Future Generation Computer Systems 185 (2026) 108677

Contents lists available at ScienceDirect

Future Generation Computer Systems

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

MixloadSched: An interference aware and energy efficient scheduling method for mixed workloads in cloud data centers

Jianzhuo Li [a], Weiwei Lin [a,b,*], Haijie Wu [a], JiaHe Chen [c], Duanyang Du [d], Keqin Li [e]

[a] Department of Computer Science and Engineering, South China University of Technology, GuangZhou, 510000, China
[b] Pengcheng Laboratory, Shenzhen, 518066, China
[c] School of Mathematics, South China University of Technology, GuangZhou, 510000, China
[d] Aberdeen Institute of Data Science and Artificial Intelligence, South China Normal University, GuangZhou, 510000, China
[e] Department of Computer Science, State University of New York, NY, 12561, New Paltz, USA

ARTICLE INFO

Keywords:
Container scheduling
Mixed workload colocation
Energy efficiency
Performance interference

ABSTRACT

As the scale of cloud computing continues to expand, the widespread adoption of mixed colocation strategies has significantly improved resource utilization; however, it has also introduced severe performance interference, while energy management in such scenarios is frequently overlooked. To address these challenges, this paper proposes MixloadSched, an interference-aware and energy-efficient scheduling method tailored for mixed workloads. The method employs a delayed reordering strategy that buffers and resequences incoming batch tasks upon arrival, extending the scheduler's temporal window and global decision-making horizon. Building on this, we design a priority-driven task pairing mechanism and conduct a joint evaluation of interference costs and server energy efficiency using a Random Forest model, constructing a unified optimization objective to guide placement. Unlike prior schedulers that address interference or energy in isolation, or learned policies that broaden the decision horizon at the cost of unbounded latency, MixloadSched co-designs these mechanisms into a single non-iterative pipeline that satisfies interference suppression, energy efficiency, and bounded per-task latency simultaneously. On real-world Alibaba traces, MixloadSched lowers LRA CPI by about 3.3% and cluster power by 1% on average relative to the strongest interference-aware baselines (Horus and PISM), while keeping per-task scheduling overhead below 15 μs. It remains Pareto-non-dominated in the majority of evaluated settings, matching specialized baselines on QoS and energy within a single lightweight, non-iterative framework. These results provide an effective equilibrium between scheduling efficiency, Quality of Service, and energy efficiency, offering a lightweight and high-performance resource scheduling solution for cloud-native environments.

1. Introduction

1.1. Background

With the rapid proliferation of Large Language Models (LLMs) and Artificial Intelligence (AI) applications, cloud computing infrastructure is confronting unprecedented challenges in both computation and energy efficiency. Modern data centers are expanding continuously, with power consumption reaching staggering levels. According to the International Energy Agency (IEA) [1], the annual electricity consumption of global data centers has surpassed the total usage of some countries. Power expenditures now account for over 56% of the operational costs in many hyperscale data centers, approaching or even exceeding hardware manufacturing costs [2]. In this context, even a marginal reduction in energy consumption can yield substantial annual economic returns and significantly mitigate the carbon footprint. Consequently, energy efficient resource management has emerged as a core research direction in cloud systems, where **job scheduling**, acting as the critical nexus between upper-layer applications and underlying infrastructure, plays a pivotal role [3].

Concurrently, the growing user base for cloud services has imposed diversified requirements on service providers, ranging from real-time analytics to offline training. To enhance resource utilization and amortize unit computing costs, data centers commonly adopt **mixed workload colocation** strategies. This involves deploying different types of applications with varying priorities, such as batch processing jobs and long-running latency-sensitive applications, onto the same physical

* Corresponding author at: Department of Computer Science and Engineering, South China University of Technology, GuangZhou, 510000, China.
E-mail addresses: ljz_zhuo@qq.com (J. Li), linww@scut.edu.cn (W. Lin), cswuhj@mail.scut.edu.cn (H. Wu), 2143323938@qq.com (J. Chen), Dudy0550@163.com (D. Du), lik@newpaltz.edu (K. Li).

https://doi.org/10.1016/j.future.2026.108677
Received 4 February 2026; Received in revised form 6 June 2026; Accepted 15 June 2026
Available online 29 June 2026

Table 1
Characteristics of different workload types.

Type	Migratability	Resource consumption	Duration	Latency sensitivity
BPJ	High	Stable	Short	Low
LRA	Low	Volatile	Long	High

node [4]. The rise of microservice architectures has further accelerated this trend, as fine-grained, loosely coupled service units are naturally suited for dynamic orchestration and dense deployment [5]. However, while colocation improves resource efficiency, it introduces complex performance interference. High CPU or memory bandwidth contention from certain tasks can severely degrade the performance of colocated latency-sensitive microservices [6]. Such interference often leads to resource over-provisioning, which in turn incurs additional energy costs. Navigating the synergistic optimization of high performance and low energy consumption while accounting for colocation interference remains a critical challenge in cloud-native environments.

1.2. Motivation

Existing data center clusters typically host mixed workloads comprising user-experience-oriented **Long Running Applications (LRA)** and voluminous, repetitive **Batch Processing Jobs (BPJ)**. Their respective characteristics are summarized in Table 1.

Given the performance sensitivity of LRA workloads, specialized scheduling methods are required to manage mixed workloads, ensuring they are placed rationally and efficiently to minimize colocation interference. Moreover, as data center scales grow, server energy consumption has become a non-negligible factor, posing further challenges to scheduling methodologies. While some existing works explore interference-aware scheduling, they are often restricted to specific workload types. For instance, [7] proposes prediction-based resource management for deep learning. Similarly, [8] utilizes machine learning to evaluate interference between BPJ and LRA workloads based on migration characteristics, proposing a proactive scheduling method. While this approach accounts for workload heterogeneity, it fails to optimize server energy efficiency. Conversely, other studies target energy efficiency, such as [9], which improves server efficiency through power-peak-aware virtual machine consolidation. Some research focuses on single-type interference, such as LRA-to-LRA interference [10] or BPJ performance optimization [11]. These solutions often treat performance and energy as isolated or hierarchical problems, lacking a unified framework capable of co-optimizing both in a mixed workload environment.

Furthermore, when processing large volumes of batch requests submitted by users, traditional scheduling algorithms often prioritize immediate matching. While this reduces queuing time, the lack of a global perspective often leads to local optimal decisions, resulting in severe interference and reduced energy efficiency—effectively a case of "haste making waste". To overcome this limitation, advanced methods have shifted toward **Buffer or queue mechanisms** to implement delayed scheduling. This design grants the scheduler a broader global view. As illustrated in Fig. 1, we compare two task placement strategies under colocation. Although both aim to minimize interference, the queue-based method introduces a moderate delay to reorder tasks based on utilization. Experimental observations indicate that this reordering significantly reduces interference between BPJ and LRA tasks and promotes more balanced resource allocation. By sacrificing minor scheduling immediacy, the system gains superior QoS guarantees and optimized resource configuration. This constitutes the core motivation for our proposed queue-based delayed scheduling framework.

Additionally, existing scheduling methods lack an effective trade-off between scheduling efficiency and decision quality, making them difficult to apply in modern cloud-native environments characterized

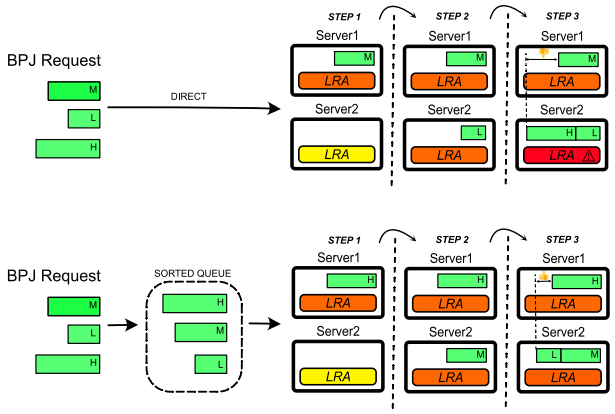


Fig. 1. Comparison of two task placement strategies. The more vibrant the color, the more severe the performance interference.

by highly dynamic, heterogeneous workloads. For instance, reinforcement learning-based colocation schemes [12,13] can improve performance and reduce energy consumption by proactively considering future arrivals and resource evolution. However, these methods suffer from excessive decision latency, high retraining costs due to state space changes, and poor interpretability (black-box nature). While lightweight heuristic methods [11] improve decision efficiency and ease of deployment, they fail to leverage historical data to adapt to dynamic environments. There is a pressing need for a method that combines heuristics with model training to achieve near-optimal global placement within a limited decision time, effectively balancing scheduling efficiency, service quality, and energy efficiency.

1.3. Contributions

This paper presents **MixloadSched**, an interference-aware and energy-optimized scheduling method for mixed workloads. The core philosophy of MixloadSched is to enhance global decision quality through a lightweight mechanism while maintaining low overhead. Specifically, it introduces a **delayed reordering strategy** that buffers and resequences incoming tasks to expand the scheduler's temporal window and global view. This allows the scheduler to evaluate a greater number of potential task-server pairings and explicitly model performance interference. Building on this, we design a priority-driven pairing method that combines task latency performance with server energy states to dynamically generate optimal placement decisions. To support this process, we propose a joint interference-energy evaluation metric that integrates colocation-induced performance degradation and server power characteristics, providing a unified optimization objective.

The primary contributions of this work are as follows:

Problem Formulation: We model the mixed workload colocation scheduling problem as a multi-objective optimization problem. This model aims to maximize the performance of LRA while minimizing the total energy consumption of the server cluster, providing a mathematical foundation for resolving resource contention.

Methodology Framework: We propose the MixloadSched framework, comprising a delayed reordering mechanism, an interference-energy evaluation model, and a priority pairing strategy. The novelty of the framework lies not in introducing any single component in isolation—buffering, ML-based interference prediction, and energy-aware placement have each been studied—but in how they are co-designed into a single non-iterative scheduling pipeline that simultaneously controls interference and energy at production-grade latency. Specifically: (i) the delayed reordering mechanism converts the streaming arrivals into bounded mini-batches with a time-bounded fallback,

which expands the scheduler's decision horizon without the unbounded queuing latency that reinforcement-learning-based schedulers typically incur; (ii) the interference-energy evaluator unifies a Random Forest interference estimator with an Energy-Efficiency Sweet Spot (ESS)-based normalized energy index $P_s \in [0, 1]$ into a single dimensionless cost, making the two objectives directly comparable inside one $\arg\min$ step rather than handled by hierarchical decisions or post-hoc migration; and (iii) the priority-pairing dispatcher exploits the typed reorder produced by (i) to combine large-task-first placement with the joint score of (ii) in $O(|S|)$ per task. To our knowledge, no prior mixed-workload scheduler couples a typed delayed-reordering buffer with a unified RF-plus-ESS scalar cost in this manner; existing methods either optimize one objective and ignore the other, or rely on heavier learned policies that violate the sub-15 μs per-task budget required for production schedulers.

Experimental Validation: Extensive experiments using real-world workload traces and simulated cluster environments demonstrate that MixloadSched performs competitively with state-of-the-art algorithms. It achieves a substantial reduction in performance degradation and an improvement in energy efficiency while maintaining extremely low scheduling latency and computational overhead. These results validate the framework's ability to effectively balance scheduling efficiency, task performance, and system energy efficiency for large-scale BPJ scheduling. Extensive experiments on real and synthetic traces show that MixloadSched attains QoS and energy efficiency on par with or better than state-of-the-art interference-aware schedulers—reducing CPI by ∼3.3% and power by ∼1% over the strongest baselines on real workloads—while uniquely doing so within a unified, sub-15 μs scheduling pipeline. We further show it is Pareto-non-dominated in most settings, with a per-condition breakdown of mean and variance deferred to Section 5.

The remainder of this paper is organized as follows. Section 2 reviews related work on cloud computing and task scheduling. Section 3 presents the system model and mathematically defines the colocation scheduling problem. Section 4 details the proposed strategy for mixed workload colocation. Section 5 presents simulation results on real-world datasets, comparing MixloadSched with several mainstream algorithms. Finally, Section 6 concludes the study.

2. Related work

2.1. Energy-efficient scheduling in cloud data centers

Cloud computing and microservice architectures play a crucial role in large-scale data processing and the deployment of large models. They provide on-demand resources (compute, storage, network) that align perfectly with the agility and dynamism of microservices, enabling modern, flexible, and scalable software systems. However, they continue to face challenges regarding energy efficiency, latency, and resource allocation. Existing research on energy optimization for microservices primarily focuses on dynamic resource management, intelligent scheduling, and elastic control. Xu et al. [14] proposed the IBrownout framework, which implements runtime degradation (brownout) of non-critical microservice functions in containerized environments, adjusting resource quotas to significantly reduce energy consumption while meeting QoS constraints. Al-Moalmi et al. [15] designed a container placement strategy based on the Whale Optimization Algorithm (WOA) to minimize total power consumption by mapping containers to physical servers under multi-dimensional resource constraints. Canosa-Reyes et al. [16] introduced a dynamic performance-energy trade-off model combined with contention-aware container consolidation to adaptively adjust resource configurations. Ranjan et al. [17] focused on microservice-based workflows, leveraging the programmability of software-defined data centers to minimize energy via optimized task mapping. For edge computing, Singh et al. [18] proposed a container-aware load balancing mechanism that reduces idle energy consumption by dynamically migrating and consolidating microservice instances.

2.2. Colocation algorithms

Existing research has explored various deployment strategies for mixed workload colocation in cloud and edge environments. These methods can be systematically categorized into **proactive** and **reactive** paradigms based on whether scheduling decisions rely on runtime performance feedback.

Proactive methods perform resource allocation before task deployment based on predictive models or prior knowledge. For instance, Chen et al. [19] proposed OLPart, an online learning-based resource partitioning mechanism that optimizes colocation by dynamically modeling job behavior. Feng et al. [20] designed Tango, which models cluster topology as a graph and combines Graph Neural Networks with Deep Reinforcement Learning to guide batch task placement using neighbor resource states. However, such methods often involve high scheduling overhead due to complex model inference and fail to explicitly model server energy consumption. Patel and Tiwari [21] proposed CLITE, a Bayesian optimization-based multi-resource partitioning technique. While deep learning-driven strategies can generate superior decisions, they rely heavily on historical data and may lack adaptability to new workloads. Notably, Yeung et al. [7] proposed Horus, a lightweight proactive scheduler predicting interference based on utilization, though its generalization is limited by predefined application types.

Reactive methods rely on runtime monitoring feedback to adjust decisions post-deployment. Yang et al. [8] proposed a strategy based on real-time scoring of compatibility between task types to guide placement. Xiang et al. [22] developed Gödel, employing feedback-driven resource coordination for multi-business workloads at ByteDance. However, these methods typically schedule at the container level, often missing fine-grained resource contention details. Hu et al. [23] modeled colocation as a minimum-cost flow problem with multi-resource constraints, which ensures strict constraint satisfaction but struggles to respond to sudden load spikes in highly dynamic environments. Kabir et al. [24] proposed RISA, using a round-robin heuristic to reduce migration costs in disaggregated data centers, but lacking closed-loop feedback on performance interference. Carvalho et al. [25] introduced a QoE-Aware scheduler that triggers migration upon detecting quality degradation. While effective for user experience, migration introduces network and cold-start overheads.

In summary, existing methods are either too complex for deployment or lack a unified framework to co-optimize interference suppression, energy efficiency, and scheduling latency. Viewed across these three constraints, prior approaches satisfy at most two of them simultaneously: interference-aware schedulers suppress contention but disregard server energy efficiency; energy-aware consolidation methods improve efficiency but ignore colocation interference; and reinforcement-learning schedulers can address both objectives yet incur high and unbounded decision latency that is incompatible with production scheduling. This three-way gap, rather than the novelty of any single technique, is the problem MixloadSched is designed to close.

3. System modeling and problem definition

3.1. System modeling

We consider a mixed workload colocation scenario where BPJ are dynamically deployed onto servers hosting LRA. This scenario is widely adopted in the industry to enhance energy efficiency and system throughput by improving resource utilization. As shown in Fig. 2, users submit a series of batch task requests and their corresponding profiles to the server cluster. These requests arrive sequentially. The cluster consists of multiple physical servers, each hosting both LRA and BPJ workloads. A resource monitor continuously collects the operating status of each server (including CPU, memory usage, power consumption, and online service performance) and reports this

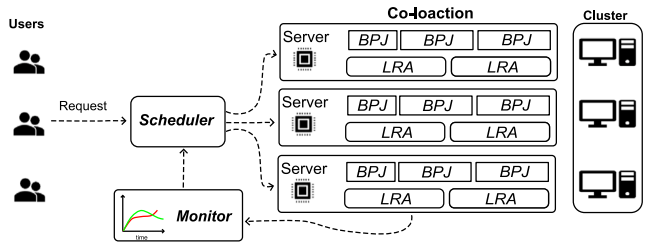


Fig. 2. Mixed workload scheduling system scenario.

Table 2
Key notations and descriptions.

Symbol	Description
$\mathcal{S}$	Set of servers
$\mathcal{L}$	Set of LRA tasks
$\mathcal{B}$	Set of BPJ tasks
$\mathcal{V}$	Set of BPJ task types
$\mathcal{C}$	Set of all tasks, $\mathcal{C} = \mathcal{L} \cup \mathcal{B}$
$\mathcal{R}$	Set of resource types (CPU, Memory, etc.)
$\mathcal{Q}$	Buffer queue
S	Number of servers
s_k	The kth server
N_L	Number of LRA tasks
l_i	The ith LRA task
B	Number of BPJ tasks
b_j	The jth BPJ task
V	Number of BPJ task types
v	BPJ task type
$x_{c,k}$	Binary variable: 1 if task c is on server s_k, 0 otherwise
$n_{k,v}$	Number of tasks of type v on server k
γ_v	Inherent interference coefficient of type v
$d_{c,r}(t)$	Resource demand of task c for resource r at time t
$res_{k,r}$	Capacity of server k for resource r
$C_{\text{int}}(l_i)$	Interference cost of LRA task l_i
C_{int}	Total interference cost of the cluster
$p(t)$	Power of the server at time t
$power(u)$	Server power function based on utilization u
$u(t)$	CPU utilization of the server at time t
E_i	Total energy consumption of server i
E_{total}	Total energy consumption of the cluster
$P_i(t)$	Power of server i at time t
$a_i(t)$	Active status of server i at time t
T	Time period or number of sampling points
t_s, t_e	Start and end time
Δt	Sampling interval
peff(u)	Energy efficiency of server at utilization u
ps(u)	Processing speed of server at utilization u
u^{ESS}	Energy-Efficiency Sweet Spot (utilization)
P_s	Utilization-based server energy efficiency index
L_q	Queue length
T_{max}	Maximum waiting time
λ	Trade-off weight
w_j	Comprehensive resource demand intensity of task b_j
ω_b	Priority strength of task b
RF_θ	Random Forest Model

information to the scheduler. As the core component, the scheduler analyzes the performance, resource utilization, and energy efficiency of all servers to select the optimal host for each newly arriving batch task. This scheduler corresponds to the central scheduling module in cloud-native systems (e.g., the Volcano scheduler in Kubernetes), aiming to maximize BPJ scheduling efficiency and total cluster energy efficiency while guaranteeing LRA QoS.

3.2. Problem definition

We model the objectives, including task performance and cluster energy efficiency, and define the relevant constraints. Key symbols are listed in Table 2. We denote the S servers contained in the cluster as $\mathcal{S} = \{s_1, s_2, \ldots, s_S\}$. The servers host a set of LRA tasks, $\mathcal{L} = \{l_1, l_2, \ldots, l_{N_L}\}$. BPJ tasks continuously arrive and are assigned to servers, denoted by $\mathcal{B} = \{b_1, b_2, \ldots, b_B\}$, with types $\mathcal{V} = \{v_1, v_2, \ldots, v_V\}$. The Set of all tasks is $\mathcal{C} = \mathcal{L} \cup \mathcal{B}$. We define a binary variable $x_{c,k} \in \{0, 1\}$ to represent the deployment relationship:

$$x_{c,k} = \begin{cases} 1, & \text{if task } c \text{ is deployed on server } s_k \in \mathcal{S}, \\ 0, & \text{otherwise.} \end{cases} \tag{1}$$

LRA tasks are treated as first-class citizens within the cluster; their performance (e.g., Cycles Per Instruction, CPI) must be prioritized. The performance of an LRA task l_i primarily depends on the type and quantity of colocated BPJ tasks due to resource contention [8]. We define the interference cost of an LRA task l_i as follows:

$$C_{\text{int}}(l_i) = \sum_{v \in \mathcal{V}} \gamma_v \cdot n_{k,v} \cdot x_{l_i,k}, \tag{2}$$

where $n_{k,v} = \sum_{b_j \in B} x_{b_j,k}$ is the number of tasks of type v on server s_k, and γ_v is the inherent interference coefficient. Task types are determined by key features, namely resource utilization and duration. The total cluster interference cost is the sum of costs for all LRA tasks:

$$C_{\text{int}} = \sum_{l_i \in \mathcal{L}} C_{\text{int}}(l_i). \tag{3}$$

Servers dominate data center energy consumption. Server power is modeled as a function of CPU utilization $u(t)$ [9]. Other components contribute negligibly to dynamic power. Therefore, we define the power consumption at time t as:

$$p(t) = \text{power}(u(t)), \tag{4}$$

where the model power() depends on the inherent properties of the server. The energy consumption over a period T is:

$$E_i = \int_{t_s}^{t_e} p(t) a_i(t) dt, \tag{5}$$

where $a_i(t)$ indicates if server i is active. In practice, we use discrete sampling with interval Δt:

$$E_i = \sum_{t=0}^{T} P_i(t) \cdot \Delta t, \tag{6}$$

where, $P_i(t)$ denotes the power of server i at time t. The total cluster energy consumption is:

$$E_{total} = \sum_{i=1}^{S} E_i = \sum_{i=1}^{S} \sum_{t=0}^{T} P_i(t) \cdot \Delta t \tag{7}$$

The constraints include the single-assignment constraint (each task on at most one server):

$$\sum_{k=1}^{S} x_{c,k} \leq 1, \quad \forall c \in C \tag{8}$$

and the resource capacity constraint:

$$\sum_{c=1}^{|C|} x_{c,k} d_{c,r}(t) \leq res_{k,r}, \quad \forall k \in \mathcal{S}, r \in \mathcal{R} \tag{9}$$

The scheduling problem is modeled as a multi-objective optimization:

$$\min \quad (C_{int}, E_{total}), \quad s.t. \quad (8) \quad and \quad (9) \tag{10}$$

3.3. Complexity analysis

We demonstrate that this scheduling problem belongs to the **NP** class and prove its **NP-hardness** via a polynomial-time reduction from the classic Bin Packing Problem (BPP).

3.3.1. Proof of membership in NP

Given a candidate solution (assignments of $x_{c,k}$), its feasibility and objective value can be verified in polynomial time. Verifying constraint (8) takes $O(|\mathcal{C}| \cdot |\mathcal{S}|)$, and constraint (9) takes $O(|\mathcal{S}| \cdot |\mathcal{R}| \cdot |\mathcal{C}|)$. Calculating C_{int} and E_{total} also requires polynomial time relative to the input size, with time complexities $O(|\mathcal{L}| \cdot |\mathcal{S}| \cdot |\mathcal{V}|)$ and $O(|\mathcal{S}| \cdot T)$, respectively. Thus, the problem is in **NP**.

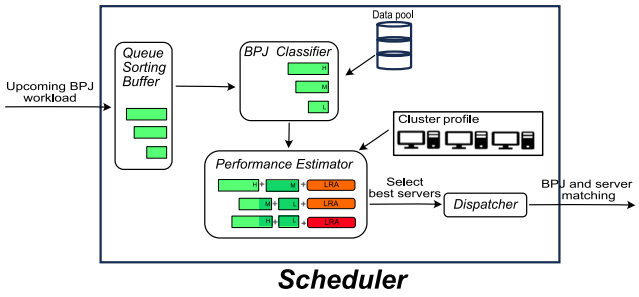


Fig. 3. Overview of MixloadSched.

3.3.2. NP-hardness via reduction

Consider the decision version of the Bin Packing Problem (BPP): Given n items with sizes $s_1, \ldots, s_n \in (0, C]$ and k bins each of capacity C, does there exist a scheme to pack all items into no more than k bins such that the total size of items in each bin does not exceed C? We reduce this problem to a special case of our scheduling problem. The construction is as follows: Create k servers, each with only one type of resource (e.g., CPU) and a capacity of C; introduce n BPJ tasks, where the resource demand of task j is constant as $d_{j,1}(t) = s_j$; set $\mathcal{L} = \emptyset$ (no LRA tasks), so the interference cost $C_{\text{int}} = 0$; define the energy consumption model as: if a server hosts at least one task, its energy consumption is 1; otherwise, it is 0. At this point, the objective function is simplified to minimizing the number of active servers, while Constraints (8) and (9) ensure task single-assignment and resource capacity limits, respectively. The correctness of the reduction lies in: BPP has a solution if and only if the scheduling problem has a feasible solution using no more than k servers. This construction can be completed in $O(n + k)$ time, which is a polynomial-time reduction.

3.3.3. Conclusion

Since the problem contains the NP-complete BPP as a special case, the general multi-resource, multi-objective problem is **NP-hard**. Consequently, exact polynomial-time algorithms do not exist (assuming $P \neq NP$), necessitating heuristic or approximate approaches.

4. Methodology

To achieve the objectives of Problem (10), we propose an interference aware mixed workload scheduling method called MixloadSched. Adhering to the principles of lightweight and non-intrusiveness, this method heuristically schedules BPJ onto clusters hosting LRA, achieving dual improvements in energy conservation and system performance. The framework can be easily ported to current mainstream container scheduling frameworks. As illustrated in Fig. 3, the main modules of MixloadSched include a Buffer, Classifier, Estimator, Dispatcher, and Data Pool. The Buffer is used to receive time-sequential BPJ task requests and perform priority reordering; the Classifier is a data-driven classification model that outputs category labels based on task profiles, which can reflect resource usage characteristics; the Estimator calculates the performance metrics of LRA tasks and the energy efficiency metrics of the cluster in the cluster, and these metrics are used to guide task-server matching (e.g., screening candidate servers). The Dispatcher executes the operation of placing tasks onto servers; the Data Pool collects historical data, including BPJ task runtime information, past BPJ task placement on servers, and the corresponding performance of LRA tasks.

4.1. Classifier and estimator

To enable MixloadSched to accurately characterize the performance interference experienced by LRA tasks, we implement a two-stage approach: first, classifying BPJ tasks (represented by the Classifier in Fig. 3), followed by evaluating their impact on LRA tasks (represented by the Estimator in Fig. 3). For task classification, based on a clustering analysis of BPJ tasks from the Alibaba trace and common resource consumption patterns [8], we categorize tasks into three primary dimensions: CPU, Memory (MEM), and Duration. Each dimension is further divided into three discrete levels to represent different resource intensities, as summarized in Table 3. According to Table 3, each task can be classified into one of the 27 ($3 \times 3 \times 3$) possible categories—for instance, a task characterized by a low CPU level, medium memory level, and long execution duration. For each incoming BPJ task, we predict its specific category based on its accessible task profile prior to deployment. We elaborate on the classification pipeline below to facilitate reproducibility.

Table 3
Task classification.

	CPU	Memory	Duration
Level	Low	Low	Short
	Medium	Medium	Medium
	High	High	Long

Feature extraction. For each BPJ task, we build an 8-dimensional feature vector $\mathbf{x}_b \in \mathbb{R}^8$ from information available at admission time, before the task starts running. Five of the features describe the task's dependency structure, which we parse from the `task_name` field of the `batch_task` table (task names encode the directed acyclic graph (DAG) of intra-job dependencies): (i) the number of upstream dependencies; (ii) a binary indicator of whether the task has any dependency; (iii) the largest and (iv) the smallest referenced dependency index; and (v) the number of distinct dependencies. The remaining three features are the requested resource profile and scale of the task: (vi) the number of instances (`instance_num`); (vii) the requested CPU quota (`plan_cpu`); and (viii) the requested memory quota (`plan_mem`). All eight features are known when the task is admitted and require no runtime telemetry of the task being scheduled, so the classifier is usable before deployment. We emphasize that `plan_cpu` and `plan_mem` are the resources the user requests, whereas the CPU and MEM labels defined below are derived from the actually observed utilization; predicting realized usage levels from requested quotas is precisely the request-to-usage mapping the classifier is meant to learn, and no label-derived quantity enters the feature vector. Features are drawn from the same Alibaba cluster trace used in our experiments [26].

Label generation. The ground-truth label of each historical task along every dimension is obtained by thresholding a measured quantity at data-driven percentile cut points computed on the training set. The CPU and MEM levels are derived from the task's actually observed mean CPU and memory utilization, split at the 33rd and 67th percentiles so that the three levels are near-equally populated. The Duration level is derived from the task's measured runtime, split at the 40th and 80th percentiles; this asymmetric partition reflects the heavy-tailed runtime distribution of the trace and intentionally isolates the small fraction of long-running tasks, which dominate colocation interference, into a distinct "long" class rather than forcing equal-sized bins. Computing thresholds from the training set alone prevents information from the test period leaking into label definitions.

Ensemble structure. We decompose the 27-class problem into three independent 3-class subproblems—one per dimension—and train an independent LightGBM gradient-boosting classifier for each. Two properties of the trace motivate this factorization. First, the three dimensions are only weakly correlated (Spearman $|\rho| < 0.3$ across all pairs), so predicting them separately and recombining loses little joint information relative to a flat 27-way classifier. Second, each subproblem has a single semantic axis and only three ordered levels, which makes it easier to fit and to interpret than a 27-way target whose

classes are products of three axes. The final 27-way category of a task is the Cartesian combination of the three predicted per-dimension levels.

Implementation and hyperparameters. The three classifiers are implemented with LightGBM. Hyperparameters are selected per dimension by grid search with 3-fold cross-validation, optimizing macro-averaged F1 over a search space spanning the number of estimators, maximum tree depth, number of leaves, learning rate, and minimum number of samples per leaf (108 configurations in total). Because the full training set contains on the order of 8.6 million tasks, running the cross-validated search over all of them is computationally prohibitive; we therefore adopt a two-stage procedure—the grid search is performed on a uniformly sampled subset of one million tasks, and the configuration with the best cross-validated macro-F1 is then used to retrain each classifier on the complete training set. Features are standardized with a `StandardScaler` fitted on the training data; although tree-based models are scale-insensitive, this keeps the preprocessing pipeline uniform across all learned components. To match the deployment setting in which the classifier predicts tasks that have not yet run, the trace is partitioned chronologically into training, validation, and test segments, so the test segment contains only tasks arriving after all training tasks. Per-dimension performance is reported as macro-averaged F1, which weights all three levels equally and therefore remains informative under the asymmetric Duration partition, and task-level performance as joint 27-way top-1 accuracy; the trained models and their selected configurations are persisted for use by the online scheduler.

Upon determining the classification of BPJ tasks, we represent the task placement on a server as a multi-dimensional vector to evaluate the performance interference of colocated LRA tasks, as initially formulated in (2). Although Eq. (2) provides a linear superposition model, resource contention among different task types in real-world systems often exhibits significant non-linearity and coupling effects. To enhance modeling precision, we employ a Random Forest (RF) regression model [27] to directly predict LRA performance interference from various combinations of BPJ task types, thereby avoiding rigid assumptions regarding fixed interference coefficients.

Why a tree ensemble? The interference predictor sits on the scheduling critical path—it is invoked once per candidate server for every task—and therefore must deliver low and predictable inference latency. The input is low-dimensional (the 27 type-count features) yet exhibits strong non-linear, non-monotonic interactions among colocated task types, which simple parametric models such as linear regression cannot capture. Tree-ensemble models are well suited to these constraints: they capture non-linear feature interactions without explicit feature crosses, are robust to feature scale and to the moderate noise present in CPI measurements, and—once their trees are compiled to native code—offer deterministic, sub-20 μs inference without any GPU or special runtime, unlike deep models such as Multi-Layer Perceptrons (MLP). Among tree ensembles, Random Forest (RF) and gradient boosting are the two natural candidates. As we show empirically in Section 5.5, both attain statistically indistinguishable accuracy and, after compilation, comparable inference latency on this task. We adopt RF for its conceptual simplicity, its minimal hyperparameter footprint, and the variance-reduction property of bagging, which is well matched to the low signal-to-noise regime of CPI prediction. Because the Estimator is a modular component, gradient boosting can be substituted without altering the scheduling framework.

Training pipeline. Random Forest is an ensemble learning model that aggregates the outputs of multiple weak learners (decision trees) to achieve robust predictive performance.

We collect the counts of different BPJ task types and the corresponding CPI of colocated LRA tasks as training data. To reduce training complexity and to expose ordinal interference severity to the scheduler, we discretize the continuous CPI metric into a finite set of levels based on percentiles of historical CPI data; a higher level indicates a higher historical percentile range, signifying more severe interference. The RF regressor uses `n_estimators` = 30, `max_depth` = 8, and `min_samples_leaf` = 30, selected by validation on a chronological data split (Section 5.5). At deployment the trained forest is compiled to native code via Treelite to minimize inference latency, with predictions identical to the uncompiled model. Consequently, the performance interference degree of an LRA task l_i can be expressed as:

$$C_{\text{int}}(l_i) = \text{RF}_\theta(\langle n_{v_1}, n_{v_2}, \dots, n_{v_{|\mathcal{V}|}} \rangle). \tag{11}$$

where RF_θ denotes the trained Random Forest model, and n_{v_1} represents the count of tasks belonging to type v_1, and so forth. This model provides a quantitative assessment of the performance interference between LRA tasks and various types of colocated BPJ tasks. Within the Estimator component, it is utilized to estimate the performance impact on LRA tasks across different candidate servers.

In addition to characterizing the performance interference of LRA in the current cluster, the Estimator also evaluates the cluster's energy efficiency using energy efficiency metrics to guide task scheduling. Inspired by [9], we define server energy efficiency $\text{peff}(u)$ as the ratio of performance to power:

$$\text{peff}(u) = \frac{\text{ps}(u)}{\text{power}(u)}, \tag{12}$$

where $\text{ps}(u)$ is the processing speed when the PM's current utilization is u. We identify the Energy-Efficiency Sweet Spot (ESS) utilization u^{ESS} as follows:

$$u^{\text{ESS}} = \arg\max_u \text{peff}(u), \tag{13}$$

where $u \in [0, 1]$ is the CPU utilization of the server. We then define a normalized energy efficiency index P_s:

$$P_s = 1 - \left| u_s - u_s^{\text{ESS}} \right|. \tag{14}$$

$P_s \in [0, 1]$ guides the scheduler to select servers operating near their optimal energy efficiency point.

4.2. Buffer

To avoid the pitfalls of First-Come-First-Served (FCFS), we introduce a delayed reordering mechanism, which is represented by the Buffer in Fig. 3. The process is detailed in Algorithm 1.

Algorithm 1 Buffer Queue Reordering Algorithm

```
1: Initialize empty buffer queue Q
2: for each arriving BPJ task b_j do
3:   if |Q| < L then
4:     Enqueue b_j into Q
5:   else
6:     Sort Q in descending order of resource demand
7:     return Q
8:   end if
9: end for
```

The system maintains a buffer queue Q. When a new task b_j arrives, if $|Q| < L$, the task is appended to the end of the queue (Lines 3 and 4); Once the size of Q reaches L, the tasks are reordered in descending order based on their comprehensive resource demand intensity (Line 6). This reordering ensures that tasks with high resource requirements are prioritized, reducing the likelihood that subsequent, smaller tasks cannot be scheduled due to resource fragmentation. After the reordered batch is dispatched, the queue is cleared before the next batch is accumulated.

It is noteworthy that the queue length L is a critical parameter: if L is too small, the mechanism degrades to near-First-Come-First-Served (FCFS); if L is too large, the per-batch reordering cost grows. The experimental section will analyze how L affects the number of batches a task waits.

Moreover, to prevent the queue from remaining underutilized for extended periods, we introduce a timeout mechanism: if the average waiting time of tasks in the queue exceeds the threshold $T_{\max}$, the

tasks in the queue are immediately reordered and proceed to the next scheduling step. We set $T_{\max}$ to one scheduling period, the smallest natural time unit in our system, so that a task is never carried over to the next period, while the timeout path is exercised only when arrivals are too sparse to fill the buffer within a period. The modified algorithm is presented as Algorithm 2.

Algorithm 2 Time-Bounded Buffer Queue Reordering Algorithm

Require: Queue capacity L, maximum waiting time $T_{\max}$
Ensure: Reordered task queue $\mathcal{Q}_{\text{out}}$
1: Initialize empty buffer queue $\mathcal{Q}$
2: **for** each arriving BPJ task b_j at time t_{arr} **do**
3: b_j.arrival_time $\leftarrow t_{\text{arr}}$
4: Enqueue b_j into $\mathcal{Q}$
5: **if** $|\mathcal{Q}| = L$ **or** $(t_{\text{now}} - \text{head}(\mathcal{Q}).\text{arrival\_time}) \geq T_{\max}$ **then**
6: **for** each $b \in \mathcal{Q}$ **do**
7: Predict type v_b using trained classifier
8: Compute priority strength: $\omega_b \leftarrow \text{CPU\_level}(v_b) + \text{MEM\_level}(v_b)$
9: **end for**
10: Sort $\mathcal{Q}$ in descending order of ω_b
11: $\mathcal{Q}_{\text{out}} \leftarrow \mathcal{Q}$
12: Clear $\mathcal{Q}$
13: **return** $\mathcal{Q}_{\text{out}}$
14: **end if**
15: **end for**

With the incorporation of the maximum waiting time constraint, the reordering process is initiated either when the queue length reaches the capacity $|\mathcal{Q}| = L$ or when the waiting time of the head task exceeds a predefined threshold $T_{\max}$. Upon triggering this process, the algorithm traverses all tasks currently residing in the Buffer and inputs their features into the pre-trained Classifier to determine their respective categories v_b. Subsequently, these tasks are resequenced in descending order based on their comprehensive resource demand intensity, $w_b = \text{CPU\_level}(v_b) + \text{MEM\_level}(v_b)$, ensuring that resource-intensive tasks are prioritized. Finally, the buffer queue is cleared, and the reordered task batch $\mathcal{Q}_{\text{out}}$ is transmitted to the Dispatcher for resource allocation.

4.3. Overall execution of MixloadSched

The overall workflow of the MixloadSched algorithm is formalized in Algorithm 3. This algorithm achieves the synergistic optimization of high performance and low energy consumption through a combination of buffer queue reordering (Buffer), task classification (Classifier), joint interference-efficiency evaluation (Estimator), and priority-matching strategies (Dispatcher). Initially, the algorithm initializes an empty buffer queue $\mathcal{Q}$ to temporarily store incoming BPJ tasks. For each arriving BPJ task b_j, the algorithm invokes Algorithm 2 to obtain a reordered task sequence. Tasks are resequenced in descending order based on their comprehensive resource demand intensity, ensuring that resource-intensive tasks are prioritized (Line 3).

For each task b in the reordered queue, the algorithm first performs a feasibility screening by traversing all servers $k \in \mathcal{S}$ to verify whether task b can be accommodated within the remaining CPU and memory resources of server k (Line 7–8). Servers that satisfy these resource constraints are added to the candidate set $\mathcal{S}_{\text{cand}}$. Subsequently, a joint evaluation is conducted for each candidate server $k \in \mathcal{S}_{\text{cand}}$ (Line 11–15). Specifically, a Random Forest model RF_θ is employed to predict the interference cost $\hat{C}_{\text{int}}(k) = \text{RF}_\theta(\{n_{k,v}\} \cup \{v_b\})$ if task b were placed on server k. The algorithm then computes the updated CPU utilization $u_k^{\text{new}} = u_k + \Delta u_b$ and derives an energy efficiency metric $P_k = 1 - |u_k^{\text{new}} - u_k^{\text{ESS}}|$ relative to the ESS point u_k^{ESS}. A comprehensive cost is then calculated as $\text{Cost}(k) = \lambda \cdot \hat{C}_{\text{int}}(k) + (1 - \lambda) \cdot (1 - P_k)$, where λ (default 0.5) serves as a weighting factor for the trade-off.

The server with the minimum comprehensive cost is selected as the optimal host, $k^* = \arg\min_{k \in \mathcal{S}_{\text{cand}}} \text{Cost}(k)$ (Line 17). Finally, task b is dispatched to server k^*, and the server's state, including CPU utilization u_{k^*}, resource availability, and task type counts $\{n_{k^*,v}\}$, is updated accordingly (Line 19). Once all tasks in the current batch have been processed, the queue $\mathcal{Q}$ is flushed to receive the next burst of incoming tasks.

Algorithm 3 MixloadSched: Interference Aware Energy Efficient Scheduling

Require: Task stream $\{b_j\}$, Queue size L, Max wait time $T_{\max}$, Trade-off weight $\lambda \in [0, 1]$
Ensure: Task-to-server assignment
1: Initialize empty buffer $\mathcal{Q}$
2: **for** each arriving BPJ task b_j at time t_{arr} **do**
3: Gets $\mathcal{Q}$ from Algorithm 2
4: **for** each $b \in \mathcal{Q}$ **do**
5: $\mathcal{S}_{\text{cand}} \leftarrow \emptyset$
6: **for** each server $k \in \mathcal{S}$ **do**
7: **if** b fits in k's remaining CPU and memory **then**
8: Add k to $\mathcal{S}_{\text{cand}}$
9: **end if**
10: **end for**
11: **for** each $k \in \mathcal{S}_{\text{cand}}$ **do**
12: $\hat{C}_{\text{int}}(k) \leftarrow \text{RF}_\theta(\{n_{k,v}\} \cup \{v_b\})$
13: $u_k^{\text{new}} \leftarrow u_k + \Delta u_b$
14: $P_k \leftarrow 1 - |u_k^{\text{new}} - u_k^{\text{ESS}}|$
15: $\text{Cost}(k) \leftarrow \lambda \cdot \hat{C}_{\text{int}}(k) + (1 - \lambda) \cdot (1 - P_k)$
16: **end for**
17: $k^* \leftarrow \arg\min_{k \in \mathcal{S}_{\text{cand}}} \text{Cost}(k)$
18: Assign b to server k^*
19: Update u_{k^*}, resource usage, and $\{n_{k^*,v}\}$
20: **end for**
21: **end for**

The computational complexity of the proposed MixloadSched algorithm is primarily determined by the buffer reordering mechanism and the server selection process. For a scheduling cycle involving a buffer of L tasks, the reordering phase (Algorithm 2) incurs a complexity of $O(L \log L + L \cdot C_{\text{cls}})$, where C_{cls} represents the constant time overhead for task classification using the pre-trained LightGBM models. Subsequently, in the task dispatching phase (Algorithm 3, Line 17), the algorithm iterates through the server set $\mathcal{S}$ for each task to perform feasibility checks and cost evaluations. Since the inference time of the Random Forest model and the calculation of energy efficiency metrics are constant operations ($O(1)$) determined by the fixed depth and number of trees in the model, the optimal server selection for a single task has a complexity of $O(|\mathcal{S}|)$. Consequently, the overall time complexity for scheduling a stream of N tasks is $O(N \cdot |\mathcal{S}|)$. Given that the buffer size L is a small constant system parameter and the scheduling overhead scales linearly with the cluster size $|\mathcal{S}|$, the algorithm operates in polynomial time, ensuring low-latency decision-making suitable for large-scale real-time production environments. As validated by experimental results showing scheduling overhead of less than 15 μs even for clusters with 1000 servers (Table 8), the proposed algorithm achieves a favorable balance between decision quality and computational efficiency for the NP-hard mixed workload colocation scheduling problem.

5. Evaluation

5.1. Experimental settings

Our algorithm is evaluated through a Python-based simulation framework [28] built upon a widely used and realistic workload trace.

The framework is capable of simulating the real-world operational environment of cloud data centers. The experiments are conducted on a machine running Ubuntu 20.04, equipped with an Intel(R) Xeon(R) Silver 4316 CPU, an NVIDIA GeForce RTX 4090 GPU, and 64 GB of RAM.

Cluster resource profiles: The experimental evaluation leverages real-world server cluster traces [26], which capture the execution dynamics of two distinct workload types colocated on servers. The workload data for BPJ and LRA are derived from Alibaba's `batch_task` and `container_usage` in the `alibaba_trace` dataset, respectively. These traces span an 8-day period by setting the timeslot as 1 h, recording critical metrics including resource utilization, Cycles Per Instruction (CPI), and application types for LRAs, as well as DAG dependencies and instance counts for BPJs. Additionally, we utilize synthetic data by generating uniformly distributed tasks at random timestamps to augment the experimental evaluation. Server power consumption profiles are sourced from the SPEC benchmark [29], encompassing server specifications, power consumption curves, and energy efficiency curves.

Data and replication: The five evaluated cluster scales (5, 50, 100, 500, and 1000 servers) are obtained by randomly subsampling a different set of servers from the Alibaba trace for each scale, while reusing the real, time-aligned BPJ arrivals originally assigned to the sampled servers. For the synthetic experiments, BPJs are generated synthetically with arrival times following a Poisson process; each configuration is replicated 5 times with independent random seeds, and reported metrics are averaged over the 5 runs. The synthetic BPJs' resource profiles follow the empirical distribution observed in the trace.

Baseline algorithms: To validate the effectiveness of our proposed method, we compare it against the following algorithms:

- **Default (alibaba)** [30]**:** Represents the native Alibaba scheduler, reflecting the original task-server assignments recorded in the trace dataset.
- **Round Robin (RR)** [24]**:** A baseline strategy that assigns incoming BPJ tasks to available servers in a sequential, circular order.
- **PISM** [8]**:** An algorithm that allocates BPJ tasks to servers using an interference-aware model but without a delayed scheduling mechanism.
- **Horus** [7]**:** A state-of-the-art interference aware scheduling algorithm that utilizes a utilization-based interference model to guide BPJ task allocation.
- **PEAP** [9]**:** A Peak Efficiency-Aware heuristic placement method that focuses on reducing cluster energy consumption through task placement and re-placement.

Metrics: To evaluate both the performance and energy consumption of the algorithm, we quantify performance interference cost using the classic metric, Cycles Per Instruction (CPI) [31], where a lower value indicates better performance. We record the average CPI of all LRA tasks in the cluster as an interference cost metric. Additionally, we record the average power over time across all servers as a power consumption metric. Besides, we evaluate the resource imbalance degree of the cluster by calculating the variance of resource utilization across all servers, which is also important for the data center. A lower variance indicates a more balanced distribution of resources throughout the cluster. We further report the buffer queuing delay incurred by the BPJ stream, which captures the end-to-end execution cost of enabling interference-aware reordering. The two workload classes play asymmetric roles in our setting: LRA tasks are the target of interference protection and are evaluated by CPI, whereas BPJ tasks are the throughput-oriented workload being scheduled. BPJ task durations are taken from the trace and are independent of placement, so the only execution cost the reordering option imposes on the BPJ side is the time a task spends in the buffer before dispatch, measured as the number of batches it waits within its arrival timeslot. We report this queuing delay and compare it against the execution time of the BPJ tasks to assess the cost of interference-awareness relative to the runtime of the affected workload.

5.2. Experimental results

In this section, we evaluate the effectiveness of MixloadSched. We compare MixloadSched against baseline algorithms. To thoroughly validate its performance across clusters of varying scales, we vary the number of servers from 5 to 1000. Figs. 4 and 5 present the cluster-scale sweep as grouped bar charts with error bars. For each cluster scale, the bar height reports the mean of the corresponding metric across replications, and the error bar denotes ± 1 standard deviation.

Fig. 6 plots CPI against average power over all (method, dataset, scale) configurations and marks the global Pareto front. MixloadSched is the only method appearing on the front in both datasets (SYNTH s = 500, REAL s = 1000); PISM attains the lowest CPI (SYNTH s = 1000, 0.1023) and PEAP the lowest power (REAL s = 500, 90.86 W), each sacrificing the other axis. All five front points lie at scale ≥ 100—small-scale runs (s = 5/50) are excluded by a high baseline power of ∼95 W—while the non-front REAL s = 100 cluster around 108 W reveals a structural power inflation that scheduling cannot recover from.

As shown in Fig. 4(a), on real traces MixloadSched attains the lowest or second-lowest CPI at every cluster scale. Relative to the strongest interference-aware baselines (Horus and PISM), the average CPI reduction is about 3.3%, rising to 10.1% when averaged over all five baselines, which include Round-Robin and the production Default. At the smallest scale, the methods are close, as greedy placement is already adequate under low contention. On the synthetic traces (Fig. 5(a)), the ordering is mixed: PISM, which optimizes interference in isolation, attains a marginally lower CPI, and MixloadSched remains within about 1.2% of the strongest baselines. A per-condition significance analysis is reported at the end of this subsection.

As shown in Figs. 4(b) and 5(b), MixloadSched reduces average power by about 1% on real traces relative to the strongest baselines, with a smaller margin on synthetic traces. Its power consumption is comparable to the energy-specialized PEAP, which minimizes power directly but at a higher interference and imbalance, indicating that the energy term in the joint cost recovers most of the available power savings without a dedicated consolidation pass.

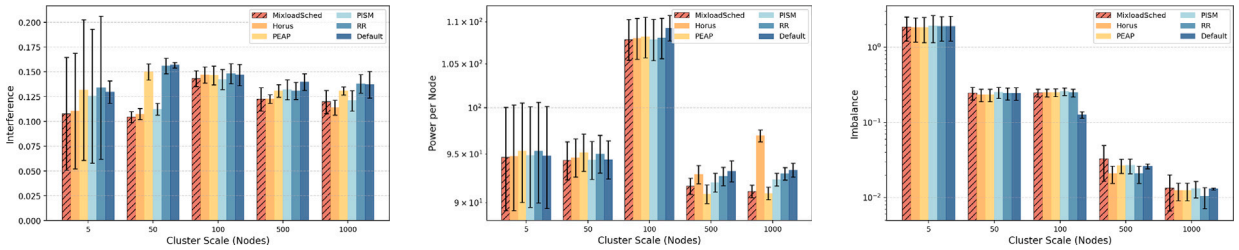
As shown in Figs. 4(c) and 5(c), MixloadSched does not improve resource balance. Its imbalance is comparable to the baselines at small and medium scales and higher than the strongest baselines at the largest scales (most pronounced at 500–1000 servers). This follows from the cost function in Eq. (10), which weights only interference and energy: balance is affected only indirectly, and steering utilization toward the per-server efficiency sweet spot does not by itself reduce utilization variance across nodes. We report this metric for completeness; adding an explicit balance term to the cost is left to future work.

To examine how the interference-awareness option affects overall execution performance on the BPJ side, we decompose the per-task completion time as

$$\text{finish}(b_j) = \text{arrival}(b_j) + w_j^{\text{queue}} + t^{\text{sched}} + d_j, \tag{15}$$

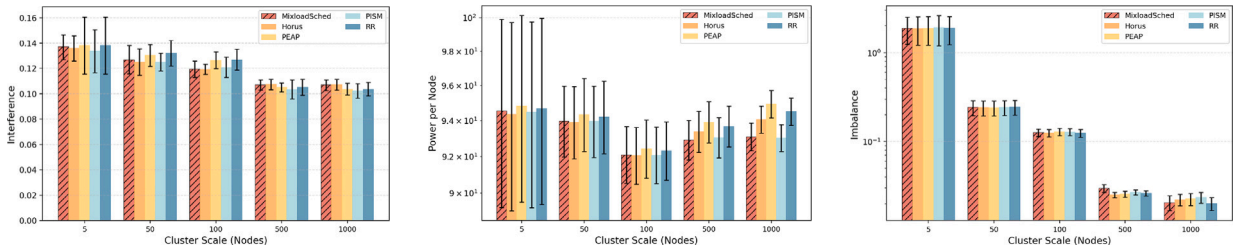
where d_j is the trace-determined execution duration, t^{sched} is the per-task scheduling overhead, and w_j^{queue} is the time the task spends in the reordering buffer before dispatch. The execution duration d_j is fixed by the trace and identical across algorithms, and resource-wait at dispatch is absent under the evaluated loads because the candidate server set S_{cand} is non-empty at every dispatch decision. The component the reordering option introduces is therefore w_j^{queue}, which we characterize next.

Within each timeslot, the tasks that arrive are scheduled in batches of size L: the buffer reorders and dispatches up to L tasks at a time, so a task placed in the kth batch of its timeslot waits for the $k-1$ preceding



(a) Interference cost. (b) Power consumption. (c) Imbalance degree.

Fig. 4. Evaluation metrics under different cluster scales (Real traces).



(a) Interference cost. (b) Power consumption. (c) Imbalance degree.

Fig. 5. Evaluation metrics under different cluster scales (Synthetic data).

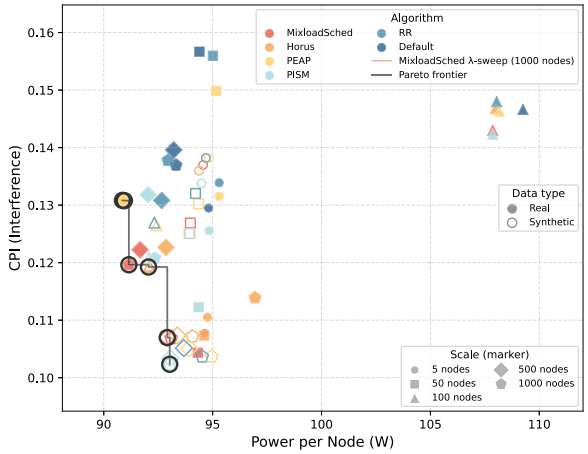


Fig. 6. CPI–power trade-off and Pareto front across all configurations.

batches before being dispatched. Its queuing delay is therefore the number of batches it waits multiplied by the per-batch processing time, where a batch of L tasks is processed at the per-task overhead t^{sched} (Table 8), i.e. $L \cdot t^{\text{sched}}$ per batch. Table 4 reports the measured average waiting batches and the resulting queuing delay across the five cluster scales. Taking the 100-server cluster as an example: at the default queue length $L = 29$, a task waits 3.44 batches on average, the per-batch processing time is $29 \times 9.91\,\mu\text{s} \approx 0.29$ ms, and the average queuing delay is about 0.99 ms. The average queuing delay grows with scale but remains at the millisecond level, reaching 14.4 ms at 1000 servers. Even the busiest timeslot is fully drained well within the one-hour timeslot at every scale (at most 95 ms at 1000 servers, Table 4); tasks are thus dispatched within their arrival timeslot and are never held across timeslots.

We compare this queuing delay against the execution time of the BPJ tasks. The execution time is heavy-tailed: the median is about 12 s and the mean about 50 s, while the tail extends to 10^4 s. Against a median execution time of 12 s, the average queuing delay ranges from

Table 4
BPJ Buffer queuing delay across cluster scales.

Cluster size	5	50	100	500	1000
Queue length L	4	16	29	115	216
Per-task overhead t^{sched} (μs)	4.99	7.70	9.91	12.74	14.19
Avg. waiting batches per task	0.67	2.95	3.44	4.44	4.69
Per-batch time $L \cdot t^{\text{sched}}$ (ms)	0.02	0.12	0.29	1.47	3.07
Avg. queuing delay $\bar{w}$ (ms)	0.01	0.36	0.99	6.50	14.38
Busiest-timeslot dispatch (ms)	0.14	2.46	6.61	42.49	95.02
$\bar{w} / d_{\text{median}}$ (%)	0.0001	0.003	0.008	0.05	0.12

0.01 ms at 5 servers to 14.4 ms at 1000 servers, i.e. below 0.12% of the median execution time at every scale (last row of Table 4). The reordering buffer operates at the time scale of scheduling decisions (μs per task), whereas BPJ tasks execute at the scale of seconds to hours, so reordering changes the dispatch order of a batch without materially affecting any task's completion relative to its own runtime.

On the LRA side, this reordering yields the differences reported above, an average CPI reduction of about 3.3% and an average power reduction of about 1% on real traces relative to the strongest baselines. BPJ tasks are best-effort jobs without a strict response-latency target, and the queuing delay introduced on the BPJ side is orders of magnitude shorter than their execution time. The interference-awareness option therefore trades a millisecond-scale queuing delay on the throughput-oriented BPJ workload for the QoS improvement on the latency-sensitive LRA workload.

To verify that these differences are systematic rather than run-to-run noise, every configuration is repeated five times at each of the five scales and on both workload sources, and MixloadSched is paired against each baseline across the resulting ten conditions using the Wilcoxon signed-rank test with a Bonferroni correction over the fifteen comparisons [32]. The differences are small and only partly significant: after correction, the single significant result is the power reduction over Round-Robin (+0.75%, $p_{\text{Bonf}} = 0.029$), while none of the CPI or power differences against the strongest interference-aware baselines, Horus and PISM, is distinguishable ($p_{\text{Bonf}} > 0.7$); the CPI differences over Round-Robin and PEAP reach $p \approx 0.02$ before correction but not after.

Table 5
Ablation of the reordering mechanism (mean ± std over 5 runs; lower is better). Priority is the full MixloadSched buffer; No-queue schedules each task on arrival; Random buffers but reorders randomly. Power is per active server. Best of the three is bold.

Data	Scale	CPI			Power (W/server)		
		Priority	No-queue	Random	Priority	No-queue	Random
Real	5	**0.1077 ± 0.0569**	0.1255 ± 0.0675	0.1310 ± 0.0705	**94.64 ± 5.44**	94.84 ± 5.28	99.53 ± 5.54
	50	**0.1043 ± 0.0055**	0.1123 ± 0.0060	0.1170 ± 0.0062	**94.32 ± 1.99**	94.35 ± 1.99	98.27 ± 2.08
	100	0.1430 ± 0.0079	**0.1424 ± 0.0101**	0.1489 ± 0.0106	107.86 ± 2.43	**107.86 ± 2.44**	112.94 ± 2.56
	500	**0.1222 ± 0.0117**	0.1318 ± 0.0102	0.1382 ± 0.0107	**91.67 ± 0.81**	92.04 ± 0.94	95.92 ± 0.98
	1000	**0.1196 ± 0.0117**	0.1208 ± 0.0103	0.1260 ± 0.0107	**91.15 ± 0.64**	92.35 ± 0.64	96.52 ± 0.67
Synth	5	0.1370 ± 0.0098	**0.1338 ± 0.0171**	0.1401 ± 0.0179	94.56 ± 5.34	**94.48 ± 5.27**	98.83 ± 5.51
	50	0.1269 ± 0.0112	**0.1251 ± 0.0071**	0.1301 ± 0.0074	93.98 ± 2.00	**93.96 ± 2.02**	98.53 ± 2.11
	100	**0.1194 ± 0.0065**	0.1208 ± 0.0081	0.1257 ± 0.0084	92.10 ± 1.58	**92.08 ± 1.57**	96.66 ± 1.65
	500	0.1070 ± 0.0040	**0.1034 ± 0.0076**	0.1077 ± 0.0079	**92.92 ± 1.09**	93.05 ± 1.13	96.95 ± 1.18
	1000	0.1069 ± 0.0042	**0.1023 ± 0.0058**	0.1069 ± 0.0060	93.10 ± 0.77	**93.03 ± 0.74**	97.02 ± 0.77

Resource imbalance is not reduced—the paired difference favors the baselines—consistent with the cost function in Eq. (10), which targets interference and energy and affects balance only indirectly. Considering the three metrics jointly, a Friedman test on the per-cell normalized composite does not separate the leading group of MixloadSched, Horus and PISM at the 0.05 level ($\chi^2 = 9.04$, $p = 0.060$), and MixloadSched is Pareto-non-dominated in seven of the ten conditions; restricted to the two optimized objectives (interference and energy), it ranks in the leading pair with PISM. MixloadSched thus matches the strongest interference-aware baselines on QoS and energy and ranks ahead of the energy-only (PEAP) and naive (Round-Robin) schedulers, within the sub-15 μs per-task budget reported in Table 8.

5.3. Ablation study

To validate the effectiveness of the proposed queuing mechanism, we conducted ablation studies to assess its impact on both system performance and power consumption. Table 5 compares the full priority buffer against scheduling each task on arrival (no-queue) and against random reordering. Random reordering is the worst of the three in every condition, on both CPI and power (about 4%–5% higher power and up to 18% higher CPI), confirming that the order in which a batch is dispatched materially affects placement quality. Relative to no-queue, priority reordering lowers real-trace CPI, most clearly at small and medium scales (14.2% at 5 servers, and 7.1%/7.3% at 50/500 servers); on power and on the synthetic workload the two are comparable. The reordering mechanism therefore contributes mainly by avoiding poor dispatch orders and by improving real-trace interference, rather than by reducing power over immediate scheduling.

5.4. Parameter analysis

The queue length L sets the batch size used to schedule the tasks arriving within a timeslot. A larger L clears the same arrivals in fewer batches, so each task waits behind fewer preceding batches. Fig. 7 shows, for the 100-server cluster, the average number of batches a task waits as a function of L: it decreases from 9.76 at $L = 11$ to 2.46 at $L = 39$. It holds because under batch scheduling L determines the number of batches. Translating to physical time, each batch takes $L \cdot t^{\text{sched}}$ to process, so the two effects—fewer batches but a larger per-batch time—nearly cancel, and the physical queuing delay stays about 1 ms across the range (right axis). Increasing L therefore widens the reordering horizon while keeping the per-task queuing delay at the millisecond scale, so a larger reordering window does not come at the cost of additional queuing latency. This decoupling is what allows L to be set primarily to match the arrival rate (Table 6): L is chosen so that the buffer fills approximately once per timeslot, which fixes the reordering granularity without introducing a latency penalty.

Task arrival characteristics serve as a prior input for determining the algorithm's queue length, rather than a result of running the algorithm. Concretely, before deployment we profile the trace once to obtain

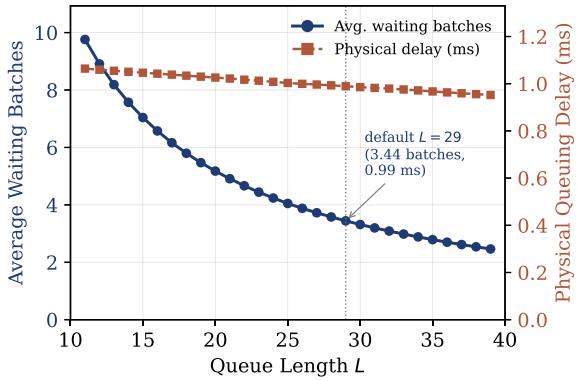


Fig. 7. Average waiting batches and physical queuing delay versus queue length L (100-server cluster).

Table 6
Average task arrival rates per effective timestamp and corresponding queue lengths.

Cluster scale	5	50	100	500	1000
Avg. Arrivals	3.57	16.18	28.94	115.32	215.62
Queue length L	4	16	29	115	216

the average arrival rate per effective timestamp, i.e. a timestamp at which at least one BPJ task arrives; this profiling is performed on the historical trace independently of MixloadSched and its output is fed in as a configuration parameter to the buffer. The arrival-rate statistics for the five evaluated cluster scales are reported in Table 6: these values describe the workload itself (how many BPJ tasks arrive per effective timestamp at each cluster scale), and are computed once from the trace prior to scheduling. We then set L to match the average per-effective-timestamp arrival volume so that, in expectation, the buffer fills approximately once per effective timestamp; the resulting queue lengths for each scale are listed in Table 6.

The reordering operation is the principal source of buffer-induced overhead, as each trigger entails an $O(L \log L)$ sort together with L classifier inferences (Algorithm 2). Fig. 8 reports the measured reordering rate, defined as the number of reorder triggers per effective timestamp, against the average arrival rate. With the queue length configured to track the arrival rate, the reordering rate saturates at approximately 1.7 triggers per effective timestamp for clusters of 50 servers and above and is largely independent of cluster scale. The rate is driven mainly by the arriving batch filling the buffer to its capacity L; the timeout bound $T_{\max}$ contributes only the residual fallback trigger that flushes any sub-L remainder, so that no task is held across timeslots when arrivals are too sparse to fill the buffer. At the smallest scale (5 servers) arrivals are sparse and the rate drops to 1.41. Since the per-trigger cost is small,

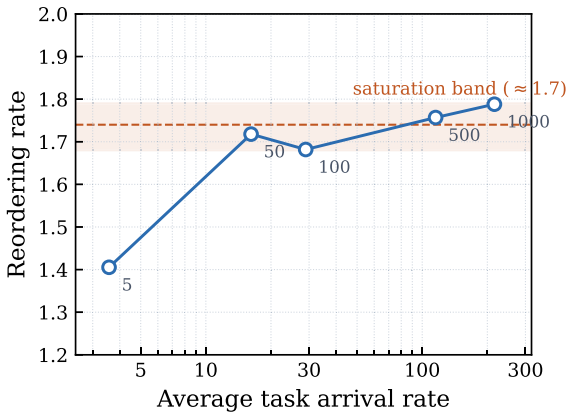


Fig. 8. Reordering rate versus average task arrival rate.

this bounded reordering rate keeps the end-to-end scheduling overhead in Table 8 15 µs even at 1000 servers.

We emphasize that L is not tuned for solution quality but is fixed by a measurable workload property—the per-timeslot arrival rate (Table 6)—so it is not a free hyperparameter. Its effect on latency is bounded and essentially flat across the practical range (Fig. 7), while the benefit of the reordering it enables, relative to no buffering or random reordering, is quantified by the ablation in Table 5.

5.5. Interference model selection

The Estimator relies on a learned model to predict the CPI-based interference an LRA task experiences given the type-count vector of colocated BPJ tasks (Section 4). Because this predictor is queried once per candidate server for every scheduled task, both its accuracy and its per-query inference latency directly affect scheduling quality and overhead. We therefore conduct a dedicated study to justify the choice of model.

Setup. We extract training samples from the Alibaba trace, where each sample pairs the per-type colocation counts on a server with the measured mean CPI of the LRA tasks on that server, yielding an 18-dimensional feature representation that aggregates the 27 task types together with machine-level and temporal context. To avoid temporal leakage, samples are split chronologically into 70% training, 10% validation, and 20% test. We compare four candidate models: Linear Regression, an MLP (two hidden layers, 64–32 units), XGBoost (gradient boosting), and Random Forest; XGBoost and RF use the same ensemble size (30 trees, depth 8) for a fair comparison. All models share the same features and splits. We report accuracy as Test RMSE and Mean Absolute Error (MAE), the Spearman rank correlation ρ between predicted and true interference, and the Top-10% hit rate (the fraction of truly lowest-interference servers correctly identified among the predicted lowest-interference candidates), each averaged over five random seeds. Inference latency is measured under the online, single-sample, single-thread regime that matches the scheduler's invocation pattern: after a warm-up phase, each model performs single-sample predictions, and we report the median (P50) and tail (P99) latency per query. For the tree ensembles, which are deployed as native code compiled via Treelite, latency is measured on the compiled model; compilation leaves predictions unchanged. Training time is also reported, as the interference model is periodically retrained.

Results. Table 7 reports the comparison. All four models attain very similar accuracy: RMSE ranges from 0.2311 (XGBoost) to 0.2331 (Linear Regression), a spread under 1% that is comparable to the cross-seed standard deviation (±0.001). The two tree ensembles (XGBoost, RF) are marginally more accurate than Linear Regression and statistically indistinguishable from each other (RMSE 0.2311 vs. 0.2314, within one standard deviation), and they share nearly identical ranking quality ($\rho \approx 0.35$, Top-10% ≈ 27.6%). The key practical differences lie in latency and training cost. Compiling the trained trees to native code via Treelite is essential here: it reduces per-query inference latency by over two orders of magnitude (from about 2000 µs to 16.9 µs for RF), bringing both ensembles to ∼17 µs median latency—lower than the MLP (75.1 µs) and even Linear Regression (51.1 µs)—with tight tails (P99 within 1.4× of P50) and without altering predictions. On training cost, XGBoost is fastest among the ensembles (1.9 s vs. 15.7 s for RF), owing to its histogram-based split finding.

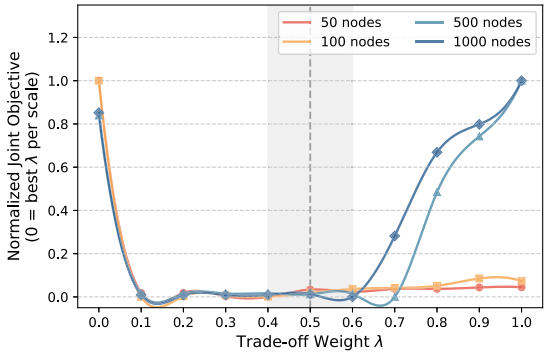
Analysis. After compilation, XGBoost and RF occupy the accuracy–latency frontier in Table 7: Linear Regression and MLP are dominated on both accuracy and latency, leaving the two ensembles as the only sensible choices. Between them, the accuracy gap is 0.13% in RMSE—within one cross-seed standard deviation and thus not statistically meaningful—and their compiled inference latencies are effectively equal (16.9 vs. 17.1 µs). On the two axes the scheduler is directly sensitive to, namely predictive quality and per-query latency, the two models are interchangeable. XGBoost holds a training-time advantage (1.9 vs. 15.7 s), but this is an offline, periodic cost that does not lie on the scheduling hot path and therefore does not affect runtime scheduling quality. We adopt Random Forest for its conceptual simplicity and smaller, less sensitive hyperparameter footprint: it is governed primarily by the number and depth of trees, whereas gradient boosting additionally requires tuning the learning rate, subsampling ratios, and regularization terms, complicating deployment across heterogeneous clusters. The variance reduction of bagging is also well matched to the low signal-to-noise nature of CPI prediction, where a substantial fraction of the variance is irreducible measurement noise (all models cluster within a 1% RMSE band, indicating the task is noise-limited rather than model-limited). Crucially, because the Estimator exposes a fixed (count-vector → interference) interface, gradient boosting can be substituted with no change to the scheduling framework; the framework's contribution is therefore independent of this particular choice. This experiment also confirms that the Estimator's compiled inference cost (16.9 µs) is consistent with the negligible per-task model-inference overhead reported in Table 8.

5.6. Sensitivity to the trade-off weight λ

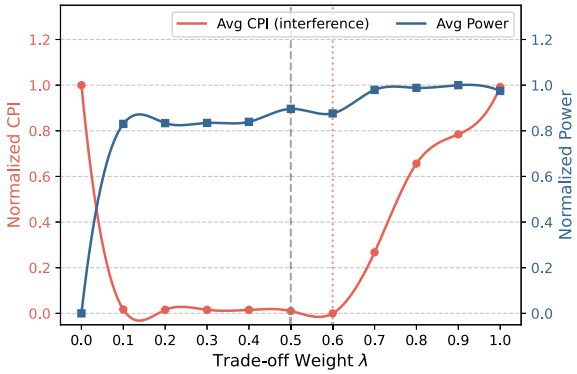
The trade-off weight $\lambda \in [0, 1]$ in $\text{Cost}(k) = \lambda \cdot \hat{C}_{\text{int}}(k) + (1-\lambda)\cdot(1-P_k)$ balances interference suppression against energy efficiency. To verify that the default $\lambda = 0.5$ is not a fragile artifact of hand-tuning, we sweep $\lambda \in \{0.0, 0.1, \ldots, 1.0\}$ at every evaluated cluster scale (5–1000 servers) on the real trace and record CPI, average power, and resource imbalance. For each scale we form the normalized joint objective $J(\lambda) = \frac{1}{2}\left(\text{CPI}/\text{CPI}_{\max} + \text{Power}/\text{Power}_{\max}\right)$, with the maxima taken over the sweep at that scale. The results are summarized in Fig. 9. Three observations emerge. First, λ acts as a meaningful Pareto-control knob rather than a noise term: average power increases with λ (it is lowest at the pure-energy extreme $\lambda = 0$), confirming that down-weighting the energy term costs efficiency. The interference response, however, is not monotone—it is U-shaped with an interior minimum (Fig. 9(b)). At 1000 servers the CPI minimum occurs near $\lambda = 0.6$, and the pure-interference extreme $\lambda = 1.0$ yields 3.6% higher CPI than this minimum. The reason is that the energy/balance term also discourages co-locating heavy tasks and thereby incidentally suppresses realized interference; removing it entirely ($\lambda = 1$) degrades CPI as well. The two objectives are therefore complementary rather than purely adversarial, which is exactly the behavior the joint formulation is designed to exploit. Second, and most important for the credibility of the default setting, the joint objective exhibits a broad flat basin around the middle of the range. Across all non-degenerate scales (50–1000 servers), $\lambda = 0.5$ lies within 0.03% of the per-scale optimum of J, whereas the single-objective extremes are clearly dominated: $\lambda = 0$ is 0.3%–1.5% worse, and $\lambda = 1$ is up to 1.8% worse at the larger scales. Hence $\lambda = 0.5$ is

Table 7
Comparison of candidate interference-prediction models.

Model	RMSE (±std)	MAE (±std)	ρ	Top-10%	Train (s)	P50 (µs)	P99 (µs)
Linear Reg.	0.2331 ± .0009	0.1682 ± .0007	0.345	25.9%	**0.11**	51.1	56.8
MLP (64-32)	0.2312 ± .0013	0.1680 ± .0006	0.355	27.5%	27.6	75.1	81.3
XGBoost	**0.2311** ± .0011	**0.1673** ± .0006	**0.354**	27.6%	1.94	17.1	22.5
RF (ours)	0.2314 ± .0011	0.1680 ± .0006	0.350	27.6%	15.7	**16.9**	**22.5**



(a) Normalized joint objective $J(\lambda)$ across cluster scales (50–1000 servers).



(b) CPI and power response at 1000 servers: CPI reaches minimum near $\lambda \approx 0.6$, while power increases monotonically with λ.

Fig. 9. Sensitivity to the trade-off weight λ.

not a knife-edge choice; any value within the basin yields essentially equivalent system-level outcomes, and an operator may shift along it according to whether the deployment prioritizes QoS or energy. In production, λ can be re-tuned per cluster by a one-time offline sweep on representative trace data; we report all results in this paper with the single fixed default $\lambda = 0.5$ so that every comparison uses one parameter setting. Resource imbalance also generally improves with λ and remains low at $\lambda = 0.5$; we omit it from the figure for brevity. Third, the smallest scale is a degenerate special case and is omitted from Fig. 9(a): with only five servers the placement is essentially fixed and the metrics barely respond to λ, consistent with the observation in Section 5 that greedy placement already suffices at small scale. At 50 and 100 servers the CPI metric is flat for all $\lambda \geq 0.1$ because CPI is discretized into ordinal severity levels (Section 4): once interference receives any positive weight the level-optimal placement is selected and the coarse level metric cannot resolve finer differences, while the continuous power metric still varies with λ.

Table 8
Algorithm overhead.

Cluster size	5	50	100	500	1000
Time overhead (µs)	4.99	7.70	9.91	12.74	14.19

5.7. Overhead

The computational overhead of the algorithm comprises classification inference time, evaluation time, queue rescheduling time, and server matching time. Among these, the first two components are negligible due to the high efficiency of model inference. Table 8 presents the algorithmic overhead across varying scales. It is observed that the overhead increases marginally as the scale expands; however, this increase remains well within acceptable limits.

6. Conclusion

This paper proposes MixloadSched to address performance interference and energy inefficiencies in mixed workload colocation. By introducing a delayed reordering strategy and a joint interference-energy evaluation model based on Random Forest and ESS, the framework synergistically optimizes LRA performance and cluster energy efficiency. Experimental results confirm that MixloadSched effectively reduces interference costs and energy consumption while maintaining extremely low scheduling latency, offering a promising solution for large-scale cloud-native environments.

Despite the demonstrated effectiveness of MixloadSched, this study acknowledges certain limitations. Our interference prediction model primarily targets CPU and memory resources. The performance impact of I/O-intensive bottlenecks—such as network bandwidth and disk storage—as well as heterogeneous accelerators (e.g., GPUs and TPUs) remains to be fully integrated. Future work will focus on two directions. First, we plan to extend the framework to support heterogeneous infrastructure, incorporating interference modeling for GPUs and FPGAs to accommodate cloud-native AI workloads. Finally, leveraging the ultra-low latency of our scheduling algorithm, we intend to investigate its applicability and energy efficiency in resource-constrained edge computing scenarios.

CRediT authorship contribution statement

Jianzhuo Li: Writing – review & editing, Writing – original draft, Methodology. **Weiwei Lin:** Writing – review & editing, Methodology. **Haijie Wu:** Writing – review & editing. **JiaHe Chen:** Writing – review & editing, Methodology. **Duanyang Du:** Writing – review & editing, Methodology. **Keqin Li:** Writing – review & editing.

Declaration of competing interest

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

Acknowledgments

This work is supported by Guangxi Key Research and Development Project (2024AB02018), The Innovative Development Joint Fund of Natural Science foundation in Shandong Province (ZR2024LZH012), Guangdong Provincial Natural Science Foundation Project (2025A1515010113) and the Major Key Project of PCL, China under Grant PCL2025A08 and PCL2025A11.

Data availability

Data will be made available on request.

References

[1] International Energy Agency, Energy and ai, 2025, URL: https://www.iea.org/reports/energy-and-ai. (Accessed 1 January 2026).
[2] Qianzhan Intelligence, Analysis of regional competition and market share in China's data center industry in 2021, 2021, URL: https://bg.qianzhan.com/report/detail/300/210926-3675a71a.html. (Accessed 1 January 2026).
[3] H. Huang, W. Lin, J. Lin, K. Li, Power management optimization for data centers: A power supply perspective, IEEE Trans. Sustain. Comput. (2025).
[4] M. Zakarya, L. Gillam, K. Salah, O. Rana, S. Tirunagari, R. Buyya, Colocateme: Aggregation-based, energy, performance and cost aware vm placement and consolidation in heterogeneous iaas clouds, IEEE Trans. Serv. Comput. 16 (2) (2022) 1023–1038.
[5] L.M. Al Qassem, T. Stouraitis, E. Damiani, I.M. Elfadel, Containerized microservices: A survey of resource management frameworks, IEEE Trans. Netw. Serv. Manag. 21 (4) (2024) 3775–3796.
[6] T. Zheng, P. Yang, Q. Wang, S. Zhang, H. Li, W. Lv, Cortex: Enhancing resource utilization in edge clusters through efficient co-location of LC and BE workloads, IEEE Internet Things J. 12 (10) (2025) 14733–14751, http://dx.doi.org/10.1109/JIOT.2025.3526607.
[7] G. Yeung, D. Borowiec, R. Yang, A. Friday, R. Harper, P. Garraghan, Horus: Interference-Aware and prediction-based scheduling in deep learning systems, IEEE Trans. Parallel Distrib. Syst. 33 (1) (2022) 88–100, http://dx.doi.org/10.1109/TPDS.2021.3079202, URL: https://ieeexplore.ieee.org/document/9428512/?arnumber=9428512.
[8] D. Yang, K. Zheng, S. Qian, Q. Hua, K. Zhang, J. Cao, G. Xue, Mitigating interference of microservices with a scoring mechanism in large-scale clusters, J. Supercomput. 81 (1) (2025) 104, http://dx.doi.org/10.1007/s11227-024-06534-7, URL: https://link.springer.com/10.1007/s11227-024-06534-7.
[9] W. Lin, W. Wu, L. He, An on-line virtual machine consolidation strategy for dual improvement in performance and energy conservation of server clusters in cloud data centers, IEEE Trans. Serv. Comput. 15 (2) (2022) 766–777, http://dx.doi.org/10.1109/TSC.2019.2961082.
[10] L. Liu, X. Dou, Y. Chen, Intelligent resource scheduling for co-located latency-critical services: A {Multi-Model} collaborative learning approach, in: 21st USENIX Conference on File and Storage Technologies, FAST 23, 2023, pp. 153–166.
[11] W. Peng, Y. Li, X. Liu, G. Wang, Lavender: An efficient resource partitioning framework for large-scale job colocation, ACM Trans. Archit. Code Optim. 21 (3) (2024) 1–23.
[12] S. Li, L. Wang, W. Wang, Y. Yu, B. Li, George: Learning to place long-lived containers in large clusters with operation constraints, in: Proceedings of the ACM Symposium on Cloud Computing, ACM, 2021, pp. 258–272, http://dx.doi.org/10.1145/3472883.3486971, URL: https://dl.acm.org/doi/10.1145/3472883.3486971.
[13] R. Nishtala, V. Petrucci, P. Carpenter, M. Sjalander, Twig: Multi-agent task management for colocated latency-critical cloud services, in: 2020 IEEE International Symposium on High Performance Computer Architecture, HPCA, IEEE, 2020, pp. 167–179.
[14] M. Xu, A.N. Toosi, R. Buyya, Ibrownout: An integrated approach for managing energy and brownout in container-based clouds, IEEE Trans. Sustain. Comput. 4 (1) (2018) 53–66.
[15] A. Al-Moalmi, J. Luo, A. Salah, K. Li, L. Yin, A whale optimization system for energy-efficient container placement in data centers, Expert Syst. Appl. 164 (2021) 113719.
[16] R.M. Canosa-Reyes, A. Tchernykh, J.M. Cortés-Mendoza, B. Pulido-Gaytan, R. Rivera-Rodriguez, J.E. Lozano-Rizk, E.R. Concepción-Morales, H.E. Castro Barrera, C.J. Barrios-Hernandez, F. Medrano-Jaimes, et al., Dynamic performance–energy tradeoff consolidation with contention-aware resource provisioning in containerized clouds, Plos One 17 (1) (2022) e0261856.
[17] R. Ranjan, I.S. Thakur, G.S. Aujla, N. Kumar, A.Y. Zomaya, Energy-efficient workflow scheduling using container-based virtualization in software-defined data centers, IEEE Trans. Ind. Inform. 16 (12) (2020) 7646–7657.
[18] A. Singh, G.S. Aujla, R.S. Bali, Container-based load balancing for energy efficiency in software-defined edge computing environment, Sustain. Comput.: Inform. Syst. 30 (2021) 100463.
[19] R. Chen, H. Shi, Y. Li, X. Liu, G. Wang, OLPart: Online learning based resource partitioning for colocating multiple latency-critical jobs on commodity computers, in: Proceedings of the Eighteenth European Conference on Computer Systems, 2023, pp. 347–364.
[20] Y. Feng, S. Shen, M. Xu, Y. Ren, X. Wang, V.C. Leung, W. Wang, Tango: Harmonious management and scheduling for mixed services co-located among distributed edge-clouds, in: Proceedings of the 52nd International Conference on Parallel Processing, 2023, pp. 595–604.
[21] T. Patel, D. Tiwari, Clite: Efficient and qos-aware co-location of multiple latency-critical jobs for warehouse scale computers, in: 2020 IEEE International Symposium on High Performance Computer Architecture, HPCA, IEEE, 2020, pp. 193–206.
[22] W. Xiang, Y. Li, Y. Ren, F. Jiang, C. Xin, V. Gupta, C. Xiang, X. Song, M. Liu, B. Li, et al., Gödel: Unified large-scale resource management and scheduling at bytedance, in: Proceedings of the 2023 ACM Symposium on Cloud Computing, 2023, pp. 308–323.
[23] Y. Hu, H. Zhou, C. de Laat, Z. Zhao, et al., Concurrent container scheduling on heterogeneous clusters with multi-resource constraints, Future Gener. Comput. Syst. 102 (2020) 562–573.
[24] R. Kabir, R. G. Kim, M. Nikdast, RISA: Round-robin intra-rack friendly scheduling algorithm for disaggregated datacenters, in: Proceedings of the SC'23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis, 2023, pp. 1512–1520.
[25] M. Carvalho, D.F. Macedo, Qoe-aware container scheduler for co-located cloud environments, in: 2021 IFIP/IEEE International Symposium on Integrated Network Management, IM, IEEE, 2021, pp. 286–294.
[26] H. Tian, Y. Zheng, W. Wang, Characterizing and synthesizing task dependencies of data-parallel jobs in alibaba cloud, in: Proceedings of the ACM Symposium on Cloud Computing, in: SoCC '19, Association for Computing Machinery, New York, NY, USA, 2019, pp. 139–151, http://dx.doi.org/10.1145/3357223.3362710.
[27] L. Breiman, Random forests, Mach. Learn. 45 (1) (2001) 5–32.
[28] F. Li, B. Hu, DeepJS: Job scheduling based on deep reinforcement learning in cloud data center, in: Proceedings of the 2019 4th International Conference on Big Data and Computing - ICBDC 2019, ACM Press, Guangzhou, China, 2019, pp. 48–53, http://dx.doi.org/10.1145/3335484.3335513.
[29] N. Schmitt, J. Bucek, J. Beckett, A. Cragin, K.-D. Lange, S. Kounev, Performance, power, and energy-efficiency impact analysis of compiler optimizations on the spec cpu 2017 benchmark suite, in: 2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing, UCC, IEEE, 2020, pp. 292–301.
[30] S. Luo, H. Xu, C. Lu, K. Ye, G. Xu, L. Zhang, Y. Ding, J. He, C. Xu, Characterizing microservice dependency and performance: Alibaba trace analysis, in: Proceedings of the ACM Symposium on Cloud Computing, 2021, pp. 412–426.
[31] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, J. Wilkes, CPI2: CPU performance isolation for shared compute clusters, in: Proceedings of the 8th ACM European Conference on Computer Systems, 2013, pp. 379–391.
[32] J. Demšar, Statistical comparisons of classifiers over multiple data sets, J. Mach. Learn. Res. 7 (2006) 1–30.

Jianzhuo Li received the B.S. degree in network engineering from Anhui University in 2018. He is currently pursuing a Ph.D. degree at the School of Computer Science and Engineering, South China University of Technology. His research interests mainly focus on cloud computing, energy-efficiency optimization, and container scheduling.

Weiwei Lin(Senior Member, IEEE) received the B.S. and M.S. degrees from Nanchang University in 2001 and 2004, respectively, and the Ph.D. in Computer Application from South China University of Technology in 2007. He has been a visiting scholar at Clemson University from 2016 to 2017. Currently, he is a professor in the School of Computer Science and Engineering at South China University of Technology. His research interests include distributed systems, cloud computing, and AI application technologies. He has published more than 200 papers in refereed journals and conference proceedings. He is currently served on the editorial boards of Future Generation Computer Systems, TBench, and Medical Data Mining. He is a distinguished member of CCF and a senior member of the IEEE.

Haijie Wu is currently working toward the Ph.D. degree in the School of Computer Science and Engineering, South China University of Technology, Guangzhou, China, supervised by Dr. Weiwei Lin. His research interests mainly include cloud edge collaboration, edge computing, and AI algorithms.

Jiahe Chen is an undergraduate student majoring in Mathematics and Applied Mathematics at South China University of Technology. His current interests include cloud computing, big data, and large language models.

Duanyang Du is an undergraduate student majoring in Data Science and Artificial Intelligence at South China Normal University, China. His current interests include cloud computing, big data systems, and natural language processing.

Keqin Li received a B.S. degree in computer science from Tsinghua University in 1985 and a Ph.D. degree in computer science from the University of Houston in 1990. He is a SUNY Distinguished Professor at the State University of New York and a National Distinguished Professor at Hunan University, China. He has authored or coauthored more than 1290 journal articles, book chapters, and refereed conference papers. He holds nearly 80 patents announced or authorized by the Chinese National Intellectual Property Administration. He is among the world's top few most influential scientists in parallel and distributed computing, regarding singleyear impact (ranked #2) and careerlong impact (ranked #3) based on a composite indicator of the Scopus citation database. He is listed in Scilit Top Cited Scholars and is among the top 0.02% out of over 20 million scholars worldwide based on topcited publications in the last ten years. He is listed in ScholarGPS Highly Ranked Scholars and is among the top 0.002% out of over 30 million scholars worldwide based on a composite score of three ranking metrics for research productivity, impact, and quality in the recent five years. He is a Member of Academia Europaea (MAE), a Member of the European Academy of Sciences and Arts (EASA), a Fellow of the American Association for the Advancement of Science (AAAS), a Member of the USA National Academy of Artificial Intelligence (NAAI), a Fellow of the International Academy of Artificial Intelligence Sciences (AAIS), a Fellow of the International Artificial Intelligence Industry Alliance (AIIA), a Member of the World Academy of Sciences (WAS), a Fellow of the World Academy of Engineering & Technology (WAET), a Fellow of the Institute of Electrical and Electronics Engineers (IEEE), a Fellow of the Asia Computational Intelligence Society (ACIS), and a Member of the SUNY Distinguished Academy.