



GPU-Enabled Platforms on Kubernetes

Daniele Polencic, LearnKube

Saiyam Pathak, vCluster



The GPU Challenge at Scale

Table of contents

Foundations - How GPUs Meet Kubernetes	7
Syscalls: The Kernel's API Surface	8
Control Groups (cgroups): Enforcing Resource Limits	11
Namespaces: Isolating What a Process Can See	13
From Kernel Primitives to Docker	15
CPU: Preemptive Multitasking	15
Memory: Page-by-Page Allocation	18
CUDA Fundamentals: Understanding Contexts, Kernels, and Memory	20
CUDA Kernels: The GPU Programs	23
Why GPUs Don't Support Kernel Preemption	27
GPU Memory: A Completely Different Model	29
The Critical Differences	32
Kubernetes and the Container Runtime Interface	34
The GPU Problem in Kubernetes	37

Running a GPU Pod: End-to-End	45
The Sharing Challenge	51
Key Takeaways	52
Why GPU Multi-Tenancy Is Hard	53
Traditional Kubernetes Isolation: The Foundation That Works	54
Namespaces: Control Plane Separation	55
Cgroups: The kernel is always in control	57
RBAC: Access Control Boundaries	58
Why This Multi-Tenancy Model Works	62
Why This Model Collapses for GPUs	62
GPU Scheduling Happens Outside the Kernel	63
CUDA Contexts Span Containers	64
Kubernetes Enforces Scheduling Only at the API Layer	65
Threat Scenarios in Practice	66
Time-Slicing: Multiplying GPU Count	72
vGPU: The Enterprise Solution	79
The Trust Spectrum: Choosing Your Strategy	81
Comparing All Options	82

Orchestrating GPU Sharing - How Kubernetes Manages Turn-85 Taking

The Hidden Truth: GPUs Already Support Sharing	86
Understanding GPU Context Switching	87
The Memory Illusion	89
The Kubernetes Orchestration Problem	93
KAI-Scheduler's Reservation Pod Strategy	94
KAI: The Clever Parts and the Limitations	99
NVIDIA "Time-Slicing" - The Misleading Name	100
Why "Time-Slicing" Is Misleading	102
Time-slicing Critical Limitations	103
Comparing the Approaches: Same Reality, Different Tricks	105
A Practical Example: The Scheduler Showdown	107
Orchestrating sharing	109
Key Takeaways	110
Hardware Isolation and Enforcement	111
The Fundamental Difference: Parallel vs Sequential	112
MIG: When NVIDIA Decided to Fix This in Silicon	114
Why Seven Slices?	115

Creating MIG Instances	117
Using MIG Instances in Pods	120
MIG's Achilles' Heel: Cost and Architecture Trade-offs	123
HAMi: The Software Enforcement Revolution	125
Compute Throttling: Token Bucket in Action	128
The Reality of Software Enforcement	133
Production Considerations	135
The Evolution of GPU Sharing	137
Key Takeaways	138
Monitoring GPU Clusters	139
The Three Views of GPU Reality	141
How nvidia-smi Actually Measures	143
The Three-Layer Memory	144
The Zombie Process Problem	145
Why Zombie Processes Hold GPU Memory	146
The Cascade Effect of GPU Hoarding	154
DCGM: Bridging the Reality Gap	156
The Allocation vs Utilization Gap	158

Finding GPU Hoarders	159
Monitoring Patterns That Actually Matter	162
The Reality Check	165
Key Takeaways	166
Multi-Tenant GPU Platforms with vCluster	167
The Alternative	168
Understanding the vCluster Architecture: A Primer	171
Demo: Running Kai Scheduler with vCluster for Fractional GPU Sharing (with Ollama RAG Use Cases)	186
1. Prerequisites	188
2. NVIDIA GPU Operator Setup	189
3. Install KAI-Scheduler	191
4. Create a vCluster with Kai Integration	192
5. Deploy Application	193
5. Final Outcome	196
Conclusion	196

Chapter 2

Why GPU Multi- Tenancy Is Hard

In this chapter, you will learn:

- How traditional Kubernetes isolation mechanisms (namespaces, cgroups, RBAC) work and why they fail for GPUs
- The specific security vulnerabilities that emerge when sharing GPUs across tenants
- How to evaluate different GPU sharing approaches (MPS, time-slicing, MIG, vGPU) based on your trust model
- A practical decision framework for choosing the right GPU sharing strategy for your organization

Let's begin with how Kubernetes enforces multi-tenancy when no GPUs are involved.

Traditional Kubernetes Isolation: The Foundation That Works

Kubernetes workloads often belong to different teams, departments, or even customers.

They must share the same physical nodes without interfering with each other.

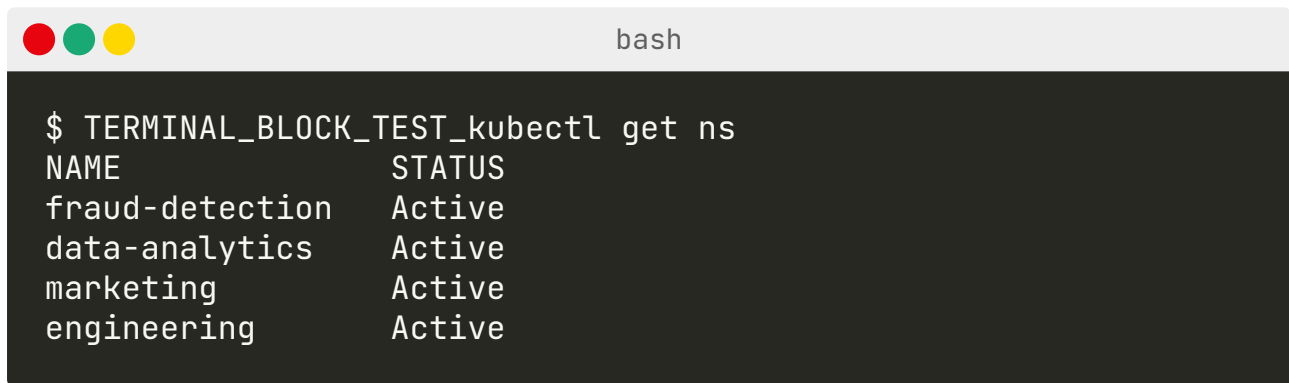
Kubernetes enforces this through three pillars: namespaces, cgroups, and RBAC.

Namespaces carve the cluster into logical domains, Cgroups enforce physical resource limits directly at the Linux kernel level, RBAC governs who can access which resources through the API.

Together, these layers give Kubernetes a powerful model: workloads can run side by side, each isolated in what it can see, consume, and control.

Namespaces: Control Plane Separation

Namespaces divide Kubernetes resources into logical domains. Each tenant—whether a team, department, or customer—gets its own namespace.

A terminal window with a light gray title bar containing three colored window control buttons (red, green, yellow) on the left and the text 'bash' on the right. The terminal area has a dark background and displays the output of the command 'kubectl get ns'.

```
$ kubectl get ns
NAME                STATUS
fraud-detection     Active
data-analytics      Active
marketing           Active
engineering         Active
```

At first glance this looks like strong isolation.

But namespaces only exist in the Kubernetes API.

They define who can see or manage resources through the control plane, not what happens at the kernel or hardware level.

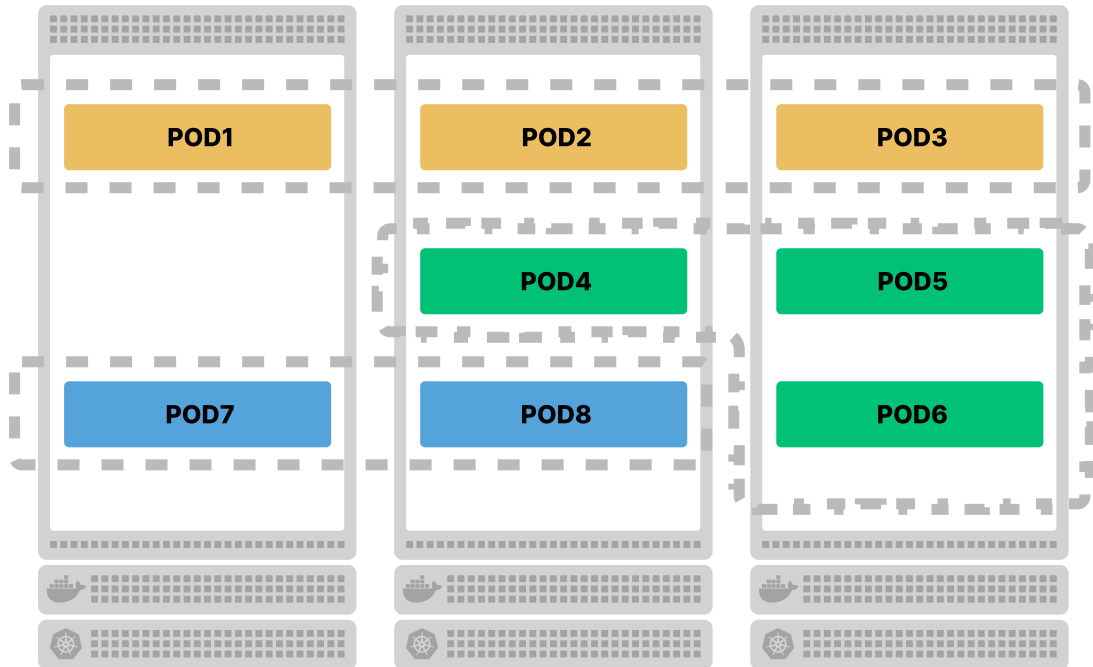


Fig. Kubernetes namespaces don't provide isolation and can span several physical nodes

To illustrate, let's deploy pods across namespaces:

```
bash

$ TERMINAL_BLOCK_TEST_kubectl get pods -A -o wide
NAMESPACE          NAME                STATUS    NODE
fraud-detection    fraud-inference     Running   minikube
data-analytics     analytics-job        Running   minikube
marketing           campaign-analysis    Running   minikube
engineering         model-training       Running   minikube
```

Every pod runs on the same physical node.

Chapter 7

Multi-Tenant GPU Platforms with vCluster

Now that you understand the need for GPUs in modern AI workloads, let's examine a typical enterprise scenario.

Imagine an organization that has provisioned **5 bare metal GPU nodes** and **5 bare metal CPU nodes**, whether on-prem or in the cloud (the same principles apply).

These resources are meant for internal teams working on AI workloads, such as building LLM-based agentic applications, training models, or running inference pipelines.

Now consider this: there are **10 separate AI/ML teams** in the organization, and each team wants its own Kubernetes cluster to maintain autonomy and isolation.

As the infrastructure engineer, you're responsible for enabling this.

Let's say you use kubeadm to create clusters.

At a bare minimum, each cluster needs:

- **1 control plane node**
- **1 CPU node**
- **1 GPU node**

That's **3 nodes per team × 10 teams = 30 nodes**, but you only have **10 nodes total**.

Even if you use only two nodes per team, you're still far from being able to provision dedicated clusters without ordering new hardware (which takes time) or scaling up cloud instances (which can be prohibitively expensive).

The Alternative

How do you share the same Kubernetes cluster across multiple teams without compromising their autonomy? That brings us to multi-tenancy.

First Attempt: Kubernetes Namespaces

The default multi-tenancy primitive in Kubernetes is the Namespace. It allows you to segment resources within the same cluster logically.

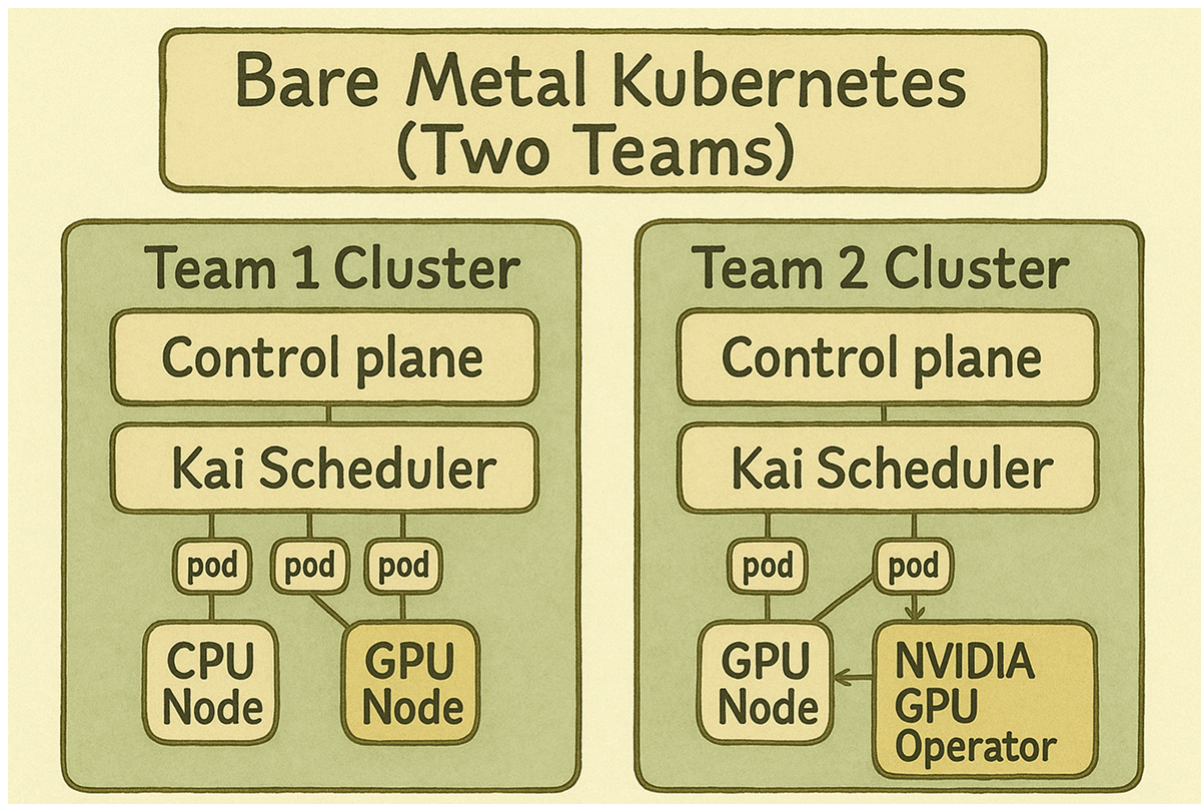


Fig. Bare metal Kubernetes setup for two teams showing Team 1 and Team 2 clusters, each with control plane, Kai Scheduler, pods, and dedicated CPU/GPU nodes, with Team 2 also having NVIDIA GPU Operator

Instead of using the default scheduler, you set up a Kubernetes cluster with Kai Scheduler and create separate Kubernetes clusters for the teams.

But let's return to the main problem: we still want to utilize all GPUs, and other teams are still waiting for their Kubernetes clusters.

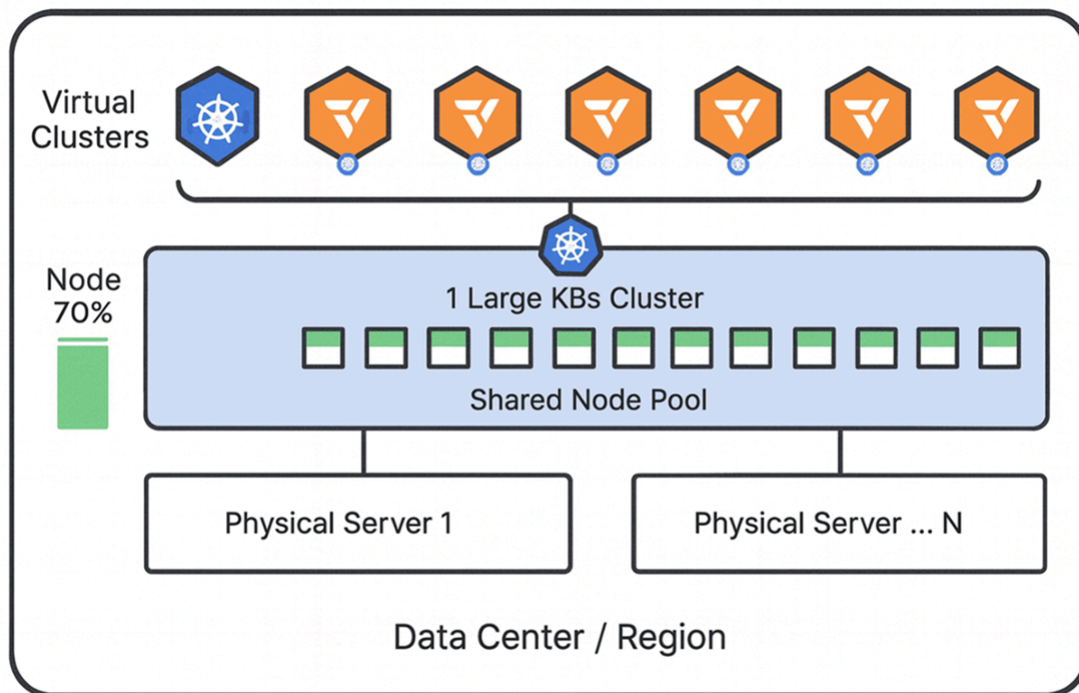


Fig. Data center architecture showing multiple virtual clusters, a single large Kubernetes cluster with shared node pool, and physical servers with 70% node utilization

The data center should now have a single large Kubernetes cluster and multiple virtual Kubernetes clusters, one for each team.

Demo: Running Kai Scheduler with vCluster for Fractional GPU Sharing (with Ollama RAG Use Cases)