Software: Practice and Experience

WILEY

RESEARCH ARTICLE

C-Koordinator: Interference-Aware Management for Large-Scale and Co-Located Microservice Clusters

Shengye Song[1,2] | Minxian Xu[2] | Zuowei Zhang[3] | Chengxi Gao[2] | Fansong Zeng[3] | Yu Ding[3] | Kejiang Ye[2] | Chengzhong Xu[4]

[1]Southern University of Science and Technology, Shenzhen, China | [2]Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences, Shenzhen, China | [3]Alibaba Cloud Computing, Beijing, China | [4]State Key Lab of IOTSC, Faculty of Science and Technology, University of Macau, Macau, China

Correspondence: Minxian Xu (mx.xu@siat.ac.cn)

Received: 31 October 2025 | **Revised:** 15 January 2026 | **Accepted:** 4 February 2026

Keywords: co-location | interference | Koordinator | microservices

ABSTRACT

Objective: Microservices transform traditional monolithic applications into lightweight, loosely coupled application components and have been widely adopted in many enterprises. Cloud platform infrastructure providers enhance the resource utilization efficiency of microservices systems by co-locating different microservices. However, this approach also introduces resource competition and interference among microservices. Designing interference-aware strategies for large-scale, co-located microservice clusters is crucial for enhancing resource utilization and mitigating competition-induced interference. These challenges are further exacerbated by unreliable metrics, application diversity, and node heterogeneity.

Methods: In this paper, we first analyze the characteristics of large-scale and co-located microservices clusters at Alibaba and further discuss why cycle per instruction (CPI) is adopted as a metric for interference measurement in large-scale production clusters, as well as how to achieve accurate prediction of CPI through multi-dimensional metrics. Based on CPI interference prediction and analysis, we also present the design of the C-Koordinator platform, an open-source solution utilized in Alibaba cluster, which incorporates co-location and interference mitigation strategies.

Results: The interference prediction models consistently achieve over 90.3% accuracy, enabling precise prediction and rapid mitigation of interference in operational environments. As a result, application latency is reduced and stabilized across all percentiles (P50, P90, P99) response time (RT), achieving improvements ranging from 16.7% to 36.1% under various system loads compared with state-of-the-art system.

Conclusion: These results demonstrate the system's ability to maintain smooth application performance in co-located environments.

1 | Introduction

In the evolving landscape of cloud computing, the management of expansive cluster scale presents a unique set of challenges and opportunities. These clusters support a diverse suite of applications, each with distinct functional roles, resource preferences, and core-affinity requirements [1–3]. This diversity, while advantageous, introduces significant complexities in effective cluster management [4, 5]. A prominent issue in such a complex environment is the interference scenario where the performance of one application detrimentally affects another. This can manifest in various forms, from increased RT and reduced throughput to significant service latency, directly impairing user experience and potentially resulting in substantial business losses [6–8]. To

https://doi.org/10.1002/spe.70059

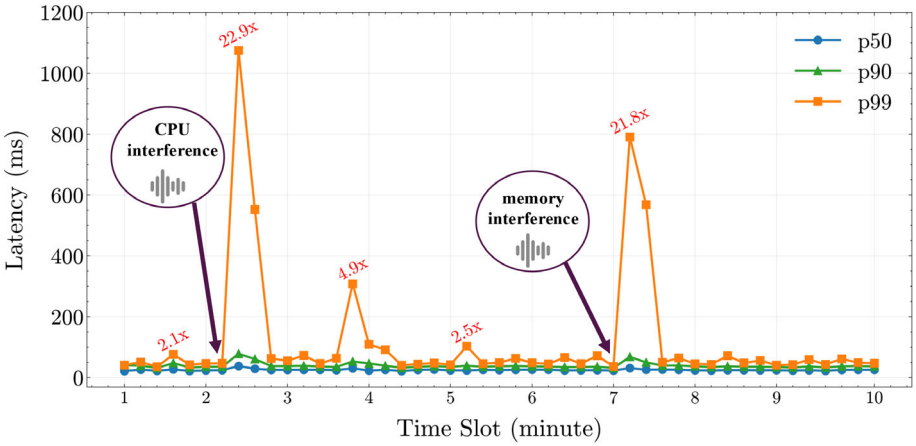


FIGURE 1 | Latency and SLO violations under interference in a production Nginx service. At a constant request rate of 100 QPS, the P99 latency remains relatively stable under normal conditions, with occasional fluctuations reaching 2–5 times the average baseline. However, when CPU or memory interference is injected, the P99 latency can spike up to 22.9×, leading to severe SLO violations and substantial degradation of user experience.

illustrate the real-world impact of such interference in a production environment, we conducted a controlled experiment using a Nginx-based online service deployed in Alibaba's production cluster. A steady input load of 100 QPS (query per second) was applied to the application, during which we introduced varying levels of CPU and memory interference. As shown in Figure 1, under normal conditions without interference, the P99 latency remained relatively stable, with occasional increased RT by 2.1× to 4.9× than the normal. However, once interference was injected, the latency surged dramatically, reaching up to 22.9× of the normal level. This resulted in severe SLO violations and substantial degradation in service quality, which underscores the severe performance degradation caused by interference, highlighting the importance of accurately predicting potential contention and ensuring stable application performance under co-location.

Additionally, interference can cause disproportionate resource allocation, leading to some nodes being overburdened while others remain underutilized [9, 10]. Such imbalances degrade overall resource efficiency and increase operational costs significantly [11]. Furthermore, the interdependent nature of cluster applications, such as microservices [12] with dependencies and share nodes, implies that a failure in one can disrupt others, potentially resulting in partial or total service outages.

Researchers have been dedicated to enhancing the resource efficiency of data centers through co-location, allowing multiple applications to share the same physical resources [13–16]. For example, Google's Borg system [14] employs advanced machine learning (ML) models to optimize task allocation and resource utilization. Similarly, Alibaba's Fuxi system [17] orchestrates resource management with precision, ensuring efficient utilization and supporting rapid business iteration. By co-locating multiple applications on the same physical resources, data centers can better utilize idle resources.

However, while co-location helps to maximize resource utilization, it also exacerbates the issue of interference [18–21], particularly due to uncontrolled competition for shared resources. This adds another layer of complexity, where not only latency-sensitive applications but also best-effort applications experience performance fluctuations. Furthermore, the inherent dynamic nature of resource utilization within these clusters makes the analysis of applications, nodes, and individual pods even more formidable. For instance, Alibaba's ecosystem [22, 23] comprises millions of nodes, each supporting diverse pods (e.g., can be up to 50 pods) with varying quality of service (QoS) demands (e.g., ranging from millisecond to minutes), resource consumption profiles, and shared resources. Although recent advancements in hardware isolation technologies have shown potential in mitigating interference [24–26], they are largely impractical to apply within Alibaba's infrastructure without considering co-located and large-scale application resource usage characteristics. As a result, interference-induced performance issues are becoming increasingly difficult to detect and predict [26].

In complex ecosystems such as Alibaba Cloud, which are now entirely based on microservices architectures, unhandled interference can significantly impact system performance [27, 28], most notably manifested as increased latency. This latency not only affects the real-time performance of applications but also indicates underlying resource competition that can disrupt the smooth operation of our systems. This paper outlines our approach within Alibaba's large-scale, co-located microservices clusters, where we leverage various low-level system metrics to train a predictive model. The model is designed to forecast potential interference by continuously monitoring system performance for anomalies. By detecting these anomalies early, the model enables proactive management of potential disruptions. Unlike existing interference-aware schedulers that rely primarily on reactive signals such as IPC, tail latency, or utilization thresholds, C-Koordinator leverages *predicted CPI* as a core signal for proactive interference-aware scheduling. To the best of our knowledge, C-Koordinator is the first system that systematically applies CPI-based prediction for interference management in large-scale, production co-located microservice clusters. By using CPI as a hardware-level, application-agnostic indicator and predicting it from fine-grained node-, pod-, and application-level metrics, C-Koordinator enables robust and scalable interference mitigation across diverse workloads.

The key contributions of this paper are as follows:

1. Characterization of Alibaba's applications: We characterize the features of Alibaba's large-scale and co-located applications, underscoring the complexities of detecting interference and identifying robust metrics and methodologies for its effective management. Our analysis of complex metrics aims to seek the optimal metrics and strategies for predicting and mitigating interference, thereby enhancing system resilience and operational efficiency.
2. CPI-based interference prediction model: We introduce a CPI-based interference prediction approach to identify the potential interference when applications are co-located. This predictive model achieves over 90.3% accuracy in forecasting interference for Alibaba's latency critical applications, while also balancing time costs of prediction.
3. Interference-aware management strategy: We present Alibaba's interference-aware management framework, C-Koordinator, to mitigate interference and ensure QoS. This framework seamlessly integrates into Alibaba's existing management infrastructure and has been validated for 4 years, enabling efficient and rapid detection of interference, with highly effective interference elimination results.

The structure of this paper is as follows: Section 2 reviews the challenges in resource management and interference mitigation in large-scale distributed systems, particularly within cloud-native environments. Section 3 introduces the architecture of C-Koordinator, an enhanced version of Alibaba's Koordinator, designed to address performance interference in microservice clusters. Section 4 details the design of the interference prediction and mitigation mechanisms, focusing on CPI-based prediction models. Section 5 presents experimental evaluations of C-Koordinator, demonstrating its effectiveness in optimizing resource allocation and minimizing performance degradation. Finally, Section 6 concludes with a summary of key findings and outlines future research directions. For clarity and ease of reference, the acronyms and notations used throughout this paper are summarized in Table 1.

TABLE 1 | Summary of acronyms and notations used in this paper and their corresponding meanings.

Symbol	Meaning
CPI	Cycles Per Instruction
CSI	CPI Severity Index for mitigation decisions
QoS	Quality of Service
SLO	Service Level Objective
PMU	Performance Monitoring Unit
BE	Best-Effort workload
LS	Latency-Sensitive workload
LSR	Latency-Sensitive Reserved workload
RT	Response Time
QPS	Queries Per Second
ΔCPI	Deviation between predicted CPI and reference CPI
U_n^t	Comprehensive resource utilization of node n at time t
C_n^t	CPU utilization of node n at time t
M_n^t	Memory utilization of node n at time t
$C_n^{s,t}$	Shared CPU pool utilization of node n at time t
TH_{select}	Threshold for interference candidate selection
TH_{CPI}	CPI-based interference detection threshold
I_p	Pod-level performance indicators
I_n	Node-level performance indicators
I_s	Application-level performance indicators
L_{current}	Current observed application latency

2 | Background and Related Work

This section highlights the characteristics of large-scale and co-located microservice cluster management with Alibaba's scenario, and discusses why the existing related approaches cannot satisfy our objectives.

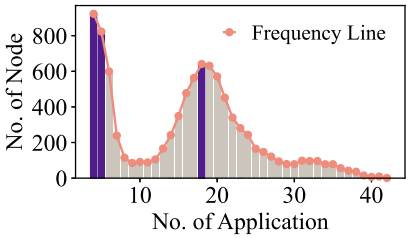
2.1 | Characterization of Alibaba's Microservice Clusters

We first demonstrate data derived from Alibaba realistic trace to show the co-location related characterization.

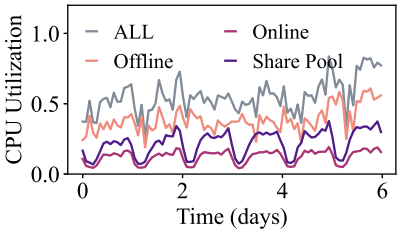
2.1.1 | Co-Location With Diverse Applications

Alibaba's microservice-based clusters support a variety of applications, such as Taobao (e-commerce), Alipay (financial services), and Alibaba Cloud (infrastructure). As shown in Figure 2a that illustrates the complexity and diversity of application distribution within a cluster, which depicts the Probability Density Function (PDF) of the number of applications running on each node, showing a notable variance with approximately 600 nodes hosting around 19 applications each, while about 1000 nodes manage only 4–5 applications. This distribution highlights the intricate and varied nature of node utilization across the cluster. Figure 2b presents the CPU utilization over several days for different categories of applications within a single node, including online (stringent QoS requirements for latency and availability), offline (not latency-sensitive and can tolerate being preempted, delayed, or scheduled flexibly), and shared pool (mixed latency requirement) usage. The diagram reveals substantial fluctuations in CPU usage, with high utilization rates that underscore the challenges in resource management and interference prediction within such a dynamic environment. The complexity of these patterns significantly complicates the analysis and prediction of potential resource conflicts and performance interference.

The applications in Alibaba's cluster are highly co-located, with over 60% of nodes running more than 15 applications each. These applications have distinct resource requirements and usage patterns. Over 30% of online service experienced resource contention during peak load periods. For instance, during high-traffic events like Singles' Day, e-commerce services can monopolize CPU and memory resources, causing contention and interference with other applications. Additionally, Alibaba's services demand varied QoS; real-time bidding services in advertising require extremely low latency, while batch processing jobs in data



(a) PDF shows the Node distribution by number of running applications.



(b) CPU utilization over time for different categories in Node.

FIGURE 2 | Co-located application distribution and CPU utilization patterns in Alibaba's cluster.

analytics can tolerate higher latency. Ensuring these diverse QoS requirements are met without mutual interference is a significant challenge, especially in a co-located environment. Alibaba's solutions must be exceptionally agile and responsive to handle these spikes without degrading the performance of critical services like Alipay. Existing solutions often struggle to dynamically reallocate resources efficiently across such diverse and fluctuating demands.

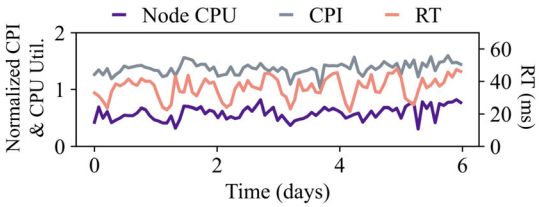


FIGURE 3 | Complexity in metrics collection and analysis.

2.1.2 | Complexity in Metrics Collection and Analysis

Alibaba's ecosystem generates vast amounts of metrics from user interactions, transaction logs, and system health indicators. Collecting, processing, and analyzing these metrics in real-time to detect interference and performance bottlenecks is challenging due to the sheer scale and diversity of data. Providing real-time performance insights for applications like Alibaba Cloud services requires advanced, scalable monitoring systems. These systems must detect and resolve interference swiftly to maintain QoS across a broad range of applications. Based on Alibaba's practice, interference should be detected within 1–2 s based on performance counters, and should be mitigated within 5–30 s. In most cases (over 90%), interference affecting online services should be fully resolved within 10 s, otherwise the users' experience could be significantly degraded.

Figure 3 illustrates the intricate relationship between resource utilization and performance metrics within Alibaba's cluster over a six-day period. The diagram shows the fluctuations in node CPU usage, alongside the CPI and average RT, showing asynchronous and inconsistent changes. The diverse fluctuations across these metrics highlight the dynamic nature of system performance, with notable variations that can be attributed to operational interference. The node CPU utilization curves demonstrate the variability in resource consumption, while the CPI index offers insights into the efficiency of CPU operations under different load conditions. Concurrently, the RT metric provides a direct measure of application responsiveness. This combined view highlights the complexity of correlating these metrics to effectively identify and mitigate performance interference.

The distinct behavior of each metric over the same timeline illustrates the challenges in collecting and analyzing data from multiple sources to diagnose and address potential system inefficiencies, including categories such as BE (Best Effort, without strict QoS requirement), LS (Latency Sensitive, with strict latency

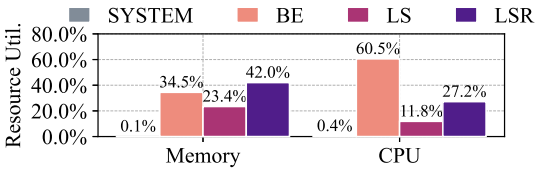


FIGURE 4 | CPU and memory usage across application types.

demand), LSR (Latency Sensitive Reserved, applications with reserved resources), and SYSTEM (operating system services). In addition, as shown in Figure 4, the high resource utilization across different application types increases the difficulty of maintaining resource isolation at the underlying level. This elevated resource usage raises the likelihood of interference, further complicating system management and performance optimization.

2.2 | Limitations of Existing Solutions

Kubernetes [29] has become the de facto standard for container orchestration, offering a comprehensive and scalable platform for managing microservices. Its key benefits include automated deployment and scaling of containers across clusters. Kubernetes simplifies the management of microservices architectures, providing features such as service discovery, load balancing, storage orchestration, automated rollouts and rollbacks, and self-healing capabilities [30]. Additionally, the portability and flexibility of Kubernetes make it an attractive choice for organizations seeking to adopt a cloud-native approach and manage their applications consistently across on-premises, hybrid, and multi-cloud environments.

While Kubernetes offers robust orchestration capabilities, it still falls short in interference management due to its lack of built-in mechanisms for handling resource contention. Its reliance on static resource allocation, such as CPU and memory limits, often

leads to inefficient resource use and performance degradation among co-located applications. Furthermore, Kubernetes' scheduler lacks the ability to adapt to the dynamic and fluctuating workloads of applications, resulting in resource contention. This issue is particularly problematic in complex, multi-tenant environments where minimizing interference is crucial for maintaining stable and predictable application performance. Although it integrates basic tools like Prometheus for monitoring, the absence of predictive models that forecast resource contention further limits its proactive capabilities, hindering its ability to prevent performance degradation.

Koordinator [31] cluster management system, an open-source project, aiming at enhancing the Kubernetes's monitoring and predictive capabilities. It is designed to optimize the co-location of microservices, AI, and big data workloads on Kubernetes. As a modern solution tailored for high-demand environments, Koordinator addresses the challenges of resource allocation, utilization, performance management, and interference mitigation in large-scale, diverse workloads. It achieves this through a combination of advanced scheduling techniques and resource management strategies, ensuring that resources are used efficiently while minimizing interference between co-located applications.

Although Koordinator has proven highly effective in large-scale production environments, it exhibits several limitations in interference detection. The system's proactive interference detection capabilities are relatively simple and lack the ability to detect interference with fine granularity and unified way for diverse applications. These limitations result in a coarse detection process that struggled to accurately capture subtle performance fluctuations in complex workloads.

To address these challenges, Koordinator required a more unified, fine-grained and accurate interference-aware scheduling mechanism. This paper introduces the C-Koordinator (CPI-based Koordinator) system, which builds on the foundation of Koordinator by enhancing its interference detection capabilities. C-Koordinator integrates more granular and precise interference sensing, utilizing advanced monitoring metrics and machine learning (ML) algorithms to predict interference based on CPI. These improvements enable the system to identify potential interference early, allowing for timely interventions that ensure better resource utilization and maintain QoS.

2.3 | Discussions of Related Work

The focus of current related work can be mainly divided into three categories, including performance metrics selection, predictive model selection, and interference prediction model.

2.3.1 | Performance Metrics Selection

To address the issue of application performance interference in co-location environments, it is required to timely detect or predict the interference [32]. One typical approach is to identify interference by monitoring the real-time latency of applications [33, 34], such as employing tail latency to detect interference [35, 36]. Due to the proprietary management practices of data centers and strict data privacy and security policies, it remains difficult to access fine-grained latency metrics for applications operating in production environments. Moreover, as an application-layer metric, tail latency is inherently influenced by workload characteristics such as QPS, which can fluctuate independently of underlying resource contention. This coupling makes it challenging to isolate and attribute performance degradation directly to interference effects in co-located systems. As a result, researchers often turn to alternative, indirect performance indicators—drawn from both the software system and hardware resource levels—to more accurately assess the presence and severity of interference in multi-tenant environments, like instructions per cycle (IPC) [37], requests waiting time [38], counter value per instruction (VPI) [39], system level entropy [40], or CPI [15, 41], which become crucial in addressing the interference. However, as the number of resource types to manage increases, the behavior of applications under multi-dimensional resource interference becomes more intricate. Solely relying on indirect indicators to separately evaluate interference in various resource dimensions makes it increasingly difficult to comprehensively assess an application's overall interference susceptibility.

The CPI metric has proven to be an effective indicator of interference for CPU-intensive applications (such as Cpi2 [41]) to capture resource contention. Prior work has explored various metrics for interference detection. For instance, Bubble-flux [37] identified a close correlation between the IPC metric and query latency, while Shenango [42] monitored thread and network packet queuing times to gauge CPU resource utilization. PerfCloud [38] focused on I/O interference by analyzing variance in blkio wait times, and Holmes [39] introduced the VPI metric to diagnose SMT interference in memory access using hardware performance events. Liang et al. [40] proposed System-Level Entropy as a method to quantify resource contention through entropy analysis across time series data. Additionally, in co-location environments, competition for shared resources spans multiple dimensions, as highlighted by works like Caladan [43] and Alita [44].

While these methods are insightful, they are limited in their applicability to large-scale clusters. Most of them rely on metrics that are either challenging to collect at scale, such as tail latency [2, 34, 45], or depend on detailed hardware monitoring, which may not be feasible in environments with privacy concerns or access restrictions. Moreover, some approaches are focused on specific environments that do not generalize well to the complexity of large-scale systems. For this reason, we utilize CPI, which offers a more scalable and holistic view of application interference, making it more suitable for large-scale and co-located microservice clusters.

2.3.2 | Predictive Model Selection

Given the difficulty of directly measuring interference effects in production cloud environments (e.g., restricted access to fine-grained application metrics and the dynamic, multi-tenant nature of modern data centers), researchers increasingly focus on predictive approaches to model and manage performance interference. Statistical techniques, machine learning models, and deep neural networks, such as decision trees and domain adversarial neural networks, have been employed to construct

interference models for applications [46, 47]. These predictive models capture the complex relationships between application performance and interference factors across multiple resource dimensions, enabling the estimation of performance variations under diverse workload and co-location scenarios. Compared to direct measurement, predictive models offer a scalable and non-intrusive solution, capable of generalizing to unseen interference patterns and dynamically adapting to workload fluctuations. Once interference is predicted or detected, resource management strategies, including dynamic resource allocation and intelligent job scheduling, can be applied proactively to mitigate performance degradation and ensure QoS.

In the process of constructing interference prediction models, a key step is collecting interference data, with one widely adopted technique being interference-injection-based data collection methods [47–49]. This method involves manually introducing interference to analyze applications offline and adjusting resource competition intensity on various shared resources to quantify the degree of application performance degradation. However, given the diversity of applications in large-scale environment, it is infeasible to explore all the cases manually. Currently, the existing prediction models are mainly based on single or limited metrics to predict interference. Additionally, how to balance the prediction accuracy and efficiency in large-scale environment is still an open question. In this work, we aim to provide Alibaba's practice on selection of interference metrics and prediction models.

2.3.3 | Interference Prediction Model

When applying traditional prediction models to microservice-based environments, the fine granularity and complex chains of microservices demand more sensitive interference detection mechanisms, finer resource isolation measures, and more efficient resource allocation strategies to ensure consistent application performance [50, 51]. While prior research has made significant progress in developing prediction models and orchestration frameworks for such environments, several limitations remain. For example, Lu et al. [52] developed Optum, a unified scheduler for large data centers, focusing on balancing resource utilization and scalability, but it does not fully address the fine-grained interference detection required for diverse, co-located microservices. Similarly, while Luo et al. [50] proposed optimization methods for microservice performance in interference-prone environments, their approaches lack real-time adaptability, which is essential for managing the dynamic nature of microservice chains. Adeppady et al. [53] introduced iPlace, a heuristic algorithm designed to minimize deployment interference, but this solution may prove insufficient in large-scale environments where manual adjustments become impractical. Frameworks such as Adrias [54] and Perph [55], which leverage deep learning for performance prediction, show promise but encounter overhead and scalability challenges in heterogeneous clusters with diverse workloads.

However, these models often rely on a limited set of metrics, which may not fully capture the multidimensional nature of interference in real-world, large-scale environments. In contrast, our work addresses these limitations by providing a more scalable and efficient approach to interference prediction and resource management, specifically designed for Alibaba's vast microservice architecture. We focus on selecting practical interference metrics and models that strike a balance between prediction accuracy and system efficiency.

Table 2 summarizes the key differences between C-Koordinator and existing schedulers in terms of metrics, prediction granularity, and deployment scalability, where the bold values indicate the results achieved by C-koordinator. Existing cluster schedulers and interference-aware systems differ significantly in terms of performance metrics, prediction granularity, and scalability. Kubernetes and Koordinator primarily rely on reactive signals such as response time and resource utilization, and do not provide proactive interference prediction. Prior research efforts have explored metrics such as IPC, tail latency, or entropy-based indicators for interference detection; however, these approaches are often sensitive to workload-specific behaviors and are difficult to deploy at production scale.

In contrast, C-Koordinator employs predicted CPI as a unified, hardware-level signal for interference-aware scheduling. By predicting CPI rather than reacting to instantaneous measurements, C-Koordinator enables early detection of performance degradation and proactive mitigation. Moreover, its design supports fine-grained prediction across node-, pod-, and application-level metrics, making it suitable for large-scale, heterogeneous, production environments.

3 | Design of C-Koordinator

In this section, we introduce the design of C-Koordinator to address the limitations of the existing approaches and systems. Proven in large-scale production environments within Alibaba for more than 4 years, C-Koordinator has demonstrated its robustness and effectiveness at scale. Alibaba's diverse range of services, including e-commerce, cloud computing, financial

TABLE 2 | Comparison of C-Koordinator with existing systems.

System	Metric used	Prediction	Granularity	Proactive	Production scale
Kubernetes	RT/Utilization	No	Node-level	No	Yes
Koordinator	RT/Utilization	No	Pod-level	No	Yes
IPC-based schedulers	IPC	Limited	Node/Task-level	Partial	No
Tail-latency-based	Tail RT	Limited	Service-level	Partial	No
C-Koordinator (ours)	**Predicted CPI**	**Yes**	**Node + Pod + App**	**Yes**	**Yes**

Note: Bold values highlighting best accuracy and training time

services, and AI tasks, rely on C-Koordinator to maintain high performance and efficiency across millions of containers.

3.1 | Overall Design Objectives

The discussions in Section 2 inspire us to achieve the following objectives to achieve interference-aware management for large-scale and co-located clusters:

3.1.1 | Unified Interference Detection

Our system aims to incorporate a unified framework for detecting resource contention and interference across diverse workloads. By leveraging comprehensive metrics such as CPI and other key performance indicators, the system provides a holistic view of interference patterns. This unified detection mechanism ensures that all potential sources of interference are identified promptly, regardless of the application type or resource demands.

3.1.2 | Proactive Management

To effectively manage co-located applications, the system must employ proactive resource management strategies. This includes real-time monitoring and predictive analytics to forecast potential interference before they impact application performance. By dynamically adjusting resource allocations based on these predictions, the system can mitigate interference preemptively, ensuring optimal application performance and resource utilization.

3.1.3 | Fine-Grained Monitoring

The system should offer comprehensive and granular monitoring capabilities that provide real-time insights into resource usage, performance metrics, and interference sources. Advanced monitoring tools and dashboards will facilitate the detailed analysis of resource consumption patterns, enabling quick identification and resolution of performance issues. This fine-grained monitoring ensures that even subtle interference effects are detected and addressed promptly and efficiently.

3.2 | Motivation of Using CPI

In large-scale, co-located microservice-based clusters such as those operated by Alibaba, monitoring and managing performance interference is a persistent challenge due to the dynamic nature of workloads, heterogeneous hardware configurations, and the sheer scale of deployment. Traditional high-level application performance metrics, such as RT and resource utilization, often fall short in this context. RT, while meaningful at the application layer, is influenced by a variety of external factors including QPS, network delays, and upstream/downstream service dependencies, making it difficult to directly attribute variations in RT to underlying resource contention or interference. Moreover, acquiring fine-grained RT data across thousands of production nodes is constrained by privacy, overhead, and access limitations. As our study highlights, the diversity of Alibaba's application workloads further exacerbates this issue, as different applications exhibit varying latency sensitivities and performance baselines, rendering RT an inconsistent and unreliable interference indicator in large, heterogeneous environments.

To address these challenges, CPI emerges as a more suitable and reliable metric for interference detection and prediction in large-scale, co-located clusters. However, instead of using CPI in isolation, we leverage multiple system-level indicators, such as memory utilization, network latency, and disk I/O, to predict CPI. This approach allows us to integrate multiple factors affecting CPI, providing a comprehensive prediction model that accounts for a wide range of potential performance bottlenecks. As a low-level performance counter available on modern processors, CPI reflects the average number of CPU cycles consumed per executed instruction, capturing the combined effects of CPU contention, memory access delays, cache interference, and other micro-architectural stalls. Unlike RT, CPI is largely independent of application-specific request patterns and external service dependencies, providing a uniform, hardware-level signal of performance degradation due to resource interference. Furthermore, CPI can be efficiently collected through lightweight, node-level monitoring tools without exposing sensitive application-layer information, making it scalable and practical for deployment in production-grade clusters. By predicting interference using CPI in conjunction with fine-grained hardware and software metrics across node-, pod-, and application-level data, cloud providers can implement more proactive, precise, and infrastructure-aware resource management strategies to mitigate performance degradation in multi-tenant environments. This allows for accurate interference prediction across a wide variety of workloads, from compute-heavy tasks to network- and data-bound applications, ensuring that CPI remains a suitable metric for diverse application types, with varying resource demands.

3.2.1 | CPI Collection and Overhead

C-Koordinator collects CPI using hardware performance counters available on modern processors. Specifically, lightweight kernel-level instrumentation (PMU counters exposed via perf-like interfaces) is used to periodically sample the number of retired instructions and CPU cycles at the node level. CPI is then computed as the ratio between these two counters.

To ensure scalability and low overhead in production environments, CPI is sampled at a coarse-grained interval (tens to hundreds of milliseconds), which is sufficient for interference prediction while avoiding high-frequency polling. Since the collection relies on hardware-supported counters and does not require application-level instrumentation or code modification, the measurement process is non-intrusive and efficient. In our production deployment, the CPU overhead introduced by CPI collection and related monitoring components remains within 1%−3%, as reported in Section 5.

3.3 | Architecture of C-Koordinator

As depicted in Figure 5, two components are inherited from Koordinator: ④ Koordinator Scheduler and ⑤ Koordlet located

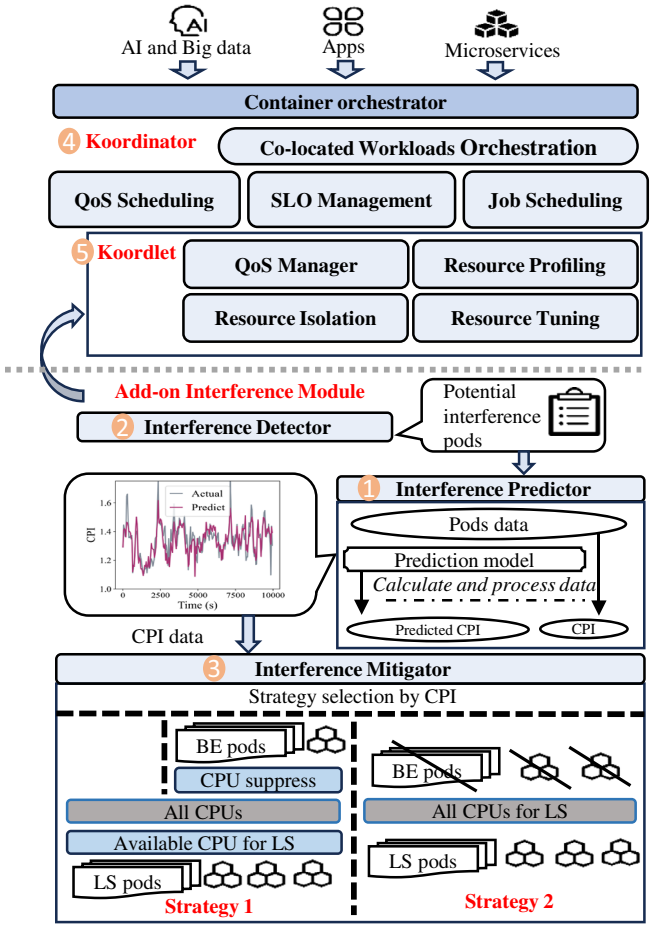


FIGURE 5 | C-Koordinator architecture overview.

above the dashed line. The Koordinator Scheduler, deployed as a Kubernetes Deployment, enhances resource scheduling with QoS-aware strategies, differentiated SLO management, load balancing, and resource overcommitment to optimize low-priority workloads. It also manages fine-grained CPU orchestration and QoS policies for memory and bandwidth. The Koordlet manages resource overcommitment, interference detection, and QoS guarantees through modules for CPU, memory, network, and disk profiling, isolation, and contention monitoring. The QoS Manager adjusts node co-location dynamically based on interference detection and SLO configurations, while resource tuning optimizes container performance.

Below the dashed line, in addition to the inherited components, C-Koordinator has significant enhancements to the original Kubernetes and Koordinator, tailored to meet the demands of Alibaba's co-located microservices clusters. By integrating an advanced interference prediction model into the system architecture, the system builds upon core functionalities like QoS Scheduling and SLO Management to deliver a more interference-aware orchestration framework.

The core strength of C-Koordinator lies in its Add-on Interference Module, which integrates three essential components, as highlighted in the diagram: ① Interference Predictor, ② Interference Detector and ③ Interference Mitigator. These modules interact seamlessly to manage performance and resource contention at every stage of the application lifecycle. Interference Predictor continuously monitors the system to predict early signs of resource contention. Once interference is detected, the CPI-based Interference Detector identifies potential performance degradation, allowing the system to take preemptive actions. The Interference Mitigator dynamically adjusts resource allocation to maintain high-priority application performance, especially during peak loads. It works alongside Koordinator's orchestration capabilities, ensuring efficient interference management while maintaining system stability. The detailed design of these modules is discussed in Section 4.

3.4 | Design of Key C-Koordinator Modules

In this subsection, we will discuss the design of the key components in C-Koordinator.

3.4.1 | Interference Predictor

Our study in Section 2 demonstrates that common metrics (e.g., RT and resource utilization) cannot accurately reflect the relationship between application performance and inference given the diversity of Alibaba's applications. Therefore, We propose an interference-aware approach based on CPI predicted by fine-grained metrics from node-level, pod-level, and application-level for large-scale cluster. Traditional metrics often fall short in providing accurate and scalable insights across a vast array of nodes and diverse applications. In our approach, we employ CPI as the primary metric for detecting and predicting application interference. CPI serves as an advantageous metric due to its ability to reflect the performance characteristics of applications across a substantial number of nodes, unlike traditional metrics which may not scale well or accurately represent application performance in heterogeneous environments. This module needs to address which software and hardware metrics should be selected for prediction interference for co-located applications.

3.4.2 | Interference Detector

In the daily operations of Alibaba's scheduling system, the Interference Detector serves as the fundamental monitoring component. This routine monitoring primarily maintains the QoS for all applications running within the system and monitors the shared resource utilization at each node. Specifically, it tracks the total CPU utilization and total memory utilization of each node. Additionally, it measures the shared CPU utilization across various applications on node. If the utilization of these shared resources surpasses a dynamically adjusted threshold, the corresponding application is flagged and added to the list for further interference detection and optimization. This module needs to address which model should be used for predicting interference to balance accuracy and efficiency in large-scale cluster.

3.4.3 | Interference Mitigator

After detecting and confirming the presence of interference, the system's interference mitigation policy is activated. This

component is responsible for issuing alerts to the affected application, collecting detailed data on the application's resource usage, and deciding which mitigation strategies to employ. The system dynamically adjusts resource allocations based on the specific characteristics of the affected application and the node on which it resides. This module needs to address which application should be suppressed and which pods should be evicted from the original nodes to reduce resource usage.

4 | Implementation and Practice of the Key Components of C-Koordinator

In this section, we will discuss the implementation details and Alibaba's practice on developing C-Koordinator.

4.1 | Interference Predictor: Fine-Grained Prediction for Interference

CPI is a reliable indicator of application performance, as higher CPI values typically signal inefficiencies caused by resource contention, cache conflicts, or memory bottlenecks. Its hardware-level nature makes it more consistent across diverse applications compared to metrics like RT, which can be influenced by workload patterns, service logic, and external dependencies. However, real-time CPI measurements in production environments are subject to several challenges. CPI naturally fluctuates with changes in instruction set architectures, workload characteristics, system background noise, and transient events like cache misses or branch mispredictions. These fluctuations can introduce considerable noise into real-time data, making it difficult to distinguish between short-lived, benign spikes and sustained interference that requires corrective action. Furthermore, relying on real-time CPI data for interference management introduces operational risks. Instantaneous CPI readings may reflect momentary anomalies rather than meaningful performance trends, potentially triggering unnecessary or suboptimal resource adjustments.

To address this, we propose a predictive CPI-based interference detection approach. Instead of reacting to unstable real-time CPI readings, we use models trained on historical and system-level metrics to forecast CPI trends under varying co-location and workload conditions. This predictive strategy enables proactive interference management, smoothing out short-term fluctuations while capturing meaningful performance trends, thus offering a scalable and accurate solution for interference detection.

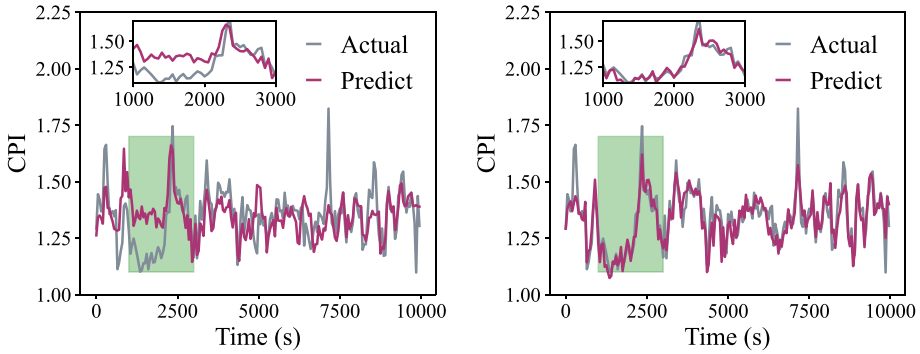
While CPI is physically collected at the node level, C-Koordinator attributes CPI variations to individual microservices through a correlation-based inference process rather than direct per-request instrumentation. Each node hosts a limited number of co-located pods, whose execution windows, CPU shares, and scheduling statistics are already tracked by the container runtime and Koordinator. By jointly analyzing CPI fluctuations with pod-level resource usage and scheduling signals over time, C-Koordinator infers which microservices are contributing to or affected by interference. This design avoids tracing CPI down to individual microservice calls, which would be prohibitively expensive and intrusive at scale. Instead, CPI serves as a node-level interference signal, while fine-grained attribution is achieved through temporal and resource-usage correlation across pods. This level of attribution is sufficient for interference detection and mitigation decisions, which are performed at the pod and application level.

4.1.1 | Practice 1: Additional Metrics for Interference Prediction

In our pursuit to develop an accurate and efficient predictive model for CPI as an interference metric in large-scale and co-located microservice-based clusters, we carefully select the metrics. The challenge lies in balancing the need for comprehensive data to ensure prediction accuracy with the practical constraints of time and resource consumption associated with data collection. Our goal was to filter out a set of key metrics that would provide the most predictive performance while minimizing overhead.

Initially, we focused on the cluster's running status, specifically examining node CPU utilization. So, we introduced commonly used utilization metrics within the container environment, such as node CPU utilization, pod CPU utilization, and memory utilization. These metrics offer a more granular view of resource consumption across different levels of the system. Node CPU utilization provides a broad overview of the total computational load across the entire node, which helps in identifying potential bottlenecks at the node level. However, we recognized that pod CPU utilization is more critical in pinpointing resource usage for individual applications, as each pod represents an isolated environment for specific workloads. By monitoring pod-level CPU usage, we can better assess resource contention and its impact on CPI. Additionally, memory utilization plays an important role, as memory contention often leads to performance degradation, especially in memory-intensive applications. Monitoring this metric helps in identifying whether resource contention arises from insufficient memory allocation or other factors affecting overall system performance. These metrics were chosen for their ability to provide insights at both the node and pod levels, allowing us to capture the necessary details for accurate interference prediction.

While these are fundamental metrics, as Figure 6a shows that we observed that their effectiveness in predicting CPI is limited in environments where resources are reserved. In such scenarios, actual resource usage tends to be significantly lower than the allocated resources. This gap occurs because reserved resources may not reflect real-time consumption, making it challenging to rely solely on these metrics for accurate interference predictions. To overcome this limitation, we expanded our metric set to include more granular indicators that better capture resource contention and performance degradation. As shown in Figure 7 that the correlation between four different Alibaba's application performance and various low-level metrics varied significantly. This variation highlighted the need for a more tailored approach to interference prediction, as relying solely on general utilization metrics was insufficient. Each application may respond differently to system resource constraints, with some being more sensitive to CPU utilization, while others may be more affected by memory usage or cache performance.



(a) Training with util. metrics. (b) Training with key metrics.

FIGURE 6 | Training results evaluated across different metrics. (a) Results based solely on resource utilization metrics; (b) results incorporated the key metrics investigated in this study.

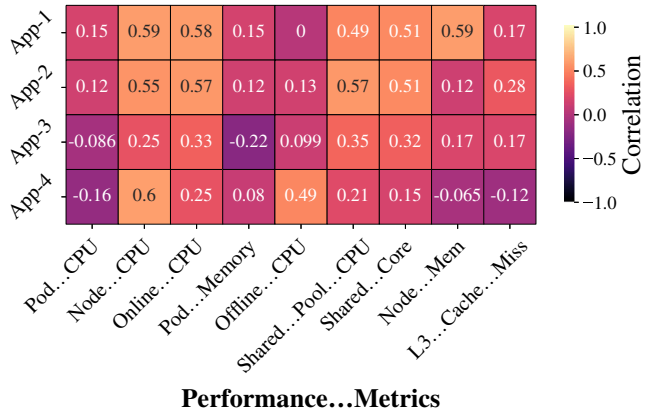


FIGURE 7 | Correlations between performance metrics and CPI across diverse applications. X-axis represents the metrics, and Y-axis displays each application's CPI.

At the same time, we carefully considered the overhead associated with data collection, such as system resource usage and the time required to gather and process these metrics. After years' observations, we selected a refined set of metrics that not only provided higher accuracy in predicting interference but also minimized the computational costs associated with real-time monitoring. These metrics were chosen based on their ease of collection across all applications and their representativeness of the underlying factors influencing CPI.

The selected nine metrics include *node CPU* and *memory utilization, pod CPU* and *memory usage, core usage (offline and online), shared pool utilization*, and *L3 cache misses*. We selected core usage as a metric because of its ability to differentiate between resource demands of latency-sensitive online applications and flexible offline workloads. In addition, shared pool utilization is another critical metric, which represents the resources shared by co-located BE, LS, and LSR applications. This resource pool enables dynamic resource sharing based on current application demands, but its complex architecture also makes it susceptible to inefficiencies and contention. If multiple applications with varying resource requirements are simultaneously accessing the shared pool, contention can arise, leading to performance degradation. Finally, L3 cache misses were chosen because they signal memory contention among applications sharing CPU cores. Cache misses force the CPU to fetch data from slower memory, causing significant delays. By monitoring these diverse metrics, we gain a comprehensive view of resource contention.

CPI prediction in C-Koordinator does not rely solely on raw monitoring metrics. Instead, we apply lightweight feature engineering to improve robustness while keeping preprocessing overhead low. Specifically, metrics are aggregated over short sliding windows to extract statistical features such as mean, variance, and short-term trends. All metrics are temporally aligned across node-, pod-, and application-level signals to ensure consistency. We further normalize features to account for scale differences across nodes and workloads. These engineered features help the model capture workload dynamics and interference patterns more effectively, without introducing complex transformations that would hinder online deployment.

As shown in Figure 6b, after a thorough analysis and selection process, the chosen metrics demonstrated a strong ability to predict the CPI, which is highly correlated with application performance. The experimental results confirm that by utilizing these metrics, we can accurately anticipate CPI fluctuations.

4.1.2 | Practice 2: Selection of Interference Prediction Models

To identify the most suitable model for Alibaba's cluster environment, we conducted a comparative experiment using real-world data collected from our system. In this experiment, we controlled for the same input metrics (as established in the previous section), ensuring that each model was evaluated on the same dataset. This controlled environment allowed us to focus solely on the performance differences between the models, without any variability from the input data. We tested a range of models, including traditional ML approaches and more advanced methods like Gradient Boosted Decision Trees (GBDT), XGBoost, Random Forest (RF), Long Short-Term Memory (LSTM), and Multi-Layer Perceptron (MLP). The choice of CPI prediction models in C-Koordinator is driven by practical considerations in large-scale production environments rather than purely theoretical optimality. The monitored system metrics are heterogeneous,

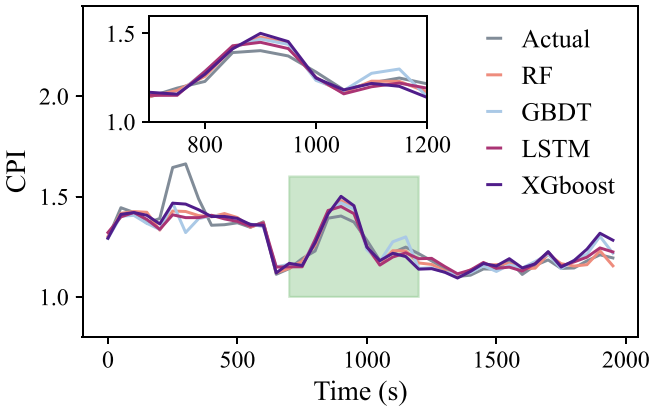


FIGURE 8 | Various models training for same application.

noisy, and exhibit non-linear interactions, which motivates the use of tree-based models such as XGBoost, GBDT, and RF. These models are well-suited for capturing complex feature interactions while maintaining stable performance under noisy monitoring data. We additionally evaluate LSTM as a representative sequence model to explore whether temporal dependencies in CPI evolution can further improve prediction accuracy. However, due to its higher computational overhead and limited deployment efficiency at scale, LSTM is included primarily as a comparative baseline rather than a default choice.

Our primary focus in model selection was prediction accuracy, and we obtained prediction results as shown in the Figure 8. The results demonstrated that most models, including XGBoost, MLP, and GBDT, produced predictions that closely aligned with the actual values, making it difficult to distinguish their performance solely based on accuracy.

Given this, we shifted our evaluation towards other critical factors, such as computational efficiency and the ability of each model to meet the demands of Alibaba's large-scale cluster, which is essential for a model to not only deliver accurate predictions but also handle the real-time processing needs of thousands of microservices while maintaining low latency and minimal computational overhead.

As shown in our further analysis in Table 3, while models like GBDT, RF, MLP, and LSTM show potential, they lack the efficiency and scalability required for Alibaba's large-scale environment. GBDT tends to be slower and more memory-intensive, RF struggles with computational efficiency, MLP requires extensive tuning and is less interpretable, and LSTM, designed for sequential data, is too resource-intensive. In contrast, XGBoost combines high accuracy with parallel processing and efficient memory usage, making it ideal for handling Alibaba's vast data volumes and complex workloads in real-time with minimal overhead. Its ability to handle missing data, optimize memory, and execute gradient boosting efficiently allows it to process large data volumes quickly while maintaining accuracy, which is essential for real-time predictions across thousands of microservices at minimal computational cost. Moreover, its boosting technique iteratively refines predictions by focusing on past errors, achieving high precision without overfitting, even when applied to vast datasets. Hyperparameter tuning for CPI prediction models is

TABLE 3 | Model performance metrics.

Metrics	XGboost	RF	GBDT	LSTM
MSE	0.006	0.006	0.007	0.006
MAE	0.060	0.058	0.061	0.058
R2	0.784	0.784	0.768	0.793
ACC	0.948	0.947	0.938	**0.953**
Training time (s)	**35.745**	55.814	89.848	89.594

Note: Bold values highlighting best accuracy and training time.

conducted offline using historical production traces. The tuning process focuses on achieving stable and robust performance across diverse workloads, rather than optimizing for peak accuracy on specific applications. To avoid overfitting and operational complexity, the search space is intentionally constrained. In production deployment, C-Koordinator adopts fixed and conservative hyperparameter configurations. This design choice ensures predictable inference latency, minimizes runtime overhead, and simplifies system maintenance, which is critical for large-scale clusters hosting thousands of co-located microservices.

4.1.3 | Practice 3: Working Process of Interference Prediction Model

Once the system flags an application from the $List_{Apps}$, which is the output of the Interference Detector module, list as a potential interference source, it conducts a more in-depth analysis to monitor its performance fluctuations and detect any significant changes. During this detailed monitoring phase, the system collects metrics at multiple levels, including pod-level CPU utilization, L3 cache miss rate (N_{miss}), and pod memory utilization (M_{pod}). These metrics are used to calculate the CPI and assess the overall impact on application performance.

We utilize these metrics as the inputs of a fine-tuned XGBoost-based model. The model processes both historical and real-time data to predict potential performance degradation. Below, we outline the main components and formulas involved in this analysis.

Our model's input metrics are categorized into three groups: (1) pod-level metric inputs I_p include pod CPU utilization C_{pod} and pod memory utilization M_{pod}; (2) node-level metric inputs I_n include node total CPU utilization $C^{\text{node}}_{\text{total}}$, node offline CPU utilization $C^{\text{node}}_{\text{off}}$, node shared pool CPU utilization $C^{\text{node}}_{\text{sh}}$, and node online CPU utilization $C^{\text{node}}_{\text{on}}$; and (3) system-level metric inputs I_s include the L3 cache miss rate N_{miss} and system-wide metrics such as total CPU utilization C_{total} and total memory utilization M_{total}.

The CPI prediction model utilizes these inputs to estimate potential performance degradation, combining data to calculate CPI_{pred} which is then used to estimate the actual CPI CPI_a from the pod, node, and system levels through a weighted aggregation of decision trees. The CPI prediction follows a gradient boosting framework, where the prediction for sample i at iteration t is updated as

$$\hat{y}_i^{(t)} = \hat{y}_i^{(t-1)} + \eta f_t(x_i), \tag{1}$$

with η denoting the learning rate and f_t representing the regression tree fitted at iteration t. The model optimizes an objective function that combines the training loss and a regularization term, expressed as

$$\mathrm{Obj}^{(t)} = \sum_{i=1}^{n} l\left(y_i, \hat{y}_i^{(t)}\right) + \sum_{k=1}^{t} \Omega\left(f_k\right), \tag{2}$$

where the loss function $l(\cdot)$ is approximated by its second-order Taylor expansion around the previous prediction $\hat{y}_i^{(t-1)}$:

$$l\left(y_i, \hat{y}_i^{(t)}\right) \approx l\left(y_i, \hat{y}_i^{(t-1)}\right) + g_i f_t\left(x_i\right) + \frac{1}{2} h_i f_t^2\left(x_i\right), \tag{3}$$

in which the first and second derivatives are

$$g_i = \left.\frac{\partial l}{\partial \hat{y}_i}\right|_{\hat{y}_i^{(t-1)}}, \quad h_i = \left.\frac{\partial^2 l}{\partial \hat{y}_i^2}\right|_{\hat{y}_i^{(t-1)}}. \tag{4}$$

The regression tree $f_t(x)$ is defined as

$$f_t(x) = w_{q(x)}, \tag{5}$$

where $q(x)$ maps input x to a leaf index and w_j denotes the corresponding leaf weight. The regularization term controlling model complexity is formulated as

$$\Omega(f) = \tau T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2, \tag{6}$$

with T representing the number of leaves in the tree, τ penalizes tree complexity, and λ penalizes large weights. Consequently, the objective simplifies to

$$\tilde{\mathrm{Obj}}^{(t)} = \sum_{j=1}^{T} \left[G_j w_j + \frac{1}{2}\left(H_j + \lambda\right) w_j^2 \right] + \tau T, \tag{7}$$

where

$$G_j = \sum_{i \in I_j} g_i, \quad H_j = \sum_{i \in I_j} h_i, \tag{8}$$

with I_j being the set of samples assigned to leaf j. The optimal leaf weights are obtained by minimizing the objective, given by

$$w_j^* = -\frac{G_j}{H_j + \lambda}, \tag{9}$$

and the corresponding minimized objective value is

$$\tilde{\mathrm{Obj}}^{*} = -\frac{1}{2} \sum_{j=1}^{T} \frac{G_j^2}{H_j + \lambda} + \tau T. \tag{10}$$

To guide tree growth, the splitting gain is computed as

$$\mathrm{Gain} = \frac{1}{2}\left(\frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{\left(G_L + G_R\right)^2}{H_L + H_R + \lambda} \right) - \tau, \tag{11}$$

where L and R refer to the left and right child nodes, respectively. Finally, the overall CPI prediction aggregates the weighted outputs of all trees as:

$$CPI_{\mathrm{pred}} = \sum_{k=1}^{K} \eta f_k(x) = \sum_{k=1}^{K} w_k \cdot T_k\left(I_p, I_n, I_s\right). \tag{12}$$

This formulation enables robust integration of pod-, node-, and system-level metrics, facilitating accurate CPI prediction under dynamic cloud computing conditions. Each tree $T_k(\cdot)$ contributes based on its relevance and accuracy, weighted by w_k, which is dynamically adjusted in accordance with practical applications at Alibaba.

To enhance the reliability of our predictions, we employ rolling statistics. The rolling mean *RM*, calculated over a predetermined number of samples, provides an average value that helps smooth short-term data volatility. The rolling standard deviation R_{Std}, on the other hand, measures the variability around this mean, offering insights into the consistency of the data. Their calculation method is as follows:

$$RM(x, n) = \frac{1}{n} \sum_{i=t-n}^{t} x_i, \tag{13}$$

$$R_{\mathrm{Std}}(x, n) = \sqrt{\frac{1}{n} \sum_{i=t-n}^{t} \left(x_i - RM(x, n)\right)^2}, \tag{14}$$

where x represents the time series data, t is the time slot, and n is the rolling window size. For our implementation, we set a rolling window of 60 samples with a sampling interval of 5 s, capturing 5 min of data. The rolling mean is essentially an average of the data points within the specified window, helping to level out spikes and drops in the CPI collection. The standard deviation of these means further aids in understanding the extent of fluctuation around the average, indicating the stability of node performance.

The difference between the predicted and actual CPI is pivotal in assessing system performance. This discrepancy, ΔCPI, is used to set a threshold for interference detection. If the variance exceeds a pre-defined limit, it suggests potential issues. The system evaluates whether the variance between the predicted CPI (CPI_{pred}) and the actual collected CPI data (CPI_{Act}) surpasses the threshold TH_{CPI}, which is dynamically calculated based on current system conditions. This threshold helps determine whether the fluctuations in performance metrics are within acceptable limits or indicative of underlying issues requiring intervention:

$$\Delta CPI = \left| \mathrm{Mean}\left(CPI_{\mathrm{pred}} - RM\left(CPI_{Act}\right)\right) \right|. \tag{15}$$

To adapt to changing system conditions, the threshold TH_{CPI} should be more dynamic, considering not only recent fluctuations but also system stress factors like current load and resource contention. We introduce a load factor L for the prediction volatility:

$$TH_{CPI} = k_1 \cdot R_{\mathrm{Std}}\left(CPI_a\right) + k_2 \cdot \frac{L_{\mathrm{current}}}{L_{\max}}, \tag{16}$$

where k_1 and k_2 adjusts the sensitivity of our interference detection, tailored to the operational demands and L_{current} represents the current load which is calculated as a weighted sum of normalized CPU, memory, and cache pressure:

$$L_{\mathrm{current}} = 0.5 \times \frac{C_{\mathrm{util}}}{C_{\mathrm{req}}} + 0.3 \times \frac{M_{\mathrm{util}}}{M_{\mathrm{req}}} + 0.2 \times \frac{N_{\mathrm{miss}}}{N_{\max}}, \tag{17}$$

where C_{util} and M_{util} are the actual CPU and memory usage of the pod, C_{req} and M_{req} are the requested CPU and memory resources, and N_{miss} is the observed L3 cache miss rate, normalized by the maximum observed value $N_{\max}$ in the cluster. The weights (0.5, 0.3, 0.2) are determined empirically to reflect the relative impact of each resource on CPI and overall pod performance, and can be predefined.

Correspondingly, $L_{\max}$ denotes the maximum possible composite load for the pod when CPU, memory, and cache pressure all reach their respective normalized upper bounds simultaneously. This normalization ensures that the load factor remains within the range [0, 1], making the detection threshold TH_{CPI} adaptive and comparable across different pods and time windows.

Finally, the system checks whether interference is detected based on the following condition, which is confirmed if the calculated variance ΔCPI exceeds the established threshold TH_{CPI}:

$$\text{Interference Detected} = \begin{cases} 1, & \text{if } \Delta CPI > TH_{\text{CPI}} \\ 0, & \text{otherwise} \end{cases} \tag{18}$$

when interference is detected, the system logs the affected applications and initiates corresponding actions. The system then calculates the *CSI* (CPI Severity Index),

$$CSI = \frac{\Delta CPI}{TH_{\text{CPI}}}, \tag{19}$$

This severity index is subsequently passed to the Interference Mitigator module. The Interference Mitigator evaluates *CSI* and determines the appropriate mitigation strategy.

4.2 | Interference Detector: Timely Identification of Interference

Algorithm 1 shows the combination of Interference Detector, Interference Predictor and Interference Mitigator, Algorithm 1 illustrates the complete interference-aware scheduling workflow of C-Koordinator, which consists of three tightly coupled stages: interference detection, interference prediction, and interference mitigation.

First, the algorithm performs interference detection (Line 3) by continuously monitoring node-level resource utilization, including CPU usage, memory usage, and shared CPU pool utilization. Based on the comprehensive utilization U_n^t and the adaptive threshold TH_{select}, nodes exhibiting abnormal resource pressure are identified, and the corresponding applications are added to the candidate list $List_{Apps}$ for further analysis.

Next, for each application in $List_{Apps}$, the algorithm enters the interference prediction phase (Line 12). In this phase, fine-grained indicators collected from node-, pod-, and application-level metrics are used to predict the future CPI using a trained XGBoost model (Line 15). The predicted CPI deviation ΔCPI is then computed and compared against the dynamically calculated CPI threshold TH_{CPI} (Line 20) to determine whether the application is experiencing significant performance degradation. If ΔCPI exceeds TH_{CPI}, the application is marked as interfered and its *CSI* is calculated (Line 24).

ALGORITHM 1 | C-Koordinator interference-aware scheduling algorithm.

1: **Input:** Interference indicators $\{I_p, I_n, I_s\}$, current latency L_{current}, actual CPI CPI_{Act}
2: **Output:** Application list $List_{Apps}$, Co-located Sensitivity Index *CSI*, and mitigation action

3: // **Step 1: Interference Detection**
4: **for** each node n **do**
5: Monitor C_n^t, M_n^t, and $C_n^{s,t}$
6: Compute utilization U_n^t using Equation (20)
7: Calculate threshold TH_{select} using Equation (21)
8: **if** $U_n^t > TH_{\text{select}}$ **then**
9: Flag applications on n and append to $List_{Apps}$
10: **end if**
11: **end for**

12: // **Step 2: Interference Prediction**
13: **for** each application a in $List_{Apps}$ **do**
14: Collect $\{I_p, I_n, I_s, CPI_{Act}\}$
15: Predict CPI_{pred} using XGBoost model
16: Compute $RM(CPI_{Act})$ and $R_{\text{Std}}(CPI_{Act})$ via Equation (14)
17: Calculate ΔCPI using Equation (15)
18: **end for**
19: **for** each node n containing $List_{Apps}$ **do**
20: Compute TH_{CPI} using Equation (16)
21: **end for**
22: **for** each application a in $List_{Apps}$ **do**
23: **if** $\Delta CPI > TH_{CPI}$ **then**
24: Mark a as interfered and calculate its *CSI*
25: **end if**
26: **end for**

27: // **Step 3: Interference Mitigation**
28: **for** each interfered application with index *CSI* **do**
29: **if** $CSI < \frac{5}{3}$ **then**
30: Execute mitigation strategy 1 (Section 4.3)
31: **else**
32: Execute mitigation strategy 2 (Section 4.3)
33: **end if**
34: **end for**
35: **return** $List_{Apps}$ and updated mitigation actions

Finally, based on the computed *CSI*, the algorithm triggers the interference mitigation stage (Line 27). Applications suffering from mild interference are handled using lightweight mitigation strategies, while severe interference cases invoke stronger actions such as resource suppression or pod eviction, ensuring that high-priority workloads maintain their QoS and SLO requirements. Through this closed-loop process, C-Koordinator continuously adapts resource allocation decisions to mitigate interference in large-scale, co-located clusters. The Interference Detector serves as the fundamental monitoring component. This routine monitoring primarily maintains the QoS for all applications running within the system and monitors the shared resource utilization at each node. Specifically, it tracks the total CPU utilization C_n^t and total memory utilization M_n^t of each node n at time interval t. Additionally, it probes the $C_n^{s,t}$ (which measures the shared CPU utilization across various applications on node n).

The comprehensive resource utilization U_n^t of node n at time interval t is computed using the following formula:

$$U_n^t = \alpha \cdot \left(\frac{M_n^t}{1 + M_n^t} \right) + \beta \cdot \sqrt{C_n^t} + \gamma \cdot \left(2 \cdot C_n^{s,t} - \left(C_n^{s,t} \right)^2 \right), \quad (20)$$

where α, β, and γ are weighting coefficients that determine the relative importance of each resource metric—namely, memory utilization, CPU utilization, and shared pool CPU utilization, respectively—in the calculation of node-level comprehensive resource usage. These coefficients are not fixed system-wide, but are instead adaptively assigned for each node to capture node-specific characteristics and workload patterns. For example, on nodes hosting memory-intensive workloads, a higher α value is chosen to emphasize memory pressure, whereas on nodes with high CPU contention, β is given greater weight. The value of γ can be increased on nodes where the shared CPU pool is a key bottleneck. The coefficients can be determined through historical analysis, profiling, or performance tuning, allowing the model to better reflect resource contention sensitivity unique to each node's configuration and operational context. This adaptive weighting mechanism improves the accuracy of interference detection by accounting for the heterogeneity and dynamic behavior of nodes within the cluster.

Based on our analysis of cluster data, it is evident that resource competition at the node level is primarily influenced by the total utilization of CPU, memory, and shared pool resources on each node. Monitoring at the node level, rather than at the individual pod level, significantly reduces the system's monitoring overhead while ensuring accurate detection of resource contention.

The dynamic utilization threshold TH_{select} is calculated using the following formula:

$$TH_{\text{select}} = \left(\frac{1}{n} \sum_{i=1}^{n} U_{n,i}^t \right) + k \cdot \left(\frac{1}{n} \sum_{i=1}^{n} \left(U_{n,i}^t - \frac{1}{n} \sum_{i=1}^{n} U_{n,i}^t \right)^2 \right), \quad (21)$$

where k is an adjustment factor, which is set to 3 in our practice.

If the calculated comprehensive utilization U_n^t exceeds the dynamic threshold TH_{select}, as $U_n^t > TH_{\text{select}}$, the system marks the application as a potential source of interference. The flagged application is then added to the $List_{Apps}$ list for further detailed monitoring, which will be used in Interference Mitigator. Once the comprehensive utilization stabilizes below the threshold for a sustained period, the application is removed from the list.

4.3 | Interference Mitigator: Application Performance Assurance

The system computes a dynamic threshold TH_{CPI} for CPI variations and determines whether the interference is mild or severe by comparing the difference ΔCPI to this threshold. The severity of interference is categorized based on ΔCPI, for mild interference, when ΔCPI is less than $\frac{5}{3}$ (this value is configured based on Alibaba's practice) of TH_{CPI} the scenario is treated as mild:

$$CSI < \frac{5}{3} \quad \left(\text{i.e.,} \quad TH_{CPI} < \Delta CPI < \frac{5}{3} \times TH_{CPI} \right). \quad (22)$$

In such cases, the system initiates the CPU suppress strategy, which temporarily reduces the CPU usage of the interfering pods, thereby minimizing its impact on other applications. For severe interference, indicated when ΔCPI exceeds $\frac{5}{3}$ of the dynamic threshold:

$$CSI > \frac{5}{3} \quad \left(\text{i.e.,} \quad \Delta CPI > \frac{5}{3} \times TH_{CPI} \right). \quad (23)$$

The response escalates to the pod eviction strategy, which involves forcibly evicting low-priority pods from the node to free up resources for higher-priority applications and stabilize the system's performance.

After confirming interference, the following strategies will be executed based on the severity of the interference:

4.3.1 | Strategy 1: Curtailing CPU Consumption Through CPU Suppress

This strategy is employed in situations where the system experiences mild interference. It works by limiting the CPU usage of low-priority pods, such as BE pods, to ensure that the performance of critical pods, like LS pods, remains unaffected. The CPU Suppress strategy dynamically adjusts the CPU allocation for BE pods based on the current CPU usage of LS pods. The specific CPU limitation is calculated using the following formula:

$$CPU_{\text{restriction}} = CPU_{\text{total}} - \left(CPU_{LS_{\text{used}}} + CPU_{\text{reserve}} \right), \quad (24)$$

where CPU_{total} represents the total CPU resources available on the node, $CPU_{LS_{\text{used}}}$ represents the current CPU usage of LS pods, CPU_{reserve} is a reserved portion of CPU resources allocated to the LS pods to ensure performance stability.

This strategy ensures the performance stability of LS pods by reducing the CPU resources available to BE pods, preventing excessive resource consumption that could degrade overall system performance.

4.3.2 | Strategy 2: Resource Reclamation Through Pod Eviction

This strategy is applied in cases of severe interference. It directly evicts pods that are consuming excessive resources from the node to rapidly restore node performance. The pod eviction strategy assesses the CPU usage of BE pods and evaluates their impact on the performance of LS pods. It selects pods for eviction that exceed a predefined CPU usage ratio. The eviction formula is as follows:

$$\text{Evict} = \left\{ \text{Pod} | CPU_{\text{Pod}} > \mu \times CPU_{\text{total}} \right\}, \quad (25)$$

where μ is a predefined threshold for CPU usage, used to determine which BE pods should be evicted, and this value can be configured to control the eviction ratio.

This strategy quickly frees up a significant amount of resources, ensuring that LS pods have sufficient CPU and memory resources

to maintain their performance. The combination of CPU suppress and pod eviction allow the system to mitigate the effects of resource contention efficiently, ensuring that critical workloads receive the necessary resources to function optimally.

5 | Performance Evaluations of C-Koordinator

In this section, we provide evaluations of the C-Koordinator system in terms of its ability to accurately predict and mitigate interference in Alibaba's cluster. All experimental results reported in this section are obtained by averaging over 5 independent runs and workload traces. For each configuration, we repeat the experiments across different time windows and workload instances to account for runtime variability. In addition to average values, we analyze the variance of key metrics, such as CPI prediction error and tail latency reduction, to ensure that the observed improvements are stable and not driven by outliers.

5.1 | Interference Prediction Evaluation

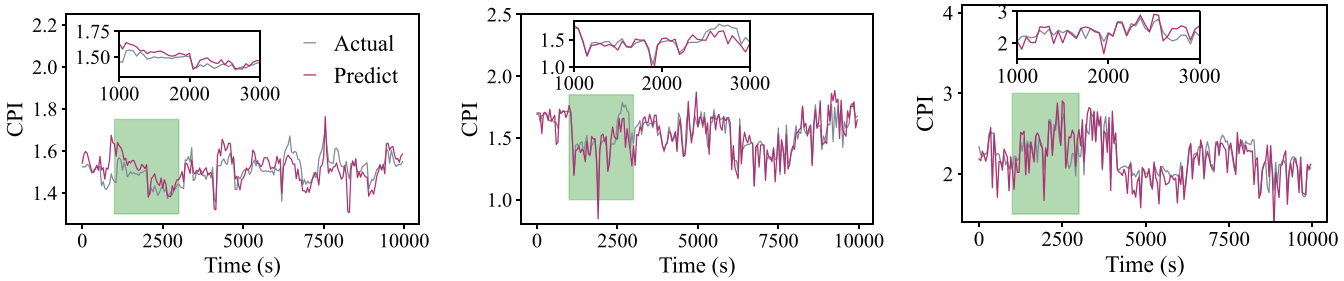
In this section, we evaluate the performance of the Interference Prediction Module using real-world data from Alibaba's internal cluster applications. We selected three representative applications (anonymized for privacy reasons, App-1 represents online web service, App-2 represents database service, and App-3 represents e-business service) that have different CPI fluctuations during runtime to test the module's accuracy in predicting potential resource contention and interference across a wide range of applications in the cluster. As demonstrated in Figure 9, our model consistently achieved an accuracy of over 90.3% for these different applications, showcasing its reliability even with Alibaba's highly diverse and dynamic workloads. Each application in the cluster exhibits different resource consumption patterns and performance characteristics, yet the prediction module has demonstrated robust and adaptive capabilities, making it highly suitable for large-scale deployment.

However, achieving this level of accuracy does come with certain trade-offs and practical considerations for large-scale cluster. The average training time per application is less than 30 s. Importantly, training is only triggered when an application is added to the potential interference list. This ensures that training is conducted only when necessary, keeping overhead minimal. Based on long-term data from Alibaba's clusters, about 0.03% of total applications are regularly added to this list. While this approach does lead to a slight increase in CPU and memory overhead during training, C-Koordinator has already reserved ample resources across all nodes to mitigate resource contention. This pre-emptive resource reservation ensures that system stability remains uncompromised, even when resource usage increases due to interference detection and mitigation. In summary, despite the slight increase in CPU and memory usage, the C-Koordinator's resource reservation strategy ensures that these overheads remain within acceptable limits, making the module both scalable and efficient for large-scale, real-time deployment.

5.2 | Interference Mitigation Evaluation

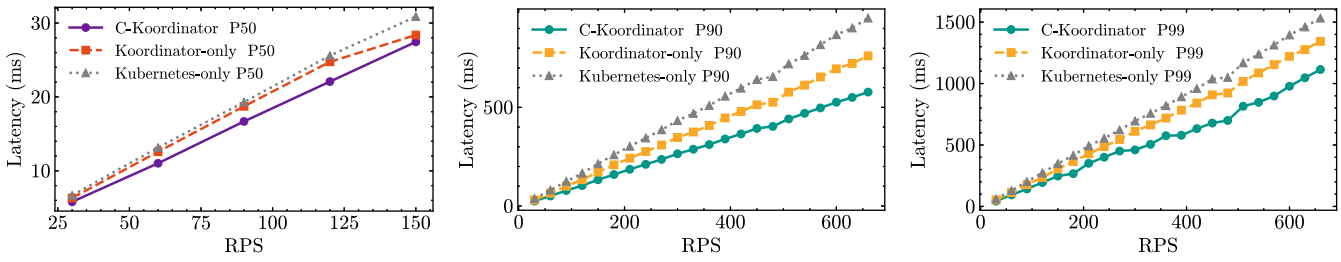
The effectiveness of the Interference Mitigator in C-Koordinator has been evaluated through experiments on a production-scale Kubernetes cluster comprising approximately 7000 nodes. This cluster hosts a diverse set of workloads, including stateless web services, stateful databases, data processing pipelines, and latency-sensitive real-time applications. The applications are deployed using a variety of resource allocation strategies as well as different CPU binding configurations, including strict CPU pinning, NUMA-aware placement, and CPU share-based scheduling.

To make experimental results reproducible, we also evaluate the interference mitigation impact on latency reduction under increasing request pressure during normal operation in our 10-node Alibaba g9i instance (4 vCPU and 16 GB memory) cluster with MySQL, Redis and Nginx applications. Compared to the Koordinator, as illustrated in Figures 10–12, our approach can reduce latency across all percentiles, for completeness, we additionally include results under the default Kubernetes scheduler as a baseline (Kubernetes-only), which represents a configuration without interference-aware detection or mitigation. For example, for Nginx application as shown in Figure 12 that RPS is increased from 0 to 200, latency is reduced by 10.7% to 36.1% across various percentiles, especially optimization on P99 tail latency. Similar optimization effects can be observed in Figures 10 and 11, and C-Koordinator can keep P50 latency in Redis increasing more smoothly when RPS is increase significantly compared with Koordinator. In contrast, the Kubernetes-only baseline exhibits the fastest latency growth under increasing load, highlighting the lack of



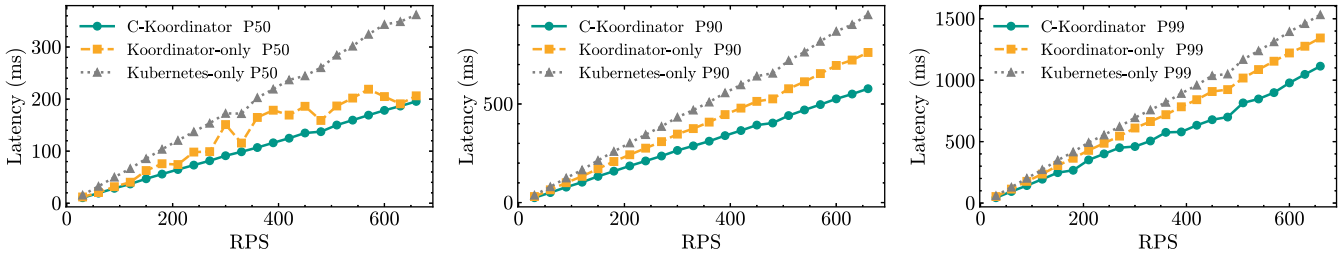
(a) CPI prediction for anonymous App-1. (b) CPI prediction for anonymous App-2. (c) CPI prediction for anonymous App-3.

FIGURE 9 | Evaluating CPI prediction across diverse applications.



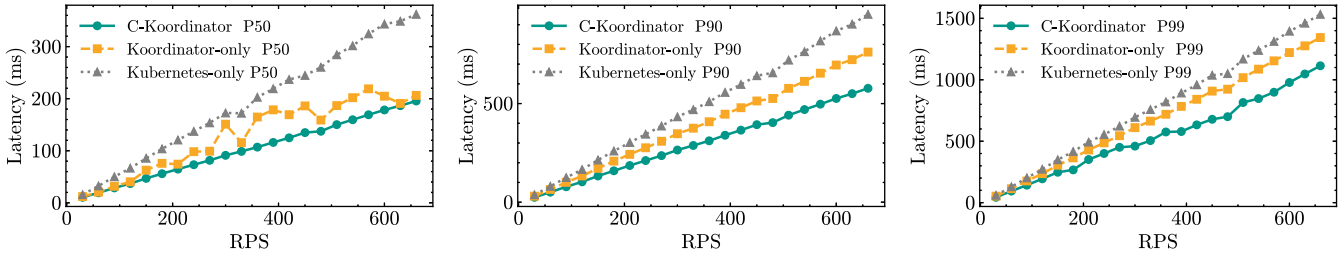
(a) MySQL P50 latency (b) MySQL P90 latency (c) MySQL P99 latency

FIGURE 10 | Latency comparison with MySQL under increasing RPS: (a) P50 latency, (b) P90 latency, and (c) P99 latency.



(a) Redis P50 latency (b) Redis P90 latency (c) Redis P99 latency

FIGURE 11 | Latency comparison with Redis under increasing RPS: (a) P50 latency, (b) P90 latency, and (c) P99 latency.



(a) Nginx P50 latency (b) Nginx P90 latency (c) Nginx P99 latency

FIGURE 12 | Latency comparison with Nginx under increasing RPS: (a) P50 latency, (b) P90 latency, and (c) P99 latency.

interference awareness in default scheduling. The reason results from that C-Koordinator can proactively identify load pressure and reallocate resources in advance, and it can prevent performance degradation and ensures smoother application execution. This behavior is enabled by the CPI-based interference prediction and CSI-guided mitigation mechanism described in Algorithm 1, which allows mitigation actions to be triggered before severe contention manifests. We observe that the performance improvements are consistent across repeated runs, with limited variance observed in CPI prediction accuracy and application-level latency.

The module efficiently managed resource allocation and scheduling, validating its effectiveness in mitigating interference and optimizing performance under different load conditions. Following the successful reduction in RT, we also assessed the system overhead introduced by the module. This overhead primarily stems from job scheduling processes inherited in Koordinator, triggered during high-pressure conditions when applications are migrated. However, since job scheduling is a native feature of Koordinator, since C-Koordinator already performs numerous scheduling operations daily, this additional overhead is minimal, typically ranging from 1% to 3% CPU usage, which is comparable to the overhead of existing production schedulers and remains acceptable in large-scale operational environments. The reported 1%–3% CPU overhead reflects the overall system-wide cost introduced by C-Koordinator. To better understand its sources, we further decompose the overhead across the three core modules shown in Algorithm 1.

The *Interference Detector* (Section 4.2) introduces negligible overhead, as it relies on lightweight periodic monitoring of node-level CPU and memory utilization, largely reusing existing Koordinator metrics. The *Interference Predictor* (Section 4.1) accounts for the majority of the overhead, mainly due to feature aggregation and CPI inference using the XGBoost model. However, this process is executed at coarse-grained intervals rather than per request, keeping its cost bounded. The *Interference Mitigator* (Section 4.3) incurs minimal steady-state overhead, as mitigation actions are event-driven and only triggered when interference is detected. Overall, the overhead is dominated by prediction-related computation, while detection and mitigation contribute marginal additional cost.

5.2.1 | Scalability and Workload Diversity Analysis

C-Koordinator is designed to operate in large-scale production clusters consisting of thousands of nodes and diverse co-located workloads. Importantly, CPI prediction in our system relies on node-local metrics and lightweight inference, rather than global coordination or cross-node synchronization. As a result, the computational cost and prediction latency remain stable as cluster size increases.

To assess robustness under workload diversity, we evaluate CPI prediction using production traces collected from clusters with varying application mixes. We observe that prediction accuracy remains stable across different workload compositions, including compute-intensive, memory-intensive, and mixed-service scenarios. This behavior stems from the fact that CPI captures micro-architectural effects of resource contention and is predicted from fine-grained local signals, making the approach inherently scalable and insensitive to global cluster size. These scalability properties are consistent with the design of Algorithm 1, where interference detection and CPI prediction are performed locally on each node, and mitigation decisions are triggered independently without requiring global synchronization.

6 | Conclusions and Discussions

In this paper, we have present C-Koordinator that effectively selects relevant metrics and predictive models based on CPI suitable for large-scale and co-located microservice cluster. The accuracy of the predictions has been impressive, with most models achieving over 90.3% precision, and the selected XGBoost-based model can balance the prediction accuracy and computation efficiency in practice. Furthermore, the results have shown significant improvements, demonstrating a noticeable reduction in RT. These findings indicate that the proposed approach is not only feasible but also highly effective in real-world scenarios, ensuring that application performance remains robust even under varying conditions.

Through the development and evaluation of C-Koordinator, several important lessons have emerged that can provide valuable insights for future research and real-world applications. These lessons highlight key aspects of our approach and its impact on system performance and management:

6.1 | CPI as the Prediction Metric

CPI is a proactive and effective metric for detecting interference across various applications, capturing performance degradation due to resource contention. Its generalizability makes it ideal for maintaining stable performance in complex multi-tenant systems.

6.2 | Fine-Grained Optimization

Selecting metrics at node, pod, and hardware levels enables precise, fine-grained resource optimization, improving overall system efficiency. This multi-layered approach helps reduce bottlenecks and significantly lowers response latency.

6.3 | Sufficiency of ML Models

ML models are sufficient for accurately predicting CPI, balancing training and inference time effectively. This allows real-time interference detection in large-scale environments without introducing significant computational overhead.

Author Contributions

All authors participated in discussions and contributed to this manuscript. Specific contributions are as follows: Shengye Song conceived the core research direction, designed C-Koordinator's framework, implemented the interference prediction model, conducted experiments, and drafted the main content. Minxian Xu provided overall guidance on research methodology and manuscript development, refining CPI-based interference measurement logic. Zuowei Zhang, Fansong Zeng, Yu Ding (Alibaba Group) offered Alibaba's cluster traces and experimental platforms, verifying C-Koordinator's compatibility with existing infrastructure. Chengxi Gao handled multi-dimensional metric collection/analysis and drafted the "Background and Motivation" section. Kejiang Ye and Chengzhong Xu provided critical insights into application scenarios and scalability considerations, and also contributed to the refinement of the manuscript.

Acknowledgments

This work is supported by National Natural Science Foundation of China (62572462), Guangdong Science and Technology Cooperation Project (2025A0505020065), Guangdong Basic and Applied Basic Research Foundation (2024A1515010251, 2023B1515130002), Key Research and Development and Technology Transfer Program of Inner Mongolia Autonomous Region (2025YFHH0110) and Shenzhen Science and Technology Program (JCYJ20240813155810014). We also thank Alibaba Group Team for providing large-scale platform and technical support.

Funding

This work was supported by National Natural Science Foundation of China (62572462); Guangdong Science and Technology Cooperation Project (2025A0505020065); Guangdong Basic and Applied Basic Research Foundation (2024A1515010251 and 2023B1515130002); Key Research and Development and Technology Transfer Program of Inner Mongolia Autonomous Region (2025YFHH0110); and Shenzhen Science and Technology Program (JCYJ20240813155810014).

Conflicts of Interest

The authors declare no conflicts of interest.

Data Availability Statement

The data that support the findings of this study are available on request from the corresponding author. The data are not publicly available due to privacy or ethical restrictions.

References

1. Y. Zhang, W. Hua, Z. Zhou, G. E. Suh, and C. Delimitrou, "Sinan: ML-Based and QoS-Aware Resource Management for Cloud Microservices," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)* (ACM, 2021), 167–181.

2. C. Delimitrou and C. Kozyrakis, "Quasar: Resource-Efficient and QoS-Aware Cluster Management," *ACM SIGPLAN Notices (SIGPLAN Notices)* 49, no. 4 (2014): 127–144.

3. C. Delimitrou and C. Kozyrakis, "Paragon: QoS-Aware Scheduling for Heterogeneous Datacenters," *ACM SIGPLAN Notices (SIGPLAN Notices)* 48, no. 4 (2013): 77–88.

4. J. Zhu, R. Yang, X. Sun, et al., "QoS-Aware co-Scheduling for Distributed Long-Running Applications on Shared Clusters," *IEEE Transactions on Parallel and Distributed Systems* 33, no. 12 (2022): 4818–4834.

5. L. Wen, M. Xu, S. S. Gill, et al., "StatuScale: Status-Aware and Elastic Scaling Strategy for Microservice Applications," *ACM Transactions on Autonomous and Adaptive Systems* 20, no. 1 (2025): 1–25.

6. L. Zhao, Y. Yang, Y. Li, X. Zhou, and K. Li, "Understanding, Predicting and Scheduling Serverless Workloads Under Partial Interference," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21)* (ACM, 2021), 1–15.

7. Y. Liu, X. Deng, J. Zhou, M. Chen, and Y. Bao, "Suppressing the Interference Within a Datacenter: Theorems, Metrics and Strategies," *IEEE Transactions on Parallel and Distributed Systems* 35, no. 5 (2024): 732–750.

8. D. Masouros, S. Xydis, and D. Soudris, "Rusty: Runtime Interference-Aware Predictive Monitoring for Modern Multi-Tenant Systems," *IEEE Transactions on Parallel and Distributed Systems* 32, no. 1 (2020): 184–198.

9. A. Margaritov, S. Gupta, R. Gonzalez-Alberquilla, and B. Grot, "Stretch: Balancing QoS and Throughput for Colocated Server Workloads on SMT Cores," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA '19)* (IEEE, 2019), 15–27.

10. M. Xu, C. Song, S. Ilager, et al., "CoScal: Multifaceted Scaling of Microservices With Reinforcement Learning," *IEEE Transactions on Network and Service Management* 19, no. 4 (2022): 3995–4009.

11. H. Kasture, D. B. Bartolini, N. Beckmann, and D. Sanchez, "Rubik: Fast Analytical Power Management for Latency-Critical Systems," in *Proceedings of the 48th International Symposium on Microarchitecture (MICRO '15)* (ACM, 2015), 598–610.

12. H. Bai, M. Xu, K. Ye, R. Buyya, and C. Xu, "DRPC: Distributed Reinforcement Learning Approach for Scalable Resource Provisioning in Container-Based Clusters," *IEEE Transactions on Services Computing* 17 (2024): 3473–3484.

13. N. Zhao, V. Tarasov, H. Albahar, et al., "Large-Scale Analysis of Docker Images and Performance Implications for Container Storage Systems," *IEEE Transactions on Parallel and Distributed Systems* 32, no. 4 (2020): 918–930.

14. M. Tirmazi, A. Barker, N. Deng, et al., "Borg: The Next Generation," in *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys '20)* (ACM, 2020), 1–14.

15. A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-Scale Cluster Management at Google With Borg," in *Proceedings of the Tenth European Conference on Computer Systems (EuroSys '15)* (ACM, 2015), 1–17.

16. P. Delgado, F. Dinu, A. M. Kermarrec, and W. Zwaenepoel, "Hawk: Hybrid Datacenter Scheduling," in *2015 USENIX Annual Technical Conference (USENIX ATC '15)* (USENIX, 2015), 499–510.

17. Z. Zhang, C. Li, Y. Tao, R. Yang, H. Tang, and J. Xu, "Fuxi: A Fault-Tolerant Resource Management and Job Scheduling System at Internet Scale," in *Proceedings of the VLDB Endowment (PVLDB '14)* (ACM, 2014), 1393–1404.

18. T. Patel and 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 '20)* (IEEE, 2020), 193–206.

19. Q. Chen, H. Yang, M. Guo, R. S. Kannan, J. Mars, and L. Tang, "Prophet: Precise QoS Prediction on Non-Preemptive Accelerators to Improve Utilization in Warehouse-Scale Computers," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)* (ACM, 2017), 17–32.

20. W. Chen, J. Rao, and X. Zhou, "Preemptive, Low Latency Datacenter Scheduling via Lightweight Virtualization," in *2017 USENIX Annual Technical Conference (USENIX ATC '17)* (USENIX, 2017), 251–263.

21. C. Iorgulescu, R. Azimi, Y. Kwon, et al., "PerfIso: Performance Isolation for Commercial Latency-Sensitive Services," in *2018 USENIX Annual Technical Conference (USENIX ATC '18)* (USENIX, 2018), 519–532.

22. Aliware, "Unified Scheduling System First Implemented on a Large Scale While Supporting Alibaba's Businesses During Double 11," 2021, accessed September 10, 2023, https://www.alibabacloud.com/blog.

23. Y. Zhang, Y. Yu, W. Wang, et al., "Workload Consolidation in Alibaba Clusters: The Good, the Bad, and the Ugly," in *Proceedings of the 13th Symposium on Cloud Computing (SoCC '22)* (ACM, 2022), 210–225.

24. X. Wang, S. Chen, J. Setter, and J. F. Martínez, "SWAP: Effective Fine-Grain Management of Shared Last-Level Caches With Minimum Hardware Support," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA '17)* (IEEE, 2017), 121–132.

25. H. Kasture and D. Sanchez, "Ubik: Efficient Cache Sharing With Strict QoS for Latency-Critical Workloads," *ACM Sigplan Notices (SIGPLAN Notices)* 49, no. 4 (2014): 729–742.

26. Y. Zhang, J. Chen, X. Jiang, et al., "Libra: Clearing the Cloud Through Dynamic Memory Bandwidth Management," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA '21)* (IEEE, 2021), 815–826.

27. J. Guo, C. Hu, and Y. Bao, "Survey on Guaranteeing the Performance of Co-Located Applications," *Journal of Computer Research and Development (JCRD)* 61, no. 1 (2024): 43–65.

28. M. Xu, L. Yang, Y. Wang, et al., "Practice of Alibaba Cloud on Elastic Resource Provisioning for Large-Scale Microservices Cluster," *Software: Practice and Experience* 54, no. 1 (2024): 39–57.

29. B. Burns, J. Beda, K. Hightower, and L. Evenson, *Kubernetes: Up and Running: Dive Into the Future of Infrastructure* (O'Reilly Media, 2022).

30. C. Carrión, "Kubernetes Scheduling: Taxonomy, Ongoing Issues and Challenges," *ACM Computing Surveys* 55, no. 7 (2022): 1–37.

31. Community K, "Koordinator: QoS-Based Scheduling for Efficient Orchestration of Microservices, AI, and Big Data Workloads on Kubernetes," accessed June 19, 2024, https://koordinator.sh/zh-Hans/.

32. W. Chen, K. Ye, and C. Z. Xu, "Co-Locating Online Workload and Offline Workload in the Cloud: An Interference Analysis," in *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)* (IEEE, 2019), 2278–2283.

33. Y. Cui, N. Dai, Z. Lai, et al., "TailCutter: Wisely Cutting Tail Latency in Cloud CDNs Under Cost Constraints," *IEEE/ACM Transactions on Networking* 27, no. 4 (2019): 1612–1628.

34. C. Delimitrou and C. Kozyrakis, "Hcloud: Resource-Efficient Provisioning in Shared Cloud Systems," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)* (ACM, 2016), 473–488.

35. D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving Resource Efficiency at Scale," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA '15)* (ACM, 2015), 450–462.

36. Z. Li, Q. Chen, S. Xue, et al., "Amoeba: QoS-Awareness and Reduced Resource Usage of Microservices With Serverless Computing," in *2020*

IEEE International Parallel and Distributed Processing Symposium (IPDPS '20) (IEEE, 2020), 399–408.

37. H. Yang, A. Breslow, J. Mars, and L. Tang, "Bubble-Flux: Precise Online QoS Management for Increased Utilization in Warehouse Scale Computers," *ACM SIGARCH Computer Architecture News* 41, no. 3 (2013): 607–618.

38. P. Lama, S. Wang, X. Zhou, and D. Cheng, "Performance Isolation of Data-Intensive Scale-Out Applications in a Multi-Tenant Cloud," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS '18)* (IEEE, 2018), 85–94.

39. A. Pi, X. Zhou, and C. Xu, "Holmes: SMT Interference Diagnosis and CPU Scheduling for Job Co-Location," in *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing (HPDC '22)* (ACM, 2022), 110–121.

40. Y. Liang, S. Zeng, and L. Wang, "Quantifying Resource Contention of Co-Located Workloads With the System-Level Entropy," *ACM Transactions on Architecture and Code Optimization* 20, no. 1 (2023): 1–25.

41. X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes, "CPI2: CPU Performance Isolation for Shared Compute Clusters," in *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys '13)* (ACM, 2013), 379–391.

42. A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan, "Shenango: Achieving High CPU Efficiency for Latency-Sensitive Datacenter Workloads," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19)* (USENIX, 2019), 361–378.

43. J. Fried, Z. Ruan, A. Ousterhout, and A. Belay, "Caladan: Mitigating Interference at Microsecond Timescales," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)* (USENIX, 2020), 281–297.

44. Q. Chen, S. Xue, S. Zhao, et al., "Alita: Comprehensive Performance Isolation Through Bias Resource Management for Public Clouds," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis (SC '20)* (ACM, 2020), 1–13.

45. L. Zhao, Y. Yang, K. Zhang, et al., "Rhythm: Component-Distinguishable Workload Deployment in Datacenters," in *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys '20)* (ACM, 2020), 1–17.

46. S. A. Javadi and A. Gandhi, "Dial: Reducing Tail Latencies for Cloud Applications via Dynamic Interference-Aware Load Balancing," in *2017 IEEE International Conference on Autonomic Computing (ICAC '17)* (IEEE, 2017), 135–144.

47. T. Shi, Y. Yang, Y. Cheng, X. Gao, Z. Fang, and Y. Yang, "Alioth: A Machine Learning Based Interference-Aware Performance Monitor for Multi-Tenancy Applications in Public Cloud," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS '23)* (IEEE, 2023), 908–917.

48. S. Chen, C. Delimitrou, and J. F. Martínez, "Parties: QoS-Aware Resource Partitioning for Multiple Interactive Services," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)* (ACM, 2019), 107–120.

49. H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer, "FIRM: An Intelligent Fine-Grained Resource Management Framework for SLO-Oriented Microservices," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)* (USENIX, 2020), 805–825.

50. S. Luo, H. Xu, C. Lu, et al., "An In-Depth Study of Microservice Call Graph and Runtime Performance," *IEEE Transactions on Parallel and Distributed Systems* 33, no. 12 (2022): 3901–3914.

51. R. Li, M. Du, Z. Wang, H. Chang, S. Mukherjee, and E. Eide, "LongTale: Toward Automatic Performance Anomaly Explanation in Microservices," in *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering (ICPE '22)* (ACM, 2022), 5–16.

52. C. Lu, H. Xu, K. Ye, et al., "Understanding and Optimizing Workloads for Unified Resource Management in Large Cloud Platforms," in *Proceedings of the Eighteenth European Conference on Computer Systems (EuroSys '23)* (ACM, 2023), 416–432.

53. M. Adeppady, P. Giaccone, H. Karl, and C. F. Chiasserini, "Reducing Microservices Interference and Deployment Time in Resource-Constrained Cloud Systems," *IEEE Transactions on Network and Service Management* 20 (2023): 3135–3147.

54. D. Masouros, C. Pinto, M. Gazzetti, S. Xydis, and D. Soudris, "Adrias: Interference-Aware Memory Orchestration for Disaggregated Cloud Infrastructures," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA '23)* (IEEE, 2023), 855–869.

55. J. Zhu, R. Yang, C. Hu, et al., "Perph: A Workload Co-Location Agent With Online Performance Prediction and Resource Inference," in *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid '21)* (IEEE, 2021), 176–185.