UNDERSTANDING AI WORKLOADS FOR KUBERNETES 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

Introduction 04

- Purpose and Scope 04
- Context 04

Understanding AI Workloads 05

- The AI/ML Lifecycle 05
  - Data Preparation 05
  - Model Development 06
  - Model Training 06
  - Hyperparameter Tuning 07
  - Model Inference 07
  - Emerging Patterns 08
- What Makes AI Workloads Different 08
- Resource Characteristics by Lifecycle Stage 10
- What Makes Training and Real-Time Inference Workloads Different 11
  - Scheduling Needs 11

What's Next 12



Series Introduction

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

1. **Understanding AI Workloads for Kubernetes Scheduling (this paper):** 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:** 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

AI and machine learning workloads have fundamentally different characteristics than the stateless microservices Kubernetes was originally designed for. Training jobs consume expensive GPUs for days or weeks, require all workers to start simultaneously, and are sensitive to hardware topology. Inference workloads must maintain low latency while managing substantial state in GPU memory. This paper explains the AI/ML lifecycle, examines the resource requirements of each stage, and identifies why these workloads present unique scheduling challenges that traditional Kubernetes scheduling cannot address.

Introduction

Purpose and Scope

This paper provides the foundational understanding of AI workloads necessary to address their scheduling requirements on Kubernetes.

What this paper does:

- Explains the stages of the AI/ML lifecycle and their resource characteristics
- Identifies what makes AI workloads different from traditional cloud-native applications
- Examines the distinct scheduling needs of training and inference workloads

Who this paper is for:

- Platform engineers building infrastructure for ML teams
- ML engineers who need to understand the infrastructure layer
- Infrastructure teams evaluating scheduling solutions for AI workloads

What this paper assumes:

- Basic familiarity with Kubernetes concepts (pods, nodes, deployments)
- No deep expertise in machine learning algorithms is required

Context

Kubernetes has become the standard platform for deploying containerized applications. Its default scheduler excels at placing stateless microservices: it finds a node with sufficient CPU and memory, starts the pod, and moves on. If a pod fails, the system restarts it. If demand increases, horizontal autoscaling adds more pods. This model works well for web servers, APIs, and similar workloads.

AI workloads break these assumptions. Understanding why requires examining the AI/ML lifecycle in detail.

Suggested pre-reading:

If you are new to Kubernetes scheduling, you may find these resources helpful:

- Kubernetes Scheduler documentation
- Pod Priority and Preemption

- Resource Management for Pods and Containers
- Dynamic Resource Allocation
- Workload API

Understanding AI Workloads

The AI/ML Lifecycle

AI and machine learning projects follow a lifecycle with distinct stages. Each stage has different resource requirements and scheduling characteristics.

- Data Preparation
- Model Development
- Model Training
- Model Inference
- Emerging Patterns

Data Preparation

Data preparation transforms raw data into a format suitable for model training. This includes:

- **Data collection:** Gathering data from various sources (databases, APIs, file systems)
- **Data cleaning:** Removing duplicates, handling missing values, fixing inconsistencies
- **Data transformation:** Normalizing, encoding categorical variables, feature scaling
- **Data splitting:** Dividing data into training, validation, and test sets

From a scheduling perspective, data preparation is typically CPU and I/O intensive rather than GPU-intensive. That said, GPU-accelerated frameworks can significantly speed up large-scale data processing tasks such as filtering, joining, and aggregating datasets. Additionally, GPUs work well for unstructured data like images because image processing involves massive parallel math operations. Jobs are often parallelizable—you can clean different partitions of a dataset independently. Event-driven scheduling is common: new data arriving triggers a preparation pipeline.

Kubernetes resources like Jobs and CronJobs handle these workloads reasonably well. Workflow orchestrators (Airflow, Argo Workflows, Flyte) coordinate multi-step pipelines.

Model Development

Model development has two distinct activities that are often combined:

- **Feature engineering** transforms prepared data into input features the model can use. This involves creating new variables, encoding categorical data, and selecting which features to include. Feature engineering is computationally similar to data preparation—CPU and I/O bound, parallelizable, often triggered by new data.
- **Model architecture** involves selecting the type of model (linear regression, decision tree, neural network, transformer) and designing its structure. For deep learning, this means defining layers, attention mechanisms, and other architectural choices. This work is often interactive—a data scientist experimenting in a notebook—and does not require significant compute resources until training begins.

The scheduling needs for model development are modest. Interactive notebook environments need stable, long-running pods. Feature engineering jobs look like data preparation jobs. These characteristics require additional Kubernetes scheduler capabilities to support efficient all-or-nothing placement and topology-aware scheduling.

Model Training

Model training is the process of exposing a model to data and iteratively adjusting its parameters (weights and biases) so it learns patterns that generalize to new inputs.

Training requirements vary enormously depending on the model:

- Training a linear regression model takes seconds on a laptop
- Training a random forest might take minutes to hours on a single machine
- Training a large language model takes weeks to months on thousands of GPUs

For deep learning at scale, training is distributed across multiple machines. Workers communicate using collective operations like all-reduce, which requires all participants to synchronize. This creates the gang scheduling requirement: if you need 64 workers and only 60 are available, the job cannot start. If one worker fails mid-training, the entire job may need to restart from a checkpoint.

Training jobs are:

- **Long-running:** Days to months for large models
- **Resource-intensive:** Hundreds to thousands of GPUs
- **Tightly coupled:** All workers must run simultaneously
- **Sensitive to topology:** Communication speed depends on GPU interconnects

These characteristics require additional Kubernetes scheduler capabilities to support efficient all-or-nothing placement and topology-aware scheduling.

Hyperparameter Tuning

During the model training phase, users want to tune hyperparameters to optimize how the model learns rather than the structure of the model itself. This includes adjusting parameters such as learning rate, batch size, optimizer choice, number of epochs, and dropout rates. Unlike architecture design, hyperparameter tuning is compute-intensive because it requires repeatedly training and evaluating many model variants. These OptimizationJobs are highly parallelizable and are commonly distributed across GPUs or clusters.

Model Inference

After training, the model is deployed to make predictions on new data. Inference has two common patterns:

- **Batch inference** processes large datasets offline. For example, running a trained model over satellite imagery for an entire continent, or scoring millions of records for a recommendation system. Batch inference looks like a large parallel job—it can use many GPUs, but individual tasks are independent. Gang scheduling is typically not required.
- **Real-time inference** responds to individual requests with low latency. A user sends a query; the model returns a prediction in milliseconds. Real-time inference requires:
  - Models preloaded in GPU memory (loading a model can take longer than serving a request)
  - Horizontal scaling to handle variable request rates
  - Low-latency networking

Resource requirements per request are often lower than those of large-scale training, but real-time inference creates its own scheduling challenges: keeping models loaded, autoscaling with demand to meet SLOs (Service Level Objectives), and handling multiple base models competing for GPUs.

Emerging Patterns

- **Agentic pipelines** chain multiple model calls together, with models making decisions about what to do next. These create dynamic, unpredictable workloads that are difficult to schedule in advance.

- **Disaggregated inference** separates LLM serving into distinct phases; typically prefill (processing the input prompt) and decode (generating output tokens) that run on different hardware. Prefill is compute-intensive; decode is memory-bandwidth bound. This architecture enables independent optimization and scaling of each phase but introduces new scheduling requirements: KV cache (a data structure that stores intermediate computations from the prefill phase so they don't need to be recalculated during decoding) must transfer between phases, components need coordinated topology aware placement, and the system must maintain minimum viable combinations while scaling each component type independently.

What Makes AI Workloads Different

Traditional cloud-native applications—web servers, APIs, microservices—share common characteristics that Kubernetes handles well:

- Stateless or easily replicated state
- Short request/response cycles
- Horizontal scaling by adding more pods
- Graceful degradation under load (serve some requests slower, drop others)
- Pods are fungible; any pod can handle any request

AI workloads break these assumptions:

- **Long-running jobs.** A training run is not a request that completes in milliseconds. It is a job that runs for days or weeks. Interrupting it wastes all the work done since the last checkpoint. The scheduler must account for job duration, not just instantaneous resource needs.

- **Massive resource consumption**. Training large models requires hundreds or thousands of GPUs running simultaneously. A single job can consume

the majority of a cluster's capacity for extended periods. This is not "scale horizontally by adding pods"—it is "reserve a large fraction of the cluster for one workload."

- **Tightly coupled distribution.** Distributed training uses collective communication patterns where all workers must participate. You cannot start with 7 of 8 workers and add the 8th later. You cannot lose one worker and continue with the remaining 7. Either all workers are running, or the job cannot proceed. This is fundamentally different from web services, where losing one replica just shifts load to the others.
- **A large amount of state.** AI inference workloads answer user queries by creating and maintaining substantial state in GPU memory. This state management is key to consider when scaling AI inference server pods up or down while maintaining latency and SLOs.
- **GPU memory constraints.** LLM models consume all available GPU memory. A model that requires 80GB of VRAM cannot share a GPU with another workload—there is no memory left. This makes GPU sharing difficult. Loading and unloading models is slow (tens of GB transferred from storage), so swapping models for different requests is impractical for real-time inference.
- **I/O patterns.** AI workloads have distinct I/O phases:
  - Data loading at the start of training can take 10-70% of total execution time (per research from Google and Microsoft), leaving GPUs idle
  - Checkpointing periodically saves model state to storage, creating bursty write patterns
  - Model loading for inference transfers large model files from storage to GPU memory before serving can begin
- **Topology sensitivity.** Performance depends on which specific GPUs are allocated and how they are connected. Two GPUs on the same node with high-speed interconnects can communicate at hundreds of GB/s. Two GPUs on different nodes communicate over the network at 10-100 GB/s. For workloads dominated by communication (like distributed training with frequent synchronization), this difference determines whether a job takes hours or days.
- **Latency requirements.** Some jobs have high latency tolerances and others, such as inference, have much lower latency tolerances, as there is an end user waiting for a response.

Resource Characteristics by Lifecycle Stage

The following table summarizes the resource profile of each lifecycle stage:

Stage	Primary Resources	Duration	Scheduling Characteristics
Data Preparation	CPU, storage I/O, network	Minutes to hours	Parallelizable, event-driven, elastic by nature
Feature Engineering	CPU, memory, storage I/O	Minutes to hours	Similar to data preparation
Model Development	CPU (notebooks), minimal GPU for experiments	Interactive sessions	Long-running pods, modest resources
Training (small models)	1-8 GPUs	Hours to days	Standard job scheduling
Pre-Training (large models)	100s-1000s of GPUs	Days to months	Gang scheduling, topology awareness, checkpointing
Post Training (fine-tuning)	1-64 GPUs	Hours to days	Similar as pre-training
Batch Inference	Variable GPU count	Hours to days	Parallelizable, throughput-oriented
Real-time Inference	GPUs with models preloaded	Continuous	Low latency, autoscaling, model serving

The key insight: different stages need different scheduling strategies. A cluster running the full ML lifecycle must handle event-driven pipelines, interactive notebooks, gang-scheduled training, and latency-sensitive inference—often simultaneously, competing for the same GPU resources.

Of these stages, training and real-time inference present the most significant scheduling challenges and the most significant conflicts when sharing infrastructure. The following section examines their requirements in detail.

What Makes Training and Real-Time Inference Workloads Different

Model training and model inference represent different workload classes with distinct resource consumption patterns, scheduling needs, and optimization objectives. Infrastructure supporting both workload types requires scheduling mechanisms capable of managing both of their requirements.

Scheduling Needs

Training workloads have the following scheduling needs:

- **Gang scheduling:** Distributed training using collective communication (e.g., all-reduce) requires all workers to execute simultaneously. Partial allocation results in resource waste and potential deadlock.
- **Topology awareness:** Inter-worker communication performance depends on GPU interconnect topology. Suboptimal placement can degrade training throughput by an order of magnitude.
- **Long resource commitment:** Resources remain allocated for the duration of training, which may span days to weeks.

Inference workloads impose different needs and objectives:

- **Gang scheduling:** Inference workloads spanning multiple pods also require all-or-nothing scheduling. A model-parallel or disaggregated deployment is non-functional if only some components are scheduled.
    - **Hierarchical gang scheduling:** Disaggregated inference adds complexity. The system must guarantee minimum viable component combinations (e.g., at least one prefill and one decode worker) while allowing independent scaling. Prefill scales with prompt length and request rate; decode scales with output length and concurrent sessions. Traditional gang scheduling conflicts with this need by forcing all components to scale together.
- **Topology awareness:** Multi-pod inference deployments (whether model-parallel or disaggregated) require low-latency interconnects between pods. This topology must be preserved across rolling updates when new versions are deployed.
- **SLO and availability:** SLOs specify response times in milliseconds. Unlike batch jobs, real-time inference services run continuously with no completion state. Resources must remain available while meeting latency and throughput targets.

- **Scaling:** Inference scaling spans multiple layers: autoscalers monitor request metrics and adjust replica counts; cluster schedulers handle pod placement; inference request routers distribute requests within the fleet. Effective scaling requires coordination across all three.
    - **Multi-level autoscaling:** Disaggregated inference requires autoscaling at multiple levels; individual components (prefill workers for traffic spikes), related component groups (prefill leaders with their workers), and entire service replicas for overall capacity. These levels affect one another: scaling prefill workers may require more decode capacity, and new service replicas need proper component ratios.
- **Cold start tradeoff:** Pre-deployed replicas guarantee low latency but consume resources when idle. Reactive scaling is efficient but incurs cold start latency (seconds to minutes) while model weights load. Accelerated model loading (fast storage, model streaming) reduces this penalty. The choice depends on SLO requirements and cost tolerance.

What's Next

This paper established what AI workloads are and why they differ from traditional cloud-native applications. The next paper in this series, **Scheduling Fundamentals for AI Workloads**, examines how Kubernetes scheduling works, introduces the different units of scheduling for different workload types, and explains why additional layers like meta-schedulers and queue managers are needed for AI workloads.


CLOUD NATIVE
COMPUTING FOUNDATION