Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

A Latency-and-Affinity-Based Service Migration in Mobile Edge Computing

Xialin Liu

liuxialin@xupt.edu.cn

Xi'an University of Posts and Telecommunications

Shuqin Liu

Xi'an University of Posts and Telecommunications

Xinfeng Shu

Xi'an University of Posts and Telecommunications

Xinrui Nie

Xi'an University of Posts and Telecommunications

Research Article

Keywords: MEC, Service Migration, Multi-Component Service, DRL, Affinity-Aware Optimization

Posted Date: November 2nd, 2026

DOI: https://doi.org/10.21203/rs.3.rs-11403291/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: No competing interests reported.

A Latency-and-Affinity-Based Service Migration in Mobile Edge Computing

Xialin Liu[1,2,*], Shuqin Liu[1,2], Xinfeng Shu[1,2], Xinrui Nie[3]

[1] School of Computer Science and Technology, Xi'an University of Posts and Telecommunications, Xi'an Shaanxi, China

[2] Shaanxi Key Laboratory of Network Data Analysis and Intelligent Processing, Xi'an University of Posts and Telecommunications, Xi'an Shaanxi, China

3 School of Communication and Information Engineering, Xi'an University of Posts and Telecommunications, Xi'an Shaanxi, China

Abstract

The advancement of 5G, smart user equipment, and the Internet of Things has enabled latency-sensitive and computation-intensive applications. However, existing research often treats services as monolithic entities or overlooks component affinity—the intensity of internal communication. Migrating interdependent components independently leads to severe performance degradation due to inflated internal communication costs. To address this gap, we propose LASM, a novel Latency- and Affinity-aware Service Migration framework for multi-component services. The key novelty lies in a two-stage decomposition strategy that separates latency optimization (User-Facing Component, UFC migration) from affinity optimization (Data-Plane Component, DPC placement), avoiding the arbitrary weighting required by single-stage joint optimization, where latency and affinity have incommensurable units. The first stage introduces a cooperative multi-agent Deep Reinforcement Learning (DRL) approach to minimize end-to-end latency under dynamic user mobility. The second stage proposes a novel affinity-aware DPC placement that clusters component pairs by their communication intensity and network distance, then performs UFC-anchored greedy placement to ensure high-affinity components are deployed on proximate edge servers. We evaluate LASM on the EdgeCloudSim platform using four synthetic traffic patterns (CBR, ON-OFF, Poisson, and Self-Similar) with 10,000 training and 3,000 testing samples, averaged over 10 independent runs. Extensive simulations demonstrate that LASM significantly outperforms five baseline strategies, including the state-of-

the-art dependency-aware method (DM). Specifically, LASM achieves up to 11% reduction in average service time and 33% reduction in total system cost compared to the strongest DRL baseline (DRLM) that ignores affinity. These results validate that explicit affinity modeling, coordinated with latency-driven migration, is essential for efficient multi-component service migration in MEC.

Keywords: MEC; Service Migration; Multi-Component Service; DRL; Affinity-Aware Optimization

1 Introduction

With the rapid growth of 5G and IoT, latency-sensitive and computation-intensive applications (e.g., AR/VR and face recognition) are widely deployed. However, user equipment (UE) has limited computational power and battery capacity. While cloud computing can offload tasks, the physical distance to cloud servers introduces intolerable delays.

Mobile Edge Computing (MEC) addresses this problem by deploying Edge Servers (ESs) at the network edge, reducing latency and conserving energy. As more services are hosted at the edge, user mobility becomes critical. When a user moves, the associated service must be seamlessly migrated to maintain Quality of Service (QoS). Existing service migration strategies can be categorized by granularity: monolithic migration [1-17,20-24,29,43,44] relocates entire services, while component-based migration [24,26-28,42,45,46] moves individual components independently. Methodologically, these strategies include Markov Decision Process (MDP)-based [1-5], Machine Learning (ML)[10,43,45], Lyapunov optimization [6-9], and prediction-based approaches [13-15,25,30,44,46].

While several works consider inter-component dependencies (e.g., traffic-aware VM placement, VNF/SFC placement, and dependency-aware microservice placement), to the best of our knowledge, none simultaneously (1) distinguishes component roles (UFC vs. DPC), (2) models affinity as a joint function of traffic and network distance, and (3) addresses user-mobility-triggered migration in real time. Fig.1 illustrates a virtual Set-Top-Box (vSTB) service [25] with interdependent components. Existing

component-level migration strategies for VNF/SFC are typically triggered by network failures rather than user mobility, and most lack a holistic framework that coordinates latency-driven user-facing migration with affinity-aware backend placement. Our work directly addresses this gap. We propose LASM, a framework specifically designed for multi-component services. LASM explicitly models component affinity and decomposes the problem into two synergistic stages: (1) latency-optimized migration of UFCs via a multi-agent DRL approach, and (2) affinity-driven placement of DPCs via a novel clustering-and-placement heuristic. To make the problem tractable, we adopt two modeling assumptions—no co-location of same-service components and UFC placement within the user's coverage set—which are justified in Section 3. Unlike existing two-stage schemes that split placement and migration temporally, our decomposition is structural: it separates components by their role in the service (user-facing vs. data-plane) and assigns each stage an optimization criterion aligned with its role (latency vs. affinity). This role-based decomposition avoids the unit-incompatibility problem of joint scalarization and enables the use of DRL for the latency-critical stage and a lightweight heuristic for the affinity-critical stage. Building on this framework, we contribute:

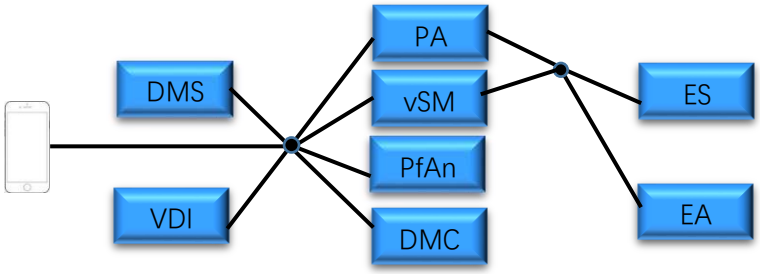


Fig.1. vSTB service architecture example.

- A role-based two-stage decomposition that separates latency optimization (UFC migration) from affinity optimization (DPC placement). This decomposition is not merely sequential: it is structural, separating components by their role in the service. We prove that the resulting lexicographic solution is Pareto-optimal (Theorem 1) and bound the suboptimality of the secondary objective (Theorem 2).

- A tailored multi-agent DRL algorithm for the UFC migration stage, with custom state/action/reward design addressing the unique characteristic of user mobility.
- A lightweight affinity-aware DPC placement for the DPC placement stage, with linear complexity suitable for real-time execution.
- Comprehensive simulation-based validation. LASM outperforms five baselines on EdgeCloudSim, reducing service time by up to 11% and total cost by up to 33%, with statistical significance confirmed.

The remainder of this paper is organized as follows. Section 2 reviews related work. Section 3 presents the system model and problem formulation. Section 4 details our proposed LASM solution. Section 5 presents the experiments and results. Finally, Section 6 concludes the paper and outlines future research directions.

2 Related work

Service migration is a core functionality in MEC, typically triggered by user mobility, server overload, or network dynamics. However, migration itself incurs substantial costs in terms of latency, energy, and resource consumption. Consequently, the fundamental research problem is to design migration strategies that optimally orchestrate computation and communication resources under uncertainty, effectively balancing service quality against migration overhead [24]. This section systematically surveys existing solutions, categorizing them according to their underlying methodology and optimization focus. We critically analyze these approaches and identify a critical research gap concerning the coordinated migration of modular, multi-component services.

Service migration thus requires real-time decision-making in a highly uncertain and stochastic setting. Numerous studies have modeled the service migration problem as an MDP and employed RL or DRL techniques. [1] investigated service migration in a MEC-enabled vehicular network, aiming to minimize the total service latency and migration cost. They modeled the problem as a multi-agent MDP and solved it using Deep Q-Learning (DQL), enabling proactive migration while ensuring service continuity under high mobility. [2] employed an MDP framework and derived a

modified policy iteration algorithm to obtain optimal actions based on the distance between user and its serving ES. [3] introduced a migration-zone-based solution with a comprehensive cost model, applying Value Iteration (VI) and Policy Iteration (PI) to obtain optimal policies. [4] tackled container migration in fog computing as a large-scale MDP, designing a DRL algorithm to reduce state space and accelerate decision-making. [5] utilized an RL-based model to consider long-term objectives, making joint service migration and communication decisions more efficient. While MDP/DRL approaches effectively handle complex dynamics and long-term optimization, they often grapple with challenges such as state-space explosion and high training costs.

To circumvent the need for complete state information required by MDP solutions, several studies have adopted Lyapunov optimization. This technique is particularly adept at handling long-term constraints and enabling online decision-making without prior knowledge of system dynamics. [6] leveraged Lyapunov optimization to design an online algorithm for service migration. [7] transformed the optimization problem into a queue stability control problem using Lyapunov theory to derive migration decisions. [8] proposed a two-stage algorithm for Parked Vehicle Edge Computing (PVEC), decoupling the problem into ES-to-ES migration and parked vehicle selection, and applying Lyapunov optimization to the former. [9] addressed Service Function Chain (SFC) migration under a long-term cost budget, using Lyapunov optimization to decompose the problem into real-time sub-problems solved by a Markov approximation heuristic. A key trade-off is that while Lyapunov optimization excels in managing long-term constraints and requires no prior state information, it may offer less precision for instantaneous, fine-grained optimization compared to DRL.

Beyond standard MDP/DRL, other machine learning techniques have been explored. [10] modeled the problem as a Partially Observable MDP (POMDP) and designed an encoder network combining Long Short-Term Memory (LSTM) with an embedding matrix to extract hidden information. [11] proposed SMDQN, a DRL-based algorithm designed to reduce the MDP state space for fast dynamic decisions. Given the inherent

uncertainty in user mobility, prediction-based methods have gained traction to enable proactive migration. [14] introduced a method using a migration radius and energy consumption model, categorizing user regions to guide migration decisions. The Follow-Me Fog (FMF) framework [16] pre-migrates computation jobs before wireless handover to reduce latency. [17] leveraged the features from a user's camera in AR application to predict target cloudlet. [18] deployed service based on predicted future costs, including data transmission cost, processing cost, and service migration cost. An approach for predicting user trajectories based on map data has been proposed in [19], and service will be migrated to ES based on prediction. While proactive, these methods are highly dependent on prediction accuracy, where errors can lead to ineffective migrations.

For complex combinatorial problem, heuristic and meta-heuristic algorithms offer computationally efficient solutions. [21] used transmission power control to divide user activity areas, minimizing handoffs and server energy consumption via Particle Swarm Optimization (PSO). [22] designed the Particle Swarm-based Service Migration (PSSM) scheme, integrating queuing delay prediction and resource allocation. These methods are practical and easy to implement but do not guarantee optimality, and their performance can be sensitive to parameter tuning.

Significant research tailor migration strategies to specific contexts like Vehicular EC (VEC) and Network Function Virtualization (NFV). In VEC, [12] utilized idle resources from Parked Vehicles (PVs) and RSUs as ESs. [13] employed a Hidden Markov Model (HMM) to optimize service migration latency and the interference among vehicles. [15] proposed a multi-agent DRL (MADRL) algorithm to jointly optimize migration and routing in VEC. The work in [20] introduced an Edge Cognitive Computing (ECC) architecture that merges edge computing with cognitive intelligence, proposing an RL-based mechanism to trade off cost minimization and Quality of Experience (QoE) improvement. [23] proposed a context-aware task migration scheme for Fiber-Wireless (FiWi) infrastructure, decomposing tasks to migrate cognitive subtasks across human users, cloudlets, and collaborative robots to minimize response

time. For NFV-based environments, studies increasingly address the migration of modular services, such as Virtual Network Functions (VNFs) and Service Function Chains (SFCs). [26] proposed a policy clustering Virtual Objects (VOs) based on user proximity and inter-VO affinity. [27] used context-aware grouping and graph partitioning for VNF resource allocation. [28] proposed a concurrent VNF migration method using a hybrid genetic algorithm. [29] employed a Graph Neural Network (GNN) to extract network features and a DRL agent for VNF remapping upon failures. [30-33] further investigated VNF/SFC migration focusing on energy consumption, QoS-energy trade-offs, and migration timing. While the above efforts advance modular service migration and explore novel paradigms, including the dependency-aware work [36], they exhibit common limitations: they are often triggered by network failures rather than user mobility, and most lack a holistic framework for coordinated, multi-component migration that simultaneously optimizes for latency and component affinity in general MEC scenarios.

Recent works have further advanced migration strategies in emerging scenarios. [42] proposed a VNF fusion and soft migration method for 6G networks, enabling efficient slice reconfiguration. [43] introduced a blockchain-enabled elastic task offloading and migration scheme for edge computing. [44] developed MAPSM, a mobility-aware proactive service migration framework for the Consumer Internet of Vehicles, which predicts user mobility to enable anticipatory migration. [45] designed an intelligent resource management framework for 6G-enabled IoT networks. [46] studied collaborative and latency-aware microservice migration in MEC, demonstrating the importance of coordinated component placement. Among these, [36] is particularly relevant as it proposes a trajectory prediction-based container migration strategy that explicitly considers dependencies among containers. This work represents a significant step toward dependency-aware service migration in MEC. However, a key distinction exists between their dependency model and our affinity formulation. [36] considers only the communication between containers, which indirectly affects communication delay. In contrast, our affinity model explicitly incorporates both communication traffic

and the network distance between hosting ESs. This dual consideration captures a more nuanced reality: even high-traffic component pairs can tolerate larger separation if placed on well-provisioned, low-latency links, while low-traffic pairs may still require proximity if their communication is delay-sensitive. Collectively, these recent works confirm the growing interest in mobility-aware and component-coordinated migration, but they either focus on single component services or lack a unified framework for jointly optimizing latency and affinity. Our LASM framework fills this gap.

A comparative summary of the afore-mentioned approaches is presented in Table 1, highlighting their respective objectives, methods, advantages, and limitations.

Table 1 Comparison of existing service migration approaches.

Ref. ID	Optimization Objective	Method Used	Advantages	Limitations
[1]	Minimize total service latency and migration cost	Multi-agent MDP, DQL	Ensures service continuity under high mobility; enables proactive migration.	High complexity in multi-agent systems; significant training and coordination overhead.
[2]	Optimal action based on user-ES distance	MDP, Modified PI	Clear theoretical foundation; yields deterministic optimal policy.	Relies on precise distance/state information; may lack adaptability in highly dynamic environments.
[3]	Maximize overall expected return	Migration-zone based MDP, VI & PI	Comprehensive cost model; simplifies problem scale via zoning.	Static zoning may not adapt to highly dynamic user mobility patterns.
[4]	Fast decision-making, reduce large MDP state	Large-scale MDP, DRL	Designed for large-scale problems; fast decision speed.	DRL training can be unstable; requires substantial data and computational

	space			resources.
[5]	Efficient service migration & communication decisions (long-term goal)	RL	Considers long-term optimization objectives for forward-looking decisions.	Does not explicitly address dependencies among multi-component services.
[6]-[9]	Long-term avg. latency, energy consumption constraint, cost budget balance.	Lyapunov Optimization	Requires no prior state information; suitable for online decision-making; excels at handling long-term constraints; decomposes complex problems into per-slot sub-problems.	Less precise for instantaneous optimization; queue stability assumptions may not hold in all scenarios.
[10]	Decision-making under partial observability	Partially Observable MDP (POMDP), LSTM Encoder Network	Effectively handles environmental uncertainty and partial observability.	Model complexity is high; difficult to train; heavily reliant on sequential data.
[11]	Fast decision-making considering total delay & migration cost	DRL (SMDQN)	Reduces state space via DRL for fast dynamic decisions.	Shares general DRL drawbacks: high training cost, poor policy interpretability.
[15]-[19]	Proactively reduce migration latency via prediction	Prediction-based methods (user trajectory, application features, map data)	Strong proactivity; can significantly reduce delay caused by decision lag.	Performance highly dependent on prediction accuracy; prediction errors may lead to ineffective migrations or resource waste.

[21], [22]	Minimize wireless handoffs, server energy consumption, queuing delay.	Heuristic/Meta-heuristic algorithms (e.g., PSO)	Computationally efficient, easy to implement, suitable for complex combinatorial problems.	No guarantee of optimality; parameter tuning is experience-dependent; generalizability may be limited.
[12]-[14]	Minimize latency/interference in VEC; maximize comprehensive utility of comm.-comp.-routing	Customized solutions for Vehicular Edge Computing (VEC) (e.g., HMM, Multi-Agent DRL)	Deeply optimized for specific scenarios (high mobility), achieving good performance.	Solutions are highly specialized, with limited generalizability; not directly applicable to general MEC scenarios.
[20], [23]	Trade-off cost vs. QoE; minimize cross-platform task response time	Mechanisms for specific architectures (Cognitive Computing, FiWi integrated networks)	Explores integration of edge computing with emerging paradigms (cognition, robotics), offering inspiration.	Also suffers from scenario limitations; methodological generalizability is weak.
[26]-[33]	Support user mobility, minimize end-to-end latency, optimize energy consumption, handle network	Migration strategies for modular services/VNFs/SFCs (clustering, graph partitioning, DRL, heuristics)	Begin to address service modularity and affinity (e.g., [31]), laying groundwork for multi-component migration.	Most works are triggered by network failures rather than user mobility; insufficient investigation into coordinated migration sequencing and glob

	failures			al cost balancing.
[36]	Reduce service latency & load balance with dependency awareness	RNN trajectory prediction; Hung-er Games Search (HGS)	Considers container traffic dependencies; validated on real datasets	Models traffic only, not distance; relies on prediction; single-stage; no online adaptation
[42]	Efficient slice reconfiguration in 6G networks	VNF fusion and soft migration	Reduces reconfiguration overhead; enables flexible network slicing	Focuses on network-level VNFs; not applicable to general MEC service migration
[43]	Elastic task offloading and migration in edge computing	Blockchain-enabled optimization	Enhances security and trust; enables elastic resource allocation	Blockchain consensus introduces additional latency and energy overhead
[44]	Proactive service migration for Internet of Vehicles	Mobility prediction and proactive migration (MAPSM)	Anticipates user movement; reduces migration latency	Highly dependent on prediction accuracy; designed for vehicular scenarios
[45]	Intelligent resource management for 6G-enabled IoT	Softwarized resource allocation with personalized intentions	Handles heterogeneous service intentions; enables intelligent orchestration	Scenario-specific to 6G-IoT; lacks component-level affinity modeling
[46]	Collaborative and latency-aware	Multi-component coordination with	Considers microservice collaboration; reduces end-to-end latency	Does not jointly optimize user-facing latency and

microservice migration	latency awareness	component affinity

Existing research has made substantial progress in addressing service migration in MEC from various angles. However, the treatment of component dependencies remains limited in the following specific respects:(1) Dependency modeling granularity: Several works consider inter-component traffic as a proxy for dependency. For example, [36] models container dependencies as communication traffic, and [26] clusters virtual objects by inter-object affinity. However, these works model only traffic volume, not the network distance between hosting ESs. Our affinity metric explicitly captures both dimensions. (2) Component role differentiation: VNF/SFC placement works [26-33] typically treat all components uniformly. In contrast, our model distinguishes UFCs, which must be co-located with the user, from DPCs, which can be placed flexibly. This distinction enables independent optimization of user-perceived latency and internal communication overhead. (3) Triggering conditions: Most VNF/SFC migration works [29-33] are triggered by network failures or topology changes. In contrast, our work addresses migration triggered by user mobility, which is the dominant factor in MEC service continuity. (4) Optimization architecture: Existing approaches either optimize all components jointly in a single stage [36] or decompose the problem without explicitly accounting for inter-component affinity [26-28]. Our LASM framework adopts a two-stage decomposition that separates latency optimization (UFCs) from affinity optimization (DPCs), with the former serving as anchors for the latter. While individual works may address one aspect—such as traffic-based dependency [36] or component coordination [46]—none provides a unified framework that simultaneously distinguishes component roles, models affinity as a joint function of traffic and distance, and addresses user-mobility-triggered migration in real time. This is the gap that our work fills.

3 Problem formulation

This section formalizes the service migration problem in MEC. We begin by introducing system model, followed by the detailed models for different service

component. Subsequently, we decompose the original problem into two sequential sub-problems based on component characteristic. For clarity, all key notations are summarized in Table 2.

Table 2 Summary of key notations.

Category	Symbol	Definition
System Sets & Indices	I, m	Set and total number of UEs.
	J, n	Set and total number of ESs.
	L, r	Set and total number of Service Components.
	L_U,L_D	Subsets of UFCs and DPCs.
	$U_k \subseteq J$	coverage set of UE k.
	T	Total number of time slots.
	k	Index for a UE.
	i, j, j^-	Indices for ESs.
	l	Index for a service component.
	t	Index for a time slot.
Component & ES	D_l, C_l, L_l	Input data size, required CPU cycles, and max latency for l.
	$Size_{kl}^{up}, Size_{kl}^{dw}$	Upload and download data size for component l of k's service.
	c_{kl}^t	CPU cycles required by component l of UE k's service at slot t.
	f_i	Computing capacity of ES i.
	N_i^t	Number of services on ES i at time slot t.
	$H(p, q)$	Network distance (e.g., hop count) between ES p and ES q.
Decision Variables	$x_{ikl}^t \in \{0,1\}$	1 if ES i is the access ES for UFC l of UE k at slot t.
	$y_{jkl}^t \in \{0,1\}$	1 if ES j is the serving ES for component l of UE k at slot t.
	$\omega_{ikl}^t \in \{0,1\}$	1 if DPC l of UE k is placed on ES i at slot t.
	$z_{j^-jkl}^t$	1 if a migration of component l from ES j^- to j occurs at slot t.
	b'_{klt}	Migration decision for component l of UE k at slot t.
	$\boldsymbol{b'_{kt}}$	Service migration decision vector for UE k at slot t.
	$\pi(k, i, t)$	Function returning the ES hosting component i of k at time t.
Affinity model	A	Affinity matrix.
	A_{ij}	Affinity between components i and j.
	Tra_{ij}	Traffic volume between components i and j per time slot.
	Dis_{ij}	Hop count between the ESs hosting components i and j.
	δ	Small constant to prevent division by zero.
Cost models	γ_{ki}^t	Signal-to-noise ratio between UE k and ES i at slot t.
	g_{ki}^t	Channel gain between UE k and ES i at slot t.
	d^t	Euclidean distance between UE and access ES at time slot t.
	R_{ki}^t	Data rate between UE k and ES i at slot t.
	CD_{ikl}^t	Total communication delay for component l.
	C_{ikl}^t	Computation delay of component l on ES i for UE k at slot t.
	M_{jjkl}^t	Migration delay for moving component l from ES j^- to j.
	$Size_{kl}$	Data size of component l of the requested service of UE k.
	d_{j^-j}	Network distance between j^- and j.
	Φ^t	Total affinity-aware communication benefit at slot t.
	ω_{ikl}^t	Placement variable, which equals 1 if DPC l is placed on ES i.
DRL model	s_j^t	Local state of ES agent j.
	x^{t-1}	Global UFC placement vector from previous slot.
	$\mathcal{S}$	State space.
	$\mathcal{A}$	Action space.
	$\mathcal{P}(s_{t+1}\|s_t, a_t)$	State transition probability.
	$\mathcal{R}(s_t, a_t)$	Received reward after performing the action a_t on state s_t.
	$(\mathcal{D}, \mathcal{C}, \mathcal{L})$	UFC requirements.
	Loc_k^t	Current location of user k.
	a_j^t, a_{jl}^t	Action vector of agent j; $a_{jl}^t = 1$ proposes to host UFC l.
	r_j^t	Reward for agent j at slot t.
	$R_{penalty}$	Large negative penalty for infeasible actions.
	$Q(s, a; \omega)$	Q-function approximated by DQN with parameter ω.
	$\omega, \hat{\omega}$	Parameters of MQN and TQN.

	$\gamma, \alpha, \varepsilon$	Discount factor, learning rate, and exploration rate.
Clustering Model	K	Number of affinity clusters.
	C_t	The t −th cluster, a set of component pairs $\{(i,j)\}$.
	$center[t]$	Center of cluster t.

3.1 System model

3.1.1 Network and service model

We consider an MEC system consisting of multiple users and multiple ESs, as illustrated in Fig.2. Let $I = \{1,2,\dots,m\}$ denote the set of UEs, which includes IoT devices such as smartphones and vehicles. Each UE k requests a service. The set of ESs is denoted by $J = \{1,2,\dots,n\}$. These servers are distributed at network edge, providing computational services to UE within their radio coverage. Each ES i, $i \in J$ is characterized by a tuple (p_i, f_i, u_i, l_i), where p_i is the maximum power of the communication between UE and ES, f_i is the computing capacity, u_i is coverage radius, and l_i is geographical location. We define the set of ESs that can directly connect to UE k as its coverage set, denoted as $U_k \subseteq J$. The UE's access ES is selected from U_k and determines the last-hop wireless latency. The ES that runs the UE's service but is not directly connected to the user is defined as the UE's serving ES.

A service can be defined by a set of $L = \{1,2,\dots,r\}$ components. Let D_l, C_l, L_l denote input data size, required total CPU cycles, and the maximum latency allowed of service component $l, l \in L$, respectively. We make two key modeling assumptions: (1) each service component is encapsulated within a dedicated virtual machine or container; (2) the components belonging to the same service are not allowed to be placed on the same ES. The second assumption is a modeling choice rather than a hard requirement of real-world MEC systems. We adopt it primarily for fault domain isolation: distributing components across multiple ESs reduces the probability that a single ES failure causes simultaneous failure of all components of the same service. We note that this bounds the blast radius of a single failure rather than guaranteeing service availability. We acknowledge that containerization with proper resource allocation can mitigate resource contention, and that replication can provide fault tolerance; our assumption represents an alternative design point that avoids the additional resource cost of replication in resource-constrained edge environments. We further note that this

constraint can be relaxed when resource capacity is abundant, and that a systematic comparison against co-location-enabled designs is left as future work.

An SDN controller, located in the cloud layer, orchestrates the system. It collects user request, makes optimal service placement and migration decision, and returns result to user via ES. The total operation time is discretized into T time slots. At each slot t, service request is generated randomly and sent to connected ES, triggering the SDN controller's decision-making process. We categorize the service components into two disjoint subsets:

- UFCs, denoted as $L_U \subset L$. These components directly interact with the end-user's UE, receiving input and sending output.
- DPCs, denoted as $L_D \subset L$. These components do not interact with the user directly, but communicate intensively with other components (both UFCs and DPCs) to complete the service logic.

As illustrated in Fig. 2, our MEC system comprises a three-tier architecture. User tier consists of multiple UEs (e.g., smartphone, vehicle) that generate service requests. Edge tier is composed of distributed ESs, which provide computational resources in proximity to user. Cloud tier hosts a centralized SDN controller that orchestrates service placement and migration decision across the edge infrastructure. UEs connect to their associated ESs via wireless links, while ESs interconnect through high-speed backhaul networks.

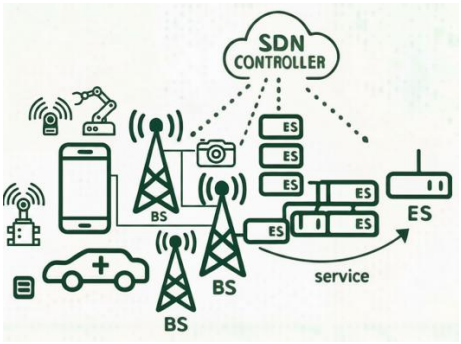


Fig.2. Illustration of system model.

3.1.2 Affinity definition

We quantify the interdependence between service components using an affinity

matrix $A \in \mathbb{R}^{r\times r}$. The element A_{ij} represents the affinity between component i and component j, defined as:

$$A_{ij} = \frac{Tra_{ij}}{Dis_{ij}+\delta} \quad (1)$$

where Tra_{ij} denotes the average traffic volume (in Mbps) between i and j per time slot, and Dis_{ij} is the network distance (e.g., hop count) between the ESs hosting them. δ is a small constant to prevent division by zero. This formulation captures the intuitive notion that affinity increases with higher communication traffic (Tra_{ij}) and decreases with larger network separation (Dis_{ij}). Consequently, a higher A_{ij} value signifies a stronger dependency, and thus greater benefit from placing components i and j on proximate ESs. The diagonal elements A_{ii} are conceptually set to infinity, indicating that a component is trivially co-located with itself. Note that the affinity matrix is dynamic, evolving with changing traffic patterns and network conditions.

3.1.3 Service migration and decision variables

A fundamental design choice in our model is that each UFC must be placed on one of the ESs within the user's coverage set U_k, which then becomes the user's access ES. This constraint is driven by the need to minimize the last-hop wireless latency between the user and the service front-end. Consequently, optimizing UFC migration involves selecting the optimal access ES for the user from U_k, considering the trade-off among communication delay, computation delay, and migration cost across all candidate ESs. In contrast, DPCs are not bound by this constraint and can be placed on any ES with sufficient resources, guided primarily by their affinity with other components to reduce internal communication overhead. This inherent difference in placement constraints naturally motivates a two-stage decomposition of the overall problem: first, determining the optimal placement for UFCs to ensure low user-perceived latency; second, given the UFC locations, deciding the optimal placement for DPCs to minimize affinity-aware communication costs.

We use the variable x_{ikl}^t to indicate whether ES i is the access ES of component l of the service requested by UE k at time slot t, e.g., $x_{ikl}^t == 1$ denotes true. We use another variable y_{jkl}^t to indicate whether ES j is the serving ES of component l of the

service requested by UE k at time slot t, e.g., $y_{jkl}^t == 1$ denotes true. We use z_{jjkl}^t to indicate whether a migration happens from access ES, $z_{jjkl}^t = x_{jkl}^{t-1} x_{jkl}^t$. We use b'_{klt} to indicate the migration decision of the component l of the service requested by UE k at time slot t and use a vector $\boldsymbol{b'_{kt}} = [b'_{klt}]$ to indicate the service migration decision of UE k.

A migration triggering event is defined by a change of UE's access ES. This event triggers the re-evaluation of the placement for all components of user's service. Our migration strategy, LASM, makes decisions for the two types of components differently:

- For a UFC $l \in L_U$, the decision is *where to place it* (e.g., which ES i to set $x_{ikl}^t == 1$) to minimize the end-to-end latency.
- For a DPC $l \in L_D$, the decision is *how to co-locate* it with relevant UFCs based on their affinities to minimize internal communication costs.

3.2 Cost and affinity benefit models

3.2.1 Cost model for UFCs

- *Communication delay*

To maintain QoS, service should be migrated to the ESs adjacent to user as user moves. After migrating service, communication delay is caused by the transmission between user and ES. Communication delay consists of two parts: the uplink delay between UE and access ES, and the downlink delay between access ES and UE. Note that the communication delay between access ES and serving ES has not been taken into account, as the migration of the component on the serving ES, namely DPC, will be optimized based on affinity (cost model provided in section 3.2.2). Uplink delay is determined both by wireless environment and by the data size of service. The channel power gain between UE k, $k \in I$ and access ES i, $i \in J$ at time-slot t is denoted by g_{ki}^t, incorporates both large-scale path loss and small-scale fading. It is modeled as:

$$g_{ki}^t = G_0 (\frac{d_0}{d^t})^\gamma \quad (2)$$

where G_0 is a constant, d^t is the Euclidean distance between UE and access ES at time slot t, d_0 is the antenna far-field test distance of the base station to which ES is connected, and γ is the path-loss exponent.

The Signal-to-Interference-plus-Noise Ratio (SINR) at the ES is then:

$$\gamma_{ki}^t = \frac{p_i g_{ki}^t}{\sigma^2} \tag{3}$$

where p_i is the transmit power of ES i, and σ^2 is noise power.

Assuming Gaussian signaling and interference-free transmission, the achievable data rate on this link is given by the Shannon-Hartley theorem:

$$R_{ki}^t = Blog_2(1 + \gamma_{ki}^t) \tag{4}$$

where B is channel bandwidth. Consequently, the uplink delay between ES i and UE k at time-slot t is given by:

$$UD_{ikl}^t = \frac{Size_{kl}^{up}}{R_{ki}^t} \tag{5}$$

where $Size_{kl}^{up}$ is the upload data size of component l of the requested service of UE k.

The downlink delay between ES i and UE k at time-slot t is given by:

$$DD_{ikl}^t = \frac{Size_{kl}^{dw}}{R_{ki}^t} \tag{6}$$

where $Size_{kl}^{dw}$ is the download data size of component l of the requested service of UE k.

Overall, the total communication delay of component l of the requested service of the UE k at time slot t can be given by:

$$CD_{ikl}^t = UD_{ikl}^t + DD_{ikl}^t \tag{7}$$

- *Computation delay*

When a service component arrives on ES i, some other service components may be queuing up for execution. We assume that the service component is executed last in the ES. Therefore, the computing delay of component l of the service of user k on ES i at time-slot t is given as follows:

$$C_{ikl}^t = \frac{c_{kl}^t N_i^t}{f_i} \tag{8}$$

where c_{kl}^t denotes the CPU cycles required by component l of the service of UE k,

and N_i^t denotes the number of services on ES i at time-slot t. The computing capacity of ES i is given by f_i.

- *Migration delay*

Service migration delay is defined as the transmission time of the service from original ES to target ES, which mainly depends on the hop distance along the communication path from original ES to target ES and the data size of the requested service. Hence, the migration delay of component l of the requested service of UE k from ES $\bar{\jmath}$ to ES j at time-slot t is given as follows:

$$M_{\bar{\jmath}jkl}^t = \frac{Size_{kl}}{\eta_{\bar{\jmath}j}^t} + 2\lambda_m d_{\bar{\jmath}j} \quad (9)$$

where $\eta_{\bar{\jmath}j}^t$ is the bandwidth of the link between $\bar{\jmath}$ and j at time slot t. $Size_{kl}$ is the data size of component l of the requested service of UE k. λ_m is a coefficient of the migration delay, and $d_{\bar{\jmath}j}$ is the hop count between $\bar{\jmath}$ and j.

3.2.2 Affinity benefit model for DPCs

For DPCs, we aim to avoid communication over unnecessarily long distances by selecting an optimal migration strategy. We assume that the network traffic between service component i and j in a time slot is fixed. The network distance between component i and j at time slot t is a function of their placement. Let $\pi(k,i,t)$ denote the ES hosting component i of user k at time t. We define the instantaneous distance as $Dis_{ij}^t = H\big(\pi(k,i,t), \pi(k,j,t)\big)$, where $H(p,q)$ is the known network distance between ES p and q. For DPCs, we introduce a dedicated placement variable $\omega_{ikl}^t \in \{0,1\}$, which equals 1 if DPC l is placed on ES i. The affinity benefit model for the entire service at time t is expressed as:

$$\Phi^t = \sum_{i\in L}\sum_{j\in L} A_{ij} = \sum_{i\in L}\sum_{j\in L, j>i} \frac{Tra_{ij}}{H(\pi(k,i,t),\pi(k,j,t))+\delta} \quad (10)$$

3.3 Problem formulation

Our objective is to enhance QoS of users by simultaneously minimizing overall service latency and maximizing overall component affinity, which can be expressed as:

$$\text{P0: } \min_{\boldsymbol{b'_{kt}}} \sum_{t=1}^{T} \sum_{k\in I} \sum_{l\in L_U} \Big(\sum_{i\in U_k} x_{ikl}^t CD_{ikl}^t + \sum_{i\in U_k} x_{ikl}^t C_{ikl}^t + \sum_{\bar{\jmath}\in U_k} \sum_{j\in U_k} z_{\bar{\jmath}jkl}^t M_{\bar{\jmath}jkl}^t \Big)$$

$$\max_{\{\omega_{ikl}^t\}} \sum_{t=1}^{T} \Phi^t \tag{11}$$

subject to:

$$x_{ikl}^t \in \{0,1\}, k \in I, \forall i \in U_k, l \in L_U, \forall t \in T \tag{12}$$

$$z_{\bar{j}jkl}^t \in \{0,1\},\ \forall \bar{j} \in U_k, \forall j \in U_k, k \in I, l \in L_U, \forall t \in T \tag{13}$$

$$\sum_{i=1}^{|U_k|} x_{ikl}^t \leq 1, k \in I, l \in L_U, \forall t \in T \tag{14}$$

$$\boldsymbol{b'_{kt}} \in [1,n]^{|L_U|}, k \in I, \forall t \in T \tag{15}$$

$$CD_{ikl}^t + C_{ikl}^t + \sum_{\bar{j} \in U} \sum_{j \in U} M_{\bar{j}jkl}^t \leq L_l, k \in I, \forall i \in J, \forall \bar{j} \in U_k, \forall j \in U_k, l \in L_U, \forall t \in T \tag{16}$$

$$\sum_{k \in I} (\sum_{l \in L_U} x_{ikl}^t c_{kl}^t + \sum_{l \in L_D} y_{ikl}^t c_{kl}^t) \leq f_{i,}\ \forall i \in U_k, \forall t \in T \tag{17}$$

$$\sum_{i \in J} \omega_{ikl}^t = 1\ , k \in I, l \in L_D, t \in T \tag{18}$$

$$\omega_{ikl}^t \in \{0,1\} \tag{19}$$

$$\sum_{k \in I} (\sum_{l \in L_U} x_{ikl}^t c_{kl}^t + \sum_{l \in L_D} \omega_{ikl}^t c_{kl}^t) \leq f_{i,}\ \forall i \in J, \forall t \in T \tag{20}$$

The first objective function sums the total latency for all UFCs across all UEs and time slots, where the three terms correspond to communication delay, computation delay, and migration delay, respectively. Constraints (12) and (13) are binary and migration trigger definitions. Constraint (14) ensures each UFC is placed on at most one ES of U_k. Constraint (15) – (17) enforce decision feasibility, latency bounds, and capacity limits, respectively. The benefit Φ^t in second objective function encompasses the communication benefits between DPCs and between DPCs and UFCs. Note that the two factors determining affinity are traffic and distance. At time slot t, the traffic between two service components is fixed, while the distance between them is the optimization goal. Constraint (18) ensures the uniqueness of DPC deployment. Constraint (19) is binary. Constraint (20) is the resource capacity constraint of ES.

Because the dimensions of the maximizing and minimizing parts of P0 are different, we cannot transform it into a single objective optimization problem through weighted summation. Additionally, P0 is not suitable for classic multi-objective optimization algorithm (e.g., NSGA2). Due to the highly time-varying nature of the MEC, P0's

decisions must be completed within each time slot. NSGA-II needs to be run again in each time slot, and the computation time is unacceptable. Therefore, we adopt Lexicographical Optimization (LO) [40], which prioritizes one objective (latency) over another (affinity), but optimizing the latter as much as possible while satisfying the former's optimality. In MEC, latency is the primary indicator of QoS. If the delay is too high, users will directly perceive a decrease in QoS; Although internal communication efficiency is important, its impact on user perception is indirect. Therefore, prioritizing latency over affinity is a reasonable choice in line with practical system design.

Theorem 1. *Let* X^* *be the optimal solution for LO and* X^{P0} *be the Pareto optimal solution set of P0. Then* X^* *is Pareto optimal and corresponds to the specific preference of 'latency first' in P0.*

Proof. We denote primary objective (latency) of P0 as f_U and the secondary objective (affinity) as f_Φ. X^* is the optimal solution for f_U. If X^* is not Pareto optimal, then there exists X' such that $f_U(X') < f_U(X^*)$ and $f_\Phi(X') > f_\Phi(X^*)$. This contradicts the fact that X^* is the optimal solution for f_U. Therefore, X^* must be Pareto optimal.

Theorem 2. *For P0, its feasible domain is denoted as* $\mathcal{X}$*, and denoted as* $f_U^{min} = \min_{X\in\mathcal{X}} f_U(X)$, $f_\Phi^{max} = \max_{X\in\mathcal{X}} f_\Phi(X)$. *The optimal solution set for* f_U *is defined as* $\mathcal{X}_U^* = \{X \in \mathcal{X} | f_U(X) = f_U^{min}\}$. *If* $X^{U+\Phi}$ *is the solution obtained by LO, then the following performance upper bound holds:*

$$f_U(X^{U+\Phi}) - f_U^{min} = 0 \text{ and}$$

$$0 \le f_\Phi^{max}(\mathcal{X}) - f_\Phi(X^{U+\Phi}) \le f_\Phi^{max} - f_\Phi^{min}(\mathcal{X}_U^*)$$

Proof. (1) $X^{U+\Phi}$ is the solution obtained by minimizing f_U over the entire feasible domain, so $f_U(X^{U+\Phi}) = f_U^{min}$. Therefore, the delay optimization loss is 0.

(2) Due to $X^{U+\Phi} \in \mathcal{X}_U^*$, therefore $f_\Phi(X^{U+\Phi}) = f_\Phi^{max}(\mathcal{X}_U^*)$, also, due to $\mathcal{X}_U^* \subseteq \mathcal{X}$, there are $f_\Phi^{max}(\mathcal{X}_U^*) \le f_\Phi^{max}(\mathcal{X})$. Therefore, $f_\Phi(X^{U+\Phi}) \le f_\Phi^{max}(\mathcal{X})$. Thus, $f_\Phi^{max}(\mathcal{X}) - f_\Phi(X^{U+\Phi}) \ge 0$. That is, the loss of affinity optimization is not negative. On the other hand, due to $X^{U+\Phi} \in \mathcal{X}_U^*$, $f_\Phi(X^{U+\Phi}) \ge f_\Phi^{min}(\mathcal{X}_U^*)$. Therefore, $f_\Phi^{max}(\mathcal{X}) - f_\Phi(X^{U+\Phi}) \le f_\Phi^{max}(\mathcal{X}) - f_\Phi^{min}(\mathcal{X}_U^*)$. Taking into account the upper and

lower bounds, the theorem is proven.

Theorem 2 states that LO optimization has no loss on the primary objective (latency) and a clear upper bound on the loss on the secondary objective (affinity). In practical MEC systems, this upper bound is usually small because: (1) In MEC systems, the coverage radius R_{max} of ES is limited (usually 100-500 meters). Therefore, the range of variation in UFC position $B(p_k, R_{max})$ is limited to a relatively small area, much smaller than the diameter of the entire ES network, where $B(p_k, R_{max})$ represents the coverage area with radius R_{max} centered around the user k; (2) Even if UFC changes within $B(p_k, R_{max})$, the second objective can dynamically adjust the placement position of DPC to compensate for the slight difference in UFC position. Therefore, LO optimization has approximate optimality in practice.

Based on the lexicographic optimization framework (Theorem 1 and Theorem 2), we implement the two-stage decomposition as follows: Stage 1 (P1) solves the primary objective f_U (latency minimization), and Stage 2 (P2) solves the secondary objective f_Φ (affinity maximization) within the optimal solution set of P1.

Stage 1: Latency-Optimal Migration for UFCs (P1)

For UFCs, we should consider the relationship with user, rather than the relationship with each other. Consequently, we aim to minimize the total latency for UFCs by selecting optimal migration strategy $\boldsymbol{b'_{kt}}$, which can be expressed as:

$$\text{P1: } \min_{\boldsymbol{b'_{kt}}} \sum_{t=1}^{T} \sum_{k\in I} \sum_{l\in L_U} \left(\sum_{i\in U_k} x_{ikl}^t CD_{ikl}^t + \sum_{i\in U_k} x_{ikl}^t C_{ikl}^t + \sum_{\bar{j}\in U_k} \sum_{j\in U_k} z_{\bar{j}jkl}^t M_{\bar{j}jkl}^t\right) \tag{21}$$

$$\text{subject to: } (12) - (17)$$

Stage 2: Affinity-Aware Placement for DPCs (P2)

After solving P1 and fixing UFC placements, we now decide DPCs placement to minimize the affinity-aware communication cost, treating the UFC positions as anchors :

$$\text{P2: } \max_{\{\omega_{ikl}^t\}} \Phi^t \tag{22}$$

$$\text{subjec to: } (18) - (20)$$

A natural question is whether P0 can be solved directly as a single-stage optimization. We argue that the two-stage decomposition is a necessary design choice for three reasons: (1) Unit incompatibility: The two objectives cannot be combined into a single scalar without arbitrary normalization. (2) Coupled feasibility: In joint optimization, DPC resource consumption constrains UFC placement, making the problem NP-hard with exponential solution space. (3) Real-time constraint: P0 must be solved within each sub-second time slot. Joint optimization via evolutionary algorithms (e.g., NSGA-II) cannot meet this requirement. Our decomposition addresses all three limitations: it separates objectives into stages with distinct criteria, solves UFC placement first to decouple feasibility, and uses DRL (online) and a linear-complexity heuristic (offline) for P1 and P2, respectively. As proven in Theorem 1, the resulting lexicographic solution is Pareto-optimal for P0.

4 Proposed solution

This section presents the core algorithmic components of our LASM framework. Building upon the problem decomposition in Section 3, we develop solutions for both sub-problems. First, we detail a DRL approach to solve the latency-optimal UFC migration problem (P1). Then, we introduce an affinity-based clustering and placement heuristic to address the DPC placement problem (P2). The synergy between these two stages constitutes the complete LASM strategy.

4.1 DRL-based UFC migration

The UFC migration problem (P1) presents several challenges that motivate our DRL-based approach: it is inherently nonconvex, involves strict delay constraints, and operates in a dynamic environment with time-varying user demands. Traditional optimization methods struggle with such complexity, often resulting in prohibitive computational overhead or inability to adapt in real-time. DRL is particularly well-suited for this problem for three reasons: (1) it handles nonconvex optimization without requiring convexity assumptions; (2) it learns policies that adapt to dynamic environments without re-solving from scratch at each time slot; (3) it supports real-time decision-making through fast neural network inference. To further address the

scalability challenge, we formulate P1 as a cooperative multi-agent MDP. This formulation has two key advantages over a single-agent approach: (1) each ES makes decisions based on its local observations, avoiding the need for a centralized controller to process global state information; (2) the decision space is distributed across agents, making the approach scalable to large-scale MEC deployments with many ESs. A central SDN controller handles only conflict resolution to ensure global feasibility. Based on these considerations, we employ a multi-agent DQN to learn an efficient migration policy, as detailed in the following subsections.

4.1.1 The MDP formulation

We model P1 as a cooperative multi-agent MDP where each ES acts as an independent agent making local UFC placement decisions, while a central SDN controller coordinates actions to ensure global feasibility. Formally, the MDP is defined by the tuple: $(\mathcal{S}, \mathcal{A}, \mathcal{P}(s_{t+1}|s_t, a_t), \mathcal{R}(s_t, a_t))$, where $\mathcal{S}$ is state space, $\mathcal{A}$ is action space, $\mathcal{P}(s_{t+1}|s_t, a_t)$ is state transition probability, and $\mathcal{R}(s_t, a_t)$ is the received reward after performing the action a_t on state s_t.

A. State space

In the service migration problem, the state encapsulates the environment information observed by ES agent when a service component migrates. It comprises four kinds of information, including local ES resources, UFC requirements, the placement of all UFCs in the previous time slot and user location. We denote the state as $s_j^t = \left(f_j, (\mathcal{D}, \mathcal{C}, \mathcal{L}), x^{t-1}, Loc_k^t\right)$. Where f_j is the available computing capacity of ES j, $(\mathcal{D}, \mathcal{C}, \mathcal{L})$ is UFC requirements, here $\mathcal{D} = \{D_1, D_2, \dots, D_{|L_U|}\}$, and $\mathcal{C}$ and $\mathcal{L}$ can be similarly deduced. $\mathcal{D}, \mathcal{C}$ and $\mathcal{L}$ represent the data size, CPU cycle requirement, and latency constraint for each UFC, respectively. $x^{t-1} = (x_1^{t-1}, x_2^{t-1}, \dots, x_{|L_U|}^{t-1})$, where $x_l^{t-1}$is the index of the ES that hosted component l at the end of slot $t-1$, which is crucial for estimating migration costs. Loc_k^t represents the current location of user k, from which the coverage set U_k is derived. U_k determines the feasible placement options for UFCs.

B. Action space

Each ES agent proposes a placement decision for all UFCs through a binary action vector. For agent j at time slot t, the action is defined as:

$$a_j^t = (a_{j1}^t, a_{j2}^t, \dots, a_{j|L_U|}^t) \tag{22}$$

Where $a_{jl}^t = 1$ indicates that UFC l is placed on ES j at time slot t, otherwise it is the opposite. Each ES agent j communicates its action a_j^t to SDN controller, which aggregates them into a global action profile $a^t = (a_1^t, a_2^t, \dots, a_n^t)$, where n is the number of ESs. The action space of ES agent j at time-slot t is given by set $\mathcal{A}_j^t = \{0,1\}^{|L_U|}$. Note that for a given user k, only ESs within its coverage set U_k are eligible to host its UFCs. The SDN controller considers only the proposals from ESs in U_k. It also needs to resolve potential conflicts (e.g., the same UFC being selected by multiple ESs). In this work, we adopt a simple rule: if a UFC is selected by multiple ESs, it is assigned to the ES that reported the highest estimated value (e.g., the Q-value) for accepting the UFC; if a UFC is not selected by any ES, it is assigned to the ES with the most available resources within the user's coverage set U_k. This mechanism ensures that globally optimal placements are derived from local agent proposals while maintaining resource feasibility and respecting the coverage constraint.

C. Reward function

The reward function aligns each agent's objective with the global goal of minimizing end-to-end latency (Eq.11). Specifically, an agent receives a higher reward for actions that result in lower communication, computation, and migration delays. The reward for ES agent j at time slot t is defined as:

$$r_j^t = \begin{cases} -\sum_{l \in L_U} (a_{jl}^t \cdot CD_{ijl}^t + a_{jl}^t \cdot C_{jl}^t + a_{jl}^t \cdot 1_{[x_l^{t-1} \neq j]} \cdot M_{x_l^{t-1} jl}^t), & if\ a_j^t\ is\ feasible \\ R_{penalty}, & otherwise \end{cases} \tag{23}$$

where $1[\cdot]$ is indicator function. A feasible action must satisfy two conditions: (1) the local resource capacity constraint (Constraint 17 of P1), and (2) the UFC must be

placed within the user's coverage set U_k. SDN controller performs this feasibility check before computing the reward. The penalty term $R_{penalty}$ (a large negative value) discourages agent from proposing infeasible placement.

D. State transition and central coordination

The SDN controller orchestrates the multi-agent decision process as follows: **(1) Feasibility check & reward calculation.** Upon receiving an action a_j^t from each ES agent j, the controller first checks its individual feasibility against the local resource constraint. Based on this check and Eq. (23), the reward r_j^t is computed and assigned to the agent. This step uses the agent's proposed action a_j^t and the known system state (including x^{t-1} and user location). **(2) Conflict resolution & global decision.** After all agents are rewarded, the controller resolves conflicts among the proposed actions a_j^t using the rule described in above. This yields a conflict-free, feasible global placement decision x^t, which will be executed. **(3) State Update.** The system transitions to time slot $t+1$. The new state s_j^{t+1} for each agent is constructed, with the historical placement component now updated to the executed decision x^t.

4.1.2 Training with multi-agent double DQN

To handle the high-dimensional state-action space, each ES agent employs a Deep Q-Network (DQN) to approximate its Q-function $Q(s_t, a_t; \omega)$. We adopt the Double DQN architecture [34] to mitigate overestimation bias, using both a Main Q-Network (MQN, parameter ω) and a Target Q-Network (TQN, parameter $\hat{\omega}$). The loss function for updating ω is the mean squared Bellman error:

$$Loss(\omega) = \mathbb{E}[(r_t + \mathrm{r} \max_{a_t \in \mathcal{A}^t} Q(s_t, a_t; \hat{\omega}) - Q(s_t, a_t; \omega))^2] \quad (24)$$

where r is discount factor for balancing immediate and long-term reward, r_t is the obtained reward. $Q(s_t, a_t; \omega)$ is the Q-value of action a_t given the state s_t, which is calculated using MQN with parameter ω while $r_t + \mathrm{r} \max_{a_t \in \mathcal{A}^t} Q(s_t, a_t; \hat{\omega})$ is optimal Q-value, which is calculated using TQN with parameter $\hat{\omega}$.

The training process of our multi-agent DQN is detailed in Algorithm 1 and

illustrated in Fig.3. In each training episode, each ES agent observes its state and selects an action according to an ε-greedy policy (Lines 13–18). The chosen actions are then submitted to the SDN controller. The SDN controller performs feasibility check on each proposed action against local resource constraints of $P1$ (Lines 20–27). Feasible actions are assigned an immediate reward calculated via Eq. (23), which penalizes latency and migration overhead; infeasible ones receive a large negative penalty $R_{penalty}$. Subsequently, the controller resolves conflicts among the proposed actions using a rule based on the agents' Q-values (Line 28), yielding a conflict-free global placement decision x^t. This decision is executed, and the system transitions to the next global state (Line 29). Each ES agent then stores its local experience tuple $(s_j^t, a_j^t, r_j^t, s_j^{t+1})$ into a shared replay memory (Line 32). Once enough experiences are accumulated, a mini-batch is sampled randomly (Lines 34–35) to update the agent's MQN. Specifically, MQN is used to predict the Q-value for the current state-action pair, while TQN provides target Q-value. The parameter ω of MQN is updated by gradient descent on the mean-squared error between the predicted and target Q-values (Line 36). To maintain training stability, TQN parameter $\hat{\omega}$ is periodically synchronized with ω (Line 38). The exploration rate ε decays exponentially with episode count (Line 41), ensuring that agents explore widely initially and gradually exploit the learned policy.

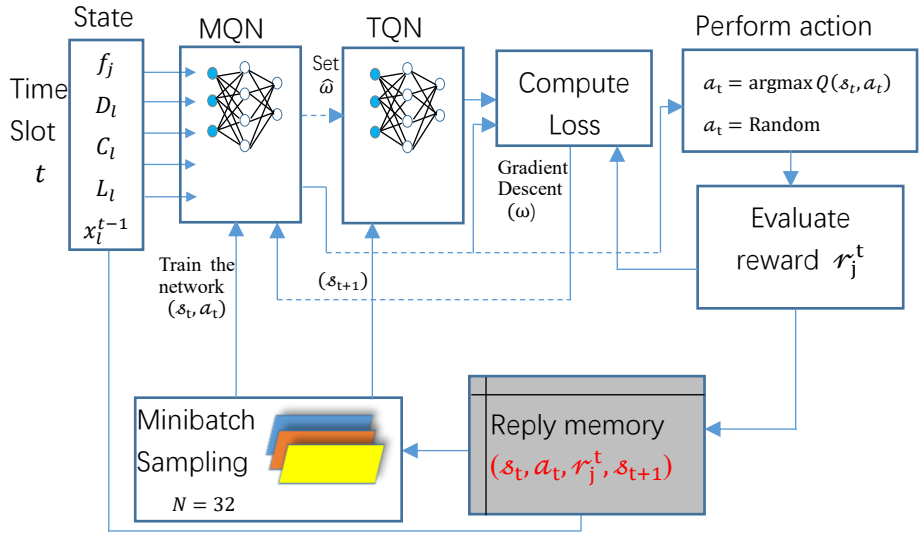


Fig.3. The training process of DQN.

Algorithm 1 Multi-agent DQN training for UFC migration

Input: ES agent set J , environment simulator, hyperparameters

Output: Trained MQN j for all agents $j \in J$

Initialization:

Initialize MQNs with weight ω

Construct TQNs with weight $\hat{\omega}$

Initialize replay memory with capacity M and size of mini-batch N

Initialize maximum number of episodes Max_epis and

Initialize probability of randomly selecting an action ε

Initialize half-life period HT

Initialize global system state $\mathcal{s}^0$ (including initial placement x^0)

for $episode\ e$: $1\ to\ Max_epis$ **do**

  Reset and build each ES agent's environment with $\mathcal{s}^0$

  for $timeslot\ t$: $1\ to\ HT$ **do**

    for $ES\ j$: $1\ to\ n$ **do**

      Extract local state $\mathcal{s}_j^t$ from global state $\mathcal{s}^t$

      Generate a random number x between 0 and 1

      if $x \leq \varepsilon$

        Randomly choose an action a_j^t for a place decision

      else

        Choose an optimal action a_j^t as $a_j^t = \arg\max_{a^t \in \mathcal{A}_j^t} Q(\mathcal{s}^t, a^t)$

      end if

    end for

    for $ES\ j$: $1\ to\ n$ **do**

      Check if a_j^t satisfies Constraint (17) of P1.

      if feasible then

        Compute reward r_j^t using Eq. (23)

      else

25 $r_j^t = R_{penalty}$

26 **end if**

27 **end for**

28 x^t = ConflictResolution(a_j^t, $Q(s_j^t, a_j^t; \omega_j)$) //Rule: for each UFC l, assign to ES j with highest Q-value among those proposing a_{jl}^t

29 Execute x^t, observe next global state s^{t+1}

30 **for** $ES\ j:\ 1\ to\ n$ **do**

31 Extract local next state s_j^{t+1} from s^{t+1}

32 Store $(s_j^t, a_j^t, r_j^t, s_j^{t+1})$ as an interaction experience into the replay memory

33 **if** $batch_size \geq N$

34 Randomly sample the mini-batch with the size N from the replay memory

35 Train the DQN taking the mini-batch as input

36 Update parameter ω using the loss function as Eq. (24) by a gradient descent step

37 **endif**

38 Periodically update TQN by copying ω to $\hat{\omega}$

39 **end for**

40 **end for**

41 $\varepsilon = \varepsilon(0) \times 2e^{-\frac{episode}{HT}}$

42 **end for**

43 Return trained MQNs

Decision-making process for UFC migration after training is presented in Algorithm 2. In this online phase, each ES agent uses its trained MQN to greedily select the action that maximizes the Q-value for placing UFCs (Lines 5–8). The SDN controller then resolves conflicts among the proposed actions using the same Q-value-based rule as in training (Line 9), yielding feasible global placement decision x^t. This decision is executed, and system state is updated for the next time slot (Lines 10–12). The output of Algorithm 2 is a sequence of placement decisions over the decision horizon, which can be directly deployed in MEC system.

Algorithm 2 Online UFC migration decision-making

Input: The trained MQNs

Initial global state s^0

Decision horizon T

Output: Sequence of global UFC placement decisions $\{x^1, x^2, \ldots, x^T\}$

Initialization:

Load the trained MQN for each ES agent

Set current global state $s \leftarrow s^0$

Initialize decision list $decisions \leftarrow []$

for $timeslot\ t: 1\ to\ T$ **do**

  for ES $j: 1\ to\ n$ **do**

    Extract local state s_j from global state s

    Choose an action a_j that maximize the Q-value of the trained MQN

  end for

  $x^t \leftarrow$ ConflictResolution($\{a_j\}$, $\{Q(s_j, a_j;\ \omega_j)\}$)//Conflict resolution rule same as Algorithm 1

  Execute placement x^t //migrate UFCs if needed

  $decisions.\text{append}(x^t)$

  $s \leftarrow \text{UpdateGlobalState}(s,\ x^t)$//Update global state: user movement, ES resource usage

end for

Return $decisions$

4.1.3 computation complexity

The computational complexity of the DRL training phase is $O(E \cdot T \cdot n \cdot |L_U| \cdot H)$, where E is the number of episodes, T is the number of time slots, n is the number of ES agents, $|L_U|$ is the number of UFCs, and H is the hidden layer dimension. The online decision complexity is $O(n \cdot |L_U| \cdot H)$, enabling real-time execution.

4.2 Affinity-based DPC placement

Having determined UFC placements via the DRL approach, we now address the DPC placement problem (P2). As P2 is NP-hard, we propose an efficient two-phase heuristic: (1) affinity-based clustering of component pairs, and (2) UFC-anchored greedy

placement. The key insight is to prioritize placing highly interdependent components (high affinity) in close physical proximity.

4.2.1 Affinity-based Component Pair Clustering

Unlike traditional component-centric clustering, which groups service components based on their individual attributes (e.g., resource demand and function type), our heuristic introduces a novel relationship-centric clustering paradigm. Instead of clustering components, we cluster component pairs (i, j) based on their pairwise affinity values A_{ij}. This fundamental shift in perspective offers two key advantages for optimizing internal communication overhead: 1. Direct optimization target. It directly operates on the relationships that generate traffic overhead, which is the actual target of our cost minimization in P2. By grouping high-affinity pairs together, we explicitly prioritize reducing network distance for the most costly communication flows. 2. Preservation of relative importance. It naturally preserves and utilizes the spectrum of affinity strengths across all component interactions. A component-centric approach might force a component with both strong and weak dependencies into a single cluster, potentially diluting the optimization focus. In contrast, our method allows the same component to appear in multiple pairs across different clusters, ensuring that its strongest dependency (highest A_{ij}) is prioritized in the placement chain of the highest-ranked cluster.

Formally, using affinity matrix A defined in Eq. (1), we apply a K-means variant to the set of non-diagonal affinity values A_{ij} $(i \neq j)$. We choose K-means over alternative clustering methods (e.g., spectral clustering, graph partitioning) for three reasons: (1) K-means is computationally lightweight with complexity; (2) the affinity values are continuous and naturally form clusters in one-dimensional space; (3) our relationship-centric view transforms component clustering into affinity-value clustering, for which K-means is the most direct and efficient approach. While spectral clustering could capture more complex patterns, the added complexity is unnecessary given the clear separation of affinity levels observed in our experiments. The clustering process (Algorithm 3) operates as follows: Step 1, randomly select k affinity values from the

non-diagonal elements of matrix A as initial cluster centers (Lines 1-4). Step 2, calculate the Euclidean distance from each non-diagonal affinity value $A[i][j]$ to each cluster center (Lines 8-15). Step 3, assign each component pair (i, j) to the cluster whose center is nearest (Lines 16-22). Step 4, recalculate each cluster center as the mean of all affinity values in that cluster (Lines 26-28). Step 5, check if centers have changed; if yes, return to step 2. Finally, sort clusters by their mean affinity values in descending order (Line 34). This yields a hierarchy where cluster C_1 contains the most critical (highest-affinity) component relationships, guiding the subsequent placement phase.

Algorithm 3 Affinity-based component pair clustering

Input: affinity matrix A, number of clusters k

Output: clusters $C_1, C_2, \dots, C_k$

Initialization: termination iteration condition variable $term_{iter}$, maximum number of iterations N

1 **for** t: 1 *to* k **do**

2 Randomly select distinct indices $i, j \in [1, r], i \neq j$

3 $center[t] \leftarrow A[i][j]$

4 **end for**

5 Initialize clusters $C_1, C_2, \dots, C_k$

6**while** (! $term_{iter}$ *and iteration* $< N$) **do**

7 clear $C_1, C_2, \dots, C_k$

8 **for** i: 1 *to* r **do**

9 **for** j: 1 *to* r **do**

10 if $i == j$ then continue

11 **for** t: 1 *to* k **do**

12 $distance[i][j][t] \leftarrow$ Calculate the Euclidean distance between $A[i][j]$ and $center[t]$

13 **end for**

14 **end for**

15 **end for**

16 **for** i: 1 to r **do**

17 **for** j: 1 to r **do**

18 if $i == j$ then continue

19 Find the smallest distance $min_distance[i][j]$ and the index t of $center[]$

20 $C_t \leftarrow C_t \cup \{(i,j)\}$

21 **end for**

22 **end for**

23 **for** t: 1 to k **do**

24 $center_old[t] \leftarrow center[t]$

25 **end for**

26 **for** t: 1 to k **do**

27 $center[t] \leftarrow$ Recalculate the center of C_t as mean affinity in each cluster

28 **end for**

29 **if** the elements in $center[]$ no longer change compared to the elements in $center_old[]$

30 $term_{iter} \leftarrow true$

31 **end if**

32 $iteration \leftarrow iteration + 1$

33 **end while**

34 Sort $C_1, C_2, \ldots, C_k$ by mean affinity descending

35 return $C_1, C_2, \ldots, C_k$

4.2.2 UFC-anchored greedy placement

With clusters sorted by affinity importance, Algorithm 4 performs the actual DPC placement. The algorithm iterates through clusters from highest to lowest affinity. For each UFC (serving as an anchor), it traverses the current cluster to find unplaced DPCs connected to already-placed components (UFCs or DPCs placed in previous steps). When such a connection is found, the DPC is placed on the ES nearest to the connected component's ES that has sufficient resources. This creates a placement chain that propagates from UFC anchors along high-affinity relationships. Any DPC not placed through affinity links is assigned to the nearest feasible ES as a fallback.

Algorithm 4 UFC-anchored greedy DPC placement

Input: Clusters $C_1, C_2, \ldots, C_k$, UFCs placement mapping, number of clusters k

Output: DPCs placement mapping: DPCs → ESs

Mark all DPCs as unplaced
for $t = 1$ *to* k **do**
  for each UFC u in $\{UFCs\}$ **do**
    $i \leftarrow u$
    while $\exists\, j$ *such that* $[(i,j) \in C_t$ *or* $(j,i) \in C_t]$ and j is unplaced DPC **do**
      $feasible_ESs \leftarrow \{e \mid$ ES e has resources for component j$\}$
      $target_ES \leftarrow argmin_{e \in feasible_ESs} hop_distance(ES(i), e)$
      Place j on $target_ES$
      Mark j as placed
      $C_t \leftarrow C_t - \{(i,j),(j,i)\}$
      $i \leftarrow j$
    end while
  end for
end for
for each unplaced DPC d **do**
  Place d on nearest ES with sufficient resources
end for
return DPCs → ESs

4.2.3 computation complexity

The clustering algorithm processes $r(r-1)$ non-diagonal affinity values, where r is the total number of service components. Each iteration computes Euclidean distances to K cluster centers, yielding complexity $O(I \cdot r^2 \cdot K)$, where I is the number of iterations until convergence. The placement algorithm iterates through clusters and, for each unplaced DPC, evaluates feasible ESs to find the nearest to an already-placed component. The complexity is $O(K \cdot |L_D| \cdot n)$, where K is the number of clusters, $|L_D|$ is the number of DPCs, and n is the number of ESs. The

total online complexity of LASM (Algorithms 2 and 4) is $O(n\,|L_U|\,H\,+\,K|L_D|\,n)$, which scales linearly with the number of ESs and components. This confirms that LASM is suitable for real-time MEC deployments.

4.2.4 Illustrative Example

We demonstrate the workflow of our affinity-based placement heuristic through a concrete example. Consider a service with UFC A and DPCs B, C, D, E. After clustering (Algorithm 3), we obtain three affinity clusters sorted by descending mean affinity:

C_1: $\{(A, B), (A, C)\}$ (mean affinity = 8.5)

C_2: $\{(B, D), (C, E)\}$ (mean affinity = 5.2)

C_3: $\{(D, E)\}$ (mean affinity = 3.1)

Assume UFC A is already placed on ES_1 by Algorithm 2. Algorithm 4 proceeds as follows:

Processing C_1: Starting from placed UFC A, the algorithm finds B and C in C_1 and places them on ESs nearest to ES_1 with sufficient resources.

Processing C_2: With B and C now placed. From B, the algorithm identifies D connected to B and E connected to C in C_2. It places D near ES(B) and E near ES(C).

Processing C_3: The pair $(D,\ E)$ is already satisfied as both are placed in proximity during step 2.

Fallback Placement: Any DPC without affinity links (not present in this example, e.g., a DPC with no affinity to any placed component) would be handled by Lines 16-18 of the algorithm.

5 Simulation and evaluation

This section validates the effectiveness of our proposed LASM framework through comprehensive simulation experiments. We employ EdgeCloudSim [37] to simulate our MEC system. Our evaluation is structured as follows: Section 5.1 details the experimental setup and workload generation; Section 5.2 presents a systematic hyperparameter tuning study for our DRL model and affinity clustering; and Section

5.3 compares LASM against several state-of-the-art baseline strategies across multiple performance metrics, including additional validation on the EUA dataset with real-world user distributions.

5.1 Experimental setup and workload

We construct a heterogeneous MEC infrastructure comprising 40 ESs and 1 central cloud server. The cloud server handles non-real-time batch processing and global resource orchestration, while latency-sensitive services are hosted on ESs. Tables 3-5 summarize the hardware specifications of ES, cloud server, and VM instance, respectively. These values are representative of mid-range edge servers commonly used in MEC simulations. ESs are randomly deployed in a 2D geographic area with a minimum inter-distance of 400 meters. ESs are interconnected via a high-speed Local Area Network (LAN), while the cloud server connects to the edge tier through a Wide Area Network (WAN) with higher latency. Each service component is encapsulated within a VM, enabling flexible migration. Key system and communication parameters are listed in Table 6. The system and communication parameters in Table 6 are set to values commonly adopted in MEC simulation studies [18], [37]. Specifically, the transmit power (48 mW) and noise power (9.5 dBm) represent typical settings for IoT devices and edge servers in urban environments; the path-loss exponent (γ = 3.0) corresponds to a standard urban propagation model; the antenna far-field distance (d_0 = 1000 m) follows the typical base-station antenna configuration. The migration cost coefficient (λ_m = 0.0002) is calibrated to reflect a moderate migration overhead relative to service latency. User mobility follows the Nomadic Mobility Model, where users exhibit semi-random movement patterns with periods of stability. Service requests arrive at UE according to a Poisson process.

To generate realistic workloads for our MEC environment, we implement four distinct service types (Svc1--Svc4) with different traffic patterns: Constant Bit Rate (CBR), ON-OFF, Poisson Process, and Self-Similar traffic. These traffic patterns are widely adopted in MEC simulation studies to model diverse IoT applications: CBR for video streaming, ON-OFF for voice/web interactions, Poisson for low-load packet

arrivals, and Self-Similar for file transfers and cloud gaming [37]. The service workload parameters in Table 7 are adopted from the EdgeCloudSim configuration used in its official case studies [37]. Each service type is modeled with a distinct traffic pattern (CBR, ON-OFF, Poisson, Self-Similar) and associated task parameters, consistent with the simulator's load generator module. The resource demands (e.g., CPU utilization, etc.) of service components are simulated using Poisson distribution, and different service types use Poisson distribution with different parameters.

The network topology among the 40 edge servers is generated using the BRITE topology generator (version 1.0) [38]. Inter-component communication links and their associated traffic volumes are generated using iperf3 (version 3.9) and D-ITG (version 2.8.1) tools [39], following the traffic patterns specified in Table 7.

All simulation experiments are conducted on the EdgeCloudSim platform (version 1.0). The dataset is partitioned as follows: 10,000 state-transition samples (approximately 70%) are used for DQN training, and 3,000 samples (approximately 30%) are reserved for testing. To ensure statistical reliability, all results are averaged over 10 independent runs with different random seeds (seeds 1--10). Error bars in all figures represent the standard deviation across these runs.

Table 3 Hardware configuration of ES.

Parameter	**Value**
Core	8
Computing Power (Million Instructions Per Second, MIPS)	40000
RAM (MB)	32000
Storage (MB)	500000
Architecture	x86
OS	Linux
Hypervisor	xen
WLAN bandwidth	200

Table 4 Hardware configuration of the cloud server.

Parameter	Value
Core	4
Computing Power (MIPS)	10000
RAM (MB)	32000
Storage (MB)	1000000
Architecture	x86
OS	Linux
Hypervisor	xen

Table 5 VM instance specification.

Parameter	Value
core	2
Computing Power (MIPS)	10000
RAM (MB)	8000
Storage (MB)	125000
Hypervisor	xen

Table 6 System and communication model parameters.

Parameter	Value
Transmit power p	48 mW
Channel gain constant G_0	0.01
Noise power σ^2	9.5 dBm
Antenna far-field test distance d_0	1000 m
Path-loss exponent γ	3.0
WLAN bandwidth B_1	200 Mbps
WAN bandwidth B_2	15 Mbps
Migration cost coefficient λ_m	0.0002

Table 7 Service workload characteristics.

Service	Poisson arrival	Task	CPU	Upload	Download	Active	Idle period

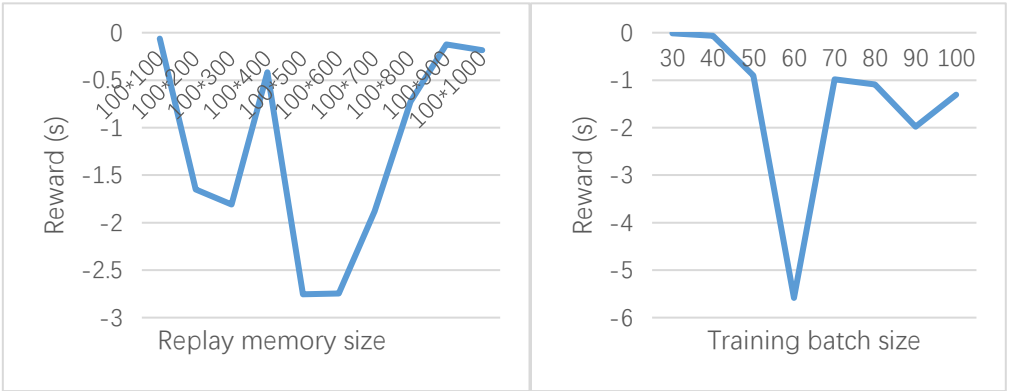
type	interval	length	utilization	data size	data size	period	
Svc1	5	20	20	1500	25	45	15
Svc2	30	4	5	1250	20	10	20
Svc3	60	30	30	2500	250	60	60
Svc4	7	7.5	10	2000	25	15	45

Our multi-agent DQN is implemented using Deeplearning4j library. Each agent's Q-network is a fully-connected neural network with three layers: an input layer matching the state dimension, two hidden layers with 64 neurons each using ReLU activation, and an output layer representing Q-values for each action. We employ the Adam optimizer with a learning rate of 0.001. The target network is updated periodically using a soft update rule with $\tau = 0.01$. Training is conducted for 1000 episodes, with each episode spanning 100 time slots. The model is trained on a dataset comprising 10,000 state-transition samples and evaluated on a separate test set of 3,000 samples.

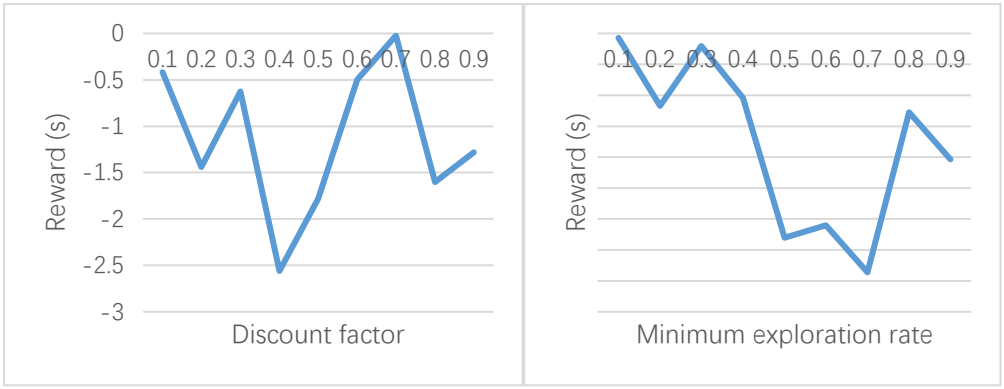
5.2 Hyperparameter tuning and analysis

5.2.1 Hyperparameter optimization

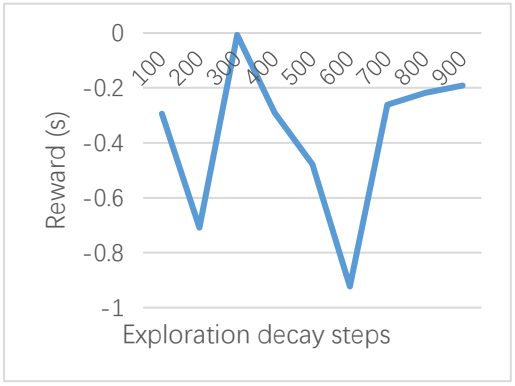
The performance of our DRL-based UFC migration module is sensitive to several key hyperparameters. We conduct a systematic grid search to determine optimal values for: (1) replay memory size, (2) training batch size, (3) discount factor (4) minimum exploration rate, and (5) exploration decay steps. Fig.4 illustrates how these parameters influence the agent's cumulative reward during training.



(a) (b)



(b) (d)



(e)

Fig.4. Influence of DQN parameters on reward.

As shown in Fig.4 (a), experience replay memory size is a crucial hyperparameter in DQN. Its value should match the complexity of the problem. When replay memory increases, reward is always smaller than when the size is equal to 100 *100. This can be explained as when replay memory is larger, it is filled with a large number of outdated experiences, many of which were collected in the early stages of exploration and have low quality, e.g., all failed experiences of random actions. When randomly sampling, the probability of extracting these useless and outdated experiences is high, leading to a decrease in reward. We also notice that reward will fluctuate with the increase of replay memory, mainly due to the high mobility scenario of our edge computing environment. Therefore, we set the replay memory size as 100*100.

Batch size also requires tuning. There is no universally optimal value, only values suitable for specific problems. Fig.4 (b) shows that only when the batch size is 30 or 40, reward is larger. This can be explained as a large batch size may cause the model to

converge prematurely and stably to a local optimum, lacking sufficient randomness to explore a better solution space, resulting in a smaller reward. The reason for reward fluctuations when batch size increases is similar to that of replay memory. Therefore, we set the batch size as 30.

Discount factor controls the agent's foresight. As shown in Fig.4 (c), if it is too small, such as less than 0.6, it will sacrifice long-term benefits for greater short-term rewards; If it is too large (e.g., > 0.8), the agent becomes overly optimistic, treating all future rewards equally regardless of their temporal distance. Given that our learning process is a long cycle task that requires thousands of steps to obtain critical rewards, we set the discount factor to 0.7.

A high minimum exploration rate leads to over-exploration, where agents continue taking random actions even after convergence, degrading the learned policy performance. This will seriously interfere with the execution of excellent policies that have been learned. Given that the action space is exponential in the number of ESs and service components, random actions may cause significant performance degradation. Therefore, we set a smaller minimum exploration rate, e.g., 0.1.

The exploration decay step is configured over 300, ensuring sufficient state-action space coverage before exploitation dominates.

5.2.2 Training convergence analysis

We evaluate the convergence behavior of our multi-agent DQN training process. Fig. 5 presents the reward convergence curve over 1500 training episodes. The reward improves rapidly from -7.59 at episode 1 to -2.63 at episode 500, demonstrating effective initial learning. From episode 500 to 900, the reward continues to improve steadily from -2.63 to -0.98. After approximately 900 episodes, the reward stabilizes around -0.95 with minor fluctuations (±0.05), indicating successful convergence. The small oscillations after convergence are expected due to the stochastic nature of user mobility and service requests in the MEC environment. This convergence behavior confirms that the multi-agent DQN effectively learns a near-optimal policy for UFC migration.

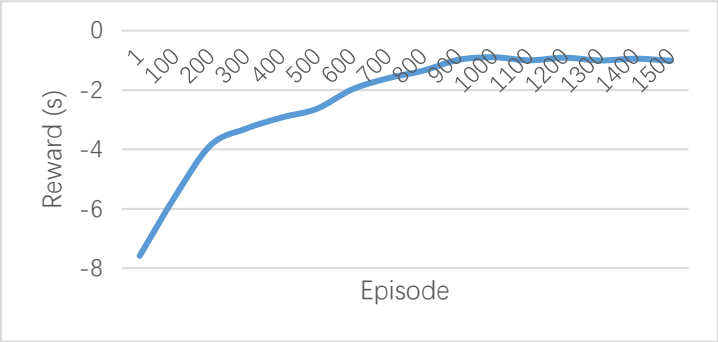


Fig.5. DQN training convergence

5.2.3 Impact of Cluster Number K

The number of clusters K in our affinity-based placement heuristic critically influences the final DPC deployment. We evaluate K across three representative MEC deployment scenarios:

1. **Single-Tier:** A simplified edge-only cluster without cloud collaboration.
2. **Two-Tier:** The classic MEC architecture with edge and cloud layers.
3. **Two-Tier with Edge Orchestrator (EO):** Enhanced with an intelligent orchestrator for cross-edge task offloading.

For each scenario, we vary the number of user devices from 300 to 700. Performance is measured using five key metrics: Service Time (task completion latency), Processing Time (pure computation latency), Network Delay (transmission latency), Server Utilization (CPU usage), and Total Cost (Eq. 11).

Fig.6 reveals that network delay is relatively insensitive to K, as it is primarily determined by user-to-ES distance. However, service time, processing time, and total cost exhibit clear optima. Setting $K < 3$ produces coarse-grained clusters that group components with divergent affinities, leading to suboptimal placements where low-affinity pairs are unnecessarily co-located. Conversely, $K > 6$ creates overly fine-grained clusters, potentially splitting tightly coupled components and increasing cross-ES communication overhead. Across all three scenarios and device scales, $K = 5$ consistently delivers the best or near-optimal performance across the five sensitive metrics. This value provides sufficient granularity to distinguish affinity levels while maintaining meaningful cluster sizes.

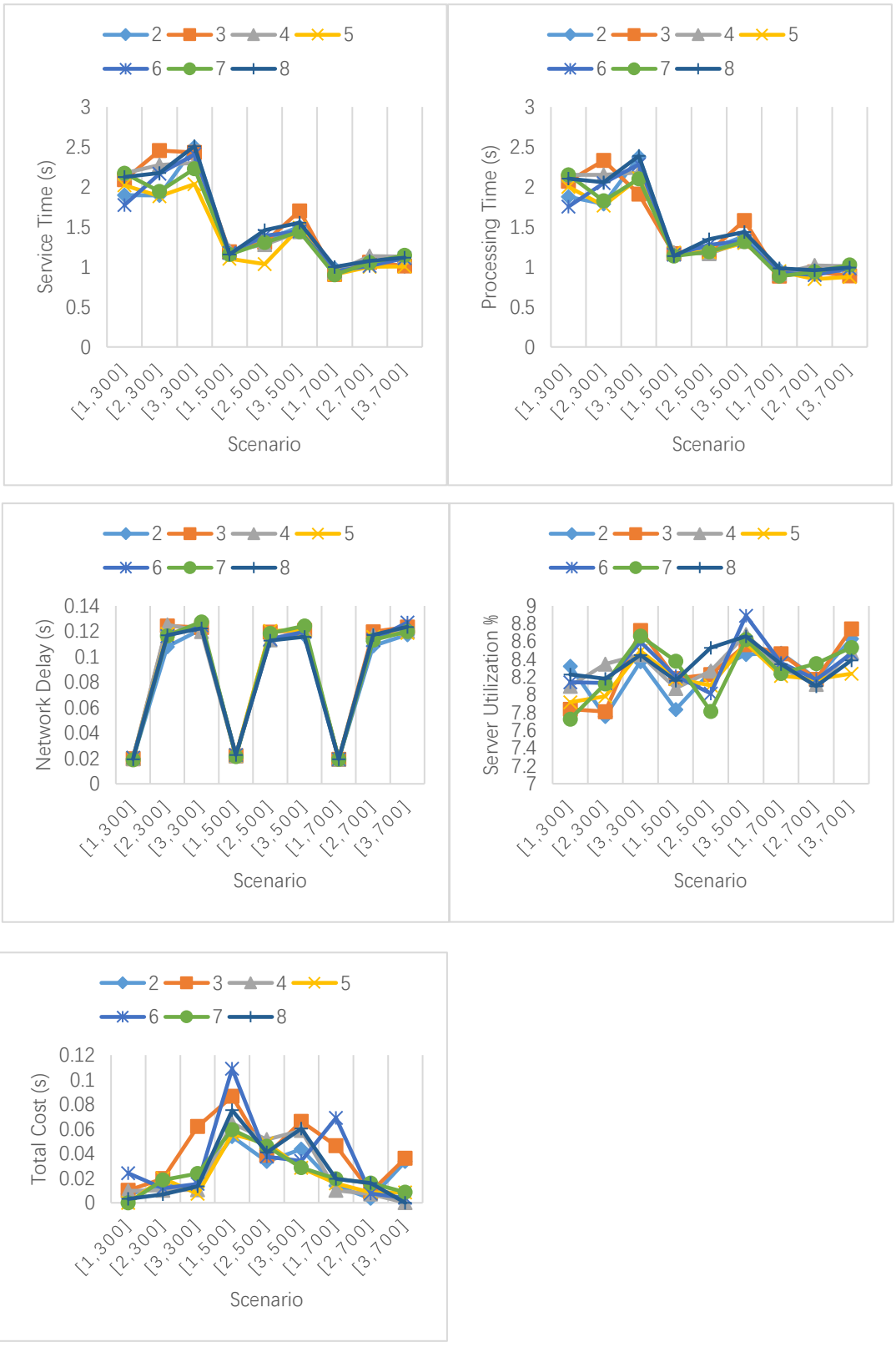


Fig.6. Influence of the number of clusters K on service time, processing time, network delay, server utilization and total cost.

Table 8 Summary of hyperparameters for LASM and baseline methods.

Method	Key Hyperparameters
LASM	Replay memory: 100 × 100, batch size: 30, discount factor: 0.7, min exploration rate: 0.1, exploration decay steps: 300, learning rate: 0.001
NM	None (static placement)
AM	None (always migrate)
LL	None (minimum CPU utilization selection)
DRLM	Same as LASM (see above)
DM	RNN: LSTM (64 units, lr=0.001); HGS: population size: 50, max iterations: 100

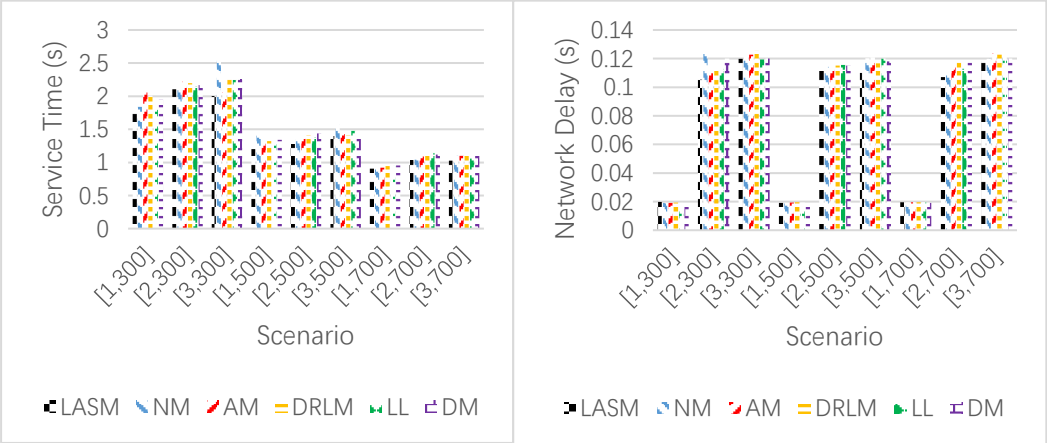
5.3 Performance comparison with baseline strategies

We compare LASM against five representative baseline migration strategies, chosen to cover different design philosophies:

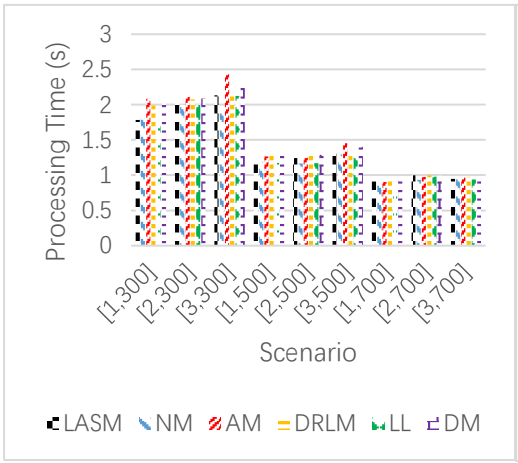
- Never Migration [10] (NM): Components are initially placed optimally and never migrated, regardless of user movement.
- Always Migration [10] (AM): All service components are migrated to the ES closest to user at every movement event, provided resources are available. This is a user-proximity-obsessed, aggressive migration strategy.
- DRL based Migration [1] (DRLM): Uses a multi-agent DRL framework identical to our UFC migration module (Section 4.1) but without the affinity-aware DPC placement.
- Least Load [35] (LL): Components are always migrated to the ES with the lowest current CPU utilization, ignoring user distance and component affinity.
- Dependency-aware Migration [36] (DM): Considering dependencies among service components, optimizes service latency and load balancing through service migration.

To ensure a fair comparison, all baseline methods are implemented with carefully

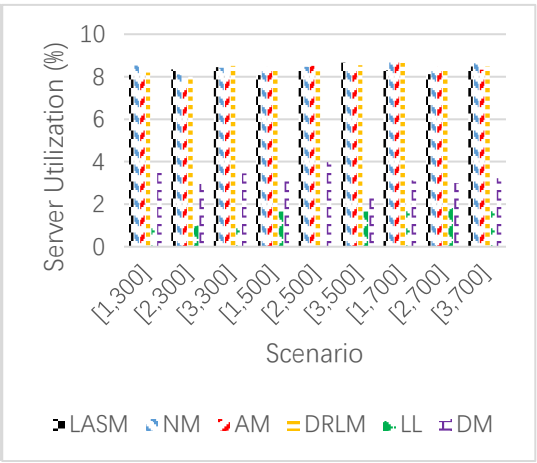
tuned parameters. For NM and AM, no hyperparameters require tuning by nature. For LL, the strategy selects the ES with the minimum current CPU utilization; no additional hyperparameters are involved. For DRLM, we adopt the same DQN architecture and hyperparameters as LASM (see Section 5.2.1), including replay memory size 100*100, batch size 30, discount factor 0.7, minimum exploration rate 0.1, and exploration decay steps 300. This ensures that the comparison isolates the effect of affinity-aware DPC placement. For DM, we implement the RNN-based trajectory prediction and HGS optimizer following the original settings in [36]. The RNN uses a single LSTM layer with 64 hidden units and a learning rate of 0.001. The HGS optimizer is configured with a population size of 50, maximum iterations of 100, and the same convergence threshold as reported in the original work. These parameters were tuned on a validation set (10% of the training data) to ensure optimal performance for our MEC scenario. All hyperparameters for LASM and the baselines are summarized in Table 8. The hyperparameters of DM are from [36]. Other hyperparameters were tuned via grid search (Section 5.2.1). The selected values are reported here. All baseline methods are evaluated on the same training and test partitions to ensure fair comparison.



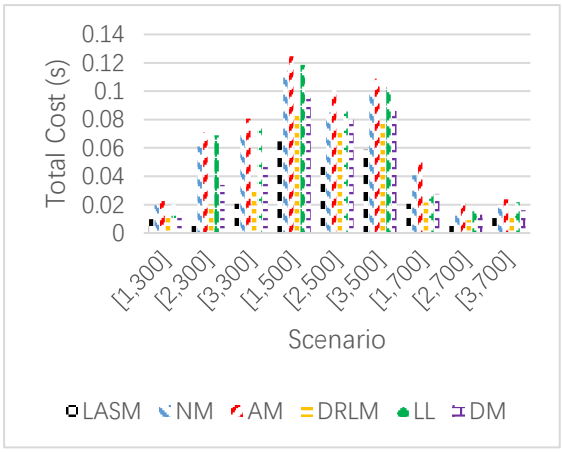
(a) (b)



(c)



(d)



(e)

Fig.7. Comparison of the proposed approach with different baselines.

We evaluate all strategies across nine experimental scenarios, formed by combining three architectural scenarios (Single-Tier, Two-Tier, Two-Tier + EO) with three network scales (300, 500, and 700 devices). The comprehensive results are summarized in Fig. 7.

LASM consistently achieves the lowest service time across all scenarios (Fig.7 (a)). This superiority stems from its two-stage optimization: the DRL-based UFC migration minimizes user-to-service latency, while the affinity-aware DPC placement drastically reduces internal communication delays. LL performs second-best, demonstrating that load balancing alone provides significant latency benefits. NM suffers from high latency when users move away from initial ES placements. AM's excessive migration overhead offsets its proximity gains. DRLM, while optimizing UFC placement intelligently, incurs high internal communication costs due to uncoordinated DPC

placement. DM achieves service times between DRLM and LL in most scenarios. Several factors limit its performance relative to LASM: (1) DM relies on RNN-based trajectory prediction—when prediction errors occur, migration decision becomes suboptimal; (2) DM models only traffic volume as an indirect proxy, whereas our affinity model jointly optimizes both traffic and distance; (3) DM's single-stage optimization cannot separately optimize user-access paths and internal DPC communication.

LASM achieves the lowest network transmission delay (Fig.7 (b)). Its DRL module optimizes the user-access path for UFCs, while the affinity placement minimizes backhaul network traversal for DPC communication. DRLM also maintains relatively low network delay for user-facing traffic due to its intelligent UFC migration, but it incurs higher internal communication delay because it may place highly affiliated DPCs on distant ESs, leading to more backhaul network traffic. Notably, LL—despite ignoring user proximity—achieves competitive network delay, sometimes outperforming AM. This is because LL's load-balancing naturally avoids congested network paths and overloaded ESs, reducing queuing delays. NM incurs the highest network delay as all traffic must traverse potentially long paths to fixed ESs. AM minimizes access network delay but suffers from unoptimized internal DPC communication. DM achieves network delay between DRLM and LL in most scenarios.

LL achieves the lowest pure computation time (Fig. 7(c)), as expected from its load-balancing design that minimizes resource contention. LASM attains a close second. While LASM's DRL module may trigger migrations that temporarily increase processing load, its affinity-aware placement ensures that highly interdependent DPCs are deployed on proximate ESs. This significantly reduces the network latency and protocol processing overhead for inter-component RPCs compared to uncoordinated placements. NM also performs well due to runtime stability. AM suffers from constant VM state transfer and restart costs. DRLM shows higher processing time than LASM, as its lack of affinity optimization can result in high-affinity component pairs being placed on distant ESs, incurring substantial network communication overhead during

service execution. DM exhibits significantly higher processing time than LASM and even DRLM in most scenarios due to two computational bottlenecks: (1) real-time RNN inference for location prediction adds considerable overhead; (2) solving the migration problem via HGS requires multiple iterations and maintains a population of candidate solutions, whereas LASM uses lightweight DRL inference and linear-complexity DPC placement.

Fig.7 (d) presents the average CPU utilization across ESs and cloud server. NM exhibits the highest utilization because fixed placements prevent load redistribution. LASM maintains moderate utilization, benefiting from both intelligent load distribution via DRL and reduced communication overhead via affinity placement. LL achieves the lowest utilization by design, consistently offloading to the least busy servers. DM achieves the second-lowest utilization, as it explicitly targets load balancing alongside latency minimization. DRLM shows higher utilization than LASM due to unoptimized component communication. AM's utilization is inflated by constant migration overhead. The slightly elevated utilization for LASM and NM in the Two-Tier (300-device) and Two-Tier+EO (500-device) scenarios is attributed to specific topology-resource mismatches in those configurations.

The total system cost (Eq. 11), which aggregates communication, computation, and migration costs, serves as our holistic performance metric. As shown in Fig.7 (e), LASM achieves the lowest total cost across all scenarios, demonstrating its ability to optimally balance the three cost components. Specifically, NM incurs exorbitant communication costs due to fixed placements. AM incurs prohibitive migration costs from constant relocation. LL balances costs moderately but suffers from higher communication costs due to ignoring affinity. DM improves upon LL by considering container dependencies, achieving lower communication cost, but its RNN prediction and HGS optimization introduce additional computation overhead that prevents it from matching LASM's efficiency. DRLM significantly improves over NM and AM but is outperformed by LASM due to lack of affinity optimization.

LASM's 33% reduction in total cost compared to DRLM (the strongest DRL baseline)

and up to 11% reduction in average service time quantitatively validate the value of our integrated latency-and-affinity optimization.

To assess the statistical significance of LASM's performance improvements, we conducted the Wilcoxon signed-rank test comparing LASM against each baseline across the nine experimental scenarios. The test results show that LASM significantly outperforms all baselines ($p < 0.05$) in terms of service time, processing time, and total system cost. For network delay, LASM significantly outperforms NM, AM, DRLM, and DM ($p < 0.05$), while the improvement over LL is not statistically significant ($p = 0.072$), consistent with LL's competitive performance in this metric due to its load-balancing effect. These results confirm that LASM's superiority is statistically reliable.

We also evaluate the computational overhead of LASM. The DRL-based UFC migration decision takes approximately 15.68 ms per time slot on an AMD Ryzen 5 3600 CPU, while the heuristic DPC placement completes within 7.24 ms for a service with 10 components. This is well within typical MEC control loop requirements (sub-second), demonstrating LASM's suitability for real-time deployment. The decision latency of LASM is comparable to DRLM and significantly lower than DM, which requires RNN inference and HGS optimization.

To further validate the practical relevance of LASM, we conduct additional experiments using the EUA dataset [41], which provides real-world geographic coordinates of users and edge servers from the Melbourne metropolitan area. Specifically, we use the Melbourne CBD subset, which contains 816 user positions and 125 edge server locations. Unlike the main experiments where user mobility is driven by the Nomadic Mobility Model, this validation adopts random walk model to generate user trajectories. Starting from the real geographic coordinates provided by the EUA dataset as initial positions, each user moves with a controllable step size and direction. The movement speed is set to 1.4 m/s (walking speed), with a direction change probability of 0.15 per time step and a maximum turning angle of 90 degrees. Fig. 8 presents the performance comparison under this realistic user distribution with random walk mobility. The results in Fig. 8 are consistent with our main findings in Fig. 7,

confirming that LASM's advantages hold under realistic user distributions and more challenging mobility patterns.

LASM achieves the lowest service time among all migration-based strategies across the majority of scenarios. In low- and medium-scale scenarios, LASM consistently outperforms DRLM, DM, AM, and LL, demonstrating that its two-stage optimization effectively reduces end-to-end latency. In large-scale scenarios, NM achieves slightly lower service time due to its zero-migration overhead, but this advantage comes at the cost of significantly higher latency in low- and medium-scale scenarios. Overall, LASM provides the most balanced and robust service time performance across all scenarios.

Network delay across all strategies is comparable, with differences among methods remaining small. LASM maintains competitive network delay in all scenarios, confirming that its two-stage optimization does not compromise user-access path efficiency. This is expected, as network delay is primarily determined by the physical distance between users and edge servers rather than the migration policy.

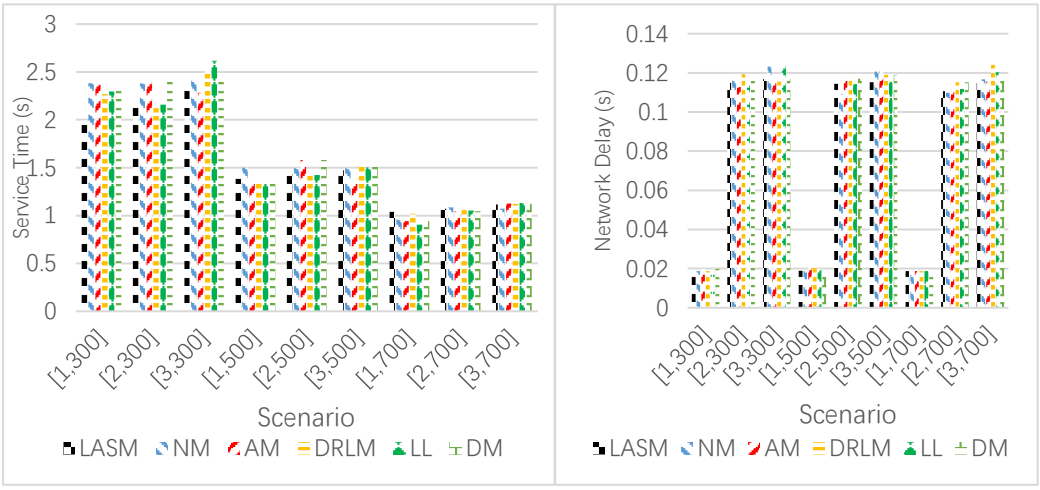
LASM achieves the lowest processing time in the majority of scenarios, outperforming even LL in some low-scale cases. This confirms that LASM's affinity-aware placement significantly reduces inter-component communication overhead, compensating for migration-related processing costs. DRLM and AM exhibit higher processing times than LASM in most scenarios, while NM achieves competitive processing time in large-scale scenarios due to its static placement strategy.

LL achieves the lowest server utilization, as expected from its pure load-balancing design. LASM maintains utilization levels close to LL, ranking among the lowest across all scenarios. This confirms that LASM's affinity-aware placement indirectly lowers CPU utilization by reducing inter-component communication overhead. NM and AM exhibit the highest utilization, as fixed placements prevent load redistribution or constant migration inflates CPU usage.

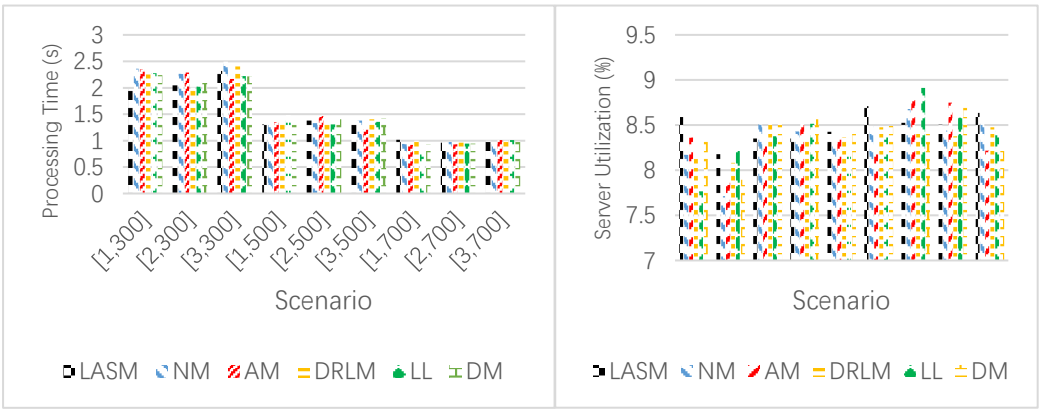
For scenarios with non-zero migration costs, LASM achieves the lowest total cost in most cases. In the few scenarios where LASM's cost is slightly higher than DRLM, the

additional cost is justified by substantially lower service time, confirming that LASM achieves a better overall trade-off. In large-scale scenarios, all strategies achieve near-zero cost due to minimal migration events. NM achieves zero total cost by never migrating, but this comes at the expense of significantly higher service time.

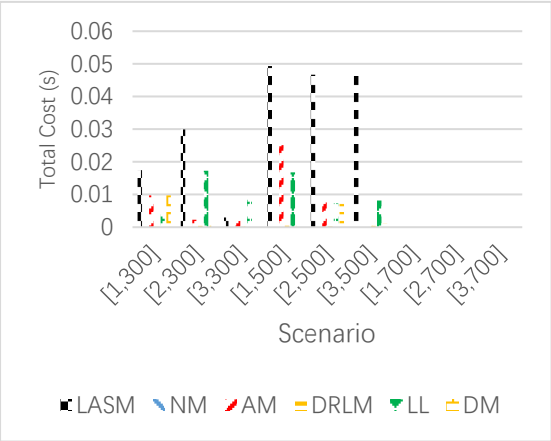
We acknowledge that the current evaluation assumes static traffic patterns (each component uses only one service type, such as CBR or ON-OFF) and reliable infrastructure. Practical MEC deployments may involve dynamic traffic fluctuations (e.g., switching from CBR to ON-OFF or self-similar traffic), node failures, link congestion, and heterogeneous resource demands. While these aspects are beyond the scope of this initial study, they represent important directions for future work, as discussed in Section 6.2.



(a) (b)



(c) (d)



(e)

Fig.8. Comparison of the proposed approach with different baselines under EUA dataset.

6 Conclusion and future work

6.1 Conclusion

This paper addresses the fundamental tension in MEC service migration: satisfying strict user-facing latency while minimizing internal communication overhead among distributed components. To navigate this trade-off, we propose LASM, a two-stage cooperative optimization framework. The core innovation lies in systematically decomposing the joint migration problem into two synergistic sub-problems aligned with component characteristics. We first introduce a principled system model that classifies components into UFCs and DPCs, enabling a natural decomposition into P1 and P2. For P1, we formulate a cooperative multi-agent MDP solved by a DQN, where each ES acts as an independent agent coordinated by an SDN controller. For the NP-hard P2, we design a two-phase heuristic: relationship-driven clustering of component pairs followed by UFC-anchored greedy placement, ensuring tightly coupled components are deployed on proximate ESs with linear complexity.

We validate LASM on EdgeCloudSim against five baselines (NM, AM, DRLM, DM, LL). LASM achieves up to a 11% reduction in average service time and a 33% reduction in total system cost compared to the strongest baseline (DRLM), while effectively balancing user proximity, migration overhead, computational efficiency, and component affinity. These results demonstrate that holistic, two-stage coordination of latency-driven UFC migration and affinity-aware DPC placement is essential for

efficient service migration in next-generation MEC systems.

6.2 Future work

While LASM establishes a solid foundation, several promising research avenues emerge: 1. Proactive and predictive migration. The current LASM framework is reactive to user mobility. Future work could integrate mobility prediction and workload forecasting to enable proactive, anticipatory migration. 2. Robustness under dynamic and unreliable conditions. The current evaluation assumes static traffic patterns and reliable edge infrastructure. Future work should develop online affinity learning mechanisms, enabling LASM to adapt to sudden traffic pattern shifts (e.g., from CBR to ON-OFF or self-similar traffic). Incorporating fault-tolerant mechanisms into the DPC placement heuristic and enabling the SDN controller to react to node failures would enhance system robustness. 3. Enhanced resource and cost models. Our current model focuses on CPU resources. Extending LASM to incorporate multi-dimensional resource constraints and more realistic, time-varying cost models (e.g., energy prices, network tariffs, QoS-SLA violations) would enhance its practicality for heterogeneous edge hardware and diverse business scenarios. 4. Relaxed co-location assumption. Relaxing the assumption that components of the same service cannot share an ES and investigating the trade-off between co-location benefits and fault tolerance risks is another promising direction. A possible approach is to allow co-location only when the ES has sufficient redundancy. 5. Extensions to broader architectures: Investigating LASM's applicability to related paradigms such as Serverless Edge Computing (SEC) or Federated Learning (FL) workflows presents another fruitful direction. Exploring integration with digital twin platforms for simulation-in-the-loop optimization is also promising.

References

[1] Abouaomar A, et al. A Deep Reinforcement Learning Approach for Service Migration in MEC-enabled Vehicular Networks[J]. 10.48550/arXiv.2201.11831,2022.

[2] Shiqiang Wang, et al. 2014. Mobility-induced service migration in mobile micro-

clouds. 2014 IEEE Military Communications Conference (MILCOM'14). 835–840.

[3] Xuhui Zhao, et al. Service Migration Policy Optimization considering User Mobility for E-Healthcare Applications[J]. JOURNAL OF HEALTHCARE ENGINEERING,2021.

[4] Zhiqing Tang, et al. Migration modeling and learning algorithms for containers in fog computing. IEEE Transactions on Services Computing (2018).

[5] Gao Z, et al. Deep Reinforcement Learning Based Service Migration Strategy for Edge Computing[C]. Service Oriented Software Engineering. IEEE, 2019.

[6] Rahul Urgaonkar, et al. Dynamic service migration and workload scheduling in edge-clouds. Performance Evaluation 91 (2015), 205–228.

[7] Rui L, et al. A dynamic service migration strategy based on mobility prediction in edge computing[J]. International Journal of Distributed Sensor Networks, 2021, 17.

[8] Ge S, et al. A Two-Stage Service Migration Algorithm in Parked Vehicle Edge Computing for Internet of Things[J]. Sensors (Basel, Switzerland), 2020, 20(10).

[9] Yudong Qin, et al. Service function chain migration with the long-term budget in dynamic networks[J]. Computer Networks, Volume 223,2023.

[10] Wang J, Hu J, Min G. Online Service Migration in Edge Computing with Incomplete Information: A Deep Recurrent Actor-Critic Method[J]. 2020.DOI:10.48550/arXiv.2012.08679.

[11] Wang H, et al. Service migration in mobile edge computing: A deep reinforcement learning approach[J]. International Journal of Communication Systems, 2020.

[12] S. Ge, M. Cheng, X. Zhou. Interference Aware Service Migration in Vehicular Fog Computing[J]. IEEE Access, vol. 8, pp. 84272-84281, 2020.

[13] Q. Yuan, et al. Shen. Exploring the effectiveness of service migration strategies for virtual network. IEEE Transactions on Vehicular Technology, vol. 69, no. 8, pp. 9041-9052, Aug, 2020.

[14] Xu Y, et al. Three-way decisions based service migration strategy in mobile edge

computing[J]. Information Sciences: An International Journal, 2022.

[15] Q. Yuan, et al. A Joint Service Migration and Mobility Optimization Approach for Vehicular Edge Computing. IEEE Transactions on Vehicular Technology, vol. 69, no. 8, pp. 9041-9052, Aug, 2020.

[16] Wei Bao, et al. Follow me fog: Toward seamless handover timing schemes in a fog computing environment. IEEE Communications Magazine 55, 11 (2017), 72–78.

[17] Haoxin Wang, Jiang Xie, Tao Han. A smart service rebuilding scheme across cloudlets via mobile AR frame feature mapping. 2018 IEEE International Conference on Communications (ICC'18). 1–6.

[18] Wang S, et al. Dynamic Service Placement for Mobile Micro-Clouds with Predicted Future Costs[J]. IEEE Transactions on Parallel & Distributed Systems, 2015.DOI:10.1109/ICC.2015.7249199.

[19] Yang R, He H, Zhang W. Multitier Service Migration Framework Based on Mobility Prediction in Mobile Edge Computing[J]. 2024.

[20] Chen, et al. A Dynamic Service Migration Mechanism in Edge Cognitive Computing[J].ACM Transactions on Internet Technology, 2019, 19(2).

[21] Cao Q, et al. An Optimization Method for Mobile Edge Service Migration in Cyberphysical Power System[J]. Wireless Communications and Mobile Computing, 2021, 2021:1-12. DOI:10.1155/2021/6610654.

[22] L. Liang, et al. Particle Swarm Based Service Migration Scheme in the Edge Computing Environment[J]. IEEE Access, vol. 8, pp. 45596-45606, 2020.

[23] Chowdhury M, et al. Context-Aware Task Migration for HART-Centric Collaboration over FiWi Based Tactile Internet Infrastructures[J]. IEEE Transactions on Parallel & Distributed Systems, 2018:1-1.

[24] J. Hu, M. Jiang, et al. Joint optimization of UAV position, time slot allocation, and computation task partition in multiuser aerial mobile-edge computing systems. IEEE Trans. Veh. Technol., vol. 68, no. 7, pp. 7231–7235, Jul. 2019.

[25] Roberto Bruschi, et al. An SDN/NFV Platform for Personal Cloud Services[J].

IEEE transactions on network and service management,2017,14(4):1143-1156.

[26] R. Bruschi, F. Davoli, P. Lago and J. F. Pajo. Move with Me: *2018 IEEE International Conference* on Communications (ICC), Kansas City, MO, USA, 2018, pp. 1-6.

[27] Song S, et al. Clustered Virtualized Network Functions Resource Allocation based on Context-Aware Grouping in 5G Edge Networks[J]. IEEE Transactions on Mobile Computing, 2019:1-1. DOI:10.1109/TMC.2019.2907593.

[28] Li B, et al. Joint Resource Optimization and Delay-aware Virtual Network Function Migration in Data Center Networks[J]. IEEE Transactions on Network and Service Management, 2021, PP(99):1-1.DOI:10.1109/TNSM.2021.3067883.

[29] Federico Giarrè, Hadjadj-Aoul Y. Exploring the effectiveness of service migration strategies for virtual network embedding[J]. Computer Networks, 2024, 250.DOI:10.1016/j.comnet.2024.110553.

[30] Pham T M, Nguyen T M. Function traffic-aware VNF migration for service restoration in an NFV-enabled IoT system[J]. ICT Express, 2024, 10(2):374-379.DOI:10.1016/j.icte.2023.11.004.

[31] Eramo V, et al. An Approach for Service Function Chain Routing and Virtual Function Network Instance Migration in Network Function Virtualization Architectures[J]. IEEE/ACM Transactions on Networking, 2017, 25(4):2008-2025.DOI:10.1109/TNET.2017.2668470.

[32] Badri H, et al. Energy-Aware Application Placement in Mobile Edge Computing: A Stochastic Optimization Approach[J]. IEEE Transactions on Parallel and Distributed Systems, 2020, 31(4):909-922. DOI:10.1109/TPDS.2019.2950937.

[33] Cziva R, et al. Latency-Optimal vNF Placement at the Network Edge[J]. IEEE, 2018. DOI:10.1109/INFOCOM.2018.8486021.

[34] Mnih, V., et al. Human-level control through deep reinforcement learning. Nature, 2015, 518(7540), 529-533

[35] S. Wang et al. Load-Aware Service Migration in Mobile Edge Computing, in Proc. IEEE INFOCOM Workshops, 2020, pp. 916-921.

[36] Zhang et al. A Trajectory Prediction-Based and Dependency-Aware Container Migration for Mobile Edge Computing. IEEE Transactions on Services Computing, 16,2023, 3168-3181.

[37] C.Sonmez, A.Ozgovde, and C.Ersoy. EdgeCloudSim: An environment for performance evaluation of edge computing systems. Proc. Int. Conf. Fog Mobile Edge Comput. (FMEC)}, 2017, pp. 39--44.

[38] A. Medina et al. BRITE: An approach to universal topology generation. Proc. IEEE Int. Symp. Modeling, Anal. Simulation Comput. Telecommun. Syst. (MASCOTS)}, 2001, pp. 346--353.

[39] A. Botta et al. D-ITG 2.8.1 manual, Univ. Napoli Federico II, Tech. Rep., 2013.

[40] Lai, L., et al. Pure and mixed lexicographic-paretian many-objective optimization: state of the art. Natural Computing, 22, 227–242.

[41] P. Lai, et al. Optimal edge user allocation in edge computing environments. Proc. Int. Conf. Service-Oriented Comput. (ICSOC), 2018, pp. 93--107.

[42] H. Cao, et al. A novel slice reconfiguration method based on VNF fusion and soft migration for 6G networks. IEEE Trans. Veh. Technol., Early Access, 2026, doi: 10.1109/TVT.2026.3690930.

[43] Q. He et al. Blockchain-enabled elastic task offloading and migration scheme in edge computing system. IEEE Trans. Services Comput., Early Access, 2025, doi: 10.1109/TSC.2025.3634215.

[44] X. Zhao, et al. MAPSM: Mobility-aware proactive service migration framework for mobile-edge computing in consumer Internet of Vehicles. IEEE Trans. Consum. Electron., vol. 71, no. 2, pp. 3753--3766, May 2025.

[45] H. Cao, et al. An intelligent softwarized resource management and allocation framework for services with personalized intentions in 6G-enabled IoT networks. IEEE Internet Things J., vol. 13, no. 5, pp. 8261--8274, Mar. 2026.

[46] L. Zeng, et al. Toward collaborative and latency-aware microservice migration in mobile edge computing. IEEE Internet Things J., vol. 12, no. 13, pp. 25286--25299, Jul. 2025.

Funding Statement

This study was funded by Science and Technology Department of Shaanxi Province, grant numbers: 2026CY-YBXM-145 and grant numbers: 2024JC-YBQN-0730. This organization did not influence the study design, data collection, analysis, interpretation, manuscript writing, or the decision to submit the manuscript for publication.

Data Availability

All simulation code, configuration files, network topologies are publicly available at https://github.com/liuxialin77/LASM-service-migration.

Ethical statement

No conflict of interest exits in the submission of this manuscript, and manuscript was approved by all authors for publication. I would like to declare on behalf of my co-authors that the work described was original research that has not been published previously, and not under consideration for publication elsewhere, in whole or in part. All the authors listed have approved the manuscript that is enclosed.