OJ Logo

Received XX Month, XXXX; revised XX Month, XXXX; accepted XX Month, XXXX; Date of publication XX Month, XXXX; date of current version 9 November, 2024.

HOODIE: Hybrid Computation Offloading via Distributed Deep Reinforcement Learning in Delay-aware Cloud-Edge Continuum

ANASTASIOS GIANNOPOULOS[1] *(Member, IEEE)*, ILIAS PARALIKAS[2], SOTIRIOS SPANTIDEAS[1], AND PANAGIOTIS TRAKADAS[1]

[1]Department of Ports Management and Shipping, National and Kapodistrian University of Athens, Psachna, Evia, P.C. 34400, Greece
[2]Research & Development Department, Four Dot Infinity, Chalandri, Athens, P.C. 15231, Greece

CORRESPONDING AUTHOR: ANASTASIOS GIANNOPOULOS (e-mail: angianno@uoa.gr).

This work was partially supported by the "Towards a functional continuum operating system" (ICOS) Project funded from the European Union's HORIZON research and innovation programme under grant agreement No 101070177.

ABSTRACT Cloud-Edge Computing Continuum (CEC) system, where edge and cloud nodes are seamlessly connected, is dedicated to handle substantial computational loads offloaded by end-users. These tasks can suffer from delays or be dropped entirely when deadlines are missed, particularly under fluctuating network conditions and resource limitations. The CEC is coupled with the need for hybrid task offloading, where the task placement decisions concern whether the tasks are processed locally, offloaded vertically to the cloud, or horizontally to interconnected edge servers. In this paper, we present a distributed hybrid task offloading scheme (HOODIE) designed to jointly optimize the tasks latency and drop rate, under dynamic CEC traffic. HOODIE employs a model-free deep reinforcement learning (DRL) framework, where distributed DRL agents at each edge server autonomously determine offloading decisions without global task distribution awareness. To further enhance the system pro-activity and learning stability, we incorporate techniques such as Long Short-term Memory (LSTM), Dueling deep Q-networks (DQN), and double-DQN. Extensive simulation results demonstrate that HOODIE effectively reduces task drop rates and average task processing delays, outperforming several baseline methods under changing CEC settings and dynamic conditions.

INDEX TERMS Cloud computing, cloud-edge continuum, cognitive network, deep reinforcement learning, edge computing, internet of things, task offloading.

I. INTRODUCTION

THE Cloud-Edge Computing Continuum (CEC) represents a distributed synergistic computing paradigm that integrates cloud computing resources with edge computing resources to offer a unified, seamless computational environment for the underlying end-devices [1]. Given that user equipment is battery-sensitive and resource-constrained, the CEC is designed to support a wide range of applications by leveraging the combined capabilities of centralized cloud data centers and decentralized edge nodes, offering computational capacities closer to the data source and end-users [2]. By seamlessly integrating Cloud-Edge layers, the CEC enables more responsive and scalable services, accommodating the increasing demands of modern applications such as Internet of Things (IoT), smart cities, and cognitive autonomous systems [3]. The primary advantages of the CEC include reduced latency, improved bandwidth efficiency, and the ability to support real-time applications. By processing data at the edge, CEC reduces the need for data to traverse the network to central cloud servers, which in turn minimizes latency and enhances the user experience. This paradigm allows for the dynamic multi-scale allocation of resources across the continuum, optimizing both latency-sensitive and computation-intensive tasks through intelligent task distribution and resource management strategies.

Despite its advantages, the CEC introduces several challenges, particularly in the context of task handling and load balancing. One of the key challenges is the heterogeneity of computational resources and the load dynamics across the CEC, which can lead to complex decision-making processes when determining where to offload tasks. Additionally, the dynamic and distributed nature of edge environments makes it difficult to predict resource availability and network conditions, further complicating task scheduling and load balancing [4].

Furthermore, the CEC environment demands "beyond verticalized" task offloading (from Edge to Cloud), imposing the need for shifting towards a hybrid (vertical and horizontal/peer-to-peer) task offloading [1]. This hybrid approach, enabled by the seamless connectivity within the CEC, allows tasks to flow not just vertically to the Cloud, but also horizontally across different Edge nodes. This flexibility introduces new dimensions to the task offloading problem, requiring the development of more sophisticated, distributed and collaborative strategies that can dynamically adapt to varying network conditions, resource availability, and application requirements. The complexity of managing both vertical and horizontal task flows necessitates a reconsideration of traditional task offloading models, which were primarily designed for simpler, vertically-oriented systems [5]. These challenges are exacerbated by the need to meet strict latency requirements and to manage limited resources and load dynamics at both Edge and Cloud.

A. TOWARDS HYBRID TASK OFFLOADING AND DISTRIBUTED ML

In the context of CEC, task offloading involves dynamically deciding whether to process tasks locally at the edge, offload them to another edge node (horizontal offloading) or to a cloud server (vertical offloading), under time-varying task arrival traffic. The primary goal of task offloading is to optimize system performance by minimizing target indicators, such as latency, energy consumption, and task drop ratio, to prevent bottlenecks and ensure smooth and efficient flow of CEC operations. As an optimization problem, task offloading is constrained by several factors, including limited computational resources at the Edge, bandwidth limitations, and strict latency requirements. These constraints often lead to non-convexity of the optimization objective, where the goal is to minimize the overall task completion delay or drop rate while ensuring that tasks meet their deadlines. The non-convexity arises from the interdependent nature of offloading decisions, where the optimal choice for one task depends on the offloading decisions of other tasks, as well as the dynamic state of network resources and conditions [6].

Traditionally, task offloading in CEC has been predominantly "verticalized," where tasks are offloaded directly from users to nearby Mobile Edge Computing (MEC) servers, and if these servers are overloaded, the tasks are further offloaded to the Cloud for processing [7]. While this approach can manage task execution within certain latency bounds, it falls short in fully leveraging the potential of the CEC [5]. To overcome these limitations, there is a growing need for a more flexible approach, referred as hybrid task offloading. This approach allows tasks to be dynamically transferred not only vertically but also horizontally across edge nodes, ensuring more balanced resource utilization and reducing the likelihood of overloading any single node. Enabling hybrid task flows is particularly beneficial in practical CEC considerations, where multiple edge nodes have varying levels of computational power and network connectivity [8].

Hybrid task offloading in the CEC can be significantly enhanced by the use of distributed Machine Learning (ML) agents. These agents can make autonomous ML-driven offloading decisions by continuously learning from the CEC environment. As opposed to centralized ML, distributed agents require low-size state/action space representation, as they can operate autonomously at different nodes within the CEC, adapting to local conditions while contributing to global optimization. By leveraging techniques such as Deep Reinforcement Learning (DRL) [9], distributed agents can deal with the non-convexity of the task offloading problem, dynamically adjusting their strategies to minimize delays, and prevent task drops.

B. RELATED WORK

This subsection summarizes the outcomes of existing studies addressing the task offloading problem, categorizing them in heuristic methods, centralized learning and distributed learning algorithms.

1) Heuristic Methods

Heuristic algorithms have been widely explored in the context of task offloading, with most studies focusing in MEC, and limited studies in fog-cloud environments. These algorithms often focus on finding near-optimal solutions by simplifying the complex task offloading environment, and relaxing the constraints of the non-convex optimization. For instance, Wang et al. [10] proposed a heuristic-based algorithm for minimizing energy consumption in MEC by intelligently offloading tasks based on the current load and available resources. Similarly, Sardellitti et al. [11] developed a joint optimization algorithm for radio and computational resources, which relies on heuristic methods to balance the computational load across the edge and cloud. Other approaches have also focused on specific metrics such as latency and energy efficiency. Mao et al. [12] introduced a heuristic task offloading algorithm aimed at minimizing the total task completion time by dynamically adjusting the offloading strategy based on real-time network conditions. You et al. [13] presented an energy-efficient resource allocation strategy for mobile-edge computation offloading that utilizes heuristic optimization to reduce the computational burden on edge devices.

TABLE 1. Acronyms

Acronym	Meaning
A&V	Advantage & Value layer
BCO	Ballanced Cyclic Offloader
CPU	Central Processing Unit
CEC	Cloud-Edge Continuum
DM	Decision Maker
DQL	Deep Q-Learning
DQN	Deep Q-Network
DRL	Deep Reinforcement Learning
EA	Edge Agent
EC	Edge Controller
FIFO	First In First Out
FLC	Full Local Computing
HO	Horizontal Offloader
HOODIE	Hybrid Computation Offloading via Distributed Deep Reinforcement Learning in Delay-aware Cloud-Edge Continuum
ID	Identifier
IoT	Internet of Things
LSTM	Long Short-Term Memory
MEC	Mobile Edge Computing
ML	Machine Learning
MLEO	Minimum Latency Estimation Offloader
MSE	Mean Squared Error
Pub-Sub	Publisher-Subscriber Protocol
ReLU	Rectified Linear Unit
RO	Random Offloader
TRL	Tabular Reinforcement Learning
VO	Vertical Offloader

While these heuristic algorithms provide significant insights and performance improvements, they often fall short in addressing the dynamic and highly variable nature of CEC environments. Given that heuristic methods often require iterative-based algorithms of polynomial (or higher) complexity to derive solutions, their applicability in the online network operation is questionable. The main limitations include a lack of adaptability to changing network conditions, inability to operate in large-scale networks such that of CEC, and reliance on predefined rules or approximations that may not generalize well across different scenarios.

2) Centralized ML Methods

Centralized ML approaches have also been extensively researched to tackle the task offloading problem in MEC. These methods typically involve training a central model to predict optimal decisions based on historical data or real-time global inputs from all individual nodes. For instance, Chen et al. [14] proposed a DRL-based framework for dynamic task offloading, which trains a central DRL agent to optimize task allocation in MEC. Similarly, Li et al. [15] introduced a centralized DRL approach for task offloading to avoid resource over-distribution based on resource reservation in MEC. Another notable work is the DRL-based offloading framework by Wang et al. [16], which focuses on meta-learning for fast policy convergence to minimize application latency in MEC. Their approach demonstrates the potential of centralized ML models to handle complex offloading scenarios under changing environment. The AutoScale method [17], [18] proposes an RL method to optimally select the execution target for model inference to minimize energy consumption, while also penalizing for accuracy losses due to quantization at the target. It considers different types co-processors, given the characteristics of the neural model. The authors in [19] propose a DRL-based dynamic framework considering the 5G CEC, where deep neural networks are used to predict the optimal task offloading strategies based on real-time system metrics, optimizing the energy consumption of 5G applications. There are also works [20] studying the optimal service function chaining by considering offloading decisions in the CEC for fluid task distribution, as well as Kubernetes-based scheduling for price cost minimization and service relocation [21].

Despite their advantages, centralized ML approaches face scalability limitations, as the central model must handle a large number of edge devices and tasks, leading to potential bottlenecks. Given the centralization of data collection that is required, centralized ML may also exhibit pronounced security concerns and bandwidth overhead. Additionally, these methods often assume that the central model produce accurate decisions for each distributed node that handles, not offering any level of personalized learning on the local task traffic. Moreover, centralized models typically lack the flexibility to adapt to hybrid offloading scenarios involving both vertical and horizontal task flows, thereby limiting their effectiveness in inter-connected CEC environments.

3) Distributed ML Methods

In response to the limitations of centralized approaches, distributed ML involve deploying ML models across multiple edge nodes, allowing each node to make local decisions while also collaborating with other nodes to optimize global performance. For example, He et al. [22] developed a federated learning-based approach for edge computing, where agents train local models and periodically aggregate them to create a global model. This method enables task offloading decisions that are both privacy-preserving and adaptive to local conditions. In another study, Tang et al. [23] proposed a distributed DRL framework where mobile devices independently learn optimal offloading strategies to locally compute tasks or vertically offload them to edge servers. Another study [24] introduces a DRL-based distributed task offloading framework aimed at balancing task execution between edge servers and cloud resources for efficient use of computational resources across the edge-cloud continuum. In another work [25], the authors propose an offloading scheme

where the agents learn to cooperate by exchanging information about their local states, aiming to better distribute the tasks and reduce the delays from bottlenecks. Similarly, other works presented a collaborative learning approach that leverages edge-cloud synergy to offload tasks from end-devices either to edge or cloud nodes in real-time [26], [27].

Despite advancements, existing distributed ML approaches also present limitations, mainly that many of these models do not fully align with the CEC principles, particularly in supporting hybrid task flows that involve both vertical and horizontal offloading. Most existing distributed ML approaches focus primarily on either IoT-to-cloud, IoT-to-edge offloading, or edge-to-cloud (vertical) collaboration. This lack of hybrid task flow support can lead to suboptimal resource utilization and increased latency in more complex CEC scenarios. Additionally, federated learning models often require frequent communication between nodes to synchronize their learning processes [28], which may introduce additional delays and network traffic. Lastly, the coordination overhead between distributed agents can become significant, especially in large-scale networks, potentially offsetting the benefits of distributed learning.

C. PROPOSED SCHEME AND CONTRIBUTIONS

In this paper, we propose a Hybrid Computation Offloading via Distributed Deep Reinforcement Learning (HOODIE), appropriate for delay-aware decision-making in CEC. Under a multi-server topology graph of CEC, HOODIE leverages distributed and autonomous DRL agents to intelligently assist on task placement decisions. The objective of HOODIE is to jointly optimize task computation delay and CEC throughput (or minimal drop rate) by dynamically balancing hybrid task flows across CEC resources. Unlike existing offloading schemes that rely on (i) iterative approaches with high inference delay (as in heuristic methods), (ii) huge amount of data exchange in central locations (as in centralized ML), (iii) periodical learning synchronization (as in federated learning), and (iv) vertical task flows engaging battery-sensitive end-devices (as in distributed ML in MEC), HOODIE enables dual functionality at edge nodes, allowing them to host both local and external workloads arrived from IoT layer. To reduce complexity, each HOODIE agent does not require knowledge on the decisions made by other agents. The optimization problem is formulated with known task timeouts and other task features under stochastic traffic at each agent. To solve this problem, HOODIE state input received both local task characteristics and forecasts about the upcoming load of the CEC nodes.

The contributions of the present work can be summarized as follows:

- An analytical modelling and problem formulation is presented for the mathematical description of the hybrid task offloading in CEC. HOODIE particularly fits to scenarios where both edge and cloud resources need to be efficiently managed to support latency-sensitive and resource-intensive applications of the end-devices.
- HOODIE algorithm is fully aligned with CEC principles to jointly minimize task latency and drop rate, supporting both vertical (edge-to-cloud) and horizontal (edge-to-edge) task flows. This is achieved by enabling dual functionality at each agent (i.e. each agent may serve either as local task processor or as host for offloaded tasks). This hybrid model significantly enhances resource utilization and avoids bottlenecks.
- The proposed scheme enforces autonomy in the distributed decision-making, with each agent not knowing the decisions of the other agents. In this context, HOODIES facilitates the process for removing old or adding new agents in CEC.
- HOODIE agents exploit the Double and Dueling DRL techniques for learning stability [29]. To proactively adapt to time-varying workloads, the proposed algorithm considers as DRL inputs both local task features and forecasts about the upcoming load in the CEC.
- During the training, HOODIE considers only single-step offloading decisions in order to enforce each agent provide fast decisions and avoid ping-pong effects. Also, the delay introduced in multi-hop offloading strategies is eliminated. Under this option, HOODIE simplifies the interfacing requirements (needed in multi-scale CEC) for multi-hop offloading and, thus, can be deployed in multi-agent CEC systems.
- Extensive analyses and simulations were conducted to showcase the learning process of the HOODIE scheme and its complexity, as well as to quantify its scalability when the Edge layer in densified by multiple servers. Also, quantitative simulations were carried out to provide HOODIE behavior insights under dynamic task traffic settings and comparisons against baseline algorithms.

The paper is structured as follows. In Section II, all the system model elements are analytically specified. Section III formulates the optimization problem for distributed DRL-based hybrid offloading. In Section IV, we outline the proposed HOODIE scheme, along with its complexity and convergence analyses. Finally, Section V includes all the numerical outcomes for training and validating HOODIE performance, whereas Section VI contains the work summary and extensions. To enhance readability, Table 1 lists all the acronyms used in this article.

II. SYSTEM MODEL

In this section, we thoroughly describe all the systemic, architectural and functional principles underlying the proposed decentralized Cloud-Edge computing system. We first present the overall system model and then we outline each of the system elements and components. Since the rest of the methodology uses multiple mathematical symbols, Table

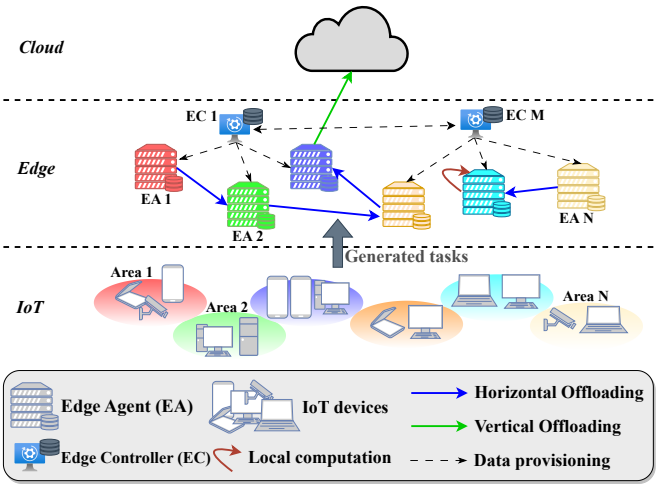

FIGURE 1. **A three-layer computing system model where each Edge Agent is able to locally compute its assigned tasks or to offload the tasks either horizontally towards another Edge Agent or vertically towards the Cloud.**

2 summarizes all the mathematical notations for the ease of exposure.

A. SYSTEM ARCHITECTURE

As shown in Fig.1, a three-layer (IoT-to-Edge-to-Cloud) network with N Edge Agents (EAs) and one[1] Cloud is considered for covering multiple IoT areas. The network supports Edge Computing (via the multiple EAs) and Cloud computing (via the Cloud) capabilities. Let the set of EAs plus the Cloud be $\mathcal{N} = \{1, 2, \ldots, N, N+1\}$, where the element $N+1$ indexes the Cloud. For convenience, we also let $\mathcal{E} = \mathcal{N} - \{N+1\} = \{1, 2, \ldots, N\}$ denote the set including only the EA indices. Each EA $n \in \mathcal{E}$ is responsible for computing all tasks (or applications) generated by the respective IoT area. Each task is assumed to be non-divisible, meaning that it should be computed by an EA or the Cloud as a whole (no partial computation of the same task by multiple EAs). Considering densified 6G-compliant network traffic (i.e. massive devices request the computation of numerous tasks), an efficient decision-making policy for the task computations is required.

To enable flexibility in the computation decisions, the system model allows each EA to select among three actions: (i) to compute the task *locally* based on its CPU processing capacity, (ii) to offload the task *horizontally* to another EA, or (iii) to offload the task *vertically* to the Cloud. Each task is also associated with a timeout index that poses the maximum waiting time of the task until computation. To ensure intelligent and decentralized decision-making, each EA runs a DRL model which receives task features and traffic information of the other EAs, with the aim to output an offloading decision that jointly minimize (i) the task throw ratio (respecting the timeout of the tasks) and (ii) the task

[1]Here we consider a single Cloud entity without loss of generality. The system model can be easily modified to support multiple Cloud entities.

TABLE 2. **Mathematical symbols**

Symbol	Meaning [Unit]	Symbol	Meaning [Unit]
$\mathcal{N}$	Set of EAs and Cloud	$\psi_n^{priv}(t)$	Completion time slot of private task $u_n(t)$
$\mathcal{E}$	Set of EAs	$w_n^{priv}(t)$	Waiting time of private task $u_n(t)$ [time slots]
$\mathcal{M}$	Set of ECs	$f_n^{EA,priv}$	Private Processing Capacity of EA n [cycles/sec]
$\mathcal{T}$	Set of time slots	$\mathcal{R}$	Set of link data rates
$\mathcal{H}$	Set of task sizes	R_H	Horizontal data rate [bps]
N	Number of EAs	R_V	Vertical data rate [bps]
M	Number of ECs	$R_{n,k}^{off}$	Data rate between EA n and node k [bps]
T	Number of time slots	$w_n^{off}(t)$	Waiting time of offloading task $u_n(t)$ [time slots]
Δ	Time slot duration [sec]	$\psi_n^{off}(t)$	Completion time slot of offloading task $u_n(t)$
$\mathcal{P}$	Task arrival probability	$u_{n,k}^{pub}(t)$	Task ID offloaded by EA n to node k at time slot t
$x_n(t)$	Task arrival binary index	$\eta_{n,k}^{pub}(t)$	Size of task $u_{n,k}^{pub}(t)$ [bits]
$u_n(t)$	Task ID arrived in EA n at time slot t	$l_{n,k}^{pub}(t)$	Length of public queue n of node k at time slot t [bits]
$\eta_n(t)$	Size of task $u_n(t)$ [bits]	$\mathcal{A}_k(t)$	Set of active queues of node k at time slot t
$\rho_n(t)$	Processing density of task $u_n(t)$ [CPU cycles/bit]	$A_k(t)$	Number of active public queues of node k at time slot t
$\phi_n(t)$	Timeout of task $u_n(t)$ [time slot]	$f_n^{EA,pub}$	Public processing capacity of EA n [cycles/sec]
$d_n^{(1)}(t)$	Binary decision for local computing or offloading for task $u_n(t)$	f^{Cloud}	Cloud processing capacity [cycles/sec]
$d_{n,k}^{(2)}(t)$	Binary decision for offloading task $u_n(t)$ from EA n to node k	$m_{n,k}^{pub}(t)$	Size of the tasks thrown by public queue n of EA k at time slot t [bits]
$\mathbf{D}_n(t)$	Decision tuple for task $u_n(t)$	$\psi_{n,k}^{pub}(t)$	Completion time slot of task $u_{n,k}^{pub}(t)$
$U_n(t)$	Number of tasks arrived in node n at time slot t	$\tilde{\psi}_{n,k}^{pub}(t)$	Computation starting time slot of task $u_{n,k}^{pub}(t)$
$\mathbf{L}(t)$	Historical load matrix at time slot t	W	Lookback window of load history [time slots]
$\mathbf{l}_n^{pub}(t)$	Public queues length hosting tasks offloaded by EA n at time slot t	Λ	Set of public queue length values [bits]
$\mathbf{G}$	Adjacent matrix for Edge topology	$G(i,j)$	Binary variable indicating connection between EA i and EA j

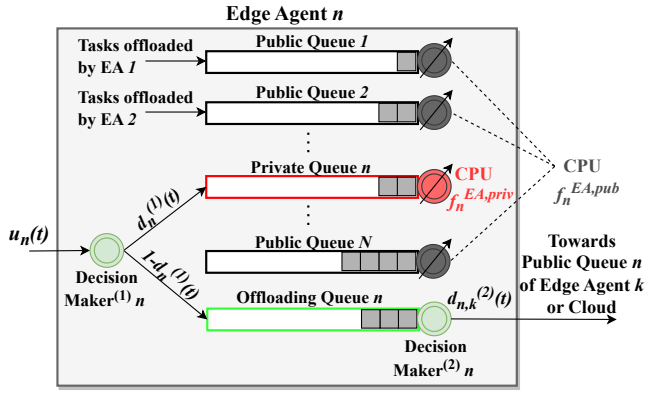

FIGURE 2. The internal structure of a single Edge Agent that includes 1 private queue (for local task computation), $N-1$ public queues (for computation of offloaded tasks) and 1 offloading queue (for task offloading).

latency (fast processing of the tasks) of the system. We also assume M Edge Controllers (ECs) that are distributed in the Edge layer. In this system model, each EC is in charge of monitoring a single EA ($M = N$, i.e. each EA has an associated controller) or a cluster of EAs ($M < N$). ECs are also able to transfer data from an EA to another, whereas inter-controller communication is also allowed to enable data provisioning across different clusters of EAs[2]. For the following, we focus on one episode that includes a finite set of time slots $\mathcal{T} = \{1, 2, \ldots, T\}$, with each time slot $t \in \mathcal{T}$ having a duration Δ (in seconds).

From the communication perspective, we assume that IoT devices are served by an area-specific base station through wireless links (e.g. LTE, 5G), while each base station $n \in \mathcal{E}$ is connected with the EA $n \in \mathcal{E}$ through a wired fronthaul link (e.g. optical fiber)[3]. Also, all EAs communicate with the Cloud through e.g. the Internet (uploading/downloading information).

The internal structure of a single EA is depicted in Fig.2. At the beginning of each time slot, EA n has a new task arrival with a certain probability $\mathcal{P}$ [30]. Additionally, each EA has dual behavior, meaning that it can either (i) act as local computing node for the tasks associated with the corresponding IoT area or (ii) host computational tasks that are offloaded by another EA. To this end, each EA $n \in \mathcal{E}$ is equipped with N FIFO computation queues to execute tasks, and 1 offloading queue to stack the tasks to be offloaded. For EA n, the computation queue n is called *private queue* because it is used for computing the local tasks, whereas the rest of $N-1$ computation queues are called *public queues* because they are used to host external tasks. The public queue $n' \in \mathcal{E} - \{n\}$ of each EA $n \in \mathcal{E}$ stacks the tasks offloaded by EA n'. For instance, if EA 5 offloads a task to EA 3, then the offloaded task is placed in public queue 5 of EA 3[4]. The Cloud includes only N public queues, with its public queue $n \in \mathcal{E}$ hosting the tasks forwarded by EA n. We assume also that when the computation (or offloading) of a task is completed in a time slot, then the next task in the queue will be computed (or offloaded) at the beginning of the next time slot [30].

B. TASK CHARACTERISTICS

We let $u_n(t) \in \mathbb{Z}_+$ denote the unique task identifier (ID) assigned to the task arrived in EA $n \in \mathcal{E}$ at $t \in \mathcal{T}$. At a given time slot t, we use the binary variable $x_n(t)$ to denote if a new local task is arrived in EA n as:

$$x_n(t) = \begin{cases} 1, & \text{if a new task arrived in EA } n \text{ at time } t \\ 0, & \text{otherwise} \end{cases} \tag{1}$$

Under this definition, the task identifier (ID) can be written as $u_n(t) = x_n(t) \cdot u_n(t)$, i.e. the task ID is set to zero if no task arrived. We also define the size of task $u_n(t)$ as $\eta_n(t)$ (in bits). The values of task size are drawn from the discrete set $\mathcal{H} = \{\eta_1, \eta_2, \ldots, \eta_{|\mathcal{H}|}\}$. The size of task $u_n(t)$ can be also written as $\eta_n(t) = x_n(t) \cdot \eta_n(t)$, meaning that the size of task is zero when there is no local task arrived in EA n at time slot t. According to the above-mentioned notations, we have $\eta_n(t) \in \mathcal{H} \cup \{0\}, \forall n \in \mathcal{E}$. Furthermore, each task $u_n(t)$ is associated with a processing density $\rho_n(t)$ (in CPU cycles per bit) and a timeout index ϕ_n (in time slots). The former defines the number of CPU cycles required to process a unit of data, whereas the latter implies that the task $u_n(t)$ should have been computed until the time slot $t + \phi_n - 1$, otherwise it is dropped.

C. OFFLOADING DECISION MODEL

Once a new task $u_n(t)$ arrives in EA $n \in \mathcal{E}$, it requires a decision-making process regarding the computation destination of the task. The offloading decision-making is achieved via two decision maker (DM) modules which make two successive decisions. The first-level DM$^{(1)}$ is placed at the entrance of EA and the second-level DM$^{(2)}$ is located at the exit of the offloading queue (see Fig.2). Specifically, DM$^{(1)}$ decides whether the task is going to be computed locally or to be offloaded. Given a new task, EA n places the new task either in the private queue (if DM$^{(1)}$ decides local task computation) or in the offloading queue (if DM$^{(1)}$ decides task offloading). If a task has been placed in the offloading queue n, DM$^{(2)}$ decides the offloading destination

[2]The Edge Controller represents a logical entity that is able to collect and share metrics within an edge cluster using monitoring tools and interfaces. In practice, it can be in the same or different physical location from the EAs that it monitors.

[3]Without loss of generality, it is also possible to assume that the base station and the respective Edge Agent are co-located.

[4]This association policy between the offloading EA and the destination (public) queue avoids task collision that may occur when multiple tasks are offloaded to the same EA at the same time. Practically, a unique wired connection is established between the offloading EA $n \in \mathcal{E}$ and the destination queue n of computing node $k \in \mathcal{N}$.

node $k \in \mathcal{N} - \{n\}$ to which the task will be computed. Formally, the output of $\text{DM}^{(1)}$ is:

$$d_n^{(1)}(t) = \begin{cases} 1, & u_n(t) \text{ is placed in private queue} \\ 0, & u_n(t) \text{ is placed in offloading queue} \end{cases} \quad (2)$$

where $d_n^{(1)}(t)$ is the binary decision taken by $\text{DM}^{(1)}$ $n \in \mathcal{E}$ for task $u_n(t)$. Moreover, the output of $\text{DM}^{(2)}$ is:

$$d_{n,k}^{(2)}(t) = \begin{cases} 1, & \text{if } d_n^{(1)}(t) = 0 \ \& \ u_n(t) \text{ is offloaded} \\ & \text{from EA } n \text{ to node } k \\ 0, & \text{if } d_n^{(1)}(t) = 1 \text{ (local computation)} \end{cases} \quad (3)$$

where $d_{n,k}^{(2)}(t)$ is the binary decision taken by $\text{DM}^{(2)}$ $n \in \mathcal{E}$ and denotes that, if the task is not locally computed (i.e. $d_n^{(1)}(t) = 0$), the task $u_n(t)$ will be offloaded from the offloading queue of EA $n \in \mathcal{E}$ to the public queue $n \in \mathcal{E}$ of computation node $k \in \mathcal{N} - \{n\}$ (i.e. the task can be offloaded towards an EA or Cloud and the task cannot be offloaded to a public queue of the source EA).

Consequently, the number of bits occupied in the private queue (and offloading queue) of EA n at time slot t is equal to $d_n^{(1)} \cdot \eta_n(t)$ (and $(1 - d_n^{(1)}) \cdot \eta_n(t)$). Also, a task can be offloaded to at most one destination node, as reflected by the next formula:

$$\sum_{k \in \mathcal{N} - \{n\}} d_{n,k}^{(2)}(t) \leq 1, \forall n \in \mathcal{E} \quad (4)$$

Furthermore, the number of tasks arrived at the computation queues (private plus public queues) of computation node $n \in \mathcal{N}$ at time slot t is given by:

$$U_n(t) = \begin{cases} d_n^{(1)}(t) + \sum_{i \in \mathcal{E} - \{n\}} d_{i,n}^{(2)}(t), & \text{if } n \in \mathcal{E} \\ \sum_{i \in \mathcal{E}} d_{i,n}^{(2)}(t), & \text{if } n = N + 1 \end{cases} \quad (5)$$

where $U_n(t)$ is differently calculated for EAs (they have both private and public queues) and the Cloud (it has only public queues). The first branch reflects the number of tasks arrived in EA n at time slot t as the sum of the tasks that are going to be locally computed (first term) plus the tasks that are offloaded by other EAs (second term). The second branch represents the number of tasks arrived in the Cloud at time slot t as the sum of the offloaded tasks that have the Cloud as computing destination. Finally, we define a decision tuple $\mathbf{D}_n(t) = \left(d_{n,k}^{(2)}, k \in \mathcal{N} - \{n\}\right)$ which reflects the computing destination of task $u_n(t)$. For example, if task $u_2(t)$ is offloaded to EA 3, then $\mathbf{D}_2(t) = \Big[(0,1),(1,3),(0,4),\ldots,(0,N),(0,N+1)\Big]$.

D. PRIVATE QUEUE MODEL

Each task $u_n(t)$ that is going to be locally processed is placed in the FIFO private queue n of EA $n \in \mathcal{E}$. The local tasks stored in the private queue of EA n are processed in the local Central Processing Unit (CPU) with a processing capacity of $f_n^{EA,priv}$ CPU cycles per second (or Hz). With no loss of generality, $f_n^{EA,priv}$ is is assumed constant for each EA $n \in \mathcal{E}$.

Assuming that a given task $u_n(t)$ is placed in the private queue at time slot $t \in \mathcal{T}$, we designate $\psi_n^{priv}(t)$ as the task completion time slot. When no task is placed in the private queue at time t, we set $\psi_n^{priv}(t) = 0$. Note that completing a task means that it is either successfully processed before the respective timeout or it is overdue and, thus, it is thrown. To derive the completion time slot of the task $u_n(t)$, it is necessary to know how long the task remained in the private queue. To this end, we define $w_n^{priv}(t)$ as the waiting time (or the number of time slots) of task $u_n(t)$ in the private queue until completion. This means that, if task $u_n(t)$ is decided to be locally computed, it will stay for $w_n^{priv}(t)$ time slots in the private queue[5]. For a given task $u_n(t)$ placed in the private queue of EA $n \in \mathcal{E}$ at time slot t, the waiting time before completion can be given by:

$$w_n^{priv}(t) = \max\left\{0, \max_{t' < t}\{\psi_n^{priv}(t')\} - t + 1\right\}, \forall n \in \mathcal{E} \quad (6)$$

where the outer $\max\{\cdot\}$ operation ensures that negative values of $w_n^{priv}(t)$ are set to zero, whereas the inner $\max\{\cdot\}$ operation reflects the completion time slot of the previous task with the highest completion time. Specifically, the waiting time of the task $u_n(t)$ is equal to the (positive) difference between the completion time slot of the most time-consuming previous task (placed in the private queue at $t' \in \{0, 1, \ldots, t-1\}$) and the arrival time slot of task $u_n(t)$. Based on the above, it is also feasible to calculate the completion time slot of task $u_n(t)$ if it is placed in the private queue at time slot t, as follows:

$$\psi_n^{priv}(t) = \min\Big\{t + w_n^{priv}(t) + \left\lceil \frac{\eta_n(t) \cdot \rho_n(t)}{f_n^{EA,priv} \cdot \Delta} \right\rceil - 1, \\ t + \phi_n(t) - 1\Big\} \quad (7)$$

where the first argument of the $\min\{\cdot\}$ operation is the time slot of the task completion when the task is locally processed, whereas the second argument is the time slot when the task is thrown. Also, $\lceil \cdot \rceil$ denotes ceiling operation. Specifically, the task $u_n(t)$ starts to be processed at time slot $t + w_n^{priv}(t)$ and the processing duration is $\left\lceil \frac{\eta_n(t) \cdot \rho_n(t)}{f_n^{EA,priv} \cdot \Delta} \right\rceil$ time slots. Otherwise, if the sum of the waiting and the processing duration exceeds the task timeout, the task will be thrown.

For instance, if a new task arrived in EA 3 at time slot 5 and it is decided to be locally computed, then $u_3(5)$ is placed in the private queue. Suppose that previous tasks $u_3(1)$, $u_3(2)$, $u_3(3)$ and $u_3(4)$ have already been placed in the private queue for local computation

[5]Notably, the value of $w_n^{priv}(t)$ is computed by the EA n before placing the task $u_n(t)$ in the private queue.

with completion time slots $\psi_3^{priv}(1) = 2$, $\psi_3^{priv}(2) = 6$, $\psi_3^{priv}(3) = 3$ and $\psi_3^{priv}(4) = 10$, respectively. To calculate the waiting time of $u_3(5)$, we apply (6) taking $w_3^{priv}(5) = \max\{0, \max\{2, 6, 3, 10\} - 5 + 1\} = 6$ time slots for $u_3(5)$ to wait until start of processing.

E. OFFLOADING QUEUE MODEL

Similar to the private queues, the offloading queue of each EA is a FIFO queue which is responsible for stacking the local tasks to be offloaded. Once a local task $u_n(t)$ of EA $n \in \mathcal{E}$ is selected for offloading, the offloading queue of the source EA is connected to the target public queue of the destination EA or the Cloud through a wired link[6]. Since it is not always possible to assume all-to-all connectivity across EAs in practice, we can let the symmetrical matrix $\mathbf{G}$ denote the adjacent graph of the Edge layer topology, with each element $G(i,j) = G(j,i) = 1$ when EA i can establish direct bidirectional connection with EA j, whereas $G(i,j) = 0$ otherwise. When all-to-all connectivity is assumed, all elements of matrix $\mathbf{G}$ are equal to 1[7]. Without loss of generality, the link data rate for the horizontal EA-to-EA communication is assumed constant for each pair of EAs and notated as R_H, whereas the link data rate of the vertical EA-to-Cloud communication is R_V, and it is also assumed constant for each EA-Cloud pair. Let also $\mathcal{R} = \{R_H, R_V\}$ denote the set of available data rates, where R_H and R_V are measured in $bits/sec$ with $R_H > R_V$. If EA $n \in \mathcal{E}$ offloads the $u_n(t)$ to the computation node $k \in \mathcal{N} - \{n\}$ (another EA or Cloud), the data rate $R_{n,k}^{off}$ of the communication link is calculated as follows:

$$R_{n,k}^{off} = \begin{cases} R_H, & \text{if } k \in \mathcal{E} - n \\ R_V, & \text{if } k = N+1 \end{cases}, \forall n \in \mathcal{E} \tag{8}$$

where the first (and second) branch corresponds to the data rate of the horizontal EA-to-EA (and the vertical EA-to-Cloud) communication link.

If we assume a task $u_n(t)$ that is placed in the offloading queue of EA n at the beginning of time slot t, the waiting time in the offloading queue $w_n^{off}(t)$ and the task completion time slot $\psi_n^{off}(t)$ can be defined, as in the private queue case. Note that the completion time slot for offloading the task is equal to the time slot number when the task is either sent to the destination node or dropped. For example, $\psi_6^{off}(5) = 16$ means that the task arrived at $t = 5$ in EA 6 and placed in the offloading queue will be completed at $t = 16$. The number of time slots required by the task $u_n(t)$ to wait in the offloading queue (until completion) can be computed as follows:

$$w_n^{off}(t) = \max\left\{0, \max_{t'<t}\{\psi_n^{off}(t')\} - t + 1\right\}, \forall n \in \mathcal{E} \tag{9}$$

$w_n^{off}(t)$ reflects the number of time slots required for waiting in the offloading queue[8]. Thus, as implied by (9), $w_n^{off}(t)$ cannot be negative and is equal to the difference between the completion time slot of the most time-consuming previous task $u_n(t')$ ($\forall t' \in \{0, 1, \ldots, t-1\}$) and the arrival time of task $u_n(t)$. We set $\psi_n^{off}(t) = 0$ if there is no task for offloading at time slot t. To compute the completion time slot of a task $u_n(t)$ that is placed in the offloading queue of EA n, the following formula should be used:

$$\begin{aligned} \psi_n^{off}(t) = \min\Big\{ & t + w_n^{off}(t) + \\ & + \left\lceil \sum_{k \in \mathcal{N}-\{n\}} \frac{d_{n,k}^{(2)}(t) \cdot \eta_n(t)}{R_{n,k}^{off} \cdot \Delta} \right\rceil - 1, \\ & t + \phi_n(t) - 1 \Big\}, \forall n \in \mathcal{E} \end{aligned} \tag{10}$$

where $d_{n,k}^{(2)}$ is equal to 1 if and only if EA n offloads to node k. It is evident that (10) computes the completion time slot as the minimum between the time slot for successfully offloading the task and the time slot of the task timeout. The time slot when the task is successfully offloaded to the destination node is equal to the sum of the arrival time slot t, the waiting time $w_n^{off}(t)$ and the offloading time $\left\lceil \sum_{k \in \mathcal{N}-\{n\}} \frac{d_{n,k}^{(2)}(t) \cdot \eta_n(t)}{R_{n,k}^{off} \cdot \Delta} \right\rceil$. Finally, $R_{n,k}^{off} = R_V$ when $d_{n,N+1}^{(2)}(t) = 1$ (task is offloaded towards the Cloud), otherwise $R_{n,k}^{off} = R_H$ (task is offloaded towards another EA).

F. PUBLIC QUEUE MODEL

Each EA $n \in \mathcal{E}$ has $N-1$ public queues to host the processing of external tasks offloaded by other EAs. The Cloud entity maintains N public queues, being capable of hosting the processing of the tasks offloaded by all EAs. Each public queue is matched with the offloading EA, meaning that the public queue i of a given node acts as the offloading destination of EA $i \in \mathcal{E}$. For convenience, it is assumed that, if a task offloaded by EA $n \in \mathcal{E}$ arrives at the destination node $k \in \mathcal{N} - \{n\}$ at time slot $t \in \mathcal{T}$, then it is placed in the FIFO public queue n for processing at time slot $t+1$. Upon placement of the offloaded task $u_n(t)$ in the public queue n of EA k at time slot t, a new task ID $u_{n,k}^{pub}(t) \in \mathbb{Z}_+$ is assigned to the task, ensuring that the task with source node n, destination node k and queue placement time slot t has a unique identifier. The task ID assignment rule follows the next equation:

[6]For the ease of exposure, here we assumed wired connections between EAs and EAs-Cloud. This assumption can be easily modified without loss of generality.

[7]Under dynamic Edge layer topologies, matrix $\mathbf{G}$ can be assumed as time-varying (i.e. $\mathbf{G} = \mathbf{G}(t)$), but here we assume constant adjacency between EAs.

[8]The value of $w_n^{off}(t)$ is calculated by the EA n before making the decision on which queue the task $u_n(t)$ is placed.

$$u_{n,k}^{pub}(t) = \begin{cases} u_n(t'), & \text{if EA } n \text{ offloaded to node } k \\ 0, & \text{otherwise} \end{cases} \quad (11)$$

where $u_n(t')$ stands for the task ID arrived at EA $n \in \mathcal{E}$ at time slot $t' \in \{0, 1, \ldots, t-1\}$. This means that the ID assigned to the task at the public queue of node k is identical with the original task ID assigned when the task was initially arrived in the source node n.

We also define $\eta_{n,k}^{pub}(t) \in \mathcal{H} \cup \{0\}$ as the size (in bits) of the task arrived in the public queue $n \in \mathcal{E}$ of node $k \in \mathcal{N} - \{n\}$ at time slot $t \in \mathcal{T}$, meaning that it is the size of task $u_{n,k}^{pub}(t)$. For completeness, if $u_{n,k}^{pub}(t) = 0$ then $\eta_{n,k}^{pub}(t) = 0$. Let also $l_{n,k}^{pub}(t)$ represent the length (in bits) of the public queue n of node k at the end of the time slot t. For instance, if the third public queue of the Cloud has two tasks of 8 bits each at the end of time slot 14, then we set $l_{3,N+1}^{pub}(14) = 16$ bits.

A public queue n of node k is called *active queue* at time slot t if and only if either a new task is inserted to the public queue at time slot t (i.e. $\eta_{n,k}^{pub}(t) > 0$) or there are queued tasks at the end of the previous time slot (i.e. $l_{n,k}^{pub}(t-1) > 0$). Therefore, the set of the active public queues of node $k \in \mathcal{N}$ at time slot $t \in \mathcal{T}$ is defined as:

$$\mathcal{A}_k(t) = \begin{cases} \Big\{ n | \eta_{n,k}^{pub}(t) > 0 \text{ or } l_{n,k}^{pub}(t-1) > 0, \ldots \\ n \in \mathcal{E} - \{k\} \Big\}, \text{if } k \in \mathcal{E} \text{ (for EAs)} \\ \Big\{ n | \eta_{n,k}^{pub}(t) > 0 \text{ or } l_{n,k}^{pub}(t-1) > 0, \ldots \\ n \in \mathcal{E} \Big\}, \text{if } k = N+1 \text{ (for Cloud)} \end{cases} \quad (12)$$

where the first branch is the set of the active queues for an EA and the second branch is the set of the active queues for the Cloud. Evidently from (12), the set $\mathcal{A}_k(t)$ contains the indices of the public queues with pending tasks that have been offloaded by the respective source EAs to the node k. The number of elements of set $\mathcal{A}_k(t)$ is $A_k(t) = |\mathcal{A}_k(t)|$ and is equal to the number of active public queues of node k at time slot t.

Apart from the CPU for running the local tasks $f_n^{EA,priv}$, each EA $n \in \mathcal{E}$ is equipped with another public CPU $f_n^{EA,pub}$ (in Hz) for processing the tasks stacked in the public queues. Without loss of generality[9], we let $f_1^{EA,pub} = f_2^{EA,pub} = \cdots = f_N^{EA,pub}$, whereas the Cloud has a CPU of f^{Cloud} Hz, with $f^{Cloud} > f_n^{EA,pub}$, $\forall n \in \mathcal{E}$. Assuming no priority across EAs[10], the public CPU of each EA n (or the Cloud) is equally allocated to all the active public queues, which means that the processing capacity allocated to an active task $u_{n,k}^{pub}(t')$ ($t' < t$) at time slot t is $f_k^{EA,pub}/A_k(t)$ (or $f^{Cloud}/A_k(t)$). This equal distribution of the processing capacity amongst the active public queues follows the principle of the generalized processor sharing model [31]. Under this definition, the processing capacity of the public queues dynamically depends on the number of active queues at each time slot and cannot be known in advance. Instead, each EA is only aware of the public CPU processing capacity of the other EAs $f_n^{EA,pub}$ and the Cloud f^{Cloud}.

Letting $m_{n,k}^{pub}(t)$ denote the size (in bits) of the tasks thrown by the public queue $n \in \mathcal{E}$ of node $k \in \mathcal{N} - \{n\}$ at the end of time slot t, the public queue length can be retrospectively updated as:

$$l_{n,k}^{pub}(t) = \max\Big\{0, l_{n,k}^{pub}(t-1) + \eta_{n,k}^{pub}(t) - \\ - m_{n,k}^{pub}(t) - \frac{\Delta \cdot f_k^{EA,pub}}{\rho_n(t) \cdot A_k(t)}\Big\}, \quad (13)$$

where the number of bits of the public queue $n \in \mathcal{E}$ of EA $k \in \mathcal{E} - \{n\}$ at the end of time slot t is equal to number of bits stayed at the queue minus the number of bits left the queue. Specifically, $l_{n,k}^{pub}(t)$ is equal to the length of the public queue at the end of the previous time slot (first term), plus the number of bits arrived at time slot t (second term), minus the number of bits dropped by the queue at the end of the time slot t (third term), minus the number of bits processed between $t-1$ and t (fourth term). To compute $l_{n,N+1}^{pub}(t)$ (i.e. length of public queues of the Cloud), we can apply (13) by replacing $f_k^{EA,pub}$ with f^{Cloud}.

Regarding the end of the computation for the tasks stored in the public queues, we define $\psi_{n,k}^{pub}(t) \in \mathcal{T}$ as the completion (task computation or timeout) time slot of the task $u_{n,k}^{pub}(t)$ (i.e. the task has been either computed or thrown by node k). Given the load uncertainty over time and that the $\psi_{n,k}^{pub}(t)$ cannot be known by neither the EA n nor the node k before the task has been actually addressed (processed or thrown), we can indirectly define the value of $\psi_{n,k}^{pub}(t)$ based on the time slot of starting the computation (of task $u_{n,k}^{pub}$) $\tilde{\psi}_{n,k}^{pub}(t)$. Thus, the relationship between $\psi_{n,k}^{pub}(t)$ and $\tilde{\psi}_{n,k}^{pub}(t)$ can be expressed as:

$$\tilde{\psi}_{n,k}^{pub}(t) = \max\Big\{t, \max_{t'<t} \psi_{n,k}^{pub}(t') + 1\Big\}, \quad (14)$$

where (14) reflects that the task $u_{n,k}^{pub}(t)$ either starts being processed immediately after its arrival to node k (i.e. at time slot t) or waits the most-demanding of the previously stored tasks (i.e. $\max_{t'<t} \psi_{n,k}^{pub}(t')$, $\forall t' = \{0, 1, \ldots, t-1\}$) to be finished and starts being processed at the next time slot. In addition, the size of task $u_{n,k}^{pub}(t)$ can be bounded as:

[9]It can be easily assumed that the public CPU of each EA has different processing capabilities or even that each EA has multiple public CPUs with different or equal processing capacities.

[10]Prioritized policies can be also adopted, without loss of generality. For example, it is possible to consider higher priority for the tasks of specific EAs than others by assigning different weights per EA. In this case, the allocation of the public queue processing capacity is achieved based on these weights.

$$\sum_{i=\tilde{\psi}^{pub}_{n,k}(t)}^{\psi^{pub}_{n,k}(t)-1} \frac{\Delta \cdot f_k^{EA,pub}}{\rho_n(t) \cdot A_k(i)} < \eta^{pub}_{n,k}(t) \leq \sum_{i=\tilde{\psi}^{pub}_{n,k}(t)}^{\psi^{pub}_{n,k}(t)} \frac{\Delta \cdot f_k^{EA,pub}}{\rho_n(t) \cdot A_k(i)} \tag{15}$$

where the right part of the two-sided inequality (15) implies that the size of the task $u^{pub}_{n,k}(t)$ does not exceed the number of bits processed by node k within the computation time interval of the same task, whereas the left part implicitly forces the number of bits processed within $\tilde{\psi}^{pub}_{n,k}(t)$ and $\psi^{pub}_{n,k}(t)-1$ to be lower than the size of task $u^{pub}_{n,k}(t)$. Overall, inequality (15) tells that there are still bits of task $u^{pub}_{n,k}(t)$ to be processed within $\psi^{pub}_{n,k}(t)-1$ and $\psi^{pub}_{n,k}(t)$. Note that (15) stands for EA-to-EA offloading (i.e. $n \in \mathcal{E}$ and $k \in \mathcal{E}-\{n\}$) and, at a given slot $i \in \mathcal{T}$, it should be $n \in \mathcal{A}_k(i)$. In the case of EA-to-Cloud offloading, inequality (15) can be used by replacing $f_k^{EA,pub}$ with f^{Cloud}.

G. HISTORICAL LOAD LEVELS

ECs keep track of the historical load timeseries of each EA and the Cloud. The load of a node $k \in \mathcal{N}$ at time slot t is expressed as the number of active queues $A_k(t)$. To this end, ECs maintain a matrix of the load levels per node, denoted as $\mathbf{L}(t)$[11]. This matrix has dimensions $W \times (N+1)$, where W is the lookback window and $N+1$ is the number of the computing nodes (i.e. all EAs plus the Cloud). As such, $\mathbf{L}(t)$ contains in each column $j \in \mathcal{N}$ the load of node j from the time slot $t-W$ to time slot $t-1$. Each element (i,j) of matrix $\mathbf{L}(t)$ reflects the number of active queues of node j at time slot $t-W+i-1$, as expressed in the following:

$$\begin{aligned} L_{i,j}(t) &= A_j(t-W+i-1), \\ \forall i &\in \{1,2,\ldots,W\}, \forall j \in \mathcal{N}, \end{aligned} \tag{16}$$

where $\max\{L_{i,j}(t)\} = N-1$, for $i \in \{1,2,\ldots,W\}$ and $j \in \mathcal{E}$ (max number of active queues in the EAs), whereas $\max\{L_{i,j}(t)\} = N$, for $i \in \{1,2,\ldots,W\}$ and $j = N+1$ (max number of active queues in the Cloud). Based on the above, the matrix $\mathbf{L}(t)$ has the following structure:

$$\mathbf{L}(t) = \begin{bmatrix} A_1(t-W) & \ldots & A_{N+1}(t-W) \\ A_1(t-W+1) & \ldots & A_{N+1}(t-W+1) \\ \vdots & \ddots & \vdots \\ A_1(t-1) & \ldots & A_{N+1}(t-1) \end{bmatrix} \tag{17}$$

Given that there are M distributed ECs in the Edge Layer of the CEC system, each EA is capable of receiving a single or multiple columns of matrix $\mathbf{L}(t)$ upon request to the associated EC (e.g. the nearest EC or the EC that controls the respective cluster of EAs). Specifically, at each time slot t, each EA requires the W previous load values of the other nodes for purposes of inferring the local HOODIE DRL model, as shown in Section A.

Overall, the time spent on profiling computation and communication performance can be assumed minimal due to two reasons: (i) Profiling is performed instantly and in parallel during other tasks' execution and communication, meaning that the system collects and computes data without introducing significant delays; (ii) Profiling data are computed and stored at the end of each time slot, allowing the RL agent to use the data in the next decision-making phase without interrupting the current operations.

[11]We assume that the number of active queues is shared by each EA and the Cloud at the end of each time slot. The bandwidth overhead for sharing the load levels is negligible, since, in the worst-case, the Cloud broadcasts $\lceil \log_2 N \rceil$ bits to the ECs, whereas each EA transmits $\lceil \log_2 (N-1) \rceil$ bits to the associated EC.

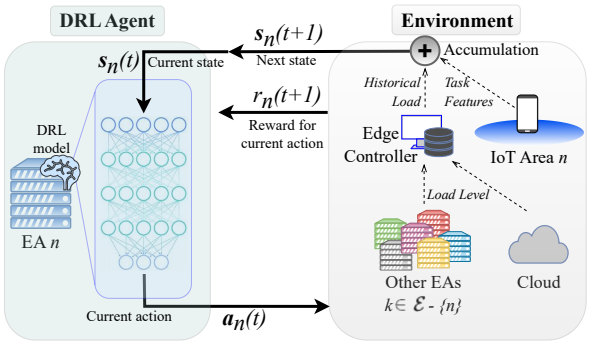

FIGURE 3. The DRL interaction loop between a given EA n and its environment for optimal handling the task offloading in the CEC. The environment of EA n involves its IoT area tasks and all the CEC computing nodes, including the other EAs and the Cloud.

III. DECENTRALIZED VERTICAL AND HORIZONTAL TASK OFFLOADING IN CEC

In this section, the main principles and the mathematical modelling underlying the HOODIE algorithm are outlined. Considering a multi-agent DRL scheme, where each EA acts as a DRL agent, a decentralized hybrid (i.e. vertical and horizontal) task offloading across CEC is proposed. In brief, at a given time slot, each EA observes the CEC environment state (local task characteristics, load forecasting of the other nodes) and makes a decision (local computation, vertical offloading or horizontal offloading). The way that EA-specific HOODIE models are trained aims to ensure that long-term offloading decisions jointly minimizes a two-fold cost function: (i) the task execution latency, and (iii) the task loss rate due to timeout disrespect.

A. DRL MODELLING ELEMENTS

It is assumed that each EA $n \in \mathcal{E}$ runs a DRL model for purposes of making efficient decisions on task computation or offloading. In general, multi-agent decentralized DRL schemes are characterized by: (i) the state that is observable by each DRL agent at each time slot, (ii) the available actions that each DRL agent is able to take, and (iii) the reward or punishment received by each DRL agent as a result of taking

TABLE 3. DRL terminology

Symbol	Meaning [Unit]	Symbol	Meaning [Unit]
$\mathcal{S}$	Set of state spaces of all DRL agents	$\mathcal{S}_n$	State space of DRL agent n
$\mathbf{s}_n(t)$	State vector of DRL agent n at time slot t	$\mathbf{a}_n(t)$	Action vector of DRL agent n at time slot t
$r_n(t)$	Reward received by DRL agent n at time slot t	$\Phi_n(t)$	Cost for processing task $u_n(t)$
C	Penalty for task loss	$\Phi_n^{priv}(t)$	Cost for processing task $u_n(t)$ locally
ϵ	Value of ϵ-greedy policy	$\Phi_n^{pub}(t)$	Cost for processing task $u_n(t)$ externally
π_n	Policy of EA n for task offloading	π_n^*	Optimal policy of EA n for task offloading
γ	Discount factor	θ_n	Q-model parameters of DRL agent n
$Q_n(\cdot)$	Q-value of a state-action pair of EA n	$V_n(\cdot)$	State-value of a given state $\mathbf{s}_n(t)$
$\hat{Q}_n(\cdot)$	$\hat{Q}$-value of a state-action pair of EA n	N_{copy}	Update frequency of $\hat{Q}$-model
$A_n(\cdot)$	Advantage-action value of action $\mathbf{a}$ under $\mathbf{s}_n(t)$	MSE$(\cdot)$	Loss function of HOODIE model
α_{lr}	Learning rate	N_E	Number of training episodes
N_R	Size of replay memory	N_B	Batch size
$\hat{\theta}_n$	Target Q-model parameters of DRL agent n	$\mathcal{D}_n(t)$	Set of previous tasks of EA n completed exactly at time slot t

a given action from a particular state. As in Markov Decision Processes, a DRL agent transits from an environment state to another according to its actions, while also receives a reward in each of the state transitions. Since each DRL agent has local observability of its own state in multi-agent DRL schemes, rewards are the only way for a DRL agent to sense whether a local (and, usually, optimistic) decision is beneficial for the global system. Hence, the main goal is to find, for each DRL agent, the (sub)optimal mapping from local states to local actions, so as to ensure system-wide cost minimization.

For the ease of exposure, Table 3 tabulates all the acronyms related the DRL terminology and symbols. Note that, as shown in the general DRL interaction loop of Fig. 3, each DRL agent treats the local task features and the rest of the DRL agents (and the Cloud) as part of its observable external environment, meaning that the local states and actions concerning the EAs are independent between each other, whereas all actions affect the common CEC system. In other words, the load of EA 5 is part of the environment of EA 2, and vice versa. The HOODIE model used by each EA n is depicted in Fig. 4. Below, we present the local state, the local action and the system-level cost function for each DRL agent.

1) State space

We let $\mathcal{S} = \{\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_N\}$ denote the set of EA-specific sets, which means that the element $n \in \mathcal{E}$ of set $\mathcal{S}$ is the finite and discrete state space of EA n, which is defined as $\mathcal{S}_n = \{\mathbf{s}_n(1), \mathbf{s}_n(2), \ldots, \mathbf{s}_n(T)\}$. Specifically, at the start of time slot $t \in \mathcal{T}$, each EA model $n \in \mathcal{E}$ observes the state vector $\mathbf{s}_n(t)$ which is given by:

$$\mathbf{s}_n(t) = \Big[\eta_n(t), w_n^{priv}(t), w_n^{off}(t), \mathbf{l}_n^{pub}(t-1), \mathbf{L}(t)\Big] \tag{18}$$

where, for each EA $n \in \mathcal{E}$, the vector $\mathbf{l}_n^{pub}(t-1) = \Big(l_{n,k}^{pub}(t-1), k \in \mathcal{N} - \{n\}\Big)$. For instance, we can compute the length of the public queues hosting the tasks offloaded by EA 2 at time slot $t-1$ as $\mathbf{l}_2^{pub}(t-1) = \Big[l_{5,1}^{pub}(t-1), l_{5,3}^{pub}(t-1), l_{5,4}^{pub}(t-1), \ldots, l_{5,N+1}^{pub}(t-1)\Big]$. Thus, the state vector of an EA n is comprised of some local observations at time slot t (i.e. the values of $\eta_n(t)$,$w_n^{priv}(t)$ and $w_n^{off}(t)$), and the local computation[12] of $\mathbf{l}_n^{pub}(t-1)$ at time slot $t-1$ according to (13). Notably, each EA knows the number of bits that it has offloaded to another node at each time slot. Since each EA broadcasts the number of active queues at each time slot, it can also compute the number of bits related to its offloaded tasks and processed by another node. This is achieved by checking (i) whether there are remaining bits of a given task to be processed at the current time slot, and (ii) whether the timeout of the task has passed. For instance, if EA 5 has offloaded a task of size 100 bits to EA 2, it can compute the number of its bits processed at the end of time slot t by locally computing the term $\frac{\Delta \cdot f_2^{EA,pub}}{\rho_5(t) \cdot A_2(t)}$. To this end, EA 5 needs to know only (i) the processing capacity $f_2^{EA,pub}$ of EA 2 (it is known from the beginning of the training process as global variable) and (ii) the number of active queues $A_2(t)$ of EA 2 at time slot t (it is shared by each EA through EC).

Noteworthy, the state vector of each DRL agent at time slot t involves, among others, a forecasting value of the load level of each computing node at the next time slot $t+1$. This is achieved by passing the W previous values of the load timeseries through an LSTM network, as illustrated in Fig. 5.

Regarding the state space dimensionality of vector $\mathbf{s}_n(t)$, the number of all possible states that can be visited by a single DRL agent is equal to $\mathcal{H} \times \mathcal{T}^2 \times \Lambda^N \times \{0, 1, \ldots, N\}^{W \cdot (N+1)}$, where Λ is the set of public queue length values.

[12] To compute $l_{n,k}^{pub}(t-1)$ based on (13), each EA n knows how many bits have been removed (processed or thrown) from the public queue n of all the other nodes k (other EAs and Cloud). Thus, the terms $m_{n,k}^{pub}(t)$ and $\frac{\Delta \cdot f_k^{EA,pub}}{\rho_n(t) \cdot A_k(t)}$ of (13) are known by each EA n.

2) Action space

Once a new task $u_n(t)$ is arrived in EA n at the beginning of the time slot t, the DRL agent observes the state vector $\mathbf{s}_n(t)$ and takes an action $\mathbf{a}_n(t)$ for this task. The action is two-fold and concerns both the DM$^{(1)}$ (whether the task is going to be locally processed or not) and the DM$^{(2)}$ (to which node the task will be offloaded), as defined in Section C. This dual decision is reflected by the action vector $\mathbf{a}_n(t)$, which is selected by EA $n \in \mathcal{E}$ at time slot slot $t \in \mathcal{T}$, as follows:

$$\mathbf{a}_n(t) = \Big[d_n^{(1)}(t), \mathbf{D}_n(t)\Big] \tag{19}$$

It is worth mentioning that every new action $\mathbf{a}_n(t)$ that is taken at time slot t changes the environment state, thus $\mathbf{s}_n(t+1) \neq \mathbf{s}_n(t)$. This is attributed to the fact that each task offloading decision changes the environment state in terms of the length of the private and public queues, the waiting time for processing or offloading and the upcoming load. Also, the number of available actions for a given DRL agent is equal to $\{0,1\}^{N+1}$, denoting that each element of vector $\mathbf{a}_n(t)$ can be either 1 or zero[13].

3) Cost function

By taking an action $\mathbf{a}_n(t)$ from a given state $\mathbf{s}_n(t)$ at time slot t, the DRL agent $n \in \mathcal{E}$ causes a new environment state $\mathbf{s}_n(t+1)$, while also receiving a reward $r_n(t+1)$. The reward $r_n(t+1)$ received at time slot $t+1$ refers to the action $\mathbf{a}_n(t)$ taken at time slot t, and can be either positive or negative. Also, the aim of using rewards, which are returned by the environment to the DRL agent, is to represent the cost function of the problem.

Here we consider the joint minimization of the task execution delay (i.e. the time interval between task arrival and task execution) and the task loss ratio (due to timeout violation). To this end, the reward $r_n(t+1)$ is given by:

$$r_n(t+1) = \begin{cases} \textit{NaN}, & \text{if } x_n(t) = 0 \text{ (no task arrived)} \\ -\Phi_n(t), & \text{if } \psi_n^{priv}(t) < t + \phi_n - 1 \\ & \text{or } \psi_{n,k}^{pub}(t') < t + \phi_n - 1 \\ & \text{(task successfully processed)} \\ -C, & \text{otherwise (task thrown)} \end{cases} \tag{20}$$

where $\Phi_n(t)$ is the cost (combining delay for local or offloaded processing) of task $u_n(t)$ (from task arrival to task execution) and $C > 0$ is a constant penalty received if the task is thrown. The three cases of the rewarding function includes: (i) the reward is omitted (denoted as 'Not-a-Number' (*NaN*)) if no task arrived, (ii) a negative value of $-\Phi_n(t)$ if the task was successfully processed, or (iii) a negative penalty $-C$ if the task was thrown. Note that $\psi_{n,k}^{pub}(t')$ is the time slot when the task $u_n(t)$ was processed upon offloading ($u_n(t)$ arrived in EA n at time slot t and was placed in the public queue of node k at time slot $t' > t$). The cost received upon processing the task $u_n(t)$ reflects the delay term, and is defined as follows:

$$\Phi_n(t) = \begin{cases} \Phi_n^{priv}(t), & \text{if } d_n^{(1)} = 1 \text{ (local processing)} \\ \Phi_n^{pub}(t), & \text{if } d_n^{(1)} = 0 \text{ (offloading)} \end{cases} \tag{21}$$

where $\Phi_n^{priv}(t)$ is the cost for processing the task $u_n(t)$ locally and is given by:

$$\Phi_n^{priv}(t) = \psi_n^{priv}(t) - t + 1 \tag{22}$$

and $\Phi_n^{pub}(t)$ is the cost for processing the task in another node and is computed as:

$$\Phi_n^{pub}(t) = \sum_{k \in \mathcal{N} - \{n\}} \sum_{t'=t}^{T} d_{n,k}^{(2)}(t) \Big(\psi_{n,k}^{pub}(t') - t + 1 \Big) \tag{23}$$

Note that, the term $+1$ in (22) and (23) implies that the processing of task arrived at t starts at the next time slot.

B. PROBLEM FORMULATION

This section presents the optimization problem for optimally handling the offloading decisions in CEC systems. The goal is to enable each distributed DRL agent (located in each EA) making optimal offloading decisions so as to ensure low task delay and low task loss rate. Let π_n denote the policy learned by DRL agent $n \in \mathcal{E}$, which is the mapping from the state space to the action space, i.e. $\mathbf{s}_n(t) \xrightarrow{\pi_n} \mathbf{a}_n(t)$, $\forall \mathbf{s}_n(t) \in \mathcal{S}_n$. The main objective of each DRL agent n is to find the optimal policy π_n^* that maximizes the expected long-term negative reward (which is cumulatively collected during a series of time slots). This is expressed in the following objective function:

$$\begin{aligned} \pi_n^* = \arg\max_{\pi_n} \mathbb{E}\Big\{ \sum_{t \in \mathcal{T}} \gamma^{t-1} \cdot r_n(t) \Big| \pi_n \Big\} \\ \textit{subject to:} \text{ constraints (4),(6)-(7),(9)-(11),} \\ \text{(13)-(15), (20)-(23)} \end{aligned} \tag{24}$$

where function $\mathbb{E}\{\cdot\}$ is the expectation over the random and time-varying task arrivals, task characteristics, and the decisions made by the other DRL agents. Also, $\gamma \in (0,1]$ is the discount factor which scales the future rewards. Evidently, (24) implies that the HOODIE problem is efficiently solved by finding the optimal series of actions which return the maximum accumulative costs. Note that, the ideal policy would converge in zero rewards, since we have defined negative rewards (because delays reflect negative impact on the task completion).

[13]The action vector is comprised of $N+1$ elements, with each element taking values from $\{0,1\}$. In most cases, one element is set to 1 and the rest are set to 0, but there are also chances to have multiple 1's. This is the case when we have equally good actions (e.g. it is equally good to offload a task to either EA 3 or EA 5). In this case, we select randomly one of those options.

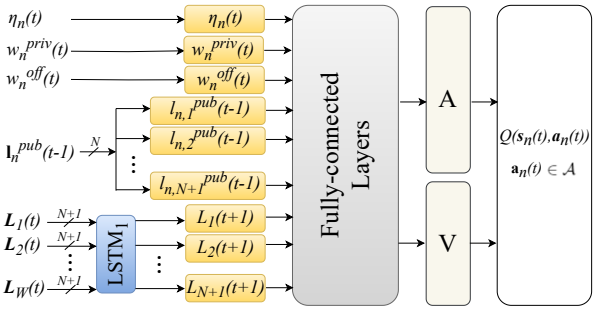

FIGURE 4. The HOODIE neural network structure for DRL agent (or EA) n, which estimates the Q-value of being in state $\mathbf{s}_n(t)$ and taking any of the available actions $\mathbf{a}_n(t)$. Symbols $\mathbf{L}_i(t)$ denotes the i^{th} row of matrix $\mathbf{L}(t)$.

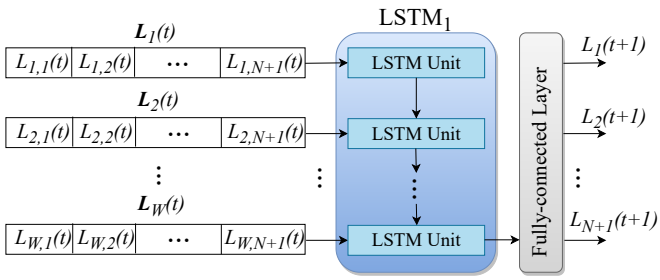

FIGURE 5. LSTM model for predicting the upcoming load of each node at time slot $t+1$ based on W previous values (relative to the current time slot t) of each node-specific load.

IV. HOODIE SOLUTION

To solve the optimization problem defined in (24), this paper proposes the HOODIE algorithm. This algorithm is based on the principle of DRL and, especially the Deep Q-Learning (DQL) [7], [32]. HOODIE enables each EA to use a DQL-based model for making decisions on task offloading in a decentralized manner. As DQL is a model-free approach, HOODIE can be applied for efficient task offloading in the CEC, without requiring analytical and complicated information of the large-scale system state (e.g. there is no need for an agent to know the others' decisions).

According to the DQL, each DRL agent $n \in \mathcal{E}$ holds a deep neural network (the so-called deep Q-network) which estimates the 'quality' (or the Q-value) of each state-action pair. In other words, each DRL agent estimates the Q-function $Q\Big(\mathbf{s}_n(t), \mathbf{a}_n(t)\Big)$. In specific, the $Q(\cdot)$ function reflects the expected long-term reward of being in state $\mathbf{s}_n(t)$ and taking action $\mathbf{a}_n(t)$. Thus, by properly training the deep Q-network, each EA can continuously select the actions with the minimum cost.

A. THE HOODIE MODEL

The structure of the HOODIE neural network model is depicted in Fig. 4. Note that the training of the HOODIE model of DRL agent n is to properly adjust all the network parameters θ_n (i.e. the matrix containing the weights of all connections and the biases of all neurons), such that the Q-values of all possible state-action pairs are accurately predicted. Below, the HOODIE layers are outlined.

1) Input layer

The input layer consists of the state vector, including the task size, the waiting periods in the private and the offloading queues, the length of the public queues of the other nodes, and the predicted load levels of all nodes. This means that, to produce beneficial task offloading decisions, the HOODIE model of a DRL agent takes into account predictions about the load level of the other nodes. Intuitively, being aware of the future load of the other nodes can guide the model to select delay-aware decisions (i.e. offloading towards over-loaded nodes will be avoided). Thus, the HOODIE model receives a W-element history (provided by matrix $\mathbf{L}(t)$) for each node and passes it through an LSTM model to obtain the predicted load in each node at the next time slot.

2) Fully-connected hidden layers

The $Q(\cdot)$ function is estimated via the usage of multiple fully-connected layers which are stacked consecutively. The internal structure of the each fully-connected layer contains a pre-defined number of hidden neurons, each one having a Rectified Linear Unit (ReLU) as activation function. Each neuron of a given layer is connected to all neurons of the next layer.

3) Dueling DQL layer

The output of the fully-connected layers is inserted to an Advantage & Value (A&V) layer, which implements the technique of dueling DQL [29]. Dueling DQL has been widely used as an efficient DRL technique with improved learning ability in mapping state-action pairs to Q-values. The idea underlying dueling DQL method is to decompose the learning of the $Q(\cdot)$ function in two different components: (i) the state-value function $V_n\Big(\mathbf{s}_n(t)\Big|\theta_n\Big)$ (quantifies the contribution of the state in the Q-value), and (ii) the action-advantage function $A_n\Big(\mathbf{s}_n(t), \mathbf{a}\Big|\theta_n\Big)$ (quantifies the contribution of the action in the Q-value, given a state). In this sense, dueling DQL can better approximate the Q-values, since the long-term rewards are assessed separately for states and actions [29]. Both functions $A(\cdot)$ and $V(\cdot)$ are estimated by two fully-connected layer-based subnetworks, mentioned in Fig. 4) as A and V, respectively.

4) Output layer

By passing a state-action pair through the HOODIE model of EA n with parameter θ_n, the output layer provides the Q-value of being at being at state $\mathbf{s}_n(t)$ and taking action $\mathbf{a} \in \{0,1\}^{N+1}$, which is given by:

$$Q_n\Big(\mathbf{s}_n(t),\mathbf{a}\Big|\theta_n\Big) = V_n\Big(\mathbf{s}_n(t)\Big|\theta_n\Big) + \Big[A_n\Big(\mathbf{s}_n(t),\mathbf{a}\Big|\theta_n\Big) - \frac{1}{2^{N+1}}\sum_{\mathbf{a}'\in\{0,1\}^{N+1}} A_n\Big(\mathbf{s}_n(t),\mathbf{a}'\Big|\theta_n\Big)\Big] \tag{25}$$

where $\{0,1\}^{N+1}$ represents the action space of each DRL agent, with size 2^{N+1} (all possible actions). As imposed by (25), the Q-value is computed as the sum of the state-value and the action-advantage value, with the latter being relative to the mean action-advantage value across all possible actions.

B. THE HOODIE ALGORITHM

This section describes the training and the inference phase of the proposed algorithm. The HOODIE algorithm incorporates N distributed models (one model per EA), each one acting as a task offloading suggester. We consider each EA hosting both the training and the inference of the local HOODIE model.

1) Training phase

Algorithm 1 presents the pseudocode for training a single-agent model based on the *experience replay* technique [32]. Specifically, each transition from a given state to another is recorded in the so-called experience tuple $\Big(\mathbf{s}_n(t),\mathbf{a}_n(t),r_n(t+1),\mathbf{s}_n(t+1)\Big)$ and is stored in the experience replay memory, which has a size of N_R rows. In each of the N_E training episodes, the DRL agent runs for T time slots. To estimate the Q-values, each DRL agent n holds two deep neural networks: (i) the Q_n*-model* with parameters θ_n which is used for selecting actions, and (ii) Target $\hat{Q}_n$*-model* with parameters $\hat{\theta}_n$ which is used for estimating the expected long-term rewards. The idea behind using the Target $\hat{Q}_n$*-model* is to provide target or groundtruth values for calculating the error of the Q-values predicted by the Q_n*-model*. Both neural networks have exactly the same architecture, whereas the parameters $\hat{\theta}_n$ are uploaded more rarely than θ_n. Thus, every N_{copy} training episodes, θ_n parameters are copied to the $\hat{\theta}_n$ parameters.

The input of the Algorithm 1 is the set of $T,\alpha_{lr},\gamma,N_E,N_R,N_B,N_{copy}$ learning hyperparameters (that influence the training convergence), whereas the output is the learned policy π_n^* defined in (24). This way each agent $n\in\mathcal{E}$ can take placement decisions on tasks arriving from its IoT area according to the locally learned Q-function. The algorithmic steps can unfold as follows. In each training episode, the HOODIE agent initializes (*Line* 7) the state as $\mathbf{s}_n(1)=[\eta_n(1),w_n^{priv}(1),w_n^{off}(1),\mathbf{l}_n^{pub}(0),\mathbf{L}(1)]$, where $l_{n,k}^{pub}(0)=0$ and $\mathbf{L}(1)$ is a zero matrix. Then, the agent enters in T iterations (*Lines* $8-37$) for the episode time slots. In each time slot t, a new task $u_n(t)$ is arrived from IoT are n with probability $\mathcal{P}$ (*Lines* $9-13$). In case

Algorithm 1 HOODIE training in EA $n\in\mathcal{E}$

1: **Learning inputs:** T, α_{lr}, γ, N_E, N_R, N_B, N_{copy}
2: Initialize Replay Memory with N_R rows
3: Initialize Q_n-model with random θ_n
4: Initialize target $\hat{Q}_n$-model with random $\hat{\theta}_n$
5: Initialize $CurrentEpisode=0$ and $\epsilon=1$
6: **for** each episode $i=1,2,\ldots,N_E$ **do**
7: Initialize $\mathbf{s}_n(1)$
8: **for** each time slot $t=1,2,\ldots,T$ **do**
9: Set $x_n(t)=0$ {Assume no task arrived}
10: Pick a random number $rand\in[0,1]$
11: **if** $rand<\mathcal{P}$ **then**
12: Set $x_n(t)=1$ {New task arrived}
13: **end if**
14: **if** $x_n(t)=1$ **then**
15: Set a new task ID $u_n(t)$
16: Select action $\mathbf{a}_n(t)$ for $u_n(t)$ based on (26)
17: **end if**
18: Observe next state $\mathbf{s}_n(t+1)$
19: **for** each completed task $u_n(t')\in\mathcal{D}_n(t)$ (27) **do**
20: Collect reward $r_n(t')$ of task $u_n(t')$ based on (20)
21: Store experience $\Big(\mathbf{s}_n(t'),\mathbf{a}_n(t'),r_n(t'),\mathbf{s}_n(t'+1)\Big)$
22: **end for**
23: Sample a random batch B of N_B experiences
24: **for** each experience row $i\in B$ **do**
25: Assume row format as $\Big(s_{n,i},a_{n,i},r_{n,i},s'_{n,i}\Big)$
26: **if** $t=T$ {Terminal time slot} **then**
27: Set target $y_{n,i}=r_{n,i}$
28: **else**
29: {Double Q-learning}
30: Set target $y_{n,i}=r_{n,i}+\gamma\cdot\hat{Q}\Big(s'_{n,i},\arg\max_{a'}Q\Big(s'_{n,i},a'\mid\theta_n\Big)\mid\hat{\theta}\Big)$
31: **end if**
32: Set predicted $z_{n,i}=Q_n\Big(s_{n,i},a_{n,i}|\theta_n\Big)$
33: **end for**
34: Set target values $\mathbf{Y}_n^{Target}(t)=\{y_{n,i}\},\forall i\in B$
35: Set predicted values $\mathbf{Y}_n^{Pred}(t)=\{z_{n,i}\},\forall i\in B$
36: Update parameters θ_n to minimize loss function: $\mathrm{MSE}\Big(\mathbf{Y}_n^{Target}(t),\mathbf{Y}_n^{Pred}(t)\Big)$ (see (28))
37: **end for**
38: Set $CurrentEpisode=CurrentEpisode+1$
39: **if** mod($CurrentEpisode$,N_{copy})$=0$ **then**
40: Set $\hat{\theta}_n=\theta_n$
41: **end if**
42: **if** $CurrentEpisode\leq N_E/2$ **then**
43: Set $\epsilon=1-2(CurrentEpisode-1)/N_E$
44: **else**
45: Set $\epsilon=0$
46: **end if**
47: **end for**
48: **Output:** Optimal policy π_n^* for action-value function Q

of a new task arrival, the HOODIE agent selects either an exploratory or exploitative action $\mathbf{a}_n(t)$ following the ϵ-greedy policy as (*Line* 16):

$$\mathbf{a}_n(t) = \begin{cases} \text{Random value of (19)}, & \text{if } rand < \epsilon \\ \arg\min_a Q_n\Big(s_n(t), a|\theta_n\Big), & \text{if } rand \geq \epsilon \end{cases} \quad (26)$$

where $rand$ is a random number in $[0, 1]$ and ϵ is a value updated in each episode according to the *Lines* $42 - 46$. Based on the above, with probability ϵ, the agent chooses a random action (i.e. exploration), whereas with probability $1 - \epsilon$, choose the lowest Q-value action by inferring the Q_n-*model* (i.e. exploitation). The ϵ starts at 1 and decreases linearly to 0 during the first $N/2$ episodes, whereas it is set at 0 for the remaining $N/2$ episodes (i.e. only exploitation). Since other agents operate under the same CEC environment, each HOODIE agent n observes the new state (*Line* 18), regardless of whether a new local task was arrived or not.

Since the processing and/or the offloading of task may take many time slots to be completed, the reward of previous tasks arrived at $t' < t$ may be collected at the current time slot t. To this end, the HOODIE agent n collects (at time slot t) the set $\mathcal{D}_n(t)$ of previous tasks that completed exactly at time slot t, which is given by:

$$\mathcal{D}_n(t) = \Big\{u_n(t')\Big|t' < t \text{ and } \eta_n(t') > 0 \text{ and } \psi_n^{priv} = t \text{ or } \psi_{n,k}^{pub} = t\Big\} \quad (27)$$

where $\mathcal{D}_n(t)$ contains the ID of tasks that are completed exactly at time slot t. This implies that $\mathcal{D}_n(t)$ aggregates the tasks with positive size ($\eta_n(t') > 0$) arrived in EA n in a past time slot ($t' < t$) and had completion time slot equal to t, either after local ($\psi_n^{priv} = t$) or external computation at node k ($\psi_{n,k}^{pub} = t$). Thus, in *Lines* $19 - 22$, the agent n collects the rewards for all previous tasks in set $\mathcal{D}_n(t)$ and stores their corresponding experience tuples into the replay memory. In *Lines* $23 - 33$, the agent prepares a batch B of samples to which the Q_n-*model* will be updated on. To do so, it samples randomly N_B rows from the replay memory (*Line* 23), and, for each sample $i \in B$, calculates (i) the target values $\{y_{n,i}\}$ (*Lines* $26 - 31$) according to the Double DQL formula [33] and (ii) the predicted values $\{z_{n,i}\}$ (*Line* 32) by inferring the Q_n-*model*. In *Lines* $34 - 36$, parameters θ_n are adjusted so as to minimize the loss function:

$$\text{MSE}\Big(\mathbf{Y}_n^{Target}(t), \mathbf{Y}_n^{Pred}(t)\Big) = \frac{1}{N_B}\sum_{i=1}^{N_B}\Big(z_{n,i} - y_{n,i}\Big)^2 \quad (28)$$

where MSE$(\cdot)$ reflects the mean (across batch samples) squared error between the vector of predicted $\mathbf{Y}_n^{Pred}(t)$ and target $\mathbf{Y}_n^{Target}(t)$ values. Finally, *Lines* $39 - 41$ ensure that the parameters θ_n are copied to $\hat{\theta}_n$ every N_{copy} episodes.

2) Inference phase

The inference phase of HOODIE scheme is performed by deploying distributed HOODIE models (only the Q-*models*) at each one of the N EA sites. Upon receiving a new task, each HOODIE model is inferred by a state vector for purposes of properly selecting the best computing destination. In this sense, the inference algorithm is a short version of Algorithm 1 by (i) avoiding *Lines* $23 - 36$ in each time slot, (ii) neglecting *Lines* $38 - 46$ in each episode and (iii) setting constantly $\epsilon = 0$ to select only exploitative actions in *Line* 16.

C. COMPLEXITY CONSIDERATIONS

Time complexity can quantified from Algorithm 1 as the overall and worst-case number of iterations performed by each agent, as a function of the learning hyper-parameters using the big-$\mathcal{O}$ notation. Since the training contains N_E episodes and T time slots per episode, the training complexity has at least $N_E \cdot T$ iterations. Also, in the worst-case, there are (i) N_{prev} (with N_{prev} denoting the length of $\mathcal{D}_n(t)$) iterations per time slot (in *Lines* $19 - 22$) that loop over the previous tasks completed at time t, as well as (ii) N_B iterations (*Lines* $24 - 33$) that loop over batch samples. The most time-consuming loop between (i) and (ii) is (ii), since it contains updates of the target values upon inference of Q- and $\hat{Q}$-models; thus the training complexity has at least $N_E \cdot T \cdot N_B$ iterations[14]. In *Line* 36, it is implied that at each time slot, the parameters θ_n of the Q-model are updated through feed-forward and back-propagation steps, according to the Gradient Descent (GD) method. Let N_{FF} and N_{BP} denote the number of arithmetic operations (multiplications, weighted summations, differentiation) required for the feed-forward and back-propagation steps, respectively, of a single experience tuple[15]. Integrating the above, the computational complexity for training the HOODIE scheme is $\mathcal{O}\Big(N_E \cdot T \cdot N_B \cdot (N_{FF} + N_{BP})\Big)$.

Note that, since the training of the multi-agent HOODIE scheme is performed once (or rarely in time for soft retraining), the above-defined complexity corresponds exclusively to the HOODIE training. When HOODIE is used for inference purposes, the associated complexity reflects only the feed-forward passes of the new inference samples (i.e. only multiplications and weighted summations to produce the Q-values of the input vector). As such, the inference complexity for a single inference sample is $\mathcal{O}\Big(N_{FF}\Big)$.

[14]More strictly, we could write $N_E \cdot T \cdot (N_B + N_{prev})$ iterations; however the term N_prev is neglected for simplicity of presentation, since it reflects the count of simple arithmetic operations and, hence, $\mathcal{O}(N_{prev}) << \mathcal{O}(N_B)$.

[15]The time cost for performing feed-forward and back-propagation steps is positively correlated to the input size of the experience tuples, the size of the output (i.e. output neurons) and the DQN density (i.e. number of hidden layers and neurons).

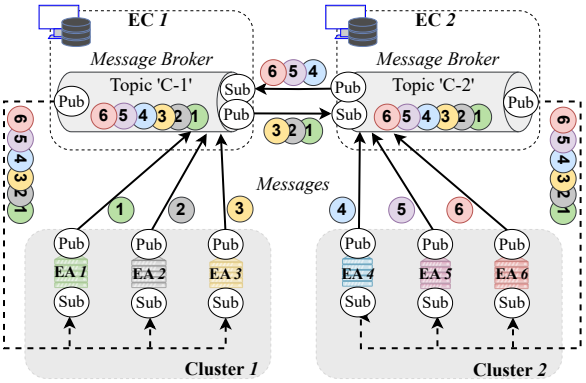

FIGURE 6. The Pub-Sub communication protocol for data sharing within and across clusters of EAs via the usage of ECs as message brokers.

D. CONVERGENCE CONSIDERATIONS

The convergence assurance of reinforcement learning algorithms can be only proven in the simple tabular RL (TRL), where the Q-function is computed by using a table of state-action pairs [34]. This Q-table has a number of rows equal to the number of all possible states (that the agent receives as input) and a number of columns equal to the number of all available actions. In this case, for each state-action pair, the Q-function is calculated using the Bellman's equation [34]. As a result, TRL can be efficiently used only in simple small-scale problems (in which the agent considers a small state-action space) and cannot be considered in practical and large-scale problems (due to time and memory inefficiency for storing the Q-table).

In the case of HOODIE algorithm, we use approximation of the Q-function through the use of deep neural networks (Q-model and target $\hat{Q}$-model), given that the state-action space size is $\mathcal{H}\times\mathcal{T}^2\times\Lambda^N\times\{0,1,\ldots,N\}^{W\cdot(N+1)}$ (all available states) and $\{0,1\}^{N+1}$ (all available actions), respectively (see Table 2 for the symbols meaning). The formalistic convergence of such high-dimensional DQN models can be only proven under mild assumptions [35] and, consequently, remains an open theoretical problem [36], in which DQN convergence cannot be assured by closed formulas (due to stochasticity), rather than using exhaustive simulations. Thus, the convergence of HOODIE scheme is empirically evaluated through an extensive simulations-driven approach, by applying different values of all learning inputs (see *Line* 1) of Algorithm 1, as presented in Sections A and B.

E. INTER-AGENT COMMUNICATION PROTOCOL AND RECOVERY MECHANISM

HOODIE considers inter-agent data sharing at the end of each time slot, since the state vector metrics (or messages) $\mathbf{l}_n^{pub}(t-1)$ and $\mathbf{L}(t)$ need to be shared across EAs in order to be available at the next time slot. Although many different communication protocols can be established (e.g. request/response model, peer-to-peer, Gossip protocol), here we propose the publish-subscribe (Pub-Sub) protocol [37], as shown in Fig. 6. According to this protocol, each cluster of EAs is managed by an EC, which acts as a broker for message exchanges [38]. EAs subscribe to a dedicated topic (e.g., 'C-1' for EC 1, 'C-2' for EC 2, and so on) and publish updates (in the form of messages) on load history and public queue length at the end of each time slot. Similarly, ECs are also subscribers to their own topic to receive updates from EAs in their cluster, whereas, to achieve also inter-ECs communication, ECs can communicate with one another by subscribing to each other's topics for cross-cluster data sharing. ECs disseminate the aggregated messages to all subscribing agents within the cluster, enabling all other EAs in the cluster to receive this shared information in the next time slot. This decentralized approach enables efficient and scalable communication across agents.

However, communication delays due to network congestion or latency can prevent some EAs from receiving up-to-date information in time for the next decision-making phase. To address the potential impact of communication delays, HOODIE can integrate a recovery mechanism that leverages the temporal learning capabilities of LSTM models. When an EA fails to receive updated information from its peers due to a delay, the system occasionally exploits the previous time-slot's output of the LSTM model as an input for the current decision-making process. Similarly, if the metric $\mathbf{l}_n^{pub}(t-1)$ is delayed at time slot t, the agents can occasionally use $\mathbf{l}_n^{pub}(t-2)$ to avoid network blocking. This allows the delayed EA to continue making informed decisions based on historical data, without waiting for the delayed information to arrive. Once the missing message is eventually reached, ECs can integrate it in the load history for providing valid samples in the future slots.

The combined use of Pub-Sub protocol with the proposed recovery mechanism ensures that the system remains resilient to delays, given that it provides (i) non-blocking operation (i.e. the use of LSTM outputs from previous time slots allows EAs to make decisions without waiting for updated information), (ii) eventual consistency (i.e. although agents may operate on historical data during periods of delayed communication, the system guarantees eventual consistency as soon as the missing information is received), (iii) indirect synchronization (i.e. the recovery mechanism reduces the need for strict synchronization ensuring that HOODIE agents will have inference data for the next time slot within the time frame of the current time slot. Agents can continue making decisions autonomously based on local observations and historical data, ensuring minimal disruption to task processing.

Note that, since the message shared by each EA is low-sized (i.e. contains the number of active queues) and the connections between EAs/ECs are considered wired with high bandwidth (e.g. fiber-optics or high-speed Ethernet), we can safely assume that the bandwidth overhead and the propagation delay associated with message publishing,

exchanging, and receiving is negligible for practical purposes (i.e. the overall messaging delay is highly reduced compared to the considered waiting, offloading and task processing delays).

V. EXPERIMENTAL RESULTS

In this section, we conduct a numerical and simulations-driven evaluation of the HOODIE scheme performance within an CEC system. We begin by examining the optimal training convergence against the critical learning parameters of the distributed HOODIE agents, and then we evaluated the impact of several DRL environment parameters to reveal additional behavior and scalability insights of the proposed scheme. Finally, we proceed with a comparative analysis of HOODIE algorithm against existing baseline methods.

All DRL and LSTM models presented in the following subsections were implemented in Python 3.0, using PyTorch library (version 1.9.0), and the CUDA (version 10.2). The derived models were trained on a PC with an AMD Ryzen 7 1800X Eight-Core Processor CPU at 3.60 GHz and 32 GB of RAM.

A. LEARNING PARAMETERS AND CONVERGENCE CURVES

To efficiently implement the HOODIE scheme, various system parameters were carefully configured, with particular emphasis on fine-tuning critical hyperparameters essential for optimizing the multi-agent DRL framework. In the training phase of HOODIE, we set the system and learning parameters as tabulated in Table 4. We adopt an application-agnostic approach, where the computational tasks are characterized only by their size ($\eta_n(t)$) and processing density ($\rho_n(t)$) rather than being tied to specific application types. As such, these parameters represent a wide range of real-world applications that HOODIE can handle, from lightweight tasks (e.g. updating or deleting operations to databases, text editing or processing) to more computationally demanding tasks (e.g. image or video processing, ML model inference). Noteworthy, the number of time slots per episode was set to $T = 110$, where at the first 100 slots, the HOODIE agents are able to take actions, while the last 10 slots were considered for emptying any non-empty queue, allowing the reward collection for pending tasks. Note also that, if we consider varying values of one or many parameter(s) in the rest of the subsections, the rest of the system settings correspond to Table 4, unless explicitly specified. Also, in this subsection, the number of HOODIE agents is equal to 20, whereas the considered Edge layer topology is shown in Fig. 7, upon plotting the adjacent matrix $\mathbf{G}$ as undirected graph.

Initially, a key focus was on selecting the appropriate learning rate α_{lr}, which was tested across a range of values: $\alpha_{lr} = [10^{-9}, 5 \cdot 10^{-9}, 10^{-8}, 10^{-7}, 5 \cdot 10^{-7}, 7 \cdot 10^{-7}]$ (see Fig. 8a). In general, the choice of learning rate plays a crucial role in balancing the speed of convergence with the stability of the learning process, since it has direct influence on how intensely the DQN weights are adjusted during the back-propagation steps. The specific tuning of α_{lr} was guided by the system parameters detailed in Table 4, under moderate task arrival traffic across all HOODIE agents ($\mathcal{P} = 0.5$). As presented in Fig. 8a, where the learning curve of the HOODIE scheme across different learning rates is depicted, A learning rate of α_{lr} proved to be the most effective, striking an optimal balance between fast learning and maximum accumulated reward. As such, a joint minimization of the task completion delay and task drop rate is guaranteed. The performance metric of HOODIE training

TABLE 4. System and Learning Parameters

Parameter	Symbol	Value
Task Arrival Probability	$\mathcal{P}$	0.5
Horizontal Data Rate	R_H	30 Mbps [23]
Vertical Data Rate	R_V	10 Mbps
Task size	$\eta_n(t)$	$[2, 2.1, \ldots, 5]$ Mbits [39]
Task processing density	$\rho_n(t)$	0.297 gigacycles/Mbit [39]
Number of EAs	N	20
Adjacent matrix	$\mathbf{G}$	See Fig. 7
CPU frequency in private queues	$f_n^{EA,priv}$	5 GHz
CPU frequency in public queues	$f_n^{EA,pub}$	5 GHz
CPU frequency in Cloud	f^{Cloud}	30 GHz
Number of Training Episodes	N_E	5000
Number of Time slots	T	110
Time slot duration	Δ	0.1 sec
Task timeout	ϕ_n	20 time slots (i.e. 2 sec)
Learning rate	α_{lr}	$7 \cdot 10^{-7}$
Discount factor	γ	0.99
Q-network hidden layers	N_L	3×1024 neurons
Optimizer	Opt	Adam
Loss function	MSE	See (28)
Update frequency	N_{copy}	2000 iterations
LSTM lookback window	W	10 steps
LSTM hidden layers	N_L	1×20 LSTM cells
Replay Memory size	N_R	10000 samples
Task Drop Penalty	C	40
Batch size	N_B	64 samples

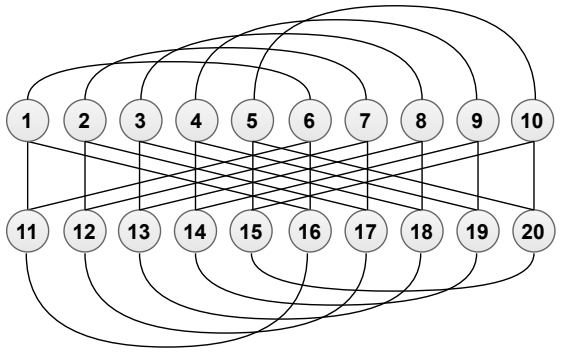

FIGURE 7. Edge layer topology graph of matrix $\mathbf{G}$ with 20 EAs.

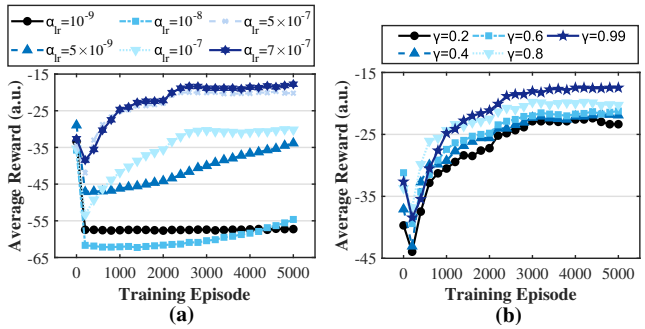

FIGURE 8. Accumulated reward time-course averaged across the distributed HOODIE agents as a function of the training episodes for different (a) Learning rate α_{lr} and (b) Discount factor γ.

is the cumulative reward, as defined in (20) and (24), that was collected in a series of 5000 episodes, and averaged across all the distributed HOODIE agents. Note that, since the task delay is considered a negative metric, the ideal value is zero, which explains why the reward curves are negative and increasing.

In Fig. 8b, the impact of the discount factor γ is also demonstrated, where different reward curves for varying values of $\gamma = [0.2, 0.4, 0.6, 0.8, 0.99]$ are shown. Stabilizing the value of γ may have generally serious impact on the DRL policy convergence, since it directly regulates balance between immediate and future rewards, the stability and speed of learning. Evidently from Fig. 8b, the optimal performance was observed for $\gamma = 0.99$, which means that HOODIE agents exhibit more importance on future rewards, making them more focused on long-term outcomes.

Noteworthy, we observed that agents tended to act selfishly at the beginning of the training by focusing on maximizing their immediate rewards. This may happened because they had not yet fully explored the long-term consequences of their actions. As the training episodes progress, the agents begin to unlearn selfish behavior and shift toward cooperative task distribution. This improved behavior emerges as the agents' Q-function better reflects the future states of the network, discouraging agents to monopolize the resources of other agents. Although the agents are not explicitly programmed to coordinate, cooperative behavior among agents is implicitly fostered.

B. HOODIE BEHAVIOR AND SCALABILITY ANALYSIS

Using the optimal values of learning rate and discount factor, here we quantitatively analyze the HOODIE behavior and scalability (with increasing number of DRL agents) under the variation of several system parameters.

1) The impact of Task Arrival Probability

Initially, to further explore the impact of the task arrival probability ($\mathcal{P}$), which is directly proportional to the task traffic density underlying the IoT areas, Fig. 9a shows the HOODIE performance as a function of increasing system load and edge layer density ($N = [10, 15, 20]$). To assess the HOODIE performance, we used the optimally trained Q-models of each agent (as they resulted from Section A) and we calculated the average reward collected during a series of 200 validation episodes (i.e. all agents performed exploitative actions).

Evidently, as the task arrival probability increases, the average reward decreases across all configurations, indicating the system's increased strain under higher task loads. This is attributed to the complexity introduced when the system load is significantly increased, imposing further suboptimality in the task placement decision. The task arrivals density directly influences the task load on the system, with higher probabilities indicating more tasks that arrive simultaneously, requiring the agents to make more frequent and complex decisions about whether to compute locally or offload. Notably, with fewer agents ($N = 10$), the reward declines more steeply, whereas with more agents ($N = 20$), the decline is less pronounced, demonstrating improved scalability and better handling of the increased workload. However, even with additional agents, the system shows diminishing returns as the task arrival probability approaches 1, suggesting potential saturation points. These results highlight the importance of density-complexity tradeoff, which balances computational resources and task load effectively, thereby maintaining higher performance levels under varying conditions. Note that the high negative rewards do not reflect the average delay of the tasks, but they indicate the high negative penalties received for task drops, which is more frequent as the $\mathcal{P}$ increases.

Under the same simulations, Fig. 9b depicts the distribution of actions taken by the HOODIE agents across different task arrival probabilities, categorized into local computation, horizontal, and vertical offloading. The results show a clear preference for horizontal offloading, especially as the task arrival probability increases, suggesting that agents often opt to share tasks with neighboring EAs rather than process them locally or offload them to the Cloud. However, as the task arrival probability reaches higher values (e.g., 0.9), there is a noticeable increase in vertical offloading actions, indicating a shift towards utilizing cloud resources under heavier task loads. The relative consistency in local computation actions across different probabilities implies that HOODIE maintains a baseline of local processing, but dynamically adjusts its strategy towards offloading as task traffic intensifies. These findings confirm that HOODIE effectively balances action selection based on current system conditions, with a tendency to prioritize horizontal offloading in most scenarios.

2) The impact of Computational Capacity

In Fig. 9c, the average reward is presented as a function of the CPU computation capacity (ranging from $4 - 9$ GHz) dedicated to local tasks, across different numbers of DRL agents. The results demonstrate a clear positive correlation

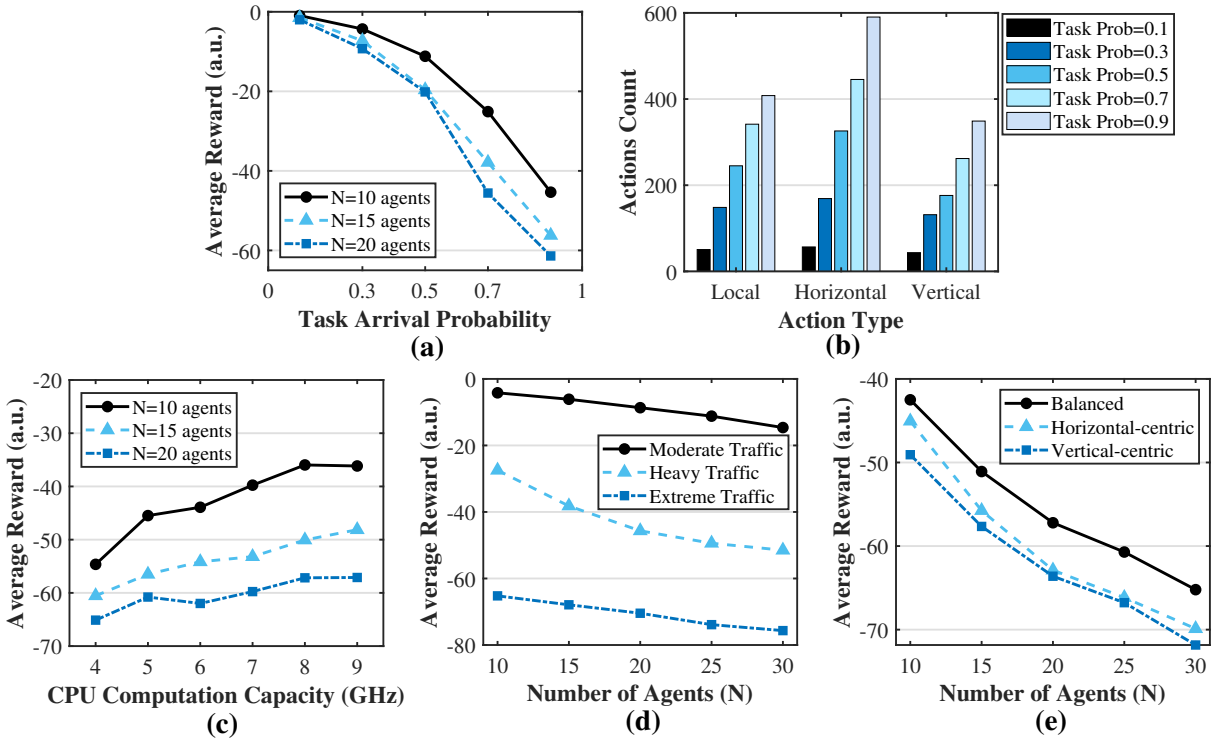

FIGURE 9. **HOODIE behavior insights under varying system parameters. (a) Average reward vs. task arrival probability for different numbers of DRL agents; (b) Action type distribution across task arrival probabilities; (c) Average reward vs. local CPU computation capacity; (d) Average reward vs. number of agents under different traffic scenarios; (e) Average reward vs. number of agents under different offloading data rate scenarios.**

between the CPU capacity and the average reward, indicating that as more computational power is allocated to local tasks, the system's performance improves. This trend is consistent across all agent configurations, although the magnitude of improvement varies. Notably, with fewer agents ($N = 10$), the reward increases more significantly with rising CPU capacity, suggesting that in a more resource-constrained environment, local computation power plays a critical role in maintaining system efficiency. Conversely, in scenarios with more agents ($N = 20$), the improvement is more modest, likely because the additional agents can offload tasks, thereby reducing the dependency on local CPU capacity. These findings underscore the importance of optimizing local CPU resources to enhance the overall reward, especially in systems with a limited number of agents.

3) The impact of Task Traffic Intensity

In Fig. 9d, the average reward is illustrated as a function of the number of DRL agents under three different task traffic scenarios:

- *Moderate Traffic:* This scenario features tasks with sizes ranging from $1-3$ Mbits and a task arrival probability of 50%, representing a quite balanced workload.
- *Heavy Traffic*: In this scenario, task sizes increase to between $2-5$ Mbits with a 70% arrival probability, reflecting a more challenging environment.
- *Extreme Traffic*: This scenario pushes the system further, with task sizes between $3-7$ Mbits and a 90% arrival probability, simulating highly demanding conditions.

As the number of agents increases from 10 to 30, the average reward declines across all traffic scenarios, indicating the growing challenge of managing more complex networks with large state-action spaces. However, the rate of decline varies significantly with traffic intensity. Under moderate traffic, the system maintains relatively high rewards, even as the number of agents increases, suggesting that the system can efficiently manage moderate workloads with additional agents. In contrast, under heavy and extreme traffic conditions, the average reward decreases more sharply, with the extreme traffic scenario showing the steepest decline. This behavior highlights the solution convergence in coping with larger task sizes and higher arrival probabilities, where adding more agents may lead to coordination complexities and reduced efficiency. These results underscore the importance of optimizing the number of agents in relation to the expected traffic load to maintain high performance in the HOODIE framework.

4) The impact of Offloading Data Rate

In Fig. 9e, we examine the average reward as a function of the number of DRL agents under three distinct offloading data rate configurations:

- *Balanced Capacity*: In this scenario, the horizontal offloading data rate is set at $R_H = 10$ Mbps, while the vertical offloading data rate is $R_V = 30$ Mbps, representing a typical distribution of offloading resources.
- *Horizontal-centric Capacity*: This scenario increases the $R_H = 20$ Mbps, balancing it equally with the $R_V = 20$ Mbps.
- *Vertical-centric Capacity*: This scenario skews the offloading resources towards vertical offloading, with a $R_H = 5$ Mbps and $R_V = 40$ Mbps, simulating a setup where the cloud is dominantly preferred for task offloading.

It is revealed that, across all scenarios, the average reward decreases as the number of agents increases, indicating the growing complexity of managing larger networks. The balanced capacity scenario maintains the highest rewards across the range of agents, suggesting that a balanced allocation of offloading resources allows for more efficient task distribution and processing. The horizontal-centric scenario shows a steeper decline in reward, particularly as the number of agents exceeds 20, indicating that when horizontal offloading capacity is prioritized, the system may struggle with increased coordination demands. The vertical-centric scenario exhibits the steepest decline in rewards, especially at higher agent counts, suggesting that an over-reliance on vertical offloading to the cloud can lead to increased latency and reduced system performance. These findings emphasize the importance of carefully configuring offloading capacities to optimize the HOODIE framework's performance as the network scales. Notice that, in Figs. 9d-e, we have further scaled up the system by considering up to 30 HOODIE agents.

It should be noted that, in the previous training and validation curves of Sections A and B, we focused mainly on the cumulative rewards collected by the agents, since they are the most accurate measures of the training performance to assess HOODIE behavior. However, although reward values reflect directly the learning course of the DRL agents, they cannot be directly interpreted because they implicitly carry information about both the task completion delay rewards and the high-valued penalties received for task drops (see (20)). For this reason, in the the comparative outcomes of Section C, we evaluate the HOODIE performance in terms of more interpretable metrics, such as the task completion delay and the task drop ratios, resulted from a series of 200 validation episodes.

C. COMPARATIVE ANALYSIS

To comprehensively investigate the HOODIE algorithm performance, this section focuses on comparing the proposed algorithm against six baseline algorithms, under different variations of the CEC system parameters. We consider the following offloading algorithms:

1) **Random Offloader (RO)**: Each agent acts randomly, choosing an action with probability $1/3$ between local execution, vertical offloading, and horizontal offloading to another EA. If the selected action is horizontal offloading, the destination EA is selected randomly with probablity $1/(N-1)$.
2) **Full Local Computing (FLC)**: Each agent constantly computes all the received tasks locally.
3) **Vertical Offloader (VO)**: Each agent constantly offloads all the received tasks to the Cloud for computation.
4) **Horizontal Offloader (HO)**: Each agent constantly offloads all the received tasks to another EA. Each destination EA has a probability of $1/(N-1)$ to be selected for offloading.
5) **Balanced Cyclic Offloader (BCO)** [40]: Each agent selects an action in a round-robin manner. Taking EA $n = 1$ as an example, the first received task is locally computed, the second is placed at the Cloud, the third is placed at EA 2, the fourth at EA 3, and so on.
6) **Minimum Latency Estimation Offloader (MLEO)** [41]: According to this powerful scheme, for each new task, each agent estimates (in advance) in which queue the task will experience the lowest delay, and then places the task in the lowest-delay queue. Specifically, given a new task $u_1(t)$ arrived at time slot t, agent $n = 1$ computes $N + 1$ estimated delay scores, corresponding to the $N + 1$ different available placement options (i.e. Option #1: the task is placed at private queue; Options #$2-N$: the task is placed at offloading queue and then at public queue of EA $2 - N$; Option #$N + 1$: the task is placed at offloading queue and then at public queue of Cloud). The delay that will be experienced if the task will be placed in the private queue is directly computed using (6). The delay (in sec) that will be experienced if the task will be placed in the public queue n of node $k \in \mathcal{N} - \{n\}$ (other EAs or Cloud) is computed using (9) and then adding the $\frac{\rho_n(t)}{f_k^{CPU}} l_{n,k}^{pub}(t-1)$ using (13). Finally, the agent n selects the action corresponding to the lowest delay estimation.

For all offloading schemes, the performance metrics selected for comparison purposes were the average computation delay and the drop ratio, both calculated over 200 validation episodes. These metrics are the main components of the objective function defined in (24), which reflects the long-term joint optimization of both task delay and drop ratio. To contrast the schemes in terms of the average delay (see Figs. 10a-c), we set a high timeout at 10 sec to allow tasks remain in the system for quite long period. In this way, we are able to compare which scheme offers the minimum

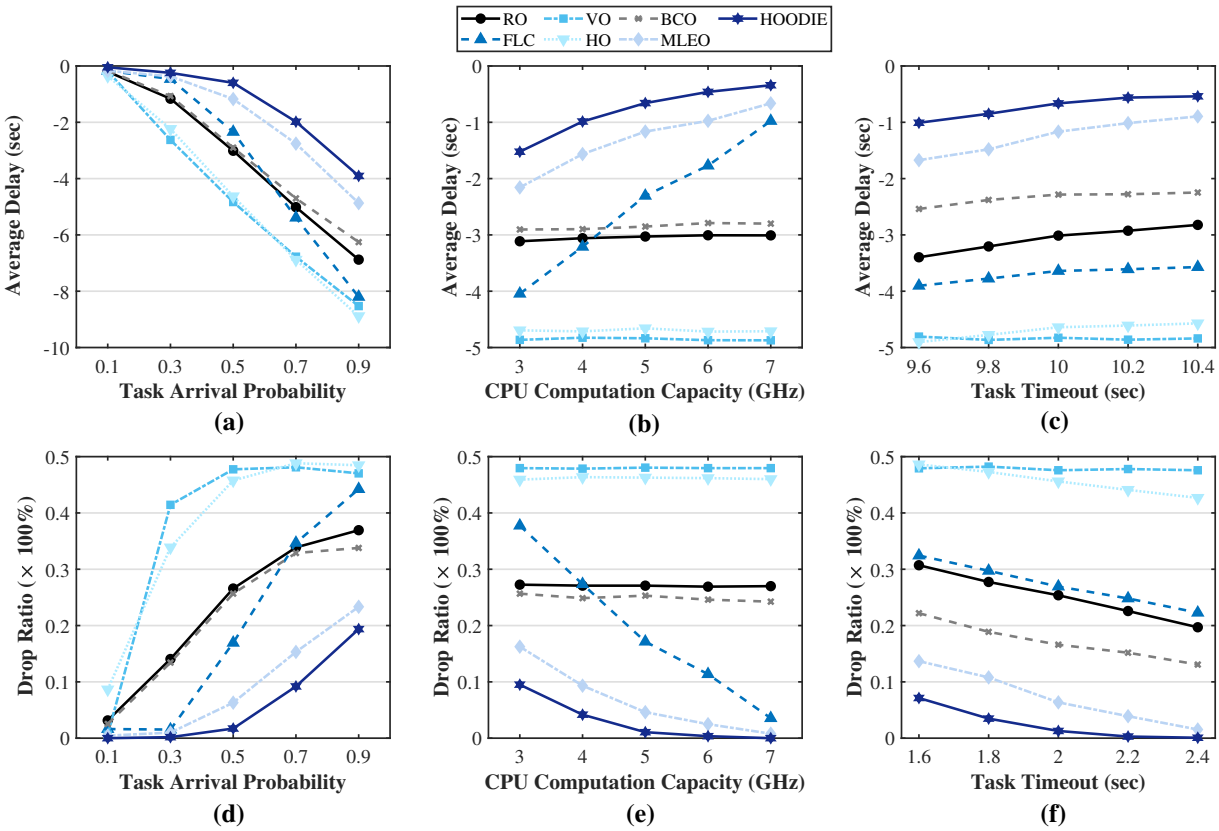

FIGURE 10. **Performance comparison of HOODIE against six baseline offloading schemes under varying conditions. (a) Average delay vs. task arrival probability; (b) Average delay vs. CPU computation capacity; (c) Average delay vs. task timeout; (d) Drop ratio vs. task arrival probability; (e) Drop ratio vs. CPU computation capacity; (f) Drop ratio vs. task timeout. Average delay is negative by convention**

computation delay among tasks that were not dropped, given that the dropped tasks are encountered in the drop ratio. On the contrary, the results in Figs. 10d-f consider a strict timeout at 2 sec, where the schemes are compared in terms of the drop ratio. Drop ratio was computed as the total number of the task dropped, divided by the total number of tasks arrived in all EAs. By convention, delay values are negative, since they reflect negative impact on the task completion.

1) The impact of Task Arrival Probability

Fig. 10a presents the average delay as a function of the task arrival probability across different offloading schemes. As the task arrival probability increases, the average delay generally decreases for all schemes, reflecting the growing challenge of handling a higher load within the system's limited computational resources. Notably, HOODIE consistently outperforms the other schemes, achieving lower average delays across the entire range of task arrival probabilities. The HO and VO schemes, while effective at lower task arrival probabilities, show significantly higher delays as the load increases. The MLEO scheme also demonstrates strong performance, particularly at moderate task loads, but it falls behind HOODIE as the task arrival probability approaches 0.9.

Fig. 10d illustrates the drop ratio as a function of task arrival probability for the same set of offloading schemes. As expected, the drop ratio increases with higher task arrival probabilities, reflecting the system's difficulty in meeting task deadlines under heavier loads. HOODIE demonstrates the lowest drop ratios across the board, indicating its effectiveness in preventing task drops even as the system becomes increasingly burdened. The FLC and HO schemes, in contrast, exhibit the highest drop ratios at higher task arrival probabilities, underscoring their limitations in adapting to varying load conditions. The MLEO scheme shows a competitive drop ratio at moderate task loads but struggles under extreme conditions, similar to its performance in average delay. These results may highlight the HOODIE superior ability to minimize task computation delays and drop rates by dynamically adjusting offloading decisions based on current system conditions.

2) The impact of Computational Capacity

Fig. 10b illustrates the average delay as a function of CPU computation capacity dedicated to local tasks across different offloading schemes. As the CPU capacity increases from $3 - 7$ GHz, most schemes show a reduction in average delay, highlighting the importance of local computational

resources in minimizing task processing times. Notably, the FLC scheme, which relies solely on local computation, maintains a consistent delay across all CPU capacities, as it directly benefits from the increased local resources. However, it does not outperform other schemes that leverage offloading strategies. HOODIE consistently achieves lower delays compared to other schemes, especially at higher CPU capacities, indicating its effective balance between local computation and offloading decisions. The MLEO scheme also performs well, but its delay reduction is less pronounced compared to HOODIE, particularly at lower CPU capacities, where offloading decisions become more critical.

Fig. 10e shows the drop ratio as a function of CPU computation capacity for the same set of offloading schemes. As expected, higher CPU capacities lead to a reduction in drop ratios across most schemes, as more tasks can be processed locally, reducing the likelihood of tasks being dropped due to deadline violations. HOODIE demonstrates the most significant reduction in drop ratio, particularly at lower CPU capacities, where its dynamic offloading strategy helps manage the limited local resources more effectively. The FLC scheme shows decreasing drop ratio as CPU capacity increases. However, it fails to outperform HOODIE and MLEO due to its limitation in handling larger task loads without the ability to offload tasks. The VO and HO schemes show poor performance, as expected. These results underscore the advantages of the HOODIE algorithm in minimizing task drop rates by optimally balancing local computation and offloading, particularly in resource-constrained environments.

3) The impact of Task Timeout

Fig. 10c presents the average delay as a function of task timeout across different offloading schemes. Evidently, as the task timeout increases from 9.6 to 10.4 seconds, the average delay is slightly improved for all schemes. This trend suggests that longer timeouts allow tasks to wait longer before being processed, potentially leading to more efficient task completion. Notably, the FLC scheme maintains relatively consistent delay values across all timeouts, reflecting its exclusive reliance on local computation, which is unaffected by offloading decisions. In contrast, HOODIE consistently achieves lower average delays compared to other schemes, indicating its ability to optimize task scheduling and offloading dynamically even as the deadline increases. The MLEO scheme also shows strong performance but with slightly higher delays than HOODIE, particularly at shorter timeouts, where the need for accurate offloading decisions is more critical.

Fig. 10f illustrates the drop ratio as a function of task timeout for the same set of offloading schemes. As expected, the drop ratio decreases as the task timeout extends from 1.6 to 2.4 seconds, allowing more time for tasks to be processed and reducing the likelihood of deadline violations.

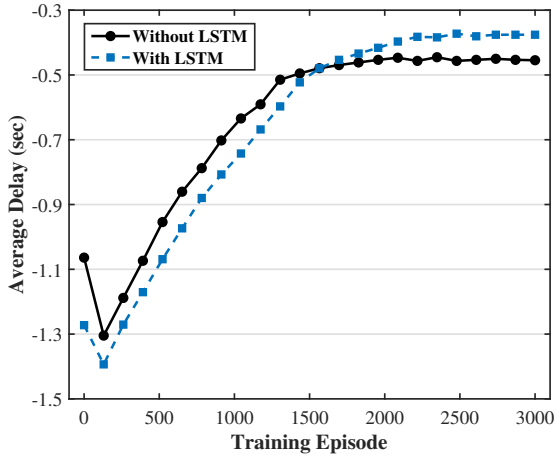

FIGURE 11. Average task delay of HOODIE with *vs* without the LSTM inclusion as a function of the training episodes.

HOODIE demonstrates the lowest drop ratios across all timeout values. These findings underscore the advantages of the HOODIE algorithm in maintaining low drop rates, particularly in scenarios with tighter deadlines, where efficient task management is crucial for overall system reliability.

D. THE IMPACT OF LSTM INCLUSION

To evaluate the impact of the LSTM module, we conducted an ablation study comparing the performance of the HOODIE framework with and without the inclusion of the LSTM. The system parameters for this analysis were configured with 20 Edge Agents (EAs) an the optimal values of γ and α_{lr}, as presented in Section V.A. We considered a task arrival probability $P = 0.3$ and task deadlines $\phi_n = 1$ sec for all tasks, whereas the rest of the environment parameters are noted in Table 4. As shown in Fig. 11, over the course of training, HOODIE with LSTM converges to lower task delays (about 0.35 sec) compared to the HOODIE without LSTM (about 0.48 sec). This might be attributed to the fact that the LSTM-based HOODIE can better handle the dynamic traffic patterns, which is reflected in more efficient offloading decisions and task handling. It is also remarkable that LSTM-based HOODIE scheme has slower convergence than the non-LSTM scheme, potentially due to the concurrent learning of the traffic patterns. However, it provides faster task execution delay by about 0.13 sec, which might be critical for the tasks to reach their deadlines. Note also that the inclusion of LSTM is also valuable for the recovery mechanism presented in Section IV.E, where the LSTM outputs are utilized for replacing missing state inputs from delayed or disrupted agents.

E. ADAPTABILITY TO TRAFFIC VARIATIONS

HOODIE scheme has been trained under various traffic conditions, with the task arrival probability P determining the traffic intensity across agents. There might be situations of

unexpected traffic variations (e.g. traffic burst or decline) in which HOODIE could adopt an adaptive behavior to handle these changes. One potential solution would be the P *value-baled model-switching mechanism* that allows the agents to respond flexibly to varying traffic conditions without the need for retraining in real-time. Specifically, after the training phase, agents could store their multiple models (each trained for different P) in a model catalogue. This catalogue allows agents to operate effectively under both moderate and high traffic loads by switching models as needed. As traffic conditions change, agents continuously monitor the current traffic load and select the most appropriate model from the catalogue based on the observed P value. Although during the transition from one traffic condition to another, the system may experience temporary network congestion as an unavoidable consequence of adjusting to the new traffic conditions, this mechanism can enhance the adaptability and minimize the system degradation. Noteworthy, further minimization of these transient delays might be achieved by combining HOODIE with other traffic anomaly detection algorithms (e.g. [42]).

VI. CONCLUSION AND FUTURE DIRECTIONS

A. CONCLUSIONS

This paper presented the HOODIE scheme, a distributed DRL-based framework for hybrid task offloading in the Cloud-Edge Computing Continuum (CEC). HOODIE introduces a flexible, scalable solution that enables both vertical (edge-to-cloud) and horizontal (edge-to-edge) offloading, offering delay-aware distributed task placement decisions in a multi-agent and dynamic CEC. By deploying distributed DRL agents at each edge server, HOODIE can make autonomous decisions based on local conditions while dynamically adapting to changing network environments. Through simulations, we demonstrated that HOODIE significantly reduces task drop rates and average task delays compared to baseline schemes, especially under heavy traffic conditions. The results highlight the effectiveness of HOODIE in balancing resource utilization across the CEC, making it highly applicable in collaborative cloud-edge scenarios.

B. FUTURE DIRECTIONS

There are direct extensions of the HOODIE framework that might be identified. The first extension would involve incorporating energy efficiency into the objective function. By considering different power consumption levels of fast-processing edge nodes, HOODIE could be enhanced to find an optimal trade-off between reducing latency and minimizing power consumption, under both latency and energy constraints. This would allow the agents to dynamically decide when it is more efficient to exploit high-power nodes or conserve energy. A second extension would be to accommodate a dynamic CEC topology, where the connections between edge nodes and their computational capacities change over time. This could be modeled within the current HOODIE framework, allowing the agents to adjust their task offloading strategies based on evolving network topologies and node heterogeneity. Finally, a third extension would introduce a parallel distributed DRL framework that operates between the IoT and Edge layers. Running in parallel with the HOODIE framework, this extension would enable DRL agents at end-devices to autonomously decide whether to compute tasks locally or offload them to edge servers, depending on the computational intensity of the tasks and available user resources.

ACKNOWLEDGMENT

The authors would like to thank the ICOS Project partners for their contributions. This work was supported in part by the ICOS Project funded from the EU HORIZON RIA programme under grant agreement No 101070177 (https://www.icos-project.eu/).

REFERENCES

[1] P. Trakadas, X. Masip-Bruin, F. M. Facca, S. T. Spantideas, A. E. Giannopoulos, N. C. Kapsalis, R. Martins, E. Bosani, J. Ramon, R. G. Prats *et al.*, "A reference architecture for cloud–edge meta-operating systems enabling cross-domain, data-intensive, ml-assisted applications: Architectural overview and key concepts," *Sensors*, vol. 22, no. 22, p. 9003, 2022.

[2] S. Spantideas, A. Giannopoulos, M. A. Cambeiro, O. Trullols-Cruces, E. Atxutegi, and P. Trakadas, "Intelligent mission critical services over beyond 5g networks: Control loop and proactive overload detection," in *2023 International Conference on Smart Applications, Communications and Networking (SmartNets)*. IEEE, 2023, pp. 1–6.

[3] Y. Mao, C. You, J. Zhang, K. Huang, and K. B. Letaief, "A survey on mobile edge computing: The communication perspective," *IEEE communications surveys & tutorials*, vol. 19, no. 4, pp. 2322–2358, 2017.

[4] K. Fu, W. Zhang, Q. Chen, D. Zeng, and M. Guo, "Adaptive resource efficient microservice deployment in cloud-edge continuum," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 8, pp. 1825–1840, 2021.

[5] K. Wang, J. Jin, Y. Yang, T. Zhang, A. Nallanathan, C. Tellambura, and B. Jabbari, "Task offloading with multi-tier computing resources in next generation wireless networks," *IEEE Journal on Selected Areas in Communications*, vol. 41, no. 2, pp. 306–319, 2022.

[6] Q. Fan and N. Ansari, "Towards workload balancing in fog computing empowered iot," *IEEE Transactions on Network Science and Engineering*, vol. 7, no. 1, pp. 253–262, 2018.

[7] Q. Zhang, M. Lin, L. T. Yang, Z. Chen, and P. Li, "Energy-efficient scheduling for real-time systems based on deep q-learning model," *IEEE transactions on sustainable computing*, vol. 4, no. 1, pp. 132–141, 2017.

[8] M.-T. Thai, Y.-D. Lin, Y.-C. Lai, and H.-T. Chien, "Workload and capacity optimization for cloud-edge computing systems with vertical and horizontal offloading," *IEEE Transactions on Network and Service Management*, vol. 17, no. 1, pp. 227–238, 2019.

[9] A. Giannopoulos, S. Spantideas, N. Kapsalis, P. Gkonis, L. Sarakis, C. Capsalis, M. Vecchio, and P. Trakadas, "Supporting intelligence in disaggregated open radio access networks: Architectural principles, ai/ml workflow, and use cases," *IEEE Access*, vol. 10, pp. 39 580–39 595, 2022.

[10] Y. Wang, M. Sheng, X. Wang, L. Wang, and J. Li, "Mobile-edge computing: Partial computation offloading using dynamic voltage scaling," *IEEE Transactions on Communications*, vol. 64, no. 10, pp. 4268–4282, 2016.

[11] S. Sardellitti, G. Scutari, and S. Barbarossa, "Joint optimization of radio and computational resources for multicell mobile-edge computing," *IEEE Transactions on Signal and Information Processing over Networks*, vol. 1, no. 2, pp. 89–103, 2015.

[12] Y. Mao, J. Zhang, S. Song, and K. B. Letaief, "Stochastic joint radio and computational resource management for multi-user mobile-edge computing systems," *IEEE transactions on wireless communications*, vol. 16, no. 9, pp. 5994–6009, 2017.
[13] C. You, K. Huang, H. Chae, and B.-H. Kim, "Energy-efficient resource allocation for mobile-edge computation offloading," *IEEE Transactions on Wireless Communications*, vol. 16, no. 3, pp. 1397–1411, 2016.
[14] J. Chen, H. Xing, Z. Xiao, L. Xu, and T. Tao, "A drl agent for jointly optimizing computation offloading and resource allocation in mec," *IEEE Internet of Things Journal*, vol. 8, no. 24, pp. 17 508–17 524, 2021.
[15] H. Li, K. D. R. Assis, S. Yan, and D. Simeonidou, "Drl-based long-term resource planning for task offloading policies in multiserver edge computing networks," *IEEE Transactions on Network and Service Management*, vol. 19, no. 4, pp. 4151–4164, 2022.
[16] J. Wang, J. Hu, G. Min, A. Y. Zomaya, and N. Georgalas, "Fast adaptive task offloading in edge computing based on meta reinforcement learning," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 1, pp. 242–253, 2020.
[17] Y. G. Kim and C.-J. Wu, "Autoscale: Energy efficiency optimization for stochastic edge inference using reinforcement learning," in *2020 53rd Annual IEEE/ACM international symposium on microarchitecture (MICRO)*. IEEE, 2020, pp. 1082–1096.
[18] Y. Kim and C. Wu, "Autoscale: Optimizing energy efficiency of end-to-end edge inference under stochastic variance," in *arXiv preprint*. arXiv, 2020, p. arXiv:2005.02544.
[19] G. Nieto, I. de la Iglesia, U. Lopez-Novoa, and C. Perfecto, "Deep reinforcement learning techniques for dynamic task offloading in the 5g edge-cloud continuum," *Journal of Cloud Computing*, vol. 13, no. 1, p. 94, 2024.
[20] W. Fan, F. Yang, P. Wang, M. Miao, P. Zhao, and T. Huang, "Drl-based service function chain edge-to-edge and edge-to-cloud joint offloading in edge-cloud network," *IEEE Transactions on Network and Service Management*, vol. 20, no. 4, pp. 4478–4493, 2023.
[21] S. Rac and M. Brorsson, "Cost-aware service placement and scheduling in the edge-cloud continuum," *ACM Transactions on Architecture and Code Optimization*, vol. 21, no. 2, pp. 1–24, 2024.
[22] Y. He, M. Yang, Z. He, and M. Guizani, "Computation offloading and resource allocation based on dt-mec-assisted federated learning framework," *IEEE Transactions on Cognitive Communications and Networking*, 2023.
[23] M. Tang and V. W. Wong, "Deep reinforcement learning for task offloading in mobile edge computing systems," *IEEE Transactions on Mobile Computing*, vol. 21, no. 6, pp. 1985–1997, 2020.
[24] H. Nashaat, W. Hashem, R. Rizk, and R. Attia, "Drl-based distributed task offloading framework in edge-cloud environment," *IEEE Access*, 2024.
[25] S. Ding and D. Lin, "Multi-agent reinforcement learning for cooperative task offloading in distributed edge cloud computing," *IEICE TRANSACTIONS on Information and Systems*, vol. 105, no. 5, pp. 936–945, 2022.
[26] I. Ullah, H.-K. Lim, Y.-J. Seok, and Y.-H. Han, "Optimizing task offloading and resource allocation in edge-cloud networks: a drl approach," *Journal of Cloud Computing*, vol. 12, no. 1, p. 112, 2023.
[27] G. Qu, H. Wu, R. Li, and P. Jiao, "Dmro: A deep meta reinforcement learning-based task offloading framework for edge-cloud computing," *IEEE Transactions on Network and Service Management*, vol. 18, no. 3, pp. 3448–3459, 2021.
[28] A. Angelopoulos, A. Giannopoulos, N. Nomikos, A. Kalafatelis, A. Hatziefremidis, and P. Trakadas, "Federated learning-aided prognostics in the shipping 4.0: Principles, workflow, and use cases," *IEEE Access*, 2024.
[29] Z. Wang, T. Schaul, M. Hessel, H. Hasselt, M. Lanctot, and N. Freitas, "Dueling network architectures for deep reinforcement learning," in *International conference on machine learning*. PMLR, 2016, pp. 1995–2003.
[30] J. Liu, Y. Mao, J. Zhang, and K. B. Letaief, "Delay-optimal computation task scheduling for mobile-edge computing systems," in *2016 IEEE international symposium on information theory (ISIT)*. IEEE, 2016, pp. 1451–1455.
[31] A. K. Parekh and R. G. Gallager, "A generalized processor sharing approach to flow control in integrated services networks: the single-node case," *IEEE/ACM transactions on networking*, vol. 1, no. 3, pp. 344–357, 1993.
[32] A. Giannopoulos, S. Spantideas, N. Kapsalis, P. Karkazis, and P. Trakadas, "Deep reinforcement learning for energy-efficient multi-channel transmissions in 5g cognitive hetnets: Centralized, decentralized and transfer learning based solutions," *IEEE Access*, vol. 9, pp. 129 358–129 374, 2021.
[33] H. Hasselt, "Double q-learning," *Advances in neural information processing systems*, vol. 23, 2010.
[34] R. S. Sutton, "Reinforcement learning: An introduction," *A Bradford Book*, 2018.
[35] J. Fan, Z. Wang, Y. Xie, and Z. Yang, "A theoretical analysis of deep q-learning," in *Learning for dynamics and control*. PMLR, 2020, pp. 486–489.
[36] V. A. Papavassiliou and S. Russell, "Convergence of reinforcement learning with general function approximators," in *IJCAI*, vol. 99, 1999, pp. 748–755.
[37] S. Khare, H. Sun, K. Zhang, J. Gascon-Samson, A. Gokhale, X. Koutsoukos, and H. Abdelaziz, "Scalable edge computing for low latency data dissemination in topic-based publish/subscribe," in *2018 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE, 2018, pp. 214–227.
[38] T. Rausch, S. Nastic, and S. Dustdar, "Emma: Distributed qos-aware mqtt middleware for edge computing applications," in *2018 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2018, pp. 191–197.
[39] C. Wang, C. Liang, F. R. Yu, Q. Chen, and L. Tang, "Computation offloading and resource allocation in wireless cellular networks with mobile edge computing," *IEEE Transactions on Wireless Communications*, vol. 16, no. 8, pp. 4924–4938, 2017.
[40] F. Alhaidari and T. Z. Balharith, "Enhanced round-robin algorithm in the cloud computing environment for optimal task scheduling," *Computers*, vol. 10, no. 5, p. 63, 2021.
[41] J. Liu and Q. Zhang, "Offloading schemes in mobile edge computing for ultra-reliable low latency communications," *Ieee Access*, vol. 6, pp. 12 825–12 837, 2018.
[42] Y. Ren, H. Jiang, N. Ji, and H. Yu, "Tbsm: A traffic burst-sensitive model for short-term prediction under special events," *Knowledge-Based Systems*, vol. 240, p. 108120, 2022.

ANASTASIOS E. GIANNOPOULOS (Member, IEEE) (M.Eng, Ph.D) received the diploma of Electrical and Computer Engineering from the National Technical University of Athens (NTUA), where he also completed his Master Engineering (M.Eng) degree, in 2018. He also obtained his Ph.D. at the Wireless and Long Distance Communications Laboratory of NTUA. His research interests include advanced Optimization Techniques for Wireless Systems, ML-assisted Resource Allocation, Maritime Communications and Multi-dimensional Data Analysis.

He is currently working as a Research Associate at Department of Ports Management and Shipping, in the National and Kapodistrian University of Athens, as well as Post-Doc at NTUA. He has authored more than 45 scientific publications in the fields of Wireless Network Optimization, Cognitive Resource Management, Maritime Communications, Machine Learning and Brain Multi-dimensional Analysis. Since 2022, he is a Member of IEEE and reviewer in several IEEE journals (IEEE Internet of Things Journal, IEEE Vehicular Technology Magazine, IEEE Transactions on Mobile Computing IEEE Network, IEEE Access).

ILIAS PARALIKAS (M.Eng) holds a Diploma in Electrical and Computer Engineering from the National Technical University of Athens (NTUA) since 2023. He has also obtained the Master in Engineering (M.Eng) in 2024 from NTUA, with his thesis being focused on Artificial Intelligence applications in Smart Applications. His research interests consists of a variety of fields around artificial intelligence. These are composed of Artificial Neural Networks, compression methods, such as model pruning, knowledge distillation and early exit in classification problems. Additional interests consist of medical image segmentation, leveraging data augmentation and Generative Adversarial Network techniques, as a method to battle data scarcity. Finally, he has extensively worked on Deep Reinforcement Learning methods, applied to a variety of telecommunications task offloading scenarios. He is currently working on the research & development (R&D) department of Four Dot Infinity, a company providing smart solutions to businesses.

SOTIRIOS T. SPANTIDEAS (D.Eng, M.Sc, Ph.D) obtained the Diploma of Electrical & Computer Engineering from the Polytechnic School of the University of Patras in 2010. He then attended the Master Program ”Electrophysics” at the Royal Institute of Technology in Stockholm (KTH), from which he obtained the title MSc in 2013. In 2018 he obtained his PhD from the National Technical University of Athens (NTUA) with doctoral dissertation entitled “Development of Methods for obtaining DC and low frequency AC magnetic cleanliness in space missions”. His research interests include Electromagnetic Compatibility, Machine Learning for Wireless Networks, Magnetic Cleanliness for space missions and optimization algorithms for Computational Electromagnetics.

From 2014, he is working as a Research Associate with NTUA and the National and Kapodistrian University of Athens (Department of Ports Management and Shipping - NKUA), participating in multiple Horizon projects. He has published over 35 papers in scientific journals and conferences in the fields of Electromagnetic Compatibility, Optimization Methods for Wireless Networks and Machine Learning for Resource Allocation problems.

PANAGIOTIS TRAKADAS (MEng, Ph.D) received his Diploma, Master Engineering, Degree in Electrical & Computer Engineering and his Ph.D. from the National Technical University of Athens. His main interests include 5G/6G technologies, such as O-RAN, NFV and NOMA, Wireless Sensor Networks, and Machine Learning based Optimization Techniques for Wireless Systems and Maritime Communications.

He is currently an Associate Professor at the Department of Ports Management and Shipping, in National and Kapodistrian University of Athens and has been actively involved in many national and EU-funded research projects. He has published more than 170 papers in magazines, journals, books and conferences and a reviewer in several journals and TPC in conferences.