SOLUTIONS AND PRACTICAL GUIDANCE FOR AI WORKLOAD SCHEDULING

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

Solutions Landscape 04

- Kubernetes Native Features 04
- Scheduler Extensions and Plugins 05
- Batch Schedulers and Queue Managers 05
- ML Platform Tools 06
- Data and Storage Tools 06
- Workflow Orchestration 07
- Non-Kubernetes Schedulers 07

Mapping Challenges to Solutions 08

Practical Guidance 10

- Choosing a Scheduling Stack 10
  - Common Patterns and Anti-Patterns 11
  - Getting Started 11

Conclusion 12



Series Introduction

This paper is the fifth and final in a 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:** 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 (this paper):** 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

Understanding AI workloads, scheduling fundamentals, and the challenges is necessary but not sufficient—organizations need to know what tools exist and how to apply them. This paper catalogs the solutions available for AI workload scheduling: Kubernetes native features, scheduler extensions, batch schedulers, ML platform tools, and workflow orchestrators. It provides a reference table mapping challenges to solutions, practical guidance on choosing a scheduling stack, common patterns and anti-patterns, and concrete use cases showing how organizations configure scheduling for different scenarios.

Previously

The first four papers in this series established:

- **Paper 1:** AI workloads span multiple lifecycle stages with different resource profiles. Training is GPU-intensive and tightly coupled; inference is latency-sensitive and stateful.
- **Paper 2:** The default Kubernetes scheduler operates at the pod level. Meta-schedulers and queue managers are needed for AI workloads.
- **Paper 3:** Job orchestration challenges include gang scheduling, resource fairness, queue management, preemption, priority scheduling, and resource reservation.
- **Paper 4:** Resource and infrastructure challenges include topology awareness, GPU sharing, scalability, I/O bottlenecks, fault tolerance, and cost constraints. Infrastructure considerations like GPU direct communications and network topology also affect scheduling decisions.

This paper catalogs the solutions and provides practical guidance on applying them.

Solutions Landscape

This section catalogs the tools and Kubernetes features that address the scheduling challenges discussed in Papers 3 and 4.

Kubernetes Native Features

Kubernetes includes several features relevant to AI workload scheduling:

- **Pod Priority and Preemption.** PriorityClasses assign priority levels to pods. Higher-priority pods can preempt lower-priority ones when resources are scarce. Limitation: operates at the pod level, not the job level.
- **Topology Spread Constraints.** Distribute pods across failure domains (regions, zones, nodes). Useful for resilience but not for optimizing GPU communication topology.
- **Mutable Scheduling Directives.** Allows modifying a Job's scheduling constraints while suspended. Custom queue controllers use this to implement admission control.
- **Non-preempting PriorityClass.** Jobs can have high priority (to queue ahead of others) without preempting running work. Useful for urgent jobs that can wait

for resources to free naturally.

- **Dynamic Resource Allocation (DRA).** A newer feature (GA, Kubernetes 1.34+) that enables flexible allocation of specialized resources. DRA supports on-demand GPU partitioning, allowing workloads to request specific GPU configurations rather than whole GPUs.

Scheduler Extensions and Plugins

The Kubernetes scheduler can be extended through plugins:

Scheduler Plugins (kubernetes-sigs/scheduler-plugins) is a repository of out-of-tree scheduler plugins. Notable plugins include:

- Coscheduling: Implements gang scheduling via PodGroups
- Capacity Scheduling: Elastic quota management with borrowing
- Node Resources: Enhanced resource allocation algorithms

Multiple plugins can run together, but interactions between plugins require careful configuration.

Batch Schedulers and Queue Managers

These tools add job queuing, fair sharing, and gang scheduling on top of Kubernetes:

- **Volcano** is a CNCF project providing comprehensive batch scheduling. Features include gang scheduling, fair-share policies, queue management, and topology-aware scheduling. Widely used for training workloads.
- **Kueue** (kubernetes-sigs) is a cloud-native Job scheduler that works with the default Kubernetes scheduler, the Job controller, and the cluster autoscaler to provide an end-to-end batch system. Kueue implements Job queueing, deciding when Jobs should wait and when they should start, based on quotas and a hierarchy for sharing resources fairly among teams.
- **KAI Scheduler** (formerly Run:ai Scheduler, a CNCF sandbox project) is an AI workload scheduler designed for large scale, high throughput clusters. Features include gang scheduling, topology awareness, hierarchical queues with quotas, fair-share policies (with/without time based), GPU sharing (partitioning, time-slicing), elastic workloads, and multi-framework support. Used in enterprise deployments with thousands of nodes.

- **Apache YuniKorn** provides unified scheduling for batch and interactive workloads with hierarchical queues and fair sharing.
- **Slinky** (opensource project maintained by NVIDIA) is a set of powerful integration tools designed to bring Slurm capabilities into Kubernetes, both allowing Slurm jobs to run within Kubernetes and bringing the full suite of Slurm scheduling capabilities into Kubernetes.

ML Platform Tools

These tools provide higher-level abstractions for AI workflows:

- **Kubeflow**
  - **Kubeflow Trainer** is a Kubernetes-native distributed AI platform for scalable LLM fine-tuning and training of AI models across a wide range of frameworks, including PyTorch, MLX, HuggingFace, DeepSpeed, JAX, XGBoost, and more. Provides job abstractions that handle worker coordination, including gang scheduling requirements and HPC workloads orchestration such as MPI and Flux.
  - **Kubeflow MPI Operator** enables MPI-based collective communication for distributed training. Essential for workloads using all-reduce synchronization.
  - **Kubeflow Katib** manages AutoML workloads—hyperparameter optimization and neural architecture search. Coordinates multiple training jobs with different configurations.
- **KServe** provides a standardized distributed generative and predictive AI inference platform for scalable, multi-framework deployment on Kubernetes.
- **MLflow** offers experiment tracking, model registry, and deployment tools. Not a scheduler, but integrates with scheduling systems for workflow management.
- **Ray / KubeRay** provides a distributed computing framework with its own scheduler. KubeRay is the Kubernetes operator for running Ray clusters. Useful for workloads that need Ray's programming model.

Data and Storage Tools

- **Fluid** (CNCF Sbx) accelerates data access for AI workloads through dataset caching and data movement optimization. Reduces I/O bottlenecks that leave GPUs idle.

- **HAMi** (Heterogeneous AI Computing Virtualization Middleware) enables flexible GPU sharing and partitioning across different GPU types.
- **Kubeflow** (CNCF incubating project):
  - **Kubeflow Data Cache** enables efficient data streaming for distributed training workloads.
  - **Kubeflow Spark Operator** aims to make specifying and running Apache Spark applications as easy and idiomatic as running other workloads on Kubernetes.
  - **Kubeflow Hub** provides a single pane of glass for ML model developers to index and manage models, versions, and ML artifacts metadata.

Workflow Orchestration

These tools manage multi-step ML pipelines:

- **Apache Airflow** is a general-purpose workflow orchestrator commonly used for ML pipelines. Defines DAGs (directed acyclic graphs) of tasks and handles scheduling, monitoring, and retries.
- **Argo Workflows** is a Kubernetes-native workflow engine. Workflows are defined as Kubernetes resources, making integration with other Kubernetes tools straightforward.
- **Flyte** is a workflow orchestration platform designed for ML. Provides versioning, caching, and strong typing for workflow inputs and outputs.
- **Kubeflow Pipelines** is a platform for building and deploying portable and scalable machine learning (ML) workflows using containers on Kubernetes-based systems.

Non-Kubernetes Schedulers

- **Slurm** is the dominant scheduler in HPC environments. Many organizations run Slurm alongside or instead of Kubernetes for AI workloads (especially training and multi-node inference), when they have existing HPC infrastructure. Integration approaches exist for bridging Slurm and Kubernetes environments.

Mapping Challenges to Solutions

A reference table that maps each challenge to the solutions that address it. All solutions in alphabetical order.

Challenge	Kubernetes Native	Schedulers	Platforms/ Tools	Applies To	Notes
Gang Scheduling	Workload API + GenericWorkload feature gate (beta, K8s 1.37, off by default)	Armada, Coscheduling plugin, KAI, Kueue, Slinky, Volcano, YuniKorn	Kubeflow Trainer	Both	Essential for distributed training
Resource Fairness	ResourceQuota (limited)	Armada, KAI, Kueue, Slinky, Volcano, YuniKorn	-	Both	Hierarchical quotas require external tools
Queue Management	-	Armada, KAI, Kueue, Slinky, Volcano, YuniKorn	Airflow, Argo Workflows, Flyte	Both	Core capability of batch schedulers
Preemption	PriorityClass (pod-level)	KAI, Kueue, Slinky, Volcano	-	Both	Job-level preemption needs external tools
Priority Scheduling	PriorityClass	All batch schedulers	-	Both	Job-level priority in batch schedulers
Reservation & Backfill	-	Slinky, Volcano, YuniKorn	-	Training	Advanced feature in some schedulers
Topology Awareness (Node)	Topology Manager (NUMA), DRA CPU Driver (CPU topology)	HAMi, KAI, Kueue, Slinky, Volcano	-	Both	GPU interconnect awareness varies
Topology Awareness (Cluster)	Topology Spread Constraints, DRANET (network DRA Driver) (limited)	KAI, Kueue, Slinky, Volcano	-	Both	Network topology awareness is emerging
Resource Heterogeneity	Node selectors, labels	All batch schedulers	-	Both	Standard Kubernetes features usually sufficient

GPU Sharing	DRA (GA, K8s 1.34+)	KAI	HAMi, KubeRay, Volcano	Both	MIG requires DRA or vendor tools
Scalability	Cluster Autoscaler, Karpenter	Armada, KAI, KEDA, Kueue, Slinky, Volcano	interLink	Both	Large-scale scheduling is challenging
I/O Bottlenecks	PersistentVolumes	-	Fluid	Both	Storage and caching solutions
Fault Tolerance	-	KAI, Slinky, Volcano (checkpoint support)	Kubeflow (elastic training)	Training	Framework-dependent
Elasticity	HPA, VPA	KAI, Slinky, Volcano	Kubeflow Trainer	Both	PyTorch Elastic, etc.
Budget/Cost	-	Limited support	Flyte (spot/ interruptible tasks)	Both	Emerging area
Inference request scheduling	Pod-level metrics	Dynamo Router, llm-d-inference scheduler	Kubernetes	Inference	Emerging area
Inference request autoscaling	Pod-level metrics, event-driven	Dynamo Planner, llm-d-WVA-autoscaler	KServe	Inference	Emerging area

- **Key observations:**
  - Gang scheduling was historically the most critical capability gap in default Kubernetes and is now being addressed natively: alpha in v1.35, beta in v1.37 (disabled by default), with GA targeted for v1.38.
  - Fairness and queuing require external tools for anything beyond basic ResourceQuota.
  - Topology awareness is partially addressed but remains an area of active development.
  - GPU sharing is becoming more accessible with DRA, but memory constraints limit applicability for large models.
  - Cost management is underserved by current tools.

Practical Guidance

Choosing a Scheduling Stack

The right scheduling stack depends on your workloads, scale, and existing infrastructure.

Start with the basics if:

- You run primarily single-node training or inference
- Your workloads don't require gang scheduling
- You have a small team and simple fairness requirements

In this case, standard Kubernetes features (PriorityClasses, ResourceQuota, node selectors) may suffice.

Deploy a batch scheduler if:

- You run workloads ranging from single pods to multi-pod configurations (such as distributed training or inference) that require gang scheduling.
- You have multiple hierarchical teams competing for GPU resources
- You need sophisticated fair-share policies
- You operate at significant scale (hundreds of GPUs or more)

Consider multi-cluster scheduling if:

- Your resource needs exceed a single cluster's capacity
- You have infrastructure distributed across regions or clouds
- You want unified job management across multiple clusters

Use a traditional HPC scheduler that integrates with Kubernetes if:

- You have existing HPC infrastructure and workflows
- You are running training workloads
- You have users that need a traditional HPC scheduler
- You need capabilities that Kubernetes schedulers don't yet provide
- You want hybrid abilities to run both traditional HPC workloads and Kubernetes loads on the same nodes

Common Patterns and Anti-Patterns

Patterns that work well:

- Separate queues for different workload types. Production inference, training, and experimentation have different priorities and resource profiles. Separate queues make policies clearer.
- Use non-preempting priority for urgent-but-not-critical work. Jobs that should run soon but don't need to kill running work can use high priority with preemptionPolicy: Never.
- Set realistic resource requests. Over-requesting wastes resources; under-requesting causes OOM kills. Profile your workloads.
- Use either framework or container state checkpointing for long-running jobs. Training jobs that run for days should checkpoint regularly. This enables preemption without total loss and recovery from failures and minimizes loss of valuable run time.
- Use topology constraints deliberately. For communication-heavy workloads, specify topology requirements. For resilience-critical workloads, spread across failure domains.

Anti-patterns to avoid:

- Running gang-scheduled workloads without a gang scheduler. Partial allocations waste resources and can deadlock.
- Oversubscribing without preemption policies. Promising more resources than exist only works if you have clear rules about who gets preempted when.
- Ignoring GPU memory constraints. Scheduling multiple workloads to a GPU when they won't fit in memory causes crashes, not sharing.
- Treating all GPUs as equivalent. Different GPU models, memory sizes, and interconnects matter. Jobs that need current-generation GPUs shouldn't land on previous-generation hardware.
- No quotas in multi-tenant clusters. Without quotas, one team's large job can starve everyone else indefinitely.

Getting Started

For platform engineers building new infrastructure:

1. Start with a single batch scheduler to handle basic queuing and quota management.

2. Define queues that map to your organizational structure (teams, projects, environments).
3. Set quotas based on expected resource allocation—you can adjust later.
4. Ensure gang scheduling support if you run distributed training.
5. Implement monitoring to understand actual resource usage patterns.

For ML engineers working with existing infrastructure:

1. Understand what scheduling tools are available in your cluster.
2. Use the appropriate job abstractions (MPIJob, PyTorchJob, TrainJob, etc.) rather than raw pods.
3. Set accurate resource requests based on profiling.
4. Implement checkpointing for any training job longer than a few hours.
5. Work with platform engineers to tune quotas and priorities for your workloads.

For organizations evaluating their options:

1. Inventory your workloads: What fraction need gang scheduling? GPU sharing? Low latency?
2. Assess your scale: How many GPUs? How many concurrent users/teams?
3. Evaluate your existing infrastructure: Kubernetes-only? Hybrid with HPC?
4. Start small and iterate: Deploy a scheduler, run real workloads, observe, adjust.

Conclusion

Kubernetes was not initially designed for AI workloads; however, it has become one of the de facto standards for managing them. Its default scheduler excels at placing stateless microservices but historically lacked capabilities that machine learning jobs require: gang scheduling, fair sharing across teams, topology awareness for GPU interconnects, and job-level (not just pod-level) management. The ecosystem has responded, and Kubernetes itself continues to evolve to address these gaps. Key takeaways from this series:

1. Understand your workloads. Different stages of the ML lifecycle have different requirements. Data preparation, training, and inference each need different scheduling strategies.
2. Gang scheduling is essential for AI workloads. Without it, partial allocations waste resources and cause deadlocks. If you run distributed training, you need a scheduler that supports gang scheduling.

3. Fairness requires explicit policies. ResourceQuota provides basic limits, but real multi-tenant clusters need hierarchical quotas, borrowing/lending, and fair-share algorithms.
4. Topology matters. For communication-intensive workloads, placing workers on well-connected nodes significantly improves performance. This is an area of active development.
5. The ecosystem is maturing. Projects like Kueue, Slinky, Volcano, and KAI Scheduler are production-ready. The core of Dynamic Resource Allocation graduated to general availability in Kubernetes v1.34; the DynamicResourceAllocation feature gate was subsequently locked to enabled-by-default in v1.35. The tools exist—the question is choosing and integrating them.
6. Start simple, iterate. You don't need every feature on day one. Start with basic queuing and quotas, and add capabilities as your needs grow.

The cloud-native AI landscape continues to evolve. New hardware (different GPU architectures, accelerators), new workload patterns (agentic systems, distributed inference), and new Kubernetes features will create both challenges and opportunities. This series provides a foundation; staying current requires ongoing engagement with the community.



CLOUD NATIVE
COMPUTING FOUNDATION