Journal of Data Science and Modeling, Vol. 3, No. 1, 223-253, December 2024

Research Manuscript

A placement method based on deep reinforcement learning for mission critical services in computing continuum

Reza Sookhtsaraei[1], Mehdi Sakhaei-nia*[1], Fereshteh Azadi Parand[2]

[1] Department of Computer Engineering, Faculty of Engineering,
Bu-Ali Sina University, Hamedan, Iran.

[2] Faculty of Statistics, Mathematics and Computer Sciences,
Allameh Tabatabai University, Tehran, Iran.

Recieved: 10/07/2025 Accepted: 10/03/2026

Abstract: Critical services increasingly rely on distributed computing infrastructures that can deliver fast and reliable responses under dynamic conditions. Traditional cloud-centric deployments often struggle to meet these demands due to latency and reliability constraints. This paper addresses these challenges by exploring intelligent service placement across the cloud–edge computing continuum. We introduce an adaptive placement framework that continuously adjusts deployment decisions in response to changing system states and service requirements. To improve robustness, the framework integrates learning transfer mechanisms and a criticality-aware resilience strategy that prioritizes service continuity during failures. The proposed approach enhances responsiveness and reliability, leading to a higher proportion of services meeting strict timing constraints. Comprehensive experimental evaluations confirm that the framework provides more effective support for critical and delay-sensitive services compared to existing placement strategies in continuum-based environments

Keywords: Mission critical services, Computing continuum, Service placement, Deep reinforcement learning, Fault tolerance.

Mathematics Subject Classification (2020): 68T07, 68N30

*Corresponding Author: sakhaei@basu.ac.ir

1. Introduction

Limitations of traditional centralized infrastructures, are high latency, limited scalability, and reduced reliability (Tripathi and Roshan Ramdas Markhande , 2024). By distributing computation across cloud, edge, and intermediate nodes, the continuum enables low-latency processing, improved responsiveness, and enhanced operational safety for time-sensitive services (Chauhan *et al.* , 2024). In domains such as healthcare, applications including remote patient monitoring, emergency response, and real-time diagnostics directly benefit from localized data processing at the edge (Hennebelle *et al.* , 2024). Moreover, continuum-based architectures reduce network congestion and mitigate Quality of Service (QoS) violations caused by transmission delays, making them well suited to the dynamic requirements of critical (Aslanpour *et al.* , 2020; Tripathi and Roshan Ramdas Markhande , 2024). Despite the benefits of the computing continuum, deploying critical services remains difficult, as their strict QoS requirements significantly restrict service placement decisions. Inadequate placement can result in severe performance degradation or system failure, with potentially catastrophic consequences in sensitive application domains such as healthcare, emergency management, and defense (Sandhu *et al.* , 2010). Addressing these challenges requires placement strategies that emphasize adaptability, resilience, and performance, supported by fault tolerance mechanisms, proactive resource management, and continuous QoS monitoring (Malazi *et al.* , 2022; Patwary *et al.* , 2022). Service placement in the computing continuum is inherently complex and NP-hard, as it involves evaluating a large solution space under dynamic constraints related to resource availability, network conditions, security requirements, and fluctuating service demands. This complexity is further amplified by the need for real-time adaptation to environmental changes, which is especially critical for delay-sensitive services (Tsolkas *et al.* , 2022). To tackle this problem, prior research has explored heuristic and evolutionary optimization techniques to derive near-optimal placement solutions (Herrera *et al.* , 2025; Natesha and Guddeti , 2021). While these methods offer scalability and flexibility, they often rely on simplified decision rules or incur high computational costs, limiting their effectiveness in dynamic and real-time scenarios (Hedhli and Mezni , 2021; Salaht *et al.* , 2020; Natesha and Guddeti , 2021; Vivo *et al.* , 2024; KeshavarzHaddadha *et al.* , 2024). Recent advances in Deep Reinforcement Learning (DRL) provide a promising alternative by enabling adaptive, data-driven decision-making through continuous interaction with the environment (Liu *et al.* , 2022; Ocampo and Santos , 2024). DRL techniques can capture long-term system dynamics and autonomously optimize service placement and resource allocation in complex and evolving continuum environments (Lu *et al.* , 2022). Building on these insights, this paper proposes an intelligent placement framework based on

DRL to support critical services across the computing continuum. The approach leverages the Proximal Policy Optimization (PPO) algorithm to enable stable and efficient decision-making in highly dynamic. Transfer learning is incorporated to accelerate convergence and enhance adaptability across similar deployment contexts. Furthermore, a primary–backup fault-tolerance strategy is employed to improve service reliability by ensuring seamless failover for critical services. The main contributions of this paper are summarized as follows:

Proposing an intelligent DRL-based framework for placing critical services in the computing continuum.

Employing PPO combined with transfer learning to achieve stable and fast convergence in dynamic environments.

Enhancing real-time placement decisions by explicitly considering the stringent requirements of critical services.

Improving service reliability through proactive fault tolerance mechanisms.

Evaluating the proposed approach against baseline and state-of-the-art service placement methods.

The remainder of this paper is organized as follows.Section2 reviews related work on service placement in the computing continuum. Section3 presents the system model and problem formulation. Section 4 details the proposed intelligent placement framework. Section 5 discusses performance evaluation and comparative results. Finally, Section 6 concludes the paper and outlines future research directions

2. Related Works

Existing researches on service placement in the computing continuum can be broadly categorized into metaheuristic, reinforcement learning–based, and hybrid optimization approaches. Table (1) summarizes representative studies by application scope, optimization objectives, and priority-awareness. Several works incorporate service prioritization either explicitly through task classification or implicitly via weighted objective functions, where dynamic weighting reflects sensitivity to response-time requirements.

Metaheuristic-based approaches have been widely adopted to address the complexity of continuum- based combinatorial placement. Pallewatta *et al.* (2024) model applications as serial–parallel structures and employ a hybrid Particle Swarm Optimization and Genetic Algorithm framework with proactive redundancy to improve

Table 1: Qualitative comparison of related works to the proposed

Work	Service priority	Algorithm	Optimization objective
Pallewatta *et al.* (2024)	*	Metaheuristic: PSO+GA	latency reliability price
Ghobaei -Arani and Shahidinejad (2022)	Without category	WOA	response time energy consumption
Sarrafzade *et al.* (2022)	*	GA	delay resource usage
Qin *et al.* (2023)	*	GA	response time resource loss service cost resource usage
Lin *et al.* (2024)	*	PO	latency resource utilization
Dogani *et al.* (2024)	*	GA	cost latency energy consumption
Saadian *et al.* (2023)	Hard, Firm, Soft	GA	energy consumption resource wastage network latency
Ghasemi (2024)	*	HHO	delay energy consumption
Apat *et al.* (2024)	*	GA SA PSO GA+SA GA+PSO	makespan energy consumption cost
Wang *et al.* (2024)	*	Clipped-PPO	response time load balancing
Ramezani Shahidani *et al.* (2023)	Real-time Important Non-real time	Q-learning	response time load balancing energy consumption
Siyadatzadeh *et al.* (2023)	Real-time	Q-learning	reliability delay load balancing
Sharma and Thangaraj (2024)	*	DDQN	execution time energy consumption
Ibrahim and Askar (2023)	*	DQN + GA	makespan delay load
Proposed	critical potential critical normal	Clipped-PPO with transfer learning	response time load balancing reliability

reliability. However, variable redundancy levels may overload resource-constrained edge nodes. Pallewatta *et al.* (2024) apply the Whale Optimization Algorithm to jointly minimize response time and energy consumption, but the approach does not incorporate fault tolerance under dynamic conditions. Genetic Algorithm–based placement is further explored by Sarrafzade *et al.* (2022), who introduce penalty functions to discourage suboptimal placements; nevertheless, random initial populations limit responsiveness to real-time environmental changes and heterogeneous service priorities. Qin *et al.* (2023) formulate a multi-objective MILP model supported by GA-based node selection, enabling dynamic priority adjustment, yet failure handling for high-priority demands remains unaddressed. Lin *et al.* (2024) propose a distributed placement strategy using a Political Optimizer to position high-demand services near data sources, butinter-microservice communication costs and load balancing at the edge are not considered. Similarly Dogani *et al.* (2024) employ a multi-objective GA with dynamic replication, though constraints on replica counts and fault-tolerance exploitation are not investigated. Other metaheuristic solutions address priority-aware placement (Saadian *et al.* , 2023) or general optimization (Ghasemi , 2024; Apat *et al.* , 2024), yet they often lack mechanisms for reliability, environmental adaptability, or rapid convergence for delay-sensitive services.

Reinforcement learning (RL)–based methods have gained attention due to their ability to adapt to dynamic environments. Wang *et al.* (2024) utilize Deep Reinforcement Learning to optimize weighted-cost scheduling for DAG-based IoT applications, focusing on load balancing and latency reduction; however, the absence of explicit priority differentiation limits applicability to critical services. Ramezani Shahidani *et al.* (2023) propose an RL-based scheduler that maps tasks to nodes based on time sensitivity, but the lack of fault-tolerance support weakens reliability guarantees. Siyadatzadeh *et al.* (2023) integrate RL with a primary–backup mechanism to enhance reliability and response time, yet assuming uniform task priority leads to excessive backup overhead at the edge. Sharma and Thangaraj (2024) introduce a DRL-driven node selection strategy that prioritizes tighter deadlines via queue-based mechanisms, though fault handling for high-priority services is not addressed. Ibrahim and Askar (2023) present a multi-objective DRL framework with separate agents per objective and GA-based decision fusion; however, the resulting multi-stage optimization incurs delays that are unsuitable for latency-critical scenarios and does not consider failure management.

Overall, existing solutions either lack sufficient adaptability, incur high computational overhead, or fail to jointly address priority awareness and reliability for critical services. Motivated by these limitations, this work introduces an in-

telligent service placement framework that explicitly classifies service demands by criticality, adapts placement decisions to dynamic continuum conditions, and incorporates a lightweight fault-tolerance mechanism to ensure reliable support for critical services under failures.

3. System Model and Problem Formulation

This section formalizes the computing continuum environment and defines the service placement problem addressed in this work. Table (2) summarizes the key notations used throughout this section.

3.1 System Model

We consider a heterogeneous computing continuum composed of a set of nodes N where each node, is characterized by its computational capacity, memory, and storage resources, denoted as $\{{n_j}^{cpu}, {n_j}^{ram}, {n_j}^{storage}\}$ Service placement decisions are orchestrated by a distributed Service Placement Module (SPM), depicted in Figure 1. An instance of the SPM can be deployed on any continuum node and is responsible for managing a subset of nodes within its administrative domain, enabling decentralized control and scalability across the continuum. Applications are modeled using a microservice-based architecture and represented as Directed Acyclic Graphs (DAGs).An application, consists of a set of microservices , where dependency relationships are defined through predecessor and successor sets and respectively (figure 2). Each application request is associated with a response criticality level, corresponding to Critical, Potentially Critical, and Normal services. Critical requests impose strict deadline constraints, potentially critical requests tolerate limited violations with increased risk, and normal requests allow relaxed timing requirements.

Upon receiving an application DAG, the SPM invokes its DRL-based decision-making component to determine suitable hosting nodes according to service criticality and current system conditions. The Dispatcher translates these decisions into a physical deployment plan, while a Node Watcher continuously monitors node states and provides feedback to support adaptive placement. This architecture enables coordinated yet distributed placement decisions across the computing continuum.

3.2 Problem formulation

The service placement problem is formulated as a multi-objective optimization task that jointly considers response time, load balancing, and reliability.

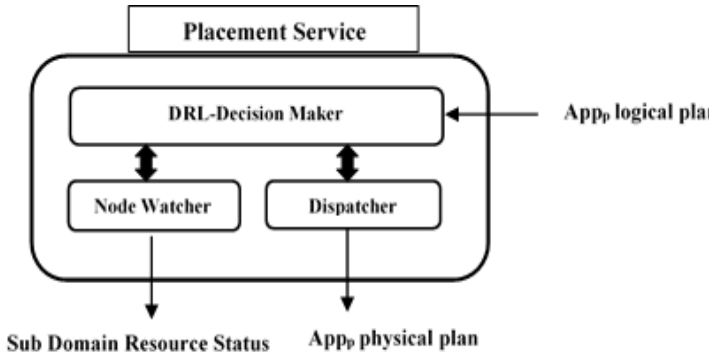


Figure 1: Service Placement Module with its components

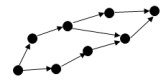


Figure 2: Example of a logical plan of an application

Table 2: List of key symbols

Symbol	Description
N	Set of computing nodes
n_j	Node j
n_j^{cpu}, n_j^{ram}, $n_j^{storage}$	Node j CPU, RAM, storage
App_p	DAG of application p
S_p	Microservice set of application p
EL	Emergency level of request
$PD(s_p^i)$, $SD(s_p^i)$	Predecessor/successor microservices
R_p^i	Response time of microservice i
$rdtt_p^i$, pt_p^i	Transfer and processing time
$tt_{n_j.n_k}$, $d_{n_j.n_k}$, $b_{n_j.n_k}$	Inter-node transfer time, data size, bandwidth
rq_k^{EL}	Ready queue at node k for priority EL
$LD_{S_p^i}$	Load deviation after placing service i
$n_j^{cpu\_req}$, $n_j^{ram\_req}$	Current CPU/RAM requests
$RelaP_P$, rl_j	Reliability-access matrix and node reliability

3.2.1 Response time

The response time of microservice is computed ${s_p}^i$ incrementally during placement as

$$ {R_p}^i = \left({rdtt_p}^i + {pt_p}^i \right)_j \tag{3.1} $$

whereR_p^i, is the response time of microservice i in application p, ${rdtt_p}^i$ denotes data transfer time from predecessors microservices, and ${pt_p}^i$ is the processing time. Subscript j indicates node assignment. Microservices within a DAG execute sequentially or in parallel. A child waits for its parents to complete, while independent nodes can run in parallel. Transfer time ${rdtt_p}^i$ is given by

$$ \text{rdtt}_p^i = \max\left(tt_{n_j,n_k}\right), \quad \text{if } n_j\left(s_p^{i-1}\right) \neq n_k\left(s_p^i\right), \quad \forall\, s_p^{i-1} \in PD\left(s_p^i\right). \tag{3.2} $$

where tt_{n_j,n_k} is the time needed to transfer data from node , hosting the predecessor microservice, to node , where microservice is placed. This transfer time is computed by

$$ tt_{n_j,n_k} = \frac{d_{n_j,n_k}}{b_{n_j,n_k}} \tag{3.3} $$

with d_{n_j,n_k} as the output data must be transferred and b_{n_j,n_k} as the available bandwidth between node j and node k . Processing time ${pt_p}^i$ on node is calculated by

$$ pt_p^i = \frac{\sum_{l=1}^{|rq_k^{EL}|} \text{size}\left(rq_k^{EL}[l]\right) + \sum_{j=EL-1}^{3} \sum_{l=1}^{|rq_k^j|} \text{size}\left(rq_k^j[l]\right) + s_p^i}{n} \tag{3.4} $$

$\forall\, rq_k[l]$: if n_k accepts s_p^i, then $rq_k[l]$ is violated.

Each node maintains a three-level priority queue named rq, sorted criticality in figure (3). Each priority level queue at a node is represented as ${rq_k}^{EL}$ where k indicates the node index, and EL refers to the criticality level of the queue. Microservice i is queued by priority, and its processing time based on (3.4) includes (i) remaining load in its priority queue, (ii) lower-priority tasks that would violate deadlines if postponed, and (iii) its own execution time. The overall objective is to minimize the total response time for all microservices in application p defined by

$$ \mathbb{F}_1 = \sum_{i=1}^{|S_p|} {R_i}^p \tag{3.5} $$

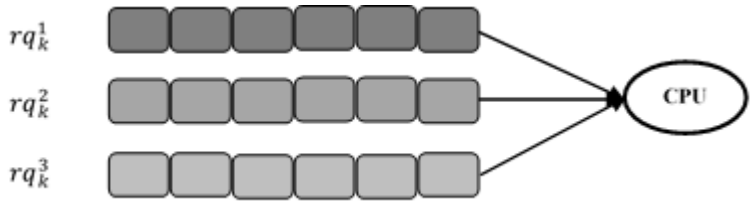


Figure 3: The queue with different priority levels on each computing node

3.2.2 load balancing

To ensure balanced load distribution, CPU and RAM usage are the key metrics in load calculation. For a node set M, load deviation is computed by

$$LD_{s_p^i} = \sqrt{\frac{1}{|M|}\sum_{m=1}^{|M|}\left(M_m^{\mathrm{cpu}_{req}} - n_j^{\mathrm{cpu}_{req}}\right)^2} + \sqrt{\frac{1}{|M|}\sum_{m=1}^{|M|}\left(M_m^{\mathrm{ram}_{req}} - n_j^{\mathrm{ram}_{req}}\right)^2} \tag{3.6}$$

where $M^{cpu_{req}}$ and $M^{ram_{req}}$ denote the average CPU and RAM demand, respectively, over subset M. Each node j has current request loads ${n_j}^{cpu_{req}}$ and ${n_j}^{ram_{req}}$ computed from its queue. All values are normalized to ensure comparability. The aim is to place microservices in a DAG such that node usage remains balanced. Thus, the placement minimizes:

$$\mathbb{F}_2 = \sum_{i=1}^{|s_p|} LD_{{s_p}^i}/|s_p| \tag{3.7}$$

A lower $\mathbb{F}_2$ implies better load distribution across nodes during placement.

3.2.3 Reliability

Requests criticality are defined at three Execution Levels ($EL \in \{1, 2, 3\}$) conveyed to the placement module along with the application's DAG.EL = 1 implies strict deadline constraints, necessitating high-reliability provisioning; EL = 2 permits partial relaxation; EL = 3 tolerates latency, allowing reactive fault tolerance. A reliability model based on the primary/backup scheme is applied by defining corresponding DAGs and constructing the RelaP matrix. For EL = 1, a full backup DAG is generated, and both DAGs are mapped across computing nodes. The placer constructs a $(2 \times K \times M)$ RelaP matrix, where K is the number of microservices and M is the number of candidate nodes. Matrix elements are derived based on whether the service is in the primary or backup DAG, as per (3.8) and (3.9), minimizing service disruption in case of failures (figure 4).

$$RelaP_p^{EL=1} = \left[X_{ij} \left(\frac{rl_j}{PD(s_p^i)} \; s_p^{\tau i} \; SD(s_p^i) \right)_{tt} \right], \quad \forall i \in |S_p + S_p^{\tau}|, \; j \in |M| \quad (3.8)$$

$$RelaP_p^{EL=1} = \left[X_{ij} \left(\frac{rl_j}{PD(s_p^{\tau i})} \; s_p^i \; SD(s_p^{\tau i}) \right)_{tt} \right], \quad \forall i \in |S_p + S_p^{\tau}|, \; j \in |M| \quad (3.9)$$

where, tt denotes inter-service transfer time,$s_p{}^{\tau i}$ is the backup instance of $s_p{}^i$, $X_{ij} \in \{0, 1\}$ encodes placement.

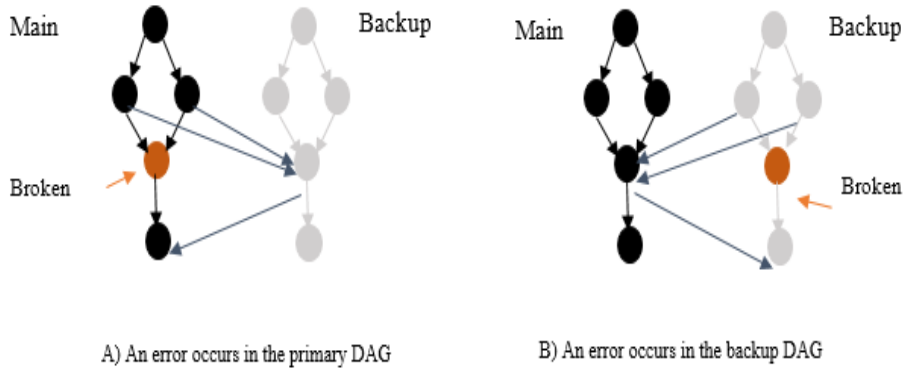


Figure 4: Relationship between services in the primary and backup DAG

As illustrated in figure (5), applications with EL = 1 require uninterrupted service continuity. Consequently, a full backup DAG is maintained alongside the primary. Upon failure of any microservice, the system seamlessly transitions to its backup while the service placer initiates re-deployment of the failed instance on a nearby node. RelaP matrix construction prioritizes both high reliability nodes and minimal inter-DAG communication latency, ensuring that failures in either DAG incur negligible performance degradation.

Applications with EL = 2 are classified as crisis-prone, warranting partial redundancy. While the RelaP matrix of size $2 \times K \times M$ is still utilized, backup instances are selectively provisioned. Specifically, after generating the physical plan, backups are created only for microservices deployed on nodes whose reliability falls below the mean reliability $\overline{rl_{DAG}}$ of all active nodes (see Figure 5). These backups are placed such that the inter-node communication delay, particularly with respect to the microservice's immediate predecessors and successors, is minimized—formalized in (3.10). Accordingly, RelaP entries are computed using (3.11): if a backup is required, reliability is normalized by the placement cost; otherwise, it is directly assigned. In cases without backup $X_{i,j}$ is equal to 0.

$$RelaP_p^{EL=2^+}\left(PD(s_p^i)\ s_p^{\tau i}\ SD(s_p^i)\right)_{tt},\quad \forall i \in DAG,\quad rl_i^j < \overline{rl_{DAG}} \tag{3.10}$$

$$RelaPM_p^{EL=2} = \begin{cases} [X_{i,j}\, rl_j], & \text{if no backup} \\ [X_{i,j}\, \frac{rl_j}{R_p{}^{EL=2^+}}], & \text{otherwise} \end{cases} \tag{3.11}$$

Let us assume that the average reliability of nodes in M at the time of placing the DAG in figure (6) is 0.5, since microservices 4 and 5 are deployed on nodes whose reliability is lower than the average reliability of the DAG, backup instances are created for these microservices simultaneously with the deployment of the physical plan to the computing continuum. The placer then selects nodes for these backup instances such that (3.10) is minimized.

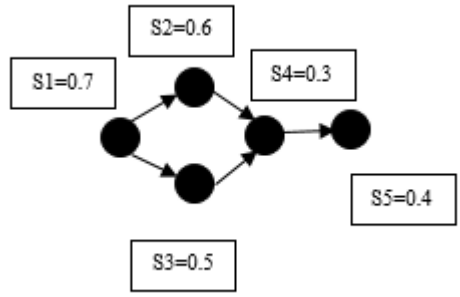


Figure 5: Host reliability of each microservice in a DAG

For applications with normal priority EL=3, there is no need to create backup copies for any of the DAG's microservices. In this case, simply computing the RelaP matrix using (3.12) is sufficient for placing the microservices across the computing continuum.

$$RelaP_p{}^{EL=3} = [X_{i,j}, rl_j], \forall i, j, i \in |S_p|, j \in |M| \tag{3.12}$$

To compute the reliability of each node, (3.13) is used, where λ_j is the failure rate of node j , TOT_j is the total operational time of node j, LS_j is the set of directly connected links from node j to its neighboring nodes λ_l and TOT_l are the failure rate and operational time of link l, respectively.

$$rl_j = e^{-\lambda_j\, ToT_j} \prod_{l \in LS_j} e^{-\lambda_j\, ToT_j},\quad j \in |M| \tag{3.13}$$

The computed RelaP matrix for a given application p based on its EL is summarized by

$$\mathbb{R}ela\mathbb{P}_P = \begin{cases} {RelaP_P}^{EL=1} & if El = 1 \\ {RelaP_P}^{EL=2} & if El = 2 \\ {RelaP_P}^{EL=3} & if El = 3 \end{cases} \tag{3.14}$$

where $\mathbb{R}ela\mathbb{P}_P$ is sum of the${RelaP_P}^{EL=2}$ entries. The goal is to maximize the reliability resulting from the placement of App_p expressed by

$$\mathbb{F}_3 = 1/\mathbb{R}ela\mathbb{P}_P \tag{3.15}$$

Finally, the overall objective function for placing the microservices of $App_p$is calculated using

$$\mathbb{F} = \alpha\mathbb{F}_1 + \beta\mathbb{F}_2 + \gamma\mathbb{F}_3 \tag{3.16}$$

where $\alpha + \beta + \gamma = 1$ and tuned by EL. For EL = 1, latency and reliability dominate. EL = 2 moderately increases the weight on load balancing due to partial backups. EL = 3 prioritizes load balancing, with reactive recovery mechanisms permitted.

4. Placement method

4.1 Reinforcement learning

In reinforcement learning, an agent interacts with the environment by taking actions, receiving rewards, and transitioning to new states based on environmental dynamics. This interaction, repeated over time, yields data used to improve the agent's policy. The policy is updated iteratively, enhancing behavior through continued interaction. The problem is formally defined as a Markov Decision Process (MDP), a five-tuple ¡ $\mathbb{S}, \mathbb{A}, \mathbb{P}, \mathbb{R}, \gamma >$ where $\mathbb{S}$ is the state space,$\mathbb{A}$, the action set, $\mathbb{P}$, the transition probability, $\mathbb{R}$, the reward function, and $\gamma \in [0, 1]$, the discount factor for future rewards. Let total learning time T be divided into steps. At each step t, the agent is in state $S_t = s$, selects action $A_t = \alpha$ based on policy $\pi(\alpha|s) = Pr[A_t = \alpha|S_t = s]$, receives reward $r = \mathbb{R}(s, a)$ and moves to next state $S_{t+1} = s'$ according to $\mathbb{P}(s'|s, a)$. The objective is to learn the optimal policy(π^*) that maximizes expected discounted return: $E_\pi[\sum_t \in T\gamma_t r_t]$ where γ balances immediate and future rewards.

4.2 Elements of used reinforcement learning methods

Based on the weighted objective function for placing the microservices of application p onto the computing continuum, the state space S, the action space $\mathbb{A}$ and the reward function $\mathbb{R}$ for MDP are defined as follows:

State Space ($\mathbb{S}$) :Since the problem focuses on the placement of microservices belonging to a given application, which are represented as a DAG, the state in this MDP corresponds to a combination of the set of candidate computing nodes N. Formally, the feature space of the current microservice ${s_p}^i$ at time step t is represented by the set:

$$\Psi_t(s_p^i) = \left\{ \Psi_t^y(s_p^i) \;\middle|\; 0 \leq y \leq |\Psi| \right\} \tag{4.17}$$

Where $\Psi_t\left({s_p}^i\right)$ denotes the $y-$ feature of ${s_p}^i$ and $|\Psi|$ is the total number of features in the feature set. This set may include, but is not limited to, the following attributes: The required resources for the microservice (e.g., CPU, RAM), the amount of input data needed from predecessor services in the DAG, the amount of output data to be sent to successor services, the data transfer delay between the current microservice and its backup version (if any). Together, these define the state of the environment with respect to the current placement decision.

Additionally, at time step t, for the set of computing nodes, several attributes can be considered as defining features of the state space. These include: the number of available nodes in N, the number of pending requests in the queues of each node, the processing and memory requirements of all pending requests in each node's queues, the CPU frequency, main memory, and storage capacity of each node and the available bandwidth between each node and its neighboring nodes. These features are collectively represented by the set:

$$\Phi_t(N) = \{|N|, {\Phi_t}^z\left(n_j, n_k\right) | n_j, n_k \in N, 0 \leq z, q \leq |\Phi|, z \in Z, q \in Q, Q \subset \Phi\} \tag{4.18}$$

Where,ϕ is the set of all possible features related to the computing nodes, Z represents the subset of features associated with individual nodes, such as CPU usage, available memory, etc., Q, represents the subset of features related to pairs of nodes, such as the bandwidth between two nodes, z and q are the feature indices for sets Z and Q, respectively. Therefore, the complete state space S is defined as:

Action Space($\mathbb{A}$):The goal is to find the near placement for the set of microservices of application p across the computing continuum, such that the objective function is minimized. Therefore, at time step t, an action corresponds to assigning a node $n_j \in N$ to the current microservice ${s_p}^i$.Formally, the action at time t can be defined as:

$$A_t = X_(s_p{}^i, n_j) = 1, n_j \in N, \tag{4.19}$$

Here,$X_(s_p{}^i, n_j)$ indicates that $s_p{}^i$ is assigned to n_j and $A = N$, i.e., the action space consists of selecting a node from the available set N Or a subset there of.

Reward Function $\mathbb{R}$: Due to the use of a weighted model for the objective function, it is necessary to define a separate reward function for each component of the objective. First, for the response time, the reward function is defined based on (4.20), in which $PD_{super}(s_p{}^i)$ represents all predecessor services of the current service that, either directly or indirectly, provide data required to $s_p{}^i$. If the placement of $s_p{}^i$ leads to a violation of the application p's deadline, D_p, the reward is calculated as the product of a constant value c and a penalty, where the penalty amount may vary depending on the priority level of the request.

$$r_t^{\mathbb{F}} = \begin{cases} D_p - \left(\sum_{k=1}^{|PD_{super}(s_p^i)|} R_p^k\right) + R_p^i, & \text{if } \geq 0 \\ -c \cdot \text{penalty}, & \text{otherwise} \end{cases} \tag{4.20}$$

To achieve appropriate load balancing, the reward function of (4.21) is used, such that if after placing $s_p{}^i$ the load balance becomes worse than the previous value, a penalty will be incurred for this decision.

$$r_t^{\mathbb{F}} = \begin{cases} LD_{s_p}^{i-1} - LD_{s_p}^i, & \text{if } \geq 0 \\ -c \cdot \text{penalty}, & \text{otherwise} \end{cases} \tag{4.21}$$

Also, in order to select nodes with appropriate reliability at each priority level, the reward function of (4.22) has been used.

$$r_t{}^{\mathbb{F}} = \begin{cases} \overline{RM_p}^i - \overline{RM_p}^{i-1} & if \geq 0 \\ -c * penalty & else \end{cases} \tag{4.22}$$

where $\overline{RM_p}^i$ and $\overline{RM_p}^{i-1}$ denote the average reliability of the nodes hosting services 1 through $i-1$, and services 1 through i, respectively. If the placement of the s_p^i results in a decrease in reliability, the reward function yields a negative value. Similarly, in (4.21) and (4.22), the penalty value can vary depending on the priority level of the service request. The final reward function corresponding to the objective function$\mathbb{F}$ is computed using (4.23), where ω_1 , ω_2 and ω_3 are control parameters. These parameters can be adjusted based on the priority level of the application.

$$r_t = \omega_1 \times r_t{}^{\mathbb{F}_1} + \omega_2 \times r_t{}^{\mathbb{F}_2} + \omega_3 \times r_t{}^{\mathbb{F}_3} \tag{4.23}$$

4.3 Deep RL Placement: Clipped PPO with transfer learning

Among deep reinforcement learning methods, Proximal Policy Optimization (PPO) stands out for its efficient convergence and sampling, thanks to importance sampling and adaptive clipping. PPO suits dynamic, discrete environments like computational grids, making it ideal for placing critical medical services. As a Policy Gradient (PG) method, PPO adjusts action probabilities based on their reward impact, optimizing the policy π_Θ

$$J(\Theta) = \mathbb{E}_{\pi_\Theta}\left[\sum y_t r_t\right] \tag{4.24}$$

Since the objective is to maximize the reward, we use the gradient ascent method to find the maximum value based on:

$$\theta^{'} = \theta + \alpha \nabla_\theta J(\theta) \tag{4.25}$$

The objective is to compute the gradient of $J(\theta)$ with respect to θ, which is referred to as the Policy Gradient algorithm. The policy gradient is represented by:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\Big[\nabla_\theta \log \pi_\theta(a_t \mid s_t)\ A_\theta(a_t \mid s_t)\Big] \tag{4.26}$$

Where $A_\theta\,(a_t|s_t)$ is the advantage function at time t, used to evaluate the a_t in s_t. According to (4.26), the expectation $[\nabla_\theta log \pi_\theta(\,a_t|s_t\,)A_\theta\,(a_t|s_t)]$ can be estimated using the empirical mean obtained from sampling trajectories. However, policy gradient based algorithms are highly sensitive to the update step size. Therefore, choosing an appropriate step size poses a significant challenge. Moreover, in practice, it has been observed that the difference between the old and new policies during training in such methods can often be substantial.
One of the effective ideas in creating a limit on the difference between $\pi_{\theta_{old}}$ and $\pi_{\theta_{new}}$ is to use the clip function, which is called CLIP-PPO. Formally, the CLIP-PPO objective function is defined by:

$$L^{CLIP}\,(\theta) = \mathbb{E}_t\left[min\left(r_t\,(\theta)\,A_t, clip\left(r_t\,(\theta)\,, 1-\varepsilon, 1+\varepsilon\right)A_t\right)\right] \tag{4.27}$$

Where

$$clip\left(r_t\,(\theta)\,, 1-\varepsilon, 1+\varepsilon\right) = \begin{cases} 1-\varepsilon & r_t\,(\theta) < 1-\varepsilon \\ r_t\,(\theta) & 1-\varepsilon \le r_t\,(\theta) \le 1+\varepsilon \\ 1+\varepsilon & r_t\,(\theta) \le 1-\varepsilon \end{cases} \tag{4.28}$$

PPO is an actor-critic algorithm combining policy gradient and temporal difference (TD) learning, both implemented using deep neural networks. The actor learns the policy to maximize cumulative rewards, while the critic evaluates the policy

and informs action updates. At time step t, the agent feeds state s_t to the actor. Then the actor generate action a_t, receives reward r_t , and moves to state s_{t+1}.The critic estimates $V_{\pi\theta}(s_t)$ and $V_{\pi\theta}(s_{t+1}))$, and the agent calculates the TD error using

$$\delta_t = r_t + \gamma V_{\pi\theta}(s_{t+1}) - V_{\pi\theta}(s_t). \tag{4.29}$$

An estimate of A_t can also be calculated by:

$$\hat{A}_t = r_t + \gamma r_{t+1} + \cdots + \gamma^{T-t-1} r_{T-1} + \gamma^{T-t} V_{\pi_\theta}(s_T) - V_{\pi_\theta}(s_t). \tag{4.30}$$

To speed up service placement, transfer learning can be used by leveraging knowledge from similar tasks (Ebrahim *et al.* , 2025). A θ-buffer is maintained at the placement node, storing a θ parameter for each application priority level, computed using:

$$\theta_{EL} = \alpha {\theta_{EL}}^{preLearn} + (1-\alpha)\, {\theta_{EL}}^{oldLearn} \tag{4.31}$$

To build a physical plan, the decision-maker reuses prior knowledge from placements at the same priority level. It initializes θ using both the latest learned parameter $\left(\theta^{preLearn}\right)$ and the average of all past θ values $\left(\theta^{oldLearn}\right)$to reflect accumulated experience. Algorithm 1 is responsible for placing the microservices of application p, including both the main and backup instances, if applicable.

Algorithm 4.1.

- ***Input:*** $App_p = \{DAG_p, EL_p\},\ M$
- ***Output:*** PP_p *//Physical placement of* App_p
- $\theta_{EL_p} \leftarrow \theta_{buffer}(EL_p)$ *Initialize policy parameters* ϑ_1 *and value parameters* φ_1 *using* θ_{EL_p}
- $k = 0, 1, 2, \ldots$
  Collect χ *trajectories* $D_k = \{\tau^i\}_{i=1}^{x}$ *using policy* π_{ϑ_k}
- *Compute rewards* r_t
- *Compute advantage estimates* $\hat{A}_t$ *using value function* V_{φ_k}
- *Update policy:*

$$\vartheta_{k+1} = \arg\max_{\vartheta} \frac{1}{xT} \sum_{\tau \in D_k} \sum_{t=0}^{T} \min\left(\frac{\pi_\vartheta(a_t \mid s_t)}{\pi_{\vartheta_k}(a_t \mid s_t)} \hat{A}_t, g(\epsilon, \hat{A}_t) \right)$$

- *Update value function:*

$$\varphi_{k+1} = \arg\min_{\varphi} \frac{1}{xT} \sum_{\tau \in D_k} \sum_{t=0}^{T} \left(V_\varphi(s_t) - r_t\right)^2$$

- $PP_p \leftarrow$ *Trained Policy Network*(DAG_p)
- *Dispatcher* $\leftarrow PP_p$
- *Update buffer:* $\theta_{buffer}(EL) \leftarrow \theta_{buffer}(EL) \cup (\vartheta_{k+1}, \varphi_{k+1})$

Once an application's DAG and backups are placed, the placement and node status are sent to the edge node. The IoT device then routes requests through the edge, cutting ties with the placer. If no response is received within $\epsilon \ll deadline_p$, the edge redirects the request to a backup (if available). Failures are reported back to the placer, which initiates recovery. This process is detailed in Algorithm 2.

Algorithm 4.2. ***Input:*** *DAG_p request*
Output: *Action*
Send DAG_p to hosts(DAG_p)
each $s_k \in s_p$
Ack_k not received
Send s_k to backup host(s_k)
Send the event to the Placer Node

5. Performance evaluation

This section evaluates the effectiveness of the proposed intelligent service placement framework under realistic and dynamic conditions. The assessment is conducted using a synthetic microservice-based application modeled as a DAG, enabling controlled analysis of placement behavior across varying service dependencies and resource demands. The computational characteristics of the microservices, system configuration parameters, and learning hyperparameters are first described. Subsequently, the simulation environment is introduced, followed by a detailed discussion of the experimental results and comparative performance analysis.

5.1 Application Case Study-Random DAG-Based Microservice Application

To systematically evaluate the proposed placement strategy without bias toward a specific application domain, a randomly generated microservice-based application is employed as the case study. The application is represented as a DAG, where nodes correspond to individual microservices and edges denote data and execution dependencies. This abstraction allows the evaluation of placement decisions under diverse execution paths, parallelism levels, and dependency structures commonly observed in real-world continuum applications. The DAG is generated using a

random graph construction process that, while varying the number of microservices, also ensures a variety of dependencies between them. Each microservice is assigned heterogeneous computational requirements in terms of CPU instructions, memory footprint, and storage demand, reflecting realistic variability in service workloads. The resulting DAG consists of eight interacting microservices, providing sufficient complexity to assess response time, load distribution, and reliability-aware placement decisions. Table (3) summarizes the computational requirements of the constituent microservices.

Table 3: Computing requirements for application microservices

service	CPU (M)	Memory (MB)	Storage (MB)
Microservices i	[200-1000]	[256-1000]	[500-1000]

5.2 implementation details

The simulation was conducted using Python along with the Torch and NetworkX libraries, all executed on Google Colab. The computing continuum used for the simulation is illustrated in Figure 6. The placement services are located in the first fog layer, which is closer to the cloud layer. These services are continuously backed up in the cloud, and in case of necessity, the backup is transferred to another node within the same layer. This study does not address the problem of selecting a host for the placement service. It is assumed that the cloud node selects the host nodes in the first fog layer based on its own predefined criteria.The simulation is conducted with one cloud data center, two fog clusters—each consisting of 25 nodes, and one edge node designed to support up to 30 users. Users can dynamically enter or leave the continuum at any time. Tables (4) and (5) respectively present the characteristics of the simulation environment, while Table 6 lists the hyper parameters used in the proposed algorithm. Hyper parameter values for the deep learning algorithm in service placement has been adopted from Wang *et al.* (2024).

Table 4: Computational capacities of nodes in different layers of the desired computing continuum

Level	CPU (MIPS)	RAM (MB)	Storage (GB)
0-Cloud datacenter	Unlimited	Unlimited	Unlimited
1-Fog cluster nodes	4000	25000	2000
2-Fog cluster nodes	2800	6000	1000
Edge node	2000	4000	500

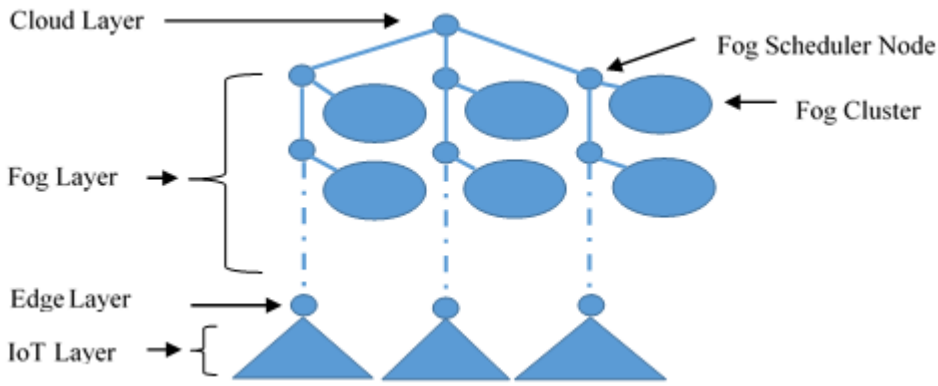


Figure 6: The desired computing continuum for simulation

Table 5: Communication delay between nodes in the computing continuum

Network Link	Latency (ms)
Cloud-Fog cluster 1	3000
Fog cluster1-Fog cluster2	1000
Fog cluster 2-Edge	500
Edge-IoT	50

To evaluate the performance of the proposed algorithm under node failure conditions, the initial failure rate of each node is randomly assigned using a uniform distribution within the range $[0.01, 0.08]$. This range ensures that even relatively healthy nodes experience multiple failures during the execution of the maximum number of requests. During the simulation, each node's failure rate is updated dynamically. In the event of a failure, the failure rate of node j increases by 0.005. If a request is successfully executed, the failure rate decreases by the same amount. The minimum failure rate is 0.01, and the maximum is 0.08. To assess the efficiency of the proposed method, we compare it not only with basic algorithms such as Random Placement and First-Fit, but also with a Genetic Algorithm-based approach (GA) (Sarrafzade *et al.* , 2022) and the DRLIS method (Wang *et al.* , 2024), which are state-of-the-art methods reviewed in the related works. The evaluation is carried out based on various workload volumes, as defined in Table 7, for end users. The deadline constraint is also set as follows: A maximum of 40 seconds for critical users, 60 seconds for potentially critical users, and 120 seconds for normal users. The figures in the following section illustrate the performance of the compared methods based on various parameters. Each chart reports the average of 30 simulation runs for each number of end users.

Table 6: Hyperparameter values for deep reinforcement learning-based placement service

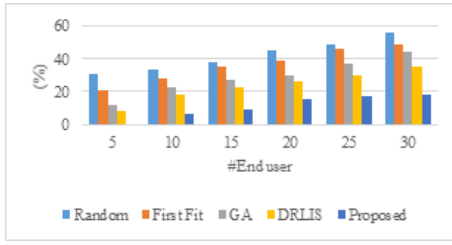
Hyperparameters	**Value**
Epoch	500
Neural network layers	3
Hidden layer units	64
Optimization method	Adam
Activation function	ReLU
Discount factor	0.9
Actor learning rate	0.0003
Critic learning rate	0.001
Policy objective function coefficient	1
Value function loss coefficient	0.5
Entropy bonus coefficient	0.01
Reward coefficients for critical services	Response time = 0.6 Load balancing = 0.1 Reliability = 0.3
Reward coefficients for potential critical services	Response time = 0.5 Load balancing = 0.2 Reliability = 0.3
Reward coefficients for normal services	Response time = 0.3 Load balancing = 0.6 Reliability = 0.1

Table 7: Different load volumes for evaluation

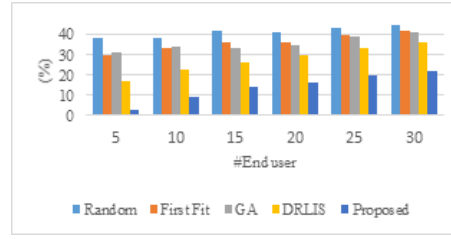
loadsize	Max request per and user	Total requests
Low	30	900
Medium	60	1800
Heavy	90	2700

5.3 Experimental Result

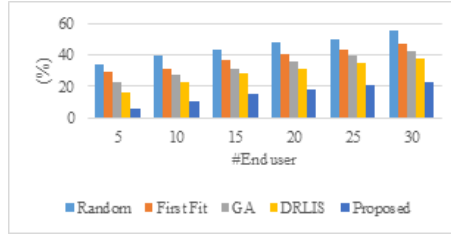
In this section, the proposed method is compared with the aforementioned methods based on several parameters and the results are analyzed.



(a) low load



(b) medium load



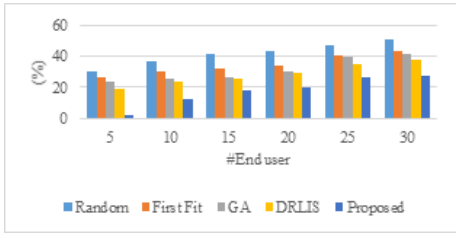
(c) heavy load

Figure 7: Critical deadline violation

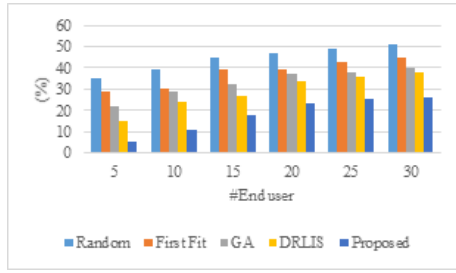
5.3.1 Deadline violation

Figures (7) and (8) illustrate the rate of missed deadlines of user requests under various workloads. Delayed response or service breakdown in the continuum is the major source of deadline violations. Figure (7) account for critical-priority requests, and figure (8) account for potentially critical requests; normal-priority requests are excluded since their deadlines are loose and the violation rate is very low. The approach consistently beats other methods, especially with tight timing constraints, due to its accurate real-time condition awareness and adaptive node

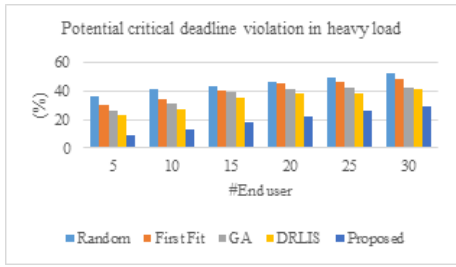
ordering. This increases robustness to both delay- and failure-induced violations, particularly in time-critical scenarios.



(a) low load



(b) medium load



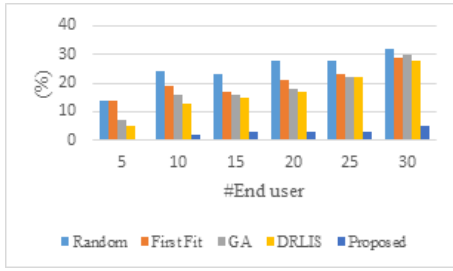
(c) heavy load

Figure 8: Potential critical deadline violation

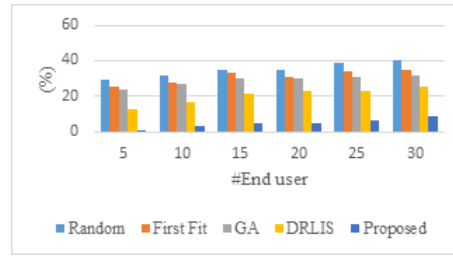
5.3.2 Deadline violation by fault

Due to the importance of generating timely results for user requests, the proposed method employs backup creation techniques to handle failures effectively. Figures (9) and (10) illustrate the performance of the compared methods in response to failures within the computing continuum. These figures specifically depict the number of violated requests caused by failure events, limited to critical and potentially critical priority levels. For normal priority requests, failures can typically be handled using reactive techniques due to more relaxed deadlines; therefore, such requests are not included in the presented graphs. In Figure (9), the proposed method clearly outperforms the compared methods in handling failures, primarily because it creates backup instances for all microservices, ensuring service continuity even in the event of node failures.

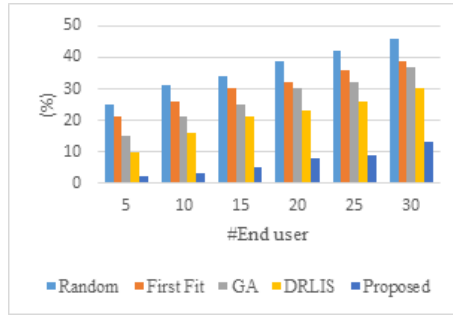
A second notable point from these figures is that the proposed method prioritizes nodes with higher reliability for hosting critical requests. This results in fewer failures occurring for these nodes, and consequently, a smaller percentage of violated critical requests. Figure (10) focus on potentially critical requests, for which



(a) low load



(b) medium load



(c) heavy load

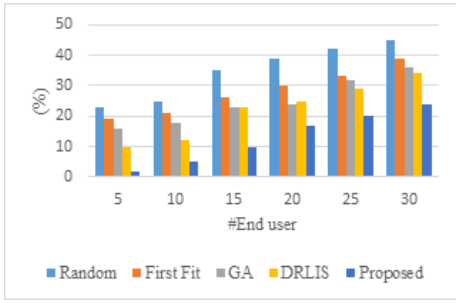
Figure 9: Critical deadline violation by fault

the proposed method only creates backups for selected microservices. Therefore, while the reduction in failure-related violations is less pronounced than for critical microservices, the method still achieves a significant improvement compared to other approaches. Overall, the proposed method effectively reduces the impact of failures, especially for high-priority user demands.

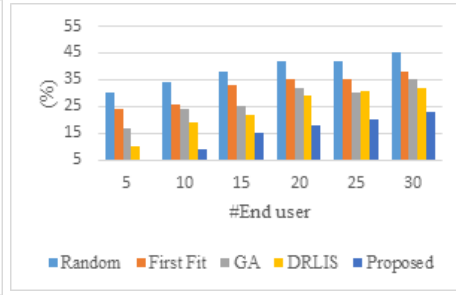
5.3.3 Node reliability index

To minimize the number of deadline violations caused by failures, selecting hosts with higher reliability is of critical importance. The proposed method incorporates this factor into its service placement decisions across various request priority levels. For critical-priority requests, this metric is given greater influence in the placer agent's decision-making process. Although the impact of reliability decreases with lower-priority requests, it is not ignored even for normal-priority demands during scheduling.

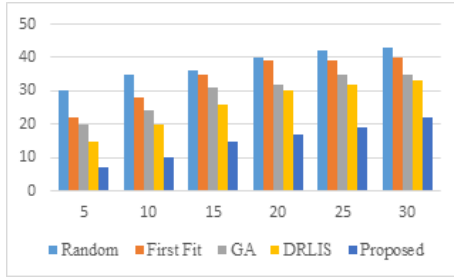
The effectiveness of the proposed method in selecting suitable hosts for each request type under varying workload volumes is demonstrated in Figures (11), (12) and (13). In Figure (11), which correspond to critical-priority requests across different numbers of end users, the proposed method consistently maintains higher



(a) low load



(b) medium load



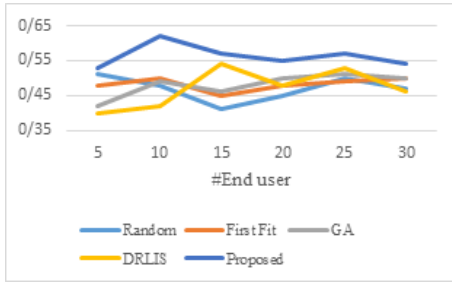
(c) heavy load

Figure 10: potential critical violation by fault

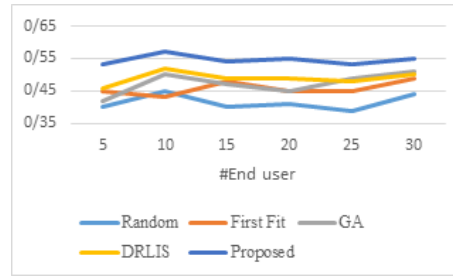
average host reliability compared to other approaches. This is due to the greater weight assigned to reliability when placing critical services, which is essential for ensuring service continuity. In Figure (12), which show results for potentially critical requests, the influence of host reliability on placement decisions is lower. As a result, the reliability values for selected hosts are generally better but at certain points closer to those of other methods. Finally, in Figure (13), which present results for normal-priority requests under various workload levels, the influence of host reliability is minimal. While some other methods may occasionally select more reliable hosts, the proposed approach still demonstrates solid performance, showing that even with reduced emphasis on reliability, its overall decision-making strategy remains effective.

5.3.4 Load balancing

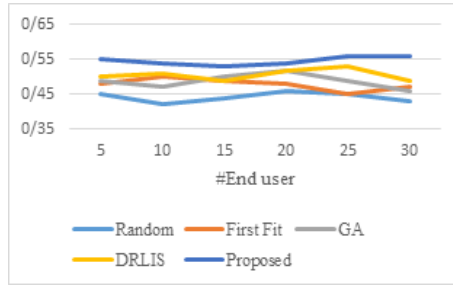
Another influential parameter in the performance of a computing continuum is load balancing across its nodes. However, this balance is not equally critical for all application types. Certain applications—especially those composed of a set of microservices—require low-latency responses, which is best achieved when the microservices are placed on the same node or on nodes located close to each other,



(a) low load

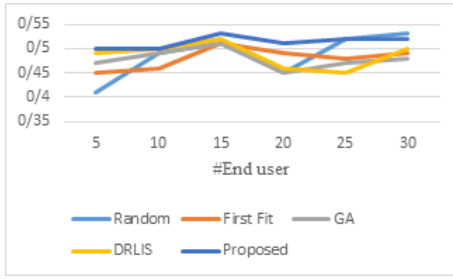


(b) medium load

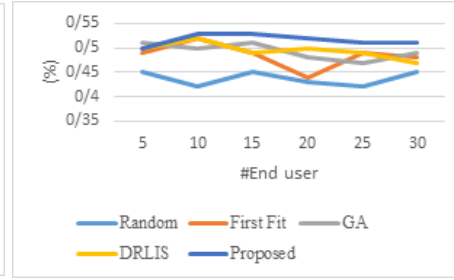


(c) heavy load

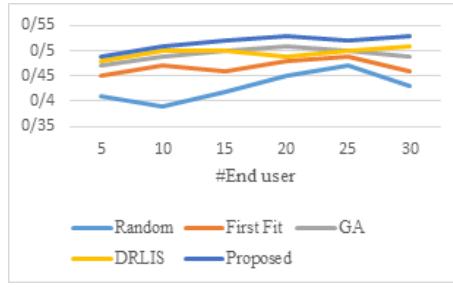
Figure 11: Critical reliability index



(a) low load

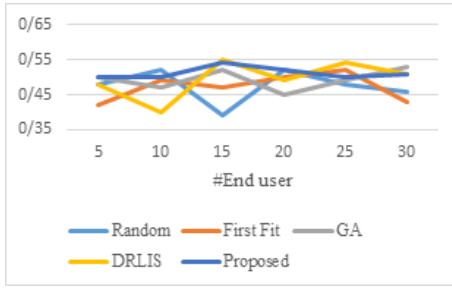


(b) medium load

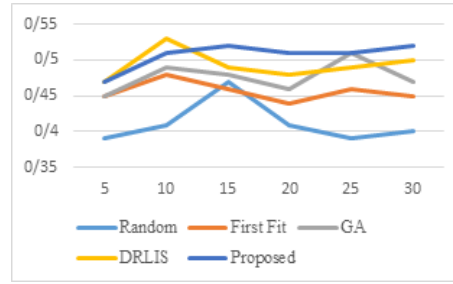


(c) heavy load

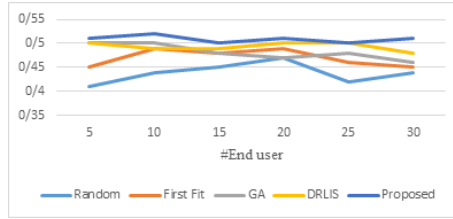
Figure 12: Potential critical reliability index



(a) low load



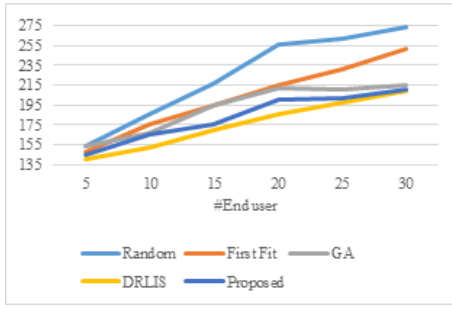
(b) medium load



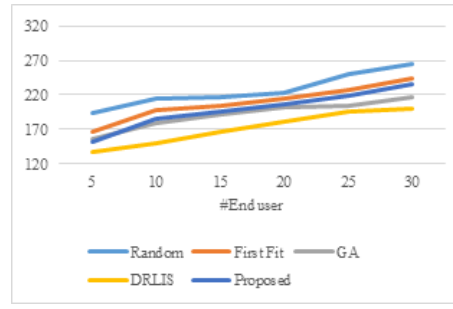
(c) heavy load

Figure 13: Normal reliability index

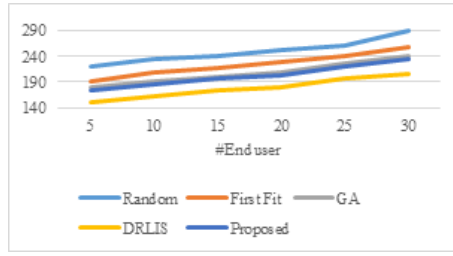
and as near as possible to the end user. The proposed method takes this requirement into account. Therefore, the influence of load balancing in placement decisions is adjusted based on the response time sensitivity of the application.
For applications handling critical requests, the placer minimizes consideration of load distribution, prioritizing fast response over balanced resource use. Conversely, for applications with normal-priority demands, load balancing receives the highest importance in the placement decision process. Figure (14) illustrate CPU load balancing across different nodes for all application priority levels under various workload intensities. Meanwhile, figure (15) depict the overall memory load balancing across the computing continuum for all application types combined. Due to the priority-aware consideration of load balancing in the proposed method, the results show intermediate behavior across different workload levels when compared to other methods. The values presented in figures (14) and (15) represent the standard deviation of the respective resource usage over time, which serves as a measure of load balance quality across the continuum.



(a) low load



(b) medium load

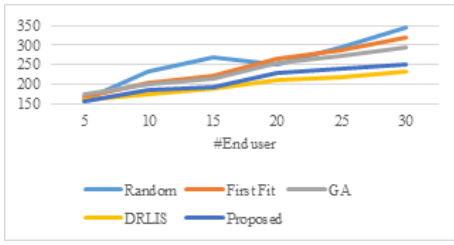


(c) heavy load

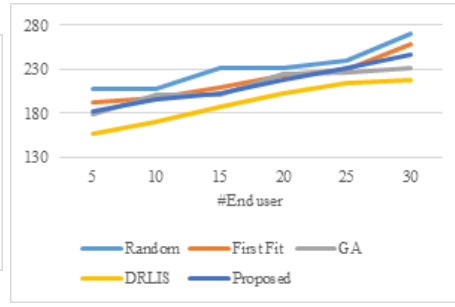
Figure 14: CPU balancing

6. Conclusion

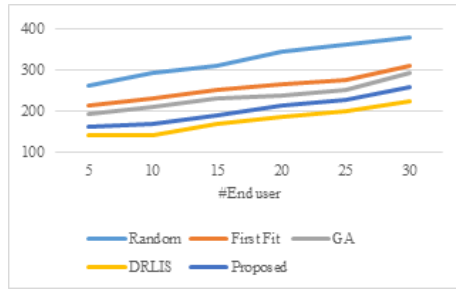
In this study, a method based on a deep reinforcement learning algorithm was proposed for the placement of services in mission-critical applications across the cloud-to-edge computing continuum. Various priority types of services in such applications were identified. The influencing parameters for ensuring the quality of these services were modeled and utilized within a weighted objective function. A placement service, located in the upper layers of the continuum, selects appropriate nodes based on the priority of each service. This placement service assigns specific weights to the influencing parameters according to the input application's priority and determines the host nodes for placing the application's microservices using the PPO (Proximal Policy Optimization) algorithm. To accelerate and improve the accuracy of decision-making, the scheduler also leverages prior knowledge through transfer learning. Following the placement of microservices for an application, the placement service enhances fault tolerance during service delivery using a primary-backup approach. Based on the application's priority, a number of backup instances are created and deployed on specific nodes—selected according to the influencing parameters—so that in case of failure of the primary instance, service delivery can continue seamlessly via the backup.



(a) medium load



(b) medium load



(c) heavy load

Figure 15: Memory balancing

For future work, by using user's context data and predicting their future scenarios can reserve resources in advance at specific times before users arrive in a certain area—especially for high-priority users—enabling faster and more reliable decision-making. Additionally, for certain critical services, multiple backup instances can be deployed across different layers based on resource availability, increasing fault tolerance. Moreover, by incorporating additional parameters such as energy, it is possible to manage energy consumption of edge devices and continuum nodes to improve service availability over time.

Declaration

Funding

No funds, grants, or other support was received.

Competing Interests

The authors declare that they have no competing interests.

Data Availability

Due to the nature of this research, supporting data is not publicly available.

References

Apat, H. K., Sahoo, B., Goswami, V., and Barik, R. K. (2024), A hybrid metaheuristic algorithm for multi-objective IoT service placement in fog computing environment, *Decision Analytics Journal*, **10**, 100379.

Aslanpour, M. S., Gill, S. S., and Toosi, A. N. (2020), Performance evaluation metrics for cloud, fog and edge computing: A review, taxonomy, benchmarks and standards for future research, *Internet of Things*, **12**, 100273.

Chauhan, D., Jain, J. K., and Singh, A. (2024), Deployment of Edge Computing for Smart Healthcare Systems on Cloud Computing Platform, *International Journal of Scientific Research in Computer Science Engineering and Information Technology*, **10**, 685-693

Dogani, J., Yazdanpanah, A., Zare, A., and Khunjush, F. (2024), two-tier multi-objective service placement in container-based fog-cloud computing platforms, *Cluster Computing*, **27**(4), 4491-514.

Ebrahim, M., Hafid, A., and Abid, M. R. (2025), Enhancing fog load balancing through lifelong transfer learning of reinforcement learning agents, *Computer Communications*, **231**, 108024.

Ghasemi, A. (2024), MOHHO: multi-objective Harris hawks optimization algorithm for service placement in fog computing, *The Journal of Supercomputing*, **80** (17), 25004-25028.

Ghobaei-Arani, M., and Shahidinejad, A. (2022), A cost-efficient IoT service placement approach using whale optimization algorithm in fog computing environment, *Expert Systems with Applications*, **200**, 117012.

Hedhli, A., and Mezni, H. (2021), A survey of service placement in cloud environments, *Journal of Grid Computing*, **19**, 23.

Hennebelle, A., Dieng, Q., Ismail, L., and Buyya, R. (2024), martEdge: Smart Healthcare End-to-End Integrated Edge and Cloud Computing System for Diabetes Prediction Enabled by Ensemble Machine Learning, *In 2024 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, 127-134.

Herrera, J. L., Moya, A., Berrocal, J., Murillo, J. M., and Navarro, E. (2025), A Developer-Focused Genetic Algorithm for IoT Application Placement in the Computing Continuum, *IEEE Transactions on Services Computing*, **18**(3), 1185-1198.

Ibrahim, M. A., and Askar, S. (2023), An Intelligent Scheduling Strategy in Fog Computing System Based on Multi-Objective Deep Reinforcement Learning Algorithm, *IEEE Access*, **11**, 133607-133622.

Islam, S., Ahammed, M., Siddique, N. A., Roy, P., Razzaque, M. A., Hassan, M. M., and Saleem, K. (2024), A Hyper-Heuristic Approach for Quality of Experience Aware Service Placement Scheme in 5G Mobile Edge Computing, *IEEE Access*, **12**, 72746-72765.

Keshavarz Haddadha, P., Rezvani, M. H., MollaMotalebi, M., and Shankar, A. (2024), Machine learning methods for service placement: a systematic review, *Artificial Intelligence Review*, **57**(3), 61.

Lin, Y., Shi, Y., and Mohammadnezhad, N. (2024), Optimized dynamic service placement for enhanced scheduling in fog-edge computing environments, *Sustainable Computing: Informatics and Systems*, **44**, 101037.

Liu, T., Ni, S., Li, X., Zhu, Y., Kong, L., and Yang, Y. (2022), Deep reinforcement learning based approach for online service placement and computation resource allocation in edge computing, *IEEE Transactions on Mobile Computing*, **22**, 3870-3881

Lu, S., Wu, J., Shi, J., Lu, P., Fang, J., and Liu, H. (2022), A dynamic service placement based on deep reinforcement learning in mobile edge computing, *Network*, **2**, 106-122.

Malazi, H. T., Chaudhry, S. R., Kazmi, A., Palade, A., Cabrera, C., White, G., and Clarke, S. (2022),Dynamic service placement in multi-access edge computing: A systematic literature review, *IEEE Access*, **10**, 32639-32688.

Manihar, S., Patel, R., and Agrawal, S. (2023), A survey on mission critical task placement and resource utilization methods in the IoT fog-cloud environment, *Recent Trends in Computational Sciences*, 284-290.

Natesha, B., and Guddeti, R. M. R. (2021), Adopting elitism-based Genetic Algorithm for minimizing multi-objective problems of IoT service placement in fog computing environment, *Journal of Network and Computer Applications*, **178**, 102972.

Ocampo, A. F., and Santos, J. (2024), Reinforcement Learning-Driven Service Placement in 6G Networks across the Compute Continuum, *In 2024 20th International Conference on Network and Service Management (CNSM)*, 1-9, IEEE.

Pallewatta, S., Kostakos, V., and Buyya, R. (2024), Reliability-Aware Proactive Placement of Microservices-Based IoT Applications in Fog Computing Environments, *IEEE Transactions on Mobile Computing*, **23**, 11326-11341.

Patwary, M., Ramchandran, P., Tibrewala, S., Lala, T., Kautz, F., Coronado, E., Riggio, R., Ganugapati, S., Ranganathan, S., and Liu, L. (2022), Edge Services and Automation, *In 2022 IEEE Future Networks World Forum (FNWF)*,1-49, IEEE.

Qin, M., Li, M., and Othman Yahya, R. (2023), Dynamic IoT service placement based on shared parallel architecture in fog-cloud computing, *Internet of Things*, **23**, 100856.

Ramezani Shahidani, F., Ghasemi, A., Toroghi Haghighat, A., and Keshavarzi, A. (2023), Task scheduling in edge-fog-cloud architecture: a multi-objective load balancing approach using reinforcement learning algorithm, *Computing*, **105**, 1337-1359.

Saadian, F., Motameni, H., and Golsorkhtabaramiri, M. (2023), Deadline-aware multi-objective IoT services placement optimization in fog environment using parallel FFD-genetic algorithm, *Pervasive and Mobile Computing*, **92**, 101800.

Salaht, F. A., Desprez, F., and Lebre, A. (2020), An overview of service placement problem in fog and edge computing, *ACM Computing Surveys (CSUR)*, **53**, 1-35.

Sandhu, R., Boppana, R., Krishnan, R., Reich, J., Wolff, T., and Zachry, J. (2010), Towards a discipline of mission-aware cloud computing, *In Proceedings of the 2010 ACM Workshop on Cloud Computing Security Workshop*, 13-18.

Sarrafzade, N., Entezari-Maleki, R., and Sousa, L. (2022), A genetic-based approach for service placement in fog computing, *The Journal of Supercomputing*, **78**, 10854-10875.

Sharma, A., and Thangaraj, V. (2024), Intelligent service placement algorithm based on DDQN and prioritized experience replay in IoT-Fog computing environment, *Internet of Things*, **25**, 101112.

Siyadatzadeh, R., Mehrafrooz, F., Ansari, M., Safaei, B., Shafique, M., Henkel, J., and Ejlali, A. (2023), Relief: A Reinforcement-Learning-Based Real-Time Task Assignment Strategy in Emerging Fault-Tolerant Fog Computing, *IEEE Internet of Things Journal*, **10**, 10752-10763.

Tripathi, D.R., and Roshan Ramdas Markhande, H. T. D. (2024), Edge-Cloud Continuum: Integrating Edge Computing and Cloud Computing for IoT Applications, *International Journal for Research in Applied Science and Engineering Technology*, **12**(10), 335-342.

Tsolkas, D., Charsmiadis, A.-S., Xenakis, D., and Merakos, L. (2022), Service and network function placement in the edge-cloud continuum, *In 2022 IEEE Conference on Standards for Communications and Networking (CSCN)*, 188-193, IEEE.

Vivó, S., Lera, I., and Guerrero, C. (2024), Comparing Evolutionary Optimization Algorithms for the Fog Service Placement Problem, *In Proceedings of the IEEE/ACM 16th International Conference on Utility and Cloud Computing, Article 57, Taormina (Messina)*, Italy: Association for Computing Machinery.

Wang, Z., Goudarzi, M., Gong, M., and Buyya, R. (2024), Deep Reinforcement Learning-based scheduling for optimizing system load and response time in edge and fog computing environments, *Future Generation Computer Systems*, **152**, 55-69.