AUT Journal of Electrical Engineering
10.22060/eej.2025.24181.5657

Optimal Resource Management in Fog-Cloud Environments via A2C Reinforcement Learning: Dynamic Task Scheduling and Task Result Caching

Mohammad Hassan Nataj Solhdar[1], Mohamad Mehdi Esnaashari[2*]

[1,2]Faculty of Computer Engineering, K. N. Toosi University of Technology, Tehran, Iran[1]

Abstract

In order to effectively manage tasks in fog-cloud environments, this paper proposes a two-agent architecture-based framework. In this framework, a task scheduling agent is responsible for selecting the computing execution node and allocating resources, while a separate agent manages the caching of results. In each decision cycle, the resource manager first checks whether a valid, fresh result already exists in the cache; if so, the cached result is immediately returned. Otherwise, the execution agent evaluates current conditions — such as network load, nodes' computational capacity, and user proximity — and assigns the task to the most appropriate node. After task execution completes, an independent storage agent is selected to store the results, potentially operating on a node distinct from the execution node. Through extensive simulations and comparisons with advanced methods (e.g., A3C-R2N2, DDQN, LR-MMT, and LRR-MMT), we demonstrate significant improvements in response latency, computational efficiency, and inter-node communication management. The proposed framework decouples execution scheduling from result storage through two distinct agents while implementing history-based caching that tracks both task request frequencies and result recency. This design enables effective adaptation to variable workloads and dynamic network conditions. The two-agent architecture and history-based caching serve as core innovations that optimize resource utilization and enhance system responsiveness. The resulting decoupled, history-based strategy delivers scalable, low-latency performance and provides a robust solution for real-time service delivery in fog-cloud environments.

Keywords: Task Scheduling, Result Caching, Reinforcement Learning, Fog-Cloud Environment, Advantage Actor-Critic (A2C), Resource Management

[1] esnaashari@kntu.ac.ir
[2] nataj.solhdar@gmail.com

1. Introduction

The rapid evolution of digital technologies has ushered in an era of unprecedented connectivity and data generation. The Internet of Things (IoT) ecosystem, with its myriads of interconnected devices, has become a cornerstone of modern technological infrastructure[1]. This proliferation of smart devices and sensors has led to an exponential increase in data production and processing demands, challenging traditional computing paradigms [2]. Cloud computing, once hailed as the panacea for large-scale data processing, is increasingly being complemented by edge and fog computing architectures. These distributed computing models aim to address the latency and bandwidth constraints inherent in centralized cloud systems [3]. The fog-cloud continuum presents a hierarchical structure where computational resources are strategically distributed from the network edge to centralized data centers, offering a more flexible and responsive computing environment [4]. However, the heterogeneous and dynamic nature of fog-cloud ecosystems introduces complex resource management challenges. The variability in computational capabilities, network conditions, and task requirements necessitates sophisticated orchestration mechanisms to ensure efficient resource utilization and optimal task execution [5].

Existing research in fog-cloud resource management has predominantly focused on task scheduling algorithms, aiming to optimize task allocation based on various performance metrics [6]. While these approaches have yielded significant improvements in resource utilization and task completion times, they often overlook the potential benefits of strategic result caching, particularly for frequent or similar tasks [7]. Unlike conventional methods that often combine task execution and caching on the same node, this paper introduces a novel approach that decouples these two operations, presenting a more flexible and efficient resource management framework.

A key feature of our system is its ability to execute tasks on suitable nodes while potentially storing the results on different, more appropriate node. This decoupling of execution and storage locations, illustrated in Fig. 1), offers greater flexibility and can lead to improved resource utilization. For instance, a computationally intensive task might be executed in the cloud, but its results could be cached in a fog node closer to potential future requesters, thereby reducing latency for subsequent similar requests.

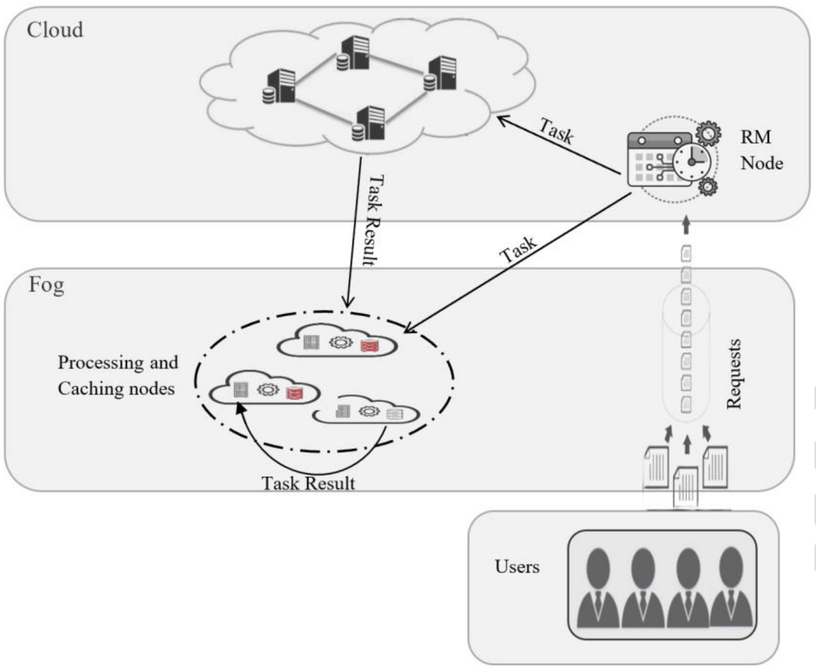


Fig. 1: System Architecture with Decoupled Task Execution and Caching.

Fig. 1) illustrates the high-level architecture of our Task Scheduling and Caching with Advantage Actor-Critic (TSC-A2C) framework. Incoming tasks first arrive at the Resource Manager (RM), which invokes the Execution Agent to select the optimal compute node—based on current CPU/RAM availability, network load, and user proximity. Once execution completes, the Caching Agent independently selects a fog node for storing the result, taking into account storage capacity, distance to potential future requesters, and task freshness. The two decision paths (execute vs. cache) are shown in parallel: the execution path (Task) directs tasks to a selected node for processing, and the caching path (Task Result) directs results to a possibly different node for storage. This decoupled design enables flexible placement: for example, a heavy compute job may run in the cloud (Node C), while its results are cached at a nearby fog node (Node F_2) to reduce latency for subsequent requests.

Main contributions of this paper are as follows:

1. **Dual-agent A2C framework.** We introduce two Advantage Actor-Critic agents—one for execution-node selection, and the other for caching-node selection—allowing flexible, independent optimization of scheduling and caching.

2. **History-aware caching** (i.e., maintaining statistics on task request frequency and cache freshness windows)**.** By tracking task frequency and data-freshness, the proposed method can only cache results of frequent tasks, avoiding storage bloat and ensuring freshness.

3. **Location-informed decisions.** We incorporate user-to-node proximity into both execution and caching policies, reducing communication latency and network congestion.

The remainder of this paper is organized as follows. Section 2 reviews related work in fog-cloud resource management and caching strategies. Section 3 formalizes our system model and problem statement. Section 4 describes the TSC-A2C[1] method in detail. Section 5 presents simulation settings and comparative results. Finally, Section 6 concludes and outlines future directions.

2. Related Work

The exponential surge in data generation, primarily driven by the widespread deployment of Internet of Things (IoT) devices, has presented significant challenges to traditional centralized cloud computing architectures in terms of processing speed, storage capacity, and network bandwidth [8]. In response to these limitations, fog computing has emerged as a critical architectural paradigm, extending computational and storage capabilities to the network edge, in closer geographical proximity to data sources and end-users [8]. The fundamental objective of fog computing is to mitigate network latency and facilitate rapid, localized data processing, thereby enabling real-time responsiveness for IoT applications [8]. This multi-layered, collaborative architecture, encompassing fog and cloud environments, is widely recognized as a promising solution for effectively managing the intricate data processing and communication demands of modern IoT applications [9].

Within these complex and dynamic distributed environments, efficient task scheduling and judicious resource management are paramount for operational success. The overarching goal is to ensure that user requirements are met within stringent time constraints, even with the inherently limited resources available at the network edge [8]. Recent reviews indicate a rapid adoption of Machine Learning (ML) and, more specifically, Deep Reinforcement Learning (DRL) techniques, such as Advantage Actor-Critic (A2C) and Double Deep Q-Network (DDQN), for various aspects of task scheduling, resource allocation, and caching [10]. This shift underscores the necessity for algorithms capable of autonomous, real-time learning and adaptation.

2.1. Novel Approaches in Task Scheduling and Resource Management

2.1.1. Deep Reinforcement Learning (DRL) for Adaptive Scheduling

DRL is increasingly recognized as a powerful paradigm for addressing the complexities of task scheduling and resource management in distributed environments, owing to its ability to learn optimal policies through iterative interaction with the environment [10].

[1] Task Scheduling and Caching with Advantage Actor-Critic

Advantage Actor-Critic (A2C) Approaches: The A2C method, often combined with DRL, is proposed for dynamic scheduling in stochastic edge-cloud environments, enabling decentralized learning and concurrent task scheduling across multiple servers [10]. A2C is particularly noted for its rapid adaptability even with limited data [10]. An A2C-DRL approach for Edge-Cloud involves developing sophisticated reward functions and integrating Hypergraph Neural Networks (HGNN) to extract complex features for dynamic resource allocation [10]. An improved A2C for Storm Workloads, leveraging Graph Neural Networks (GNN) to capture global features of dependent jobs, significantly enhances resource utilization and minimizes job completion time [11]. A2C-DRL demonstrates superior performance compared to other state-of-the-art algorithms in terms of reward value, task rejection rate, and load balancing factor [10].

Double Deep Q-Network (DDQN) Implementations: DDQN models are proposed as robust reinforcement learning solutions for complex task scheduling problems in cloud computing [12]. DDQN mitigates overestimation bias by employing two distinct neural networks, leading to a more consistent learning process [12]. This approach adaptively allocates tasks considering multiple criteria such as job priority, resource availability, execution time, and cost in dynamic cloud environments [12]. DDQN consistently outperforms conventional scheduling approaches in task success rates [12]. A DDQN-based algorithm for operating system scheduling has shown improved task completion efficiency, system throughput, and faster response speeds, particularly for I/O-intensive tasks [13].

Other Machine Learning and DRL Integrations: Federated Deep Reinforcement Learning (FDRL) is utilized for optimizing caching strategies in cloud-edge integrated environments, balancing caching efficiency and training energy expenditure [14]. A novel intelligent resource allocation algorithm combines Long Short-Term Memory (LSTM) networks for demand prediction with Deep Q-Networks (DQN) for dynamic scheduling, enhancing resource utilization by 32.5%, reducing average response time by 43.3%, and lowering operational costs by 26.6% [15]. Agile Reinforcement Learning (aRL) is an innovative DRL approach for real-time task scheduling in edge computing, incorporating "informed exploration" and "action masking" to accelerate policy convergence, achieving higher hit-ratios and faster runtime [16].

2.1.2. Multi-Agent Systems (MAS) for Decentralized Control and Coordination

Multi-Agent Systems (MAS) are gaining prominence as a robust architectural paradigm for distributed computing, offering enhanced modularity, specialized functionalities, improved collaborative learning, and more effective decentralized decision-making [17]. Recent developments include integrating Large Language Models (LLMs) as agents for task allocation, where a "planner method" (LLM generating a plan for other LLM agents) outperforms an "orchestrator method" in handling concurrent actions [17]. Agent-based frameworks are specifically proposed for fog computing to autonomously manage task scheduling, load balancing, and rescheduling, reducing reliance on a single central point of control and improving system resilience. MAS are well-suited to handle uncertainties and dynamic changes in real-time distributed environments [18].

2.1.3. Intelligent Caching Strategies for Enhanced Responsiveness and Resource Utilization

Caching mechanisms are fundamental to enhancing fog network performance by improving latency and reducing energy consumption [8]. Recent developments demonstrate a progression from static data placement to dynamic and predictive approaches. Reinforcement Learning (RL) is increasingly applied to intelligent caching, with Q-learning being used to discover optimal caching policies in a distributed manner, enabling fog access points to adapt to dynamic content popularity without additional communication overhead [19]. A cutting-edge development involves a novel dynamic federated optimization strategy that utilizes Federated Deep Reinforcement Learning (FDRL) for optimized caching in cloud-edge integrated environments, balancing caching efficiency and training energy expenditure [14]. Beyond reactive caching, predictive caching is gaining traction, involving forecasting workload resource consumption to enable more accurate scheduling and resource management [20].

2.1.4. State-of-the-Art in Load Balancing for Multi-Layered Architectures

Load balancing is an indispensable mechanism in multi-layered distributed computing architectures, including Mist-Fog-Cloud environments. Its primary function is to uniformly distribute workloads across available fog and cloud layers, minimizing latency, optimizing power consumption, and ensuring efficient resource utilization. Load balancing algorithms are categorized into traditional (e.g., FCFS, SJF, Round Robin), heuristic (e.g., Max-Min, Min-Min, Throttled Algorithm), metaheuristic (e.g., Genetic Algorithm, Ant Colony Optimization), hybrid, hyper-heuristic, and Machine Learning (ML)-based algorithms. ML/DL-based algorithms are increasingly effective for predicting unforeseen or future resource requirements in dynamic environments, particularly for managing large and unpredictable IoT requests. The overall architecture is often conceptualized as a four-layered hierarchy: IoT-Mist-Fog-Cloud, with load balancing mechanisms operating across these layers for seamless operation [21].

Traditional and heuristic algorithms, while widely used, exhibit significant limitations in dynamic Mist-Fog-Cloud environments, often leading to inefficiencies or suboptimal resource allocation. Metaheuristic algorithms offer broader applicability but can be computationally intensive. Hybrid algorithms combine strengths to address individual limitations. ML-based algorithms, including DRL, are highly promising due to their ability to learn directly from raw data and adapt to unpredictable workloads in real-time, though challenges in computational intensity and accuracy persist [21].

10.22060/eej.2025.24181.5657

Statistical techniques like LR-MMT[1][22], LRR-MMT[2][23], and MAD-MC[3] [24] offer methods for dynamic task scheduling and congestion management, but machine learning, particularly DRL (DDQN, A3C) [25-28], provides more adaptable solutions for complex and dynamic environments. DDQN [25-27] adapts to new contexts without manual reconfiguration, while A3C with R2N2 [28] integrates policy gradients with recurrent neural networks to handle temporal dynamics. These DRL methods show promise for managing tasks in fog and cloud infrastructures, especially in stochastic settings requiring rapid adaptation.

Table 1: Comparative Analysis of Proposed Method (TSC-A2C) with Recent Advancements

Feature/Aspect	Recent Advancements	Key Contributions/Research Gaps Addressed by TSC-A2C
Core Architecture	Trend towards DRL for adaptive scheduling [10] and Multi-Agent Systems for decentralized control [17].	Explicit decoupling of execution and storage via dedicated agents enhances modularity and specialized control, offering a clear architectural distinction.
Scheduling Mechanism	Advanced DRL methods (A2C [10], DDQN [12], LSTM+DQN [15]) for dynamic and multi-objective scheduling, often integrating GNNs for complex feature extraction [11].	Leverages A2C for real-time adaptability, aligning with recent DRL trends, but within a specific two-agent framework that could offer distinct benefits in coordination.
Caching Strategy	Evolution to intelligent, dynamic, and predictive caching, often using RL (Q-learning [19], Federated DRL [14]) to adapt to content popularity and minimize energy [14].	Integrates history-based caching as a core component, managed by a dedicated agent, directly addressing result reusability and efficiency. This specialized agent can optimize caching decisions independently.
Resource Management & Optimization Objectives	Strong trend towards multi-objective optimization (makespan, cost, energy, latency, resource utilization, load balancing) often achieved via DRL [8].	Aligns with multi-objective optimization trends, with a strong emphasis on real-time adaptability and robustness in dynamic environments, facilitated by the two-agent structure.
Uncertainty & Dynamism Handling	DRL and MAS are specifically designed to manage unpredictable and fluctuating workloads, uncertain events, and heterogeneous resources[18].	The two-agent architecture and history-based caching contribute to this adaptability by providing specialized and responsive mechanisms for handling dynamic changes.

3. System Model and Problem Formulation

In this Section, we will delineate the system model and explicitly articulate the problem at hand.

3.1. System Model and Problem Statement

[1] Local Regression-Minimum Migration Time
[2] Local Robust Regression-Minimum Migration Time
[3] Median Absolute Deviation-Maximum Correlation

10.22060/eej.2025.24181.5657

In the present study, the system model, illustrated in Fig. 2), integrates both fog and cloud infrastructures alongside user and device components. The resource management system, accountable for orchestrating user tasks and directing them to fog and cloud nodes, is situated within the fog infrastructure. This is conceptualized as a singular managerial entity overseeing all accessible resources. For the sake of clarity, and as depicted in Fig. 2), the cloud and fog infrastructures are collectively termed the computing layer in this framework.

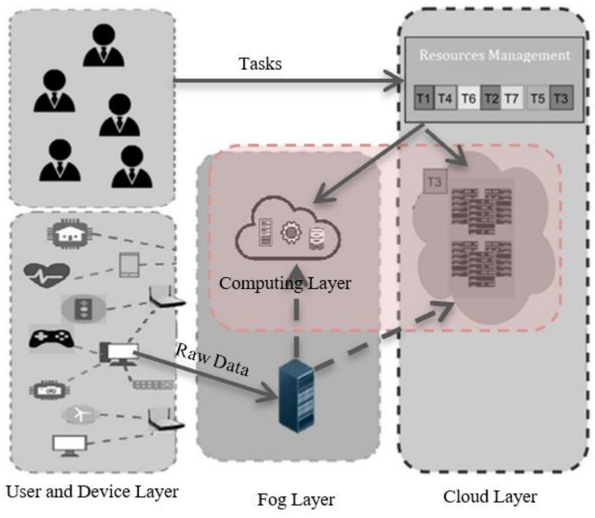


Fig. 2: System Model

Fig. 2) depicts the detailed system model, showing the hierarchy from end users through fog and cloud layers. Each end user node ($U_1 \ldots U_k$) issues tasks to the fog layer, where a single RM coordinates two agents. The fog layer comprises heterogeneous fog nodes ($F_1 \ldots F_n$) with varying CPU, memory, and storage capacities, connected in a mesh topology. The cloud datacenter sits above the fog layer and is used when fog resources are insufficient or task deadlines demand higher compute power. Solid lines indicate task submission and result dispatch paths, while dashed lines show inter-node communication for caching and neighbor discovery. This diagram highlights how the RM maintains up-to-date information on node capacities, neighbor sets, and cache contents to inform both execution and caching decisions.

The computational layer in fog computing comprises heterogeneous resources with diverse processing capabilities, memory, and storage, where fog nodes, though less computationally powerful than cloud resources, offer reduced latency due to their proximity to users. The resource management (RM) system is responsible for receiving tasks, determining if they are novel or can be served from cache, allocating new tasks to suitable resources based on CPU, RAM, disk requirements, and completion time, and deciding whether to cache task results. Despite this comprehensive architecture, a key challenge remains: how to jointly decide, for each incoming task, both its execution destination (fog or cloud) and the fog node at which to cache its result so as to balance end-to-end latency, resource cost, and cache freshness. These decisions must be made online, at each discrete scheduling cycle, based on dynamic observations of network load, node capacity, and task characteristics. In the next subsection, we formalize this joint scheduling–caching problem as an optimization over decision variables representing execution and caching assignments, subject to Service Level Agreement (SLA) deadlines and storage constraints.

3.2. Formal definition of the problem

The primary goal of the RM system is to minimize overall latency and resource utilization while maximizing the number of processed tasks, operating in discrete time cycles with scheduling performed at the end of each cycle after tasks are queued Fig. 3).

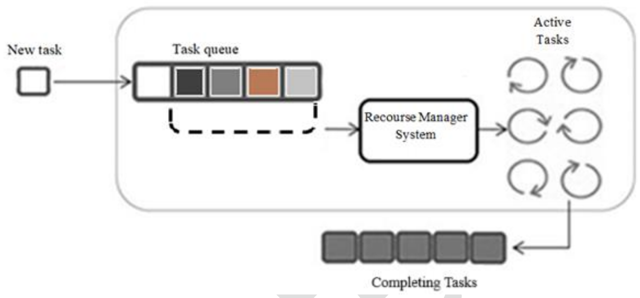


Fig. 3: Task state in queue

Fig. 3): Task lifecycle within a scheduling cycle. Tasks submitted by users enter the RM's queue, where they await processing. The RM first checks for a valid cached result. If absent or stale, the RM invokes the Execution Agent to select a processing node. Once execution completes, tasks identified as "Frequent" trigger the Caching Agent to choose a storage node for the result. The symbols employed by the suggested approach are presented in below definitions:

Definition 1 (Task Set): Let $T = \{t_1, t_2, ..., t_{|T|}\}$ denote the set of tasks submitted to the Resource Manager (RM). Each $t_i \in T$ is the i-th task arriving within a given scheduling cycle, carrying its own resource requirements and timing constraints.

Definition 2 (User Set): Let $U = \{u_1, u_2, ..., u_{|U|}\}$ denote the set of end users who submit tasks. Each user $u_k \in U$ issues tasks from the user layer and expects results by a specified deadline. We assign each u_k a fixed 2D location $loc(u_k) = [x_{u_k}, y_{u_k}]$ for proximity calculations.

Definition 3 (Task Submitter Ensemble): For each task t_i, let $\mathcal{U}(t_i) \subseteq U$ be the set of users who issue identical copies of t_i within the same cycle, causing overlapping requests.

Definition 4 (Task Frequency): Define the frequency $f(t_i)$ of task t_i as the number of times t_i has been submitted up to the current time. Formally, $f(t_i) = |\{t_j \in T \mid t_j \equiv t_i\}|$.

Definition 5 (Frequent Task Set): Let Ψ be a predefined repetition threshold. The set of frequent tasks is $F = \{t_i \in T \mid f(t_i) \geq \Psi\}$. Only tasks in F are considered for caching.

Definition 6 (Data Validity Window): For each t_i, let $w(t_i)$ be the time duration during which its cached result remains valid. After $w(t_i)$ elapses, the result must be recomputed to ensure freshness.

Definition 7 (Task Priority): Each task t_i has a priority level $p(t_i) \in \mathbb{R}^+$ that influences scheduling order; higher $p(t_i)$ means more urgent processing.

Definition 8 (Completion Deadline): Each task t_i must finish by time $dl(t_i)$. If execution on any node exceeds $dl(t_i)$, an SLA violation occurs.

Definition 9 (Execution Resource Profile): For each t_i, its execution resource requirements are given by the tuple $r_{exec}(t_i) = (m_i, c_i, s_i)$, where m_i is memory, c_i is CPU power, and s_i is storage needed for processing.

Definition 10 (Storage Requirement Profile): For each periodic task $t_i \in F$, its result storage requirement is $r_{cach}(t_i) = s_i'$, the disk space needed to cache its output for future reuse.

Definition 11 (Fog Node Set): Let $N = \{n_1, n_2, ..., n_{|N|}\}$ be the set of all fog nodes. Each n_j can play one or both roles: Processing node ($n_j \in N_{proc}$) for executing tasks, Caching node ($n_j \in N_{cache}$) for storing results.

Definition 12 (Fog Processing Node): A fog processing node, called fpn_j, is one of the fog environments that actually runs the incoming tasks. It has enough CPU and memory to handle the task t_i, and it decides which tasks to run and in what order based on how urgent they are and any service-level deadlines.

Definition 13 (Fog Caching Node): A fog caching node, called fcn_j, is one of the fog environments that stores the results of frequent tasks. It has disk space set aside to keep these results fresh for a certain time window, and it picks which results to keep so future similar requests can be served faster.

Definition 14 (Node Capacity): Each fog node n_j has capacity $Cap(n_j) = (M_j, C_j, S_j)$, denoting its available memory M_j, CPU power C_j, and storage S_j for that cycle.

Definition 15 (Node Neighborhood): For each n_j, let $\mathcal{N}(n_j) \subseteq N$ denote its directly connected neighbor nodes, forming a collaborative mesh for offloading and caching.

Definition 16 (Active & Scheduled Tasks at Node): At node n_j, let $AST_j = \{t_i \in T \mid t_i$ is executing or queued for execution on $n_j\}$ represent its current workload.

10.22060/eej.2025.24181.5657

Definition 17 (Cached & Future-Cache Tasks at Node): At node n_j, let $CFT_j = \{t_i \in F \mid t_i$'s result is cached or scheduled to be cached on $n_j\}$ capture its caching assignments.

Definition 18 (User-to-Node Proximity): The distance between user u_k and node n_j is $d(u_k, n_j) = ||loc(u_k) - loc(n_j)||$, used to factor latency into scheduling and caching decisions.

Definition 19 (Submission Delay): $\delta_{sub}(t_i, n_j)$ is the time from when t_i leaves its submitter to when it arrives at n_j for execution or caching.

Definition 20 (Processing Duration): $\delta_{proc}(t_i, n_j)$ is the wall-clock time for n_j to execute t_i from start to finish.

Definition 21 (Result Dispatch Time): $\delta_{disp}(u_k, n_j)$ is the time for node n_j to send the completed result of t_i back to user u_k.

Definition 22 (Total Execution Time): For task t_i on node n_j, $CTET(t_i, n_j) = \delta_{sub} + \delta_{proc} + \delta_{disp}$, the end-to-end latency experienced by the user.

Definition 23 (Execution Cost): $EC(t_i, n_j)$ measures the CPU and memory consumption cost for executing t_i on n_j over its processing duration.

Definition 24 (Storage Cost): $SC(t_i, n_j)$ is the cost of occupying n_j's storage to cache t_i's result for future requests.

Definition 25 (**The Maximum Allowable SLA Violation):** Denoted as V_{sla}, indicates the percentage of tasks that are accepted to be completed after their deadline. In fog-cloud environments, SLA violations indicate an inability to meet Quality of Service (QoS) requirements, such as response time or processing time. The parameter V_{sla} allows the designer to specify an acceptable level of SLA violation, enabling fewer fog nodes while still maintaining system efficiency. In other words, if a limited SLA violation is tolerable, fewer resources may be utilized, lowering system costs. This variable thus provides an option for balancing precision and efficiency, and by setting an appropriate V_{sla} value, the designer can adjust the number of fog nodes to meet SLA requirements.

The symbols employed by the suggested approach are presented in Table 2).

Table 2: Symbols

#	Term	Description
1	t_i	In the task set, t_i represents the i-th task that have been assigned to the Resource Manager (RM) by a user at during a specific timeframe.

2	ue_k	The User Ensemble (ue_k) consists of individual user each submitting tasks with the expectation of receiving outputs within a specific timeframe, represented by their unique spatial coordinates.
3	uts_i	Task Submitter (uts_i) represent multiple users who send identical tasks t_i within a given time frame, leading to overlapping task requests.
4	tf_i	Task Frequency (tf_i) represent the rate at which a particular task t_i is repeated within until now, indicating the periodic nature of the task.
5	pt_i	The Periodic Task (pti) is a task that surpass a defined repetition count ψ, highlighting tasks that frequently recur over time.
6	dv_i	The Data Validity (dv_i) Window represent the duration for which the cached results of tasks are considered valid before reprocessing becomes necessary.
7	tp_i	Task Priority (tp_i) indicates the priority of tasks t_i, representing their importance relative to other tasks in the system, thus influencing scheduling decisions.
8	tcd_i	The Task Completion Deadline (tcd_i) represent the strict time frame within which a task t_i must be completed to meet system performance criteria.
9	er_i	The Execution Resource (er_i) Profile outlines the specific resource requirements for task execution, including memory m_i, CPU power c_i, and storage s_i .
10	srr_i	The Storage Requirement for Results (srr) represent the storage needed to store the output of Periodic Task tasks pt_i , ensuring future reuse of cached results.
11	NoF	The Node of Fog (NoF) consists of set of nodes that serve dual purposes: processing tasks (Fog Processing Nodes, fpn) and, when necessary, also store results (Fog Caching Nodes, fcn).
12	fpn_j	The Fog Processing Node (fpn) represent the fog of node responsible for executing tasks within the fog environment.
13	fcn_j	The Fog Caching Node (fcn) represent the fog of node responsible for managing result caching tasks within the fog environment.
14	nc_j	Node Capacity (nc_j) Metrics represent the available memory M_j, CPU cycles C_j, and storage S_j at each fog node for accommodating tasks.
15	nn_j	The Node Neighboring (nn_j) comprises neighboring fog nodes that is interconnected, creating a collaborative processing and caching framework.
16	AST_j	Active and Scheduled Tasks (AST_j) represent the ongoing Active Tasks and upcoming Scheduled Tasks assigned to fog node fpn_j for processing.
17	CFT_j	Cached and Future Cache Tasks (CFT_j) include tasks whose results are either currently cached CT_j or scheduled to be stored SCT_j for later retrieval at a fog node fcn_j.
18	UN (ue_k, fpn_j/fcn_j)	User-to-Node (UN) Proximity calculates the geographical distance between the task-requesting user ue_k and the fog node fpn_j or fcn_j tasked with processing or caching it.
19	Δ (t_i, fpn_j/fcn_j)	Task Submission delay (Δ (t_i, fpn_j/fcn_j)) defines the time delay experienced between task (t_i) submission and its reception at the designated fog node (fpn_j/fcn_j) for execution or caching.

20	PD(t_i , fpn_j)	The Processing Duration (PD(t_i , fpn_j)) is the total time taken for a fog node fpn_j to fully execute a task t_i from the moment it starts processing.
21	RDT(ue_k, fpn_j/fcn_j)	Result Dispatch Time (RDT) specifies the time required for a fog node to transmit processed results back to the task-requesting user ue_k.
22	CTET(t_i , fpn_j)	The Comprehensive Task Execution Time (CTET) is the sum of the task submission delay, processing duration, and result dispatch time, reflecting the total time from task t_i submission to final result delivery.
23	EC(t_i , fpn_j)	Execution Cost (EC) refers to the resources consumed during task t_i processing, measured by the CPU and memory usage over the duration of execution.
24	SC(t_i , fcn_j)	Storage Cost (SC) is the expense incurred by occupying fog node fpn_j storage space to cache results for future use.
25	V_{sla}	indicates the percentage of tasks that are accepted to be completed after their deadline.

A task t_i refers to a task submitted to the Resource Management (RM) node prior to its execution. If t_i is identified as a Frequent Task (t_i ∈ F) name it as ft_i, the RM must not only schedule it for execution but also determine a suitable node for caching its results. Considering the previously defined terms and notations, the problem can be formulated as follows: Design an efficient Resource Manager (RM) capable of identifying an optimal node for executing t_i and, in the case of Frequent Tasks (ft_i), selecting an appropriate node for result caching, thereby ensuring both effective scheduling and efficient storage management.

For task t_i :

- $\sum_j \sum_i EC(t_i, fpn_j)$ is minimized and
- $\sum_j \sum_i CTET(t_i, fpn_j)$ is minimized and
- $d(u_k, fpn_j)$ is minimized

subject to the scheduling-related criteria:

$$\sum_{t_i \in AST_{fpn_j}} c_{t_i} \le C_{fpn_k}; \forall fpn_k \quad (1)$$

$$\sum_{t_i \in AST_{fpn_j}} m_{t_i} \le M_{fpn_k}; \forall fpn_j \quad (2)$$

$$CTET\,(t_i, fpn_k) \le w\,(t_i); \forall t_i \in AST_{fpn_j}, \forall fpn_j \tag{3}$$

The first two criteria state that each node *fpnj* has to have enough processing and memory capacity for running all its scheduled tasks. The third criterion indicates that task completion times for all scheduled tasks on all fog nods have to be sooner than their deadlines.

if the task is a frequent task (name it as ft_i) then:

- $\sum_{u_k \in U_{ft_i}} d\,(u_k, fcn_j)$ is minimized and
- $|N\,(n_{ft_i})|$ is maximized

subject to the Result caching-related criteria:

$$\sum_{ft_i \in CFT_{fcn_j}} r_{cach(t_i)} \le S'_{fcn_j}; \forall fcn_j \tag{4}$$

The last criterion states that each node *fcnj* has to have enough storage capacity for caching the results of all frequent tasks that are scheduled to be cached on it.

4. The Proposed Method: TSC-A2C

This section elaborates on the proposed Task Scheduling and Caching with Advantage Actor-Critic (TSC-A2C) framework. We begin with an overview of the method, followed by a detailed description of its architecture and the core reinforcement learning formulation. Finally, the operational flow of the system is presented. The TSC-A2C method is designed to dynamically manage resources in fog-cloud environments by intelligently scheduling tasks for execution and strategically caching their results, particularly for frequently occurring tasks.

4.1. Overview of the TSC-A2C Framework

The TSC-A2C framework leverages reinforcement learning (RL) to address the complexities of dynamic task scheduling and result caching in heterogeneous fog-cloud environments. At its core, the system employs a Resource Manager (RM) that orchestrates task processing. Recognizing the distinct nature of task execution and result caching, particularly the benefits of decoupling these decisions, TSC-A2C utilizes a dual-agent learning approach based on the Advantage Actor-Critic (A2C) algorithm. This choice is motivated by A2C's balance of sample efficiency and stability in complex decision-making spaces.

For every incoming task, the RM, guided by its RL agents, makes decisions to optimize performance metrics such as execution time, operational cost, and resource utilization. A key feature is the specialized handling of "Frequent Tasks," where results are considered for caching to minimize redundant computations and improve response times for subsequent identical requests.

4.2. TSC-A2C Framework Architecture

The architecture of TSC-A2C is designed around a central Resource Manager and two specialized A2C agents, as depicted conceptually in the overall system model (refer to Figure (2) for the broader system context and Figure (1) for the decoupled execution/caching concept).

4.2.1. Resource Manager (RM)

The RM serves as the primary coordination entity within the fog layer. It receives all incoming user tasks, maintains the task queue, and initiates the decision-making process by invoking the appropriate RL agents. The RM is responsible for the overall management of the task lifecycle, from arrival to completion and potential caching of results.

4.2.2. Dual-Agent A2C System

To effectively manage the distinct yet related problems of task execution and result caching, TSC-A2C employs two concurrent A2C agents:

Execution Agent (A_p): This agent is responsible for selecting the optimal node for executing an incoming task t_i. The selection considers factors such as node processing capabilities, current load, and proximity to the user to minimize execution time and cost. This agent is invoked for all tasks requiring execution.

Caching Agent (A_c): This agent is activated specifically for tasks identified as "Frequent Tasks" (ft_i). Its role is to choose a suitable fog node for caching the results of ft_i. The caching node can be different from the execution node, allowing for optimized storage placement based on factors like user distribution for the cached result and storage availability.

4.2.3. Cache Communication Protocol

To facilitate efficient interaction with the distributed cache memory across fog nodes, our framework employs a lightweight, microservice-based communication model. Each fog node hosts a dedicated caching service that exposes a simple Remote Procedure Call (RPC) interface over TCP/IP, which can be implemented using technologies like gRPC or RESTful HTTP. When the Caching Agent selects a node for storage, or when the Resource Manager performs a cache check, the RM issues an RPC 'store' or 'lookup' request to the selected node's caching endpoint, transmitting the task identifier and relevant result payload. Nodes then respond with acknowledgments or the requested cached data. This middleware layer effectively decouples the network communication specifics from the core scheduling logic, ensuring interoperability across heterogeneous platforms while introducing minimal overhead, typically in the sub-millisecond range per RPC call. In our iFogSim simulations, this interaction is abstracted by direct method calls, but the described RPC design is readily implementable in real-world deployments.

4.3. Reinforcement Learning Formulation

The core intelligence of TSC-A2C lies in its RL formulation, where agents learn optimal policies through interaction with the fog-cloud environment. The standard components of this RL formulation are detailed below. The RM serves as the primary coordination entity within the fog layer. It receives all incoming user tasks, maintains the task queue, and initiates the decision-making process by invoking the appropriate RL agents. To ensure real-time awareness of the dynamic environment, the RM maintains an in-memory registry that is continuously updated. This registry includes a lookup table detailing which fog nodes currently hold cached results for frequent tasks, alongside their freshness status. Furthermore, each fog node periodically sends heartbeat updates containing its latest resource availability (CPU, memory, and storage) to the RM via the established RPC interface (as detailed in Section 4.2.3). These dynamic updates enable the RM to always reflect real-time capacities rather than static snapshots, crucial for informed execution and caching decisions.

4.3.1. State Representation

Effective learning requires a comprehensive representation of the environment's state. The state observed by each agent is tailored to its specific decision-making context:

Processing Environment State (PS) for Agent A_p: At the time of scheduling task t_i, the state $PS(i,fp)$ is defined as a tuple: $PS(i,fp)=(task\_desc_i, node\_states_i)$.

$task_{desc_i}$ is a 5-tuple representing the current task $t_i :< p(t_i), w(t_i), m_i, c_i, s_i >$, corresponding to its priority, data validity window, and resource requirements (memory, CPU, storage) as defined in Section 3.2.

$node_{states_i} =< fpn_{1,i}, fpn_{2,i}, ..., fpn_{(|NoF|),i} >$ represents the current state of all |N| fog nodes. The state of a given fog processing node $fpn_{i,j}$ is a 4-tuple: $M_{i,j}, C_{i,j}, |AST_{i,j}|, EC_{i,j}$, denoting its available memory, available CPU, number of active/scheduled tasks, and current execution cost metrics, respectively, at task *i* for node *j*.

Caching Environment State (CS) for Agent A_c: When considering a frequent task ft_i for caching, the state $CS(i, f_c)$ is defined as a tuple: $CS(i, fc) = (cache_{task_{desc_i}}, cache_{node_{states_i}})$.

$cache_{task_{desc_i}}$ represents the storage requirement s'_i (defined as $r_{cach}(t_i)$ in Definition 10) for the result of ft_i.

$cache_{node_{states_i}} = < fcn_{1,i}, fcn_{2,i}, ..., fcn_{(|N|),i} >$ represents the current state of all |N| fog nodes relevant for caching. The state of a given fog caching node $fpn_{i,j}$ is a 4-tuple: $< S_{i,j}, CFT_{i,j}, SC_{i,j}, |N(n_j)| >$, denoting its available storage, number of cached/future-cache tasks, current storage cost metrics, and the number of its neighbors, respectively, at time *i* for node *j*.

4.3.2. Action Space

The action space defines the set of possible decisions each agent can make:

Action Space for Agent A_p: The action $a_p \in A_p$ for agent A_p is the selection of a processing node from the set of all available fog nodes N_{proc} and the cloud resource. Thus, $A_p = N_{proc} \cup \{Cloud\}$.

Action Space for Agent A_c: The action $a_c \in A_c$ for agent A_c is the selection of a caching node from the set of available fog caching nodes N_{cache}. Thus, $A_c = N_{cache}$.

4.3.3. Actor-Critic Model Design

Both Ap and Ac agents employ an A2C architecture, each consisting of an actor network and a critic network.

Actor Network: The actor network learns the policy, i.e., a mapping from state to a probability distribution over actions.

For Agent A_p (processing), the input layer size is 5 + 4 × |N|, corresponding to the 5 task status features and 4 features per fog node.

For Agent Ac (caching), the input layer size is 1 + 4 × |N|, corresponding to the 1 frequent task storage feature and 4 features per fog node.

Both actor networks utilize a normalization layer followed by two hidden layers with 64 and 32 neurons, respectively, using the ReLU activation function.

The output layer is a softmax layer with a neuron count equal to the number of possible actions (nodes for execution or caching), representing the probability distribution for selecting each node.

Critic Network: The critic network evaluates the state-value function *V(s)*, estimating the expected return from a given state.

Separate critic networks are implemented for task execution and caching. Each critic receives the relevant environmental state (*PS* or *CS*) and the probability distribution from its corresponding actor network as input.

The critic network architecture includes two hidden layers with 64 and 32 neurons (ReLU activation), an additional hidden layer with 16 neurons, and an output layer yielding a scalar value for *V(s)*. We chose ReLU for both actor and critic networks because it is widely recognized for its computational efficiency and robustness against vanishing gradients in deep reinforcement learning applications. It yields sparse activations and faster convergence compared to sigmoid or tanh functions [29]. Our A2C agents use two fully connected hidden layers (64 and 32 neurons), which aligns with typical configurations found in standard A2C implementations in the literature (e.g. 2×32 settings for actor and critic in educational and domain-specific RL examples) [30]. This configuration strikes a balance between representation capacity and inference speed, especially important in fog-node deployment. Preliminary empirical tuning indicated that increasing beyond 64 neurons per layer marginally improved performance at the cost of longer inference latency, while smaller networks underperformed in scheduling quality.

Action Selection: Actions are selected using an ε-greedy strategy based on the probability distribution output by the actor network to balance exploration and exploitation.

4.4. Reward Computing

Let the probability distribution given by the softmax layer to be represented by $\pi_\theta(a|PS(i,f_p))$ or $\pi_\theta(a|CS(i,f_c))$, where θ is the parameters of the actor network. Then the action selection mechanism would be carried out using an ϵ-greedy method given by equation (5, 6).

$$a_i^p = \begin{cases} randomaction & ;if \quad rand < \varepsilon \\ \arg\max_a \pi_\theta(a|PS(i,f_p)) & ;otherwise \end{cases} \tag{5}$$

$$a_i^c = \begin{cases} randomaction & ;if \quad rand < \varepsilon \\ \arg\max_a \pi_\theta(a|CS(i,f_c)) & ;otherwise \end{cases} \quad (6)$$

If the node selected by a_i^p lacks sufficient resources to execute the task t_i, or if the node selected by a_i^c cannot store the processed results of pt_i, the corresponding action is penalized (details on the rewarding/penalizing process is given in the subsequent sections). A new action is then chosen using the respective equation (5 or 6), excluding the previously selected node from the available options. This iterative process ensures efficient and adaptive decision-making for both task execution and caching in dynamic fog environments.

The reward signal for A_p in the cycle is computed according to the equation (7) given below.

$$R(A_p) = \begin{cases} 0 & \text{; The task is not executed on time} \\ \frac{\tilde{nn}_j}{\tilde{CTET}((t_i, fpn_j)*\tilde{EC}((t_i, fpn_j)} & \text{; Otherwise} \end{cases} \quad (7)$$

In the above equation, $\tilde{nn}_j$, $\tilde{EC}$, $\tilde{CTET}$,and $\tilde{D}$ are all normalized values computed according to the equations (6) to (11) as given below. That is to say, if the task fpt_j cannot be executed on time, the reward would be 0. Otherwise, the reward would be computed in such a way that a node with higher number of neighbours ($|nn_j|$), less execution time ($CTET$), less execution cost (EC), and less distance with the requesters of fpt_j receives higher values as the reward.

$$\tilde{nn}_j = \frac{|\tilde{nn}_j|}{nn^{\max}} \quad (8)$$

where nn^{max} is the maximum number of neighbors in the fog layer

$$\boldsymbol{\tilde{CTET}\left(t_i, fpn_j\right) = \frac{CTET\left(t_i, fpn_j\right)}{CTET^{max}}} \quad (9)$$

where $CTET^{\max}$ is the maximum value for the $CTET$

$$\tilde{CTET}(t_i, fpn_j) = \frac{CTET(t_i, fpn_j)}{CTET_{total}^{\max}} \quad (10)$$

where $CTET_{total}^{\max}$ is the maximum possible value for the total cost of executing

10.22060/eej.2025.24181.5657

That is to say, if the task t_i cannot be executed on time, the reward would be 0. Otherwise, the reward would be computed in such a way that a node with less execution time (*CTET*) and less execution cost receives higher values as the reward.

The reward signal for A_c in the cycle is computed according to the equation (11) given below.

$$R(A_c)=\begin{cases}0 & \text{; The task is not cached in node}\\ \frac{1}{\tilde{SC}(t_i,fcn_j)*\tilde{D}(t_i,fcn_j)} & \text{; Otherwise}\end{cases} \tag{11}$$

$$\tilde{SC}(t_i,fcn_j)=\frac{SC(t_i,fcn_j)}{SC_{total}^{\max}} \tag{12}$$

where $SC_{total}^{\max}$ is the maximum possible value for the total cost of caching

$$\tilde{D}(fcn_j)=\frac{\sum_{ue_k\in uts_{pt_i}} UN(ue_k,fcn_j)}{\max(\sum_{ue_k\in uts_{pt_i}} UN(ue_k,fcn_j))} \tag{13}$$

where $\max(\sum_{ue_k\in uts_{pt_i}} UN(ue_k,fcn_j))$ is the maximum of distance

That is to say, if the task pt_i cannot be cached in fog fcn_j, the reward would be 0. Otherwise, the reward would be computed in such a way that a node with less distance to users requesting similar tasks (*UN*) and less caching cost receives higher values as the reward.

4.5. Operational Flow of TSC-A2C

The proposed TSC-A2C method operates in cycles, processing tasks from a queue managed by the RM. The overall process is visually depicted in Fig. 4) and can be summarized as follows:

1. Task Arrival and Queuing: New tasks arrive and are placed in the task queue.

2. Cache Check: For each task dequeued by the RM, it first checks if a fresh, valid result for this task already exists in the cache of any fog node.

If a valid cached result is found ("Yes" path from "Is this task cached?" and "Does it have acceptable freshness?" in Fig. 4)): The RM retrieves the result and returns it to the user, bypassing execution. The process then moves to the next task.

3. Handling Cache Misses due to Node Unavailability: If a fog node that previously cached a result becomes unavailable (e.g., due to a failure), it will not respond to the RM's cache lookup request. In such instances, the Resource Manager (RM) treats the task as a cache miss. Consequently, the RM invokes the Execution Agent to re-execute the task on an available fog node or in the cloud. Following successful re-execution, the standard caching policy is applied to store the new result on the best candidate node, ensuring future requests for this frequent task can be served efficiently. This approach maintains system functionality even when cached data sources are temporarily inaccessible.

4. Execution Path (No Valid Cache): If no valid cached result exists ("No" path from cache checks in Fig. 4)), the task must be executed.

Execution Node Selection (Agent A_p): The RM provides the current Processing Environment State (PS) to Agent Ap. Agent Ap's actor network outputs a probability distribution over suitable execution nodes (fog nodes or cloud). The agent selects an execution node based on this distribution (e.g., highest probability or ε-greedy exploration). This corresponds to "The RL agent is used for scheduling..." block in Fig. 4).

The system verifies if the selected node is appropriate (e.g., has resources). If not, another node is selected.

5. Caching Decision and Node Selection (Agent A_c - For Frequent Tasks):

The system determines if the current task needs to be cached (i.e., if it's a "Frequent Task" and meets caching criteria – "Is this task need to be cached?" in Fig. 4)).

If the task is deemed a Frequent Task ("Yes" path):

The RM provides the current Caching Environment State (CS) to Agent A_c. Agent A_c's actor network outputs a probability distribution over suitable fog nodes for caching. The agent selects a caching node. This corresponds to "Choose the highest probability..." block for caching in Fig. 4).

The system verifies if the selected caching node is appropriate.

6. Task Execution and Result Caching:

* The task is executed on the node selected by Agent A_p.

* If Agent Ac selects a node for caching, the results of the frequent task are cached on the designated node after execution. Note that the execution node and caching node can be different.

7. Cycle Repetition: The RM processes the next task in the queue, and the cycle repeats.

This dual-agent, RL-driven approach, considering factors like node resources, geographical location, task frequency, execution costs, and caching benefits, aims to dynamically adapt scheduling and caching plans each cycle. The goal is to minimize latency, reduce processing and storage costs, and maximize overall resource utilization in the fog-cloud continuum.

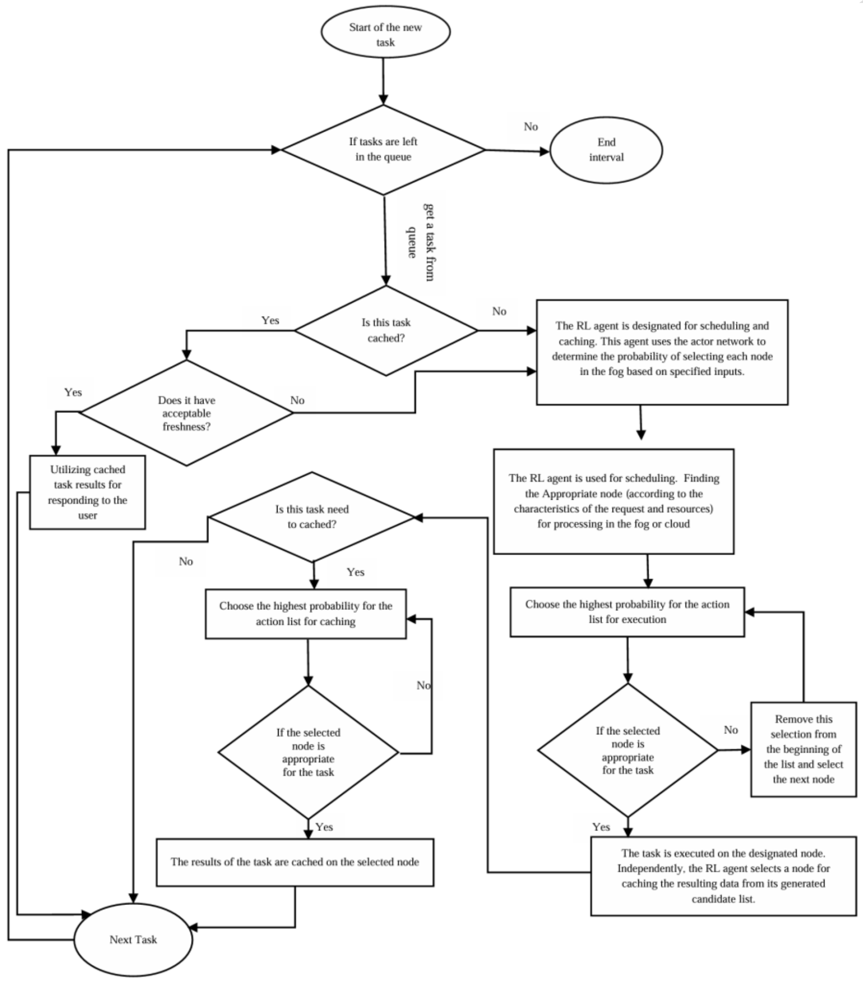

Start of the new task
If tasks are left in the queue
No
End interval
get a task from queue
Is this task cached?
Yes
No
The RL agent is designated for scheduling and caching. This agent uses the actor network to determine the probability of selecting each node in the fog based on specified inputs.
Does it have acceptable freshness?
Yes
No
Utilizing cached task results for responding to the user
The RL agent is used for scheduling. Finding the Appropriate node (according to the characteristics of the request and resources) for processing in the fog or cloud
Is this task need to cached?
No
Yes
Choose the highest probability for the action list for caching
Choose the highest probability for the action list for execution
No
If the selected node is appropriate for the task
If the selected node is appropriate for the task
No
Remove this selection from the beginning of the list and select the next node
Yes
Yes
The results of the task are cached on the selected node
The task is executed on the designated node. Independently, the RL agent selects a node for caching the resulting data from its generated candidate list.
Next Task

Fig. 4: The process of sending a task until receiving a result

4.6. Applicability to Mobile Users and Dynamic Topologies

Fog–cloud environments frequently involve mobile end devices and dynamic networking topologies, where fog nodes may join, leave, or change their neighbor relationships at runtime. Our TSC-A2C framework naturally extends to such settings as follows:

- Dynamic User Proximity: Since both the execution agent and caching agent include user-to-node distance in their state representations (Definition 18), updating a user's 2D location loc(u_k) at each scheduling cycle automatically informs the agents of mobility. As users move, the policy will adapt by favoring fog nodes that are currently nearest, thus maintaining low end-to-end latency.

- Topology Adaptation: The neighbor set $\mathcal{N}(n_j)$ for each fog node can be updated in real time (e.g., via heartbeat or discovery protocols). At the start of each cycle, the RM reconstructs the adjacency information and passes the updated neighbor counts to the caching agent's state (the fourth feature of each node's 4-tuple in Section 4). Consequently, the caching agent learns to select nodes based on both current storage metrics and their dynamic connectivity.
- Online Fine-Tuning: For highly dynamic scenarios, the pre-trained A2C agents can be periodically fine-tuned online using recent observations, ensuring that both execution and caching policies remain effective even as node availability and link qualities fluctuate.

This discussion demonstrates that TSC-A2C's decoupled, state-driven design readily accommodates mobile users and changing fog–cloud topologies without structural modifications to the core algorithm.

5. Evaluation

In this section, we first present a brief description of the dataset, the utilized simulator, and evaluation criteria. Next, we provide a detailed analysis of the comparative results between the TSC-A2C and a number of baseline algorithms, as well as other state-of-the-art algorithms such as the A3C-R2N2 [28], DDQN [25], LR-MMT and LRR-MMT [24] methods.

5.1. The Dataset

For simulating input tasks, we have used the Bitbrain dataset [31], which is an publicly available derived from real-world scenarios. It contains 181,335 tasks with heterogeneous characteristics such as varying CPU (MIPS), memory, and I/O requirements which are fed into the Bitbrain's infrastructure during three weeks. For each task, the following information are available in this dataset: CPU utilization in terms of MIPS, RAM, and disk (read/write) characteristics.

5.2. The Simulator

In order to be able to evaluate the efficacy of the TSC-A2C, we have utilized the iFogSim simulator [32], which is constructed upon the foundation of CloudSim [33]. iFogSim has been chosen due to its provision of valuable APIs pertinent to resource management within the Fog. We have modified the iFogSim so as to encompass a location parameter both for users and Fog nodes.

The infrastructure examined in this investigation is characterized as a heterogeneous cloud-fog environment. Unlike the studies [25], [34], [26], and [27], our research emphasizes fog nodes that are in proximity to the user and possess limited resources. Four distinct types of fog nodes have been considered as being present within this environment. A concise overview of these node types is provided in Table 3). Usage costs given in this table are derived from the Microsoft Azure IaaS cloud service in 2025.

Table 3: Different types of fog nodes, used in simulations

Name	Processor	Core count	RAM	Disk	Usage Cost		
					1 core	1 GB RAM	10 GB
Hitachi HA 8000	Intel i3 3.0 GHz	2	8 GB	250 GB	0.08 $/hr	0.03 $/hr	0.004 $/hr
IBM server x3250	Xeon X3470 3 GHz	4	8	250 GB	0.1 $/hr	0.05 $/hr	0.005 $/hr
DEPO Race X340H	Intel i5 3.2 GHz	4	16 GB	250 GB	0.15 $/hr	0.07 $/hr	0.007 $/hr
IBM server x3550	*Xeon X5675 3067 MHz*	6	16 GB	500 GB	0.2 $/hr	0.1 $/hr	0.01 $/hr
Deel PowerEdge R820	Intel Xeon 2.6 GHz	32	48 Gb	500 GB	2.2 $/hr	1.13 $/hr	0.14 $/hr
Deel PowerEdge C6320	Intel Xeon 2.3 GHz	64	64 Gb	1 TB	4.2 $/hr	2.30 $/hr	0.44 $/hr

The scheduling cycle has been considered to be 5 minutes as in [24], [25], and [34]. In our simulations with the FogBus [24] framework, we adopt the average service invocation latencies reported therein: an end-to-end response time of 10 s for fog gateway nodes and 100 s for cloud datacenters. These values represent the typical round-trip delay—including network transmission and node-level processing—measured under standard IoT workloads. All simulations consist of 1000 Bitbrain tasks, submitted over a period of one day.

5.3. Evaluation Criteria

To evaluate the effectiveness of the proposed TSC-A2C, we have considered the following criteria:

- Total cost: Computed according to the Definition 23
- Total Execution time: Computed according to the Definition 22
- Percentage of SLA violations

- Percentage of tasks executed on the fog

Most of the above criteria has been selected from the [35] and [25].

5.4. Analysis of the Results

To ensure the robustness and statistical significance of our findings, we conducted multiple independent simulation runs for each scenario and configuration presented in Sections 5.3 and 5.4. Specifically, for every data point in Figure (5) and Figure (6), the simulations were repeated 30 times with different random seeds to account for the stochastic nature of task arrivals and resource allocation in the iFogSim environment. This approach allows us to assess the variability of the results and determine the statistical significance of the observed performance differences.

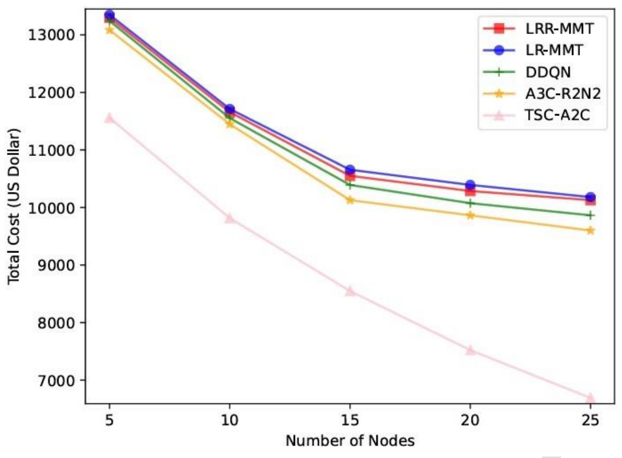
For each evaluation criterion (Total Cost, Total Execution Time, Percentage of SLA Violations, and Percentage of Tasks Executed on Fog), we calculated the mean and standard deviation across these 30 runs for TSC-A2C and all baseline methods (A3C-R2N2, DDQN, LR-MMT, and LRR-MMT).

To statistically validate the superiority of TSC-A2C, we performed independent samples t-tests to compare the mean performance of TSC-A2C against each baseline method for every configuration (i.e., varying number of fog nodes and varying frequent task percentages). The null hypothesis for each test was that there is no significant difference between the mean performance of TSC-A2C and the respective baseline method. A significance level (alpha, α) of 0.05 was used. A p-value less than 0.05 indicates that the observed difference is statistically significant, allowing us to reject the null hypothesis.

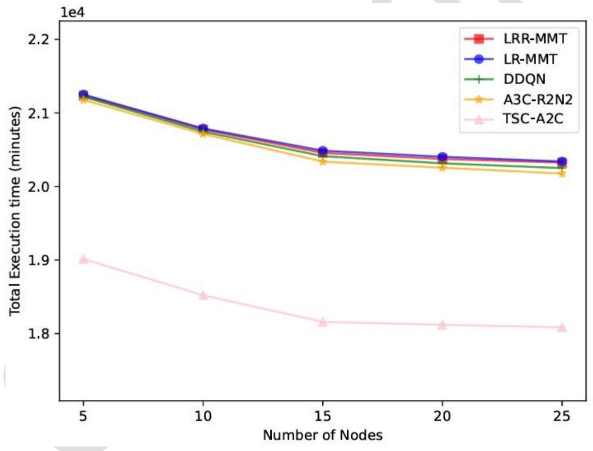
Furthermore, to provide a measure of the precision and reliability of our mean estimates, we computed 95% confidence intervals for all reported performance metrics. A 95% confidence interval indicates that if the experiment were repeated many times, 95% of these intervals would contain the true mean performance of the system. Non-overlapping confidence intervals between TSC-A2C and a baseline method further support the statistical significance of the difference.

5.4.1. Simulation Scenario 1: Changing The Number of Fog Nodes

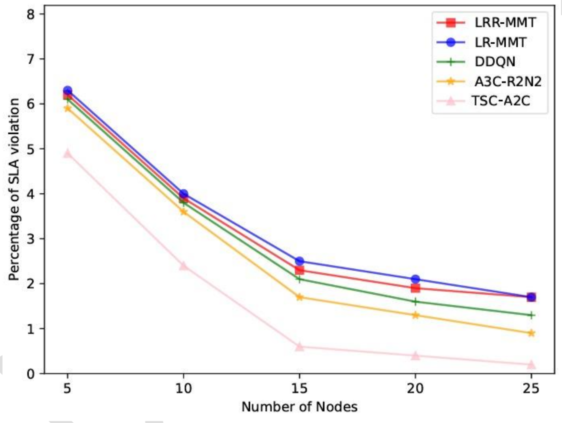
This simulation scenario has been conducted to evaluate the performance of the proposed TSC-A2C method in comparison to other existing methods when the number of fog nodes has been changed. To this end, we have changed the number of fog nodes from 5 to 25 nodes. 10 percent of submitted tasks have been considered to be frequent. Fig. 5) presents the results of this study. As it is shown, the TSC-A2C significantly outperforms other existing methods in terms of all evaluation criteria. This is primarily due to the ability of the proposed TSC-A2C to identify and cache frequent tasks, avoiding their re-executions. Unlike prior approaches, TSC-A2C decouples node selection for execution and caching, enhancing efficiency by better utilizing fog resources and preventing unnecessary cloud offloading. Fig. 5-a) illustrates that increasing the number of fog nodes from 5 to 15 reduces the execution cost for all methods, as more cost-efficient nodes become available. However, increasing the number of fog nodes above 15 does not significantly affect the cost anymore. That is to say, for executing all submitted tasks, we have to pay at least about 10K $, using available node types and their corresponding costs. Fig. 5-b) shows reduced execution time with more fog nodes due to decreased queuing and data transmission delays. Fig. 5-c) demonstrates a decrease in SLA violations with increased fog nodes due to improved load balancing and reduced latency. Finally, Fig. 5-d) indicates that TSC-A2C processes a larger proportion of tasks within the fog as node count increases, effectively leveraging fog resources and handling frequent tasks.



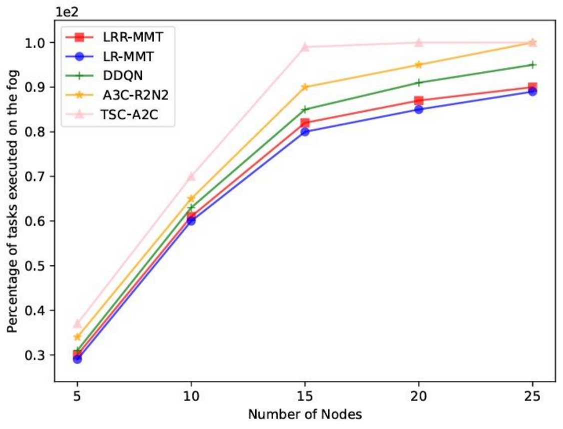
a



B



c

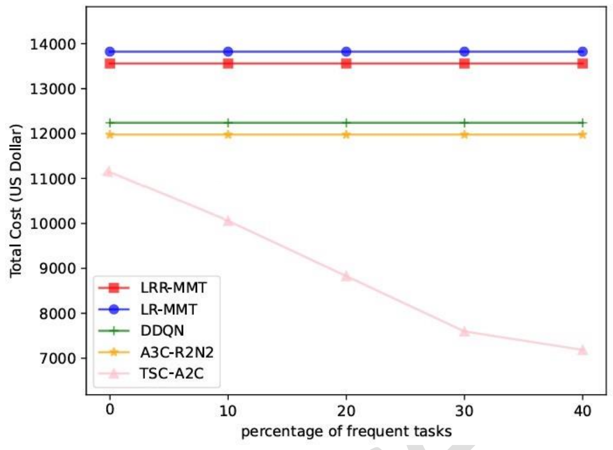


D

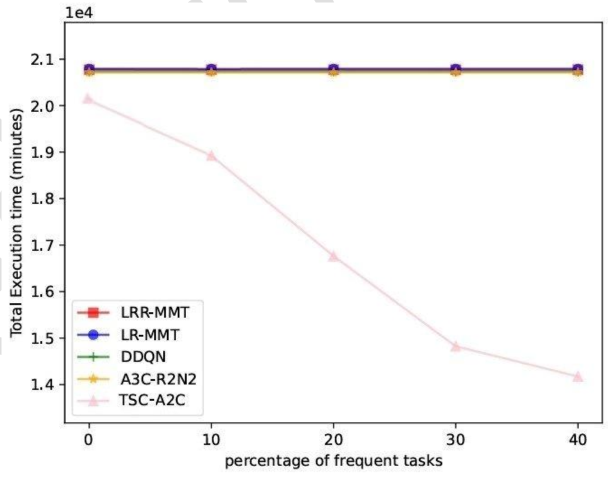
Fig. 5: Comparison of the TSC-A2C method with other methods when the number of fog nodes changes

5.4.2. Simulation Scenario 2: Changing The Frequent Tasks Percentage

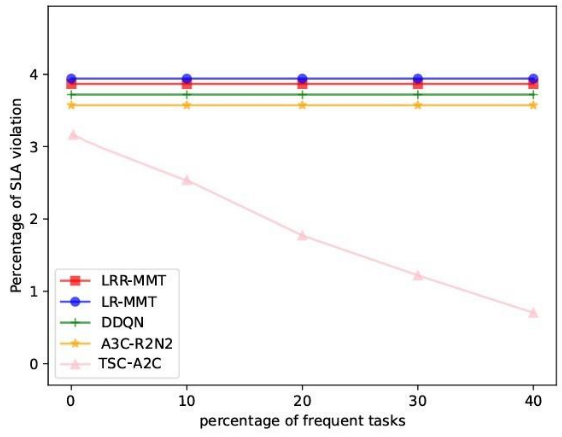
This study has been conducted for performance evaluation of the proposed TSC-A2C method against existing methods under varying percentages of frequent tasks. For that, we have changed the percentage of frequent tasks within the range [0-40]%. The number of fog nodes has been considered to be 10. Results of this study have been depicted in Fig. 6). As it can be seen in Fig. 6-a), TSC-A2C significantly reduces the total cost with increasing frequent tasks, unlike other methods which remain largely unaffected. This cost reduction stems from TSC-A2C's efficient, independent management of frequent tasks through its caching mechanism, minimizing redundant processing. Fig. 6-b) highlights a corresponding reduction in total execution time as frequent tasks increase, due to the reuse of cached results. Fig. 6-c) shows a decrease in SLA violations with higher percentages of frequent tasks, as more tasks are served directly from the cache, reducing cloud reliance and latency. Finally, Fig. 6-d) demonstrates that the percentage of tasks executed within the fog environment grows with increasing frequent tasks, as cached results lessen the workload on fog nodes, allowing the fog to handle more tasks and improve overall system performance.



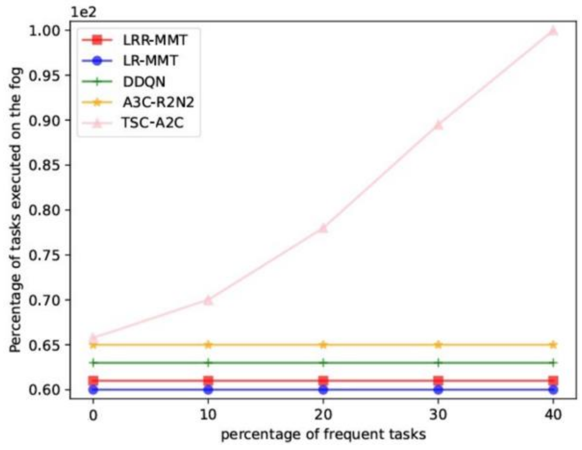
a



b



c



d

Fig. 6: Comparison of the TSC-A2C method with other methods when the percentage of frequent tasks changes

5.5. Analysis of Cache Check Overhead and Its Impact

In our simulation environment (iFogSim), the cache check process, as described in Section 4 (Operational Flow, Step 2), involves a lookup operation within the Resource Manager (RM) to determine if a valid and fresh result for an incoming task exists in any fog node's cache. This operation is fundamentally a data retrieval query across the distributed cache entries.

Evaluation of Cache Check Time: While not explicitly measured as a standalone metric in our reported results, the time required for a cache check was implicitly accounted for within the overall "Total Execution Time" (CTET) metric (Definition 22). In a typical distributed system, a cache lookup operation involves:

1. Local Lookup: Checking the RM's internal metadata or a local index for cache entry pointers. This is generally a very fast, near-constant time operation (O(1) or O(log N) depending on the indexing structure, where N is the number of cached items).
2. Network Latency (if distributed): If the RM needs to query multiple fog nodes to ascertain cache presence or freshness, this would involve minimal network communication overhead. However, given that the RM maintains "uptodate information on node capacities, neighbor sets, and cache contents", this implies a centralized or aggregated view of cache metadata, making the lookup predominantly a local operation at the RM.

In our iFogSim simulations, the overhead of this metadata lookup and decision-making process at the RM is considered negligible when compared to the much larger time scales associated with actual task execution (CPU processing, memory access, disk I/O) and network transmission delays (task submission delay, result dispatch time) across the fog-cloud continuum. The simulation model inherently incorporates the computational cost of these RM operations as part of the overall system overhead, which is reflected in the baseline performance of all methods.

The primary impact of the cache check mechanism is not its own minimal execution time, but rather the significant time savings achieved by *avoiding* redundant computations and network transfers. As demonstrated in Simulation Scenario 2 (Fig. 6)), increasing the percentage of frequent tasks directly leads to: reduced total execution time, reduced total cost, decreased SLA violations.

5.6. Consideration of Energy Consumption

Energy consumption is a paramount concern in modern fog-cloud computing systems, driven by both environmental sustainability goals and the operational costs associated with powering distributed infrastructure [8]. Optimizing energy consumption is a significant objective in task scheduling and resource allocation within fog-cloud environments, leading to reduced operational costs and a lower carbon footprint [9].

While a separate, explicit energy consumption metric (e.g., in Joules or kWh) is not presented in this study, our "Total Cost" metric (Definition 23) serves as a robust and direct proxy for energy consumption within our IaaS-based simulation environment. As detailed in Section 5.1 and Table (3), our costs are derived from Microsoft Azure IaaS pricing, which directly reflects the monetary expenditure associated with the usage of CPU, memory, and disk resources over time. In cloud and fog infrastructures, the consumption of these hardware resources is inherently and strongly correlated with their energy draw [36]. For instance, a strong linear relationship exists between CPU utilization and total power consumption [36]. Therefore, any optimization that reduces the usage or active time of these resources will directly translate into energy savings.

The demonstrated performance improvements of TSC-A2C in "Total Cost" (Figure (5-a, 6-a)) and "Total Execution Time" (Figure (5-b, 6-b)) inherently indicate significant energy efficiency gains. Specifically:

- **Minimizing Redundant Computations:** The history-based caching mechanism for frequent tasks directly reduces the need for re-executing identical tasks. This avoidance of CPU, memory, and network resource usage for repeated tasks inherently translates to lower energy consumption by reducing active processing time and data transfer.
- **Optimizing Resource Utilization:** The dual-agent A2C framework aims to optimally select execution and caching nodes, leading to more balanced load distribution and efficient use of available resources. Better resource utilization can prevent nodes from running at inefficient low-utilization states or from being over-provisioned, both of which contribute to energy waste [37].
- **Reducing Cloud Offloading:** By processing a larger proportion of tasks within the fog environment (as shown in Figure (5-d) and Figure (6-d)), TSC-A2C reduces reliance on distant cloud data centers. While cloud resources are powerful, offloading tasks to them often incurs higher network energy costs and potentially higher overall energy consumption compared to localized fog processing, especially for latency-sensitive tasks.

While iFogSim is capable of modeling energy consumption by considering power usage based on workload and task execution time, our current evaluation primarily focuses on the monetary costs as a comprehensive indicator of resource efficiency in a commercial IaaS context. The significant reductions achieved in "Total Cost" and "Total Execution Time" provide compelling evidence of our framework's energy-efficient nature.

5.7. Computational Overhead and Convergence

While our dual-agent design enhances scheduling and caching, we recognize the need to evaluate its computational footprint and learning behavior. Both actors perform a single forward pass per decision cycle. Each actor network consists of two hidden layers (64 and 32 neurons) and a softmax output layer, running on standard fog-node hardware (Intel i5 3.2 GHz, 8 GB RAM). As shown in Fig. 7), the average decision-making latency per task is presented for TSC-A2C and the comparative methods under various workload conditions. The figure clearly demonstrates that rule-based methods (LR-MMT, LRR-MMT) exhibit the lowest decision-making latency, as expected, due to their deterministic nature and computationally lightweight design. Single-agent deep reinforcement learning (DRL) approaches (A3C-R2N2, DDQN) show slightly higher, yet still very low, latencies, reflecting the inference cost of a single neural network.

10.22060/eej.2025.24181.5657

TSC-A2C, with its dual-agent architecture, incurs a modestly higher decision-making latency compared to single-agent DRL methods—particularly as the proportion of recurring tasks increases (activating both agents). However, this minor computational overhead is justified by a significant reduction in overall execution time, as TSC-A2C dramatically improves efficiency through intelligent result caching that avoids redundant computations. Furthermore, optimal resource utilization and reduced offloading to the cloud lead to lower operational costs and fewer violations of service-level agreements (SLAs). Unlike rule-based approaches, DRL agents are capable of adapting to dynamic and unpredictable workloads, ensuring robust performance in stochastic environments. Therefore, the slight increase in decision latency represents a minimal cost for substantial improvements in overall system performance, efficiency, and adaptability. Tasks are not inappropriately delayed; rather, the system becomes more efficient and responsive overall due to smarter decision-making. The dual-agent architecture inherently reduces total overhead, as the caching agent (Ac) is only invoked for recurring tasks. This means the computational overhead of dual agents is not a fixed cost per task but is conditional. If a task is non-recurring, only the execution agent (Ap) is called. If a task is served from the cache, no agent is invoked at all. This selective invocation highlights the inherent efficiency of the dual-agent design. The system intelligently incurs the "cost of intelligence" only when the benefit is highest—namely, for recurring tasks, where caching can yield significant long-term savings. This design choice is crucial for maintaining efficiency in resource-constrained fog environments and prevents unnecessary computational load for all tasks.

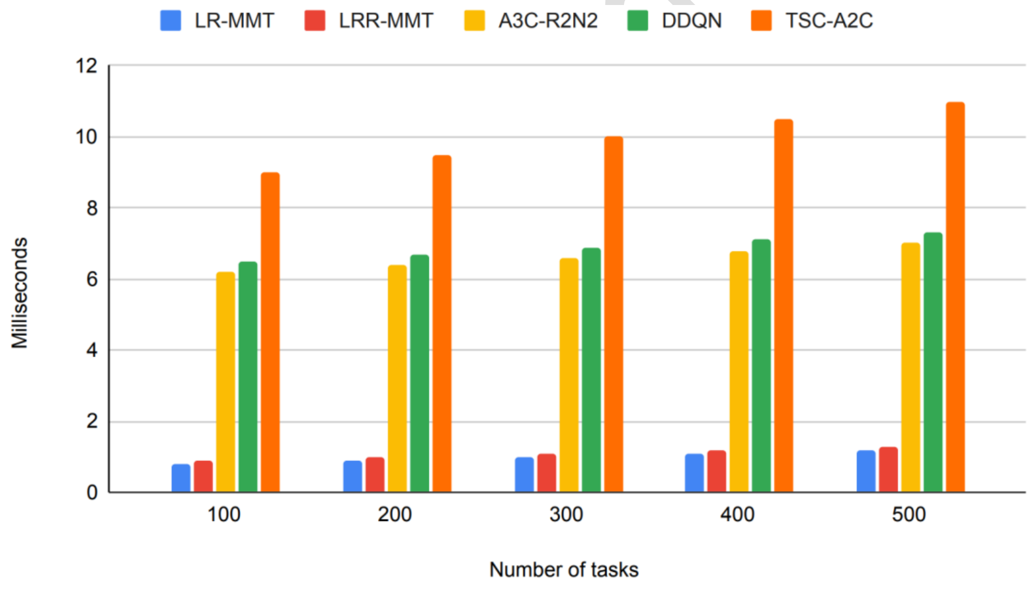


Fig. 7: Average decision-making latency per task

We train both A2C agents offline on a dedicated server (Intel Xeon 2.6 GHz, 32 GB RAM). Training over 20,000 episodes takes approximately 45 minutes for the execution agent and 30 minutes for the caching agent.

10.22060/eej.2025.24181.5657

As can be observed in Fig. 8), the moving-average reward during training for both agents demonstrate convergence after ~12,000 episodes for Ap and ~8,000 episodes for Ac.

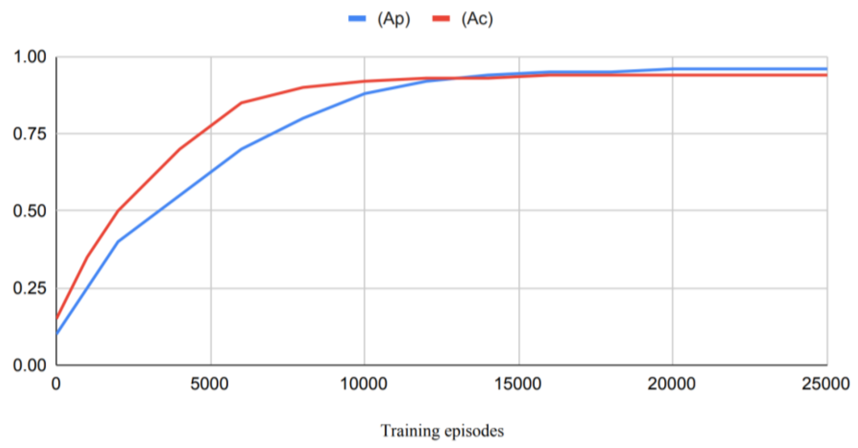


Fig. 8: The moving-average reward

By offloading training to high-performance infrastructure and restricting fog-node operations to lightweight inference, our framework remains efficient for resource-constrained environments without compromising learning quality.

Fig. 9) illustrates how the selection probabilities of different nodes by the execution agent evolve over the course of training.

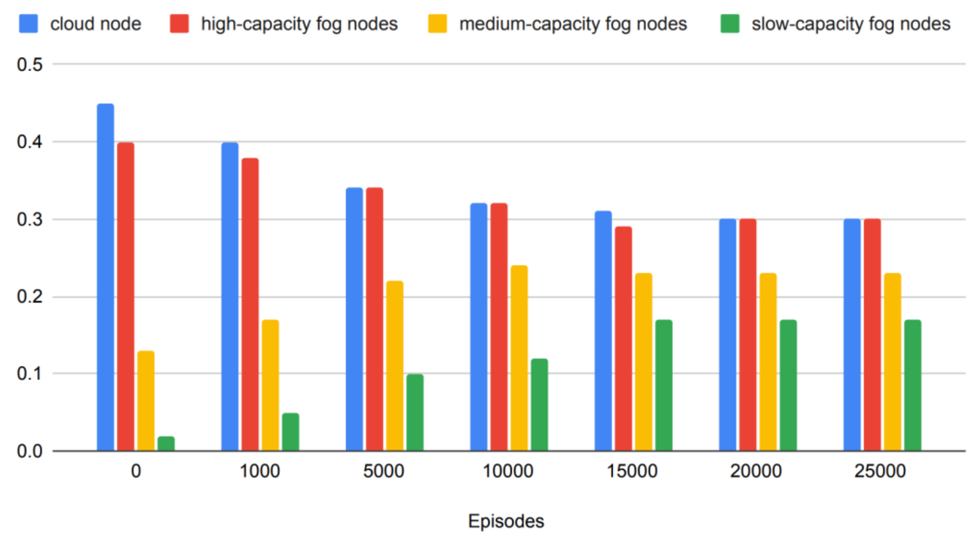


Fig. 9: Selection probabilities of nodes by the execution agent (Ap)

Fig. 9) illustrates the evolution of node selection probabilities by the execution agent (Ap) during the offline training phase. Initially (Episode 0), the agent exhibits a high propensity to select the cloud node (~0.44) and high-capacity fog nodes (~0.40), indicative of an early exploratory phase or a preference for powerful resources. As training progresses, the agent's policy converges: the probability of selecting the cloud node significantly decreases to approximately 0.30, reflecting the agent's learned ability to reduce cloud dependency. Concurrently, the agent learns to strategically utilize various fog node capacities; while the probability of selecting high-capacity fog nodes stabilizes around 0.30, the probabilities for medium- and low-capacity fog nodes increase to approximately 0.23 and 0.16, respectively. This convergence demonstrates that the agent learns a balanced policy for task distribution across the heterogeneous fog-cloud environment, aiming to optimize overall system performance (including cost and latency) while maintaining reliability.

6. Conclusion

The TSC-A2C method demonstrably outperforms baseline and state-of-the-art approaches due to its dynamic adaptability and efficient caching of frequent task results. Unlike static methods, TSC-A2C's reinforcement learning approach optimally adjusts scheduling to dynamic conditions, ensuring consistent performance, especially under high loads. Its dual-agent mechanism, with separate agents for execution and caching node selection, minimizes redundant processing, significantly reducing execution time and costs. By efficiently managing fog resources and optimizing caching, TSC-A2C maximizes fog node utilization and minimizes cloud offloading, enhancing overall system performance. Addressing a gap in prior research, TSC-A2C introduces a joint scheduling and caching strategy that caches task results, eliminating redundant computations. Validated through extensive iFogSim simulations using real-world data, TSC-A2C significantly reduces execution time, costs, and resource consumption while improving scalability. Its intelligent scheduling and caching-aware decision-making provide a robust, adaptive, and efficient solution for dynamic fog-cloud workloads, optimizing resource utilization and enhancing system responsiveness. TSC-A2C represents a crucial advancement for adaptable and sustainable distributed computing infrastructures in evolving fog computing environments. While the current version assumes ideal conditions (e.g., fixed node availability and no failures), incorporating fault tolerance mechanisms such as node failure detection and dynamic task rescheduling is a key direction for future work. Additionally, the architecture's support for spatial reasoning enables seamless extension to mobile user scenarios, which will be further explored in upcoming developments.

To clearly highlight our dual-agent decoupling and history-aware caching, Table 4) summarizes how TSC-A2C differs from leading baselines in terms of execution–caching coupling, caching policy, reinforcement-learning algorithm, and overall decision scope.

Table 4: Comparative Summary of TSC-A2C and Baseline Methods in Terms of Execution–Caching Architecture and Decision-Making

Method	**Execution–Caching**	**Caching Policy**	**Decision Scope**
A3C-R2N2 [28]	Execution	None	Single agent schedules
DDQN [25]	Execution	None	Single agent schedules
LR-MMT [22]	Execution	None	Rule-based
LRR-MMT [23]	Execution	None	Rule-based
TSC-A2C	Execution & Caching	History-aware RL	Separate exec. & cache

REFERENCES

[1] B. Huang, X. Liu, Y. Xiang, D. Yu, S. Deng, S. Wang, Reinforcement learning for cost-effective IoT service caching at the edge, Journal of Parallel and Distributed Computing, 168 (2022) 120-136.
[2] O.A. Khan, S.U. Malik, F.M. Baig, S.U. Islam, H. Pervaiz, H. Malik, S.H. Ahmed, A cache-based approach toward improved scheduling in fog computing, Software: Practice and Experience, 51(12) (2021) 2360-2372.
[3] P. Bellavista, C. Giannelli, D.D.P. Montenero, F. Poltronieri, C. Stefanelli, M. Tortonesi, HOlistic pRocessing and NETworking (HORNET): An Integrated Solution for IoT-Based Fog Computing Services, IEEE Access, 8 (2020) 66707-66721.
[4] C. Mouradian, D. Naboulsi, S. Yangui, R.H. Glitho, M.J. Morrow, P.A. Polakos, A Comprehensive Survey on Fog Computing: State-of-the-Art and Research Challenges, IEEE Communications Surveys & Tutorials, 20(1) (2018) 416-464.

[5] R. Mahmud, S.N. Srirama, K. Ramamohanarao, R. Buyya, Quality of Experience (QoE)-aware placement of applications in Fog computing environments, Journal of Parallel and Distributed Computing, 132 (2019) 190-203.
[6] A. Yousefpour, C. Fung, T. Nguyen, K. Kadiyala, F. Jalali, A. Niakanlahiji, J. Kong, J.P. Jue, All one needs to know about fog computing and related edge computing paradigms: A complete survey, Journal of Systems Architecture, 98 (2019) 289-330.
[7] M. Mukherjee, S. Kumar, Q. Zhang, R. Matam, C.X. Mavromoustakis, Y. Lv, G. Mastorakis, Task data offloading and resource allocation in fog computing with multi-task delay guarantee, Ieee Access, 7 (2019) 152911-152918.
[8] S. Bansal, H. Aggarwal, M. Aggarwal, A systematic review of task scheduling approaches in fog computing, Transactions on Emerging Telecommunications Technologies, 33(9) (2022) e4523.
[9] J. Xu, X. Sun, R. Zhang, H. Liang, Q. Duan, Fog-cloud task scheduling of energy consumption optimisation with deadline consideration, International Journal of Internet Manufacturing and Services, 7(4) (2020) 375-392.
[10] J. Lu, J. Yang, S. Li, Y. Li, W. Jiang, J. Dai, J. Hu, A2C-DRL: Dynamic Scheduling for Stochastic Edge-Cloud Environments Using A2C and Deep Reinforcement Learning, IEEE Internet of Things Journal, (2024).
[11] G. Dong, J. Wang, M. Wang, T. Su, An improved scheduling with advantage actor-critic for Storm workloads, Cluster Computing, 27(10) (2024) 13421-13433.
[12] S. Radhika, S. Keshari Swain, S. Adinarayana, B. Ramesh Babu, Efficient task scheduling in cloud using double deep QNetwork, International Journal of Computing and Digital Systems, 16(1) (2024) 1-11.
[13] X. Sun, Y. Duan, Y. Deng, F. Guo, G. Cai, Y. Peng, Dynamic operating system scheduling using double DQN: A reinforcement learning approach to task optimization, in: 2025 8th International Conference on Advanced Algorithms and Control Engineering (ICAACE), IEEE, 2025, pp. 1492-1497.
[14] X. Zhang, Z. Hu, Y. Liang, H. Xiao, A. Xu, M. Zheng, C. Sun, A federated deep reinforcement learning-based low-power caching strategy for cloud-edge collaboration, Journal of Grid Computing, 22(1) (2024) 21.
[15] Y. Wang, X. Yang, Intelligent resource allocation optimization for cloud computing via machine learning, arXiv preprint arXiv:2504.03682, (2025).
[16] A. Avan, A. Azim, Q. Mahmoud, Agile Reinforcement Learning for Real-Time Task Scheduling in Edge Computing, arXiv preprint arXiv:2506.08850, (2025).
[17] A. Amayuelas, J. Yang, S. Agashe, A. Nagarajan, A. Antoniades, X.E. Wang, W. Wang, Self-resource allocation in multi-agent llm systems, arXiv preprint arXiv:2504.02051, (2025).
[18] Y. Yang, F. Ren, M. Zhang, A Decentralized Multiagent-Based Task Scheduling Framework for Handling Uncertain Events in Fog Computing, arXiv preprint arXiv:2401.02219, (2024).
[19] L. Lu, Y. Jiang, M. Bennis, Z. Ding, F.-C. Zheng, X. You, Distributed edge caching via reinforcement learning in fog radio access networks, in: 2019 IEEE 89th Vehicular Technology Conference (VTC2019-Spring), IEEE, 2019, pp. 1-6.
[20] H. Fabelo, R. Leon, E. Torti, S. Marco, A. Badouh, M. Verbers, C. Vega, J. Santana-Nunez, Y. Falevoz, Y. Ramallo-Fariña, C. Weis, A.M. Wägner, E. Juarez, C. Rial, A. Lagares, G. Burström, F. Leporati, L. Jimenez-Roldan, E. Marenzi, T. Cervero, M. Moreto, G. Danese, S. Zinger, F. Manni, M.L. Alvarez-Male, M.A. García-Bello, L. García, J. Morera, J.F. Piñeiro, C. Bairaktari, B. Noriega-Ortega, B. Clavo, G.M. Callico, STRATUM project: AI-based point of care computing for neurosurgical 3D decision support tools, Microprocessors and Microsystems, 116 (2025) 105157.
[21] S.S. Tripathy, K. Mishra, D.S. Roy, K. Yadav, A. Alferaidi, W. Viriyasitavat, J. Sharmila, G. Dhiman, R.K. Barik, State-of-the-art load balancing algorithms for mist-fog-cloud assisted paradigm: a review and future directions, Archives of Computational Methods in Engineering, 30(4) (2023) 2725-2760.
[22] J. Lim, Versatile cloud resource scheduling based on artificial intelligence in cloud-enabled fog computing environments, Hum.-Centric Comput. Inf. Sci, 13 (2023) 54.
[23] J. Singh, J. Sidhu, Comparative analysis of VM consolidation algorithms for cloud computing, Procedia Computer Science, 167 (2020) 1390-1399.

[24] A. Beloglazov, R. Buyya, Optimal online deterministic algorithms and adaptive heuristics for energy and performance efficient dynamic consolidation of virtual machines in cloud data centers, Concurrency and Computation: Practice and Experience, 24(13) (2012) 1397-1420.
[25] D. Basu, X. Wang, Y. Hong, H. Chen, S. Bressan, Learn-as-you-go with megh: Efficient live migration of virtual machines, IEEE Transactions on Parallel and Distributed Systems, 30(8) (2019) 1786-1801.
[26] H. Mao, M. Alizadeh, I. Menache, S. Kandula, Resource management with deep reinforcement learning, in: Proceedings of the 15th ACM workshop on hot topics in networks, 2016, pp. 50-56.
[27] D. Pathak, P. Krahenbuhl, T. Darrell, Constrained convolutional neural networks for weakly supervised segmentation, in: Proceedings of the IEEE international conference on computer vision, 2015, pp. 1796-1804.
[28] S. Tuli, S. Ilager, K. Ramamohanarao, R. Buyya, Dynamic scheduling for stochastic edge-cloud computing environments using a3c learning and residual recurrent neural networks, IEEE transactions on mobile computing, 21(3) (2020) 940-954.
[29] A. Jesson, C. Lu, G. Gupta, N. Beltran-Velez, A. Filos, J.N. Foerster, Y. Gal, Relu to the rescue: Improve your on-policy actor-critic with positive advantages, arXiv preprint arXiv:2306.01460, (2023).
[30] M. Kölle, M. Hgog, F. Ritz, P. Altmann, M. Zorn, J. Stein, C. Linnhoff-Popien, Quantum advantage actor-critic for reinforcement learning, arXiv preprint arXiv:2401.07043, (2024).
[31] S. Shen, V. Van Beek, A. Iosup, Statistical characterization of business-critical workloads hosted in cloud datacenters, in: 2015 15th IEEE/ACM international symposium on cluster, cloud and grid computing, IEEE, 2015, pp. 465-474.
[32] 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, Software: Practice and Experience, 47(9) (2017) 1275-1296.
[33] R.N. Calheiros, R. Ranjan, A. Beloglazov, C.A. De Rose, R. Buyya, CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms, Software: Practice and experience, 41(1) (2011) 23-50.
[34] M. Cheng, J. Li, S. Nazarian, DRL-cloud: Deep reinforcement learning-based resource provisioning and task scheduling for cloud service providers, in: 2018 23rd Asia and South pacific design automation conference (ASP-DAC), IEEE, 2018, pp. 129-134.
[35] D. Aksu, S. Üstebay, M.A. Aydin, T. Atmaca, Intrusion detection with comparative analysis of supervised learning techniques and fisher score feature selection algorithm, in: International Symposium on Computer and Information Sciences, Springer, 2018, pp. 141-149.
[36] A. Horri, G. Dastghaibyfard, A novel cost based model for energy consumption in cloud computing, ScientificWorldJournal, 2015 (2015) 724524.
[37] S. Sarmad Shah, A. Ali, Optimizing Resource Allocation and Energy Efficiency in Federated Fog Computing for IoT, arXiv e-prints, (2025) arXiv: 2504.00791.