JOB ORCHESTRATION CHALLENGES FOR AI WORKLOADS

Authors: Sabrina Di Leva, Kevin Hannon, Rajas Kakodkar, Ekin Karabulut, Victor Lu, Abhishek Malvankar, Alexander Scammon, Yuan Tang, Andrey Velichkevich, Marlow Warnicke

Contributors: Shravan Achar, Andrew Shunichi Aikawa, Ricardo Aravena, Malini Bhandaru, Sachi Desai, Johnu George, Josh Halley, Naadir Jeewa, Boris Kurktchiev, Niki Manoledaki, Claudia Misale, Mike SH Ng, Ronald Petty, Sudhanshu Prajapati, Brian Redmond, Joel Roberts, Vijay Rodrigues, Kay Yan, Michael Yao, Adel Zaalouk, Cathy Zhang

CLOUD NATIVE COMPUTING FOUNDATION

TABLE OF CONTENTS

Series Introduction 03

Executive Summary 03

Previously 04

Job Orchestration Challenges 04

- Gang Scheduling 04
- Resource Fairness and Quota Management 05
- Queue Management 06
- Preemption 06
- Priority Scheduling 07
- Resource Reservation and Backfill 08

What's Next 08



Series Introduction

This paper is the third in a five-part series on scheduling for cloud native AI workloads:

1. **Understanding AI Workloads for Kubernetes Scheduling:** What AI workloads are and why they differ from traditional cloud-native applications
2. **Scheduling Fundamentals for AI Workloads:** How Kubernetes scheduling works and why additional layers are needed
3. **Job Orchestration Challenges for AI Workloads (this paper):** Gang scheduling, fairness, queues, preemption, priority, and reservation
4. **Resource and Infrastructure Challenges for AI Workloads:** Topology, GPU sharing, scalability, I/O, fault tolerance, and cost
5. **Solutions and Practical Guidance for AI Workload Scheduling:** Tools, reference architectures, and real-world use cases

Each paper can be read independently, but together they provide a comprehensive guide to running AI workloads on Kubernetes.

Executive Summary

Running AI workloads on Kubernetes introduces job orchestration challenges that the default scheduler was not designed to handle. Distributed training requires all workers to start simultaneously (gang scheduling). Multiple teams competing for GPUs need fair resource allocation. Jobs must be prioritized, queued, and sometimes preempted to make room for higher-priority work. This paper examines these orchestration challenges in detail: gang scheduling, resource fairness, queue management, preemption, priority scheduling, and resource reservation with backfill.

Previously

The first two papers in this series established:

- **Paper 1:** AI workloads have fundamentally different characteristics than traditional microservices—they are long-running, tightly coupled, GPU-intensive, and topology-sensitive.
- **Paper 2:** The default Kubernetes scheduler operates at the pod level and does not understand job-level semantics. Meta-schedulers and queue managers are needed to bridge the gap.

This paper examines the specific job orchestration challenges that arise when running AI workloads.

Job Orchestration Challenges

This section outlines the scheduling challenges related to job orchestration—how jobs are admitted, ordered, and coordinated.

- Gang Scheduling
- Resource Fairness and Quota Management
- Queue Management
- Preemption
- Priority Scheduling
- Resource Reservation and Backfill

Gang Scheduling

- **Problem:** Distributed training jobs require all workers to run simultaneously. If a job needs 8 GPUs and only 6 are available, the default Kubernetes scheduler will start 6 pods and leave 2 pending. Those 6 GPUs sit idle—the job cannot make progress, and other jobs cannot use those resources. Worse, this can cause deadlock: multiple partial jobs hold resources while waiting for more, and none can complete.
- **Solution approach:** Gang scheduling treats a group of pods as a single unit. Either all pods are scheduled together, or none are. The scheduler waits until sufficient resources are available for the entire job before starting any pods.

- **Lifecycle impact:** Gang scheduling is critical across multiple stages of the AI lifecycle:
  - **Distributed training:** Workers use collective communication (all-reduce) that requires every participant. A partial allocation is useless.
  - **Multi-pod inference:** Model-parallel deployments and disaggregated serving architectures require all components (e.g., prefill and decode workers) to be running before the system can serve requests.
  - **Distributed data preparation:** Parallel jobs that must complete together to produce consistent output benefit from all-or-nothing scheduling.
- **Current state:** Kubernetes-native batch schedulers that support gang scheduling include the coscheduling plugin (via PodGroups), Armada, KAI Scheduler, Kueue, and Volcano. Native gang scheduling via the Workload API was introduced in Kubernetes 1.35 as alpha and reached beta in 1.37, where it remains disabled by default.

Resource Fairness and Quota Management

- **Problem:** Without controls, a single team or job can monopolize cluster resources, starving others. A large training job might consume all available GPUs for weeks, preventing other teams from running anything.
- **Solution approaches:**
  - Fair-share algorithms allocate resources proportionally. If Team A has 60% quota and Team B has 40%, resources are divided accordingly when both teams have pending work.
  - Hierarchical quotas allow nested allocation—a department gets a quota, then subdivides it among teams.
  - Borrowing and lending lets teams use idle quota from others, with mechanisms to reclaim it when the owner needs it back.
- **Key concepts:**
  - **Min/max quotas vs. hard quotas.** Hard quotas block any work beyond the limit. Min/max quotas guarantee a minimum while allowing borrowing up to a maximum. This improves utilization—unused quota doesn't sit idle.
  - **Fairness over time.** Batch workloads don't need instantaneous fairness. It's acceptable for one job to use 80% of the cluster for a week, as long as other teams get their share over the month.

- **Namespace-based vs. job-based fairness.** Kubernetes namespaces can represent teams, with quotas applied per namespace. But within a namespace, one user submitting 1,000 jobs shouldn't crowd out another user's 10 jobs—job-level or user-level fairness may also be needed.

Queue Management

- **Problem:** A busy cluster has many jobs competing for resources: urgent production inference, routine nightly training, experimental notebooks, batch data processing. Without organization, chaos ensues.
- **Solution approach:** Queue-based scheduling organizes workloads:
  - Hierarchical queues group related work. A "production" queue might have sub-queues for "inference" and "retraining." Each queue can have its own policies and resource allocation.
  - Priority classes determine which jobs run first when resources are scarce. Production inference might be highest priority; experimental training might be lowest.
- **Key mechanics:**
  - Enqueue decisions. Before a job enters a queue, the system checks whether it can ever run—does it request more resources than exist? Does it meet the minimum requirements for gang scheduling? Jobs that can never run should be rejected early, not left pending forever.
  - Task topology. Some jobs require specific hardware (GPUs, TPUs, FPGAs). A multi-level queue system can route jobs to appropriate resource pools automatically.

Preemption

- **Problem:** Higher-priority work arrives, but the cluster is full of lower-priority jobs. Without preemption, urgent work must wait for batch jobs to finish—which could take days.
- **Solution approach:** Preemption stops lower-priority work to free resources for higher-priority work. But naive preemption causes problems for AI workloads.
- **Key considerations:**
  - **Preemption vs. reclaim.** Preemption is proactive—stopping jobs to make room for new, higher-priority work. Reclaim is reactive—taking back borrowed resources when the owner needs them. Both are necessary.

- **Checkpointing before preemption.** A training job that's been running for 3 days holds valuable state. Killing it without warning loses that work. Cooperative preemption gives jobs time to checkpoint before being stopped.
- **Preemption within jobs.** In gang-scheduled jobs, preempting one worker may crash the entire job. If a job has 64 workers and you preempt 1, you might lose all 64 GPUs worth of work. Schedulers must understand job structure.
- **Cascading failures.** AI workflows have dependencies. Preempting a data preparation job may invalidate downstream training jobs that depended on its output.
- **Cost-aware preemption.** GPU time is expensive. Some systems allow "GPU time budgets"—jobs can be preempted if they exceed their allocated GPU-hours, forcing checkpoint and resume when resources are next available.
- **Avoiding waste.** Oversubscribing a cluster (promising more resources than exist) can backfire. Constant preemption churns resources, and the overhead of stopping/restarting jobs may waste more than it saves.

Priority Scheduling

- **Problem:** Not all jobs are equal. A production inference service that users are waiting on matters more than an experimental training run that can wait until tomorrow.
- **Solution approach:** Kubernetes has pod-level PriorityClasses, but AI workloads need job-level priority. Preempting one pod from each of ten jobs to fit one high-priority pod is worse than preempting one complete low-priority job.
- **Typical priority levels:**
  - Production inference — Highest priority; user-facing latency matters
  - Production training/retraining — High priority; needed to keep models fresh
  - Development/experimentation — Medium priority; can wait for resources
  - Batch/background jobs — Low priority; run when resources are idle
- **Interactive vs. batch:** Interactive workloads (notebooks, debugging) benefit from higher priority even if they use fewer resources—a data scientist waiting for a GPU to run a quick test is blocked until they get one.

Resource Reservation and Backfill

- **Problem:** A large job needs 256 GPUs. Smaller jobs keep arriving and consuming resources. The large job can never find a 256-GPU window and starves indefinitely.
- **Solution approach:**
  - Reservation locks resources for a specific job. The scheduler identifies which resources will be needed and stops scheduling new work to them, even if they're currently idle.
  - Backfill allows small, short jobs to use reserved resources temporarily, as long as they'll finish before the reserved job needs them.
- **Mechanics:** The scheduler estimates when reserved resources will be free (based on running jobs' expected completion), then allows backfill jobs that fit within that window. This requires jobs to declare (or the system to estimate) their expected runtime.

What's Next

This paper examined the job orchestration challenges for AI workloads. The next paper in this series, **Resource and Infrastructure Challenges for AI Workloads**, covers challenges related to where and how jobs run: topology awareness, resource heterogeneity, GPU utilization and sharing, scalability, I/O bottlenecks, fault tolerance, and cost constraints.


CLOUD NATIVE
COMPUTING FOUNDATION