SCHEDULING FUNDAMENTALS 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

Scheduling Fundamentals 04

- Unit of Scheduling 04
  - Kubernetes: Pod-Level Scheduling 04
  - Traditional HPC Schedulers: Task-Level Scheduling 05
  - AI Training Workloads: Job-Level Scheduling 05
  - AI Inference: Request-Level Scheduling 06
- Distributed Scheduling in Cloud Native Systems 06
  - The Kubernetes scheduler 06
- The Need for Meta-Schedulers 06

What's Next 08



Series Introduction

This paper is the second 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 (this paper):** How Kubernetes scheduling works and why additional layers are needed
3. **Job Orchestration Challenges for AI Workloads:** 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

The default Kubernetes scheduler operates at the pod level, making placement decisions independently for each pod without understanding relationships between them. This model works well for stateless microservices but falls short for AI workloads that require job-level scheduling, coordinated placement, and queue management. This paper explains how Kubernetes scheduling works, contrasts it with traditional HPC scheduling, and introduces the meta-schedulers and queue managers that bridge the gap for AI workloads.

Previously

The first paper in this series, **Understanding AI Workloads for Kubernetes Scheduling**, established what AI workloads are and why they differ from traditional cloud-native applications. Key points:

- AI workloads span multiple lifecycle stages with different resource profiles: data preparation (CPU/I/O bound), training (GPU-intensive, long-running), and inference (latency-sensitive, stateful)
- Training jobs are tightly coupled. All workers must run simultaneously for collective communication
- GPUs cannot be shared easily because workloads must fit entirely in GPU memory
- Performance depends heavily on hardware topology and interconnects

This paper builds on that foundation by examining how scheduling systems work and where they fall short for AI workloads.

Scheduling Fundamentals

Unit of Scheduling

Different systems operate at different abstraction levels to make scheduling possible for different classes of workloads. This section makes those units of scheduling explicit for different workload classes within a single cluster.

Kubernetes: Pod-Level Scheduling

In Kubernetes, the fundamental unit of scheduling is the **pod**. The default Kubernetes scheduler generally makes placement decisions independently for each pod, assigning it to a single node based on resource availability, constraints, and policies.

Characteristics:

- Scheduling decisions are per pod.
- The scheduler does **not understand relationships** between pods (e.g., that multiple pods belong to the same distributed training job).

In Kubernetes-native scheduling, the scheduling unit is a pod within **a single Kubernetes cluster.**

Traditional HPC Schedulers: Task-Level Scheduling

In traditional high-performance computing (HPC) environments, schedulers such as Slurm operate at the task level (also called a rank or process group member).

Characteristics:

- A job submission explicitly declares the number of tasks, CPUs, GPUs, and nodes required.
- Tasks are scheduled **as a single atomic unit**: either all tasks are allocated resources, or the job does not start.
- The scheduler has native awareness of job-level structure, topology requirements, and inter-task communication.

This model naturally supports gang scheduling, topology-aware placement, and reservation-based execution. These capabilities are particularly well-suited for model training workloads.

In HPC-style schedulers, the scheduling unit is a task within a job, with all tasks scheduled together.

AI Training Workloads: Job-Level Scheduling

For distributed AI training, the natural unit of scheduling is the **job**. Here, "job" refers to a complete unit of work in the general sense, not the specific Kubernetes Job resource.

Characteristics:

- Multiple workers (e.g., data-parallel or model-parallel processes, etc)
- Strong synchronization requirements across all workers
- The job cannot make progress unless **all required workers are running simultaneously**
- Partial allocation wastes resources and may block other work

AI training scheduling refers to job-level scheduling.

AI Inference: Request-Level Scheduling

For both real-time and batch inference, the effective scheduling unit is often an **inference request**, typically an HTTP or gRPC request.

Characteristics:

- The request's input and output lengths vary based on the user prompt, thereby affecting completion time.
- State management becomes an important consideration in the system.
- Resources for serving requests need to be scheduled on the correct hardware that satisfies the service level objective (SLO).

AI inference scheduling refers to the scheduling of requests to meet the target latency.

Distributed Scheduling in Cloud Native Systems

Kubernetes extends scheduling from a single machine to a cluster. Understanding the division of responsibilities is key to understanding where AI workloads hit limitations.

The Kubernetes scheduler

The default scheduler (kube-scheduler) has one job: assign pods to nodes. When a pod is created, the scheduler:

1. Filters nodes that cannot run the pod (insufficient resources, taints, affinity rules)
2. Scores the remaining nodes based on criteria like resource balance
3. Binds the pod to the highest-scoring node

The scheduler operates on individual pods. It does not understand relationships between pods—it does not know that eight pods belong to the same distributed training job and must all be scheduled together or not at all.

The Need for Meta-Schedulers

The gap between what Kubernetes provides and what AI workloads require has led to the addition of extra scheduling layers.

Why are additional layers needed? The default scheduler is designed for:

- Services that can start with partial capacity and scale up
- Pods that are mostly independent
- Workloads where any available resources are better than none

AI workloads need:

- All-or-nothing scheduling (gang scheduling)
- Coordinated placement across multiple pods
- Awareness of job-level semantics (not just pod-level)
- Queue management and fair sharing across teams
- Understanding of hardware topology

Building this into the core Kubernetes scheduler would add complexity that most users do not need. Despite Kubernetes having a pluggable scheduler framework, various workloads (high-throughput computing, for example) demand more than can be achieved with a scheduler plugin approach alone. Instead, the ecosystem has developed meta-schedulers and queue managers that use well-defined extensions to sit alongside or extend the default scheduler.

Event-driven scheduling. Some workloads should start in response to events rather than immediately upon submission:

- New data arrives in a storage bucket → trigger a data preparation pipeline
- A model training job completes → trigger an evaluation job
- An upstream job fails → trigger a notification or retry

Tools like Argo Events, Apache Airflow, and Flyte handle event-driven orchestration. They decide when jobs should run; the Kubernetes scheduler decides where the resulting pods are placed.

Queue-based scheduling. For AI workloads, queue-based scheduling provides:

- **Job queues.** Jobs are submitted to queues rather than immediately creating pods. The queue manager decides when to admit jobs based on available resources and policies.
- **Fair sharing.** Resources are divided among queues (often representing teams or projects). A team that has been using less than its share gets priority; a team over quota must wait.

- **Borrowing and lending.** Unused quota from one team can be temporarily used by another, then reclaimed when needed.
- **Gang scheduling.** The queue manager waits until all resources for a job are available before admitting it, preventing partial allocations.

Projects like Armada, KAI Scheduler, Kueue, Slinky, and Volcano provide these capabilities. They work with the Kubernetes scheduler—typically using mechanisms like pod scheduling gates or custom schedulers—to control when and how pods are scheduled.

The result is a layered architecture:

1. **Workflow orchestrators:** Decide what jobs to run and when
2. **Queue managers / batch schedulers:** Manage job admission, fairness, and gang scheduling
3. **Device plugins / DRA:** Manage specialized resources, such as GPUs and CPU alignment.

What's Next

This paper explained how Kubernetes scheduling works and why additional layers are needed for AI workloads. The next paper in this series, **Job Orchestration Challenges for AI Workloads**, dives into the specific challenges related to job orchestration: gang scheduling, resource fairness, queue management, preemption, priority scheduling, and resource reservation.



CLOUD NATIVE
COMPUTING FOUNDATION