The Journal of Supercomputing (2025) 81:708
https://doi.org/10.1007/s11227-025-07196-9

A deep-reinforcement-learning-based strategy selection approach for fault-tolerant offloading of delay-sensitive tasks in vehicular edge-cloud computing

Vahide Babaiyan[1] · **Omid Bushehrian**[1]

Accepted: 13 March 2025

Abstract
Given the high resource demands of delay-sensitive tasks in vehicular networks, task offloading techniques have become prevalent in Mobile Edge-Cloud Computing (MECC) systems to reduce task completion times. Numerous studies have explored and addressed the task offloading problem in dynamic and unpredictable MECC environments using various Deep Reinforcement Learning (DRL) approaches. However, the critical issue of fault-tolerant (FT) task offloading in vehicular networks has not been comprehensively addressed. Failures of offloaded tasks to MECC nodes can significantly impact the quality of service in vehicular networks and potentially lead to catastrophic outcomes in critical vehicular tasks. To address this challenge, this paper proposes a DRL-based FT task offloading method for MECC environments to minimize the average response time and latency of all tasks under faulty conditions. An analytical model of the optimization problem is developed, followed by a Deep Deterministic Policy Gradient (DDPG) algorithm to determine the optimal deployment and recovery patterns for delay-sensitive tasks. The actor-critic architecture of DDPG, where the actor determines the task execution plans (including primary/backup nodes and the optimal recovery strategy) based on the current state, and the critic evaluates the decision, allows DDPG to adapt more smoothly to dynamic, continuous environments like vehicular networks, where the system's state, available resources, and failure rates are constantly changing. Our results show that by implementing diverse failure recovery strategies in high failure environments, our proposed method can reduce the total task completion time and the number of failures by an average of 29% and 78%, respectively, compared to baseline methods. Furthermore, the method's adaptability to changes in available computational resources and varying failure rates is clearly demonstrated.

Keywords Mobile edge-cloud computing · Fault-tolerant task offloading · Recovery pattern · Deep reinforcement learning

Extended author information available on the last page of the article

Published online: 08 April 2025

1 Introduction

The Mobile Edge-Cloud Computing (MECC) [1] architecture is an integration of Mobile Cloud Computing (MCC) [2, 3] and Mobile Edge Computing (MEC) [4, 5] architectures, enabling mobile devices to offload their computationally intensive tasks to the edge or cloud nodes based on their Quality of Service (QoS) requirements and resource demands. MECC benefits from the high computational capacity and reliability of the cloud servers as well as the low-latency of the edge resources. Although the MECC architecture mitigates the long latencies imposed on offloaded tasks by the cloud computing model, edge nodes in MECC environments remain susceptible to various failures due to their distributed and open structures [6, 7]. These failures occur as a result of software, hardware, or network faults and might be long-lasting, transient, or intermittent. Transient failures are temporary disruptions that do not damage the underlying hardware or software. They often result from external factors or network congestion. Examples include software/OS glitches, network congestion, and intermittent power outages. In contrast, permanent failures are more severe issues requiring hardware or software repairs. They can be caused by physical damage, software corruption, or hardware failures. Hence, Fault-Tolerance (FT) techniques and failure recovery strategies need to be part of any vehicular task offloading solution in MECC architecture, particularly when the offloaded tasks are critical and delay-sensitive and require reliable execution. For instance, consider a scenario where a vehicle offloads a deep learning-based obstacle detection task to an edge server. If the required software library is missing from the selected server, task execution might temporarily fail. In such cases, the system can seamlessly redirect the task to a backup edge server equipped with the necessary dependencies, ensuring uninterrupted operation. Similarly, a vehicle may request real-time traffic data for optimal route planning. If the designated edge server lacks the latest traffic updates due to delayed synchronization, the task can be dynamically reassigned to another node with up-to-date information. These examples illustrate how transient failures in MECC environments can be effectively mitigated through fault-tolerant mechanisms, maintaining the reliability and responsiveness of vehicular applications.

Given that the MECC paradigm defines a highly dynamic and large-scale environment, machine learning models can play a crucial role in adaptively and efficiently suggesting optimal task offloading plans compared to conventional optimization solutions [8–10]. In this regard, several approaches have employed Deep Reinforcement Learning (DRL) techniques, such as Deep Q-learning (DQL) and its variants, including Double Deep Q-Networks (DDQN) [11], to solve the optimal task offloading problem [12, 13].

Other methods, like meta-reinforcement learning frameworks, were also explored in [14], where a sequence-2-sequence algorithm for multi-task offloading in MECC, using Gated Recurrent Units (GRU) and attention mechanisms, was designed to improve task execution times.

Additionally, some studies leveraged Federated Deep Q-learning and Q-learning, respectively, to address task offloading and FT in Fog Computing (FC) environments,

optimizing task reliability and energy efficiency [15, 16]. These deep learning-based methods have advanced task offloading by integrating dynamic learning capabilities, providing scalable and adaptive solutions for complex vehicular and edge-cloud computing systems. In addition to deep learning-based approaches, meta-heuristic algorithms have been widely studied for optimizing task offloading in cloud-edge environments. Evolutionary and swarm intelligence methods, such as a hybrid Particle Swarm Optimization (PSO) and Genetic Algorithm (GA) approach in [17], balance task execution between cloud and edge nodes to optimize delay and energy. Heuristic strategies like Golden Tortoise Optimization (GTO) and Smallest Step Size Adjustment (SSSA) in [18] aim to improve resource utilization and reduce latency in fog computing. A review in [19] discusses these methods' role in dynamic task allocation. While these approaches offer near-optimal solutions with lower complexity, they often lack adaptability in dynamic, failure-prone environments.

In summary, compared to conventional optimization algorithms, heuristic and meta-heuristic methods, and traditional machine learning approaches proposed in the literature to solve the task offloading problem, the DRL approach has several key characteristics that make it an ideal solution for online, delay-sensitive task offloading in a dynamic MECC environment [20]. In such an environment, factors such as network conditions, computational resources, node failure rates, and user demands are constantly changing or often unknown in advance:

- A DRL agent does not require prior knowledge of all system parameters to find the optimal solution, as it adapts its offloading strategies through interactions with the environment over time. In contrast, conventional optimization algorithms require all model parameters to be known in advance to make decisions.
- DRL allows for continuous improvement over time, adjusting policies to reflect changing system dynamics. It can adapt in real-time by exploring various actions and observing their outcomes. Conventional optimization methods, on the other hand, need frequent recalculation due to changing environmental conditions, limiting their applicability.
- Task offloading in MECC environments is an online problem, meaning tasks are not available to the optimization algorithm all at once. Instead, tasks arrive over time, and the offloading component must allocate resources based on the current system state in a timely manner. Conventional optimization and meta-heuristic approaches are typically suited for the offline version of the task offloading problem, where all tasks are available to the algorithm at once. However, this is not feasible for delay-sensitive and online task scheduling.
- DRL makes faster decisions compared to conventional optimization and meta-heuristic approaches, which require exploring the search space to find an optimal or suboptimal solution. In contrast to other online heuristic methods, such as greedy algorithms, which make fast local decisions that may not lead to globally optimal results, DRL gradually learns to make step-by-step decisions that optimize a global objective function through the Markov Decision Process (MDP) framework.

- Unlike traditional machine learning algorithms, which rely on labeled training datasets that are often unavailable in real-world scenarios, DRL learns optimal decisions through interactions with the environment over time.

Few studies have focused on the FT vehicular task offloading problem: One approach, presented in [16], uses Q-learning and a primary-backup model for reliable offloading. Another study [21] proposes a cost-effective and failure-resistant task offloading strategy for Vehicular Edge Computing (VEC) considering only recoverable failures. However, assuming that all task failures on an edge server are recoverable is not always accurate. Restarting a task on the same server may result in another failure. Therefore, in addition to retrying a failed task, other recovery strategies should be considered when offloading tasks to the edge. Selecting the best recovery strategy for a task, especially in dynamic, resource-limited, and delay-sensitive offloading environments, is a challenging problem that has not been fully addressed in previous studies. For example, the retry strategy is unsuitable for permanent or long-lasting failures but may be appropriate for temporary system faults. Parallel recovery strategies, such as first-result, can have significantly lower latency than sequential strategies like recovery-block, but they may consume twice as many resources in no failure scenarios. Therefore, the offloading system should suggest the most suitable recovery strategy based on the current system state and the offloading objectives.

To bridge the above-mentioned research gap, a DRL-based FT task offloading method for MECC environments is proposed in this paper that explores a vast state space of potential offloading solutions, incorporating diverse failure recovery strategies to create optimal offloading plans. We aim to minimize the average response latency of all tasks under faulty conditions.

The key contributions of this study are as follows:

- Proposing an FT MECC architecture for reliable vehicular task offloading: The Fault-aware Execution Planner (FEP) component in this architecture suggests optimal Task Execution Plans (TEPs) for incoming vehicular tasks, considering their reliability requirements. FEP explores a vast state space of potential offloading solutions, incorporating diverse failure recovery strategies to create optimal TEPs.
- Developing an analytical model for the FT task offloading optimization problem considering sequential and parallel recovery strategies.
- Designing the FEP component based on a Deep Deterministic Policy Gradient (DDPG) algorithm. This DDPG algorithm determines the optimal deployment and recovery strategies for delay-sensitive tasks across the MECC infrastructure. The actor-critic architecture of DDPG, where the actor chooses actions based on the current state and the critic evaluates the action, allows DDPG to adapt more smoothly to dynamic, continuous environments like vehicular networks, where the system's state and the available resources are constantly changing.
- Conducting several experiments to evaluate the performance of the proposed method by comparing it with state-of-the-art task offloading methods.

The rest of this paper is organized as follows: Sect. 2 provides background information, covering failure recovery patterns and the DRL model. Section 3 reviews the related work in the area of task offloading and fault tolerance. In Sect. 4, we present the proposed FT MECC architecture. Section 5 defines the system model and formulates the optimization problem. The design of our DRL-based algorithm is detailed in Sect. 6. Section 7 discusses the performance evaluation, including the experimental setup, evaluation metrics, and comparison with baseline methods. The limitations of the proposed approach are explained in Sect. 8, and finally, Sect. 9 concludes the paper with a summary of the findings and future research directions.

2 Background

This section introduces the key concepts used in this study. We first define three important recovery patterns, followed by presenting the DRL model and one of its well-known implementation algorithms.

2.1 Failure recovery patterns

Failure recovery techniques are essential components of building any fault-tolerant (FT) system. Several well-known failure recovery patterns, categorized in the literature, are commonly used in various fault-tolerant systems: ignore, notify, skip, first-result, voting, recovery-block, retry, and rollback [22]. These patterns offer different strategies for handling task failures, depending on the system's needs and the nature of the task. In this study, we considered three important failure recovery patterns to be used when offloading a task:

1. Retry recovery pattern (RT): This is a sequential recovery model that restarts the failed task on the same server. It assumes that the failure was temporary and that retrying the task on the same server will succeed. This method is simple but may not be effective in cases of persistent server issues.
2. Recovery-Block recovery pattern (RB): Also, in a sequential recovery model, the Recovery-Block pattern starts a backup task on a server different from the one where the primary task failed. This approach aims to increase the chances of success by leveraging a separate, potentially more stable server for task execution. It can be useful when the failure is permanent.
3. First-Result recovery pattern (FR): This is a parallel recovery model, where both the primary and backup tasks are initiated simultaneously on different servers, and the first output is used. The primary advantage of this approach is speed, but it may increase resource consumption due to running tasks in parallel.

The terms "primary task" and "backup task" in the above definitions refer to the main and fallback executions of an offloaded task, respectively. Additionally, there are other failure recovery patterns that are less relevant to the failure types considered in this study:

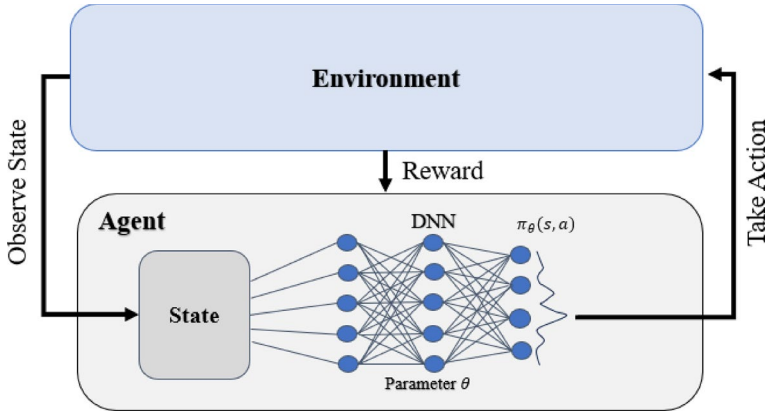

Fig. 1 The interaction between a DRL agent and its environment

- *Ignore* The system simply ignores the failure and continues operation, potentially risking reduced performance or incorrect results.
- *Notify* In this pattern, the system notifies the user or administrator of the failure but does not take any automated recovery actions. This pattern is typically used for non-critical tasks where manual intervention is needed.
- *Skip* The system skips the failed task and proceeds with the next task in the queue. This method is useful in situations where the failure does not significantly impact the overall system or task flow.
- *Voting* Multiple copies of the task are executed on different servers, and the result that receives the majority of votes is considered correct. This approach is useful in systems with arbitrary failures.
- *Rollback* If a failure occurs during task execution, the system reverts to a previous stable state. This can involve undoing any changes made by the task up to the point of failure, ensuring the system remains in a consistent state.

2.2 Deep reinforcement learning model

DRL provides a promising solution to the NP-Hard vehicular task offloading problem in MECC environments. The DRL is a machine learning-based optimization method within the framework of Markov Decision Processes (MDP) that leverages deep learning to solve decision-making problems. A learning problem is defined as a tuple $\langle S, A, P, R, \gamma \rangle$. where S and A are the state and action spaces, respectively, P is the state transition probabilities, R is a reward function and $\gamma \in [0, 1]$ is a discount factor that determines the weight of future rewards. In each time interval t., the DRL agent receives the current environment state s_t and chooses an action a_t based on its policy function $\pi(a_t|s_t)$ mapping the states to the actions. Depending on the action's effect on the environment, the agent receives a reward r_t and the next state s_{t+1}. The agent's goal is to find a policy π that maximizes the expected total of future discounted rewards [23]. The interaction between a DRL agent and its environment shown in Fig. 1.

$$V^{\pi}(s_t) = E_{\pi}\left[\sum_{t} \gamma^t r_t\right] \tag{1}$$

Deep Deterministic Policy Gradient (DDPG) [24] is a popular model-free, off-policy, and actor-critic DRL algorithm. It consists of four DNNs: actor-$\mu(s|\theta)$critic $Q(s,a|\phi)$, target actor $\mu'(s|\theta')$, and target critic $Q'(s,a|\phi')$ networks, where $\mu and Q$ are parameterized function approximators for the policy and the Q-function, respectively. The former generates actions from states and the latter generates an expected reward from a (state, action) pair. The target actor and critic networks.

$\mu' and Q'$ are used to mitigate the diversity possibility of Q-learning and enhance learning stability. The DDPG algorithm works as follows: At each time step t, state s_t is perceived by the DRL agent and action $a_t = \mu(s_t|\theta) + \omega_t$ is suggested by the actor-network, where, ω_t is an exploration noise. Afterwards, the tuple (s_t, a_t, r_t, s_{t+1}) is added to a replay buffer where s_t is the current state, a_t is the current action, r_t is the received reward and s_{t+1} is the next state. Then the actor and critic networks are updated by randomly sampling a mini-batch from the replay buffer. The critic network parameters are updated by minimizing the loss in formula (2) [24]. where, r_e is the reward value of taking action a_e in state s_e, and γ is the discount factor. The term $r_e + \gamma Q'(s_{e+1}, \mu'(s_{e+1}|\theta')|\phi')$, is the updated Q-value computed using the target networks. The actor-network is updated by the policy gradient as in formula (3) [24].

$$\text{loss} = \frac{\sum_{e=1}^{E}\left(Q(s_e,a_e|\phi) - \left(r_e + \gamma Q'(s_{e+1}, \mu'(s_{e+1}|\theta'))|\phi'\right)^2\right.}{E} \tag{2}$$

$$\text{the } \nabla_{\theta} J \approx \frac{\sum_{e=1}^{E} \nabla_{\theta}\mu(s_e|\theta)\nabla_{a=\mu(s_e)}Q(s_e,a|\phi)}{E} \tag{3}$$

where E is the number of transitions in the mini-batch. Finally, the target actor and critic networks are updated based on the formulas (4) and (5), where.

$\lambda << 1$ [24].

$$\theta' \leftarrow \lambda\theta + (1-\lambda)\theta' \tag{4}$$

$$\phi' \leftarrow \lambda\phi + (1-\lambda)\phi' \tag{5}$$

The DDPG algorithm offers several key strengths compared to other DRL algorithms: DDPG employs an actor-critic architecture, which allows it to learn a policy directly, providing greater flexibility in action selection for complex tasks. Additionally, DDPG's use of experience replay and target networks enhances training stability and efficiency, enabling it to better handle high-dimensional state spaces [24]. These characteristics make DDPG suitable for the FT task offloading problem in dynamic and complex MECC environments with continuous state space.

3 Related works

Within the expansive domain of Intelligent Transportation Systems (ITS), this section explores the related studies that address the intricate challenges posed by vehicular computing systems. The multifaceted issues at the forefront include the optimization of task processing, minimization of latency, effective energy management, and optimal resource allocation. These investigations unfold within the rich landscape of Vehicular Edge Computing (VEC), Mobile Edge Computing (MEC), Mobile Edge-Cloud Computing (MECC), and Vehicular Edge–Cloud Computing (VECC).

In the realm of ITS, Cao et al. [4] proposed a Relay-Assisted Parallel Offloading (RAPO) strategy to efficiently handle computation-intensive tasks in Internet of Vehicles (IoV) and MEC. Ta Huu Binh et al. [13] introduced a VECC network to address the challenges posed by the growing volume of data in Internet-connected devices. VECC aimed to enhance task processing on IoT devices with limited capacity by utilizing cloud servers positioned close to the tasks and tapping into the unused resources of smart vehicles. The study emphasized the importance of integrating cloud servers to prevent system overload during periods of increased data activity. The task offloading issue in VECC was conceptualized as a Markov decision process (MDP), and the authors proposed an innovative advantage-oriented task offloading approach employing a dueling actor-insulator network scheme. Chang-Lin Chen et al. [12] introduced a strategy designed to elevate the quality of service for autonomous driving applications. Their focus was on the communication, computation, and caching in a Vehicular Multi-access Edge Computing system to decrease task latencies and to ensure that tasks are completed efficiently within specified time limits. This optimization procedure involved a dual-step approach, employing a Deep Q-learning algorithm for optimal task assignment to edge servers and implementing a greedy strategy for communication, computation, and caching methods. This approach effectively managed trade-offs related to the bandwidth, power, CPU, and flexibility-hit rate. Li et al. [25] studied the complexities of 6G ITSs, with a specific focus on the trade-off between task latency and energy consumption in Vehicle-to-Everything (V2X) communications facilitated through VEC. The researchers tackled the challenge of optimizing the computation offloading in a speed-adjustable VEC, conceptualizing it as a mixed integer nonlinear programming problem. Their investigation introduced a straightforward algorithm that utilized dynamic programming for the task offloading and resource allocation problem, complemented by a speed adjustment algorithm inspired by proximal gradient principles. Dai et al. [14] proposed a meta-reinforcement learning framework for the Multi-Task Offloading (MTO) problem, which trains a meta-policy offline and then adjusts it quickly by visiting new MTO scenarios. To minimize the task execution times, a sequence-2-sequence meta-reinforcement learning algorithm based on bidirectional Gated Recurrent Units (GRU) with an attention mechanism was designed that determines the optimal offloading actions. Addressing issues in VEC, Ma et al. [26] placed emphasis on resource constraints and the dynamic

nature of high mobility. They introduced the Fully Distributed Task Offloading (FDTO) scheme, incorporating both greedy and convex optimization algorithms. These algorithms enabled the vehicles to adjust their offloading decision based on the resource utilization information received from the nearby Road Side Units (RSUs). The primary objective of this study was to shorten the task response time. Materwala et al. [27] proposed an Artificial Intelligence QoS-SLA-aware Adaptive Genetic Algorithm (QoS-SLA-AGA) that optimized the application execution time in Vehicular Ad Hoc Networks(VANETS). Focusing on the Internet of Vehicles, their work addressed SLA requirements and the limitations of existing offloading solutions. The algorithm considered overlapped multi-request processing and variable vehicle speeds, introducing an adaptive penalty function to integrate SLA constraints such as latency, processing time, deadline, CPU, and memory requirements.

Zeng et al. [28] tackled the rising computational challenges in intelligent vehicles, advocating for MEC. Their Telematics-focused work streamlined communication between vehicles and base stations during high-speed movement by offloading tasks to edge servers. To counter incomplete computations from vehicles leaving their current base stations, they proposed a trajectory-based offloading scheme. This scheme, using Long Short-Term Memory (LSTM) and Convolutional Neural Networks (CNN), accurately predicted the next base station and arrival time, ensuring timely results and efficient offloading of data-intensive tasks to the edge server. Wan et al. [29] explored real-time IoV scenarios, emphasizing efficient resource usage of idle vehicles in autonomous driving. Their proposed framework envisions all vehicles as edge nodes, minimizing busy vehicles' computation latency and maximizing the idle vehicles' resource utilization. They introduced low-complexity methods for busy vehicles and idle vehicle pairing, addressing one to one and one to many matching with an improved biogeography-based optimization algorithm, considering latency and energy consumption. Xu et al. [30] highlighted Web3's role in decentralized device collaboration. They introduced GPOV, a dynamic offloading strategy utilizing game theory and CNN partitioning to optimize deep learning tasks in vehicular edge networks, addressing challenges like server overload and delays. Wenhao Fan et al. [31] studied the VEC architecture to enhance task processing capabilities. They proposed a novel joint task offloading and resource allocation scheme to minimize overall task processing delays in the VEC system. Unlike existing works, their approach emphasized task processing flexibility. The optimization problem was tackled using a Generalized Benders Decomposition (GBD) and Reformulation Linearization (RL) algorithm for an optimal solution, complemented by a heuristic algorithm for reduced computational complexity. Lei Liu et al. [1] explored VEC by proposing a framework for collaborative task computing and on-demand resource allocation. Their research aimed to optimize task and resource scheduling policies, considering the dynamic nature of vehicular networks using an asynchronous deep reinforcement algorithm. Zhijian Lin et al. [32] proposed a novel solution for Vehicular Fog Computing (VFC), introducing a non-orthogonal multiple access (NOMA)-enabled multi-fog access point (F-AP) scheme with partial offloading. The work aimed to address the challenges of overloaded F-APs in crowded areas. They optimized power allocation

using monotonicity and employed a successive convex approximation (SCA)-based interior-point method and a game theoretic approach for efficient task splitting ratio and user association, respectively.

Mishra et al. [15] studied the task offloading problem in Vehicular Fog Computing (VFC) networks. They proposed a Fuzzy Logic-based classifier and a Federated Deep Q-learning (FEDQL) method to decide the offloading layer and the offloading node within the layer, respectively, considering the latency and energy objectives. Yang et al. [33] tackled challenges in delay-sensitive vehicular applications within IoV, highlighting the crucial role of efficient data processing for autonomous vehicles. They introduced a stochastic geometry-based traffic model to better simulate real traffic scenarios. To reduce task processing costs, the authors proposed a distributed computation offloading scheme using MEC and engaging nearby vehicles and RSUs with ample computing resources. Task splitting ratios were optimized by dividing the NP-hard problem of average cost minimization into sub-problems, utilizing the Lagrange multiplier with Karush–Kuhn–Tucker (KKT) constraints. Li et al. [34] addressed the increasing demand for in-vehicle services that rely on multiple sensors within the VEC paradigm. They emphasized the significance of optimizing completion time and energy consumption through energy-efficient task scheduling. The authors introduced the Multi-action and Environment-adaptive Proximal Policy Optimization algorithm (MEPPO), an extension of the conventional PPO algorithm, proposing a combined approach for task scheduling and resource allocation. Cong et al. [35] examined task offloading challenges in MEC for vehicular networks, prioritizing computational constraints to minimize both latency and energy consumption. Their model, integrating diverse computing resources, evaluates tasks through a loss function. They addressed optimization using the block coordinate descent technique, efficiently solving resource allocation and offloading strategy sub-problems. The convex nature of the former permits a polynomial time solution, while the non-convexity of the latter is handled through discrete variable relaxation and the Gray Wolf algorithm with an elite strategy. Xia et al. [36] explored VEC architecture to address low-latency needs in vehicular networks. They introduced a novel location-aware task offloading mechanism in a VEC-based single-vehicle multi-cell (SVMC) scenario, where both task uploading and computing were considered concurrently. They investigated single-cell and multi-cell offloading, devising low-complexity algorithms to optimize task processing delay. Zheng et al. [37] tackled the challenge of timely task offloading in pedestrian-vehicle interaction scenarios within the IoV paradigm. They proposed a location prediction-based scheme using social force and car-following models, leveraging a pre-trained neural network for task characteristics. This approach enabled pre-arrival task decisions, meeting low-latency requirements. Da Costa et al. [38] introduced MARINA, an efficient task scheduler tailored for Vehicular Clouds (VC) within the VEC framework. VEC, a promising paradigm, brings cloud services closer to vehicular users through resource pools known as VC. The proposed scheduler utilizes an approximation heuristic and resource prediction, carefully examining vehicle behavior and mobility to optimize task scheduling within the VC. In the context of minimizing data delivery latency, Ren et al. [39] proposed a UAV-based solution for IoT systems, utilizing a collaborative data acquisition model with UAVs to optimize data

transmission. By implementing a time-balancing scheduling scheme, their approach significantly reduces data delivery latency.

Prior investigations have predominantly centered on resource allocation and response latency, often neglecting the vulnerability of task execution to failures. Bridging this gap, Tang et al. [21] introduced an innovative, cost-effective, and failure-resistant task offloading strategy for VEC, aiming to minimize the average response latency. They conceptualized the issue as a nonlinear multi-constraint continuous optimization problem, incorporating closely interconnected optimization variables. To handle this intricacy, the researchers decomposed the problem into per-slot optimization sub-problems and employed an efficient algorithm for sequential resolution with low time complexity. Fu et al. [40] highlighted the impact of cascading failures in IoT systems, which can lead to network paralysis and pose a critical challenge to system reliability. To address these issues, they proposed various optimization and modeling methods, aiming to improve the long-term reliability of IoT systems.

Ray et al. [41] proposed a two-level failure recovery strategy in MEC environments. They introduced a formal method-based local recovery strategy and a greedy-based global recovery strategy for high and low-priority applications, respectively. Their recovery strategy aims to select the best alternative server to re-initialize the containerized applications once the primary server fails. Siyadatzadeh et al. [16] studied the application of the Q-learning method in the reliable task offloading problem in Fog Computing (FC) environments. They used Integer Linear Programming (ILP) to formulate the problem where the tasks were constrained by deadlines and then solved the problem using Q-learning. They also used the primary-backup model as their fault tolerance technique.

The related studies are summarized and compared in Table 1. As shown in Table 1, fault tolerance in task offloading has not been addressed in most previous studies. The main contribution of the proposed method, which differentiates it from previous studies, is the concept of the *Recovery Strategy Suggestion* based on the current context of a dynamic environment, such as MECC. Few prior works have incorporated failure models [21] or failure/recovery models [16, 41] in their problem formulations; however, they typically rely on fixed strategies, limiting their adaptability in dynamic environments. In contrast, this paper bridges this gap by proposing a DRL-based task offloading approach that explores a vast state space of potential offloading solutions, incorporating diverse failure recovery strategies (such as Retry, Recovery Block, and First Result) to create optimal execution plans. More stable learning and efficient exploration in the continuous state space are achieved through the DDPG implementation of the approach due to the actor-critic architecture of DDPG, which includes a target network for the critic and a separate target policy for the actor.

4 Proposed fault-tolerant MECC architecture

The Fault-tolerant MECC system architecture is shown in Fig. 2. There are three layers in the architecture: (1) the vehicle layer which consists of smart vehicles, (2) The edge layer which consists of RSUs and Edge Servers (ESs), and (3) the cloud layer which consists of an elastic pool of computation and storage resources.

Table 1 A comparison of task offloading approaches. L, E, and R in the Objective column denote Latency, Energy, and Reliability, respectively

Paper	Architecture	Algorithm/Method	Objective			Fault tolerance		
			L	E	R	Failure model	Recovery model	Strategy suggestion
[4]	MEC	Greedy Clustering, DE[1]	●	×	×	×	×	×
[12]	MEC	DQL, Greedy	●	×	×	×	×	×
[13]	VECC	Dueling DQL	●	×	×	×	×	×
[14]	MECC	Seq-2-seq meta DRL	●	×	×	×	×	×
[15]	VFC	Fuzzy Logic, Federated DQL[2]	●	●	●	×	×	×
[25]	VEC	Proximal Gradient Method	●	●	×	×	×	×
[26]	VEC	Greedy, Convex Optimization	●	×	×	×	×	×
[27]	VECC	GA	●	×	×	×	×	×
[28]	MEC	LSTM, CNN, DPG	●	●	×	×	×	×
[29]	VEC	RL[3], IBBO[4]	●	●	×	×	×	×
[31]	VEC	GBD[5], RL, Heuristic algorithm	●	×	×	×	×	×
[31]	VEC	A3C	●			×	×	×
[32]	SDN-based IoV	SCA-IPM[6], Coalitional game	×	×	●	×	×	×
[33]	MEC	Lagrange multiplier	●	●	●	×	×	×
[34]	VECC	Proximal Policy Optimization	●	●	×	×	×	×
[35]	MECC	Convex Optimization, GFA[7]	●	●	×	×	×	×
[36]	VEC	Heuristic optimization	●	×	×	×	×	×
[37]	MEC	DQL	●	×	×	×	×	×
[38]	VECC	LSTM, BCP[8], FFD[9]	●	●		×	×	×
[16]	FC	Q-learning	●	×	●	●	●	×
[21]	VEC	Greedy Optimization	●	●	×	●	×	×
[41]	MEC	Stochastic Game	●	×	●	●	●	×
Our work	MECC	DRL-DDPG	●	×	●	●	●	●

[1]Deferential evolutionary

[2]Deep Q-learning

[3]Reformulation linearization

[4]Improved biogeography-based optimization

[5]Generalized Benders decomposition

[6]Successive convex approximation interior-point method

[7]Gray wolf algorithm

[8]Bin covering problem

[9]First-fit decreasing

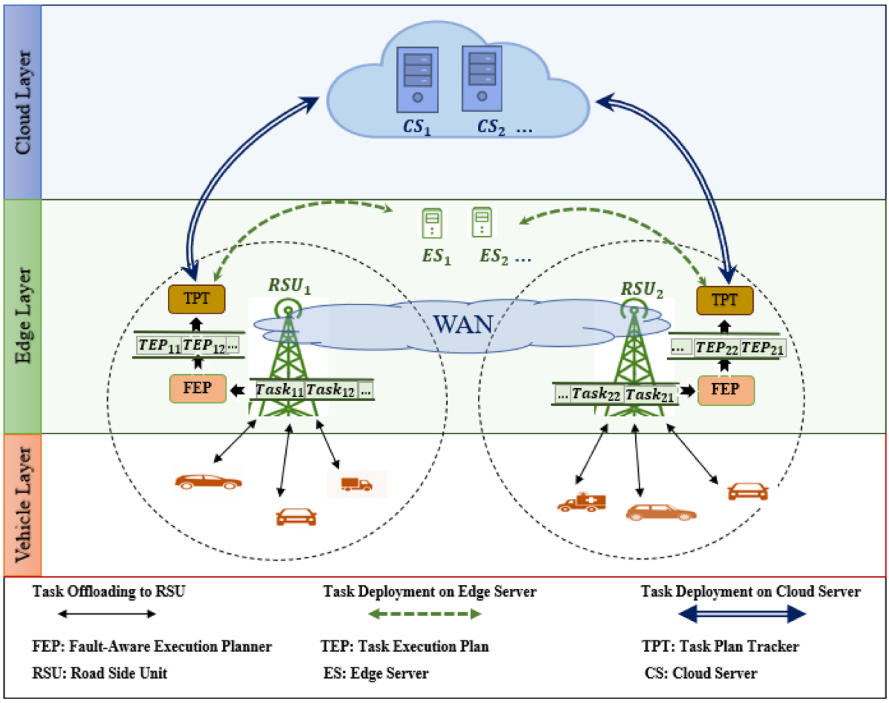

Fig. 2 Fault-tolerant MECC system architecture

Vehicles in the vehicle layer generate vehicular tasks with various resource demands depending on the task type. Simple tasks, such as basic sensor data processing and vehicle diagnostics, require minimal computation. Latency-sensitive tasks, including autonomous driving perception and collision avoidance, demand real-time processing. Computation-intensive tasks, such as high-resolution video analytics and deep learning-based decision-making, require significant processing power. Since vehicles contain limited local computation resources, simple tasks are executed locally, while complex tasks are offloaded to a nearby RSU. Each RSU is equipped with a Fault-Aware Execution Planner (FEP) unit that creates Task Execution Plans (TEPs) for the offloaded tasks. A TEP for an offloaded task k is defined as a triple $\left(node_k^{primary}, node_k^{backup}, r_k\right)$ where $node_k^{primary}$ and $node_k^{backup}$ denote the edge/cloud server identifies on which the primary and backup copies of task k should be deployed respectively, and r_k denotes the selected recovery pattern for task k which could be one of the RT, RB, or FR patterns. The FEP unit of an RSU can allocate resources to the offloaded tasks from the free capacity of its associated ESs, or from the cloud layer. As the ESs are placed between the cloud center and smart vehicles, they are able to execute delay-sensitive tasks with less response latency compared to the cloud layer. The cloud layer provides a reliable service (based on a Service Level Agreement) with longer response delay mostly due to the transmission latency, which is suitable for non-real-time tasks. However, ESs are more susceptible to software/hardware failures than the

cloud layer, which can adversely affect the Quality of Experience (QoE) from the viewpoint of client vehicles.

It is also notable that, as opposed to the assumption of the previous studies [21], a task failure in an ES might be unrecoverable locally, meaning that restarting the task on the same ES may result in another failure. Hence, besides the RT recovery pattern, the FEP unit at the RSU examines RB and FR recovery patterns and applies the best one to create TEPs. Selecting the best recovery pattern for a task by the FEP unit, particularly in dynamic, resource-limited, and delay-sensitive offloading environments, is a challenging task. For instance, the RT pattern is not an appropriate strategy for a task with permanent or long-lasting failures, whereas it could be a good choice when the task fails due to a temporary system fault. The FR pattern incurs much less latency compared to the RB pattern, though its resource utilization is twice in no failure scenarios. Hence, the FEP unit has to suggest the right recovery pattern, considering the current system state and the offloading objectives.

A Task Plan Tracker (TPT) unit deployed at the RSU starts TEPs and continuously checks their status until the output is ready. Any failure during the task execution is handled by the TPT according to the recovery pattern defined in the TEP. Once the task output is ready, the TPT unit delivers it to the local RSU to be routed inside the network of RSUs toward the requester vehicle's location. As an example, consider a task k which involves high-resolution video analytics. This task is offloaded to the nearest RSU. The RSU's FEP generates a TEP, which chooses the RB recovery pattern due to the task's criticality and system state. In this case, the FEP deploys the primary task copy on an ES and the backup copy on a different server to ensure fault tolerance. The TPT at the RSU initiates the task and continuously monitors its progress. During execution, if the primary server fails, the backup copy immediately takes over the task, ensuring no interruption in the vehicle's navigation. Once the task is completed successfully, the output is routed back to the vehicle for continued operation.

5 System model and problem formulation

In this section, we will explore the system model and problem formulation for FT task offloading from vehicles to RSUs. For clarity, a notation table is shown in Table 2, which defines the symbols and variables used throughout this section. When a vehicle v offloads a task k to a nearby RSU r, the total latency incurred before the results are ready to use at v comprises the vehicle to RSU communication delay, the task completion time on the allocated computing server (edge/cloud), and finally, the RSU to the vehicle communication delay to deliver the output results. In the presence of failures, however, the average total latency needs to be computed as presented in formula (6):

$$\overline{L}_k(t) = D^{v,r}_{k.inp}(t) + D^{r,v}_{k.out}(t) + \tilde{L}^{normal}_k(t) + \tilde{L}^{recovery}_k(t) + \tilde{L}^{failure}_k(t) \tag{6}$$

where $D^{v,r}_{k.inp}(t)$ and $D^{r,v}_{k.out}(t)$ are communication delays of transmitting the input data and the output results of the offloaded task k between RSU r and vehicle v in time

Table 2 Notations

Notation	Definition
T	The set of time slots
K	The set of vehicular tasks
V	The set of vehicles
R	The set of RSUs
$E(r)$	The set of edge servers that RSU r can use for task assignment
$\mathbb{C}$	The set of cloud layer servers
$k.inp$	The input data size of task k
$k.out$	The output data size of task k
$\omega_k(t)$	The computation demand of task k in time slot t
$\theta_k(t)$	The execution timeout of task k in time slot t
$f_n(t)$	The processing frequency of computing server n in time slot t
$\lambda_n(t)$	The failure rate of server n in time slot t
$\psi_k^n(t)$	The failure probability of task k on server n
$x_k^{n,o}(t)$	The decision variable that determines if the primary task k is assigned to server n, as the o^{th} task, in time slot t
$y_k^{n,o}(t)$	The decision variable that determines if the backup task k is assigned to server n, as the o^{th} task in time slot t
$z_k(t)$	The decision variable that determines whether the recovery of task k is based on the sequential model (RB or RT) or the parallel model (FR) in time slot t
$\varphi_k(t)$	The latency penalty incurred when task k fails
$D_m^{s,d}(t)$	The communication delay incurred when transmitting data size m from source s to destination d in time slot t
$B^{s,d}(t)$	The communication bandwidth between source s and destination d in time slot t
$\overline{P}_k^n(t)$	The average completion time of task k assigned to server n in time slot t
$\overline{Q}_k^n(t)$	The average queuing time of task k assigned to server n in time slot t
$S_k^n(t)$	The service time of task k assigned to server n in time slot t
$\overline{L}_k(t)$	The average total latency of task k in time slot t
$\tilde{L}_k^{normal}(t)$	The weighted total latency of task k in a normal execution scenario in time slot t
$L_k^{primary}(t)$	The total latency of the primary task k execution in time slot t
$L_k^{backup}(t)$	The total latency of the backup task k execution in time slot t
$\tilde{L}_k^{recovery}(t)$	The weighted latency of task k in a recoverable failure scenario in time slot t
$\tilde{L}_k^{failure}(t)$	The weighted latency of task k in an unrecoverable failure scenario in time slot t
$p_k^{normal}(t)$	The probability that task k terminates successfully in time slot t
$p_k^{recovery1}(t)$	The probability that task k recovers from a primary failure in time slot t
$p_k^{recovery2}(t)$	The probability that task k recovers from a backup failure in time slot t
$p_k^{failure}(t)$	The probability that task k fails in time slot t

slot t respectively. $\tilde{L}_k^{normal}(t), \tilde{L}_k^{recovery}(t) and \tilde{L}_k^{failure}(t)$ are the weighted total latency of task k in a normal execution scenario, a recoverable failure scenario and a failure scenario respectively. The total latency of task k in the normal execution scenario

(when no failure occurs) equals the total latency of the primary task execution denoted by $L_k^{primary}$, when the selected recovery pattern for task k is RB or RT. Otherwise, when the selected recovery pattern is FR, it equals the minimum of the primary and the backup execution latencies, as presented in formula (7):

$$\tilde{L}_k^{normal}(t) = \begin{cases} p_k^{normal}(t) \times L_k^{primary}, z_k(t) = 0 \\ p_k^{normal}(t) \times \min\{L_k^{primary}(t), L_k^{backup}(t)\}, z_k(t) = 1 \end{cases} \tag{7}$$

where, $z_k(t)$ is a decision variable that determines the selected recovery pattern of task k. If $z_k(t) = 1$, the FR recovery pattern is enabled which is a parallel recovery model. Otherwise, if $z_k(t) = 0$, the sequential recovery for task k is enabled which could be either RB or RT. The former is enabled when both the primary and the backup tasks are assigned to the same computing server, while the latter is enabled when they are assigned to different servers. The assignments of the primary and the backup tasks to the edge/cloud servers are controlled by $x_k^{n,o}(t)$ and $y_k^{n,o}(t)$ decision variables respectively. These variables are set to one to specify that primary/backup task k is assigned to server n as the o^{th} task in the queue in time slot t. Besides, $x_k^{n,*}(t)$ and $y_k^{n,*}(t)$ are used to determine the assignments of primary/backup task k to server n regardless of the order in the queue:

$$x_k^{n,*}(t) = \sum_{o \in \{1,\ldots,|K|\}} x_k^{n,o}(t) \tag{8}$$

$$y_k^{n,*}(t) = \sum_{o \in \{1,\ldots,|K|\}} y_k^{n,o}(t) \tag{9}$$

The total latency of the primary/backup task execution, denoted by $L_k^{primary}$ /L_k^{backup}, involves the RSU to the computing server communication delay, denoted by $D_{k.inp}^{r,n}(t)$, the average task completion time on the allocated computing server, denoted by $\overline{P}_k^n(t)$, and finally, the RSU to the server communication delay to deliver the output results:

$$L_k^{primary}(t) = \sum_{n \in E(r) \cup \mathbb{C}} (D_{k.inp}^{r,n}(t) + D_{k.out}^{n,r}(t) + \overline{P}_k^n(t)) \times x_k^{n,*}(t) \tag{10}$$

$$L_k^{backup}(t) = \sum_{n \in E(r) \cup \mathbb{C}} (D_{k.inp}^{r,n}(t) + D_{k.out}^{n,r}(t) + \overline{P}_k^n(t)) \times y_k^{n,*}(t) \tag{11}$$

The average completion time of task k on server n, denoted by $\overline{P}_k^n(t)$, is computed as the sum of average queuing time and the service time on server n:

$$\overline{P}_k^n(t) = \overline{Q}_k^n(t) + S_k^n(t) \tag{12}$$

$$\bar{Q}_k^n(t) = \sum_{i \in K, ord(i) < ord(k)} \left(1 - \psi_i^n(t)\right) \times S_i^n(t) \times x_i^{n,*}(t) + \sum_{i \in K, ord(i) < ord(k)} \left(\sum_{m \in E(r) \cup C} \psi_i^m(t) \times x_i^{m,*}(t) \right) \times S_i^n(t) \times y_i^{n,*}(t) \tag{13}$$

where $\psi_i^n(t)$ is the failure probability of task i executing on server n in time slot t, and $S_k^n(t)$ is the service time of task k on server n in time slot t. $S_k^n(t)$ is computed in formula (14) where $\omega_k(t)$ and $f_n(t)$ denote the computation demand of task k in time slot t and the processing frequency of computing server n in time slot t respectively.

$$S_k^n(t) = \frac{\omega_k(t)}{f_n(t)} \tag{14}$$

The probabilities that task k terminates with no failures, recovers from the primary task failure, recovers from the backup task failure and terminates in a failed state are computed as presented in formulas (15)-(18) respectively:

$$p_k^{normal}(t) = \begin{cases} \sum_{n \in E(r) \cup C} \left(1 - \psi_k^n(t)\right) \times x_k^{n,*}(t), & amp; z_k(t) = 0 \\ \sum_{n1 \in E(r) \cup C} \sum_{n2 \in E(r) \cup C} \left(1 - \psi_k^{n1}(t)\right) \times \left(1 - \psi_k^{n2}(t)\right) \times x_k^{n1,*}(t) \times y_k^{n2,*}(t), & amp; z_k(t) = 1 \end{cases} \tag{15}$$

$$p_k^{recovery1}(t) = \sum_{n1 \in E(r) \cup C} \sum_{n2 \in E(r) \cup C} \psi_k^{n1}(t) \times \left(1 - \psi_k^{n2}(t)\right) \times x_k^{n1,*}(t) \times y_k^{n2,*}(t) \tag{16}$$

$$p_k^{recovery2}(t) = \sum_{n1 \in E(r) \cup C} \sum_{n2 \in E(r) \cup C} \left(1 - \psi_k^{n1}(t)\right) \times \psi_k^{n2}(t) \times x_k^{n1,*}(t) \times y_k^{n2,*}(t) \tag{17}$$

$$p_k^{failure}(t) = \sum_{n1 \in E(r) \cup C} \sum_{n2 \in E(r) \cup C} \psi_k^{n1}(t) \times \psi_k^{n2}(t) \times x_k^{n1}(t) \times y_k^{n2,*}(t) \tag{18}$$

The weighted total latency of task k in a recoverable failure scenario is computed in formula (19). In the sequential recovery patterns RT and RB, where $z_k(t) = 0$, the total latency of task k comprises the RSU to the computing server communication delay, denoted by $D_{k.inp}^{r,n}(t)$, the task execution timeout $\theta_k(t)$ and the latency of the backup execution denoted by $L_k^{backup}(t)$. In the FR recovery model, where $z_k(t) = 1$, the total latency of task k in a recoverable failure scenario equals either the primary task execution latency or the backup task execution latency as presented in formula (19).

$$\tilde{L}_k^{recovery}(t) = \begin{cases} p_k^{recovery1}(t) \times \left(\left(\sum_{n \in E(r) \cup C} D_{k.inp}^{r,n}(t) \times x_k^{n,*}(t) \right) + \theta_k(t) + L_k^{backup}(t) \right), \\ p_k^{recovery1}(t) \times L_k^{backup}(t) + p_k^{recovery2}(t) \times L_k^{primary}(t), \end{cases} \tag{19}$$

Similarly, the weighted total latency of task k in an unrecoverable failure scenario is computed in formula (20).

$$\tilde{L}_k^{failure}(t) = p_k^{failure}(t) \times \varphi_k(t) \tag{20}$$

where $\varphi_k(t)$ represents the latency penalty incurred when task k fails. Moreover, the failure probability of task k when executing on server n is computed based on the exponential distribution:

$$\psi_k^n(t) = 1 - e^{-\lambda_n(t) \times S_k^n(t)} \tag{21}$$

where $\lambda_n(t)$ denotes the failure rate of server n in time slot t. In the above formulas, the communication delay $D_m^{s,d}(t)$ incurred when transmitting data size m from source s to destination d in time slot t is computed as follows:

$$D_m^{s,d}(t) = \frac{m}{B^{s,d}(t)} \tag{22}$$

Finally, the objective is to minimize the sum of task average latencies over the offloading time slots as presented in formula (23) subject to constraints (24)-(30):

$$\min_{x,y,z} \frac{1}{|T|} \sum_{t \in T} \sum_{k \in K} \bar{L}_k(t) \tag{23}$$

$$\forall n \in E(r) \cup \mathbb{C}, \forall o \in \{1, \ldots, |K|\}, \forall k \in K : x_k^{n,o}(t), y_k^{n,o}(t), z_k(t) \in \{0, 1\} \tag{24}$$

$$\forall k \in K : \sum_{n \in E(r) \cup \mathbb{C}} x_k^{n,*}(t) = 1 \tag{25}$$

$$\forall k \in K : \sum_{n \in E(r) \cup \mathbb{C}} y_k^{n,*}(t) = 1 \tag{26}$$

$$\forall n \in E(r) \cup \mathbb{C}, \forall k \in K : x_k^{n,*}(t) \times y_k^{n,*}(t) \times z_k(t) = 0 \tag{27}$$

$$\forall n \in E(r) \cup \mathbb{C}, \forall o \in \{1, \ldots, |K|\}, \forall k_1, k_2 \in K : k_1 \neq k_2 \Rightarrow x_{k1}^{n,o}(t) \times x_{k2}^{n,o}(t) = 0 \tag{28}$$

$$\forall n \in E(r) \cup \mathbb{C}, \forall o \in \{1, \ldots, |K|\}, \forall k_1, k_2 \in K : k_1 \neq k_2 \Rightarrow y_{k1}^{n,o}(t) \times y_{k2}^{n,o}(t) = 0 \tag{29}$$

$$\forall n \in E(r) \cup \mathbb{C}, \forall o_1, o_2 \in \{1, \ldots, |K|\}, \forall k \in K : x_k^{n,o1}(t) \times y_k^{n,o2}(t) = 1 \Rightarrow o_1 < o_2 \tag{30}$$

where constraint (24) denotes that the x, y and z are binary variables, constraints (25) and (26) ensure that each primary/backup task must be assigned to exactly one computing server, constraint (27) ensures that a primary task and its backup are not assigned to the same server in the FR recovery pattern, constraints (28) and (29) guarantee that not two tasks in the same server have the same orders and, constraint (30) ensures that a primary task executes before its backup.

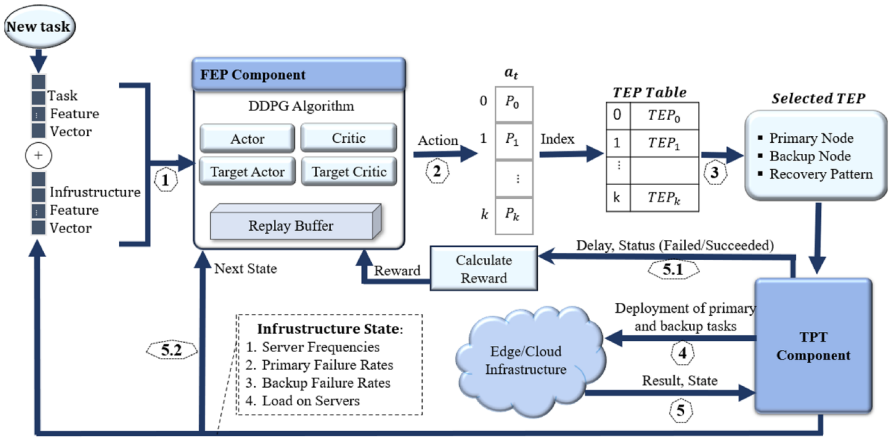

Fig. 3 The proposed method

6 Proposed method

To solve the FT vehicular task offloading problem detailed in Sect. 5 in real-world scenarios, it should be noted that not all tasks are available to the optimization algorithm at once, rather, they are received by the FEP component at the RSU over time and the FEP component creates the corresponding TEPs as soon as a new task is offloaded by a vehicle (online vs. offline task offloading problem). Both offline and online formulations of the vehicular task offloading problem are NP-Hard [1, 4, 27–29, 33, 35] as well as the new fault-tolerant formulation of this problem presented in this paper, and hence efficient algorithms are needed to solve the problem. To better understand why this problem is NP-Hard, consider that the FT vehicular task offloading problem involves selecting the optimal assignment of primary and backup tasks to different computational nodes, aiming to minimize both task latencies and task failures, while considering constraints associated with the chosen recovery patterns. The decision-making process is inherently combinatorial, as the number of ways to select recovery patterns for tasks and assign primary and backup tasks to nodes grows exponentially with the number of tasks. This complexity mirrors that of NP-Hard problems, where no known polynomial time algorithm can guarantee an optimal solution for all instances. Specifically, this problem is a variation of the NP-Hard "Multiprocessor Scheduling" problem, where tasks must be assigned to multiple processors (or servers), while optimizing certain objectives such as load balancing.

In this paper, a DDPG-based algorithm has been designed to solve the online version adaptively and efficiently, relying on the key strengths of the Deep Deterministic Policy Gradient method explained in Sect. 2.2. The proposed method is shown in Fig. 3. Once a new task is offloaded by a vehicle to a nearby RSU at time interval t, the FEP component at the RSU uses its DDPG actor-network to output an action a_t for the input task considering the current state s_t of the environment (steps 1 and 2). Afterwards, a_t is translated into a TEP, which specifies the edge/

cloud node identifiers to be used for the deployment of primary and backup copies of the input task along with a suggested recovery pattern for the task execution (step 3). Subsequently, the primary and backup copies of the task are deployed on the suggested nodes by the TPT component to start their execution according to the recovery pattern presented in the TEP (step 4). Then, the TPT component obtains the task execution result and the new environment state (step 5). Finally, the task execution delay and status (failed/succeeded) are used to compute the deployment reward that is fed back to the DDPG algorithm along with the next infrastructure state (steps 5.1 and 5.2).

The actor and critic networks of the DDPG algorithm are updated periodically based on the rewards recorded in the replay buffer. The state s_t, action a_t and reward r_t are defined as follows:

- **State:** The state information s_t is defined as $s_t = \langle taskFeatureVector_t, infrastructureFeatureVector_t \rangle$. The $taskFeatureVector_t$ captures task-related features at time t and is defined as $taskFeatureVector_t = \langle demand_t, size_t \rangle$ where, $demand_t$ denotes the estimated task computation demand and $size_t$ denotes the task size at time t. The $infrastructureFeatureVector_t$ is defined as $infrastructureFeatureVector_t = \left\langle state_1^t, state_2^t, \ldots, state_N^t \right\rangle$ where, N is the number of edge/cloud nodes and, $state_i^t$ denotes the i^{th} node state at time t. The $state_i^t$ is also defined as $state_i^t = \left\langle frequency_i^t, primaryFailureRate_i^t, backupFailureRate_i^t, utilization_i^t \right\rangle$ where, $frequency_i^t$ denotes CPU frequency of the i^{th} node, $primaryFailureRate_i^t$ denotes the failure rate of primary tasks on the i^{th} node at time t, $backupFailureRate_i^t$ denotes the failure rate of backup tasks on the i^{th} node at time t and, $utilization_i^t$ denotes the measured utilization of the i^{th} node at time t. The elements of $state_i^t$ are computed based on the task execution data collected by the TPT component. Particularly, the last element of $state_i^t$ is computed as the total computation demands of the tasks assigned to the i^{th} node at time t. All the values of state vector are normalized in the range [0, 1].
- **Action:** An action is defined as $a_t = \langle p_1, p_2, \ldots, p_n \rangle$, where $p_i \in [0, 1]$ determines the probability of suggesting the i^{th} task execution plan TEP_i for the input task k. TEP_i is defined in the i^{th} entry of the TEP Table, which contains all the possible valid execution plan triples for a task. As explained before, a task execution plan is defined as a triple $\left(node_k^{primary}, node_k^{backup}, r_k \right)$, where $node_k^{primary}$ and $node_k^{backup}$ denote the edge/cloud server identifies on which the primary and backup copies of task k should be deployed respectively, and r_k denotes the selected recovery pattern for task k, which could be one of the RT, RB, or FR patterns. Assuming that there are N edge/cloud nodes ready for the placement of primary and backup copies of task k, all the possible (and valid) triples are generated and added to TEP-Table. It is notable that not all combinations of node identifiers and recovery patterns together form a valid task execution plan. For instance, if we choose RT as the

recovery pattern in a TEP, the primary and backup nodes must be the same: $node_k^{primary} = node_k^{backup}$. In contrast, when either RB or FR patterns are selected, it is necessary that the primary and backup nodes be different to form a valid TEP. Hence, there are N, $N^2 - N$ and $\left(N^2 - N\right)/2$ valid TEPs for RT, RB, and FR patterns, respectively. By generating these $N^2 + \left(N^2 - N\right)/2$ valid TEPs and adding them to the TEP Table; this table keeps a complete list of task execution plans. The TEP entry with the highest probability based on a_t is selected for the task execution.

- **Reward:** The reward r_t is calculated based on the execution status of tasks (failed/succeeded) and their execution delay. Specifically, [42]:

$$reward = \begin{cases} -\mathrm{FM} \times delay & taskfailed \\ \log_{0.995} 1 - \frac{1}{\exp\left(\sqrt{delay}\right)} & tasksucceeded \end{cases} \tag{31}$$

In this formula, FM represents the Failure Multiplier, designed to increase the calculated penalty (negative reward) for failed tasks. This negative reward is also proportional to the delay, meaning tasks that fail with longer execution times receive a lower reward compared to those that fail with shorter execution times. For successful tasks, the reward is calculated using a logarithmic function that adjusts based on the delay, providing a positive reward that reflects better performance with shorter delays. The primary function of FM in this formula is to strike a balance between minimizing task execution delay and reducing task failures. By adjusting this parameter during simulation, the model is encouraged to prioritize not only faster execution but also more reliable task completion. This balance ensures that the model optimizes for both speed and reliability, aligning with the system's performance goals. The optimal FM value used in our experiments is shown in Table 3 (see Section 7).

Each episode in our RL-based task offloading system consists of several steps, with each step corresponding to the processing of one task. The agent begins by observing the current environment state, which includes servers' features (frequency, primary and backup failure rates and utilization) and task features (computation demand and size). The agent then selects an action using the DDPG model. This action is then mapped to a TEP, which determines the primary and backup servers, as well as the recovery pattern. The task is executed according to the generated TEP, and the environment state variables are updated accordingly. A reward is calculated based on the execution time and whether the task was completed successfully or not (using formula (31)), and the experience is stored for future learning. This cycle continues until all tasks in the episode are processed. The overall process, including state transitions, decision-making, and learning updates, is shown in Figure 4, which shows stepwise task offloading in the DRL-based system.

7 Performance evaluation

This section presents the performance evaluation of the proposed method, aimed at addressing the following research questions:

RQ1: Can a DRL approach solve the FT task offloading problem in dynamic MECC environments effectively and efficiently?

RQ2: Can utilizing various failure recovery patterns improve the performance of a DRL-based FT task offloading algorithm?

RQ3: How do the node failure rates affect the placement of tasks on a MECC infrastructure?

RQ4: What is the impact of infrastructure resource capacity on the recovery pattern selection?

RQ5: What is the impact of failure types (transient/permanent) on the recovery pattern selection?

Table 3 Simulation parameters

Component	Parameter	Value/Range
Task profile	Task number	200
	Task arrival rate	0.5
	Task arrival distribution	Poisson
	Task size (KB)	Uniform (1200, 13,000)
	Task computation demand	Normal (50, 16)
	Task timeout	1.5 times the estimated task completion time
DDPG component	Total episodes	300
	Buffer capacity	100,000
	Batch size	256
	Standard deviation	0.25
	Critic learning	0.0001
	Actor learning	0.00003
	Gamma	0.85
	Tau	0.005
	State vector dimension (actor-network)	R^{34}
	Action vector dimension (actor-network)	R^{92}
	Middle layers' shape (actor)	L1 = 300, L2 = 200
	Middle layers' activation function (actor)	ReLU
	Middle layers' shapes (critic)	L1 = 300, L2 = 200 (state path), L1 = 200 (action path), L3 = 200
	Middle layers' activation function (critic)	ReLU
	Output layer activation function	SoftMax
	FM	3

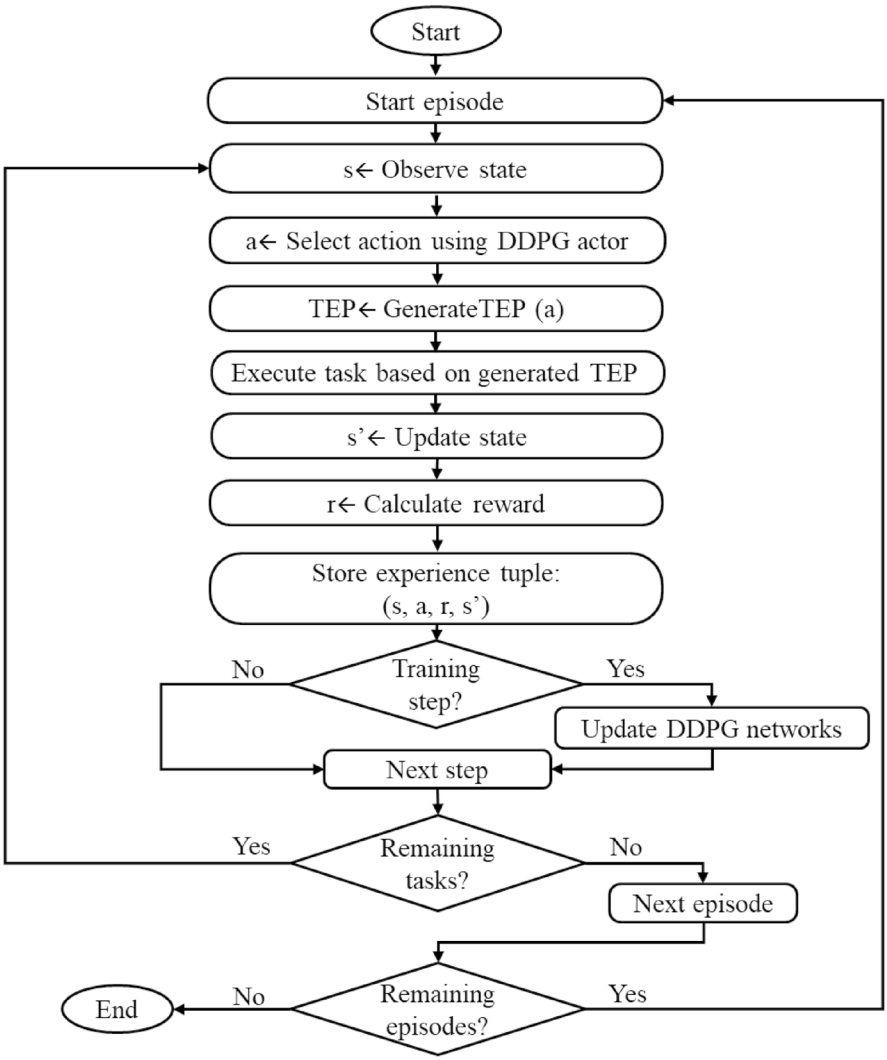

Fig. 4 Stepwise task offloading in DRL

7.1 Simulation environment setup

The experiments in this study were conducted on a device equipped with an Intel Core i5-13600K processor running at 3.50 GHz and 16 GB of RAM. The operating system used was Windows 11 Enterprise, 64-bit edition, and the device featured a 512 GB SSD for storage. The simulation environment was implemented in Python. TensorFlow and Keras were used to implement the DDPG model. NumPy and SciPy were employed for numerical and statistical computations, Pandas for data management and analysis, and Matplotlib for plotting charts and visualizing

simulation results. Openpyxl was used to work with Excel files and save data, while subprocess and OS were used for process management and system operations. SimPy was chosen as the primary tool for discrete event simulation and managing the simulation environment [43].

```
Algorithm SimPy_Simulation:
1.  Initialize env as a SimPy environment
2.  Initialize nodes as available Edge/Cloud servers
3.  Initialize ddpg as a DDPG model.
4.  For episode in [1: total_episodes] do:
5.      env.Reset()
6.      While taskCounter ≤ maxTask do:
7.          t ← env.nextTask(taskCounter, taskArrivalRate, taskParamsFile)
8.          state ← env.currentState()
9.          a ← ddpg.policy(t, state)
10.         taskObj ← Task(t, a)
11.         env.process(taskObj)
12.         Query (reward, nextState) from env
13.         Add (state, action, reward, nextState) tuple to the ddpg.buffer
14.         ddpg.train()
15.         Increment taskCounter
16.     End While
17. End For
```

Listing 1. The simulation algorithm implemented using the SimPy framework.

SimPy's framework supports key concepts such as events, processes, resources, and the simulation environment, allowing for accurate simulations of real-world processes, such as queuing and resource allocation. The simulation algorithm implemented using the SimPy framework is listed in Listing 1. This algorithm integrates the proposed DDPG model with the SimPy environment to simulate the entire task offloading process, including task arrival, server allocation, and selecting the most appropriate FT pattern based on real-time environment conditions. Each reinforcement learning episode begins with the initialization of the SimPy environment (line 5). Tasks arrive sequentially, governed by a predefined arrival rate (line 7). Upon a new task arrival, the DDPG algorithm employs its policy function (actor-network) to generate an offloading decision, considering both the incoming task and the current state of the simulation environment (line 9). A Task object is then instantiated, incorporating the task profile (computation demand and data size), the suggested primary/backup node profiles (failure rates and processing frequencies), and the suggested FT pattern (line 10). A SimPy process is subsequently initiated for the task within the simulation environment (line 11). The simulation logic implemented within the Task class is also shown in Fig. 5. The flowchart in Fig. 5 depicts how SimPy simulates the execution of both primary and backup tasks, adhering to the suggested FT pattern. The simulation algorithm then queries the environment for the status of completed tasks (failed/succeeded) and their execution delays and utilizes this information to compute the deployment reward. This reward and the subsequent environment state are fed back to the DDPG algorithm for retraining (lines 12, 13, and 14).

The code was written in VSCode, and a Python environment version 3.7.16 was set up in Anaconda for execution. The code is available online at [44]. The simulation parameters and the infrastructure setup are shown in Tables 3 and 4,

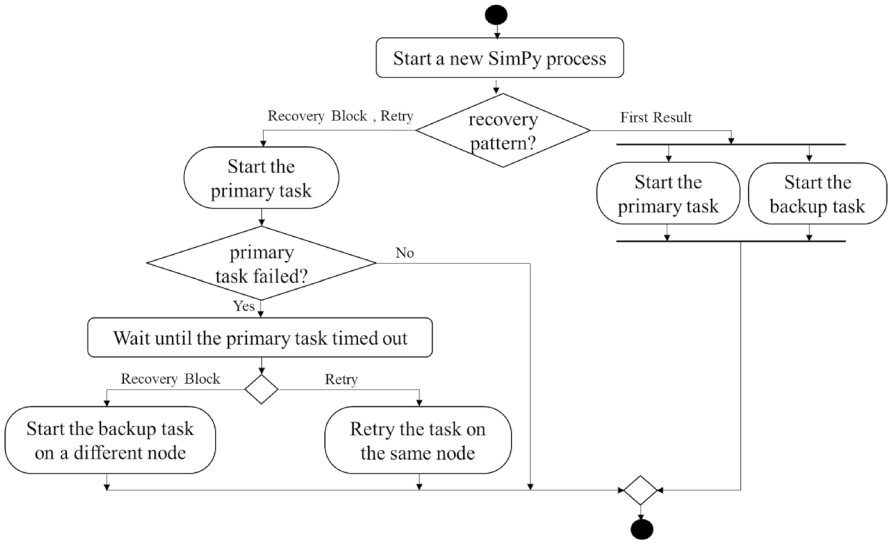

Fig. 5 Starting a new *SimPy* process (step 11)

Table 4 Infrastructure setup

Node type	Number of nodes	Processing frequency (instruction/ sec)	Node to RSU delay (sec)	Failure rates
Edge	6	Uniform (10, 15)	0	Low: (0.00894, 0.01553) High: (0.02689, 0.08254)
Cloud	2	Uniform (30, 60)	Uniform (1.25, 12.5) *	Low: (0.00776, 0.00799) High: (0.00821, 0.00827)

*The RSU to cloud bandwidth is set as 8 Mb/sec

respectively. Notably, the cloud failure rates are assumed to be much lower than the edge failure rates as the cloud servers provide a reliable service. However, the RSU-to-Cloud communication delay is assumed to be much higher than the RSU-to-Edge communication delay. The value of 8 Mbps is chosen for RSU-to-Cloud bandwidth. This value represents a practical scenario, where the RSU has a poor connection to the internet, leading to an unstable and less reliable RSU-to-Cloud network. This limitation is not due to issues on the cloud side (cloud side is assumed reliable in MECC architecture) but rather results from factors such as wireless backhaul constraints, network congestion, or infrastructural limitations affecting the RSU's internet connectivity. In real-world deployments, RSUs may rely on cellular or other wireless connections, which can suffer from variable bandwidth and higher latency compared to direct fiber-optic links.

7.2 Evaluation metrics and baseline methods

To assess the performance of the proposed method in comparison with the competing methods, the following metrics were used:

- *Average Reward* By moving a sliding window over 40 most recent learning episodes, this metric is calculated by averaging the episodic rewards within the sliding window. This metric evaluates the performance of a task offloading method in terms of both the tasks' execution delays and the tasks' failure numbers over episodes (see the reward calculation formula (30).
- *Average number of task failures* This metric calculates the average number of failed tasks during the last 40 episodes. Similar to the reward metric, a sliding window of size 40 is moved over the previous learning episodes. This metric provides a clear view of the system's fault tolerance.

These metrics were applied to compare the proposed method with three baseline methods: 1) A Q-learning-based FT task offloading method (denoted as ReLIEF) presented in [16], which employs a primary-backup task assignment strategy to enhance reliability in a fog computing environment, 2) A DDPG-based optimal task offloading method implemented based on the recovery model presented in [21], where failures are assumed to be recoverable, and the failed tasks are recovered by retrying their execution on the same node (denoted as RetOnly); and 3) A DDPG-based optimal task offloading method presented in [42] that uses no failure recovery strategy (denoted as NoRec).

7.3 Simulation results

To address the research questions, three experiments were conducted, each repeated five times. The average results of these repetitions were then analyzed. The experimental details are outlined in the following subsections.

7.3.1 First experiment: addressing RQ1, RQ2 and RQ3

In this experiment, the proposed DDPG-based FT task offloading method was compared against three baseline methods: ReLIEF, RetOnly, and NoRec (described in Sect. 7–2). The simulation program was executed for each method under both high and low node failure rate scenarios. The experimental settings are shown in Tables 3 and 4. As shown in Fig. 6, the proposed method outperformed ReLIEF, RetOnly, and NoRec by 32%, 16%, and 39%, respectively, in terms of average episodic reward value under high failure rate conditions. A higher reward value indicates a shorter completion time for offloaded tasks and fewer failed tasks, as per the reward formula (see Sect. 6). Figure 7 plots the number of task failures during the execution episodes. As shown in this figure, utilizing various

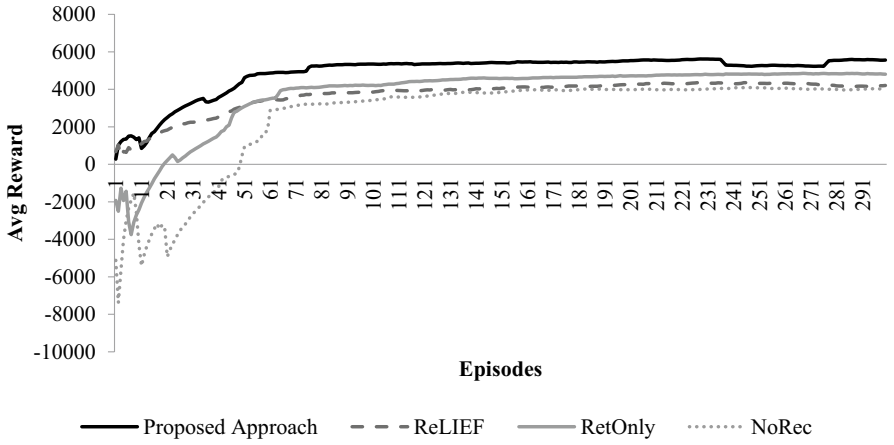

Fig. 6 Average episodic reward in the high failure rate scenario

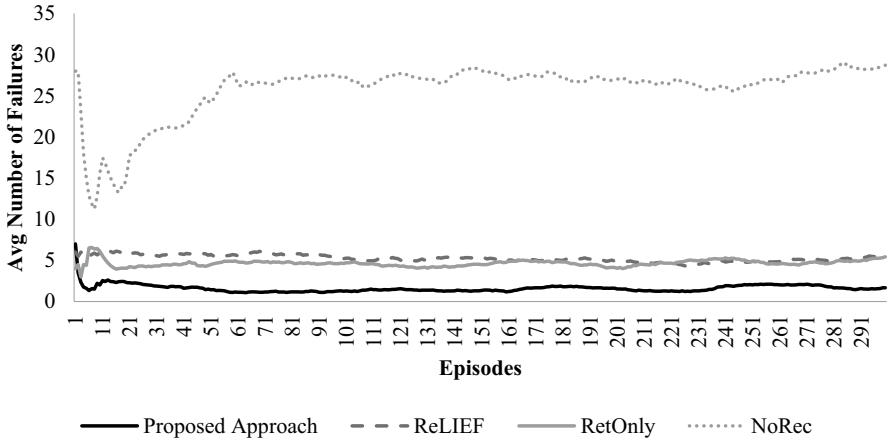

Fig. 7 Average number of failed tasks in the high failure rate scenario

recovery patterns (RT, RB, and FR) in the proposed method could decrease the number of failures significantly (converging to a value of 1.7).

As shown in Figs. 8 and 9, under low failure rates, the proposed method and RetOnly exhibit comparable performance. This suggests that the additional recovery patterns (RB and FR) in the proposed method offer minimal benefit in reducing task completion delays or failures compared to RetOnly, which relies solely on RT for recovery.

It was observed that in both high/low failure scenarios, NoRec and ReLIEF show significantly lower rewards. The lower performance of NoRec is expected as it lacks an effective error recovery pattern. ReLIEF despite utilizing a recovery pattern, struggles to improve rewards due to its reliance on a Q-learning model. This method operates on discrete state spaces, which limits its ability to accurately capture the

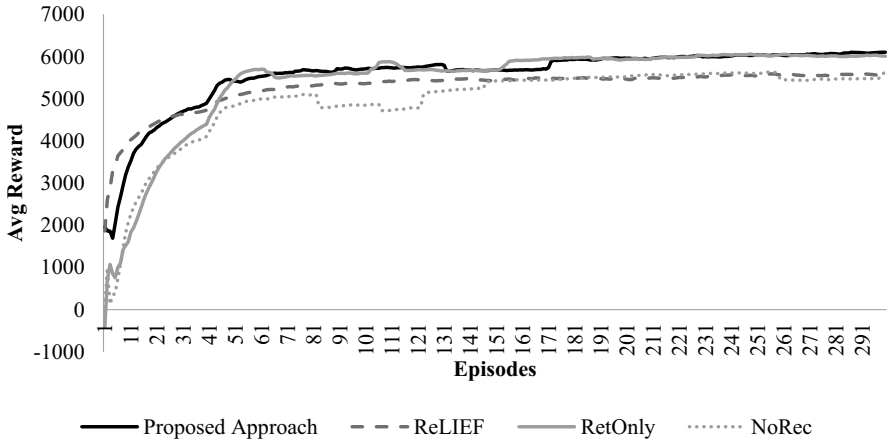

Fig. 8 Average reward in the low failure rate scenario

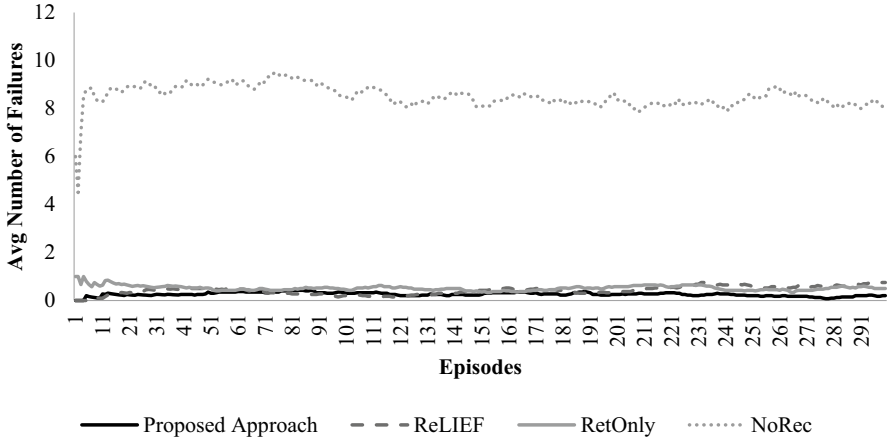

Fig. 9 Average number of failed tasks in the low failure rate scenario

continuous state of the system, unlike the proposed method and RetOnly, which are based on DDPG and can more effectively adapt to dynamic environments.

To see the impact of node failure rates on the placement distribution of primary/backup tasks, the number of primary/backup tasks executed on each node under high and low failure rates was recorded during the proposed method's simulation. As shown in Fig. 10, under high failure rates, the proposed DDPG algorithm places more tasks on the cloud layer compared to the low failure rates scenario.

7.3.2 Second experiment: addressing RQ4

This experiment aimed to address RQ4: How does infrastructure resource capacity affect recovery pattern selection? The simulation program was run using the proposed

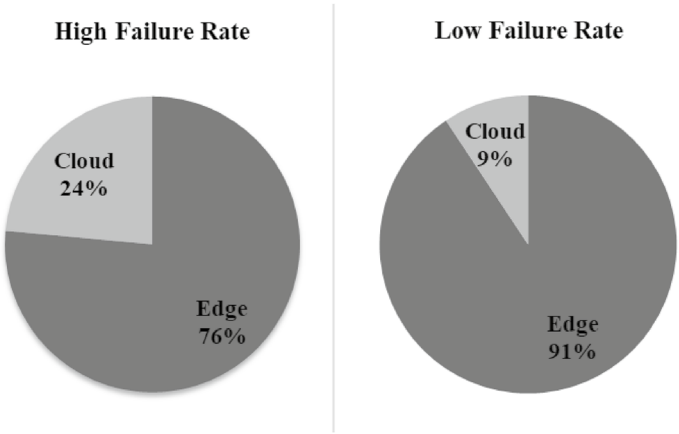

Fig. 10 The distribution of tasks on the edge/cloud layers in the high/low failure rate scenarios

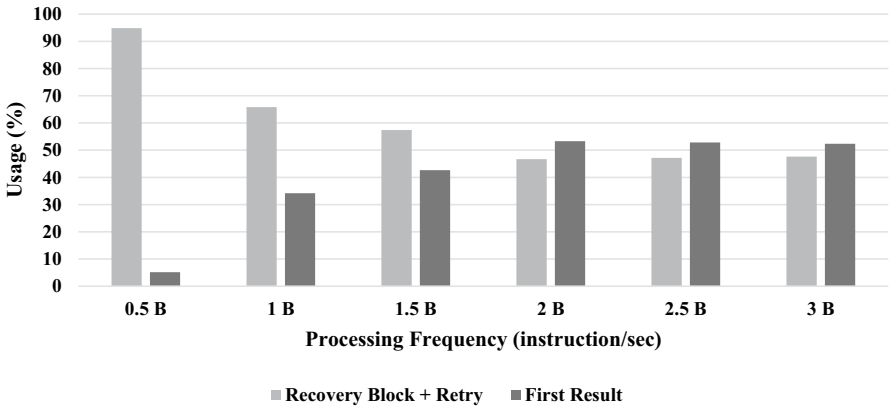

Fig. 11 Impact of computation capacity on suggested recovery patterns

method under high failure rates with varying node processing frequencies. The base processing frequency (Table 4) was multiplied by factors of 0.5, 1, 1.5, 2, 2.5, and 3 to create six different frequency ranges for cloud and edge nodes (0.5B, 1B, 1.5B, 2B, 2.5B, and 3B in Fig. 11). For each range, the simulation was repeated, and the recommended recovery patterns at the final episode were recorded. Our primary focus was on the DDPG algorithm's tendency to suggest FR. As Fig. 11 demonstrates, increasing edge or cloud node processing frequencies leads to a greater preference for FR recovery. Conversely, sequential recovery patterns (RT and RB) become less likely to be selected.

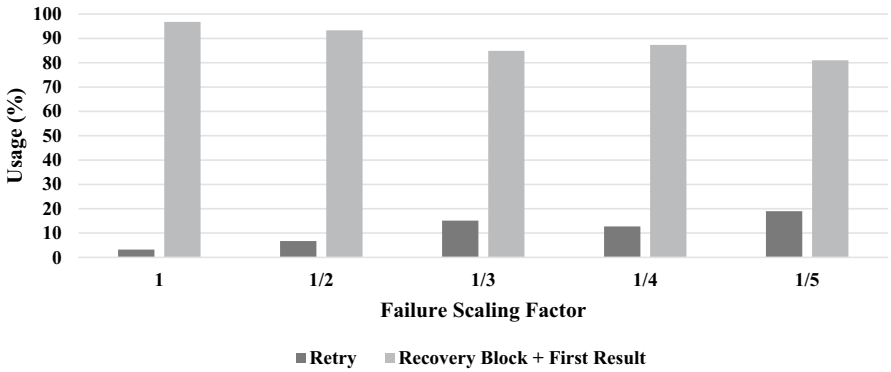

Fig. 12 Impact of failure transience degree on suggested recovery patterns

7.3.3 Third experiment: addressing RQ5

The third experiment aimed to investigate how different failure types influence the selection of recovery patterns. Specifically, we were interested in the suggestion percentage of RT patterns under transient and permanent failure conditions. Transient failures are temporary disruptions that do not damage the underlying hardware or software. They often result from external factors or network congestion. Examples include software/OS glitches, network congestion, and intermittent power outages. In contrast, permanent failures are more severe issues requiring hardware or software repairs. They can be caused by physical damage, software corruption, or hardware failures. We focused on the RT pattern in this experiment because it is the simplest and most common failure recovery method. The simulation program was run using the proposed DDPG algorithm under varying degrees of transience in task failures.

To model different degrees of transience, we scaled the failure rates of nodes by factors of 1, 1/2, 1/3, 1/4, and 1/5 in successive simulations when retrying failed tasks. The reasoning behind this is that in an environment with transient failures, retrying a failed task on the same node is more likely to succeed. Conversely, retrying a failed task in an environment with permanent failures is generally not helpful. As shown in Fig. 12, the suggestion percentage of RT patterns increased as the transience of failures in the environment increased. This is because, by lowering the failure rate after the first execution attempt, the failed task has a higher chance of success when retried on the same node.

7.3.4 Fourth experiment: scalability analysis

We conducted this experiment to analyze the scalability of the proposed method from two aspects:

- *Increase in the number of tasks* We measured the agent's overhead for different numbers of tasks, and the results are shown in Fig. 13. As shown

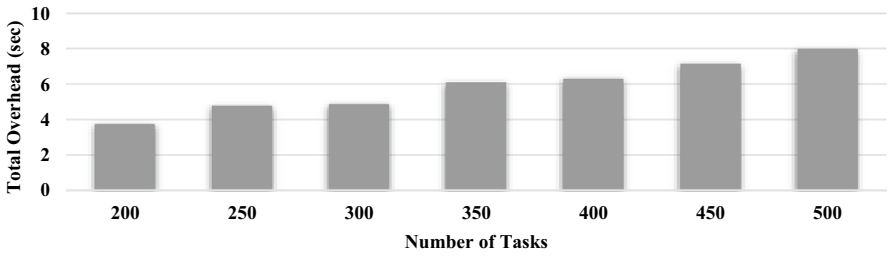

Fig. 13 The DDPG agent total overhead

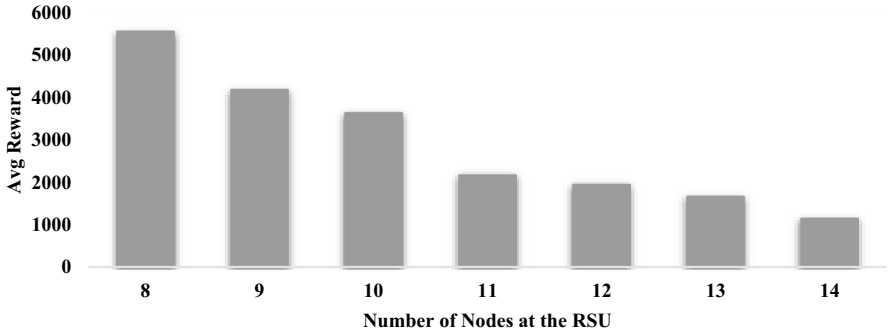

Fig. 14 DDPG algorithm performance for different numbers of nodes at the RSU

in this figure, the overhead of actor-critic networks for action generation and learning from past experiences during the simulation is very low, even in scenarios with a large number of tasks (in the range of [15, 18] milliseconds per task on the test-bed of this paper).

- *Increase in the number of nodes at the RSU* As explained in Sect. 6, the number of actions of the DDPG actor-network increases quadratically with the number of available nodes at the RSU, which is a limiting factor and could adversely affect the performance of the DDPG network when there are too many nodes at an RSU. We measured the performance of the proposed method for different numbers of available nodes at an RSU, and the results are shown in Fig. 14. As expected, it was observed that the DDPG performance decreases when splitting the same processing capacity among more nodes. However, in practice, usually, the large number of nodes at the edge layer is divided among many RSUs due to the distributed nature of the proposed architecture (shown in Fig. 2). As a result, each RSU hosts only a fraction of the total nodes at the edge layer.

7.4 Discussion

The evaluation of the proposed method across various scenarios yielded several key insights. By dynamically employing a diverse set of failure recovery patterns (RT, RB, and FR), the system was able to explore a wider range of potential task execution plans, leading to solutions that effectively managed failures and optimized task completion delays within the MECC environment (Addressing RQ1 and RQ2). It is notable that, since task failures increase response latency by at least the duration of the timeout, the proposed method prioritizes solutions with a lower likelihood of failure. Under conditions of high failure rates, our method demonstrated a clear advantage over the ReLIEF, RetOnly, and NoRec approaches in both maximizing rewards and minimizing task failures. In the low failure rate scenario, although the proposed method still outperformed ReLIEF and NoRec, the performance gap with RetOnly was negligible due to the efficiency of the RT pattern in recovering the limited number of failed tasks. Obviously, the superiority of the proposed method to the NoRec method in high/low failure scenarios is due to the fact that this method lacks an effective error recovery pattern. ReLIEF, despite utilizing a primary-backup recovery model, still showed low reward values in high/low failure scenarios due to its reliance on a Q-learning model. This model operates on discrete state spaces, which limits its ability to accurately capture the continuous state of the system, unlike the proposed method, which is based on DDPG that works on continuous state space. DDPG uses an actor-critic architecture, where the actor chooses actions based on the current state, and the critic evaluates the action. This allows DDPG to adapt more smoothly to dynamic, continuous environments like vehicular networks, where the system's state and the available resources are constantly changing.

Furthermore, under high failure rates, the proposed method tended to allocate more workload to the cloud layer due to its reliability compared to edge nodes, albeit at the cost of increased communication delays for offloaded tasks (Addressing RQ3). The computation capacity of edge/cloud servers also influenced the pattern selection using the proposed method. As server frequency increased, the model favored the FR pattern, leveraging available resources to prioritize rapid and successful task completions by doubling resource allocation compared to sequential patterns. Conversely, with reduced server frequency, the model shifted towards sequential recovery strategies like RB and RT, balancing the need for efficiency with resilience in resource-constrained environments. This adaptability underscores the method's ability to optimize recovery pattern selection based on available resources, ensuring effective task offloading and fault tolerance in dynamic MECC environments (Addressing RQ4). Additionally, the proposed method favored the RT pattern in environments with transient failures, demonstrating its ability to suggest efficient recovery patterns when retried tasks have a high probability of success (Addressing RQ5).

We also analyzed the scalability of the proposed method from two aspects: 1) Increase in the number of tasks and 2) Increase in the number of nodes at the RSU. Regarding the first aspect, the DDPG algorithm showed very low overhead, even in scenarios with a large number of tasks. With respect to the increase in the number of nodes at the RSU, we measured the performance of the proposed method for different

numbers of available nodes at each RSU. As explained in Sect. 6, the number of actions of the DDPG actor-network increases quadratically with the number of available nodes at the RSU, which is a limiting factor and could adversely affect the performance of the DDPG network when there are many nodes at an RSU. However, in practice, usually, the large number of nodes at the edge layer is divided among many RSUs due to the distributed nature of the proposed architecture (shown in Fig. 2). As a result, each RSU hosts only a fraction of the total nodes at the edge layer.

8 Limitations

We identified the following limitations for the proposed approach:

- *Tasks with hard deadlines* We have not considered tasks with hard deadlines in this study. However, the proposed DDPG solution to the online offloading problem can be extended to support hard deadlines by redefining the reward function.
- *Result delivery failure* There might be a failure in delivering a computation result to a vehicle if it leaves the coverage range of the RSU to which the task was originally offloaded. This type of failure is not addressed in this study, and we plan to explore it in future work.
- *Fixed number of nodes* The total number of computational nodes installed on an RSU is typically assumed to remain fixed in the long-term. Changing the number of computational nodes would require redesigning and retraining the DDPG component, making it impractical. Instead, we recommend increasing the computational capacity of the RSU through vertical scaling—adding more resources to the existing nodes rather than changing their number.
- *Integration with existing vehicular environments* There are limitations regarding the integration of the proposed DRL-based approach with existing vehicular environments, which may present certain challenges. Firstly, there is a need for RSU infrastructure equipped with computational resources, such as edge computing nodes, as outlined in the proposed architecture of this paper. Furthermore, there are training considerations: training the DRL model in real-world scenarios can be challenging due to dynamic environments and limited available training data. To address this, we propose leveraging simulation-based pre-training, followed by fine-tuning the model on real-world data through transfer learning or online learning techniques. This would ensure that the approach remains adaptable to real-world conditions, while minimizing deployment costs and time.

9 Conclusions and future research directions

To bridge the research gap of missing recovery strategy suggestions in previous vehicular task offloading studies, this paper introduced a DRL-based fault-tolerant task offloading method for MECC environments. Our goal was to minimize the

average response latency of all tasks under faulty conditions. A key contribution was the Fault-aware Execution Planner (FEP), which suggests optimal Task Execution Plans (TEPs) for incoming vehicular tasks considering their reliability requirements. FEP explores a vast state space of potential offloading solutions, incorporating diverse failure recovery patterns to create optimal TEPs. We developed an analytical model of the optimization problem and implemented FEP using a Deep Deterministic Policy Gradient (DDPG) algorithm. This DDPG algorithm determines the optimal deployment and recovery strategies for delay-sensitive tasks across the MECC infrastructure.

Our results demonstrated that by suggesting appropriate failure recovery strategies in high failure environments, our proposed method could significantly reduce the total task completion time and the number of failures. Compared to baseline methods, we observed an average reduction of 29% in total task completion time and 78% in the number of failures. Moreover, the DDPG-based fault-tolerant task offloading method exhibited adaptability to changes in available computational resources and failure transience degrees.

As future work, we aim to enhance the model's adaptability to highly dynamic environments, where failure rates and types fluctuate during task offloading episodes. Additionally, extending the model to tolerate other types of failures in vehicular networks, particularly those related to vehicle mobility, is a promising area for further research. Moreover, the dependency between tasks, tasks' priorities, and deadlines are not considered in the proposed solution, which could be addressed in future works.

Authors' contributions V. B. was involved in software, writing-original draft preparation. O. B. was involved in conceptualization, methodology, validation, writing-reviewing and editing.

Funding This research did not receive any specific grant from funding agencies in the public, commercial, or not-for-profit sectors.

Data availability The datasets generated and analyzed during the current study are available from the corresponding author on reasonable request.

Declarations

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

Ethical approval This article does not contain any studies with human participants or animals performed by any of the authors.

References

1. Liu L, Feng J, Mu X, Pei Q, Lan D, Xiao M (2023) Asynchronous deep reinforcement learning for collaborative task computing and on-demand resource allocation in vehicular edge computing. IEEE Trans Intell Transp Syst 24(12):15513–15526

2. Liu Z, Dai P, Xing H, Yu Z, Zhang W (2021) A distributed algorithm for task offloading in vehicular networks with hybrid fog/cloud computing. IEEE Trans Syst, Man, Cybern: Syst 52(7):4388–4401
3. Shi W, Cao J, Zhang Q, Li Y, Xu L (2016) Edge computing: vision and challenges. IEEE Internet Things J 3(5):637–646
4. Cao D et al (2024) A relay-assisted parallel offloading strategy for multi-source tasks in internet of vehicles. Sustain Energy Technol Assess 62:103619
5. Bute MS, Fan P, Zhang L, Abbas F (2021) An efficient distributed task offloading scheme for vehicular edge computing networks. IEEE Trans Veh Technol 70(12):13149–13161
6. Zhang P, Zhou M, Fortino G (2018) Security and trust issues in fog computing: a survey. Futur Gener Comput Syst 88:16–27
7. Umer A, Ali M, Jehangiri AI, Bilal M, Shuja J (2024) Multi-objective task-aware offloading and scheduling framework for internet of things logistics. Sensors 24(8):2381
8. Zaman SK, Jehangiri AI, Maqsood T, Haq NU, Umar AI, Shuja J, Ahmad Z, Dhaou IB, Alsharekh MF (2023) LiMPO: Lightweight mobility prediction and offloading framework using machine learning for mobile edge computing. Clust Comput 26(1):99–117
9. Sellami B, Hakiri A, Yahia SB (2022) Deep reinforcement learning for energy-aware task offloading in join SDN-Blockchain 5G massive IoT edge network. Futur Gener Comput Syst 137:363–379
10. Shahryari O-K, Pedram H, Khajehvand V, TakhtFooladi MD (2021) Energy and task completion time trade-off for task offloading in fog-enabled IoT networks. Pervasive Mob Comput 74:101395
11. Min H, Rahmani AM, Ghaderkourehpaz P, Moghaddasi K, Hosseinzadeh M (2025) A joint optimization of resource allocation management and multi-task offloading in high-mobility vehicular multi-access edge computing networks. Ad Hoc Netw 166:103656
12. Chen CL, Bhargava B, Aggarwal V, Tonshal B, Gopal A (2023) A hybrid deep reinforcement learning approach for jointly optimizing offloading and resource management in vehicular networks. IEEE Trans Veh Technol 73(2):2456–2467
13. Binh TH, Vo H, Nguyen BM, Binh HT (2023) Reinforcement learning for optimizing delay-sensitive task offloading in vehicular edge–cloud computing. IEEE Internet Things J 11(2):2058–2069
14. Dai P, Huang Y, Hu K, Wu X, Xing H, Yu Z (2023) Meta reinforcement learning for multi-task offloading in vehicular edge computing. IEEE Trans Mob Comput 23(3):2123–2138
15. Mishra K, Rajareddy GN, Ghugar U, Chhabra GS, Gandomi AH (2023) A collaborative computation and offloading for compute-intensive and latency-sensitive dependency-aware tasks in dew-enabled vehicular fog computing: a federated deep Q-learning approach. IEEE Trans Netw Serv Manage 20(4):4600–4614
16. Siyadatzadeh R et al (2023) ReLIEF: a reinforcement-learning-based real-time task assignment strategy in emerging fault-tolerant fog computing. IEEE Internet Things J 10(12):10752–10763
17. Rahmani AM et al (2025) Optimizing task offloading with metaheuristic algorithms across cloud, fog, and edge computing networks: a comprehensive survey and state-of-the-art schemes. Sustain Comput: Inform Syst 45:101080
18. Wang Y, Zhang P, Wang B, Zhang Z, Xu Y, Lv B (2025) A hybrid PSO and GA algorithm with rescheduling for task offloading in device–edge–cloud collaborative computing. Clust Comput 28(2):101
19. Arulkumar V, Lathamanju R, Nithya T, Rajendran T (2025) Enhancing task scheduling process in fog computing using GTO-SSSA: a metaheuristic approach. J Intell Syst Internet Things 14(1):114
20. Kar B, Yahya W, Lin Y-D, Ali A (2023) Offloading using traditional optimization and machine learning in federated cloud–edge–fog systems: a survey. IEEE Commun Surv Tutor 25(2):1199–1226
21. Tang C, Yan G, Wu H, Zhu C (2023) Computation offloading and resource allocation in failure-aware vehicular edge computing. IEEE Trans Consum Electron 70(1):1877–1888
22. van Steen M and Tanenbaum AS (2023) *Distributed Systems*. Amazon Digital Services LLC - Kdp.
23. Silver D, Lever G, Heess N, Degris T, Wierstra D, and Riedmiller M (2014) Deterministic policy gradient algorithms. In: *International conference on machine learning*, Pmlr, pp. 387–395.
24. Lillicrap T, Continuous control with deep reinforcement learning, *arXiv preprint* arXiv:1509.02971, 2015.
25. Li H, Li X, Zhang M, Ulziinyam B (2024) System-wide energy efficient computation offloading in vehicular edge computing with speed adjustment. IEEE Trans Green Commun Netw 8(2):701–715
26. Ma Q, Xu H, Wang H, Xu Y, Jia Q, Qiao C (2023) Fully distributed task offloading in vehicular edge computing. IEEE Trans Veh Technol 73(4):5630–5646

27. Materwala H, Ismail L, Hassanein HS (2023) QoS-SLA-aware adaptive genetic algorithm for multi-request offloading in integrated edge-cloud computing in Internet of vehicles. Veh Commun 43:100654
28. Zeng J, Gou F, Wu J (2023) Task offloading scheme combining deep reinforcement learning and convolutional neural networks for vehicle trajectory prediction in smart cities. Comput Commun 208:29–43
29. Wan N, Luo Y, Zeng G, Zhou X (2023) Minimization of VANET execution time based on joint task offloading and resource allocation. Peer-to-Peer Netw Appl 16(1):71–86
30. Xu X, Tang S, Qi L, Zhou X, Dai F, Dou W (2023) CNN partitioning and offloading for vehicular edge networks in Web3. IEEE Commun Mag 61(8):36–42
31. Fan W et al (2023) Joint task offloading and resource allocation for vehicular edge computing based on v2i and v2v modes. IEEE Trans Intell Transp Syst 24(4):4277–4292
32. Lin Z, Chen X, He X, Tian D, Zhang Q, Chen P (2024) Energy-efficient cooperative task offloading in NOMA-enabled vehicular fog computing. IEEE Trans Intell Transp Syst 25(7):7223–7236
33. Yang J, Chen Y, Lin Z, Tian D, Chen P (2023) Distributed computation offloading in autonomous driving vehicular networks: a stochastic geometry approach. IEEE Trans Intell Veh 9(1):2701–2713
34. Li P, Xiao Z, Wang X, Huang K, Huang Y, Gao H (2023) EPtask: deep reinforcement learning based energy-efficient and priority-aware task scheduling for dynamic vehicular edge computing. IEEE Trans Intell Veh 9(1):1830–1846
35. Cong Y, Xue K, Wang C, Sun W, Sun S, Hu F (2023) Latency-energy joint optimization for task offloading and resource allocation in MEC-assisted vehicular networks. IEEE Trans Veh Technol 72(12):16369–16381
36. Xia Y, Zhang H, Zhou X, Yuan D (2023) Location-aware and delay-minimizing task offloading in vehicular edge computing networks. IEEE Trans Veh Technol 72(12):16266–16279
37. Zheng D, Wang L, Kai C, Peng M (2023) Resource optimization for task offloading with real-time location prediction in pedestrian-vehicle interaction scenarios. IEEE Trans Wirel Commun 22(11):7331–7344
38. da Costa JB, de Souza AM, Meneguette RI, Cerqueira E, Rosário D, Sommer C, Villas L (2023) Mobility and deadline-aware task scheduling mechanism for vehicular edge computing. IEEE Trans Intell Transp Syst 24(10):11345–11359
39. Ren M, Fu X, Pace P, Aloi G, Fortino G (2023) Collaborative data acquisition for UAV-aided IoT based on time-balancing scheduling. IEEE Internet Things J 11(8):13660–13676
40. Fu X, Zheng D, Liu X, Xing L, Peng R (2024) Systematic review and future perspectives on cascading failures in Internet of Things: modeling and optimization. Reliab Eng Syst Saf 254:110582
41. Ray K, Banerjee A (2022) Prioritized fault recovery strategies for multi-access edge computing using probabilistic model checking. IEEE Trans Dependable Secure Comput 20(1):797–812
42. Li Y, Qi F, Wang Z, Yu X, Shao S (2020) Distributed edge computing offloading algorithm based on deep reinforcement learning. IEEE Access 8:85204–85215
43. SimPy T (2017) Simpy: Discrete event simulation for python, Tech. Rep. 9, 2017, URL https://simpy.readthedocs.io/en/latest.
44. Babaiyan V(2024) Fault-Tolerant Task Offloading Simulation Using DDPG, GitHub, URL https://github.com/vahide-b-84/FaultTolerantTaskOffloadingSimulation.

Authors and Affiliations

Vahide Babaiyan[1] · Omid Bushehrian[1]

✉ Omid Bushehrian
bushehrian@sutech.ac.ir

Vahide Babaiyan
v.babaiyan@sutech.ac.ir

[1] Department of Computer Engineering and Information Technology, Shiraz University of Technology, Shiraz, Iran