Cluster Computing (2025)28:381
https://doi.org/10.1007/s10586-025-05289-x

Tiny machine learning models for autonomous workload distribution across cloud-edge computing continuum

Mohammad R. Pour-Hosseini[1] Mahdi Abbasi[1,2,3] Atefeh Salimi[4] Erik Elmroth[2] Hassan Haghighi[5] Parham Moradi[6] Bahman Javadi[7]

Received: 24 October 2024 / Revised: 14 April 2025 / Accepted: 18 April 2025

Abstract
Resource management and task distribution in real-time have become increasingly challenging due to the growing use of latency-critical applications across dispersed edge-cloud infrastructures. Intelligent adaptable mechanisms capable of functioning effectively on resource-constrained edge devices and responding quickly to dynamic workload changes are required in these situations. In this work, we offer a learning-based system for autonomous resource allocation across the edge–cloud continuum that is both lightweight and scalable. Two models are presented: TinyDT, a small offline decision tree trained on state-action information retrieved from an adaptive baseline, and TinyXCS, an online rule-based classifier system that can adjust to runtime conditions. Both models are designed to operate on resource-constrained edge devices while minimizing memory overhead and inference latency. Our analysis demonstrates that TinyXCS and TinyDT outperform existing online and offline baselines in terms of throughput and latency, providing a reliable, power-efficient solution for next-generation edge intelligence.

Keywords Internet of things (IoT) · Edge computing · Cloud computing · Workload distribution · Tiny models

1 Introduction

Autonomous vehicles, real-time health monitoring, industrial automation, and smart cities are among the many latency-sensitive and resource-intensive applications that modern edge-cloud computing platforms are expected to support. These hybrid environments, in contrast to conventional cloud-only systems, have to coordinate diverse resources distributed across geographically spread nodes with different network conditions and compute capacities. The effective distribution of computational workloads among edge and cloud nodes is a basic difficulty in this context, especially when there is dynamic, non-stationary demand and limited edge resources.[1, 2]. Inefficient decision-making can result in high response times, wasted resources, or service-level agreement (SLA) violations underscoring the necessity of adaptive, low-overhead resource allocation strategies for edge intelligence. [2, 3].

The Internet of Things (IoT) encompasses a network of devices and physical objects that communicate with each other via the Internet [4]. The data generated within the IoT is transmitted to servers or central systems for controlling

Mahdi Abbasi
mabasi@ipm.ir; mabbasi@cs.umu.se; abbasi@basu.ac.ir

Atefeh Salimi
a.salimi@khuisf.ac.ir

1 Department of Computer Engineering, Bu-Ali Sina University, Hamedan, Iran

2 Department of Computing Science, Umeå University, Umeå, Sweden

3 School of Computer Science, Institute for Research in Fundamental Sciences (IPM), Tehran, Iran

4 Department of Electrical Engineering, Isfahan (Khorasgan) Branch, Islamic Azad University, Isfahan, Iran

5 LIS UMR CNRS 7020, Aix-Marseille University, Marseille, France

6 School of Computer, Data and Mathematical Sciences, Western Sydney University, Sydney, Australia

7 School of Engineering, RMIT University, Melbourne, Australia

Published online: 16 June 2025

and monitoring objects, making decisions, and improving performances. The advantages of IoT include enhanced efficiency, improved communication, and cost reduction [5]. The integration of the IoT into various aspects of daily life has greatly enhanced the quality of life. This technology finds applications in homes, workplaces, travel, and hospitals [6]. It is also utilized in smart homes, smart cities, smart industries, smart agriculture, and the healthcare industry. However, the use of the IoT has presented significant challenges, including security, privacy, data management, compatibility, stability, and scalability. To overcome these challenges, cloud computing was presented as a solution [7]. In cloud computing, processing, storage, and network management functions are moved to centralized data centers [8]. Although the IoT has made progress in overcoming its challenges, the adoption of cloud computing has introduced a new set of issues, including security, privacy, performance, predictability, and scalability.

In 2014, Cisco© introduced the idea of fog computing as a solution to the mentioned challenges [9]. In this concept, computing and storage resources are brought closer to end users, with the aim of processing a portion of the workload locally on fog machines. The deployment of fog machines as computing resources on nearby devices like routers, switches, and sensors allows for the processing and analysis of data at a local level. Fog computing is confronted with several challenges, such as managing computing resources, securing data, scalability, and communication.

Edge computing was introduced to address the challenges faced by fog computing in processing IoT requests. In this architecture, the storage and processing space are located at the edge of the network and close to end users [10]. Figure 1 shows the cloud-fog-edge three-layer architecture. The cloud layer is at the highest level of this architecture. The computing capabilities of robust data centers in this layer are substantial enough to efficiently process and store large amounts of data. The distance between objects and processing devices causes a high latency in information processing. The next layer is the fog layer. This layer is located between the cloud layer and the edge of the network. In the fog layer, the processing and storage devices are closer to the IoT. The reduction of data processing latencies is a feature of this layer. The last layer is named edge. In this layer, processing and storage devices at the edge of the network are close to things and end-users. The edge layer is capable of handling small amounts of data with high performance and minimal latency, whether it's in real-time or near-real-time.

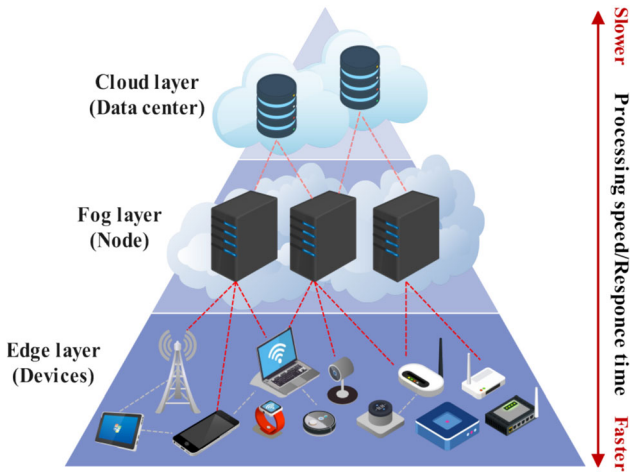


Fig. 1 The cloud-fog-edge paradigm

Although edge computing has advantages like reduced latency, enhanced performance, and more privacy, it also has significant drawbacks, particularly when it comes to resource management and energy saving. Due to their high memory and processing power needs, many task allocation techniques are not practical for edge devices with limited resources. Additionally, controlling power consumption without compromising performance remains a significant challenge, as edge devices often rely on renewable power sources or have limited battery capacity. To solve these problems, intelligent and lightweight task allocation systems that can balance computation efficiency, power consumption, and system latency are required.

The data produced by objects and sensors needs to undergo processing, and the edge layer is capable of handling a portion of this workload. When the computing resources at the edge of the network are not sufficient, the workloads are moved to the fog layer. Routers and switches make up the fog layer, which uses distributed processing to enable fast computing with low latency. If the fog layer becomes congested and unable to handle the workload, it is transferred to a higher layer known as the cloud layer. The cloud layer has significant processing power. Processing workloads in the cloud layer, however, results in higher latency.

Intelligent transportation and self-driving cars can benefit from the use of edge computing. In self-driving cars, various sensors such as cameras and radar collect information about the vehicle's surroundings. This information must be processed quickly so that the car can make the right decisions. If the processing of this information is done in cloud data centers, there will be significant latency, and this can lead to accidents. Edge computing enables sensor information to be processed in close proximity to the vehicle, which decreases latencies and enhances the performance of self-driving cars. When it comes to fast real-time data processing, edge computing is necessary in general. Self-driving cars, healthcare, and manufacturing industries are just some of the fields where this technology can be utilized [11].

To achieve a balanced workload distribution at the network edge, it is necessary to use appropriate algorithms, methods, and architectures. These tools are employed to distribute the workload in a balanced and optimal way among the edge devices [12]. Extensive research has been conducted to distribute the workload to minimize power consumption and latency. Various methods, such as clustering algorithms, reinforcement learning, and heuristic algorithms, have been used in [13]. Due to their high memory and CPU requirements, it can be challenging to implement and maintain these algorithms on edge devices with limited computing resources. Conventional models like ensemble-based classifiers or deep reinforcement learners are frequently impractical due to the limited processing and memory capacity of edge nodes [13]. Despite their modest capacity, tinyML models are designed especially for these settings and provide a feasible compromise between resource efficiency and real-time responsiveness.

Our objective in this article is to propose a solution that integrates machine learning techniques while prioritizing service quality and efficient resource management. This method aims to overcome the limitations of high memory and CPU demands on edge devices. In various domains, machine learning has been increasingly employed as an intelligent approach to forecast future situations and make decisions in order to attain optimal outcomes [12]. Utilizing machine learning for workload distribution at the network edge is highly efficient, as it enables intelligent decision-making to achieve a balanced distribution or route data processing to higher layers for optimal resource management. By considering the limitations of computing resources such as memory at the network edge, the use of tiny machine learning models becomes crucial, as it requires fewer computational resources compared to conventional models.

In our proposed approach, we not only prioritize the quality of service but also consider the constraints of computing resources in edge devices when allocating workloads. To achieve this, a tiny machine learning model based on a decision tree, which is trained offline to incorporate sufficient information about the system future state, is introduced. Furthermore, we employ the XCS algorithm with limited memory consumption as an online method for workload allocation at the edge of the network, without prior knowledge of future system conditions.

This paper introduces TinyXCS, an online reinforcement learning-based task allocation model, and TinyDT, an offline decision tree-based method. These versions are specifically designed to run on conventional edge devices with low processing power. TinyXCS dynamically optimizes workload allocation in real-time, ensuring flexibility to variable workloads, while TinyDT provides low-latency forecasts for known workload trends. These models maintain good performance while significantly reducing memory usage and computational cost, making them ideal for resource-constrained environments.

In contrast to conventional workload allocation methods like BCM-XCS, NSGA, and heuristic-based approaches, which require substantial computer resources, the proposed TinyXCS and TinyDT models are designed to operate well with low power and memory requirements. When compared to standard XCS-based models, TinyXCS achieves a 98% reduction in memory usage, whilst TinyDT needs only 3 KB of RAM to provide efficient task allocation in extremely constrained conditions. These models successfully maintain a balanced trade-off between compute cost and reaction time, allowing for scalability in dynamic edge computing scenarios.

This study demonstrates how integrating reinforcement learning and decision tree-based methods into a lightweight framework significantly improves workload allocation at the network edge. With an emphasis on power efficiency, computational scalability, and latency reduction, the proposed models outperform baseline methods in real-world cloud-edge scenarios. TinyXCS and TinyDT are practical solutions for intelligent task allocation that minimize processing delays and power consumption while ensuring optimal utilization of available resources. The following is a summary of this paper's main contributions:

1. TinyXCS is a suggested online Learning Classifier System (LCS) that can evolve lightweight rules directly on edge nodes and is optimized for adaptive workload distribution in real time.
2. Fast, low-latency inference without continuous training overhead is made possible by TinyDT, an effective offline decision tree model trained using knowledge distilled from online experience (BCM-XCS).
3. In order to improve generalization and efficiency, a knowledge transfer method that connects offline deployment with online learning is described.
4. Superior performance in terms of latency, throughput, and statistical reliability is demonstrated by both online and offline models when tested under simulated situations and contrasted with many online and offline baselines.

By integrating reinforcement learning and decision tree-based strategies into a scalable framework, this study presents a novel, practical, and highly efficient workload allocation solution for edge computing environments. Simulation parameters of this study such as latency limits, high-frequency task arrivals, and energy-aware decision-making align with real-world application areas like autonomous systems, industrial IoT, and video analytics.

The article is organized as follows: Sect. 2 reviews previous work and discusses some related issues. Section 3

reviews system modeling and formulates the problem from different perspectives. A tiny machine learning model (offline) and a learning classification system algorithm (online) are also introduced to distribute workloads at the edges of the network. In Sect. 4, we examine and contrast the proposed approach with previous endeavors. Finally, Sect. 5 of the paper outlines the conclusions and future directions.

2 Related work

In contemporary distributed systems, where computational workloads must be regularly balanced between resource-constrained edge nodes and more potent cloud infrastructures, resource allocation in edge–cloud computing has become an important challenge. Intelligent workload distribution solutions that maximize responsiveness and resource consumption are essential due to the growing prevalence of latency-sensitive and bandwidth-intensive applications, such as real-time analytics, autonomous systems, and smart healthcare. Edge-cloud systems operate in dynamic contexts with time-varying workloads, a variety of device capabilities, and shifting network conditions, in contrast to traditional cloud-centric approaches. As a result, efficient resource allocation algorithms must balance several competing objectives, such as maintaining Quality of Service (QoS), reducing latency, boosting throughput, and conserving power. To meet these demands, recent techniques have employed machine learning and adaptive optimization, enabling real-time, context-aware decision-making throughout the edge–cloud continuum.

Several recent studies have investigated multi-objective optimization strategies. For example, the integration of renewable power into mobile edge computing is explored in [14], using reinforcement learning to optimize power and latency jointly. However, low-capacity edge devices cannot use it due to its processing complexity. Similarly, Kanapram et al. [15] proposes a multi-layer energy model for cloud-fog environments using OpenStack, highlighting the efficiency-energy trade-off, but struggling with urgent demands. A post-decision state-based RL approach in [16] tackles workload allocation efficiency, but it degrades QoS by causing an uneven load across nodes.

Other strategies like the GLOBE algorithm [17] utilize geographical load balancing for edge stations and demonstrate improved system cost savings (50%) but still suffer from high computational demands. By combining balanced initialization, MPSO, and SDP algorithms, the hybrid technique (BRT) proposed by Niu et al. reduces latency at the expense of high power consumption [18]. An NSGA-II-based workload allocator introduced by Abbasi et al. [19] also optimizes power and latency but shows degraded QoS under high query volumes due to its low speed.

Deep reinforcement learning (DRL) has recently become an effective method for resource allocation and job migration in diverse cloud-edge systems [20]. Abbasi et al. [21] presents BCM-XCS, a learning-classifier-based model to reduce latency and power, but still requires offloading to the cloud under limited battery scenarios. Traditional methods based on Lyapunov optimization [22], game theory [23], or heuristic [24, 25]/meta-heuristic algorithms [26–28] are often computationally expensive and poorly adapt to dynamic environments. For example, the MGSAC approach, which was first shown in [29], models structural dependencies and filters unproductive actions by combining Graph Convolutional Networks (GCNs), Soft Actor–Critic (SAC) reinforcement learning, and a learnable mask layer to enhance task scheduling. In real-time edge–cloud situations, it outperforms current techniques by drastically lowering energy usage, response time, migration time, and SLA violations.

Reinforcement learning-based methods are further compared in terms of key objectives—power consumption, response time, migration cost, and SLA satisfaction—across works such as R2N2-A3C [30], GCN-DDPG [31], and multi-task dependency scheduling [32, 33]. But in highly stochastic contexts, these still have limited adaptability and frequently lack tools for examining structural aspects or avoiding poor scheduling choices.

When considered collectively, the research emphasizes the necessity of scalable, adaptable, power-efficient, and QoS-preserving algorithms (Table 1). This encourages more investigation into scheduling under power and latency restrictions, incorporating age-aware metrics, and combining reinforcement learning with structural awareness, particularly in IoT-driven edge environments.

3 The proposed tiny models for workload distribution

As a response to the challenges faced in existing methods, particularly regarding excessive consumption of computing resources and memory, we have devised an efficient solution to enhance network workloads at the edge. Some data streams require quick responses and cannot be sent to cloud systems. Consequently, these computations must be performed at the edge of networks. However, machine learning and workload adjustment systems in edge and fog systems can be resource-intensive. The constrained computing power and memory of devices pose challenges for implementing algorithms or programs that require significant memory allocation for scheduling tasks or assigning workflows. Such memory-intensive operations may not be suitable for execution at the edge due to these limitations. It is recommended to employ lightweight systems for efficient

workload scheduling in edge processing. This approach takes into account the limited computing power and memory of edge nodes, ensuring efficient task allocation and workflow management within these constraints. This article aims to find an efficient solution for workload scheduling at the edge of the networks by utilizing Tiny machine learning models. The focus is on leveraging these models to effectively allocate processing resources and optimize workload distribution in edge environments.

This study focuses on small machine learning models, as they well-suited for computing environments with limited resources. A range of lightweight machine learning models, including neural network variants like TinyML and MobileNet for image recognition, Decision Trees (DTs) for rule-based decision-making, and LCS like TinyXCS for adaptive reinforcement learning, can be installed on edge servers with limited computing resources. Other methods such as Support Vector Machines (SVMs) and Bayesian Networks are also used in edge computing for classification and probabilistic reasoning, but they require further optimization to minimize processing cost [34]. As answers to this issue, we propose TinyDT, a miniature decision tree model, and TinyXCS, a reinforcement learning-based LCS. Both are designed to work effectively on common edge devices.

TinyXCS and TinyDT, the models proposed in this study, consume 98% less memory than standard XCS models and less than 3 KB of memory, respectively, they are ideal for deployment in low-power edge scenarios. Additionally, these models enable on-device inference, which ensures low-latency processing and does away with the necessity of continuous data transfer to the cloud, both of which are critical for real-time applications with short turnaround times. Power efficiency is another important advantage because edge devices usually use renewable power sources or batteries. By optimizing power utilization, TinyXCS and TinyDT provide sustainable operation without compromising performance.

These models can also adapt to shifting workloads; TinyDT provides low-latency predictions for predictable tasks, while TinyXCS employs reinforcement learning to distribute workloads in real time. Lastly, by allowing distributed intelligence over a growing edge device network without overtaxing the system, tiny models enhance scalability.

3.1 System modeling

The architecture of the system is shown in Fig. 2 where the Edge model consists of a base station and a set of co-located Edge servers. The base station is responsible for determining the workload capacity at the edge as well as transmitting the workload to higher layers. Each processing resource at the edge of the network is equipped with a power source of limited capacity. To solve this problem, a shared power source in the network is used to recharge the batteries of these processing resources.

Table 2 briefly presents the system modeling parameters, shown in Fig. 2. The input workloads from end users and objects are denoted as W_t. These workloads are then fed into the decision-making system based on machine learning. This article introduces two models of offline and online decision-making systems.

The offline model uses tiny machine learning models and the future knowledge of the system to determine the number of workloads allocated for local processing in the edge devices () and the portion sent to the higher layer (W_t^H). The decision-making process also takes into account the congestion parameters (C_t) and battery capacity (B_t) of the edge devices. On the other hand, the online model is designed with a learning classification system that considers the limitations of edge resources. This model uses reinforcement learning for real-time decisions on workload distribution between the network edge and upper layers, as detailed in Sections 3.2 and 3.3.

3.1.1 Workload model

In this scenario, the time model is assumed to be discrete ($t = 0, 1, 2, ...$). The input workload at a specific time t in the edge of the network is expressed by W_t. The system determines the portion of this input workload to be processed locally at the edge, denoted by the parameter W_t^L . Therefore, the workload sent to the higher layer is shown by W_t^H. Additionally, the number of active servers during each time interval is denoted by N_t, which ranges from 0 to N. Here, N represents the maximum number of servers available at the edge of the network.

3.1.2 Delay model

There are two types of delay in this system.

- Processing delay of workloads at the edge of the network: this delay depends on the parameters $n(t)$ and $\beta(t)$, representing the number of active servers and the workload processing capacity at the edge of the network, respectively. The calculation of the processing delay for workloads at the edge of the network is determined by the queue management mechanism implemented in the edge servers. This delay is calculated using Eq. (1), where c represents the processing capacity of each server in terms of the number of requests per second.

$$d_{opt}(t) = \frac{\beta(t)}{n(t) \cdot c - \beta(t)} \quad (1)$$

Table 1 Comparison of edge–cloud resource allocation methods

Method (references)	Technique used	Strengths	Limitations	Year
RL with renewable energy [14]	Reinforcement learning for offloading with renewable energy integration	Jointly optimizes latency and energy (minimizes delay+cost), improving edge performance over baselines	High computational complexity, unsuitable for low-capacity edge devices	2017
Multi-layer energy model (Kanapram et al.) [15]	Multi-layer cloud–fog architecture (OpenStack-based)	Balances energy usage and performance via energy–QoS trade-off	Slow response to urgent workloads (struggles with bursty demand)	2017
Post-decision RL [16]	Post-decision state reinforcement learning	Fast learning control for efficient workload allocation	Tends to create load imbalance across nodes, degrading QoS	2023
GLOBE [17]	Geographical load balancing algorithm	Cost-effective operation with energy-harvesting support (□50% system cost reduction)	High computational and resource demands (heavy overhead)	2018
Hybrid BRT (Niu et al.) [18]	BRT (Balanced init+MPSO+SDP)	Reduces service latency and achieves minimum delay in edge nodes	Energy-intensive and high power usage	2019
NSGA-II (Abbasi et al.) [19]	Multi-objective genetic algorithm (NSGA-II)	Good dual optimization of power and delay	Degrades QoS under high query volumes	2020
Secure RA via DRL [20]	Action-constrained deep RL	Improves QoS and security, significant energy and cost gains	Added complexity to integrate security constraints	2025
BCM-XCS (Abbasi et al.) [21]	Learning Classifier System (XCS-based)	Energy-efficient and adaptive	Needs frequent offloading under low battery	2020
Lyapunov-guided scheduling [22]	Deep RL with Lyapunov optimization	QoS-aware and real-time decisions	High training complexity and overhead	2025
Two-layer game (He et al.) [23]	Two-level Stackelberg game-theoretic framework	Stable equilibrium for multi-agent resource competition	Needs accurate utility modeling and limited to predictable scenarios	2025
H-STEP (Chi et al.) [24]	Heuristic stable edge service placement	Improves latency for VR task placement	No dynamic scheduling, limited to placement	2025
Hybrid NOMA-RSMA [25]	Joint NOMA/RSMA resource allocation	Energy-efficient for IIoT applications	Complex parameter configuration	2025
Nature-Inspired Metaheuristics [26]	Evolutionary algorithms (GA, PSO, etc.)	Flexible and general-purpose optimization	Performance sensitive to tuning; resource intensive	2025
Context-aware clustering [27]	Device clustering+metaheuristics	Effective for smart healthcare NB-IoT	Extra overhead from clustering and iterative search	2025
Metaheuristics on Smartphones [28]	Mobile metaheuristic scheduling	Validated on real smartphone workloads	Not scalable to cloud-level loads	2025
MGSAC [29]	GCN+SAC+Learnable Mask	Reduces delay and energy; SLA aware scheduling	Model complexity is high	2025
R2N2-A3C [30]	Multi-agent A3C RL	Improves task throughput and cost	Limited scalability with many tasks or agents	2025
GCN-DDPG [31]	Graph neural net+DDPG	Learns task dependencies for smarter offloading	Resource-intensive training and inference	2025
DynMap [32]	Heuristic dynamic mapping	Efficient CGRA multi-task allocation	Architecture specific, needs tuning	2025
Multi-robot RL scheduling [33]	Decentralized RL	Reduces mission time via agent coordination	Needs retraining for unseen environments	2025

RA, Resource Allocation; DRL, Deep Reinforcement Learning; PSO, Particle Swarm Optimization; NSGA-II, Non-dominated Sorting Genetic Algorithm II; NOMA, Non-Orthogonal Multiple Access; RSMA, Rate-Splitting Multiple Access; GCN, Graph Convolutional Network; SAC, Soft Actor–Critic; DDPG, Deep Deterministic Policy Gradient; CGRA, Coarse-Grained Reconfigurable Architecture

- Delay of sending the remaining workloads to the higher layer: this delay is represented by $d_{opt}(t)$ and depends on the congestion condition in the network. The symbol $C(t)$ represents network congestion, encompassing both the traffic within the network and the delay experienced for transmitting and receiving information within the fog layer. Therefore, the delay in transmitting the remaining workloads to the higher layer, which depends on the value of $C(t)$ is given by equation (2).

$$d_{tra}(t) = (\alpha(t) - \beta(t))C(t) \tag{2}$$

Therefore, the total delay is obtained as:

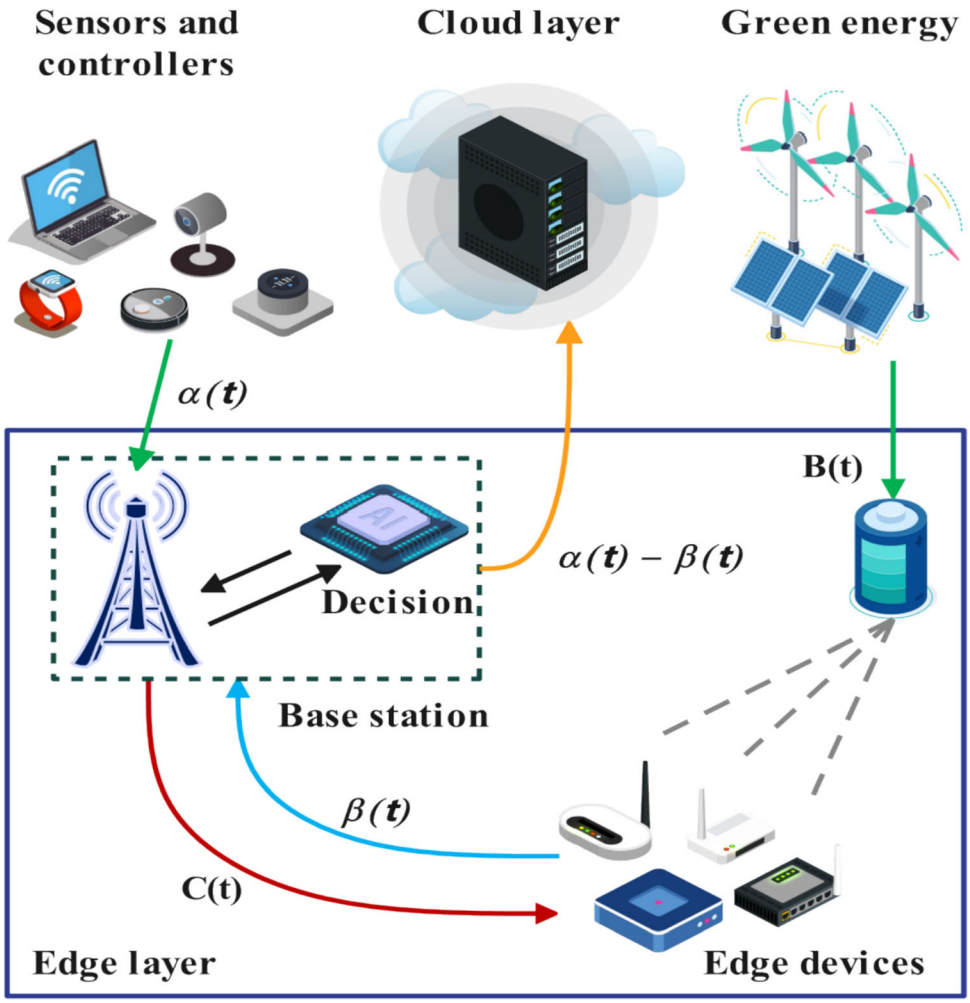


Fig. 2 The overall model of the system

Table 2 Modeling parameters

Description	Symbol
The input load	$\alpha(t)$
The workloads processed locally at the edge	$\beta(t)$
The cost of using a backup power supply	$c_{back}(t)$
The total renewable power received	$P_{gr}(t)$
The congestion status of the network	$h(t)$
The total power consumption	$P_{sum}(t)$
The battery level at the network edge	$B(t)$
The system status	$S(t)$
The latency in processing workloads on the network edge	$d_{pro}(t)$
The latency in sending workloads to the fog	$d_{tra}(t)$
The total delays	$d_{sum}(t)$
The number of active servers on the network edge in each time-slot	$m(t)$
The power consumption for operational tasks on the network edge	P_{op}
The power consumption for processing loads on the network edge	P_{pro}

$$d_{sum}(t) = d_{pro}(t) + d_{tra}(t) \tag{3}$$

3.1.3 Power consumption model

Similar to the delay, the power consumption is divided into two categories.

- Power needed for basic operations and transferring workloads at the network's edge:
  This power is determined by the input workloads at the network's edge and affected by fixed power consumption $P_{fixed}(t)$ and variable power consumption $P_{dyn}(t) = f(\alpha(t))$. Therefore, the power required for basic operations and transferring workloads at the network's edge is given by Eq. (4).
  $$P_{op}(t) = P_{fixed}(t) + P_{dyn}(t) \tag{4}$$
- Power required to process workloads at the network's edge: this power is determined by the workload capacity and the number of active servers as given by Eq. (5).
  $$P_{pro}(t) = n(t) \cdot \beta(t) \tag{5}$$
  Therefore, the total power consumption is given by Eq. (6).
  $$P_{sum}(t) = P_{pro}(t) + P_{op}(t) \tag{6}$$

3.1.4 Battery model

Each edge server has a battery denoted as $B(t)$, which has a limited capacity. These batteries are charged by solar or wind power. The processing workloads at the network's edge are dependent on the amount of servers' battery capacity. Therefore, two situations are taken into consideration for the batteries.

- Insufficient battery capacity:
  The battery capacity is insufficient to perform basic operations and transfer workloads at the network's edge. Therefore, the backup power supply is used to transfer workloads to the higher layer. Here, ϕ is the cost coefficient of the backup power supply, as given by Eq. (7). Additionally, the power level of the batteries is charged with renewable power ($P_{gr}(t)$) to facilitate processing in the subsequent time slot, as given by Eq. (8).
  $$C_{back}(t) = \emptyset.P_{op}(t) \tag{7}$$
  $$B(t+1) = B(t) + E_{gr}(t) \tag{8}$$
- Exceeding minimum power required:
  The battery's power level exceeds the minimum required for basic operations. Hence, a portion of the workload is processed at the network's edge, while the remaining workload is offloaded to a higher layer. Consequently, the battery's power level for the next interval is given by Eq. (9).
  $$B(t+1) = B(t) + P_{gr}(t) - P_{sum}(t) \tag{9}$$

TinyXCS places a high priority on running workloads on edge nodes with excess renewable power, guaranteeing optimal use of green power sources while lowering backup power expenses and dependency on batteries.

3.1.5 Markov decision processes (MDP)

MDP plays a crucial role in workload optimization in TinyXCS, as they enable adaptive resource allocation depending on current system conditions. TinyXCS can optimize performance and power consumption while dynamically adjusting task distribution by modeling decision-making as an MDP. An MDP consists of a set of different states s and a set of possible actions (A) in each state. The transition model is $P(s^{'}|\ s,a)$, where s is the current state, $s^{'}$ is the future state of the system, and a is the action applied to the environment. To minimize the system costs, the cost at the edge of the network should be reduced in each time slot, so we use MDP and Bellman's value function equation [35, 36].

$$V(s) = \min_{a \in A} \left\{ R(s,a) + \gamma \sum_{s'} P(s'|s,a).V(s') \right\} \tag{10}$$

where, $V(s)$ represents the value of the state s, $R(s,a)$ represents the immediate reward of doing action a in the state s. Also, $P(s^{'}|s,a)$ indicates the probability of entering the state if action a is performed in state $s^{'}$. Here, γ is a discount factor and a constant value that shows that future costs have less weight than current costs.

TinyXCS employs MDP as a framework to guide decision-making, leveraging n adaptive learning for workload distribution. The integration of MDP enables TinyXCS to dynamically select the best classifiers, thereby enhancing system responsiveness and resource efficiency. Equations (1) through (10), which define state, action, transition, and reward function, are part of the Markov model for the workload allocation scenario at the network edge.

(a) State Representation (s):
The MDP system state at time **t** is represented as:
$$s(t) = (m(t), d(t), B(t)). \tag{11}$$

(b) The action set consists of workload allocation decisions, denoted as A(t)={$\beta(t)$, α(t)- $\beta(t)$}.

The current state s(t) and the chosen action α(t) are used by the transition function to determine the future state s(t+1). As shown by Eqs. (9), (1) to (3), and (11) respectively, it comes after battery level update, delay update, and workload handling.

The reward function is designed to minimize system latency, power consumption, and resource costs:

$$R(s,a)(t) = \mu_1 B(t) - \mu_2 d_{sum}(t) - \mu_3 C_{back}(t), \mu_i > 0 \quad (12)$$

where, $d_{sum}(t)$ is total system delay (processing+offloading), and $C_{back}(t)$ is the backup power usage cost. The coefficients μ_1, μ_2, μ_3 regulate the trade-off between power, latency, and cost.

Optimizing task execution while taking power efficiency and system responsiveness into account is the main objective of workload allocation in edge computing. The following is the definition of the optimization objective:

$$\max \sum_t R(s,a)(t) \quad (13)$$

Tasks are sent to the cloud when edge devices' workload or power limitations are exceeded. This happens when

$$B(t) \leq P_{op}(t) ord_{sum}(t) \langle d_{max} or C_{back}(t) \rangle C_{max} \quad (14)$$

In such cases, the cloud processes the workload to maintain uninterrupted execution and prevent edge node failures.

TinyXCS's feedback-driven operation addresses this MDP, allowing the system to continuously learn and improve workload distribution strategies. By dynamically adapting to changing edge conditions, TinyXCS ensures real-time performance optimization, reduced operational costs, and efficient resource utilization. TinyXCS is preferably suited for dynamic and resource-constrained computing settings, building on this MDP-based workload optimization approach.

3.2 TinyXCS: a learning classifier system for adaptive workload allocation

Learning classifier system functions as an intelligent agent interacting with its surroundings. Its adaptability lies in its ability to choose the optimal action based on the current state of the environment. This trait improves with each new experience gained from reinforcement learning and the feedback (payoff) provided by the environment. The primary objective of an LCS is to maximize the environmental payoffs it receives [35].

The LCS focuses on developing and refining a comprehensive set of" condition-*action-payoff*" rules, referred to as classifiers. Each classifier informs the system in each state (or equivalently the input condition) about the number of payoffs for any available action. Therefore, LCS can be seen as a particular model of reinforcement learning that offers an approach to generalization with a well-defined information representation. The idea of LCS was initially introduced by Holland to enhance genetic algorithms [37]. Wilson and his colleagues introduced a novel variant called XCS. Unlike its predecessors, XCS determined the applicability of classifiers solely based on the accuracy of the performed actions [38]. The primary goal of XCS is to encapsulate the predictive patterns within the environment through classifiers. By doing so, the system can achieve optimal payoffs. This unique approach has led to widespread applications of XCS in real-world scenarios, spanning fields such as control, robotics, and data mining [39–41] . The key distinction that sets XCS apart from most other classifier systems is illustrated in Fig. 3 [42].

In XCS, the fitness of rules for the Genetic Algorithm (GA) isn't solely based on the received payoff (ρ), but rather on the accuracy of predictions concerning these payoffs. The primary objective here is to create a comprehensive and precise mapping of the problem space. Instead of merely concentrating on the higher payoff regions in the environment, XCS focuses on efficient generalizations. This approach aims to establish a thorough and accurate understanding of the problem space, showcasing a departure from traditional methods. In essence, XCS clarifies the connection between LCS and reinforcement learning. It provides a method of applying conventional reinforcement learning techniques to intricate problems where the number of potential state-action combinations is very large. During

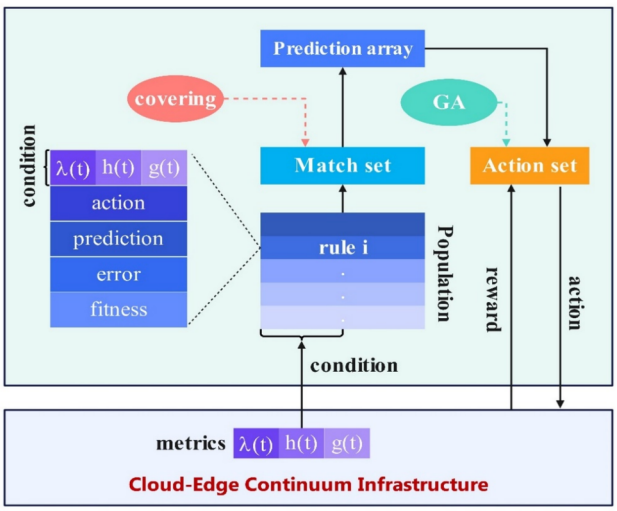


Fig. 3 The schematic of XCS

every time slot, a match set is generated within the system. Subsequently, a system prediction is computed for each action within the match set ([M]). These predictions are determined through a fitness-weighted average of the predictions derived from the rules within the action set ([A]). The system's action is then chosen, either deterministically or randomly, with a typical probability of 0.5 per trial. In cases where the match set ([M]) is empty, covering is employed to handle the situation.

Input: condition // $\alpha(t), C(t), B(t)$
Output: action for workload allocation in the edge layer // $\beta(t)$

1: *input* ← state of the environment
2: M ← CreateMachSet(P)
3: PA ← BuildPredictionArray(M)
4: A ← BuildActionSet(action selection strategy, PA)
5: *action* ← SelectAction(A)
6: *environment* ← Perform(*action*)
7: *environment* ← Reward()
8: P ← previous reward + discount factor × max prediction
9: UpdateSet(P, previous_ action_ set)
10: **if** (flag and UseGenericAlgorithm(*action_ set*)) **then**
11: GenericAlgorithm(A, previous input)
12: **end if**
13: previous action set ← A
14: previous reward ← environment of reward

Fitness reinforcement involves updating three parameters ε, ρ, and F for each relevant rule. The process of updating fitness is based on the relative accuracy of the rule within the set and involves five distinct steps: In the first step, the error of each classifier is updated.

$$\varepsilon_j = \varepsilon_j + \lambda\left(\left|\text{reward} - \rho_j\right| - \varepsilon_j\right) \tag{15}$$

The predictions of each classifier are then updated:

$$\rho_j = \rho_j + \lambda\left(\text{reward} - \rho_j\right) \tag{16}$$

In the next step, the classifier's accuracy k_j is computed. The values of v, τ, and ε_0 are constants that control the shape of the correctness function.

$$k_j = \tau\left(\frac{\varepsilon_0}{\varepsilon}\right)^v \quad \text{or} \quad k_j = 1 \text{ if } \varepsilon < \varepsilon_0 \tag{17}$$

A relative accuracy $k_j^{'}$ is determined for each rule by dividing its accuracy by the total of the accuracies in the set:

$$k_j' = \frac{k_j}{\sum_{x\in[A]} k_x} \tag{18}$$

The relative accuracy is then used to adjust the classifier's fitness. If the fitness has been adjusted $1/\lambda$ times, as

$$F_j = F_j + \lambda\left(k_j' - F_j\right) \tag{19}$$

Otherwise, F_j is set to the average of the current and previous values of k'_j.

Algorithm 1: TinyXCS algorithm for workload allocation in the edge layer

In XCS, the fitness value is inversely proportional to the reward prediction error. This means that errors smaller than this specific value do not alter the fitness value. For each action present in the match set, the classifier predictions ρ are weighted by the fitness F to calculate the system's net prediction in the prediction array. This weighted approach allows the system to integrate both the accuracy of predictions and the corresponding fitness values, ensuring a balanced and effective decision-making process.

TinyXCS is a lightweight, reinforcement learning-based LCS designed for dynamic and autonomous workload distribution at the edge of computing networks. Unlike traditional XCS, which suffers from high computational and memory overhead, TinyXCS optimizes workload allocation while keeping memory and processing demands minimal, making it suitable for deployment on resource-constrained edge devices.

The implementation steps of TinyXCS are determined in Algorithm 1. The initial step involves receiving information from the environment and utilizing the population set ([P]) to generate the match set ([M]) (lines 1 and 2). The prediction array is created according to the ε, ρ, and F parameters, and if needed, the covering operation is

performed (line 3). Next, the best action with the highest value is selected and applied to the environment (lines 4 and 5). Then, by receiving the reward from the environment, the value of [P] is calculated, and the set [A]-1 is updated (lines 6–8). The function UseGenericAlgorithm() is utilized to determine whether the genetic algorithm needs to be employed (lines 10–13). Finally, the values of A are saved and subsequently cleared for future steps (lines 13 and 14).

As explained above, the performance and efficiency of TinyXCS are highly dependent on specific design parameters, including population size, mutation rate, crossover strategy, and memory constraints. To clarify their impact, a detailed analysis of these parameters and the reasoning behind their selection.

3.2.1 Population size ([p])

The total number of classifiers in TinyXCS determines the system's ability to make effective workload distribution decisions. Traditional XCS models typically require large classifier populations, ranging from 1000 to 5000 rules, which leads to significant memory and computational overhead. In contrast, TinyXCS optimizes this by reducing the classifier population to just 200–500 rules, significantly lowering memory consumption while preserving learning efficiency. A smaller population size accelerates rule matching and decision-making, making TinyXCS well-suited for real-time edge computing. Empirical analysis indicates that increasing the number of classifiers beyond 500 yields diminishing returns, making larger populations unnecessary in resource-constrained environments. To compensate for the smaller population, TinyXCS effectively generalizes rules using wildcard-based conditions (#), which enhances adaptability while minimizing memory usage.

3.2.2 Genetic algorithm

GA plays a fundamental role in TinyXCS by driving the evolutionary process that enables the system to optimize workload distribution over time. Unlike conventional machine learning models that require extensive labeled datasets, TinyXCS leverages GA to autonomously evolve and refine classifiers, making it suitable for dynamic and non-Markov environments. The crossover and mutation mechanisms in GA allow the system to adapt to changes in workload distributions while maintaining computational efficiency. GA operates in two crucial areas of TinyXCS: rule evolution and memory management. In rule evolution, generalization and specialization techniques enable the model to cover a broader range of scenarios while ensuring accuracy. In memory management, GA helps in identifying and preserving the most effective classifiers by periodically updating rules, eliminating redundant ones, and refining decision-making through adaptive reinforcement learning.

By incorporating controlled mutation rates and memory-efficient rule evolution strategies, TinyXCS ensures low-latency workload scheduling, optimized power consumption, and scalable deployment in resource-constrained edge environments. The integration of GA into TinyXCS provides a robust and adaptive approach to workload allocation, enabling the system to continuously evolve and improve performance while minimizing resource usage. A few key strategies for this important TinyXCS module are examined and are discussed below.

3.2.2.1 Mutation The mutation rate in TinyXCS represents the probability of introducing random changes into a classifier's rule set during genetic operations. Traditional XCS models typically use mutation rates between 0.04 and 0.1 to balance exploration and exploitation. However, TinyXCS adopts a lower mutation rate, ranging from approximately 0.02–0.05, to minimize excessive classifier churn, ensuring memory stability and efficient rule evolution. Lower mutation rates help maintain system stability by preventing frequent disruptions to learned rules, which is particularly important in memory-constrained environments. In contrast, higher mutation rates can lead to frequent replacements of well-performing classifiers, increasing computational overhead—an approach that is not feasible for resource-limited edge computing scenarios.

3.2.2.2 Crossover The crossover strategy in TinyXCS is the mechanism responsible for recombining two parent classifiers to generate offspring classifiers, ensuring continuous evolution and adaptation. In TinyXCS, 12 bits are selected from one parent and 12 bits from another, forming a 24-bit condition for new classifiers. This process follows a probabilistic uniform crossover approach, which helps maintain diversity in the classifier population while preventing excessive memory growth. Additionally, TinyXCS incorporates elitism, a strategy that preserves top-performing classifiers while evolving suboptimal ones, ensuring that the system does not lose high-quality rules during evolutionary updates. The selection of this crossover strategy is justified by its ability to enable gradual evolution, ensuring classifier accuracy without unnecessary rule expansion. By leveraging uniform crossover, TinyXCS systematically refines rules over time while avoiding the generation of excessive, redundant classifiers. The elitism mechanism further strengthens the model by ensuring that well-performing rules are retained, preventing degradation in decision quality and maintaining long-term efficiency.

3.2.2.3 Fitness computation Another critical component in TinyXCS is fitness computation, which determines how

well a classifier predicts future environmental states. In traditional XCS, fitness is based on absolute payoff values, meaning that the classifiers with the highest direct rewards are favored. However, TinyXCS adopts an accuracy-weighted fitness approach, prioritizing classifiers that consistently make accurate predictions, regardless of immediate reward values. This approach ensures that rules generalize effectively across diverse workloads without unnecessarily consuming memory resources. By avoiding overfitting to particular circumstances, accuracy-based fitness calculation has the advantage of increasing TinyXCS's adaptability to dynamic and shifting workload distributions. TinyXCS can maintain long-term learning stability by emphasizing prediction accuracy above raw rewards. This enables it to function effectively in intricate, real-world edge computing contexts where conditions change over time.

Additionally, the fitness update process has been made more efficient by employing a memory structure that stores the most significant condition-action pairs and the associated fitness values. Instead of recalculating fitness for each condition-action pair, we only update the fitness of condition-action pairs that contribute to the task's successful completion (i.e., reaching the goal). Evolutionary algorithms are also used to efficiently generate new condition-action combinations. This guarantees that in addition to being simplified, the rule base will change over time to optimize efficiency. This approach simplifies the fitness calculation and removes unnecessary computations by focusing on a subset of rules that directly affect the system's performance.

3.2.3 Memory management

Operating under fundamental presumptions and memory management techniques to maximize performance while reducing resource consumption, TinyXCS is made for effective task allocation in resource-constrained edge contexts. One fundamental premise is that edge devices have just 3–5 MB of RAM available for task scheduling, and their processor and memory capabilities are constrained. Population size, mutation approach, and memory pruning techniques are all directly impacted by this limitation, necessitating careful optimization to preserve system performance without overloading it. Furthermore, TinyXCS makes the assumption that workload patterns are fairly predictable, which means that past workload distributions might guide present and future actions and enhance learning effectiveness. TinyXCS uses a FIFO-based pruning technique to avoid redundancy and only save the most effective rules in order to efficiently manage memory. A fitness-weighted policy governs classifier retention, giving priority to reliable and often used classifiers. In other words, TinyXCS avoids needless memory growth, facilitates scalable workload allocation, and preserves efficiency in resource-constrained edge contexts by consistently improving the rule base and getting rid of redundancy. These strategies ensure that TinyXCS remains scalable, efficient, and well-suited for real-time edge computing applications by minimizing computational load, enhancing adaptability, and preventing memory bloat.

3.2.4 Power efficiency

TinyXCS boosts power economy by enhancing its LCS with a genetic algorithm that only evolves the most important rules. This strategy reduces unnecessary computing cycles during continuous task allocation by removing low-fit classifiers early in the learning phase.

In addition to reducing computing complexity, TinyXCS features a flexible and power-efficient workload distribution system. This approach dynamically distributes tasks across edge nodes based on their processing power and power levels. TinyXCS emphasizes scheduling jobs on nodes with excess power when renewable power sources like solar or wind power are available, increasing the battery life of other networked devices. By ensuring that power-intensive operations are carried out where power consumption is lowest, this intelligent workload management helps to avoid unnecessary reliance on cloud computing, which frequently results in expensive transmission power costs.

Furthermore, TinyXCS actively learns workload distribution techniques that maximize power savings and performance. It diminishes dependency on battery-stored power by prioritizing processing activities on nodes with plentiful solar or wind power by utilizing renewable power data. With this dynamic method, system efficiency is maintained while edge computing workloads function with little environmental impact. Finally, TinyXCS uses lightweight, event-driven genetic operations instead of complete retraining to continuously update its rule set. Real-time learning on edge nodes is made possible by these changes, which are dependent on performance metrics and occur with minimal CPU overhead.

3.3 TinyDT: a decision tree-based model for edge computing

The decision tree is used for analyzing and processing data. The decision tree model operates hierarchically, making decisions and evaluating conditions based on a branching structure [43]. The process of building a decision tree involves initially analyzing the dataset to identify the optimal division criterion. Once determined, the tree is divided into separate branches based on this criterion. The process continues until we reach the leaf nodes, which represent the final results. The resulting decision tree can be used to

Table 3 Summary of a sample workload allocation

Example task	CPU load (%)	Memory usage (%)	Battery level (%)	Task allocation
1	80	90	20	Device A
2	30	40	90	Device B
3	70	50	80	Device C
4	85	95	10	Upload to Cloud
5	40	80	15	Device A
6	90	85	30	Upload to Cloud
7	25	35	95	Device B
8	75	55	60	Device C
9	95	98	5	Upload to Cloud
10	50	70	50	Device C

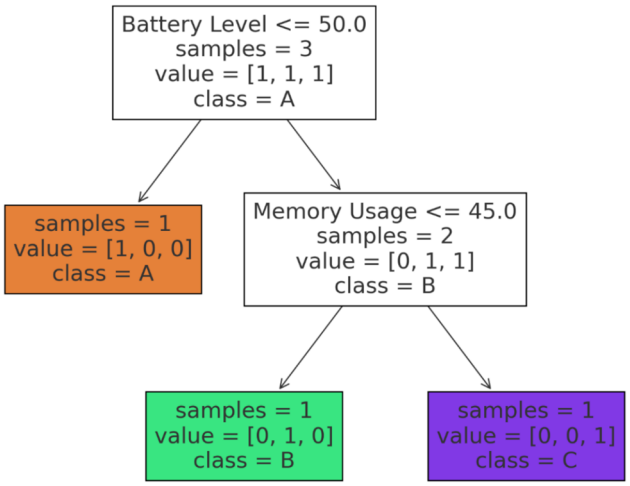


Fig. 4 The workload allocation decision tree corresponding the sample tasks in Table 3

predict or categorize data based on the specified criteria in its construction.

Depending on the edge devices' current system characteristics, such as CPU load, memory consumption, and battery level, the workload allocation decision tree dynamically divides computational duties among them. In edge computing contexts, effective job distribution reduces power consumption, maximizes resource use, and avoids system overloads. To ensure appropriate job assignment, the decision tree employs a hierarchical decision-making process wherein CPU load is assessed first, followed by memory utilization and battery level.

In the workload allocation scenario shown in Table 3, 10 jobs are dynamically allocated to either Device A (0), Device B (1), or Device C (2) according to the availability of resources in real time. The task is transferred to the cloud if resource limitations—like a high CPU load, excessive memory usage, or dangerously low battery levels—prevent local execution (e.g., Examples 4, 6, and 9). By ensuring ongoing task execution, this cloud offloading mechanism ensures operational efficiency and protects against system failures caused by resource exhaustion.

The decision-making process for workload allocation based on decision tree is illustrated in Fig. 4, where tasks are assigned based on system conditions. The decision tree prioritizes Battery Level as the primary criterion:

- Device A is given the duty if the battery level is less than 50%, guaranteeing that it will continue to run even with low power levels.
- The system assesses memory usage if the battery level is greater than 50%:
  - If Memory Usage is less than 45%, the work is sent to Device B, which has sufficient memory resources (for examples, see Examples 2 and 7).
  - Device C, which is designed to manage larger memory needs (e.g., Examples 3, 8, and 10), is given the task if Memory Usage>45%.

In order to maximize performance and avoid device overload in edge computing contexts, this structured allocation technique balances local execution and cloud offloading, ensuring effective resource use.

This workload distribution technique combines adaptive decision-making and real-time monitoring to achieve a compromise between local execution efficiency and cloud scalability. The Upload to Cloud method enhances system resilience by ensuring that critical processes are completed even when edge resources are fully utilized. This approach maximizes task execution latency, power conservation, and system reliability, making it suitable for large-scale edge computing installations.

In practical applications, this decision tree helps edge computing systems distribute the load among available devices in a dynamic manner to optimize performance and resource usage. The tree uses features like CPU load, memory utilization, and battery level to ensure that each device is assigned tasks that are within its capabilities. This

improves the efficiency and sustainability of the system. When resources are few, such as in IoT networks, smart cities, or industrial automation, where edge devices usually operate with varying workloads and resource constraints, this method works well.

By analyzing the new task data (such as CPU, memory, and battery conditions) following training, the decision tree can be used for real-time workload allocation. The tree allocates the task to an appropriate device as it traverses through the nodes according to the feature values. The task is uploaded to the cloud if no device is appropriate. The decision tree efficiently assigns jobs in edge computing scenarios according to device conditions, guaranteeing peak performance and economical use of resources.

The decision tree is employed to distribute the workloads at the network's edge. The decision tree proves to be an effective approach in distributing and managing workloads among the edge network's nodes. By analyzing the new task data (such as CPU, memory, and battery conditions) following training, the decision tree can be used for real-time workload allocation. The tree allocates the task to the right device as it traverses through the nodes according to the feature values. The task is uploaded to the cloud if no device is appropriate. The decision tree efficiently assigns tasks according to the device conditions in edge computing scenarios, guaranteeing peak performance and optimal use of resources.

- Making decisions based on conditions: the decision tree utilizes the conditions and features of edge devices to make informed decisions regarding workload distribution, considering the unique characteristics of each node. These decisions are made by appropriately assigning weights and ensuring a balanced workload distribution among the nodes.
- Quick response to changes: the decision tree enables quick updates and modifications to the decision structure. The decision tree structure can be updated with changes in the network and workload conditions and re-decisions about the workload distribution.
- Optimization of resource consumption: through the utilization of the decision tree and effective workload distribution, it becomes possible to optimize resource consumption within the network. This optimization results in enhanced efficiency and reduces costs such as workload transfer delay and power consumption reduction at the network's edge.

Algorithm 2 shows the decision tree for workload distribution at the network's edge. During the initial phase, the input parameters for training the model to the decision tree are determined. This information includes input workloads, network congestion, the battery's power level, and workloads processed at the network's edge. These factors are used to predict and make future decisions. The proposed decision tree should be optimal and manageable in terms of memory consumption.

Algorithm 2: TinyDT algorithm for workload allocation in the edge layer

Input: data // $\alpha(t),\beta(t),C(t),B(t)$
Output: prediction and decision for workload allocation in the edge layer

```
TinyDT ← MaxDepth and MinSamples // memory management parameters in tree
Define Node Class:
        Attributes:
            label: label of the Node
            children: list of subtrees
            decision: decision index
if TinyDT then
        for all x in data do
            label ← CreateLabel(x)
        end for
        Node(label) ← label
end if
best decision index ← FindingBestDecision(data)
root ← new Node (None)
root.decision ← best decision index
subtrees ← SplitDataSet(data, best decision index)
for all subtree in subtrees do
            child ← DecisionTreeTrain(subtree, TinyDT)
        root.children.append(child)
end for
```

3.3.1 Memory optimization techniques in TinyDT

To manage memory consumption in the proposed model, we use MaxDepth and MinSamples parameters to manage tree depth and remove less important branches (line 1). Then, we establish the initial values required for constructing the tree nodes (lines 2 to 5). These values comprise the label of the node, the list of its children, and the decision index. Based on the constraints for the decision tree and the given data, the label of each tree node is generated at each step using the CreateLabel function (lines 7 to 12).

The FindingBestDecision function attempts to identify the optimal decision within the proposed model in terms of the provided information and key indicators for workload allocation (line 13). The FindingBestDecision function handles the main goals, including delay and minimal power consumption. Next, the unlabeled root node is initialized, and the decision index is determined using the aforementioned function (lines 14 and 15). Then, based on the index and data of the problem, additional subtrees are constructed using the SplitDataSet function in the subsequent steps (line 16). TinyDT is constructed based on subtrees and constraints for efficient management of tree memory. Ultimately, a decision tree is constructed by incorporating a collection of root nodes and their corresponding child's nodes to handle all problem data.

Notably, TinyDT may be trained using a variety of dataset labels, including expert labeling, threshold-based heuristics, synthetic ground truth in simulated environments, and BCM-XCS (as done here).

When the decision tree reaches a node in the TinyDT model, it first assesses if the number of samples at that node is equal to or more than the MinSamples criterion. The node is classified as a leaf and no additional division takes place if the sample count drops below this predetermined threshold. On the other hand, the algorithm determines the most important feature for splitting if the sample count reaches or exceeds the threshold. Entropy serves as a guide for this procedure, guaranteeing that the most instructive features are chosen for node splitting.

Furthermore, we have described how tree pruning approaches reduce memory usage and overfitting, allowing the decision tree model to operate effectively with a limited 3 KB memory limit while maintaining classification accuracy.

Settings like MaxDepth and MinSamples have a big impact on TinyDT's memory usage. By restricting the number of layers, the MaxDepth option prevents the decision tree from becoming overly complex and consuming excessive amounts of memory. TinyDT keeps the tree small and memory-efficient by controlling its depth, preventing needless complexity. As previously stated, the MinSamples parameter also determines the minimum number of samples required to split a node. This prevents the formation of too fine-grained branches, that would raise memory usage without appreciably improving decision accuracy, by ensuring that nodes with insufficient data are not split further.

TinyDT intelligently prunes branches and controls tree depth to optimize memory and power efficiency. A shorter tree structure reduces the number of decision assessments, which lowers CPU cycles and, consequently, power consumption. In addition, by ensuring that only the most important splits remain, pruning removes unnecessary processing cost.

The MinSamples parameter for the TinyDT implementation is set to 2, which means that a node cannot be further split until it contains at least two samples. By raising this threshold, fewer splits occur, resulting in a shallower tree structure that maximizes computing efficiency and memory use. Higher thresholds, however, can result in less detail in decision-making, which may have an impact on categorization accuracy. Because it directly affects memory use and computational cost, finding the ideal balance between tree depth and split count is particularly significant in edge contexts with limited resources.

TinyDT does not require retraining once it has been trained to conduct inference. Its small tree structure supports efficient workload allocation on constrained devices by demanding less than 3 KB of RAM and introducing minimum CPU overhead.

The presented tiny models maintain decision quality in an environment with a highly fluctuating workload, despite their tiny size. These models are capable of handling complexity in constrained edge situations by combining distilled offline information with adaptive reinforcement learning.

4 Performance evaluation and simulation results

The proposed TinyXCS method is implemented on a system equipped with a dual-core processor operating at 1.8 GHz frequency and 2 GB memory with C++ language. Furthermore, the TinyDT is implemented on the same system using the Python language and trained on the dataset acquired from the BCM-XSC method. In this section, we first present the dataset and initial values assigned to the system parameters, and then the evaluation results are examined.

4.1 Dataset

The values utilized for simulation parameters are presented in Table 4. This study's dataset is identical to that published in [14]. Because it accurately depicts mobile edge

Table 4 Values for simulation parameters

Parameter	Parameter definition	Value
$\alpha(t)$	Workload arrival rate	[10 210] request/s
λ_i	Maximum workload of each server per ms	20 request/ms
$h(t)$	Network congestion	[10 20] ms
M	Maximum number of edge servers	10
K	Maximum number of cores in each edge Server	20
$g(t)$	Green power	N (520, 150) w
$\phi(t)$	Backup power usage cost factor	0.15
e_{sta}	Static power of base station	300 w
Ω	Operating cost of batteries	0.01
B_{max}	Maximum battery capacity	2000 w

computing settings, especially in situations requiring workload offloading, autoscaling, and energy harvesting, this dataset was chosen.

The dataset used in this study includes key system parameters that are critical to power efficiency and workload allocation in edge situations. The workload arrival rate fluctuates dynamically between 10 and 210 requests per second, reflecting real traffic conditions, while the network congestion delay fluctuates between 10 and 20 ms.

The renewable power available at each edge node in this simulation is referred to as green power. It is represented as a normally distributed variable with a mean of 520 W and a standard deviation of 150 W. When available, edge servers are powered by this input, which is obtained from renewable sources like solar panels. The system uses backup grid power, which has a greater operating cost, when green power is not enough. Our suggested approaches' decisions on workload distribution are heavily influenced by the availability of green power.

The importance of efficient power management is highlighted by the fact that the maximum battery capacity is fixed at 2000 W with a starting charge of zero.

The edge computing infrastructure consists of up to 10 edge servers, each with an active power consumption of 150 W and a maximum processing capacity of 20 requests per second. The cost considerations additionally include a battery operation cost per unit of 0.01 and a backup power supply cost coefficient of 0.15 to guarantee a realistic economic model for resource allocation.

This dataset, which combines network congestion models, dynamic workload fluctuations, and oscillations in renewable power, provides a comprehensive benchmark compared to previous studies. The synthetic data or small-scale empirical traces employed in many competing studies sometimes fail to depict the complexities of real edge computing environments.

TinyDT is developed on the same machine using the Python programming language and trained on the dataset acquired using the BCM-XCS technique for experimental consistency. It's important to keep in mind that TinyDT is not by default dependent on BCM-XCS. Any meaningful dataset that captures the state of edge devices (such as CPU load, memory usage, and battery level) and the choices made about task distribution can be used to train TinyDT. In the future, we intend to refine the labeling technique by employing a range of sources, such expert systems, heuristics, or hybrid approaches, to improve generality.

4.2 Simulation scenario

A regulated and realistic simulation environment was created by carefully selecting the initial conditions. Active power management was required because the base station's power consumption was set at 300 W and the battery level was started at 0 W. Both models were adjusted to operate under strict memory constraints, and TinyDT consumed less than 3 KB.

The system setup and simulation workload reflect typical edge application settings, such real-time sensor analytics or industrial control loops, where tasks must be performed under strict latency and energy limits while arriving continually. For this purpose, the workload arrival rate varied randomly between 10 and 210 requests per second to replicate dynamic edge situations. A normal distribution was used to depict the availability of renewable power in order to convey realistic changes. The idea behind this setting was to simulate realistic edge-cloud dynamics, by incorporating fluctuating workload arrival rates, renewable power variability, and network congestion delays (parameterized by h(t) in Table 4). Furthermore, anytime resource limitations like excessive CPU load, low battery, or excessive memory consumption are breached, the TinyDT model implicitly initiates fallback to cloud processing (see Table 3 and Fig. 4).

By closely simulating real edge computing challenges, these criteria were developed to provide important insights into system efficiency. By reducing memory usage without compromising speed, TinyXCS was able to achieve a 98%

reduction in memory utilization while maintaining power consumption comparable to BCM-XCS. In contrast, TinyDT used less than 3 KB of ultra-low memory and outperformed all baseline techniques.

The simulation utilizes Basic form of XCS and BCM-XCS algorithms as introduced in [21, 44], to compare with the performance of proposed tiny models: TinyXCS and TinyDT. In both proposed methods, referred to as TinyXCS and TinyDT, we have imposed limitations on memory consumption to manage the resource usage of the algorithm effectively. There is a trade-off between average delay cost, power, and memory usage by reducing memory consumption. Based on our simulations a comparison is made between the results of the original XCS and BCM-XCS algorithms and the results obtained from TinyXCS and TinyDT.

In the following analysis, it is evident that although the TinyXCS algorithm achieves a 98% reduction in memory consumption, there is only a slight difference in power consumption compared to BCM-XCS. However, it shows an improvement compared to the basic XCS algorithm. Also, the TinyDT algorithm has an outstanding performance compared to other baseline algorithms. This can be attributed to the TinyDT algorithm's remarkable performance, which has primarily low memory consumption of less than 3 kilobytes. Additionally, the algorithm's comprehensive understanding of the current and future conditions of the system enable it to make optimal decisions based on these conditions.

While the current simulation comprises 10 edge servers with varying load circumstances, evaluating the system under a broader range of device counts and task intensities is still a crucial next step for proving scalability and robustness.

4.3 Power consumption

Figure 5 illustrates the average power consumption of the system in processing workloads and renewable power for three LCS models: XCS, BCMXCS, and TinyXCS, along with the decision tree model (TinyDT).

The power consumption is measured in W. In this figure, we can see the basic XCS model requires more average power to process workloads at the edge compared to the power provided by renewable resources. Therefore, it can be inferred that a combination of power from batteries and renewable sources is essential for processing workloads. In contrast, TinyXCS shows a reduction in power consumption starting from time slot 4000 compared to the basic XCS. Following this time slot and system learning, TinyXCS results in a noticeable 10% difference in power consumption compared to the BCM-XCS. This amount of consumed power is almost equal to the renewable power at the edge of the network.

Figure 5 shows a pattern where BCM-XCS continually uses less power than TinyXCS beginning at time slot 100 and power consumption starts to plummet at time slot 4000. This pattern is caused by TinyXCS's learning curve and memory restrictions. TinyXCS initially explores workload allocation possibilities, which leads to a bit more power consumption than BCM-XCS, which benefits from a larger classifier set. TinyXCS stabilizes and optimizes task allocation after time slot 4000, reducing unnecessary processing and prioritizing the usage of renewable power, which leads to a significant reduction in power consumption. This demonstrates how TinyXCS can handle workloads with power efficiency despite having a smaller memory footprint.

As a result, TinyXCS successfully distributes workloads optimally while using fewer resources. As a result, it makes it possible to use green power resources more effectively, which lowers battery usage and effectively meets power demands. Green power is the quantity of renewable power (like solar) that the edge servers can use, which influences the decisions that they make about workload distribution and activation.

Furthermore, the TinyDT model outperforms all XCS models in terms of power consumption. This can be attributed to the offline training of the system which enables it to make optimal decisions in workload allocation. As a result, the TinyDT model effectively minimizes power consumption while ensuring efficient workload management. During the initial phase till the time interval of 500, the power consumption of the TinyDT decision tree reaches its peak. This happens because of the tree's construction, which requires a significant amount of power. During this time, the power graph of the TinyDT model shows a consistent and minor fluctuation. Eventually, TinyDT reaches its peak power consumption which is equivalent to the BCM-XCS method. The key highlight of these results is the model's capability to prioritize green power sources for meeting the system's power demands. This approach ensures the

According to the simulation results, all three XCS models demonstrate a rapid charging of the battery level within a short time slot. One notable observation in the TinyXCS model is the significant increase in the battery level which exhibits a consistent upward slope after 1000 slots. By the end of the execution, the battery level sur passes 1400 W.

In the XCS model, the battery level stabilizes at 800 W after a 4500 time-slot and remains constant. However, the battery level in the BCM-XCS model eventually reaches 1400 W. The reason for the initial fluctuations is that the population [P] memories are empty. As time passes and

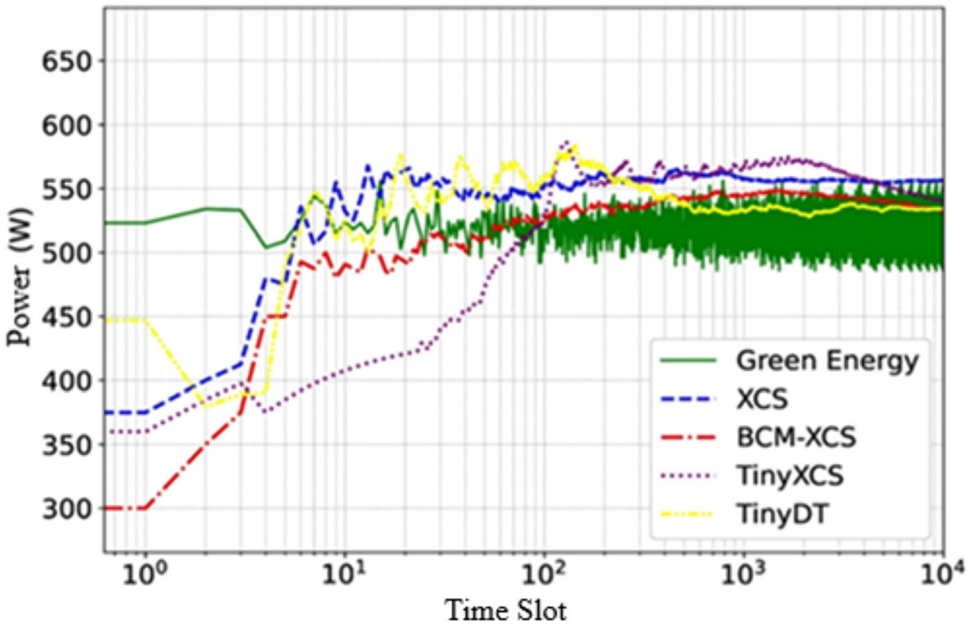


Fig. 5 Relationship between green power availability and system performance metrics

memories are filled, the frequency of random mode selection by the learning classification system decreases. Consequently, at each stage, the probability of selecting the predicted and more appropriate actions increases, leading to a consistent charging process for the battery. In the basic XCS model, the battery level reaches a stable state as random clusters are gradually added over time.

In the BCM-XCS model, the probability of selecting the optimal category from the BCM memory increases by adding optimal categories into it. Due to the limited memory capacity in the TinyXCS model, there is a higher amount of power stored in the batteries. In the TinyDT model, the batteries are charged after 500 time slots and maintain a relatively constant level of approximately 1400W during the time. This indicates minimal battery consumption in the decision tree model. The proposed TinyDT model demonstrates superior performance in terms of battery consumption and storage compared to the basic and memory XCS models. As depicted in Fig. 6, it exhibits greater efficiency in battery saving and charging compared to the proposed TinyXCS model starting from the time interval of 5000. The chat in this figure illustrates how various approaches divide duties while managing power efficiency.

By making wise decisions in real time, TinyXCS minimizes overall power use by dynamically adjusting to the availability of renewable power. Due to its training on suitable patterns, TinyDT replicates similar behavior even when it is offline. Both perform noticeably better than learning-based and heuristic baselines in preserving minimal power consumption, particularly in situations with limited power.

4.4 Latency

In Fig. 7, the average latency is presented for four models: XCS, BCM-XCS, TinyXCS, and TinyDT. The simulation includes 10,000 time slots and the latency is measured in milliseconds. In XCS-based models, the population set is initially not completely filled.

Therefore, due to the initial emptiness of the population in XCS-based methods, the average battery level in the initial 1000 intervals experiences fluctuations that diminish over time. As the number of time slots increases, XCS-based systems have the capability to select a more optimal option to apply to the environment based on the rewards they receive from the environment.

When the population of XCS-based systems is filled with randomly generated classifiers, the learning process slows down. The processing delay is approximately a fixed value before the time slot 6000. Hence, the average latencies in the basic XCS gradually decrease and converge to 4.5 milliseconds over the 6000th time slot.

For both the BCM-XCS and TinyXCS models, the average latencies exhibit a declining trend over the 1500 time slots, involving the receipt of new inputs and continuous learning. Ultimately, the average latencies stabilize at 3.2 milliseconds for the BCM-XCS model and 3.7 milliseconds for the TinyXCS model. As illustrated in Fig. 7, the latency of the decision tree model decreases over time and eventually matches the delay of the BCM-XCS model, converging to approximately 4.5 ms.

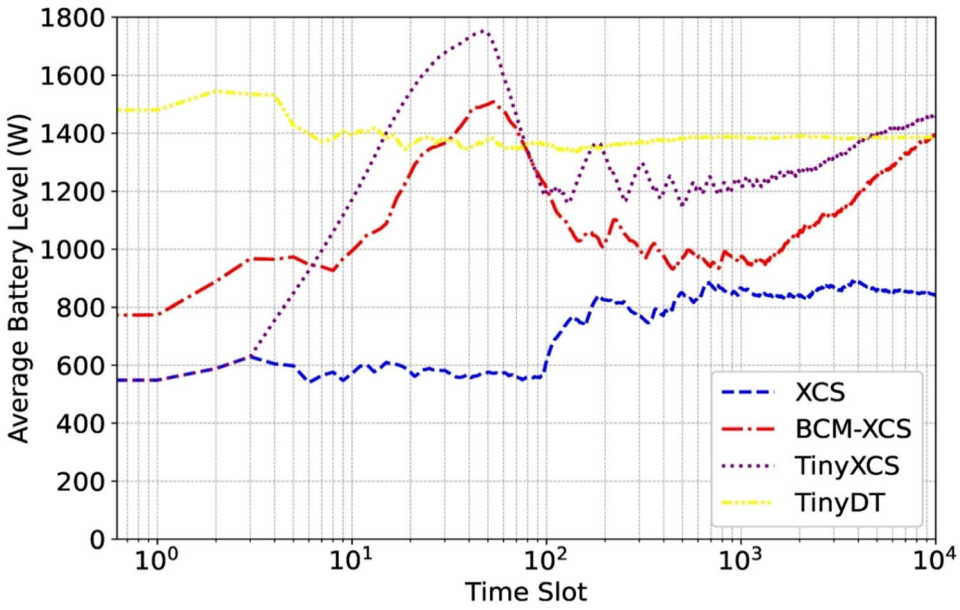


Fig. 6 The average battery level at network edge for different models

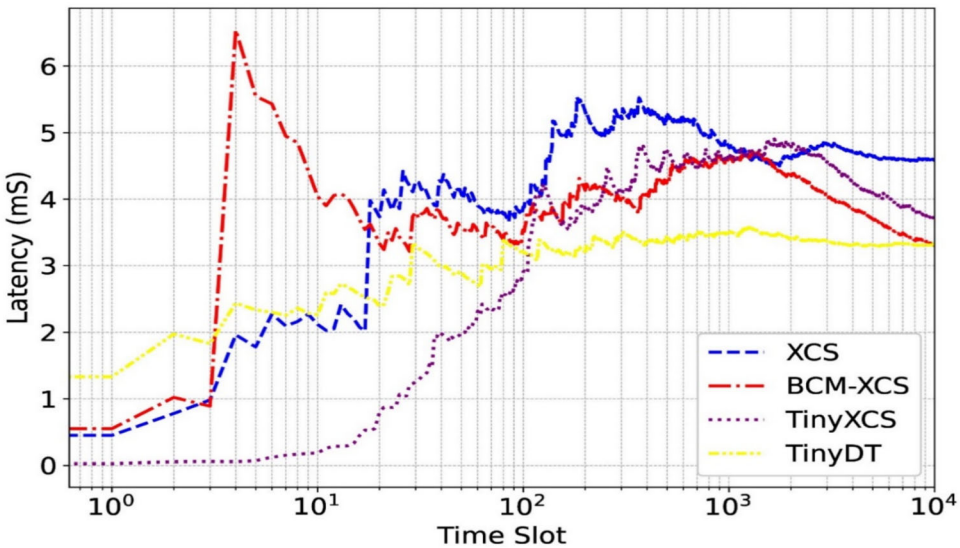


Fig. 7 The latency of different methods

4.5 Memory consumption

Table 5 compares the memory footprint of the proposed tiny models (TinyXCS and TinyDT) with a range of baseline methods, highlighting their suitability for deployment in resource-constrained edge devices.

As shown by Table 5, TinyDT has the lowest memory usage at just 3 KB and is therefore ideal for severely restricted edge devices. TinyXCS, with 3.9 MB of memory usage, achieves a practical balance between real-time adaptability and efficient memory management through its bounded classifier and memory structures.

Among the baseline methods, Fixed 1K, Fixed 4K and Myopic are the lightest (between 4 and 16 KB), but are not able to adapt to the dynamic nature of the edge. GA and BCM-XCS consume 4.6 and 5.2 MB respectively, as they rely on population management and memory-enhanced evolution. NSGAII with its multi-objective optimization and Pareto selection requires 7.5 MB, while Random Forest (RF) with 9.2 MB memory consumption reflects the overhead of such offline models.

TinyDT has little inference overhead after deployment, whereas TinyXCS uses a small rule population and selective genetic updates to handle runtime adaptation with limited CPU and memory effect. Thus, TinyDT is the most memory-efficient method, whereas TinyXCS is a good compromise for adaptive online decision making in memory-constrained scenarios.

4.6 Comparison with baseline methods

Figures 8, 9, 10 and 11 show the results of the performance evaluation, that contrasts the suggested models TinyXCS (online) and TinyDT (offline) with a wide range of baseline techniques that have been carefully categorized according to their operational behavior and learning strategies:

- Online methods: TinyXCS, GA, NSGAII, and BCM-XCS are members of this group. They all interact with the environment to continuously modify their allocation algorithms at runtime. TinyXCS makes lightweight, real-time decisions using reinforcement learning and rule evolution. As a way to improve convergence and flexibility, BCM-XCS uses rule-based learning strengthened with memory of top-performing classifiers, while NSGAII uses a multi-objective evolutionary search. Three heuristic baselines—Fixed 1K, Fixed 4K, and Myopic—represent simple, non-learning policies that react to requests using fixed or greedy rules without adaptation or optimization, in addition to these learning-based techniques.
- Offline methods: TinyDT and Random Forest (RF) are two models that conduct runtime inference without further learning or exploration. They are trained beforehand using historical or simulated data. While RF uses ensemble learning to generalize workload behaviors from training samples, TinyDT uses a decision-tree structure that is tailored for execution on restricted edge devices.

Table 5 Memory usage of online and offline resource allocation methods

Method	Type	Memory usage
TinyXCS	Online	3.9 MB
TinyDT	Offline	3 KB
BCM-XCS	Online	5.2 MB
NSGAII	Online	7.5 MB
GA	Online	4.6 MB
Fixed 1K	Online	4 KB
Fixed 4K	Online	16 KB
Myopic	Online	6 KB
Random Forest (RF)	Offline	9.2 MB

Strong full-scale baselines are provided by BCM-XCS and RF, which stand for advanced online and offline intelligence, respectively. Their inclusion aids in comparing powerful but resource-intensive models to the lightweight TinyXCS and TinyDT.

Crucially, BCM-XCS has two roles in this assessment. It serves as a knowledge source for training the offline models in addition to competing as a stand-alone online baseline. Particularly, BCM-XCS produces state-action decision pairs in a variety of workload scenarios while operating online.

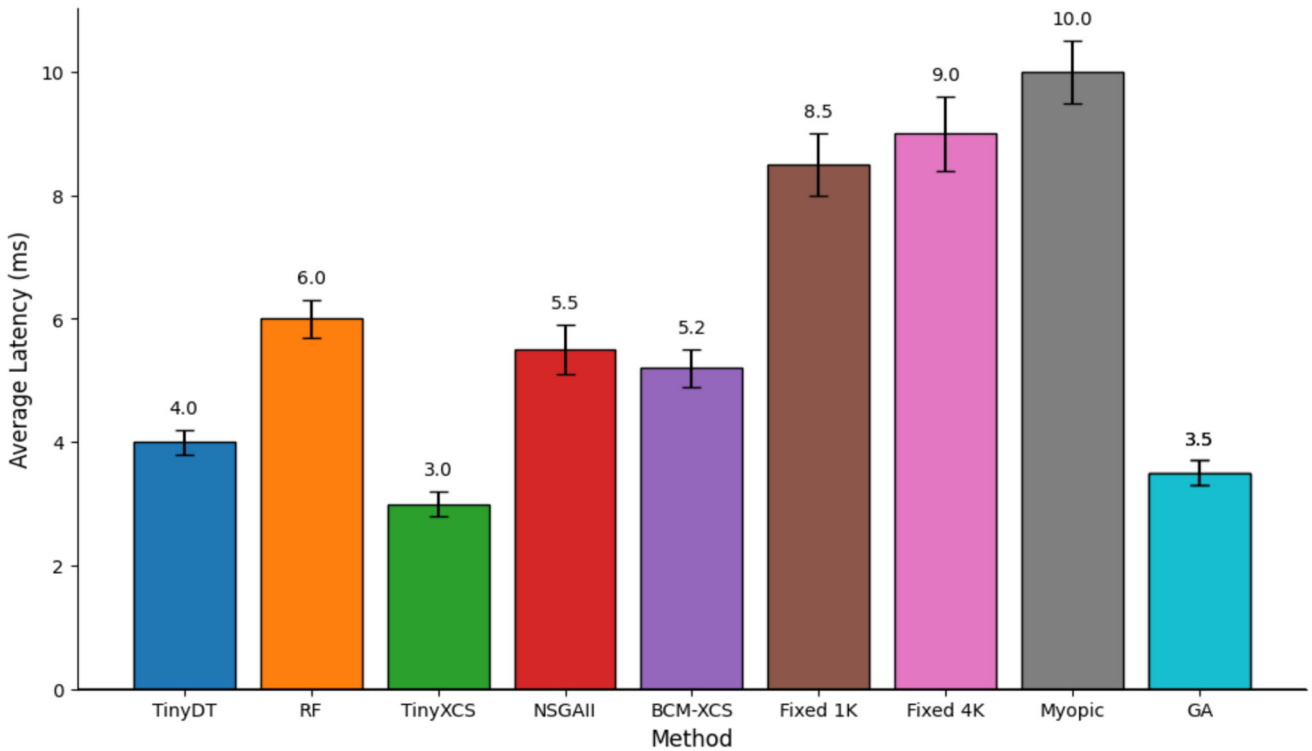


Fig. 8 Average latency of different workload allocation methods

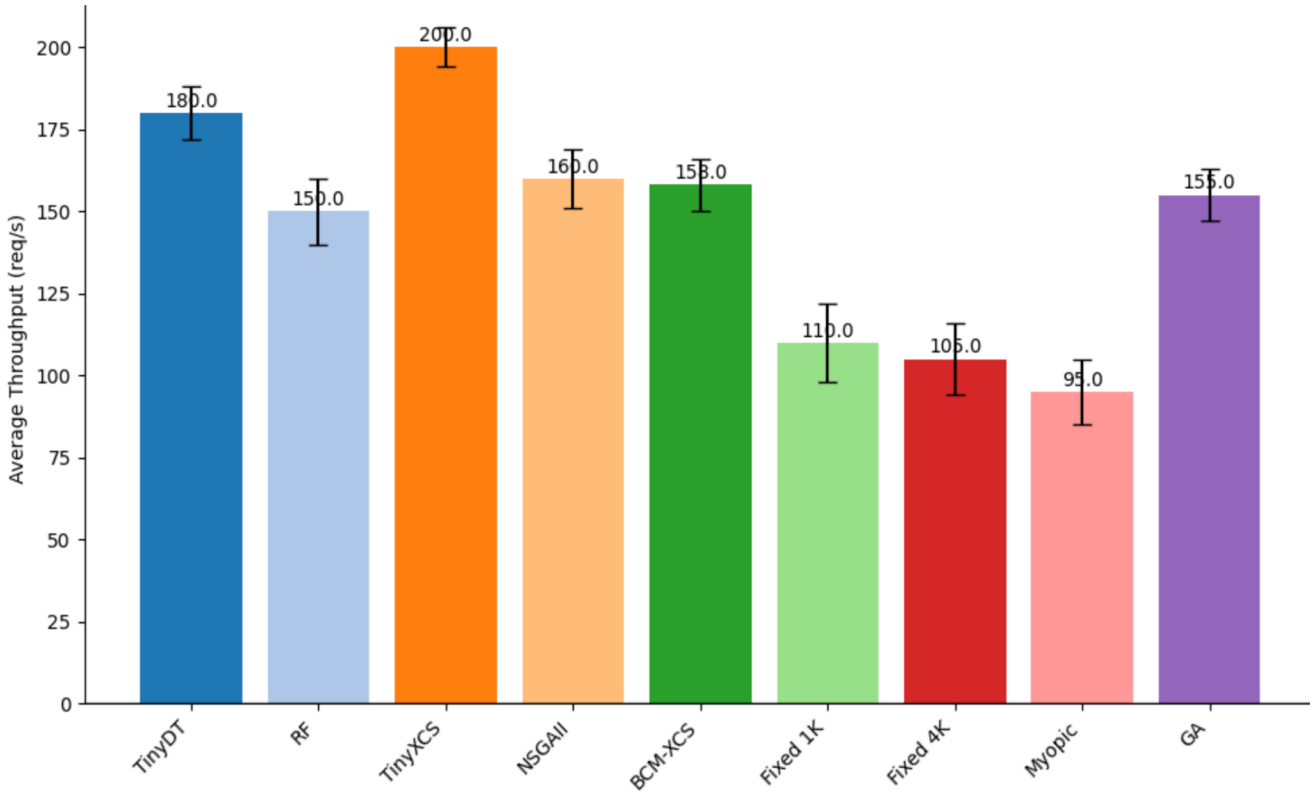


Fig. 9 Throughput of different workload allocation methods

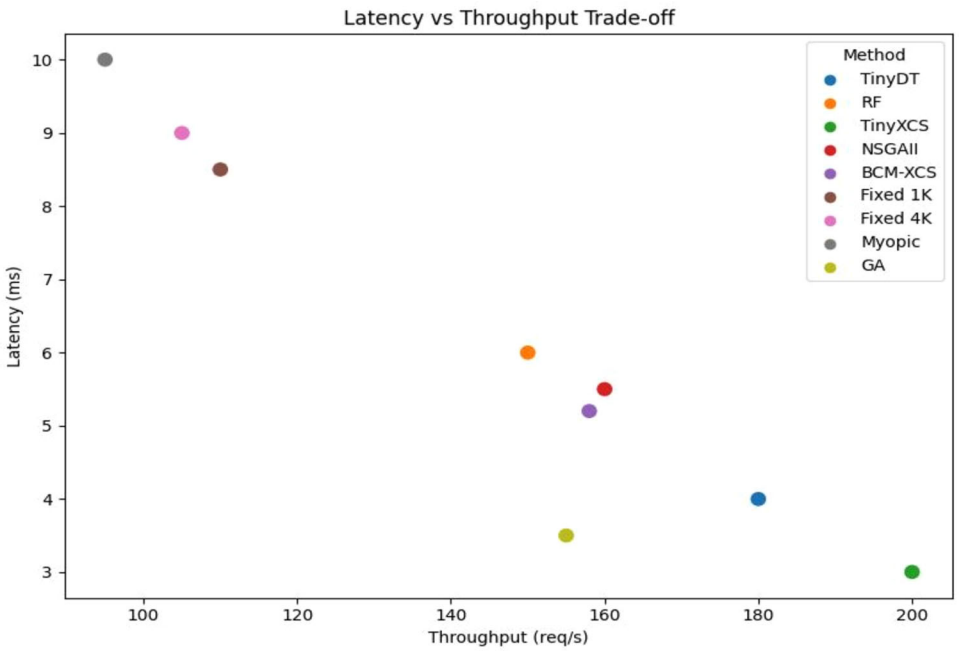


Fig. 10 Latency-throughput trade-off of different workload allocation methods

TinyDT and RF then use these pairs as labeled training data. This solution offers much lower deployment overhead while allowing the offline models to inherit the flexibility of an online learner, encapsulating the spirit of BCM-XCS's policy evolution. The offline models achieve efficient inference by shifting the learning difficulty to the training phase, which makes them ideal for situations with limited resources and latency. In essence, BCM-XCS acts as a surrogate for real-world feedback, allowing TinyDT and RF

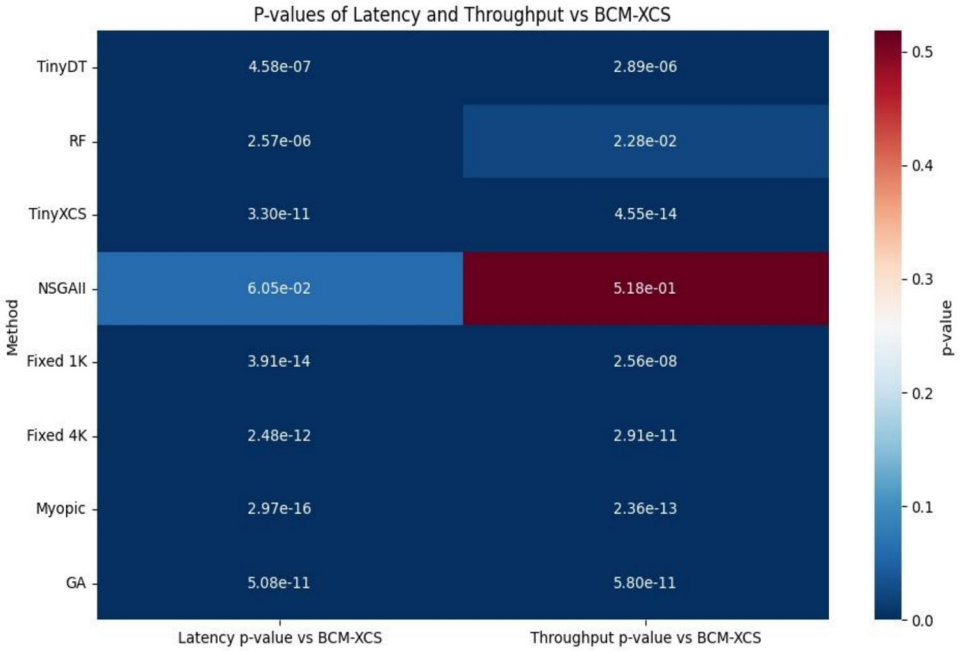


Fig. 11 Statistical significance of latency and throughput differences for each method compared to BCM-XCS

to emulate adaptive behavior with compact, deployable logic.

Two-sample t-tests on the latency and throughput findings derived from several independent simulation runs were used to assess the robustness and significance of the observed performance differences among the examined approaches. The following presumptions were confirmed before using the t-tests.

- Levene's test was utilized to verify that the variances were homogeneous, and the Shapiro–Wilk test was employed to ascertain whether the data distributions were normal. The majority of data satisfied these requirements; hence the use of parametric testing was reasonable.
- The central limit theorem was considered applicable in situations where little deviations from normality were discovered because there were sufficient independent runs.

The t-tests were conducted at a significance level (α) of 0.05, and the corresponding p-values were used in providing Figs. 8, 9, 10 and 11. Additionally, 95% confidence intervals (CIs) were provided.

The average latency attained by each workload allocation technique is contrasted in Fig. 8. Among the offline methods, TinyDT achieves the lowest latency, surpassing the Random Forest (RF) baseline. (The optimized inference path of TinyDT is superior to the modest latency achieved by the RF technique.) TinyXCS has the lowest latency in the online adaptive category, outperforming BCM-XCS, NSGA-II, and the GA baseline by a wide margin. On the other hand, the non-learning heuristic policies (Fixed 1K, Fixed 4K, and Myopic) exhibit the largest latencies, highlighting their incapacity to adapt in real time to shifting workload conditions.

According to a comparison of the best performers in each category, TinyDT works well in situations with rigorous latency limitations and advance training. By learning and adapting in real time, TinyXCS, on the other hand, excels in dynamic, uncertain contexts.

The throughput for each strategy is shown in Fig. 9, underscoring the advantages of the suggested models. TinyDT outperforms Random Forest, which handles fewer requests because of its less effective allocation mechanism, by processing the most requests per second among the offline models. TinyXCS outperforms all other techniques in the online category, including the heuristics, NSGA-II, BCM-XCS, and GA baseline, in terms of throughput. The remarkable throughput of TinyXCS is a result of its constant adaptation of its allocations to the incoming workload and available resources. The baseline online algorithms are unable to fully optimize infrastructure consumption, but TinyXCS is able to do it on the fly due to its dynamic modification. These findings point to a distinct separation of strengths: TinyDT performs exceptionally well as a pre-trained model in situations involving.

A comprehensive view of the latency–throughput trade-off for each approach is shown in Fig. 10. This trade-off plot's optimum quadrant, which combines low latency and high throughput, is where TinyDT and TinyXCS are located. This suggests that the overall efficiency of our small models is balanced and superior.

Random Forest, on the other hand, is in the intermediate range, outperforming the little models but outperforming the simple heuristics on both metrics. Although the GA baseline outperforms TinyDT and TinyXCS on both fronts, it manages to attain a little better latency–throughput balance than Random Forest. Similarly, NSGA-II and BCM-XCS are not in the ideal region due to either higher latency or lower throughput (or both). In the meantime, the heuristic methods fail to balance throughput and latency (high delay and low processing rate), clustering together in the least ideal quadrant. As a result, TinyXCS and TinyDT unquestionably rule the trade-off space, providing quick response times at large processing capacities that are difficult for all baseline techniques—both heuristic and learning-based—to match.

To confirm that the observed performance differences are robust, we ran two-sample t-tests ($\alpha=0.05$) on the latency and throughput findings. Plotting the p-values for each method's performance against the baseline (BCM-XCS) in Fig. 11 provides a summary of these tests.

As demonstrated, when compared to BCM-XCS, both suggested models achieve statistically significant gains in throughput and latency. TinyDT likewise exhibits a highly significant reduction in latency ($p \approx 4.6\times10\square\square$), while TinyXCS produces an exceptionally low p-value for latency ($p=3.3\times10\square^{11}$). These extremely low p-values attest to the fact that TinyXCS and TinyDT's latency gains are genuine, repeatable, and not the result of chance. Although not as significant as TinyDT's, the Random Forest baseline also shows a noticeable latency improvement ($p=2.6\times10\square\square$).

On the other hand, neither the GA baseline nor NSGA-II exhibit a statistically significant improvement in latency (NSGA-II's $p=0.0605$; GA's $p>0.05$), suggesting that their latency is comparable to that of BCM-XCS. The three heuristic policies are clearly less responsive than the baseline, as evidenced by their relatively significant latency deficits (e.g., Fixed 1K with $p=2.1\times10\square^{12}$, significantly worse than baseline). There is a similar pattern in the throughput results. Once more, TinyXCS achieves a much higher throughput than the baseline ($p=4.5\times10\square^{14}$), while TinyDT likewise exhibits a notable improvement ($p \approx 2.9\times10\square\square$). This demonstrates that both approaches maintain minimal latency while significantly increasing processing capacity. Throughput does not vary statistically significantly with NSGA-II ($p=0.5182$), and the GA approach also does not produce a significant throughput boost ($p>0.05$). Random Forest appears to achieve throughput comparable to BCM-XCS, as seen by its slightly significant throughput gain ($p=0.0228$). meanwhile, the straightforward Fixed 1K heuristic has a significant throughput shortfall ($p=2.6\times10\square\square$), which highlights the shortcomings of naive offloading strategies.

Reliability, economy, and robustness for edge-cloud resource allocation are reinforced by the statistical tests, which confirm the superiority of TinyXCS and TinyDT. Both approaches achieve noticeably better latency and throughput than the baseline and other approaches.

Even in the presence of challenging conditions like dynamic workloads and changing renewable power resources, TinyXCS and TinyDT continuously outperform all baseline approaches across key performance measures, according to the evaluation results (Figs. 6, 7, 8, 9, 10, 11). Specifically, the proposed tiny models outperform all the comparative methods in terms of throughput, latency, and average battery levels.

Through offline training, TinyDT produces an optimal decision model that allows high deployment throughput by providing quick, low-latency inference with minimal runtime overhead. By responding instantly to spikes in demand or variations in power availability, TinyXCS's online learning and adaptation allow it to maintain low latency and high throughput. As a result of this, TinyXCS and TinyDT both function in the ideal range of both low latency and high throughput, while the baseline methods, such as the heuristic policies, Random Forest, BCM-XCS, NSGA-II, and GA-based allocator, all fall short of this optimal balance.

These findings highlight the complementary strengths of the two proposed strategies. TinyXCS excels at quickly adjusting to unpredictable, unpredictable conditions at runtime, leveraging its lightweight reinforcement learning and rule evolution to keep performance best. Conversely, TinyDT offers a continually optimized allocation for more stable workload patterns, producing dependable low-latency decisions without the overhead of online learning. TinyXCS and TinyDT work together to address a wide variety of edge computing scenarios: While TinyDT provides a reliable, pretrained solution for steady-state or predicted conditions, TinyXCS manages volatile scenarios with real-time optimization. In both situations, the models perform noticeably better in terms of efficacy and efficiency than conventional methods (such greedy heuristics or evolutionary multi-objective allocators). According to this synergy, TinyDT and TinyXCS can be used in combination to provide state-of-the-art task allocation performance in resource-constrained edge situations. TinyDT provides baseline offline intelligence, while TinyXCS adapts online.

5 Conclusion

This research presents two lightweight edge-computing models, TinyXCS and TinyDT, designed to optimize workload distribution in resource-constrained environments by balancing resource utilization, power consumption, and

latency. TinyDT is an offline technique that provides effective offline inference with limited memory and power consumption, while TinyXCS is an online technique that uses reinforcement learning and evolutionary techniques to dynamically adapt to changes in the environment. Both models significantly outperform previous approaches that were limited to 40% battery capacity by drastically reducing latency to just 3 ms for TinyXCS and 4 ms for TinyDT, while ensuring efficient battery utilization and full charge retention. Simulation results confirm their suitability for edge networks with renewable power sources and no reliable electrical infrastructure.

Through the use of power-efficient workload distribution algorithms, TinyXCS and TinyDT reduce processing power demands while dynamically adjusting to the availability of renewable power. Low-power and resource-constrained applications benefit greatly from these enhancements, which guarantee sustainable operation in edge computing environments.

Future studies on TinyXCS and TinyDT should concentrate on improving scalability, security, and dependability in edge contexts with limited resources. To be deployed in adversarial or mission-critical edge environments in subsequent iterations, they must be strengthened with security-aware processes and reliability protocols. Also, in order to confirm the suggested models' efficacy in real-world situations, we also plan to assess them in application-specific domains, like object detection, video processing, and real-time monitoring, utilizing representative edge ML datasets.

Integrating serverless computing and approximation computing is one promising approach to increase task execution efficiency while reducing resource consumption [45]. Additionally, by utilizing failure prediction models, TinyDT can make more reliable judgments about workload allocation under uncertainty by integrating machine learning-driven reliability mechanisms [46]. Another crucial component is security, where blockchain-based access control and privacy-preserving reinforcement learning may provide safe workload distribution in multi-source heterogeneous IoT contexts [47].

Future developments will also aim to create rule-based or semi-supervised labeling techniques for TinyDT in an effort reduce its reliance on particular models like BCM-XCS and increase its resilience in a variety of edge computing settings. Additionally, future research will investigate using adaptive learning strategies, such as adaptive threshold tuning and incremental learning, to enhance TinyDT's responsiveness in dynamic edge situations. These techniques would improve accuracy and flexibility by allowing for real-time model updates and decision adjustment without the need for complete retraining. Managing idea drift amid stringent memory restrictions and preserving computational efficiency are two major issues. By addressing these, TinyDT will become more resilient and adaptable to changing workloads.

Furthermore, TinyXCS and TinyDT will be better equipped to handle unpredictable workloads and resource limitations if their adaptability and scalability are improved through the use of dynamic parameter tweaking and context-aware rule base alterations. It turns out conceivable to further optimize the utilization of resources and reduce consumption of power by implementing processes that modify model parameters in real-time based on system conditions. By guaranteeing effective performance under fluctuating workloads and changeable resource availability, these improvements would allow for more realistic deployment in extensive, real-world edge computing applications.

The current study does not specifically mimic extended edge overload or multi-stage failover spanning edge-fog-cloud tiers, even though it represents common scenarios in mobile edge computing. This can be considered as a crucial area for further research, where the resilience of both models will be evaluated in the face of severe stressors such as congestion spikes, edge server outages, or cascading cloud fallback.

To further increase efficiency and adaptability, future studies will look into a hybrid design that closely mixes TinyDT and TinyXCS. TinyDT can serve as a quick and lightweight decision-making module in this system under steady workload conditions, ensuring low-latency inference with minimal resource consumption. When environmental dynamics change (for example, when workload anomalies, concept drift, or resource degradation take place) TinyXCS enables real-time learning and adaptability. A context-aware mechanism would be developed for seamless transitions between the two models based on performance thresholds or uncertainty measures. This cooperative strategy may blend rapid decision-making with continuous adaptation, resulting in a stable and efficient task distribution across the edge–cloud continuum.

Last but not least, federated learning (FL) would enable TinyXCS and TinyDT to train jointly across edge devices without centrally storing data, enhancing scalability and privacy for applications such as healthcare, smart cities, and industrial automation. By addressing these problems, future iterations of TinyXCS and TinyDT could be made more effective, safe, and flexible, making them perfect for next-generation edge computing systems.

Future work will include feedbacks from developers and end users to improve model behavior, tweaking interfaces, and deployment tools in order to increase usability and adoption. Additionally, intuitive dashboards and APIs will be developed for smooth edge platform interaction.

Author contributions MRPH: Software, Writing—original draft. MA: Conceptualization, Software, Validation, Supervision, Writing—review & editing. AS: Conceptualization, Writing—review & editing. EE: Conceptualization, Writing—review & editing. HH: Conceptualization, Writing—review & editing. PM: Conceptualization, Writing—review & editing. BJ: Conceptualization, Writing—review & editing.

Funding The authors declare that no funds, grants, or other support were received during the preparation of this manuscript.

Data availability The data supporting the results of this study have been described in the manuscript. No datasets were generated in this study and the mentioned approach have been presented as a pseudocode in this paper.

Declarations

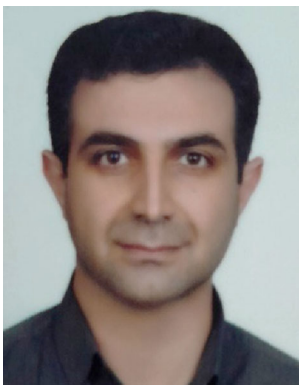
Conflict of interest The authors declare no competing interests.

References

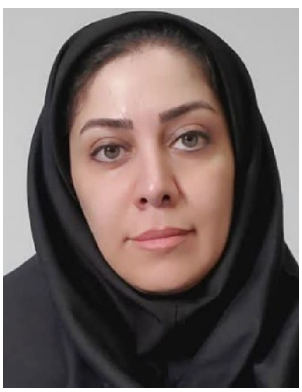
1. Peng, P., et al.: A survey on computation offloading in edge systems: from the perspective of deep reinforcement learning approaches. Comput. Sci. Rev. **53**, 100656 (2024)
2. Lin, S., Lin, W., Wu, W., Chen, H., Yang, J.: SparseTSF: modeling long-term time series forecasting with *1k* parameters. In: Presented at the Proceedings of the 41st International Conference on Machine Learning, Proceedings of Machine Learning Research (2024) [Online]. https://proceedings.mlr.press/v235/lin24n.html
3. Lin, W., Luo, X., Li, C., Liang, J., Wu, G., Li, K.: An energy-efficient tuning method for cloud servers combining DVFS and parameter optimization. IEEE Trans. Cloud Comput. **11**(4), 3643–3655 (2023). https://doi.org/10.1109/TCC.2023.3308927
4. Al-Fuqaha, A., Guizani, M., Mohammadi, M., Aledhari, M., Ayyash, M.: Internet of things: a survey on enabling technologies, protocols, and applications. IEEE Commun. Surv. Tutor. **17**(4), 2347–2376 (2015)
5. Ngu, A.H., Gutierrez, M., Metsis, V., Nepal, S., Sheng, Q.Z.: IoT middleware: a survey on issues and enabling technologies. IEEE Internet Things J. **4**(1), 1–20 (2016)
6. Shah, S.H., Yaqoob, I.: A survey: internet of Things (IOT) technologies, applications and challenges. In: 2016 IEEE Smart Energy Grid Engineering (SEGE), pp. 381–385 (2016)
7. Ometov, A., Molua, O.L., Komarov, M., Nurmi, J.: A survey of security in cloud, edge, and fog computing. Sensors **22**(3), 927 (2022)
8. Dinh, H.T., Lee, C., Niyato, D., Wang, P.: A survey of mobile cloud computing: architecture, applications, and approaches. Wirel. Commun. Mob. Comput. **13**(18), 1587–1611 (2013)
9. Bonomi, F., Milito, R., Natarajan, P., Zhu, J.: Fog computing: a platform for internet of things and analytics. In: Big Data and Internet of Things: A Roadmap for Smart Environments, pp. 169–186 (2014)
10. Shi, W., Cao, J., Zhang, Q., Li, Y., Xu, L.: Edge computing: vision and challenges. IEEE Internet Things J. **3**(5), 637–646 (2016)
11. Kaur, G., Batth, R.S.: Edge computing: classification, applications, and challenges. In: 2021 2nd International Conference on Intelligent Engineering and Management (ICIEM), pp. 254–259. IEEE (2021)
12. Dennis, D.K., et al.: EdgeML: machine learning for resource-constrained edge devices (2020). https://github.com/Microsoft/EdgeML
13. Hong, C.-H., Varghese, B.: Resource management in fog/edge computing: a survey on architectures, infrastructure, and algorithms. ACM Comput. Surv. (CSUR) **52**(5), 1–37 (2019)
14. Xu, J., Chen, L., Ren, S.: Online learning for offloading and autoscaling in energy harvesting mobile edge computing. IEEE Trans. Cogn. Commun. Netw. **3**(3), 361–373 (2017)
15. Kanapram, D., Lamanna, G., Repetto, M.: Exploring the trade-off between performance and energy consumption in cloud infrastructures. In: 2017 2nd International Conference on Fog and Mobile Edge Computing (FMEC), 8–11 May 2017, pp. 121–126. https://doi.org/10.1109/FMEC.2017.7946418
16. Zhang, J., He, X., Dai, H.: Blind post-decision state-based reinforcement learning for intelligent IoT. IEEE Internet Things J. **10**(12), 10605–10620 (2023)
17. Wu, H., Chen, L., Shen, C., Wen, W., Xu, J.: Online geographical load balancing for energy-harvesting mobile edge computing. In: 2018 IEEE International Conference on Communications (ICC), pp. 1–6. IEEE (2018)
18. Niu, X., et al.: Workload allocation mechanism for minimum service delay in edge computing-based power Internet of Things. IEEE Access **7**, 83771–83784 (2019)
19. Abbasi, M., Mohammadi Pasand, E., Khosravi, M.R.: Workload allocation in iot-fog-cloud architecture using a multi-objective genetic algorithm. J. Grid Comput. **18**(1), 43–56 (2020)
20. Sun, J., Gao, Q., Wu, C., Li, Y., Wang, J., Niyato, D.: Secure resource allocation via constrained deep reinforcement learning. arXiv preprint arXiv:2501.11557 (2025)
21. Abbasi, M., Yaghoobikia, M., Rafiee, M., Jolfaei, A., Khosravi, M.R.: Efficient resource management and workload allocation in fog–cloud computing paradigm in IoT using learning classifier systems. Comput. Commun. **153**, 217–228 (2020). https://doi.org/10.1016/j.comcom.2020.02.017
22. Xu, C., Zhang, P., Yu, H.: Lyapunov-guided resource allocation and task scheduling for edge computing cognitive radio networks via deep reinforcement learning. IEEE Sens. J. **25**(7), 2253–12264 (2025)
23. He, Z., Guo, Y., Zhai, X., Zhao, M., Zhou, W., Li, K.: Joint computation offloading and resource allocation in mobile-edge cloud computing: a two-layer game approach. IEEE Trans. Cloud Comput. **13**, 411–428 (2025)
24. Chi, X., Chen, H., Ni, Z., Sun, H., Sun, P., Yu, D.: H-STEP: heuristic stable edge service entity placement for mobile virtual reality systems. IEEE Trans. Mob. Comput. (2025). https://doi.org/10.1109/TMC.2025.3548703
25. Wang, Q., Qian, L.P., Fan, X., Li, M., Huang, C.: Energy minimization-driven communication and computation resource allocation in hybrid NOMA-RSMA industrial IoT. IEEE Internet Things J. (2025). https://doi.org/10.1109/JIOT.2025.3546394
26. Amirghafouri, F., Neghabi, A.A., Shakeri, H., Sola, Y.E.: Nature-inspired meta-heuristic algorithms for resource allocation in the internet of things. Int. J. Commun. Syst. **38**(5), e6141 (2025)
27. Sultana, N., et al.: Context aware clustering and meta-heuristic resource allocation for NB-IoT D2D devices in smart healthcare applications. Futur. Gener. Comput. Syst. **162**, 107477 (2025)
28. Tripathi, P., Sasidhar, K., Mistry, H., Shah, V.: Scheduling computing tasks on smartphones: comparative case studies of meta-heuristic algorithms on real world applications. SN Comput. Sci. **6**(4), 298 (2025)
29. Wang, Y., Chen, J., Wu, Z., Chen, P., Li, X., Hao, J.: Efficient task migration and resource allocation in cloud–edge collaboration: a drl approach with learnable masking. Alex. Eng. J. **111**, 107–122 (2025)
30. Gu, Y., et al.: Deep reinforcement learning for job scheduling and resource management in cloud computing: an algorithm-level review. arXiv preprint arXiv:2501.01007 (2025)

31. Wu, J., Zou, Y., Zhang, X., Liu, J., Sun, W., Du, G.: Dependency-aware task offloading strategy via heterogeneous graph neural network and deep reinforcement learning. IEEE Internet Things J. (2025). https://doi.org/10.1109/JIOT.2025.3549441
32. Yang, Y., Xie, C., Liu, L., Peng, X.: DynMap: a heuristic dynamic mapper for CGRA multi-task dynamic resource allocation. IEEE Trans. Comput.-Aided Des. Integrated Circuits Syst. (2025). https://doi.org/10.1109/TCAD.2025.3537975
33. Dai, W., Rai, U., Chiun, J., Yuhong, C., Sartoretti, G.: Heterogeneous multi-robot task allocation and scheduling via reinforcement learning. IEEE Robot. Autom. Lett. (2025). https://doi.org/10.1109/LRA.2025.3534682
34. Dutta, L., Bharali, S.: Tinyml meets iot: a comprehensive survey. Internet Things **16**, 100461 (2021)
35. Lanzi, P.L., Loiacono, D., Wilson, S.W., Goldberg, D.E.: Generalization in the XCSF classifier system: analysis, improvement, and extension. Evol. Comput. **15**(2), 133–168 (2007)
36. Sutton, R.S., Barto, A.G.: Reinforcement learning: an introduction. Bradford Books (2018)
37. Holland, J.H., et al.: What is a learning classifier system? In: Learning classifier systems: from foundations to applications, pp. 3–32. Springer (2000)
38. Wilson, S.W.: Classifier fitness based on accuracy. Evol. Comput. **3**(2), 149–175 (1995)
39. Bartin, B.: Use of learning classifier systems in microscopic toll plaza simulation models. IET Intel. Transp. Syst. **13**(5), 860–869 (2019)
40. Karlsen, M.R., Moschoyiannis, S.: Evolution of control with learning classifier systems. Appl. Netw. Sci. **3**, 1–36 (2018)
41. Arif, M.H., Li, J., Iqbal, M., Liu, K.: Sentiment analysis and spam detection in short informal text using learning classifier systems. Soft. Comput. **22**(21), 7281–7291 (2018)
42. Bull, L.: Applications of Learning Classifier Systems. Springer (2004)
43. Alpaydin, E.: Introduction to Machine Learning. MIT Press (2020)
44. Abbasi, M., Mohammadi-Pasand, E., Khosravi, M.R.: Intelligent workload allocation in IoT–Fog–cloud architecture towards mobile edge computing. Comput. Commun. **169**, 71–80 (2021)
45. Cao, K., Chen, M., Karnouskos, S., Hu, S.: Reliability-aware personalized deployment of approximate computation IoT applications in serverless mobile edge computing. IEEE Trans. Comput.-Aided Des. Integrated Circuits Syst. **44**(2), 430–443 (2024)
46. Xu, Z., Saleh, J.H.: Machine learning for reliability engineering and safety applications: review of current status and future opportunities. Reliab. Eng. Syst. Saf. **211**, 107530 (2021)
47. Guo, Z., Lu, Y., Tian, H., Zuo, J., Lu, H.: A security evaluation model for multi-source heterogeneous systems based on IOT and edge computing. Cluster Comput. **26**(1), 303–317 (2023)

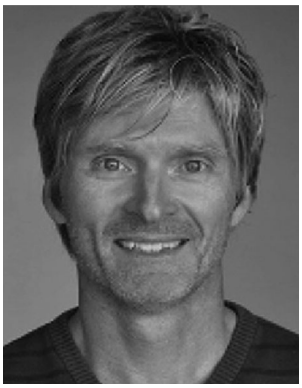
Mohammadreza Pour-Hosseini is a lecturer in the Department of Computer Engineering at Bu-Ali Sina University. He specializes in Internet of Things (IoT), fog/edge computing, workload distribution across edge-cloud paradigm, and tiny machine learning models. His recent research focuses on developing lightweight machine learning models for efficient workload distribution in edge networks, considering resource constraints and quality of service requirements.



Mahdi Abbasi holds a B.Sc. and M.Sc. in Computer Engineering from Sharif University of Technology (2000, 2005) and a Ph.D. from the University of Isfahan (2012). Since 2012, he has been affiliated with Bu-Ali Sina University, where he served as an Associate Professor in the Department of Computer Engineering. From 2021 to 2024, he was a postdoctoral researcher at the Laboratory of Informatics and Systems, Aix-Marseille University, contributing to autonomous control projects for energy grids under European Horizon funding. In 2024, he joined the Department of Computing Science at Umeå University and, since January 2025, has been a researcher at the School of Computer Science, Institute for Research in Fundamental Sciences (IPM), Iran. His research interests include autonomic computing, fog and edge computing, cyber-physical systems, machine learning architectures, and optimization in IoT networks. He was recognized among the "World's Top 2% Scientists" by Stanford University and Elsevier in 2023 and 2024 for his contributions to workload distribution and system acceleration in edge-cloud architectures.



Atefeh Salimi is an Assistant Professor of Electrical Engineering at Islamic Azad University, Isfahan (Khorasgan) Branch, Iran. She received her B.Sc., M.Sc., and Ph.D. degrees in Electronics Engineering from Isfahan University of Technology. She was a Postdoctoral Visitor at York University in Toronto, Canada, from 2018 to 2019. Her research focuses on ASIC/FPGA design, computer architecture, and biomedical sensors.



Erik Elmroth is Professor at the Department of Computing Science at Umeå University and co-founder of Elastisys AB. He has been Head and Deputy head of department of 13 years. He has established the Umeå University research on distributed systems, focusing on theory, algorithms, and systems for the autonomous management of ICT resources, spanning from individual servers to large scale cloud datacenters, federated clouds, highly distributed edge clouds, and software-defined infrastructures. Elmroth has vast experience from leading roles in major research programs, currently including the 550 million euro Wallenberg AI, Autonomous Systems and Software Program (WASP); the 30 million euro eSSENCE programme; and leading 3 projects funding 25 postdoc researchers by the Kempe Foundations. He received the Nordea Scientific Award 2011. Pre-historic highlights include being co-winner of the SIAM Linear Algebra Prize 2000, for the most outstanding linear algebra publication world-wide (in any journal) during the preceding 3-year period. Elmroth is member of the Swedish Royal Academy for Engineering Sciences. Previously, he was the Chair of the Board of the Swedish National Infrastructure for Computing (SNIC) and Chair of Swedish Research Council's expert group on e-science. International experiences include a year at NERSC, Lawrence Berkeley National Laboratory, University of California, Berkeley, and one semester at the Massachusetts Institute of Technology (MIT), Cambridge, MA. Finally, Elmroth is co-founder of Elastisys AB (www.elastisys.com), an expert on security and regulatory compliance in the cloud-native ecosystem.

Hassan Haghighi is a researcher at the LIS Laboratory of Aix-Marseille University, France. He received his Engineering License and Ph.D. degrees in Aerospace Control Engineering from Amirkabir University of Technology (Tehran Polytechnic), Tehran, Iran, in 2018. His doctoral research focused on cooperative algorithms, multi-objective optimization, and autonomous systems. His current research interests include autonomic control, computing systems, machine learning, optimization, autonomous systems, and cyber-physical systems.

Parham Moradi holds a PhD in Computer Science from Amirkabir University of Technology (AUT), completed in collaboration with the Swiss Federal Institute of Technology in Lausanne (EPFL), Switzerland. From 2011 to 2022, he served as an Assistant Professor and later as an Associate Professor at the University of Kurdistan. Since 2022, he has been a Research Fellow at RMIT University, where he is actively engaged in research, teaching, and industry collaboration.His research expertise spans a range of areas in Artificial Intelligence and Machine Learning, including deep learning, recommender systems, graph neural networks, network science, and reinforcement learning. He has published two books, two book chapters, and over 120 articles in high-ranked journals and conferences, with most of his journal papers appearing in the top 10% of journals globally. Since 2019, he has been consistently ranked among the world's top 2% of scientists. He has received several research awards in recognition of his contributions to the field.

Bahman Javadi is a Full Professor in Networking and Cloud Computing at Western Sydney University, Australia. Prior to this appointment, he was a Research Fellow at the University of Melbourne and a Postdoctoral Fellow at the INRIA Rhone-Alpes, France. He has published more than 140 papers in high quality journals and international conferences and received numerous Best Paper Awards at IEEE/ACM conferences for his research papers. He has presented many keynotes and invited talks in several conferences and universities around the world. He served as a program committee of many international conferences and workshops. His research interests include Cloud computing, Edge computing, federated learning, performance evaluation of large-scale distributed computing systems, reliability and fault tolerance, and smart applications. He is a member of Editorial Board of Future Generation Computer Systems (FGCS), Journal of Cloud Computing, and Electronics. He is a Senior Member of ACM, Senior Member of IEEE, Senior Fellow of the Higher Education Academy of UK, Member of AWS Educate Cloud Ambassador Council and Executive Member of IEEE Cloud Computing Steering Committee.