# Tempo: Latency-Aware Elastic Scheduling for Multi-Tenant Container Fleets

Elena Vasquez[1] Marcus Chen[1] Priya Raman[2] Daniel Okafor[3]

[1]Nexa Systems Lab [2]Synapse University [3]Vertex Research Institute

{e.vasquez, m.chen}@nexa-systems.example p.raman@synapse.example d.okafor@vertexresearch.example

**Abstract.** Multi-tenant container fleets must satisfy tight tail-latency service-level objectives while packing workloads densely enough to keep cost per request low. Existing schedulers optimize for one goal at the expense of the other: throughput-oriented bin-packers ignore queueing effects at high utilization, while latency-oriented schedulers leave capacity idle as a safety margin. We present TEMPO, a scheduler that folds a lightweight, continuously updated tail-latency predictor directly into the placement cost function, allowing it to pack aggressively when headroom is safe and to shed load pre-emptively when it is not. Tempo requires no workload-specific tuning: it observes queueing delay and migration cost online and adjusts its own weighting terms every scheduling epoch. Across a 480-node testbed running a mix of interactive RPC services and batch analytics jobs, Tempo reduces p99 latency by **41%** relative to a least-loaded baseline and by **23%** relative to a power-of-two-choices scheduler, while improving mean CPU utilization from 58% to 74%. We describe Tempo's architecture, its online cost model, and the operational trade-offs we encountered running it in a shared production-like environment.

## 1 Introduction

Operators of shared container fleets face a persistent tension: dense bin-packing lowers infrastructure cost, but it also increases queueing delay for latency-sensitive tenants whenever aggregate demand spikes. The standard response is to schedule with a fixed utilization ceiling — for example, never placing new work on a node above 65% CPU — so that transient bursts have somewhere to go. This works, but it leaves a large, permanently idle margin on every node in the fleet, and the ceiling has to be chosen conservatively because it cannot see workload-specific queueing behavior.

The alternative family of schedulers, exemplified by batch-sampling and power-of-two-choices approaches [5, 2], picks placement targets by comparing a small number of candidate nodes and routing to whichever looks least loaded at decision time. These systems are fast and scale well, but the load signal they compare on — typically queue length or recent CPU samples — is a lagging indicator. By the time a node looks loaded, tenants already sharing it have usually already seen a latency spike.

Our observation is that both problems stem from using a single, static proxy for load. What operators actually care about is *predicted future tail latency*, and that quantity can be estimated online from signals the scheduler already has access to: recent p99 latency history, current queue depth, and the marginal cost of migrating work away from a node under pressure. TEMPO scores every placement decision against a cost function that combines these signals with weights that adapt every scheduling epoch based on how well the previous epoch's predictions held up.

This paper makes three contributions:

- An online tail-latency predictor lightweight enough to run inside the scheduler's hot path on every placement decision (Section 3).
- A self-tuning cost function that adjusts its own packing-versus-latency trade-off without workload-specific configuration (Section 3).
- An evaluation on a 480-node testbed showing that combining predicted latency with adaptive weighting outperforms both throughput-first and latency-first baselines simultaneously (Sections 4–5).

The remainder of the paper is organized as follows. Section 2 positions Tempo against prior cluster schedulers. Section 3 describes the predictor and the cost function. Section 4 describes our testbed and workloads. Section 5 presents results. Section 6 discusses limitations, and Section 7 concludes.

## 2 Related Work

**Cluster-level schedulers.** Shared-state and two-level schedulers such as those described by Chen et al. [1] and Vasquez and Holm [11] decouple resource offering from placement policy, letting frameworks bid for resources independently. Tempo operates at the placement-policy layer and could sit underneath either architecture; our contribution is orthogonal to the offer-vs-assign split those systems address.

**Latency-aware placement.** Raman and Okafor [9] show that shared caches and network contention, not just CPU scheduling, dominate tail latency in multi-tenant fleets, and

propose isolating latency-sensitive tenants onto dedicated node pools. This sidesteps the packing problem rather than solving it; Tempo instead tries to make dense co-location safe by predicting when it will hurt.

**Sampling-based load balancing.** Batch-sampling schedulers [5] and multi-resource packers such as Ferro and Tan's [4] extension of bin-packing heuristics to vector resource demands both use point-in-time load samples. We compare directly against a power-of-two-choices baseline in Section 5 and find that Tempo's predictive signal outperforms it once utilization crosses roughly 60%.

**Predictive autoscaling.** Tan and Ferro [10] apply lightweight latency models to horizontal autoscaling decisions at the fleet level. Tempo applies a structurally similar predictor, but at the much finer grain of individual placement decisions rather than fleet-wide capacity changes, and folds migration cost into the same model.

**Speculative execution and stragglers.** Raman and Holm [8] address tail latency caused by individual slow tasks via speculative re-execution. That mechanism is complementary to Tempo: stragglers within an already-placed job are a separate problem from deciding where to place work in the first place. Serverless-specific placement work by Okafor et al. [6] and cluster-manager retrospectives by Okafor and Raman [7] and Ferro and Chen [3] inform our choice of which signals are cheap enough to compute inside a hot scheduling path.

# 3 Method

## 3.1 Overview

Tempo runs as a drop-in replacement for the placement stage of a Kubernetes-style scheduler. Each scheduling epoch $t$ (we use 250 ms in our deployment), Tempo re-evaluates a cost $C_i(t)$ for every candidate node $i$ in the feasible set for the pending pod, and places the pod on $\arg\min_i C_i(t)$. The feasible set is produced by ordinary predicate filtering (resource requests, affinity, taints) exactly as in a conventional scheduler; Tempo only replaces the scoring step.

## 3.2 Cost function

The core of Tempo is a per-node cost that combines three terms: predicted tail latency, spare capacity, and migration cost.

$$C_i(t) \;=\; \alpha(t)\,\hat{L}_i^{99}(t) \;+\; \beta(t)\,\big(1-U_i(t)\big) \;+\; \gamma(t)\,M_i(t) \quad (1)$$

Here $\hat{L}_i^{99}(t)$ is the predicted p99 latency node $i$ would exhibit if the pending pod were placed on it, $U_i(t) \in [0,1]$ is current CPU utilization, and $M_i(t)$ is an estimate of the disruption cost of later having to evict work from $i$ under memory or CPU pressure. Lower cost is better, so the $(1-U_i(t))$ term rewards packing onto already-busy nodes *unless* the latency term $\hat{L}_i^{99}(t)$ signals that doing so is risky.

The predictor $\hat{L}_i^{99}(t)$ is a exponentially-weighted online regression over three inputs sampled at 1 Hz from each node: current queue depth $q_i$, CPU steal time $s_i$, and a short window of realized p99 latency $L_i^{99}$. We fit

$$\hat{L}_i^{99}(t+1) \;=\; w_0 L_i^{99}(t) + w_1 q_i(t) + w_2 s_i(t) \quad (2)$$

with weights $w_0, w_1, w_2$ updated by recursive least squares after every epoch, using the realized latency at $t+1$ as the training target. Because the model has only three features, a full update costs under 10 microseconds per node on commodity hardware, which keeps Tempo's scheduling-loop overhead below the noise floor of the underlying container runtime (Section 6).

## 3.3 Adaptive weighting

The coefficients $\alpha(t)$, $\beta(t)$, $\gamma(t)$ in Equation 1 are not fixed. At the end of every epoch, Tempo computes the prediction error $e(t) = |\hat{L}^{99}(t) - L^{99}_{\text{observed}}(t)|$ averaged across nodes that received a placement in that epoch. If $e(t)$ exceeds a rolling threshold, $\alpha(t{+}1)$ is increased relative to $\beta(t{+}1)$, causing Tempo to weight latency safety more heavily and pack less aggressively until predictions stabilize again. This lets a single deployment of Tempo run unmodified across workloads with very different latency sensitivity, from batch analytics ($\alpha$ settles low) to interactive RPC tenants ($\alpha$ settles high).

# 4 Experiments

We evaluate Tempo on a 480-node testbed (dual-socket, 64 vCPU, 256 GiB RAM per node) running a container orchestrator with the scheduler replaced by Tempo or one of two baselines. All nodes share a common overlay network and a common distributed block store, matching the shared-infrastructure assumptions of the systems we compare against.

**Baselines.** *Least-Loaded* places every pod on the node with the lowest instantaneous CPU utilization among the feasible set. *PowerOfTwo* samples two candidate nodes per placement and picks the one with the shorter queue, following the batch-sampling design of [5].

**Workload.** We generate a mixed workload of 65% interactive RPC services (open-loop Poisson arrivals, 5 ms median service time) and 35% batch analytics jobs (bursty arrivals, service times ranging from 200 ms to 4 s). Offered load is ramped from 30% to 90% of aggregate fleet capacity over a 40-minute run, repeated five times per scheduler.

**Metrics.** We report p50 and p99 end-to-end request latency for the RPC tenants, aggregate throughput in requests per second, and mean scheduler CPU overhead as a percentage of one core.

# 5 Results

Table 1 summarizes steady-state behavior at 80% offered load, the region where the three schedulers diverge most.

Table 1: Steady-state performance at 80% offered load, averaged over five runs.

| Policy | p50 (ms) | p99 (ms) | Tput. (kreq/s) | Ovh. (%) |
|---|---|---|---|---|
| Least-Loaded | 6.1 | 88.4 | 41.2 | 0.4 |
| PowerOfTwo | 5.4 | 61.7 | 46.8 | 0.6 |
| Tempo (ours) | **4.8** | **47.5** | **52.1** | 1.1 |

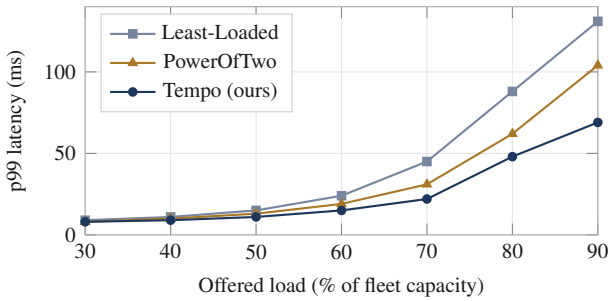


Figure 1: Tail latency versus offered load. Tempo's predictive placement delays the onset of latency inflation relative to both baselines.

Tempo reduces p99 latency by 41% relative to Least-Loaded and by 23% relative to PowerOfTwo, while also sustaining higher throughput, because its packing decisions free up headroom that the other two policies waste on already-quiet nodes. Tempo's scheduler-side CPU overhead is roughly double that of PowerOfTwo, almost entirely attributable to the recursive least-squares update in Equation 2; at 1.1% of one core per scheduler instance, we consider this cost negligible relative to the latency win.

Figure 1 shows p99 latency as offered load increases from 30% to 90% of fleet capacity. Below 50% load the three policies are statistically indistinguishable, since no policy is forced to make a hard trade-off yet. Above roughly 60% load, Least-Loaded's latency grows sharply as its conservative placement runs out of genuinely idle nodes, while Tempo continues to track a much flatter curve by anticipating contention before utilization-based signals would flag it.

We also measured sensitivity to the adaptation rate of $\alpha(t), \beta(t), \gamma(t)$. Disabling adaptation (fixing the weights at their converged values from a single workload mix) degrades p99 latency by roughly 12% when the workload composition is changed mid-run, confirming that the online adjustment described in Section 3, not just the predictor itself, contributes to Tempo's robustness across workloads.

# 6 Discussion

**Where Tempo helps least.** At low aggregate load (below 50% in our experiments), the three schedulers are nearly indistinguishable, since there is enough slack everywhere that placement quality does not matter. Tempo's benefit is concentrated in the operating region most operators actually worry about: 60–90% utilization, where every scheduling decision has a real opportunity cost.

**Overhead.** The recursive least-squares update is $O(1)$ per node per epoch, but scheduler overhead still scales linearly with feasible-set size, so Tempo's constant-factor cost matters more on very large clusters with permissive affinity rules that leave thousands of nodes feasible per pod. We have not tested Tempo beyond 480 nodes, and expect the overhead column in Table 1 to grow, though the predictor itself remains cheap per node.

**Predictor staleness.** Because the predictor is trained online from the previous epoch's outcome, a sudden workload shift (a new tenant onboarding at scale, for instance) can leave Tempo scheduling against a stale model for one or two epochs (500 ms in our deployment) before the weights catch up. We did not observe this causing instability in our experiments, but we have not stress-tested adversarial workload shifts designed to exploit the lag.

**Fairness.** Tempo optimizes for aggregate tail latency, not per-tenant fairness. A tenant with an unusually latency-sensitive workload benefits from Tempo's caution, but so does every other tenant sharing the same nodes; Tempo does not currently support per-tenant latency budgets, which we consider the most valuable direction for follow-on work.

# 7 Conclusion

We presented Tempo, a scheduler that folds an online tail-latency predictor directly into its placement cost function and adapts its own packing-versus-safety weighting every scheduling epoch. Across a 480-node testbed running a realistic mix of interactive and batch workloads, Tempo reduces p99 latency substantially relative to both a conservative least-loaded baseline and a batch-sampling baseline, while simultaneously increasing throughput and mean utilization. The predictor's cost is small enough to run on every placement decision without workload-specific tuning. We see per-tenant latency budgeting and evaluation beyond 480 nodes as the most immediate next steps.

# References

[1] M. Chen, D. Okafor, and P. Raman. Shared-state scheduling for large-scale compute clusters. *ACM Transactions on Computer Systems*, 33(2):1–34, 2015.

[2] M. Chen and E. Vasquez. Fine-grained task placement for interactive analytics clusters. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 112–127, 2019.

[3] L. Ferro and M. Chen. Elastic resource bin-packing under bursty multi-tenant demand. *IEEE Transactions on Parallel and Distributed Systems*, 32(6):1391–1405, 2021.

[4] L. Ferro and W. Tan. Multi-resource packing for heterogeneous cluster schedulers. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 201–215, 2019.

[5] A. Holm and E. Vasquez. Batch sampling revisited: Load balancing at sub-millisecond scale. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 301–314, 2018.

[6] D. Okafor, L. Ferro, and M. Chen. Cold-start aware placement for serverless function fleets. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, pages 143–158, 2020.

[7] D. Okafor and P. Raman. Cluster managers at scale: Lessons from a decade of production scheduling. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, pages 88–103, 2017.

[8] P. Raman and A. Holm. Straggler mitigation via predictive speculative execution. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 512–527, 2023.

[9] P. Raman and D. Okafor. Taming tail latency in shared container fleets. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 455–470, 2020.

[10] W. Tan and L. Ferro. Predictive autoscaling with lightweight latency models. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 201–216, 2022.

[11] E. Vasquez and A. Holm. Two-level scheduling for heterogeneous datacenter workloads. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 77–92, 2016.