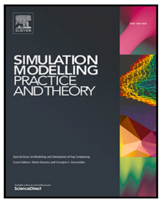
Simulation Modelling Practice and Theory 144 (2025) 103170

Contents lists available at ScienceDirect

Simulation Modelling Practice and Theory

journal homepage: www.elsevier.com/locate/simpat



Comparing Control Theory and Deep Reinforcement Learning techniques for decentralized task offloading in the edge–cloud continuum

Gorka Nieto [a,b,*], Neco Villegas [c], Luis Diez [c], Idoia de la Iglesia [a], Unai Lopez-Novoa [b], Cristina Perfecto [b], Ramón Agüero [c]

[a] *Ikerlan Technology Research Centre, Basque Research and Technology Alliance (BRTA), P°. J.Mª. Arizmendiarrieta, 2, Arrasate-Mondragón, 20500, Spain*
[b] *University of the Basque Country (UPV/EHU). School of Engineering in Bilbao, Alameda Urquijo s/n, Bilbao, 48013, Spain*
[c] *Universidad de Cantabria. Communications Engineering Dpt., Av. de los Castros, s/n, Santander, 39005, Spain*

A R T I C L E I N F O

Keywords:
Internet-of-Things (IoT)
Edge–Cloud-Continuum
Task offloading
Deep Reinforcement Learning (DRL)
Lyapunov
Energy consumption
Optimization

A B S T R A C T

With the increasingly demanding requirements of Internet-of-Things (IoT) applications in terms of latency, energy efficiency, and computational resources, among others, task offloading has become crucial to optimize performance across edge and cloud infrastructures. Thus, optimizing the offloading to reduce latency as well as energy consumption and, ultimately, to guarantee appropriate service levels and enhance performance has become an important area of research. There are many approaches to guide the offloading of tasks in a distributed environment, and, in this work, we present a comprehensive comparison of three of them: A Control Theory (CT) Lyapunov optimization method, 3 Deep Reinforcement Learning (DRL) based strategies and traditional solutions, like Round-Robin or static schedulers. This comparison has been conducted using *ITSASO*, an in-house developed simulation platform for evaluating decentralized task offloading strategies in a three-layer computing hierarchy comprising IoT, fog, and cloud nodes. The platform models service generation in the IoT layer using a configurable distribution, enabling each IoT node to decide whether to autonomously execute tasks (locally), offload them to the fog layer, or send them to the cloud server. Our approach aims to minimize the energy consumption of devices while meeting tasks' latency requirements. Our simulation results reveal that Lyapunov optimization excels in static environments, while DRL approaches prove to be more effective in dynamic settings, by better adapting to changing requirements and workloads. This study offers an analysis of the trade-offs between these solutions, highlighting the scenarios in which each scheduling approach is most suitable, thereby contributing valuable theoretical insights into the effectiveness of various offloading strategies in different environments. The source code of *ITSASO* is publicly available.

1. Introduction

IoT has enabled new applications in the industry, such as real-time health monitoring, automation of processes such as material handling and product assembly, and quality control of produced parts. IoT technology offers benefits, including increased

* Corresponding author at: Ikerlan Technology Research Centre, Basque Research and Technology Alliance (BRTA), P°. J.Mª. Arizmendiarrieta, 2, Arrasate-Mondragón, 20500, Spain.

E-mail addresses: gnieto@ikerlan.es (G. Nieto), neco.villegas@unican.es (N. Villegas), luisfrancisco.diez@unican.es (L. Diez), idelaiglesia@ikerlan.es (I. de la Iglesia), unai.lopez@ehu.eus (U. Lopez-Novoa), cristina.perfecto@ehu.eus (C. Perfecto), ramon.agueroc@unican.es (R. Agüero).

https://doi.org/10.1016/j.simpat.2025.103170

Available online 3 July 2025

efficiency, reduced costs, improved productivity, and better visibility into the supply chain [1]. The convergence of IoT and wireless communication technologies, such as 5G, is revolutionizing the way applications are developed and deployed. In addition, the market value of the IoT is expected to hit 6 trillion in 2025 at a growth rate of 15.12% [2].

However, IoT devices' capabilities in terms of computation or memory are usually not enough to perform all eventual tasks. Therefore, IoT relies on different processing architectures to handle data, each suited for specific application needs and circumstances. The combination of these different approaches form what is known as edge–cloud continuum, comprising devices ranging from the smallest ones, normally close to the end user (User Equipment (UE)), to the most powerful nodes, such as Cloud Servers (CSs). This results in hybrid architectures that allow the user to take advantage of both Cloud Computing (CC) and Fog Computing (FC) features [3].

Considering that some UEs like IoT devices often have limited processing power and battery life, offloading computationally intensive tasks to external servers allows applications to run more efficiently, conserving battery and improving overall performance. Offloading apps can also help distributing resources more efficiently, allowing applications to support a larger number of users and handle more data without hindering performance [4]. Despite computation offloading presents numerous advantages, it also introduces challenges such as potential network congestion and the need for robust algorithms to manage dynamic resource allocation effectively [5].

Hence, careful consideration of network conditions, task requirements, and server occupation is essential for successful implementation of these offloading strategies. The quality of the communication can significantly impact the effectiveness of offloading, while not all tasks are suitable for offloading. In summary, offloading has the potential to enhance performance by improving Quality-of-Service (QoS) and optimizing resources, but the algorithms need to adapt to the data nature and service requirements. Therefore, computation offloading in edge–cloud environments addresses several critical challenges associated with resource limitations, latency, and efficiency. IoT devices have restricted computational power and battery life, making it difficult to process complex tasks locally, which is crucial for battery-operated devices [6]. In addition, offloading tasks to Fog Servers (FSs) minimizes data transfer time compared to remote cloud servers, significantly reducing latency for time-sensitive applications [7]. When tasks are effectively distributed, computation offloading enhances the execution efficiency of user tasks, alleviating pressure on both fog and cloud servers [8].

To address the computation offloading problem, it becomes essential to adequately model and simulate all the elements that make up an offloading scenario, including the characteristics of nodes and servers, or the conditions of the communication channel, among others. Thus, simulation frameworks are essential for optimizing task offloading across the edge–cloud continuum, as they allow service providers and users to prevent potential bottlenecks and inefficiencies, saving both time and money by preventing costly real-world trial and error. Simulating different configurations and workloads helps in making informed decisions, ensuring that resources are used optimally and service quality is maintained, so having the opportunity to make reproducible and controllable experiments before deployment in a real environment [9].

Different techniques have been explored to solve the non-trivial problem of computation offloading. For instance, CT based approaches use mathematical models to manage and optimize offloading decisions, ensuring stability and performance. CT offers precise and predictable outcomes but may struggle with complex, dynamic environments. On the other hand, Machine Learning (ML) techniques, such as DRL, enable dynamic and adaptive offloading strategies by learning from the environment and making real-time decisions. ML provides flexibility and adaptability but it can require significant computational resources and training data. In this sense, DRL can help solve this issue, as it does not require a training dataset, but it learns by interacting directly with the environment, being compromised by its own experience [10].

In this work, we compare the performance of CT and DRL approaches in edge–cloud environments, providing quantifiable comparisons and analysis of their trade-offs. We highlight the scenarios in which each scheduling method is most suitable, implementing optimized DRL and CT algorithms for dynamic task scheduling that run on the IoT nodes, which are focused on meeting latency requirements and optimizing energy consumption. To the best of our knowledge, this is the first work to directly compare DRL and CT approaches in different computation offloading scenarios, addressing key metrics such as latency and energy consumption.

To make this comparison, we introduce *ITSASO*, a flexible and configurable simulation platform to assess the performance of offloading strategies for computing tasks. It represents different entities: IoT nodes, FS and CS, along with the communication between them. The deployment of this platform allows us to exercise flexibility in conducting detailed examinations of the practical aspects of diverse solutions. This process ensures that our theoretical findings are aligned with their applicability in real systems. Furthermore, this control allows us to customize the platform to our specific needs and research goals, and to conduct experiments and simulations that precisely match our requirements, with different scheduling algorithms.

By conducting extensive simulations with our platform, we provide a comprehensive analysis of the strengths and weaknesses of each approach, offering valuable insights for future research and practical implementations in edge computing environments. Additionally, our flexible simulation platform, *ITSASO*, provides researchers with a highly configurable tool to assess the performance of their proposed solutions.

The remainder of this article is structured as follows. Section 2 provides a thorough review of related work, highlighting the differences between this paper and existing research and methodologies. Section 3 outlines the system model, depicting the architecture and components included in the simulation platform. Section 4 describes the simulation platform, discussing the tools used to build it and the different options that can be configured. Then, Section 5 introduces the evaluated algorithms, explaining their design, functionality, and how they address the offloading challenges. Section 6 discusses the obtained results, offering a detailed analysis of the performance and effectiveness of the algorithms in the simulation platform. Finally, Section 7 concludes the paper, summarizing the key findings, discussing their implications, and suggesting directions for future research.

2. Related work

This section reviews recent studies exploring different techniques and strategies for computation Offloading, along with the most relevant simulation platforms used for this kind of problems.

2.1. Approaches to solve computation offloading problems

As stated in Section 1, the increasing demand for processing-intensive applications and the limitations of IoT devices, make computation offloading a critical area of research, as seen in many recent surveys related to this field, such as [11–13] or [14], among others.

Most of these surveys [11,12,14] discuss Reinforcement Learning (RL)-based computation offloading fundamental principles and theories in Multi-access Edge Computing (MEC) environments. Despite offloading is particularly relevant in 5G networks, where MEC is offered to UEs to transfer computing tasks, it is not only limited to 5G-MEC environments, as it is also applicable in other networks, often referred to as Edge Computing (EC) or FC. These paradigms involve the distribution of computational tasks across various nodes through different types of networks, enhancing efficiency and resource utilization. In most of the works cited in the mentioned surveys, authors address mechanisms for finding optimal offloading decisions or methods for joint resource allocation, analyzing the challenges of RL-based computation offloading solutions, from both the perspective of algorithm design and realistic requirements that deserve more attention in future research. Sadatdiynov et al. [13], on the other hand, explore more types of optimization methods, including Lyapunov optimization, heuristic techniques or Game Theory, among others.

Centralized solutions, while leveraging powerful computational resources, often face scalability and adaptability issues. For instance, Xie et al. [15] introduce an enhancement of the Proximal Policy Optimization (PPO) algorithm named RAPPO, focusing on minimizing overall energy consumption while satisfying task parameter constraints with multiple Edge servers acting as centralized schedulers, as there is no CS in the system. Kim et al. [16] regulate the offloading ratio based on network conditions, optimizing both energy consumption and latency, with a method based on Mathematical Optimization (MO). However, the framework proposed in these centralized approaches faces challenges in scalability and real-time adaptability, particularly in response to user mobility and dynamic network conditions.

Decentralized approaches distribute decision-making across multiple agents, reducing reliance on a central scheduler and enhancing system scalability. These methods aim to improve adaptability in dynamic environments, but they often require complex coordination mechanisms. Several decentralized approaches have addressed energy efficiency in IoT offloading. For instance, authors in [17] employ a Deep Deterministic Policy Gradient (DDPG)-based DRL method enabling each user to learn offloading policies independently, focusing on minimizing power consumption and delay, but lack the comparison with other intelligent schemes. Zhu et al. introduce in [18] a distributed task offloading framework that uses the Advantage Actor-Critic (A2C) algorithm to enhance energy efficiency in edge–cloud environments. However, their study does not explore varying network conditions, as in the work by Long et al. [19], who propose a decentralized Actor-Critic (AC)-based approach to minimize the integrated cost of latency and energy consumption. Goudarzi et al. [20] propose a mobile CC environment where UEs offload computation-intensive applications to CSs and cloudlets. Their multisite offloading solution, based on Genetic Algorithm (GA), aims to reduce overall energy consumption and execution time. However, similar to earlier decentralized works, their approach does not consider dynamic environments, limiting its adaptability.

Hybrid solutions attempt to balance the trade-offs between centralized and decentralized paradigms, leveraging the strengths of both approaches while mitigating their weaknesses. He et al. [21] propose a system where each service domain is equipped with one or more edge servers, increasing flexibility and reliability but relying on a centralized scheduler that leverages a DDPG model to enhance service quality, particularly for latency-sensitive applications. Alternatively, Wen introduces in [22] a Multi-Agent (MA)-DDPG solution for Vehicle-to-everything (V2X) environments, integrating centralized training and distributed implementation phases to optimize resource utilization and reduce system delay. Dai et al. take this further by proposing in [23] a semi-distributed computation offloading approach for Industrial IoT (IIoT) environments. Their method, based on AC, seeks to minimize task latency and energy consumption across all UEs by combining centralized scheduling with distributed execution, but they do not account for different application requirements, limiting its flexibility across diverse use cases. Liu et al. [24] propose a hierarchical, multi-agent DRL framework (TMADO) for wireless powered MEC networks, where a centralized agent optimizes resource allocation and decentralized agents manage offloading. While this hybrid strategy improves energy efficiency, its reliance on central coordination limits its applicability in decentralized, large-scale IoT environments.

Table 1 summarizes the most relevant features of the works described above, comparing them with our proposal. As can be seen, the optimization goals can vary significantly, with many works aiming to minimize overall system energy consumption and/or experienced latency. We focus on minimizing the energy consumption of IoT devices themselves, under latency constraints and decentralized control. Some decentralized approaches also address energy efficiency of IoT devices, for example, Zhu et al. [18], but these studies are often limited to static environments or specific network assumptions, making them less generic. Moreover, such works typically explore a single algorithmic direction, without comparing multiple decision-making paradigms. This paper addresses this gap by evaluating and contrasting two different decentralized strategies, i.e. CT and DRL, in their ability to adaptively minimize energy consumption in both static and dynamic scenarios. Therefore, our work analyzes the behavior of different approaches in various scenarios, checking the importance and the need of adaptable and context-aware strategies in both static and dynamic environments.

Table 1
Summary of computation offloading contributions.

Work	Techniques		Optimization goal	Constraints	Layers (Decider with *)		
	I	II			Device	MEC/Fog	Cloud
[15]	DRL	PPO	Overall energy	Latency	X	X*	
[16]	MO	–	Latency & energy		X	X*	
[20]	Heuristics	GA	Overall energy & latency		X		X*
[17]	DRL	DDPG	Power consumption and buffering delay		X*	X	
[18]	DRL	A2C	Overall energy & Latency		X*	X	
[19]	DRL	AC	Latency & energy	Task Latency & Computing resources	X	X*	X
[21]	DRL	DDPG	Service quality		X	X	X*
[22]	DRL	DDPG	Latency		X*	X	
[23]	DRL	AC	Latency & energy		X	X*	
[24]	DRL	PPO	Needed local energy	Energy, delay and computation	X*	X	
Ours	DRL CT	AC Lyapunov	IoT energy	Task latency	X*	X	X

2.2. Simulation platforms for computation offloading

In order to tackle the Computation Offloading problem, researchers have developed many platforms in recent years, particularly with the advent of MEC and FC. Although many studies use ML approaches to solve computation offloading problems through simulation, not all the platforms developed allow using this type of solutions. They normally just let the user configure devices from different layers and some configurations of the scheduling algorithms. We discuss herewith some of the most relevant platforms.

iFogSim2 [25], as an extension of iFogSim [26], is a simulation platform built on top of the CC simulation toolkit CloudSim [27]. It is designed for modeling and simulating resource management techniques in IoT, edge, and FC environments. iFogSim2 allows users to model task scheduling, load balancing and resource allocation, to optimize performance metrics such as latency, energy consumption, cost and network congestion, and introducing features such as mobility or dynamic clustering, so real-world scenarios with dynamic devices and services can be simulated. This platform has been used to model different types of IoT applications, such as healthcare monitoring or crowd-sensing. Its main drawback is that it just supports centralized, rule-based offloading policies, so it is not suitable for decentralized DRL or CT scheduling schemes.

CETO-Sim [28] is a modular simulation platform designed for cloud–edge task offloading that includes task management, network topology, end-user, scheduling policy, server, and operation components. Users can simulate different computing environments and monitor task parameters in real-time, building different physical entities and combining them. CETO-Sim simulates task offloading through the interaction between these components while recording the changes of various task parameters and simulated entities in real-time. However, there is no source code or binary publicly available.

EdgeAISim [29] is a simulator where Artificial Intelligence (AI) models can be used to assess the performance of resource management techniques in EC systems based on Python. It extends EdgeSimPy [30] framework and develops new AI-based simulation models for task scheduling, energy management, service migration, network flow scheduling, and mobility support for EC environments. The simulator has been tested with models like Deep Q-Network (DQN) and AC, aiming to reduce power usage at the Edge. In contrast to *ITSASO*, it is designed for AI model placement and inference at the edge, instead of any software that runs in a container.

EdgeSim++ [31] is a simulator that can generate a batch of heterogeneous devices with various features, supporting the simulation of multi-layer cloud–edge-end scenarios and various MEC architectures. It is compatible with ML and non-ML offloading strategies, and supports network topology configuration, as well as real-time monitoring of device resource changes. It can simulate task offloading for both linear and discrete workflows, allowing for finer-grained strategy customization. Despite EdgeSim++ being a simulator with extensive configuration capabilities that allow users to test complex and multiple scenarios, it may be too complex compared to *ITSASO*. Additionally, *ITSASO* provides better process isolation and resource allocation features than EdgeSim++, as it uses Docker containers to run each simulated node

Finally, Table 2 summarizes the features of the reviewed simulators, the star-shaped marker (*) in the *Layers* column represents the location in which the offloading decision is taken.

ITSASO is the only tool that provides decision capabilities in the edge/IoT layer, providing a completely decentralized decision-making environment. Written in Python, allows using ML and CT solutions, along with the possibility of configuring task generation process according to the arrival distribution chosen by the user. In *ITSASO*, each node is deployed in a separate Docker container, which provides more isolation, consistency, and resource efficiency than the analyzed tools. This also makes *ITSASO* highly portable and easier to manage across different environments.

3. System model

This section provides an overview of the architecture and components involved in this work. We described the layered architecture design and its main entities. Furthermore, we introduce the modeling of both latency and energy consumption for the computation and communication between each layer.

Table 2
Simulation platforms for computation offloading.

Name	Ref	Language	App gen	Layers			Techniques		
				E	F	C	ML	CT	Other
iFogSim2	[25]	Java		✓	✓*				✓
CETO-Sim	[28]	N/A		✓	✓*	✓			✓
EdgeAISIM	[29]	Python		✓	✓	✓*	✓		✓
EdgeSim++	[31]	C++	✓	✓	✓	✓*	✓		✓
ITSASO	–	Python	✓	✓*	✓	✓	✓	✓	✓

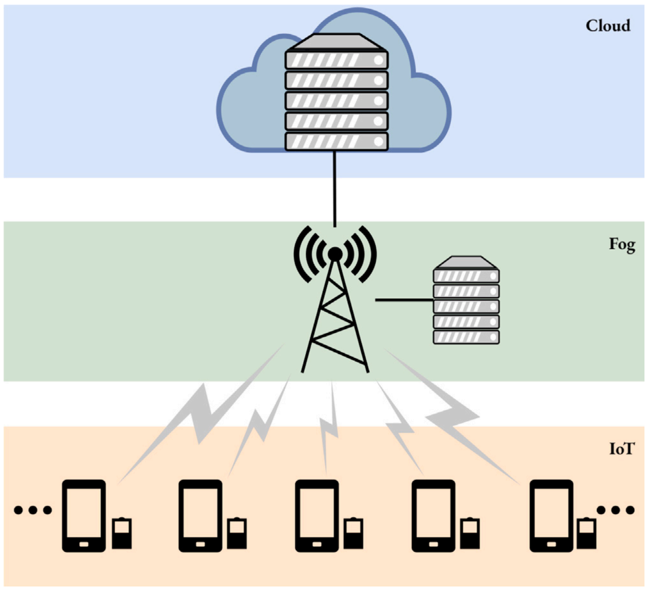


Fig. 1. System model.

3.1. Architecture

Fig. 1 illustrates the architecture that we consider, comprising three different layers, which, altogether create an efficient framework for managing different applications according to their requirements.

First, the IoT layer comprises all UEs in the environment, corresponding to resource- and energy-constrained devices where the computing applications are created. Each UE must determine how to handle incoming services using their own instances of the chosen decentralized decision-making algorithms that consider different factors, such as service length, as will be detailed later in Section 5. The nodes in this layer have reduced computation capabilities for the incoming services.

The FC layer extends UEs' computation capabilities, still rather close to the UEs. To reach the FS, the UE must send a service via a modeled wireless link, so the FS inserts it into its computing queue. Each FS is positioned adjacent to a Base Station (BS) or Access Point (AP), so no extra delay is considered between this and the FS.

Regarding the CC layer, it offers more computation power than the FC layer, but far from the devices where the applications are generated. Thus, besides the transmission delay between the UE and the AP, the communication delay between the AP and the CS must be also considered.

3.2. Latency modeling

First, we define the latency of a service (σ) as the total delay suffered by computation services from the moment they are generated until they are completely processed, and the results are available in the node where they were generated.

Thus, the latency experienced when executing a service locally ($d_{\sigma,i}$) is defined as the time elapsed since it was inserted into the application queue and processed in the node. This total latency is shown in Eq. (1):

$$d_{\sigma,i} = d_{app} + d_i + d_{dec,i} + d_{q,i} + d_{proc,i} \tag{1}$$

where d_{app} is the time the service remains in the application queue, d_i corresponds to the time it remains in the IoT node's queue before being processed, d_{dec} is the time taken by the corresponding algorithm deciding where the service must be processed, $d_{q,i}$ is the queuing time of the IoT processor and d_{proc} is the processing time of the service, when executed locally.

Regarding the total delay when a service is sent to the FS, the time the service remains in the IoT node's queue before being processed (d_i) and the time taken by the corresponding algorithm to decide where the service must be processed (d_{dec}) is the same, but some additional delays must be considered. The transmission time from the IoT node to the FS ($d_{tx,i-f}$) must be included in the calculation, along with the propagation delay between the IoT node ($d_{p,i-f}$) and the FS and the queuing delay at the FS ($d_{q,f}$), the processing time in the node ($d_{proc,f}$), the return time between the fog and the IoT node and the propagation delay back to the IoT node ($d_{rx,f-i}$), as seen in Eq. (2):

$$d_{\sigma,f} = d_{app} + d_{node} + d_{dec} + d_{tx,i-f} + d_{p,i-f} + d_{q,f} + d_{proc,f} + d_{rx,f-i} + d_{p,f-i} \tag{2}$$

The transmission delay between nodes ($d_{tx,i-f}$) is defined as the time needed to send all data from the IoT node to FS, and it is calculated as shown in Eq. (3):

$$d_{tx,i-f} = \sigma_s / R_{i-f} = (\sigma_p \cdot p_s)/R_{i-f} \tag{3}$$

this is, the total data to be transmitted divided by the transmission rate (R) between the IoT node and the FS. Here, the data to be transmitted (or service size (s_σ)) is determined by the packets' size p_s and the number of packets that compose the service (σ_p).

Regarding the propagation delay ($d_{p,i-f}$), it is negligible compared to transmission, queuing, and processing delays due to dynamic traffic control, which are the most limiting factors. The return time between the fog and the IoT node is also assumed to be 0, due to the very small length of the response and the higher transmission capabilities of the BS.

Thus, obviating neglected values, the final delay when a service is offloaded to the FS is given by Eq. (4):

$$d_{\sigma,f} = d_{app} + d_{node} + d_{dec} + d_{tx,i-f} + d_{q,f} + d_{proc,f} \tag{4}$$

When the service is sent to the CS, the transmission delay between the FS and the CS is also considered, besides queuing and processing delays within the CS. Altogether, it results in the total transmission delay shown in Eq. (5):

$$d_{\sigma,c} = d_{app} + d_{node} + d_{dec} + d_{tx,i-f} + d_{tx,f-c} + d_{q,c} + d_{proc,c} \tag{5}$$

The processing latency experienced when performing a task in any of the nodes $d_{proc,x}$ is calculated as can be seen in Eq. (6):

$$d_{proc,x} = \sigma_s / F_x \tag{6}$$

where s_σ is the service size and F_x corresponds to the processing capacity of node x.

3.3. Energy modeling

Regarding energy consumption when the decision is to execute a task locally, $e_n(t)$, it can be represented as shown in Eq. (7), which has been adapted from [32]:

$$e_{m,i} = \kappa_n (f_n)^2 \phi_i \tag{7}$$

where κ_n represents the energy coefficient of the UE, $(f_n)^2$ is the processing capacity of the UE and ϕ_i the computation consumption of service σ.

The energy consumption for the UE when offloading a service to the FS can be easily calculated as the product of the UE transmission power ($p_{n,tx}$) and the transmission time of the task to the FS ($d_{tx,i-f}$). Therefore, using the previously stated upload transmission delay (Eq. (3)), the energy consumed when offloading a task is obtained as shown in Eq. (8):

$$e_{\sigma,f} = p_{i,tx} \cdot d_{tx,i-f} = p_{i,tx} \cdot \sigma_s / R_{i-f} = p_{i,tx} \cdot (\sigma_p \cdot p_s)/R_{i-f} \tag{8}$$

Finally, the energy consumption when the decision is to offload to the cloud (Eq. (9)) is the same as when the decision is to offload to the FS (Eq. (8)). It is worth recalling that, before reaching the cloud, a task must first go through the BS where the FS is located to get to the CS, which does not imply a higher energy consumption from the UE:

$$e_{\sigma,c} = e_{\sigma,f} = p_{i,tx} \cdot (\sigma_p \cdot p_s)/R_{i-f} \tag{9}$$

4. Simulation platform

This section presents *ITSASO*, an in-house developed simulator that follows the model presented in the previous section. *ITSASO* stands for *IoT Task Simulation and Adaptive Scheduling for Offloading* and is used to assess the performance of different offloading schemes. Its source code has been made publicly available.[1] A previous version was used in [33].

The system model presented in *ITSASO* is illustrated in Fig. 2: there are 3 types of nodes, namely UE, FS, and CS, deployed in 3 layers represented in different colors.

UEs constitute the IoT layer and are designed to replicate the behavior of IoT devices, including their battery constraints by the definition of a limited battery capacity. An UE is responsible for generating independent traffic flows belonging to different

[1] *ITSASO* public repository: https://github.com/tlmat-unican/ITSASO.

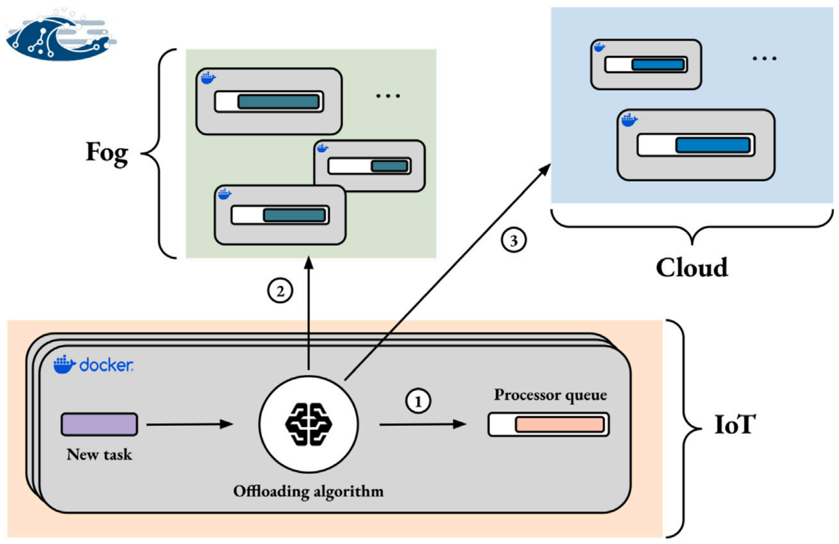


Fig. 2. *ITSASO* architecture overview.

applications with varying QoS requirements. Traffic is simulated within IoT nodes to evaluate network performance under various conditions. Applications generate services at each time slot. Then, these services can be either computed locally (at the IoT nodes) or offloaded to the FS or CS, depending on the decision taken by the IoT node itself, which is made based on its implemented scheduling policy and/or the system state.

To illustrate *ITSASO*'s internals, Fig. 3 presents a functional diagram of the platform, where different nodes are color-coded: the UE (orange), FS (green), and CS (blue). The arrows illustrate the packet flow through the various functions, most of which run in separate threads, with red arrows representing communication between nodes. The workflow is structured into four main steps, marked by numbered labels in the figure: (1) traffic generation, (2) service generation and offloading decision, (3) reception and forwarding of offloaded packets, and (4) processing and acknowledgment.

Let M be the number of m applications that generate services (σ). In the first step, the packet generation rate is defined according to pre-defined and configurable random distributions (i.e. Poisson, uniform, lognormal, etc.). Generated packets are then queued in the application buffer, awaiting the scheduling decision to process it locally or remotely (offloading). Then, in the second step, from the generated traffic, services (processing tasks) are created for each application in every slot. The size of a service is directly related to the number of packets it comprises, where each packet has a consistent, predefined size. Since the number of packets generated by the UE application within a slot follows a random distribution, the size of each service is also inherently random. Time is assumed to be slotted, so each application generates one service at every slot. Once all services for the current time slot have been created, the device sends a request to the offloading algorithm. *ITSASO* is designed to support various offloading strategies, allowing seamless integration of different algorithms.

When the algorithm decision is taken, the UE dequeues the corresponding packets and sends them to the corresponding processing point. As introduced in Section 3, there are 3 possible processing points:

- **Local.** The service is processed in the UE's local CPU, which typically has a low computation power.
- **Offloading to Fog layer.** Service processing is carried out in a FS. In this case, the UE transmits packets with some bytes of overhead, which include processing decision information, to complete the offloading process.
- **Offloading to Cloud layer.** The service can be sent to a CS to be processed. This node is characterized by its high computation capacity, but high latency due to its location.

The third step involves the reception of offloaded packets at the FS, which processes them directly or forwards them to a CS. When a packet arrives, the FS reads the headers:

- **Packet Count**. The first packet of each service specifies the total number of packets that comprise the service.
- **Offloading Flag**. A binary flag indicating the processing location for the service: 0 for the FS and 1 for the CS.
- **Source Address**. The FS is aware of the identity of the UE that initiated the service, ensuring the resulting message is returned to the correct source after the service is fully processed.
- **Service Id**. A unique identifier assigned to each service, enabling system-wide tracking and management.

If the processing point selected is the FS itself, the packets that comprise the service are queued in the FS processor buffer. Conversely, if the CS is selected as the offloading option, the FS prepares the service for transmission to the corresponding CS.

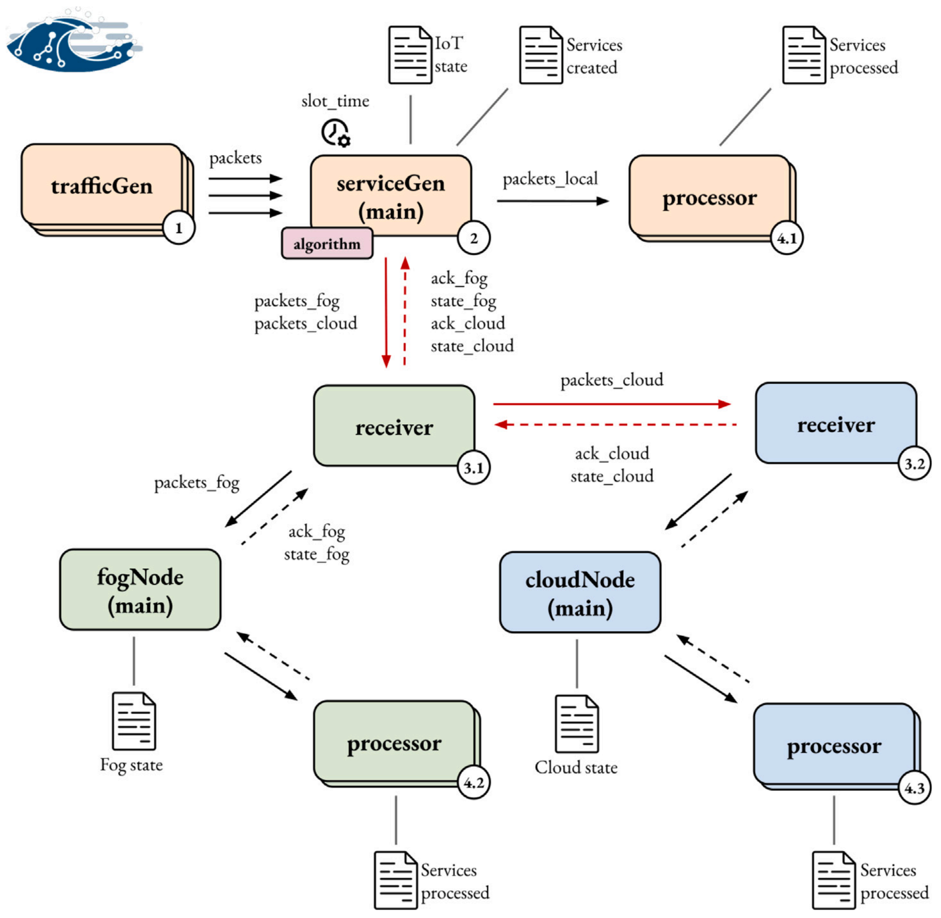


Fig. 3. Functional diagram of *ITSASO*.

Finally, in the fourth step, packets are placed in the processor buffer, awaiting execution. Each node instantiates processors with user-defined characteristics. Once a packet is processed, a log entry is generated with the corresponding timestamp. In the case of FS and CS, an acknowledgment message is sent back to the device once all packets belonging to a service have been processed. This allows the device to track service completion and, if needed, inform the offloading algorithm about the outcome. This information provides essential feedback for CT and DRL approaches to update delay information and compute the rewards, respectively, as will be discussed in Section 5. Additionally, the Fog and Cloud layers periodically communicate with the IoT devices to provide updates on the current system status. The information collected by the UE from the control messages includes periodic updates on buffer occupancy and the current computational capabilities of the Fog and Cloud layers. This information, combined with channel state data, enables offloading algorithms to make more informed decisions, optimizing overall system performance.

In addition to the FS and CS, logs are also generated in the UE, capturing key system metrics such as battery level, queue states, and service creation timestamps. *ITSASO* continuously tracks the battery level of each UE throughout the simulation, providing insights into energy consumption over time. The battery drain is modeled according to Section 3.3, which accounts for key factors such as processing and communication. This ensures that the energy usage closely mirrors realistic scenarios, enabling the evaluation of the impact of offloading decisions on device longevity.

In order to guarantee a scalable and lightweight platform, where multiple nodes can be deployed without overloading the host machine, all nodes have been containerized using Docker. The process of containerization enables users to rapidly deploy a multitude of bespoke containers within a single host. Each container represents an isolated software unit that packages code and its dependencies, thereby enabling it to run irrespective of the underlying host. In *ITSASO*, each of the containers can implement one of the aforementioned roles: UE, FS or CS, which are built from different customized Docker images. Docker executes container images that are lightweight and standalone, which are executable packages of software that include everything needed to run an application. This includes code, runtime, system tools, system libraries, and settings. The functionalities of every node have been developed in Python. Furthermore, the channel delay is emulated through the Linux Traffic Control subsystem, which applies to the red arrows in Fig. 3, representing the communication links where latency is introduced.

In summary, *ITSASO* allows different edge–cloud scenarios simulation and the evaluation of different scheduling algorithms. Furthermore, and exploiting the modular design of our architecture, moving the Docker containers that implement the Fog/Cloud functionality to real deployments could be done rather straightforwardly.

5. Proposed algorithms

This section delves into the development and implementation of the algorithms to address the computation offloading problem. We present a detailed description of the optimization problem in each case, providing the theoretical formulation and the description of the procedural steps that are needed to resolve the optimization problem by each algorithm.

5.1. Control Theory - Lyapunov

Let us define N as the number of available processing points, $a_n(t)$ as the number of services to be processed in the nth processing point at time t, and $b_n(t)$ the number of services actually processed at this point in that slot. The queue dynamics are given by Eq. (10), where $Q_n(t)$ is the queue backlog of the nth processor queue at time t.

$$Q(t+1) = \max[Q(t) - b(t), 0] + a(t) \tag{10}$$

Let $\alpha(t)$ represent the decision variable, with the following options: process the service locally, in the Fog Layer, or in the Cloud Layer. Here, $\alpha(t)$ is an array of N elements, where one element equals the service size, while the remaining elements are set to zero. Additionally, we introduce $\omega(t)$, which denotes general random parameters, such as the channel state or the service sizes.

One of the objectives of the proposed scheme is to account for the specific latency requirements of each service during decision-making. Accordingly, we introduce $D_m(t)$ as the latency requirement for the mth application. We also consider the penalties associated with service processing. We use $e_n(t)$ to represent the energy cost of using the nth processing alternative at time slot t. The penalty, denoted by $P(t)$, follows Eq. (11). In the case of local processing, the battery drained by the processor is the only factor considered. On the other hand, the offloading decision (fog or cloud) increases battery consumption due to the transmission. The penalty in both cases is proportional to the amount of traffic forwarded, i.e. the service size. Instead of using instantaneous metrics, we consider the time-average expectation $\overline{P}$, which is defined in Eq. (12).

$$P(t) = \hat{P}\left(\sum_{n=1}^{N} e_n(t) \cdot \alpha_n(t)\right) \tag{11}$$

$$\overline{P} = \lim_{T\to\infty} \frac{1}{T} \sum_{t=0}^{T} \mathbb{E}\{P(t)\} \tag{12}$$

The energy consumption, $e_n(t)$, and the time to complete a service, $d_m(t)$, can be defined as generic functions of the scheduling decisions and additional random events, as follows:

$$e_n(t) = \hat{e}(\alpha(t), \omega(t)) \tag{13}$$

$$d_m(t) = \hat{d}(\alpha(t), \omega(t)) \tag{14}$$

Altogether, we want a control policy that minimizes Problem 1:

Problem 1.

$$\min_{\alpha(t)} \quad \overline{P} \tag{15}$$

$$\textbf{s.t.} \quad \overline{d}_m \leq D_m \quad \forall m \in \{1, \ldots, M\} \tag{16}$$

$$\alpha(t) \in \mathcal{A} \quad \forall t \tag{17}$$

where $\overline{d}_m(t)$ should be equal to or lower than the specific delay requirement defined for the application, $D_m(t)$.

The stochastic optimization framework developed in [34] can be applied to transform this inequality into a set of virtual queues, which are conceptually analogous to processor queues. These virtual latency queues, denoted as $G_m(t)$, are updated according to Eq. (18):

$$G_m(t+1) = \max\{G_m(t) + (d_m(t) - D_m(t)), 0\} \tag{18}$$

Virtual queues are a strong method to ensure latency requirements in average. We define $\Theta(t)$ as the set of queues (virtual and physical). Eq. (19) defines the Lyapunov's function, $L(\Theta(t))$, and the *drift* $\Delta(\Theta(t))$.

$$\Delta(\Theta(t)) = \mathbb{E}\{L(\Theta(t+1)) - L(\Theta(t)) | \Theta(t)\} \tag{19}$$

To tackle Problem 1 we apply the *drift-plus-penalty* algorithm. At each slot t, the state of the queues is observed, and a decision, $\alpha(t)$, is taken in order to solve Problem 2. This is an Integer Linear Problem (ILP) problem, which can be solved using existing tools. The complete process is depicted in Algorithm 1.

Problem 2.

$$\min_{\alpha(t)} \quad V \cdot P(t) + \sum_{i=1}^{M} Q_i(t)[a_i(t) - b_i(t)] + \sum_{j=1}^{M} G_j(t)(d_j - D_j(t)) \tag{20}$$

$$\textbf{s.t.} \quad \alpha(t) \in \mathcal{A} \tag{21}$$

Note that in the first term of Problem 2 the cost function includes a parameter V that multiplies the penalty. V is a positive weighting factor that regulates the trade-off between the *drift* (second term) and the penalty. In other words, decreasing the value of V strengthens the need to meet latency requirements. Conversely, increasing the value of V shifts the focus towards minimizing battery consumption.

Algorithm 1 Lyapunov Process

```
1: Set $V$.
2: for each step $t = 0$ to T (until terminal state) do
3:     Observe $b(t) = [b_1(t), ..., b_M(t)], Q(t) = [Q_1(t), ..., Q_M(t)]$ and $\omega(t)$.
4:     Select decision $\alpha(t) \in \mathcal{A}$ to minimize Problem 2.
5:     Update $G(t) = [G_1(t), ..., G_M(t)]$ according to Eq. (18).
6: end for
```

5.2. Deep Reinforcement Learning

For the DRL algorithms that enable IoT nodes to learn optimal strategies through interaction with their environment, we model the system as a Markov Decision Process (MDP), which is used to represent the decision-making process of a dynamic system where the environment may evolve randomly, leading to different decisions over time. At each stage, a thorough analysis of the current state s_i is conducted by the UEs, each of which utilizes an instance of the implemented decision algorithm to determine the appropriate action for its observation. Consequently, each agent aims to learn how to optimize their decisions in a shared environment, where such actions can influence the state of the environment and, subsequently, the decisions and rewards of other UEs.

State space. The defined state represents the current information about the environment. In this scenario, agents located in the UEs have a complete view of the environment, excluding the status of other UEs, which is not relevant when taking a decision. Thus, they can access all the information that defines the environment's status, including their own processing queue status, the processing queue of the FS, the delay between the IoT node and the FS, and the number of packets and delay requirements of the service to be processed (σ_p and D_σ, respectively). The CS is supposed to have huge computation capabilities and no waiting queues. Therefore, the state is represented as the following 5-tuple vector space (Eq. (22)).

$$s_{n,t} = \{Q_n(t), Q_F(t), d_{nf}(t), \sigma_p, D_\sigma(t)\} \tag{22}$$

Action space. The action represents how an incoming task shall be executed. This can be expressed as $alpha_i \in \{0, 1, 2\}$, with $\alpha_i = 0$ meaning local execution, $\alpha_i = 1$ meaning the offload to the fog server F and $\alpha_i = 2$ meaning the offload to the cloud server C.

Reward function. The reward function utilized for this problem is calculated as the energy consumption value of the corresponding UE, as long as the delay requirements of the services are met. In case these requirements are not met, the reward is set to a punishment value η, which is set to a value of -10. This is defined in Eq. (23).

$$R(t) = \begin{cases} -e_{\sigma,i} & \text{if } D_\sigma > d_\sigma \text{ and } \alpha = 0 \\ -e_{\sigma,f} & \text{if } D_\sigma > d_\sigma \text{ and } \alpha = 1 \\ -e_{\sigma,c} & \text{if } D_\sigma > d_\sigma \text{ and } \alpha = 2 \\ \eta & \text{if } D_\sigma \le d_\sigma \end{cases} \tag{23}$$

The final optimization problem is to maximize the total reward (Eq. (23)), this is, reducing the energy consumption while ensuring the service delay requirements are met, which boils down to the following problem:

Problem 3.

$$\max_{\{a_t\}} \quad \sum_{t=0}^{T-1} R(t) \tag{24}$$

$$\textbf{s.t.} \quad a_t \in \{0, 1, 2\}, \quad \forall t \in \{0, \dots, T-1\} \tag{25}$$

where the action constraint states that each task can only be one of these options: executed locally, offloaded to the FS or offloaded to the CS.

The general DRL process followed to solve this optimization problem is shown in Algorithm 2.

Algorithm 2 DRL Process

```
1: Initialize the environment $\mathcal{E}$
2: Initialize the policy $\pi_\theta$ and parameters $\theta$
3: Reset environment $\mathcal{E}$ and observe initial state $s_0$
4: for each step $t = 0$ to $T$ (until terminal state) do
5:     Select action $a$ using policy $\pi_\theta(s_t)$
6:     Execute action $a$ in $\mathcal{E}$ and observe reward $r_t$ and next state $s_{t+1}$
7:     Update policy $\pi_\theta$ using the learning algorithm
8:     if termination condition is met then
9:         Break loop
10:    end if
11: end for
12: Return optimized policy $\pi_\theta$
13: Save DRL algorithms
14: Close the environment $\mathcal{E}$
```

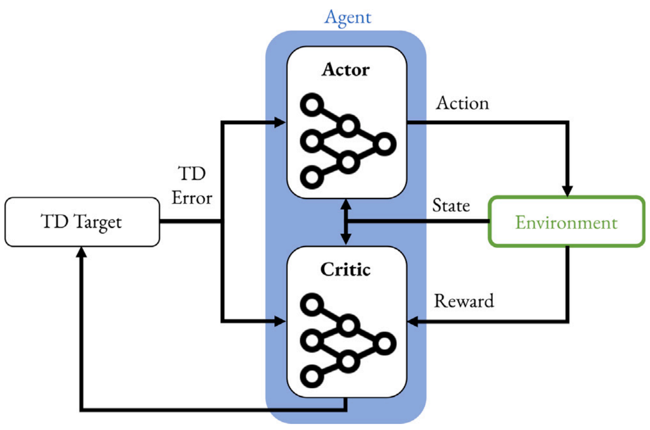


Fig. 4. Actor critic model.

Every DRL algorithm included in this work has been implemented using Tensorflow[2] 2.14.0, along with Keras[3] 2.14.0. The environment has been modeled using Gymnasium[4] library version 0.29.1, which is a maintained fork of OpenAI's deprecated Gym library.

5.2.1. Actor-Critic

We first propose the algorithm shown in Fig. 4, an AC model that integrates policy gradient and value function methods. This model consists of two distinct Deep Neural Network (DNN) architectures: the actor-network and the critic-network.

The actor, which determines the actions to take based on the environment's current state, consists of two hidden layers, each with 128 neurons. The output layer employs the softmax function, which compresses a K-dimensional vector into values within the range $[0, 1]$. The learning rate for the actor is configured to 10^{-4} in order to balance the trade-off between convergence speed and stability.

The critic network, responsible for estimating the value function or expected reward for specific actions in given states, also comprises two hidden layers with 128 neurons each, using the same Hyperbolic Tangent (TanH) activation function. In the output layer, a linear activation function is applied. The learning rate for the critic network is set to 10^{-4}.

This AC framework leverages the Temporal Difference (TD) error, which measures the difference between the predicted and the actual observed rewards, to optimize both the actor and critic networks. The Adam optimizer is used to update the parameters of both networks based on this TD error. Lastly, the discount factor employed in this algorithm is set to 0.9, so future rewards are considered but do not have more influence than the next one.

This algorithm was presented and validated in [35].

[2] Tensorflow: https://www.tensorflow.org/.
[3] Keras: https://keras.io/.
[4] Gymnasium: https://gymnasium.farama.org/index.html.

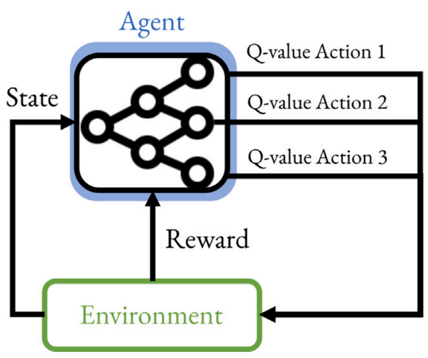


Fig. 5. DQN model.

5.2.2. *Deep Q-Network*

Fig. 5 shows the implementation of the DQN algorithm in this work. The DQN algorithm is a value-based reinforcement learning method that extends traditional Q-learning by utilizing a DNN to approximate the Q-value function, enabling it to handle larger state spaces.

The core component of the DQN model is a deep neural network used to approximate the Q-value function. This network takes the current state as input, and outputs Q-values for all possible actions in that state. For this implementation, the neural network comprises two hidden layers, each containing 128 neurons, which utilize the TanH activation function.

The output layer of the network employs a linear activation function, producing Q-values for each action. These Q-values are used to determine the action with the highest expected reward in the current state, following an epsilon-greedy policy. This policy balances exploration and exploitation by selecting a random action with a probability of ϵ, while choosing the action with the highest Q-value otherwise. This model employs the Adam optimizer with a learning rate of 10^{-5}.

To train the network, the DQN algorithm also uses a loss function based on the TD error.

5.2.3. *Proximal Policy Optimization*

PPO is designed to improve the stability and performance of earlier policy gradient methods based on the AC architecture. This implementation contains two main steps and is represented in Fig. 6.

First, the agent collects data by interacting with the environment using its current policy and generating a batch of experiences. When enough data has been collected, the PPO algorithm performs the policy update, where transitions are stored in memory and the batch is updated. Then, during the update, actor and critic networks are trained using gradient descent. The actor loss is based on PPO's clipped surrogate objective, which prevents excessively large policy updates that could destabilize learning. The critic, on the other hand, is trained using Mean Squared Error (MSE) loss between predicted and target values.

Both actor and critic networks are configured as in the AC algorithm explained in Section 5.2.1, but the discount factor value is set to 0.9 and the learning rate to $5 \cdot 10^{-4}$, respectively. The epsilon clipping value is set to 0.2, the batch size is 64 and the model is updated every 8 steps.

6. Results

This section presents the simulation results of the proposed approaches across various scenarios using *ITSASO*. The simulations have been carried out in a server with a 32 core 4th Gen. Intel Core i7 CPU at 2.00 GHz and 32 GB of RAM, running Ubuntu 20.04.6 LTS as operating system.

Solutions based on CT (Lyapunov) and DRL (DQN, AC and PPO), which have been presented in Section 5, are compared to traditional scheduling methods like Round-Robin (RR), and fixed baselines, i.e. running everything locally, offloading everything to the FS or offloading everything to the CS. Finally, we also include a random scheduling baseline for comparison.

Additionally, we analyze system behavior under 2 different scenarios, one with constant traffic and fixed delay requirements, and the other one with a Poisson distribution and variable delay service requirements. The purpose of this analysis is to evaluate different scheduling methods in both static and dynamic environments, allowing us to determine the conditions under which each scheduling method performs better considering that the optimization goal is to minimize energy consumption while meeting tasks' latency requirements. Besides, and in order to ensure a fair comparison between the two scenarios, the same figures have been plotted for each scenario, using identical metrics

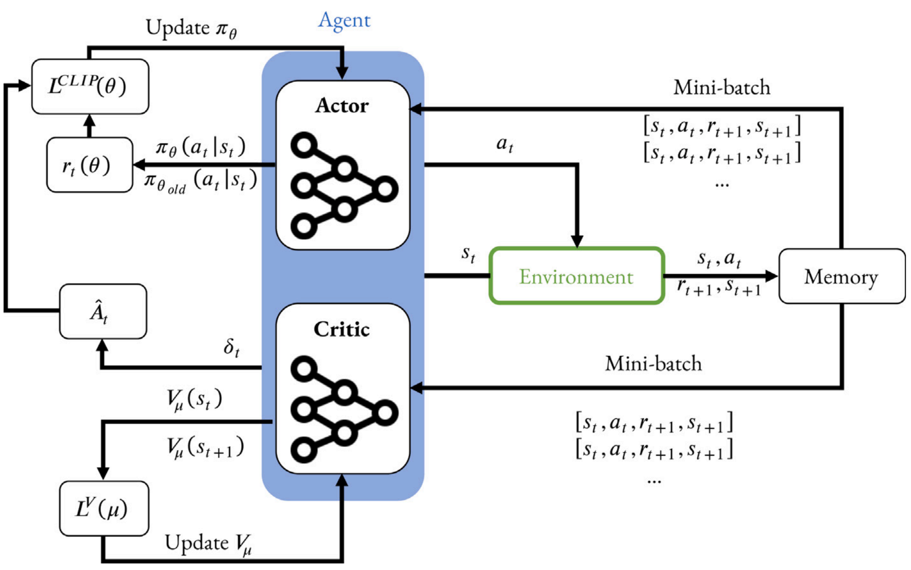


Fig. 6. PPO model.

Table 3
Simulation parameters.

	Parameter	Symbol	Value	
			Scenario 1	Scenario 2
UE	Number of UEs	N	15	15
	Computation capacity	F_n	$1.5 \cdot 10^3$ p/s	$1.5 \cdot 10^3$ p/s
	Energy coefficient	κ_n	10^{-23}	10^{-23}
	Maximum battery	B_n	10^5 J	10^5 J
FS	Number of servers	F	1	1
	Computation capacity	F_F	$1.5 \cdot 10^4$ p/s	$1.5 \cdot 10^4$ p/s
CS	Number of servers	C	1	1
	Computation capacity	F_C	$2 \cdot 10^9$ p/s	$2 \cdot 10^9$ p/s
Network	Delay IoT-Fog	d_{nf}	[50–125] ms	[50–125] ms
	Delay Fog-Cloud	d_{fc}	650 ms	650 ms
Services	Traffic rate	–	5	5
	Distribution	–	Constant	Poisson
	Delay requirements	d_{req}	750 ms	250–1000 ms
	Packet size	p	200 bytes	200 bytes
Other	Fail punishment	η	−10	−10
	Slots	–	1000	1000
	Slot time	–	1 s	1 s

6.1. Experimental setup

As mentioned above, for the experimental simulation two distinct scenarios have been considered to ensure a complete analysis of different situations. The first one focuses on a stable environment, while the second setup introduces dynamic changes, specifically in terms of traffic distribution and service delay requirements.

The configuration of each scenario is presented in Table 3.

UE-related parameters comprise the number of UEs, their computing capacity in packets/second, their available battery at the start of the simulation, and the energy coefficient used to calculate the energy spent when running services, following the model presented in Section 3.

Regarding the FC parameters, the number of servers and their computation capacity are defined, which is the same information configured in CC layer.

Network-related parameters include the delay between IoT devices and the FS, and the delay between the FS and the CS.

The traffic rate, distribution, delay requirements, and packet size are also configured for the different applications that generate computing services. If a services is not completed within its delay requirement, we consider it as failed.

Finally, we also consider additional parameters that define the overall simulation time, including the number of slots and the slot time (for these simulations, services are generated each second), as well as the penalty parameter η used for DRL algorithms.

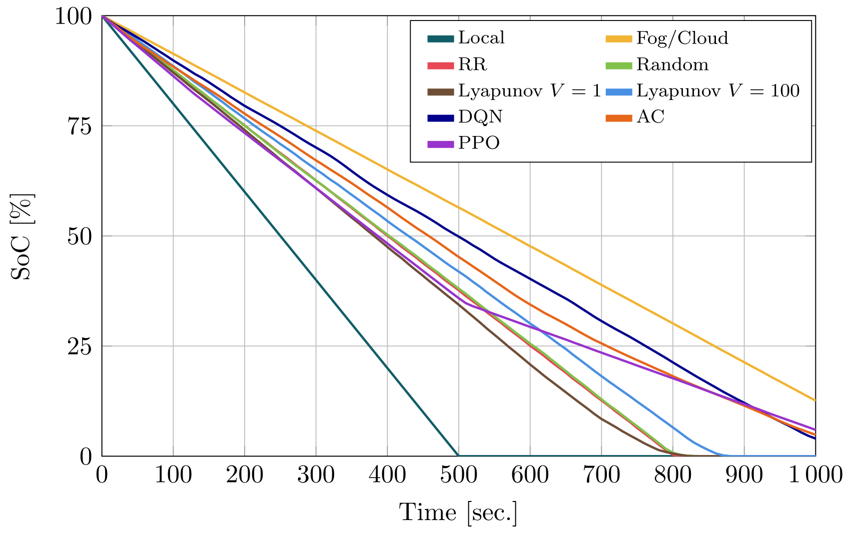


Fig. 7. SoC evolution over time for every algorithm in Scenario 1.

6.2. *Scenario 1: Fixed delay requirements*

In this first scenario, as can be seen in Table 3, we explore a system where services are generated at a constant rate, and each of them with the same delay requirement, so traffic patterns and delay constraints remain consistent. Thus, we ensure that the performance of scheduling solutions is analyzed in a controlled environment.

First, Fig. 7 illustrates the evolution of the mean SoC over time for each analyzed scheduling strategy, this is, the average of every IoT node's remaining battery percentage, which is reduced through the execution or offloading of incoming services according to the modeled behavior explained in Section 3.3. Therefore, SoC is calculated as shown in Eq. (26):

$$\text{SoC [\%]} = \frac{b_n}{B_n} \cdot 100 \tag{26}$$

where b_n is the remaining battery of the nth UE and B_n corresponds to its total battery capacity.

The x-axis represents the time, indicating the order in which services are processed; and the y-axis shows the average SoC of all IoT nodes. For instance, having 15 nodes, the plotted value represents the mean battery level across them. Running locally indicates the highest energy consumption and offloading everything strategies (either to the FS or to the CS) correspond to the least energy consumption, with their battery levels decreasing at a significantly slower pace and both being identical in their energy usage, as explained in Section 3. DRL-based strategies consume less energy than Lyapunov-based strategies, where the IoT devices SoC turns 0 between time-steps 800–850.

Considering that energy consumption is lower when the selected action is to offload, it may appear that sending services to the FS generally provides an ideal balance: lower energy consumption than local execution, while low latency values due to the fog's proximity to the UE. Nevertheless, when every service is sent to the FS, the FS gets overflowed, as the queue becomes larger over time. Regarding the CS, it does not get overflowed due to its bigger computation capabilities, but the latency experienced to reach it might become too high for certain services. Therefore, there is a need for dynamic scheduling algorithms, as static schedulers do not always yield good results.

Fig. 8 provides an overview of the number and kind of fails for each approach. As can be seen, solutions based on Lyapunov (with both parameters $V = 1$ and $V = 100$) do not show any fail while battery is on, so we can conclude this strategy is able to make a decision that meets the latency requirements of incoming services. The useful lifetime of devices is higher when the V parameter is set to 100, meeting the delay requirement at the same time. Among DRL-based approaches, DQN is the best one in this scenario, minimizing the fail rate and enhancing the devices' lifetime.

Regarding the decisions taken by the scheduling algorithms, Fig. 9 shows that, using Lyapunov, almost all services are processed locally (47.2% with $V = 1$ and 33.9% with $V = 100$) or in the FS (52.8% with $V = 1$ and 66.1% with $V = 100$), always discarding CS option. In addition, it can also be seen that changing the parameter V from 1 to 100, not only improves the battery life of the devices (as seen in 7), but it also reduces the number of failed tasks to 0 while the IoT devices are alive. DRL-based algorithms, especially DQN, can quickly identify that the worst option for this scenario is sending services to the cloud, due to the high latencies this implies. However, due to the exploration vs. exploitation dilemma, DRL approaches occasionally make suboptimal decisions trying to find better alternatives. Thus, while exploration can lead to temporary setbacks, it ultimately contributes to the robustness and adaptability of the learning process. In any case, every DRL algorithm sends most services to the FS, being a 87.5% of the services with DQN algorithm, 66.5% using AC and 57.6% using PPO.

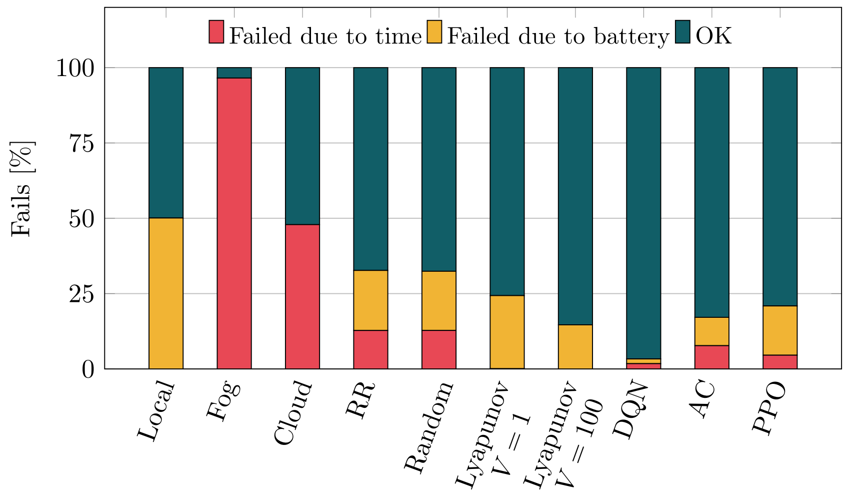


Fig. 8. Number and kind of Fails (%) with and without battery in Scenario 1.

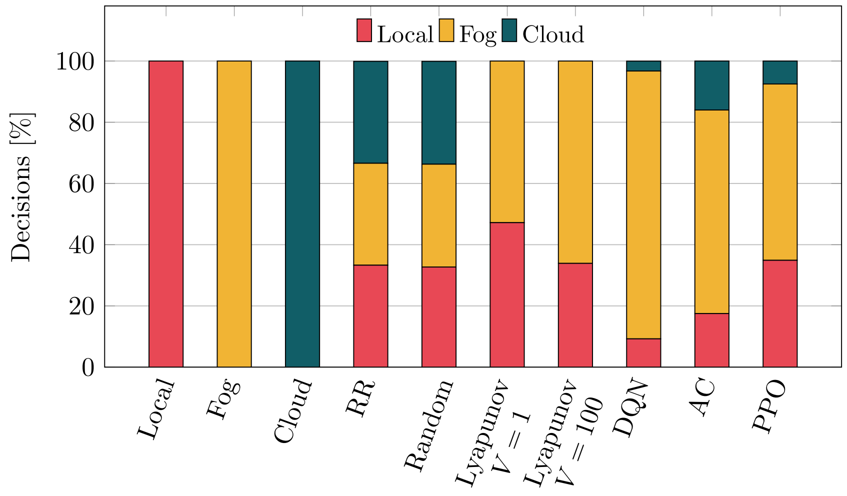


Fig. 9. Distribution of decisions (%) for every algorithm in Scenario 1.

Finally, Fig. 10 illustrates the relationship between battery life (the time IoT nodes run out of battery) and the percentage of success for every algorithm. As can be observed, four distinct groups are distinguished:

- Static (green). This group includes running every service locally at the IoT nodes, offloading every service to the FS, and offloading every service to the CS.
- Non-context aware (brown). This group include RR and Random algorithms.
- CT-based solutions (red). This group includes different configurations of the Lyapunov algorithm.
- DRL-based solutions (blue). This group includes every implemented DRL algorithm, being DQN, AC and PPO.

Any results rightwards of the dotted line means that the battery is still alive after the simulation, which, in this case, is only achieved by sending every service to the FS or the CS. It can also be seen that offloading every service to the FS or the CS gives the lowest number of successful services during the life of the IoT devices (3.5% for FS and 52.1% for CS). Although running everything locally yields good results in terms of fulfilling latency requirements (no failed services while battery is up), it is the approach with the highest energy consumption, running out of battery at step 500. DQN is the best option among the DRL-based approaches, having a mean battery-life of 961 services and more than a 98% of success. AC and PPO show a success probability of 92.3% and 95.5%, respectively, being PPO the solution with the highest energy consumption after “Local” approach. Lastly, Lyapunov provides the best results in terms of successful service completion, yielding no failed services when V is set to 100 and more than a 99% of success when set to 1.

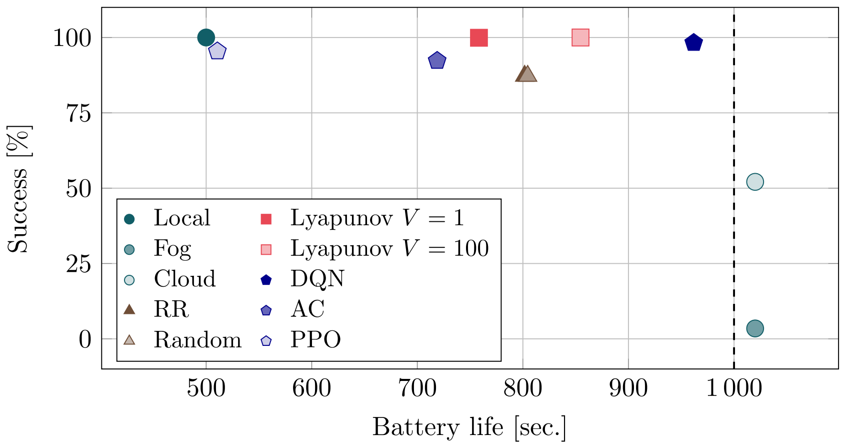


Fig. 10. Battery vs. Success (%) in Scenario 1.

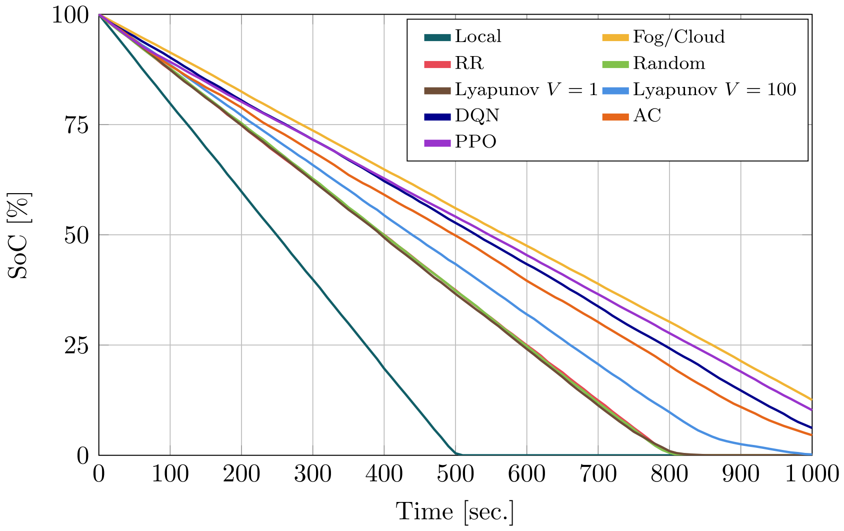


Fig. 11. SoC evolution over time for every algorithm in Scenario 2.

6.3. *Scenario 2: Variable delay requirements*

The second scenario, as seen in Table 3, is defined by a Poisson distribution of service arrivals, characterized by a mean traffic rate of 5 services per time slot. Thus, unlike the first scenario, the service distribution is not continuous and can vary over time. This scenario also introduces dynamic delay requirements, where each service is characterized by varying latency constraints, so the adaptability and efficiency of different scheduling algorithms can be evaluated under unpredictable conditions, as both the arrival rate and delay requirements fluctuate.

Fig. 11 shows the SoC of every IoT node over time. There can be seen that performing local execution of services implies a higher energy consumption than offloading, and that static scheduling algorithms are the bounds between which the rest of the algorithms can be found in terms of energy consumption. DRL-based strategies consume less energy than Lyapunov-based strategies, where the IoT devices SoC turns 0 around timestep 800.

In this scenario, Lyapunov-based methods tend to offer suboptimal solutions compared to DRL approaches, because they struggle to adapt to the dynamic nature of delay requirements for each service. Lyapunov optimization relies on static parameters, making it less effective when delay constraints frequently change. On the other hand, DRL approaches offer better results in such dynamic settings, by continuously learning and adapting their policies based on real-time feedback.

Fig. 12 shows the number and kind of fails for each approach. It reveals that scheduling algorithms struggle to find the optimal solution in this scenario, due to its dynamicity. Some DRL algorithms seem to find better results in this case, as they can adapt to changing environments by trial and error. As can be seen, solutions based on Lyapunov show more fails than in the static scenario. Among DRL-based approaches, PPO is the one showing the best performance in this scenario, enhancing the devices' lifetime and showing the second lowest fail rate while battery is on, being similar to Lyapunov with V set to 1.

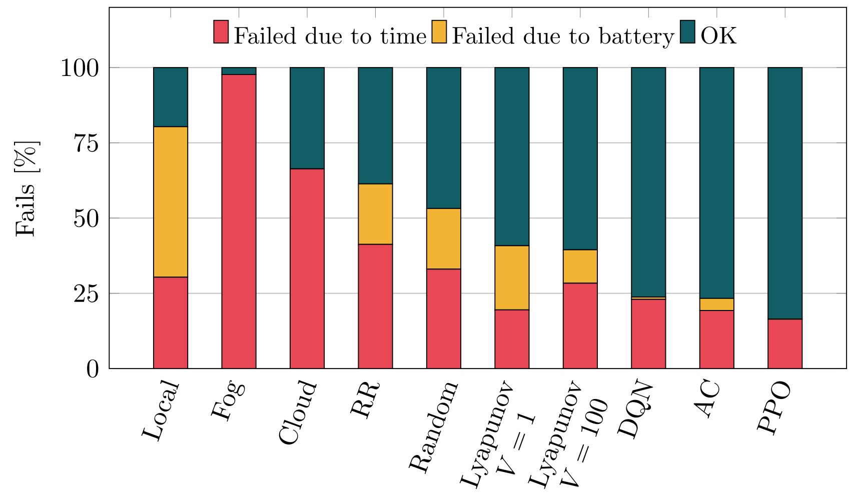


Fig. 12. Number and kind of Fails (%) with and without battery in Scenario 2.

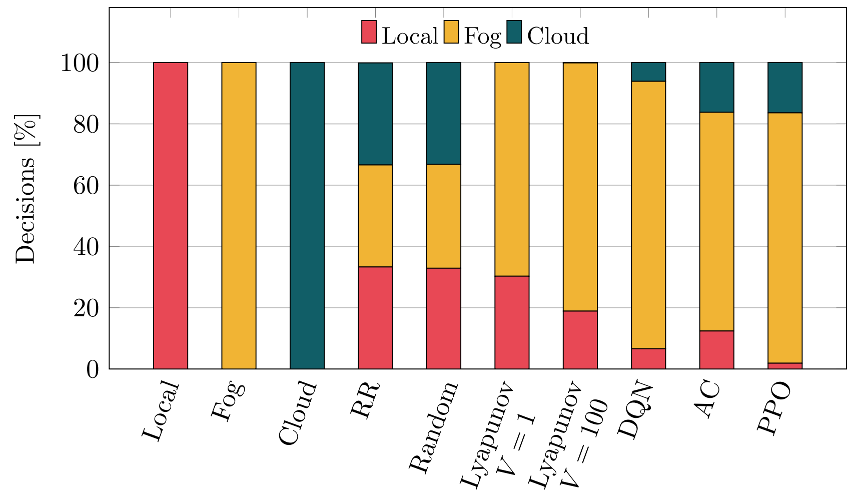


Fig. 13. Distribution of decisions (%) for every algorithm in Scenario 2.

Fig. 13 provides an overview of the decision-making behavior, so the performance variability of the evaluated scheduling algorithms can be analyzed. As in the static scenario, the Lyapunov based strategy tends to process services locally (30.3% with $V = 1$ and 18.9% with $V = 100$) or in the FS (69.7% with $V = 1$ and 81.0% with $V = 100$), discarding the CS alternative almost in all cases (0.1% of the services are sent to the CS when $V = 100$). DRL-based algorithms also send most services to the FS, being a 87.3% of the services with DQN algorithm, 71.4% using AC and 81.7% using PPO.

Finally, Fig. 14 presents the comparative analysis of every strategy in terms of battery life and fail percentage. Local executions result in the lowest battery life (500.93) and a moderate success rate (69.65%). Conversely, as we saw in the static scenario, offloading (either to fog or cloud) does not deplete the battery, but induces the lowest success rates, being 2.3% and 33.64%, respectively. The battery life in Lyapunov-based method is higher with a V value of 100 (882 vs. 778 when $V = 1$). Besides, the success rate is 80% when $V = 1$ and 72% when $V = 100$. RR and Random strategies achieve similar battery lifetime (800) but they suffer from low success rates (59% and 67%, respectively). Overall, PPO appears to yield the best balance between battery lifetime and service success rate, as the IoT devices do not run out of battery and the success rate is around 84%. AC presents a lifetime of 900.17 services and a success rate of 80.72%, being slightly worse than PPO's, while DQN's lifetime is 962 the success rate 77%.

7. Conclusion

In this research, we have provided an evaluation of different task computation scheduling strategies, paving the way for the development of more robust and efficient service management systems. We have demonstrated that both DRL and CT techniques offer significant advantages over traditional methods for workload scheduling. We have used *ITSASO*, a simulation platform that

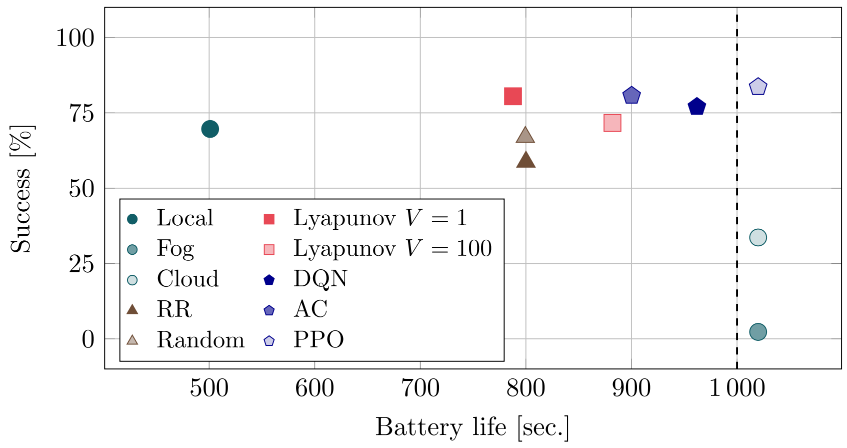


Fig. 14. Battery vs. Success (%) in Scenario 2.

allows us to test different decentralized scheduling algorithms. This platform has been made available to the research community in a public repository.

The proposed approaches achieve better performance in terms of energy consumption, while ensuring as well service success, in terms of their experienced latency. More precisely, CT techniques, in particular, Lyapunov optimization, provide stable and optimized solutions that offer good results for the considered scenarios (static and dynamic), specially for the former, offering precision and stability in scenarios where the system's behavior is more consistent.

On the other hand, DRL approaches yield better behavior in dynamic environments, showcasing their ability to adapt and learn from fluctuating conditions, which could be given in unstable real-world deployments that suffer different uncertainties, such as variable wireless connections.

Future research will focus on integrating our environment with network simulation frameworks (in particular NS-3), to enable a better modeling of network dynamics, like communication delays, and so test the system's behavior under more realistic conditions. Moreover, we plan to investigate the impact of user mobility on service scheduling, considering the challenges caused by fluctuating network connectivity and changing location.

Additionally, we aim to extend the analysis by comparing decentralized or distributed solutions with centralized approaches, focusing on the trade-offs between scalability and performance. Furthermore, we will investigate the application of alternative optimization algorithms, such as metaheuristics, to evaluate their effectiveness in addressing the complexities of service distribution.

Notation

See Table 4.

List of abbreviations

A2C Advantage Actor-Critic
AC Actor-Critic
AI Artificial Intelligence
AP Access Point
BS Base Station
CC Cloud Computing
CS Cloud Server
CT Control Theory
DDPG Deep Deterministic Policy Gradient
DNN Deep Neural Network
DQN Deep Q-Network
DRL Deep Reinforcement Learning
EC Edge Computing
GA Genetic Algorithm
FC Fog Computing
FS Fog Server
IIoT Industrial IoT
ILP Integer Linear Problem

IoT Internet-of-Things
KPI Key Performance Indicator
LSTM Long short-term memory
MA Multi-Agent
MDP Markov Decision Process
MEC Multi-access Edge Computing
MSE Mean Squared Error
ML Machine Learning
MO Mathematical Optimization
PPO Proximal Policy Optimization
PSO Particle Swarm Optimization
QoS Quality-of-Service
RL Reinforcement Learning
RR Round-Robin
SoC State of Charge
TanH Hyperbolic Tangent
TD Temporal Difference
UE User Equipment
V2X Vehicle-to-everything
WPT Wireless Power Technology

Table 4
Notation.

System parameters:	
m	Application
$D_m(t)$	Latency requirement for the mth application
D_σ	Latency requirement for the service σ
$e_n(t)$	Energy cost of using the nth processing alternative at time slot t
$d_\sigma(t)$	Time required to complete the service σ
C	Cloud server
F	Fog server
n	IoT device
N	Number of IoT devices
Q_n	Size of nth device's queue
Q_F	Size of FS queue
R	Transmission rate
σ	Service
s_σ	Size of the service
p_s	Size of a packet
σ_p	Number of packets that compose the service σ
κ	Energy coefficient determined by the chip structure of the UE
f	Frequency
ϕ	Required cycles
d_{nf}	Delay between nth IoT node and the Fog server F
CT parameters:	
$Q_n(t)$	Queue backlog of the nth processor queue at time t
$a_n(t)$	Number of services to be processed in the nth processing point at time t
$b_n(t)$	Number of services processed at this point in the time slot
$\omega(t)$	General random parameters, e.g. channel state or service sizes
P	Penalty
$\bar{\cdot}$	Time-average expectation
$\hat{\cdot}$	Arbitrary function that yields a variable $\cdot$
$G_m(t)$	Virtual latency queues
$\Theta(t)$	Set of virtual and physical queues
$L(\Theta(t))$	Lyapunov function
$\Delta(\Theta(t))$	Drift
V	Weighting factor that regulates the trade-off between the drift and the penalty
DRL parameters:	
$s_n(t)$	Observation/state of nth IoT device at time t
$\alpha_n(t)$	Action of nth IoT device at time t
η	Penalty value
$R(t)$	Reward at time t
$s_{n,t}$	Representation of nth device at state of at time t

CRediT authorship contribution statement

Gorka Nieto: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Data curation, Writing – original draft, Visualization. **Neco Villegas:** Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Data curation, Writing – original draft, Visualization. **Luis Diez:** Conceptualization, Supervision, Validation, Writing – review & editing. **Idoia de la Iglesia:** Conceptualization, Supervision, Validation, Writing – review & editing. **Unai Lopez-Novoa:** Conceptualization, Supervision, Validation, Writing – review & editing. **Cristina Perfecto:** Conceptualization, Supervision, Validation, Writing – review & editing. **Ramón Agüero:** Conceptualization, Supervision, Validation, Writing – review & editing.

Declaration of competing interest

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

Acknowledgments

This work was partially supported by the Spanish Ministerio de Asuntos Económicos y Transformación Digital and the European Union NextGenerationEU through the project FABRIC: Artificial Intelligence assisted beyond 5G virtualized distributed computing and connectivity (grant TSI-063000- 2021-27), funded by the Plan for Recovery, Transformation and Resilience. It has been also funded by the Cantabria Government through the project "Enabling Technologies for Digital Twins and their application in the chemical and communications sectors" (GDQuiC) of the TCNIC program (2023/TCN/002), and the Concepción Arenal program (grant UC-23-40).

Data availability

The link to the public repository containing the code is available in the manuscript.

References

[1] M. Soori, B. Arezoo, R. Dastres, Internet of things for smart factories in industry 4.0, a review, Internet Things Cyber-Phys. Syst. 3 (2023) 192–204, http://dx.doi.org/10.1016/J.IOTCPS.2023.04.006.

[2] J.S. Yalli, M.H. Hasan, A. Badawi, Internet of things (IOT): Origin, embedded technologies, smart applications and its growth in the last decade, IEEE Access (2024) http://dx.doi.org/10.1109/ACCESS.2024.3418995.

[3] F.C. Andriulo, M. Fiore, M. Mongiello, E. Traversa, V. Zizzo, Edge computing and cloud computing for internet of things: A review, Informatics 11 (2024) 71, http://dx.doi.org/10.3390/INFORMATICS11040071.

[4] M.A.E. Rasool, A. Kumar, A. Islam, M.N. Ahmed, Exploring task offloading in mobile edge computing environments: An in-depth review and prospective analysis, in: 7th IEEE International Conference on Advanced Technologies, Signal and Image Processing, ATSIP 2024, Institute of Electrical and Electronics Engineers Inc., 2024, pp. 622–627, http://dx.doi.org/10.1109/ATSIP62566.2024.10639042.

[5] Z. Zhao, J. Perazzone, G. Verma, S. Segarra, Congestion-aware distributed task offloading in wireless multi-hop networks using graph neural networks, in: ICASSP 2024 - 2024 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP, IEEE, 2024, pp. 8951–8955, http://dx.doi.org/10.1109/ICASSP48485.2024.10447302.

[6] A. Heidari, N.J. Navimipour, M.A.J. Jamali, S. Akbarpour, A hybrid approach for latency and battery lifetime optimization in IoT devices through offloading and CNN learning, Sustain. Comput.: Inform. Syst. 39 (2023) 100899, http://dx.doi.org/10.1016/J.SUSCOM.2023.100899.

[7] L. Lin, X. Liao, H. Jin, P. Li, Computation offloading toward edge computing, Proc. IEEE 107 (2019) 1584–1607, http://dx.doi.org/10.1109/JPROC.2019.2922285.

[8] Z. Nezami, K. Zamanifar, K. Djemame, E. Pournaras, Decentralized edge-to-cloud load balancing: Service placement for the internet of things, IEEE Access 9 (2021) 64983–65000, http://dx.doi.org/10.1109/ACCESS.2021.3074962.

[9] M. Spanopoulos-Karalexidis, C.K.F. Papadopoulos, K.M. Giannoutakis, G.A. Gravvanis, D. Tzovaras, M. Bendechache, S. Svorobej, P.T. Endo, T. Lynn, Simulating across the cloud-to-edge continuum, in: T. Lynn, J.G. Mooney, J. Domaschka, K.A. Ellis (Eds.), Managing Distributed Cloud Applications and Infrastructure, Palgrave Macmillan, Cham, 2020, pp. 93–115, http://dx.doi.org/10.1007/978-3-030-39863-7_5.

[10] F. Saeik, M. Avgeris, D. Spatharakis, N. Santi, D. Dechouniotis, J. Violos, A. Leivadeas, N. Athanasopoulos, N. Mitton, S. Papavassiliou, Task offloading in Edge and Cloud Computing: A survey on mathematical, artificial intelligence and control theory solutions, Comput. Netw. 195 (2021) 108177, http://dx.doi.org/10.1016/J.COMNET.2021.108177.

[11] Z. Luo, X. Dai, Reinforcement learning-based computation offloading in edge computing: Principles, methods, challenges, Alex. Eng. J. 108 (2024) 89–107, http://dx.doi.org/10.1016/j.aej.2024.07.049.

[12] P. Peng, W. Lin, W. Wu, H. Zhang, S. Peng, Q. Wu, K. Li, A survey on computation offloading in edge systems: From the perspective of deep reinforcement learning approaches, Comput. Sci. Rev. 53 (2024) http://dx.doi.org/10.1016/j.cosrev.2024.100656.

[13] K. Sadatdiynov, L. Cui, L. Zhang, J.Z. Huang, S. Salloum, M.S. Mahmud, A review of optimization methods for computation offloading in edge computing networks, Digit. Commun. Netw. 9 (2023) 450–461, http://dx.doi.org/10.1016/J.DCAN.2022.03.003.

[14] T. Allaoui, K. Gasmi, T. Ezzedine, Reinforcement learning based task offloading of IoT applications in fog computing: algorithms and optimization techniques, Clust. Comput. 27 (2024) 10299–10324, http://dx.doi.org/10.1007/s10586-024-04518-z.

[15] B. Xie, H. Cui, Deep reinforcement learning-based dynamical task offloading for mobile edge computing, J. Supercomput. 81 (2025) http://dx.doi.org/10.1007/s11227-024-06603-x.

[16] M. Kim, J. Jang, Y. Choi, H.J. Yang, Distributed task offloading and resource allocation for latency minimization in mobile edge computing networks, IEEE Trans. Mob. Comput. (2024) 1–17, http://dx.doi.org/10.1109/tmc.2024.3458185.

[17] Z. Chen, X. Wang, Decentralized computation offloading for multi-user mobile edge computing: A deep reinforcement learning approach, EURASIP J. Wirel. Commun. Netw. 2020 (1) (2020) 188, http://dx.doi.org/10.1186/s13638-020-01801-6.

[18] K. Zhu, S. Li, X. Zhang, J. Wang, C. Xie, F. Wu, R. Xie, An energy-efficient dynamic offloading algorithm for edge computing based on deep reinforcement learning, IEEE Access (2024) http://dx.doi.org/10.1109/ACCESS.2024.3452190.

[19] Y. Long, Q. Zeng, Y. Zhuang, Q. Pan, H. Xiao, An innovative task offloading algorithm based on deep reinforcement learning in computation resource network, in: 20th International Wireless Communications and Mobile Computing Conference, IWCMC 2024, Institute of Electrical and Electronics Engineers Inc., 2024, pp. 754–760, http://dx.doi.org/10.1109/IWCMC61514.2024.10592363.

[20] M. Goudarzi, Z. Movahedi, M. Nazari, Mobile cloud computing: A multisite computation offloading, in: 2016 8th International Symposium on Telecommunications, IST 2016, Institute of Electrical and Electronics Engineers Inc., 2017, pp. 660–665, http://dx.doi.org/10.1109/ISTEL.2016.7881904.

[21] B. He, H. Li, T. Chen, DRL-based computing offloading approach for large-scale heterogeneous tasks in mobile edge computing, Concurr. Comput.: Pr. Exp. 36 (2024) http://dx.doi.org/10.1002/cpe.8156.

[22] J. Wen, Distributed reinforcement learning-based optimization of resource scheduling for telematics, Comput. Electr. Eng. 118 (2024) http://dx.doi.org/10.1016/j.compeleceng.2024.109464.

[23] B. Dai, Y. Qiu, W. Feng, Scalable computation offloading for industrial IoTs via distributed deep reinforcement learning, in: Proceedings of the 2024 27th International Conference on Computer Supported Cooperative Work in Design, CSCWD 2024, Institute of Electrical and Electronics Engineers Inc., 2024, pp. 1681–1686, http://dx.doi.org/10.1109/CSCWD61410.2024.10580311.

[24] X. Liu, A. Chen, K. Zheng, K. Chi, B. Yang, T. Taleb, Distributed computation offloading for energy provision minimization in WP-MEC networks with multiple HAPs, IEEE Trans. Mob. Comput. 24 (4) (2025) 2673–2689, http://dx.doi.org/10.1109/TMC.2024.3502004.

[25] R. Mahmud, S. Pallewatta, M. Goudarzi, R. Buyya, iFogSim2: An extended iFogSim simulator for mobility, clustering, and microservice management in edge and fog computing environments, J. Syst. Softw. 190 (2022) 111351, http://dx.doi.org/10.1016/J.JSS.2022.111351.

[26] H. Gupta, A. Vahid Dastjerdi, S.K. Ghosh, R. Buyya, iFogSim: A toolkit for modeling and simulation of resource management techniques in the Internet of Things, Edge and Fog computing environments, Softw.: Pr. Exp. 47 (2017) 1275–1296, http://dx.doi.org/10.1002/SPE.2509.

[27] R.N. Calheiros, R. Ranjan, A. Beloglazov, C.A.D. Rose, R. Buyya, CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms, Softw.: Pr. Exp. 41 (2011) 23–50, http://dx.doi.org/10.1002/SPE.995.

[28] R. Yao, X. Xu, CETO-sim: A simulation platform for cloud-edge task offloading, in: 2022 IEEE 25th International Conference on Computer Supported Cooperative Work in Design, CSCWD 2022, Institute of Electrical and Electronics Engineers Inc., 2022, pp. 133–138, http://dx.doi.org/10.1109/CSCWD54268.2022.9776182.

[29] A.R. Nandhakumar, A. Baranwal, P. Choudhary, M. Golec, S.S. Gill, EdgeAISim: A toolkit for simulation and modelling of AI models in edge computing environments, Meas.: Sens. 31 (2024) 100939, http://dx.doi.org/10.1016/J.MEASEN.2023.100939.

[30] P.S. Souza, T. Ferreto, R.N. Calheiros, EdgeSimPy: Python-based modeling and simulation of edge computing resource management policies, Future Gener. Comput. Syst. 148 (2023) 446–459, http://dx.doi.org/10.1016/J.FUTURE.2023.06.013.

[31] Q. Lu, G. Li, H. Ye, EdgeSim++: A realistic, versatile, and easily customizable edge computing simulator, IEEE Internet Things J. (2024) http://dx.doi.org/10.1109/JIOT.2024.3434641.

[32] X. Chen, G. Liu, Energy-efficient task offloading and resource allocation via deep reinforcement learning for augmented reality in mobile edge networks, IEEE Internet Things J. 8 (2021) 10843–10856, http://dx.doi.org/10.1109/JIOT.2021.3050804.

[33] N. Villegas, L. Diez, I.D.L. Iglesia, M. Gonzalez-Hierro, R. Aguero, Energy-aware optimum offloading strategies in fog-cloud architectures: A Lyapunov based scheme, IEEE Access 11 (2023) 73116–73126, http://dx.doi.org/10.1109/ACCESS.2023.3295496.

[34] M.J. Neely, Stochastic Network Optimization with Application to Communication and Queueing Systems, Springer International Publishing, Cham, 2010, http://dx.doi.org/10.1007/978-3-031-79995-2.

[35] G. Nieto, I. de la Iglesia, U. Lopez-Novoa, C. Perfecto, Deep Reinforcement Learning techniques for dynamic task offloading in the 5G edge-cloud continuum, J. Cloud Comput. 13 (2024) 1–24, http://dx.doi.org/10.1186/S13677-024-00658-0.