.3323464.

Malektaji, S., Ebrahimzadeh, A., Elbiaze, H., Glitho, R.H., Kianpisheh, S., 2021. Deep reinforcement learning-based content migration for edge content delivery networks with vehicular nodes. IEEE Transactions on Network and Service Management 18, 3415–3431. doi:10.1109/TNSM.2021.3086721.

Mbulwa, A.I., Tung Yew, H., Chekima, A., Dargham, J.A., 2024. Handover optimisation framework for next-generation wireless networks: 5g, 5g advanced and 6g, in: 2024 IEEE International Conference on Automatic Control and Intelligent Systems (I2CACIS), pp. 409–414. doi:10.1109/I2CACIS61270.2024.10649819.

Myyara, M., Darif, A., Farchane, A., 2024. Application of sarsa-based reinforcement learning approach for resource allocation in vehicular edge computing, in: 2024 10th International Conference on Optimization and Applications (ICOA), pp. 1–6. doi:10.1109/ICOA62581.2024.10754501.

Nardini, G., Sabella, D., Stea, G., Thakkar, P., Virdis, A., 2020. Simu5g–an omnet++ library for end-to-end performance evaluation of 5g networks. IEEE Access 8, 181176–181191. doi:10.1109/ACCESS.2020.3028550.

Pang, A.C., Chung, W.H., Chiu, T.C., Zhang, J., 2017. Latency-driven cooperative task computing in multi-user fog-radio access networks, in: Proc. IEEE 37th Int. Conf. Distrib. Comput. Syst. (ICDCS), Atlanta, GA, USA. pp. 615–624.

Peng, Y., Liu, H., Zhang, X., Li, F., 2024a. Cost-aware multi-resource service migration for mobile edge computing. IEEE Transactions on Network and Service Management 21, 1023–1037. doi:10.1109/TNSM.2024.3342710.

Peng, Y., Tang, X., Zhou, Y., Li, J., Qi, Y., Liu, L., Lin, H., 2024b. Computing and communication cost-aware service migration enabled by transfer reinforcement learning for dynamic vehicular edge computing networks. IEEE Transactions on Mobile Computing 23, 257–269. doi:10.1109/TMC.2022.3225239.

Ray, K., Banerjee, A., Narendra, N.C., 2024. Learning-based microservice placement and migration for multi-access edge computing. IEEE Transactions on Network and Service Management 21, 1969–1982. doi:10.1109/TNSM.2023.3344192.

Sabella, D., Li, A., Lee, H., Cominardi, L., Huang, Q., Pateromichelakis, E., Kashyap, V., Costa, C., Granelli, F., Featherstone, W., et al., 2023. Mec support towards edge native design.

Santos, J., Wang, C., Wauters, T., De Turck, F., 2023. Diktyo: Network-aware scheduling in container-based clouds. IEEE Transactions on Network and Service Management 20, 4461–4477. doi:10.1109/TNSM.2023.3271415.

Schettler, M., Buse, D.S., Zubow, A., Dressler, F., 2020. How to Train your ITS? Integrating Machine Learning with Vehicular Network Simulation, in: 12th IEEE Vehicular Networking Conference (VNC 2020), IEEE, Virtual Conference. doi:10.1109/VNC51378.2020.9318324.

Selesnic, S., Kodsi, S., 2016. Bicycling speeds: A literature review. Accident Reconstruction Journal 26, 12–15.

Singh, R., Sukapuram, R., Chakraborty, S., 2023. A survey of mobility-aware multi-access edge computing: Challenges, use cases and future directions. Ad Hoc Networks 140, 103044. URL: https://www.sciencedirect.com/science/article/pii/S1570870522002165, doi:https://doi.org/10.1016/j.adhoc.2022.103044.

Sonmez, C., Ozgovde, A., Ersoy, C., 2018. Edgecloudsim: An environment for performance evaluation of edge computing systems. Transactions on Emerging Telecommunications Technologies 29, e3493.

Sonmez, C., Ozgovde, A., Ersoy, C., 2019. Fuzzy workload orchestration for edge computing. IEEE Transactions on Network and Service Management 16, 769–782. doi:10.1109/TNSM.2019.2901346.

Tsourdinis, T., Makris, N., Fdida, S., Korakis, T., 2023. Drl-based service migration for mec cloud-native 5g and beyond networks, in: 2023 IEEE 9th International Conference on Network Softwarization (NetSoft), pp. 62–70. doi:10.1109/NetSoft57336.2023.10175417.

Wang, P., Ouyang, T., Liao, G., Gong, J., Yu, S., Chen, X., 2022. Edge intelligence in motion: Mobility-aware dynamic dnn inference service migration with downtime in mobile edge computing. Journal of Systems Architecture 130, 102664. doi:https://doi.org/10.1016/j.sysarc.2022.102664.

Wang, X., Wang, S., Liang, X., Zhao, D., Huang, J., Xu, X., Dai, B., Miao, Q., 2024. Deep reinforcement learning: A survey. IEEE Transactions on Neural Networks and Learning Systems 35, 5064–5078. doi:10.1109/TNNLS.2022.3207346.

Wu, Q., Zhao, Y., Fan, Q., Fan, P., Wang, J., Zhang, C., 2023. Mobility-aware cooperative caching in vehicular edge computing based on asynchronous federated and deep reinforcement learning. IEEE Journal of Selected Topics in Signal Processing 17, 66–81. doi:10.1109/JSTSP.2022.3221271.

Xiao, X., Ma, Y., Xia, Y., Zhou, M., Luo, X., Wang, X., Fu, X., Wei, W., Jiang, N., 2022. Novel workload-aware approach to mobile user reallocation in crowded mobile edge computing environment. IEEE Transactions on Intelligent Transportation Systems 23, 8846–8856. doi:10.1109/TITS.2021.3086827.

Xu, Y., He, Z., Li, K., 2024. Resource Allocation and Placement in Multi-access Edge Computing. Springer Nature Singapore, Singapore. pp. 39–62. doi:10.1007/978-981-97-2644-8_3.

Yang, S., Liu, J., Zhang, F., Li, F., Chen, X., Fu, X., 2022. Caching-enabled computation offloading in multi-region mec network via deep reinforcement learning. IEEE Internet of Things Journal 9, 21086–21098.

Yuan, Q., Li, J., Zhou, H., Lin, T., Luo, G., Shen, X., 2020. A joint service migration and mobility optimization approach for vehicular edge computing. IEEE Transactions on Vehicular Technology 69, 9041–9052. doi:10.1109/TVT.2020.2999617.

Zhang, D.G., Ni, C.H., Zhang, J., Zhang, T., Zhang, Z.H., 2023. New method of vehicle cooperative communication based on fuzzy logic and signaling game strategy. Future Generation Computer Systems 142, 131–149. doi:https://doi.org/10.1016/j.future.2022.12.039.

Zhang, F., Liu, G., Fu, X., Yahyapour, R., 2018a. A survey on virtual machine migration: Challenges, techniques, and open issues. IEEE Communications Surveys & Tutorials 20, 1206–1243.

Zhang, P., Zhou, M., Fortino, G., 2018b. Security and trust issues in fog computing: A survey. Future Generation Computer Systems 88, 16–27.

Jeffrey Redondo, (Member, IEEE) Senior Researcher at Iquadrat, and PhD. Candidate in Computer Science at Northumbria University, UK. He received the B.E.E. degree from La Universidad Del Norte, Barranquilla, Colombia, in 2011, and the M.S. degree from Tallinn University of Technology, Estonia, in 2019. He was a participant in the research group Cyber Security and Network Systems (CyberNets) at Northumbria University, UK. His research interest includes evaluating the benefits of using machine learning to improve the quality of service in wireless technologies for autonomous vehicles.

Kostas Ramantas, (Member, IEEE), is a senior researcher at Iquadrat, 08006 Barcelona, Spain. He received a Diploma of Computer Engineering, M.Sc. degree in computer science, and Ph.D. degree from the University of Patras, Greece. He has been the recipient of two national scholarships and has published more than 35 journal and conference papers.

Christos Verikoukis, (Senior Member, IEEE), Received his Ph.D. degree in broadband indoor wireless communications from the Signal Theory and Communications Department, Technical University of Catalonia, Barcelona, Spain, in 2000. Since 2004, he has been a senior research associate in the Centre Tecnològic de Telecomunicacions de Catalunya. He has participated or is currently participating in several European- (ACTS, IST FP5 and FP6, Marie Curie, eTEN, and INTERREG) and national (in Spain and in Greece)-funded projects. His research interests include ARQ schemes, MAC protocols, RRM algorithms, cross-layer techniques, and cooperative communications in wireless communications systems, such as WLAN, WiMax, and wireless sensor networks. He is a Senior Member of the IEEE.

A. Migration Algorithm Knapsack

This algorithm is the one used for the migration-aware Greedy knapsack.

Algorithm 3 Migration-Aware Greedy Knapsack

Input:
    - Application demand vector d
    - UE position $\mathbf{x}$
    - MEC server set $\mathcal{E}$ with free capacities f_e
    - System parameters α, β, λ
Output:
    - Selected host server $e^\star$
$currentHost \leftarrow resolveCurrentHost(app)$
$L_{\text{before}} \leftarrow estimateLatency(currentHost, \mathbf{x})$
$e^\star \leftarrow \emptyset$
$s^\star \leftarrow -\infty$
for all $e \in \mathcal{E}$ **do**
  if not isAllocable(e, d) **then**
    continue
  end if
  $L_{\text{after}} \leftarrow estimateLatency(e, \mathbf{x})$
  if $currentHost = null$ **then**
    $gain \leftarrow 1/L_{\text{after}}$
  else
    $gain \leftarrow \max(0, L_{\text{before}} - L_{\text{after}})$
  end if
  $cost \leftarrow estimateMigrationCost(currentHost, e, d)$
  $pressure \leftarrow resourcePressure(e, d)$
  $s \leftarrow \frac{\alpha \cdot gain - \beta \cdot cost}{\max(pressure, \varepsilon)}$
  if $s > s^\star$ **then**
    $s^\star \leftarrow s$
    $e^\star \leftarrow e$
  end if
end for
return $e^\star$