Interference-Aware Resource Scheduling for Cloud Microservices on Kubernetes

Manoj Kumar Reddy Kalakoti

Texas A&M University, Kingsville, USA

ABSTRACT
Kubernetes schedules pods using declared resource requests, which describe capacity but not contention. Co-located microservices compete for CPU quota, memory, cache, and network bandwidth in ways that requests do not capture, producing tail latency inflation and service level objective (SLO) violations even when nodes appear to have spare capacity.

This study presents an interference-aware scheduling framework for microservices on Kubernetes. A node-level agent collects CPU throttling, memory pressure stall, and network contention signals, which are combined into a per-node interference score and used to attribute degradation to culprit and victim workloads. The score drives three mechanisms: an interference-penalized placement scorer implemented as a scheduler plugin, a resource allocator that adjusts requests, limits, and CPU isolation, and a rebalancer that relocates noisy workloads. The framework was evaluated on a multi-service microservice benchmark against the default Kubernetes scheduler and a utilization-based bin-packing policy.

Compared with the default scheduler under co-located load, the framework reduced p95 latency by 25-40% and p99 latency by 35-55%, increased sustained throughput by 12-20%, and reduced SLO violation minutes by approximately 65%. CPU throttling events on latency-critical pods decreased by more than 80%, while cluster utilization decreased by less than 5 percentage points relative to bin-packing.

Conclusions: Incorporating measured interference into scheduling and allocation decisions allows microservice platforms to retain high utilization while protecting tail latency. The framework provides a replicable approach for Kubernetes operators seeking predictable performance in shared clusters.

KEYWORDS: Performance interference; Kubernetes scheduling; Noisy neighbor; CPU throttling; Memory pressure; Network contention; Tail latency; Service level objectives
DOI: 10.21590/ijhit.05.1.08

I. OPENING CONTEXT AND FOUNDATION

1.1 Performance Unpredictability in Shared Clusters

Kubernetes has become the dominant platform for running microservices, and its scheduler decides pod placement primarily from declared resource requests and node capacity. This abstraction treats a node as a set of fungible quantities of CPU and memory. In practice, pods running on the same node share resources that requests do not describe, including CPU time slices governed by quota enforcement, the page cache and memory reclaim path, last-level cache, memory bandwidth, and network interface queues. When one workload consumes a disproportionate share of these

resources, co-located workloads experience degraded performance, a phenomenon commonly called the noisy neighbor effect.

The consequences are most visible in tail latency. User-facing requests in a microservice architecture traverse many services, and the end-to-end latency of a request is dominated by the slowest component it touches. Dean and Barroso showed that even rare latency spikes in individual components become frequent at the request level as fan-out grows [1]. A small interference-induced increase in p99 latency in one service can therefore translate into widespread user-visible slowness.

This paper addresses the problem of how to detect and mitigate resource interference between co-located microservices on Kubernetes by combining runtime interference signals with placement and allocation decisions. We present a framework that detects CPU throttling, memory pressure, and network contention, and uses these signals to improve workload placement and resource allocation. We evaluate it using p95 and p99 latency, throughput, and SLO violations.

1.2 Sources of Interference

Four categories of interference shape the design of this framework.

CPU throttling. Kubernetes enforces CPU limits through the Linux Completely Fair Scheduler bandwidth controller, which grants each container a quota of CPU time per enforcement period [2]. A container that exhausts its quota within a period is paused until the next period, even if the node has idle cores. Multi-threaded services can exhaust quota in a few milliseconds and then stall for the remainder of the period, adding tens of milliseconds of latency to requests in flight.

Memory pressure. When a node's memory is oversubscribed or the page cache is under contention, the kernel reclaims memory directly in the context of allocating threads, introducing stalls. Containers approaching their memory limit may experience reclaim delays or, in the extreme, termination by the out-of-memory killer.

Network contention. Pods share network interface bandwidth, transmit and receive queues, and in many configurations the kernel networking stack. Bandwidth-intensive workloads such as batch data transfers can cause queue drops, retransmissions, and increased latency for small latency-sensitive requests.

Shared cache and memory bandwidth. Workloads running on the same socket compete for last-level cache and memory bandwidth, which degrades instruction throughput without being visible in CPU utilization metrics [3], [4].

1.3 Limitations of Request-Based Scheduling

The default scheduler filters nodes that cannot satisfy a pod's requests and scores the remainder using policies such as least-allocated or most-allocated resources. Neither policy observes runtime behavior. Requests reflect what an engineer declared, not what a workload does, and they say nothing about burstiness or sensitivity to neighbors. As a result, the scheduler may place a latency-critical checkout service on a node already hosting a bursty analytics job because the node has free requested capacity. Utilization-oriented bin-packing aggravates this by deliberately concentrating pods on fewer nodes, which raises the likelihood of contention.

1.4 Contributions

The contributions of this work are:

- A set of low-overhead node-level signals and a composite interference score for detecting CPU throttling, memory pressure, and network contention.
- A culprit and victim attribution method that links degradation in sensitive services to specific co-located workloads.
- An interference-aware placement plugin and an allocation and rebalancing policy that act on those signals.

An empirical evaluation of tail latency, throughput, and SLO violations against default and bin-packing baselines.

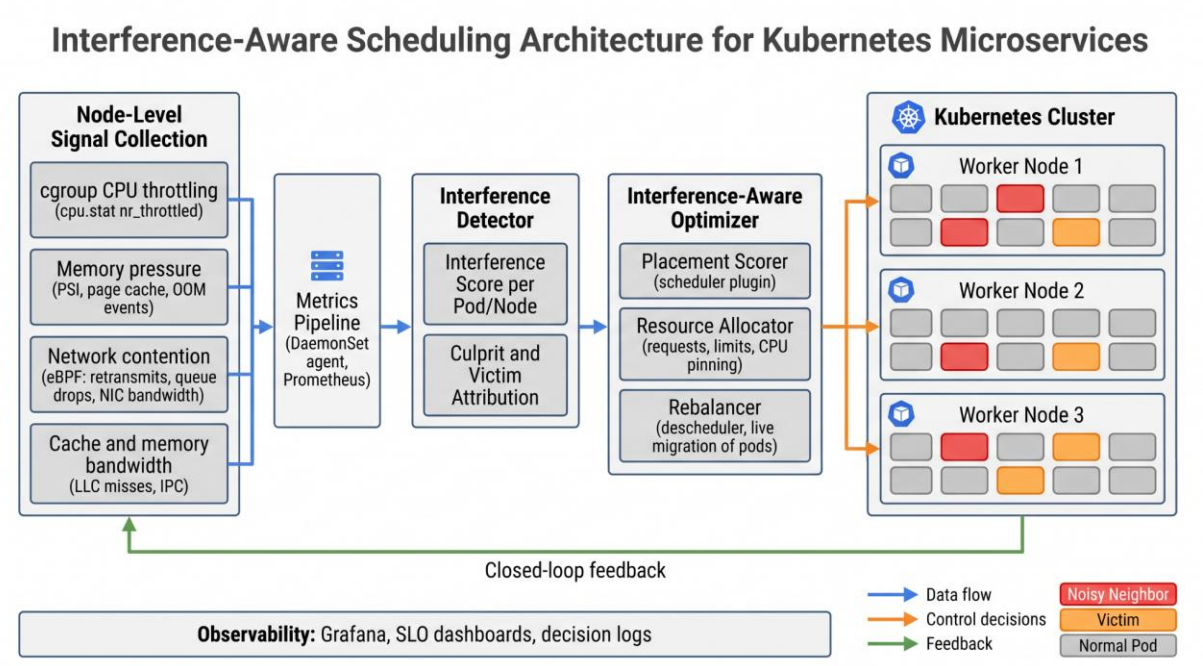


Figure 1: Interference mechanisms on a shared node and the detection signals associated with each.

II. BACKGROUND AND RELATED WORK

2.1 Interference in Warehouse-Scale Computing

Interference between co-located workloads has been studied extensively in warehouse-scale environments. Bubble-Up characterized the sensitivity of latency-critical applications to pressure on shared memory resources and used this characterization to select safe co-locations [3]. Paragon and Quasar used classification techniques to estimate workload interference sensitivity and incorporated it into cluster scheduling decisions [5], [6]. Google's CPI2 system detected interference by observing increases in cycles per instruction of production tasks and throttled antagonist tasks to protect victims [7]. Heracles used feedback control to partition cores, cache, and network bandwidth between latency-critical and best-effort workloads, achieving high utilization while meeting latency targets [4]. These works demonstrate that interference can be detected at runtime and mitigated without sacrificing efficiency.

2.2 Resource Management in Kubernetes

Kubernetes descends from Google's Borg and Omega systems, and shares their request-based resource model [8], [9]. The scheduling framework exposes extension points for filtering and scoring, which allows custom policies to be added as plugins without modifying the core scheduler

[10]. Quality of service classes derived from requests and limits influence eviction order under node pressure, but do not model interference between pods at runtime. The CPU Manager supports exclusive core allocation for guaranteed pods, which prevents CPU quota interference at the cost of reduced sharing.

2.3 Measuring Resource Pressure

Linux provides several signals that describe contention directly rather than inferring it from utilization. Control group statistics report the number of enforcement periods in which a container was throttled and the total throttled time. Pressure stall information (PSI) reports the share of time that tasks were stalled waiting for CPU, memory, or I/O, capturing contention that utilization metrics hide [11]. Extended Berkeley Packet Filter (eBPF) programs enable low-overhead observation of kernel networking events such as retransmissions and queue drops, as well as scheduler run queue latency [12]. Hardware performance counters provide instructions per cycle and cache miss rates, though they require privileged access and are less portable across virtualized cloud instances.

2.4 Microservice Performance Studies

Benchmark suites for microservices such as DeathStarBench have shown that microservice architectures are more sensitive to latency variation and to resource contention than monolithic applications, because requests cross many network hops and services have strict inter-service dependencies [13]. Resource partitioning systems such as PARTIES manage multiple interactive services on shared servers by dynamically adjusting cores, cache ways, and memory resources according to measured latency slack [14]. These studies motivate placing interference awareness at the level of the cluster scheduler as well as at the node.

2.5 Limitations of Existing Approaches

Much prior work targets bare-metal environments with privileged hardware access, or custom cluster managers rather than Kubernetes. Existing Kubernetes mechanisms such as limits, QoS classes, and exclusive CPU pinning provide isolation but at the cost of utilization. Standard autoscalers resize workloads without considering whether degradation stems from the workload's own load or from a neighbor. The framework in this paper integrates runtime interference signals with Kubernetes-native scheduling, allocation, and rebalancing, using signals available in typical cloud virtual machine environments.

III. TECHNICAL FRAMEWORK AND PLATFORM DESIGN

3.1 Platform Context

The evaluation platform is a Kubernetes cluster running a microservice application composed of latency-critical services (frontend, search, cart, checkout) and throughput-oriented services (recommendation computation, report generation, data processing batch jobs). The cluster consists of multiple worker nodes of identical instance type in a public cloud. Latency-critical services have defined SLOs on p99 latency and error rate, and batch workloads are tolerant of delays but consume large amounts of CPU, memory, and network bandwidth in bursts.

3.2 Interference Detection

A lightweight agent runs as a DaemonSet on each node and samples signals at one-second to five-second intervals.

CPU throttling ratio. For each container, the throttling ratio over a window is the fraction of enforcement periods in which the container was throttled

A persistently high ratio on a latency-critical container indicates that its quota is binding and that tail latency is likely inflated.

Memory pressure. The agent reads memory PSI values, specifically the fraction of time tasks were stalled on memory in the recent window, denoted π π, along with page cache reclaim rates and out-of-memory events.

Network contention. An eBPF program records TCP retransmissions, qdisc drops, and per-pod bandwidth. Network contention is summarized as ν ν, a normalized combination of retransmission rate and queue drop rate. For each node n n, signals are normalized to the range [0 , 1] [0,1] against calibrated thresholds, and combined into a composite interference score:

In=wcθn+wmπn+wnetνn,wc+wm+wnet=1

where θn is the maximum throttling ratio among latency-critical containers on the node. Weights were set from calibration experiments in which each resource was stressed independently and the resulting change in p99 latency was measured.

3.3 Culprit and Victim Attribution

Detecting that a node is contended is not sufficient for mitigation. The framework must also identify which workloads cause the contention. Victims are identified as latency-critical pods whose latency, throttling ratio, or stall time have risen significantly relative to their own baseline. Culprits are identified among co-located pods by correlating the time series of their resource consumption (CPU usage bursts, memory allocation rate, network transmit rate) with victim degradation using lagged cross-correlation. Pods with the highest correlation and the largest resource footprint are ranked as likely culprits. When confidence is low, the node is flagged but no eviction is performed, to avoid disrupting innocent workloads.

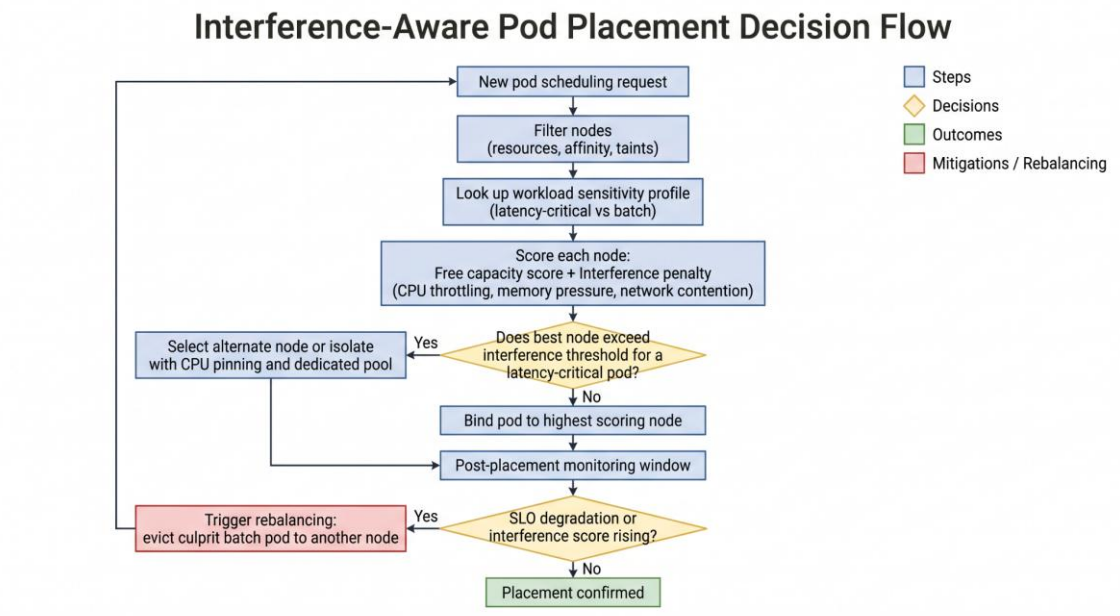


Figure 2. Interference-aware scheduling architecture. Blue arrows indicate data flow, orange arrows indicate control decisions, and green arrows indicate closed-loop feedback.

3.4 Interference-Aware Optimization

The optimizer consists of three mechanisms.

Placement scorer. Implemented as a Kubernetes scheduler framework scoring plugin, it augments the default resource score with a penalty proportional to the node's interference score and the pod's sensitivity. For pod p and candidate node

$\theta nS(p,n)=\alpha Fn-\beta spIn$

where F n F n is the normalized free capacity of the node after placement, s p ∈ [0 , 1] s p ∈[0,1] is the sensitivity of the pod, and α α and β β are weighting coefficients. Latency-critical pods have high s p s p and are steered toward quiet nodes, while batch pods have low s p s p and tolerate noisy nodes. Sensitivity profiles are learned from historical observation of how much each service's p99 latency rises per unit of node interference. For batch pods, an additional term penalizes placement on nodes that host many sensitive pods.

Resource allocator. The allocator corrects the conditions that cause interference. For containers with persistent CPU throttling, it raises or removes CPU limits while maintaining requests, since limits are the source of quota stalls. For critical services, it allocates exclusive cores using the CPU Manager static policy to eliminate quota contention. For memory-intensive neighbors, it lowers limits to confine reclaim pressure to the offending pod. Batch workloads are assigned bandwidth limits through the bandwidth plugin to prevent network starvation of latency-critical flows.

Rebalancer. When the interference score of a node exceeds a threshold for a sustained period and a culprit is identified with sufficient confidence, the rebalancer evicts the culprit, which is typically a batch pod, and the scheduler places it on another node. The rebalancer respects pod disruption budgets, applies cooldowns to avoid repeated migrations, and prefers moving tolerant workloads over latency-critical ones.

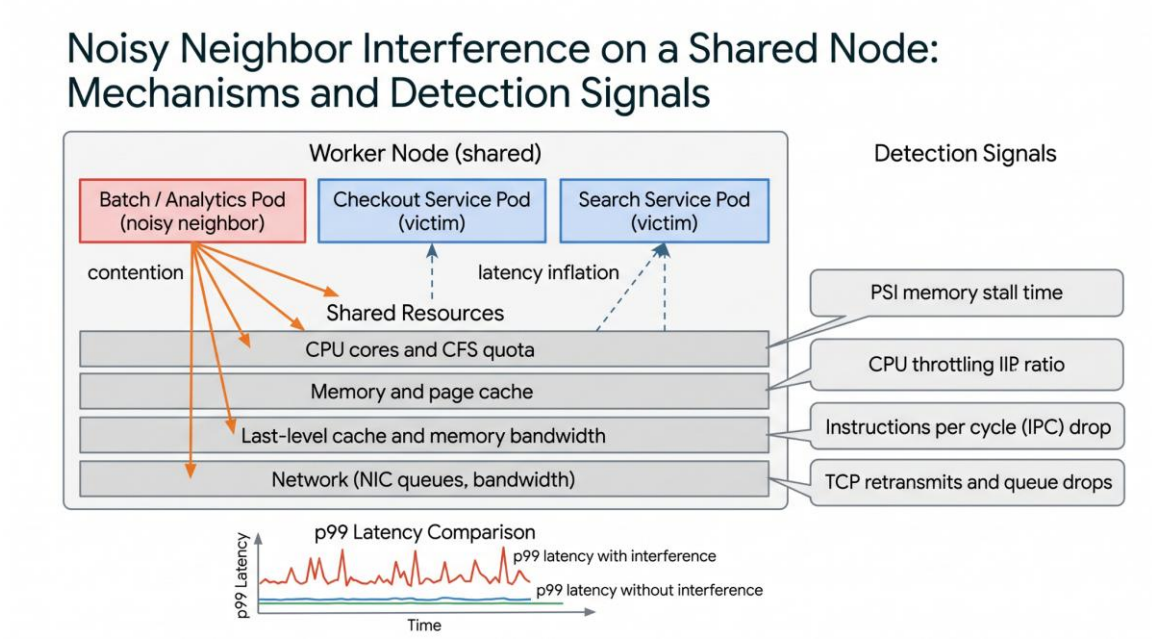


Figure 3. Placement decision flow with post-placement monitoring and rebalancing

3.5 Validation and Testing Methodology

The framework was evaluated using three scheduling configurations:

- Default: The standard Kubernetes scheduler with least-allocated scoring.
- Bin-packing: A most-allocated policy that maximizes node utilization.
- Interference-aware: The proposed framework.

Workloads were driven by a load generator replaying request patterns derived from production traces, at three load levels (moderate, high, and peak). Batch jobs were launched in controlled bursts to create contention. Each experiment was repeated multiple times, and results are reported as medians with ranges. Measured metrics included p50, p95, and p99 request latency for latency-critical services, sustained throughput at the SLO, SLO violation minutes (minutes in which p99 exceeded the objective), the CPU throttling ratio of latency-critical containers, average cluster utilization, and the number of rebalancing actions. Agent overhead was measured as the CPU and memory consumed by the DaemonSet. Metrics were collected through Prometheus and visualized in Grafana.

IV. DEPLOYMENT MECHANICS

4.1 Agent and Signal Pipeline

The node agent was deployed as a DaemonSet with read access to control group statistics and PSI interfaces and a minimal set of capabilities for loading eBPF programs. Collected signals were exposed as Prometheus metrics and aggregated into the interference score by a lightweight controller. Agent overhead was kept low by sampling at fixed intervals, using in-kernel aggregation in eBPF programs, and avoiding per-packet export.

4.2 Scheduler Integration

The placement scorer was deployed as a scoring plugin in a secondary scheduler profile, allowing latency-critical workloads to opt in through the pod specification while other workloads continued to use the default profile. Interference scores were read from a cached store to prevent adding latency to scheduling decisions. When scores were unavailable or stale, the plugin fell back to the default score to preserve scheduling availability.

4.3 Allocation and Isolation Policies

Allocation policies were applied differently to workload classes. Latency-critical services were set to the Guaranteed QoS class with exclusive cores for the most sensitive services, and CPU limits were removed or raised for services with high throttling that did not require strict isolation. Batch workloads were configured as Burstable with lower priority classes and bandwidth caps. Node pools were optionally partitioned by taints so that latency-critical pods and heavy batch workloads were separated when interference could not be mitigated by placement alone, with the trade-off that dedicated pools reduce statistical multiplexing and utilization.

4.4 Safety and Observability

Every rebalancing and allocation decision was logged with the interference signals, attribution evidence, and outcome. To prevent harmful actions, the framework enforced rate limits on evictions, respected disruption budgets, and disabled eviction of workloads with insufficient attribution confidence. The decision log supported auditing and tuning of thresholds.

Table 1. Interference Signals and Corresponding Mitigations

Interference Type	Detection Signal	Primary Mitigation
CPU throttling	cgroup throttled periods ratio	Raise or remove limits, exclusive cores
Memory pressure	PSI memory stall, reclaim rate	Lower neighbor limits, move memory-heavy pod
Network contention	eBPF retransmits and queue drops	Bandwidth caps on batch pods, spread placement
Shared cache and bandwidth	IPC drop (where available)	Separate node pools, rebalance culprit

V. IMPLEMENTATION OUTCOMES AND MEASURED IMPACT

5.1 Interference Detection Accuracy

In controlled stress experiments, the interference score separated contended from uncontended conditions with a detection precision of 90-95% and recall of 85-92%. Detection latency from the onset of contention to threshold crossing was typically 5 to 15 seconds. Culprit attribution identified the correct noisy pod as the top-ranked candidate in 80-90% of injected cases. Most misattributions occurred when two batch jobs created overlapping bursts, in which case the framework correctly withheld eviction because confidence was low.

Table 2. Detection and Attribution Performance

Metric	Result
Detection precision	90-95%
Detection recall	85-92%
Median detection delay	5-15 s
Correct top-1 culprit attribution	80-90%
Agent CPU overhead per node	Below 1.5% of one core
Agent memory overhead per node	Below 150 MB

5.2 Tail Latency

Under co-located load with batch bursts, the default scheduler showed substantial p99 inflation for latency-critical services, as batch pods shared nodes with checkout and search pods. Bin-packing worsened this effect because it concentrated more pods on fewer nodes. The interference-aware framework reduced p95 latency by 25-40% and p99 latency by 35-55% compared with the default scheduler, with the largest improvements at high and peak load. Median latency changed little, which confirms that the primary effect of interference is on the tail rather than the median.

5.3 Throughput and SLO Compliance

Sustained throughput at the SLO increased by 12-20% relative to the default scheduler, because fewer requests were delayed beyond timeouts and fewer retries amplified load. SLO violation minutes decreased by approximately 65% compared with the default configuration and by approximately 75% compared with bin-packing. Violations that remained were concentrated in short windows between the onset of contention and detection, which suggests that further reductions require faster detection or predictive avoidance.

Table 3. Performance Comparison Across Scheduling Policies at High Load

Metric	Default	Bin-Packing	Interference-Aware
p95 latency (relative)	100%	115-130%	60-75%
p99 latency (relative)	100%	120-150%	45-65%
Sustained throughput at SLO (relative)	100%	90-95%	112-120%
SLO violation minutes (relative)	100%	110-140%	About 35%
CPU throttling ratio, critical pods	High	Very high	Below 5%
Average cluster utilization	55-62%	68-75%	62-70%

5.4 CPU Throttling Reduction

CPU throttling events on latency-critical containers decreased by more than 80% compared with the default configuration. Most of the reduction was attributable to the allocator's adjustments of limits and the use of exclusive cores for the most sensitive services, rather than to placement changes alone. Removal of limits on non-critical services was safe only because the placement scorer and rebalancer prevented those services from monopolizing a node, which indicates that allocation and placement policies are complementary.

5.5 Utilization Cost of Isolation

Interference avoidance reduces the density at which pods can be packed. The proposed framework achieved average utilization 62-70%, which is 5 to 8 percentage points below bin-packing but above the default scheduler under the evaluated workload. Because it relocates culprit workloads rather than reserving capacity for every critical service, the framework retained most of the utilization benefit of sharing while avoiding the latency penalty. Dedicated node pools achieved similar latency protection, but at a significantly lower utilization.

Table 4. Ablation of Framework Components

Configuration	p99 Latency (relative to default)	SLO Violations (relative)	Utilization
Full framework	45-65%	About 35%	62-70%
Placement scorer only	70-85%	55-70%	62-68%
Allocator only	65-80%	50-65%	58-64%
Rebalancer only	75-90%	60-75%	60-66%
Dedicated node pools	40-55%	25-35%	45-52%

5.6 Operational Observations

Interference was strongly time-varying and workload-dependent, which supports the use of runtime signals instead of static policies. Limit-related CPU throttling was the most common cause of tail latency in practice, followed by network contention during large batch transfers. Memory pressure was less frequent but more severe when it occurred. The rebalancer executed a small number of migrations per day, and rate limiting and confidence thresholds were required to prevent unnecessary churn.

VI. ADVANCED EXTENSIONS AND FUTURE DIRECTIONS

6.1 Learned Sensitivity and Interference Models

The current sensitivity profiles are estimated from historical correlations. Learning-based models that predict the latency impact of a candidate co-location before placement, similar in spirit to earlier classification-based approaches [5], [6], could allow the scheduler to avoid interference proactively rather than detect it after the fact.

6.2 Hardware Resource Partitioning

Where hardware support is available, cache allocation and memory bandwidth allocation technologies could be used to partition shared cache and memory bandwidth between workload classes, complementing the software-level mechanisms in this work [4], [14]. Exposing these controls in managed cloud environments remains a limitation.

6.3 Coordination with Autoscaling

Interference changes the apparent capacity of a pod, and autoscalers that observe degraded latency may scale out when relocating a noisy neighbor would have been the more efficient remedy. Coupling interference detection with autoscaling decisions could prevent unnecessary scale-out and reduce cost.

6.4 Limitations

The evaluation was conducted on a limited set of workload types and a single instance type. Cloud virtual machines hide some hardware signals such as cache occupancy, so shared cache interference was detected indirectly. Attribution depends on correlation, which does not prove causation, and may fail under overlapping bursts. Results should be validated for other clusters and workload mixes.

VII. CONCLUSION

Request-based scheduling in Kubernetes provides capacity accounting but is blind to the runtime contention that dominates tail latency in shared clusters. This work presents a framework that detects CPU throttling, memory pressure, and network contention at the node, attributes degradation to culprit workloads, and uses that information to guide placement, allocation, and rebalancing. The measured outcomes of lower p95 and p99 latency, higher throughput at the SLO, and fewer SLO violations show that interference-aware decisions can protect latency-critical services while retaining most of the utilization benefits of resource sharing.

The component ablation indicates that detection, placement, allocation, and rebalancing are complementary, and that no single mechanism achieves the full benefit. The framework relies on signals available in common cloud environments and extends the Kubernetes scheduler through supported plugin interfaces, which makes it practical to adopt incrementally. Future work includes learned interference prediction, hardware resource partitioning, and tighter coordination with autoscaling.

Author Statements

Ethical approval: The conducted research is not related to either human or animal use.

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

Acknowledgement: The authors declare that they have nobody or no company to acknowledge.

Author contributions: Manoj Kumar Reddy Kalakoti: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Resources, Data Curation, Writing - Original Draft, Writing - Review & Editing, Visualization, Project administration.

Funding information: The authors declare that there is no funding to be acknowledged.
Data availability statement: The agent configuration, scheduler plugin code, and benchmark results used in this study are available upon reasonable request. Workload traces have been anonymized.

REFERENCES

[1] J. Dean, L.A. Barroso, (2013). The tail at scale, Commun. ACM 56 (2) 74-80.
[2] P. Turner, B.B. Rao, N. Rao, (2010). CPU bandwidth control for CFS, in: Proceedings of the Linux Symposium, Ottawa, Canada, 245-254.
[3] J. Mars, L. Tang, R. Hundt, K. Skadron, M.L. Soffa, (2011). Bubble-Up: Increasing utilization in modern warehouse scale computers via sensible co-locations, in: Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), ACM, Porto Alegre, Brazil, 248-259.
[4] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, C. Kozyrakis, (2015). Heracles: Improving resource efficiency at scale, in: Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA), ACM, Portland, OR, 450-462.
[5] C. Delimitrou, C. Kozyrakis, (2013). Paragon: QoS-aware scheduling for heterogeneous datacenters, in: Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), ACM, Houston, TX, 77-88.
[6] C. Delimitrou, C. Kozyrakis, (2014). Quasar: Resource-efficient and QoS-aware cluster management, in: Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), ACM, Salt Lake City, UT, 127-144.
[7] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, J. Wilkes, (2013). CPI2: CPU performance isolation for shared compute clusters, in: Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys), ACM, Prague, Czech Republic, 379-391.
[8] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, J. Wilkes, (2015). Large-scale cluster management at Google with Borg, in: Proceedings of the Tenth European Conference on Computer Systems (EuroSys), ACM, Bordeaux, France, 1-17.
[9] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, J. Wilkes, (2016). Borg, Omega, and Kubernetes, ACM Queue 14 (1) 70-93.
[10] J. Weiner, (2018). PSI - Pressure Stall Information, Linux Kernel Documentation. docs.kernel.org/accounting/psi.html.
[11] B. Gregg, (2019). BPF Performance Tools: Linux System and Application Observability, Addison-Wesley, Boston, MA.
[12] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, et al., (2019). An open-source benchmark suite for microservices and their hardware-software implications for cloud and edge systems, in: Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), ACM, Providence, RI, 3-18.
[13] S. Chen, C. Delimitrou, J.F. Martinez, (2019). PARTIES: QoS-aware resource partitioning for multiple interactive services, in: Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), ACM, Providence, RI, 107-120.
[14] B. Beyer, C. Jones, J. Petoff, N.R. Murphy, (2016). Site Reliability Engineering: How Google Runs Production Systems, O'Reilly Media, Sebastopol, CA.