vCluster learnKUBE

GPU-Enabled Platforms on Kubernetes

Daniele Polencic, LearnKube

Saiyam Pathak, vCluster

The GPU Challenge at Scale

Table of contents

Chapter 1

Foundations - How GPUs Meet Kubernetes

In this chapter, you will learn:

- How containers work through syscalls, cgroups, and namespaces
- Why GPUs break every assumption about resource isolation
- How device plugins bridge GPUs into Kubernetes
- The integer resource problem and its implications

Let's start with understanding how the Linux kernel provides the foundation for container isolation.

Syscalls: The Kernel's API Surface

In Linux, user-space applications can't interact with hardware directly.
Every interaction must go through the Linux kernel; the only sanctioned way to do that is via system calls.
System calls are predefined entry points that expose core Linux kernel functionality.
When your Java application accesses a file on the filesystem, it initiates a syscall.

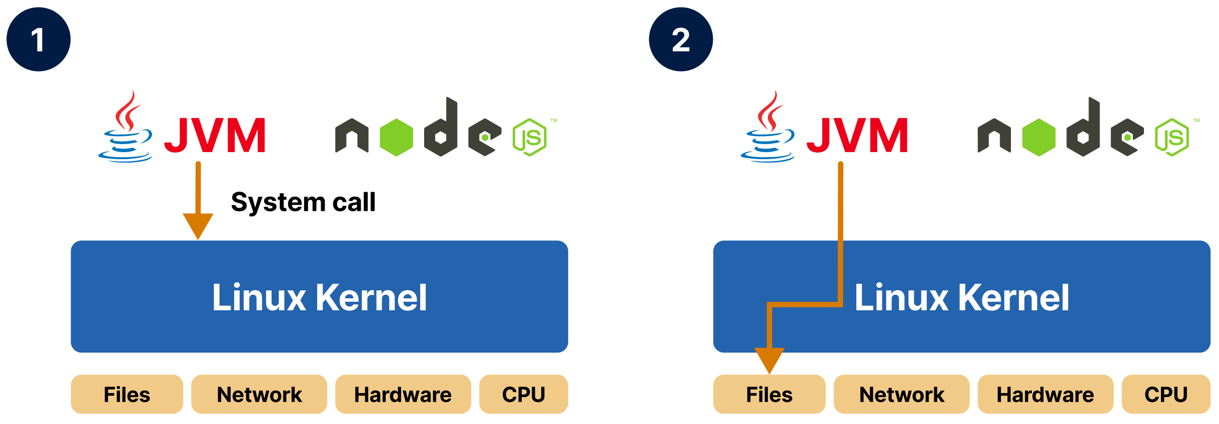


Fig. 1 The kernel works as a middle-man between your app and the hardware. When your Java application accesses a file on the filesystem, it initiates a syscall to the Linux kernel.

Fig. 2 The Kernel knows how to access the underlying storage and lets your app retrieve the files. You could think of syscalls as API calls.

The Linux kernel knows how to access the underlying storage and lets your app retrieve the files.

You could think of syscalls as API calls.

Similarly, if your Node.js app needs to initiate a network connection, it must do so through the Linux kernel with a system call.

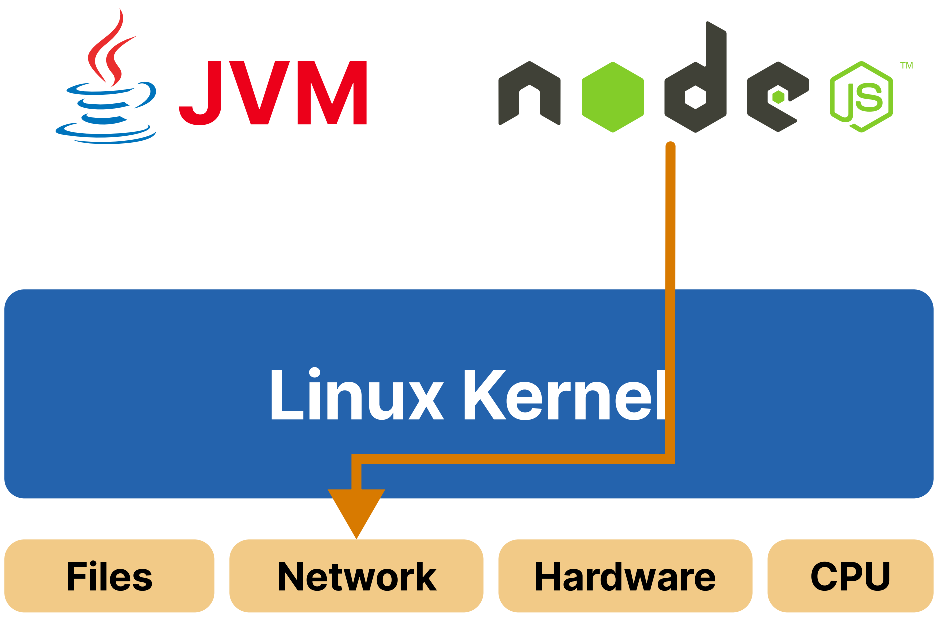


Fig. Network connection request through the Linux kernel to access network hardware

You can think of system calls as the **Linux kernel's API**—over 300 entry points grouped by what they do.

Some handle files, some manage processes, and others deal with networking or memory.

Each has a unique ID in the kernel and takes specific arguments to tell the kernel exactly what to do.

Even high-level code, like a simple network request in Node.js, still triggers a `socket` syscall behind the scenes.

This level of access gives user-space applications immense power.

However, it also exposes a large attack surface, especially when privileged capabilities are involved.

That's why Linux adds another layer to contain that risk: control groups, or cgroups.

Control Groups (cgroups): Enforcing Resource Limits

Control groups are kernel-level mechanisms that restrict the amount of CPU, memory, I/O, and other resources a process or a group of processes can consume.

Say you're running a JVM application.

You can assign it to a cgroup that caps its memory at 256MB and restricts it to a single CPU core.

Now take a second process, maybe a Node.js app. You can place it in a separate cgroup with its own constraints.

Each cgroup defines a **resource boundary**.

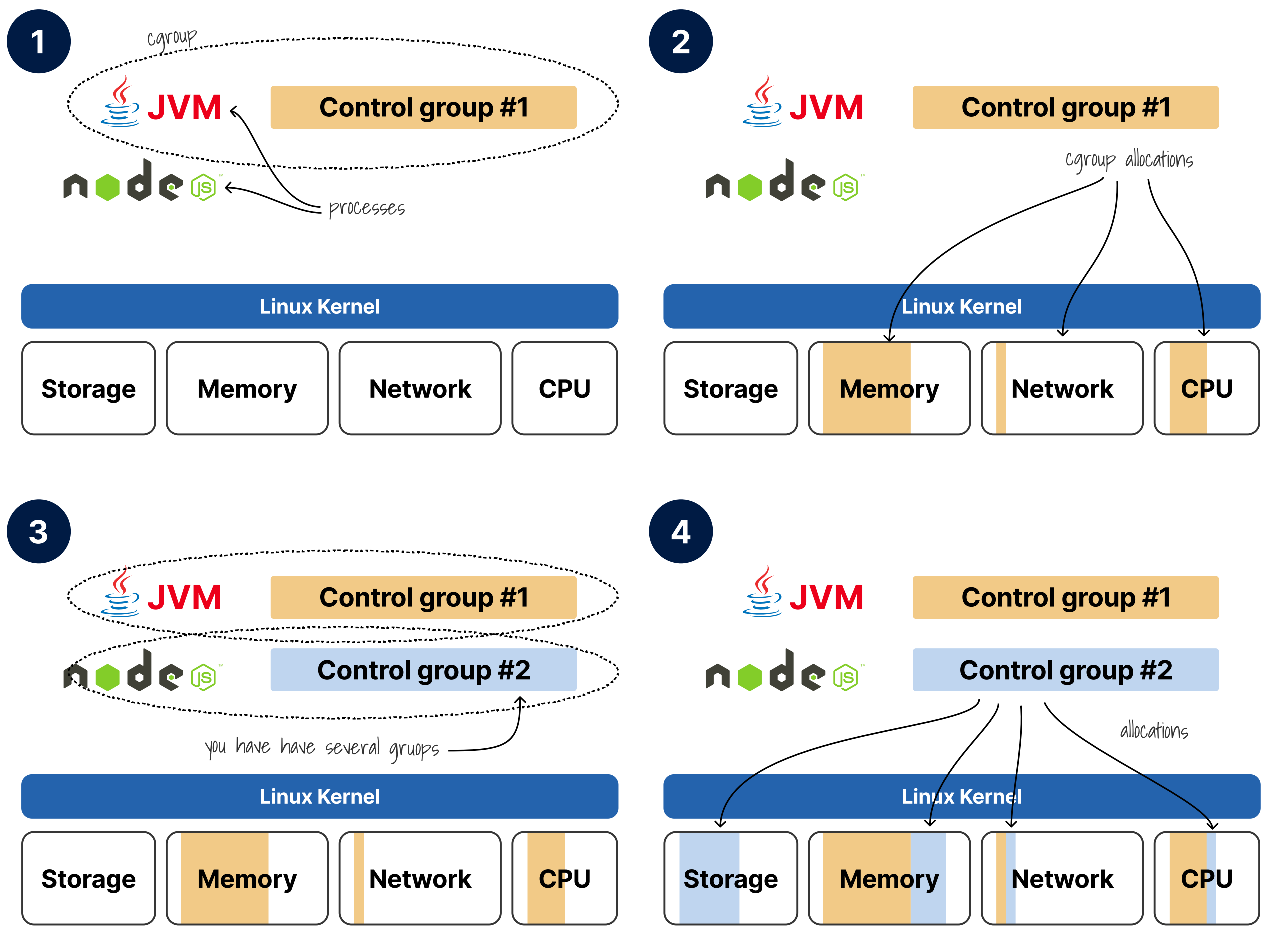


Fig. 1 In this case, I have a control group for the JVM.

Fig. 2 I can create a control group that limits access to CPU, memory, network bandwidth, etc.

Fig. 3 Each process can have its control group. I could create a second control group for the Node.js app.

Fig. 4 I can fine-tune the settings for the new control group and further restrict the available resources for that process.

You control how much each process gets, and the Linux kernel enforces it. When you set limits in cgroups:

bash

```
# CPU limit - 50% of one core
$ echo 50000 > /sys/fs/cgroup/cpu/myapp/cpu.cfs_quota_us
$ echo 100000 > /sys/fs/cgroup/cpu/myapp/cpu.cfs_period_us

# Memory limit - 256MB
$ echo 268435456 > /sys/fs/cgroup/memory/myapp/memory.limit_in_bytes
```

This lets you isolate workloads by physical limits on what they can consume.

But while cgroups enforce resource boundaries, they don't isolate what a process can see or interact with.

For that, Linux relies on another primitive: **namespaces**.

Namespaces: Isolating What a Process Can See

While cgroups control how much a process can consume, **namespaces** control what a process can see.

They determine which part of the system a process believes it's running in.

For example, with **network namespaces**, a process only sees its interfaces and traffic.

It can't see packets or sockets from outside its namespace.

With a **mount namespace**, a process sees a private filesystem view.

It might believe it sees `/etc`, `/var`, and `/home`, but it's only accessing a container-specific overlay.

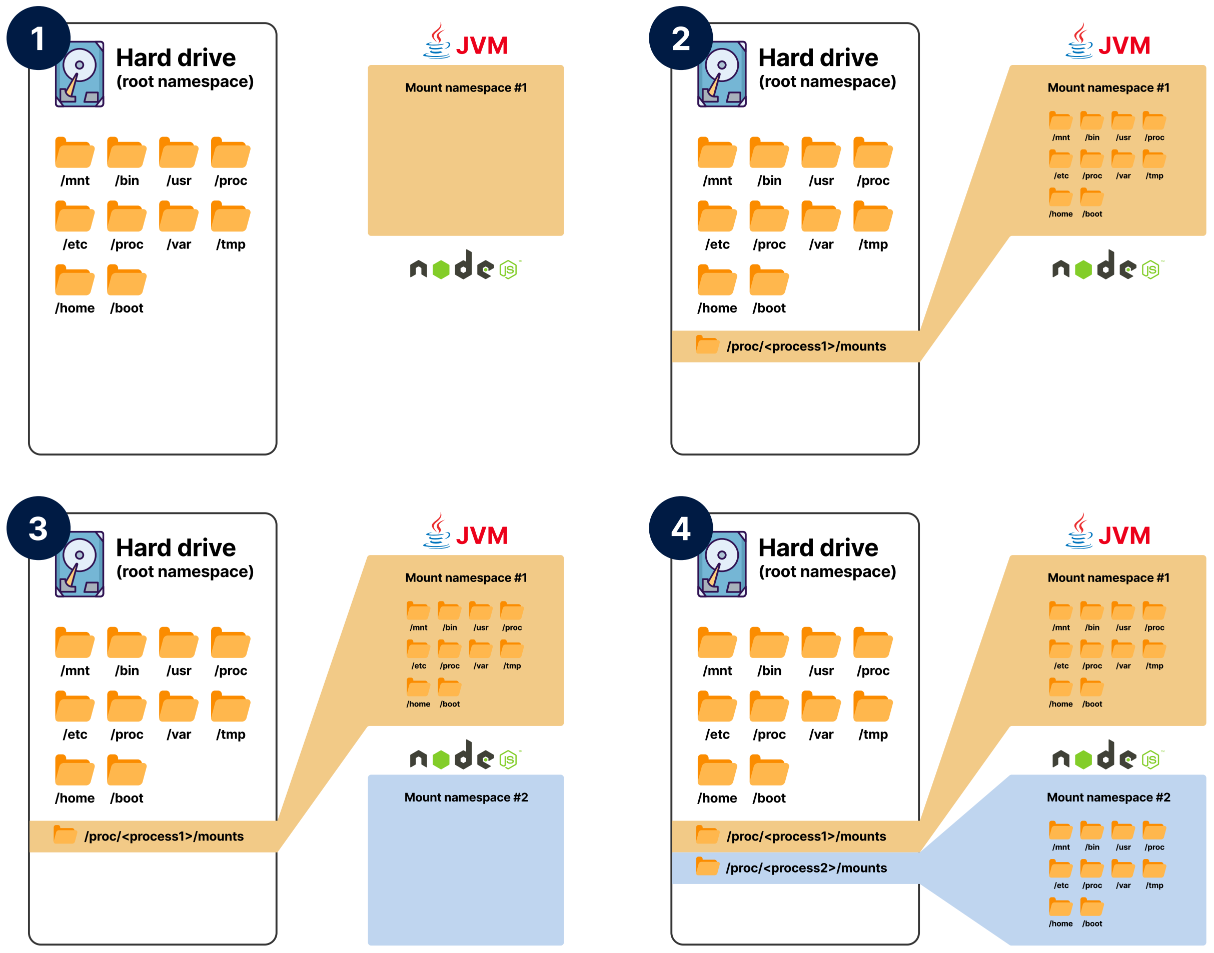


Fig. 1 Since kernel version 5.6, there are eight kinds of namespaces and the mount namespace is one of them.

Fig. 2 With the mount namespace, you can let the process believe it has access to all directories on the host, when in fact it does not.

Fig. 3 The mount namespace is designed to isolate resources — in this case, the filesystem.

Fig. 4 Each process can see the same filesystem, while still being isolated from the others.

As of Linux kernel 5.6, there are eight namespace types.

Each isolates a distinct dimension of system identity.

That means you can run multiple processes on the same machine, each in its own isolated environment, each with resource limits, and each believing it's alone.

Together with cgroups, namespaces form the backbone of container isolation.

But manually creating cgroups and namespaces is verbose and error-prone.

This is where tools such as Docker come in.

From Kernel Primitives to Docker

Docker is a popular way to manage containers in a developer-friendly way.

Instead of manually managing syscalls, cgroups, and namespaces, you define a container image, run a single command, and the runtime sets everything up behind the scenes.

Docker is not the only way to run containers, though.

Other tools abstract control groups and namespaces, such as Podman and CRI-O.

At their core, these platforms are just orchestration layers on top of three key Linux kernel features: **syscalls**, **control groups**, and **namespaces**.

Combined, these primitives give you secure, lightweight, isolated environments —**containers**.

Before introducing GPUs, let's understand how the kernel manages CPU and memory.

CPU: Preemptive Multitasking

The Linux kernel handles CPU scheduling through preemptive multitasking.

It rapidly interrupts running programs to divide execution time fairly and keep the system responsive.

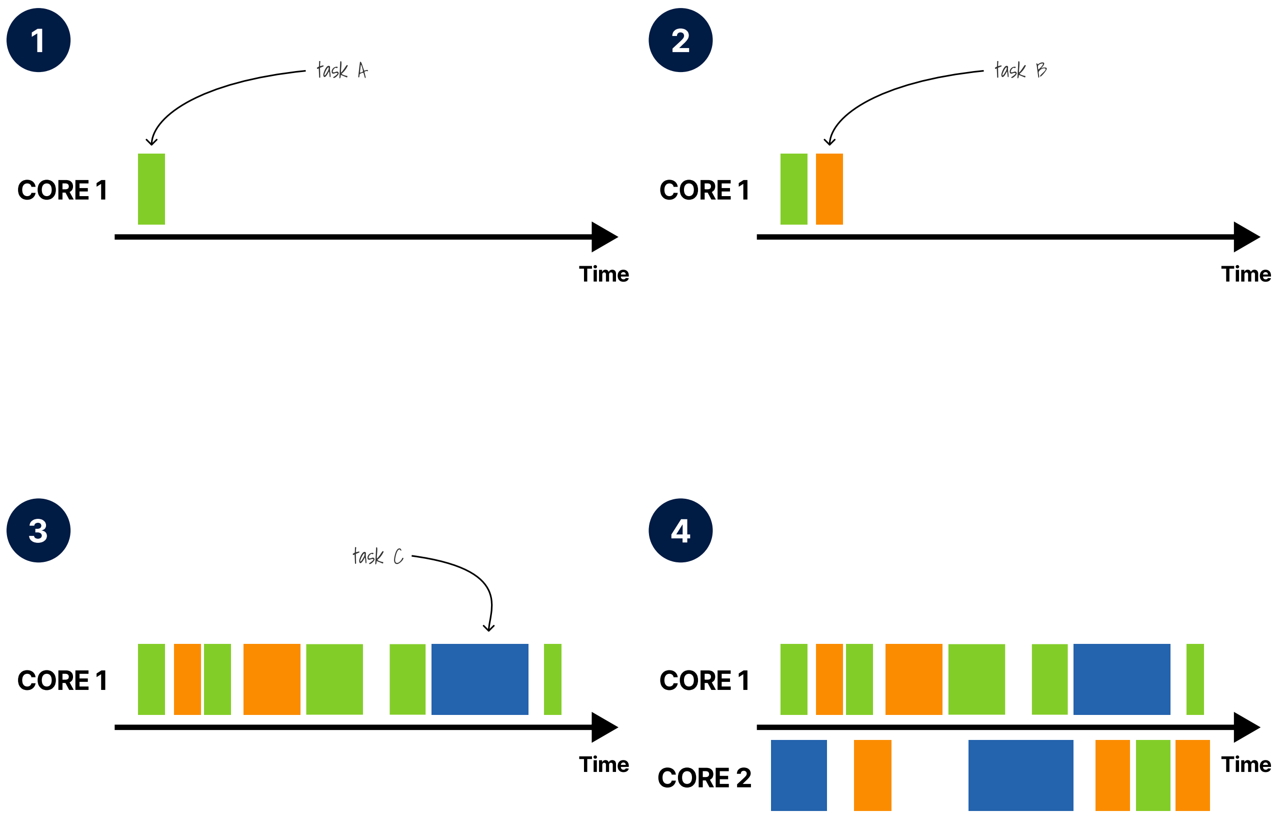


Fig. 1 In single-core preemptive multitasking, the CPU executes one task at a time.

Fig. 2 The scheduler preempts Task A and switches context to Task B (orange). Each task gets a quantum of CPU time.

Fig. 3 As more tasks are added, the scheduler continues rotating between them. Task C (blue) now joins the queue.

Fig. 4 With multi-core processors, true parallel execution becomes possible. Tasks can run simultaneously on different cores.

When the kernel switches tasks, it performs a **context switch**:

1. It saves the complete state of the running program, including its registers, stack, and memory mappings, so it can resume later exactly where it left off.
2. Then, it loads the saved state of the next program and resumes it as if it had never paused.

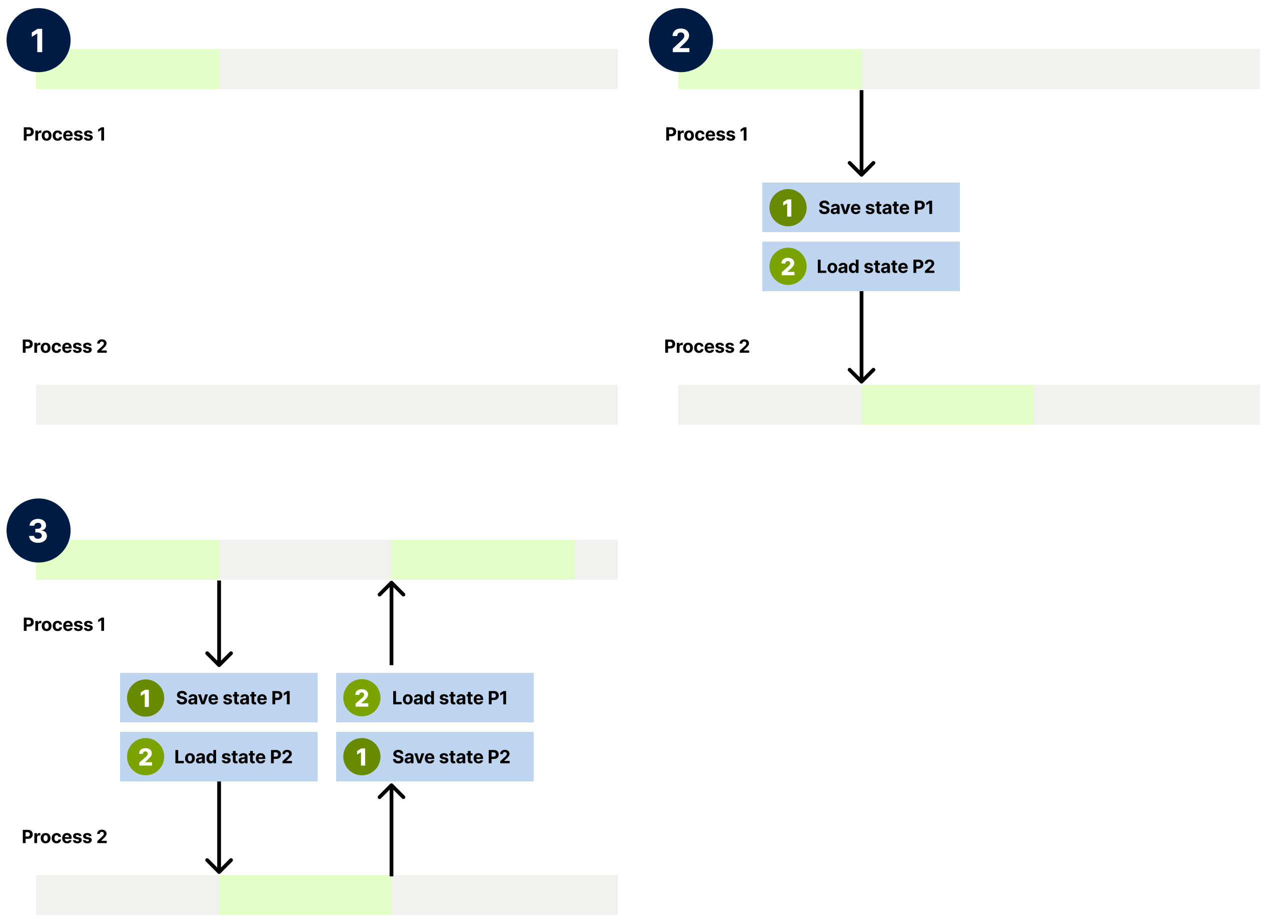


Fig. 1 Process 1 (top) is actively executing while Process 2 (bottom) is idle, waiting for its turn to execute.

Fig. 2 Process 1 saves its current state (step 1), then Process 2 loads its previously saved state (step 2) to resume execution.

Fig. 3 Process 2 first saves its state (step 1), then Process 1 loads its saved state (step 2) to continue where it left off.

This is fast and reliable on CPUs because the hardware was designed for it. The CPU has:

- Interrupt controllers to pause execution
- Hardware support for saving/restoring state
- Complex control logic for scheduling
- Sophisticated caching to make switches fast

A context switch on modern CPUs takes microseconds and gives the illusion that two processes are running simultaneously.

Memory: Page-by-Page Allocation

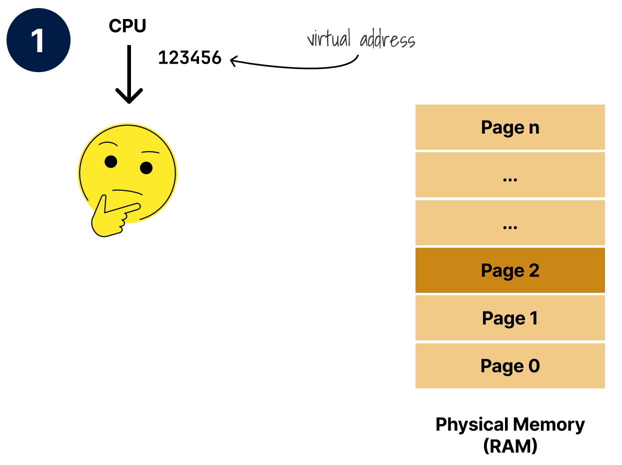
Memory allocation happens through the kernel's memory management unit (MMU).
When a process requests memory:

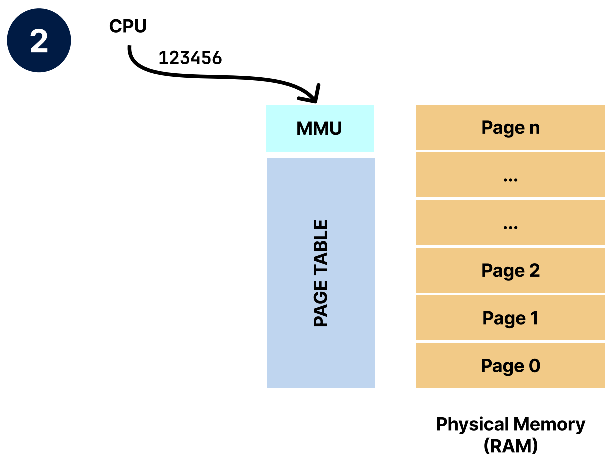
```
void *ptr = malloc(1024);  // Request 1KB
```

The Linux kernel:

1. Finds free pages in physical memory
2. Maps them into the process's virtual address space

3. Returns a pointer to the virtual address
4. Tracks every allocation in page tables





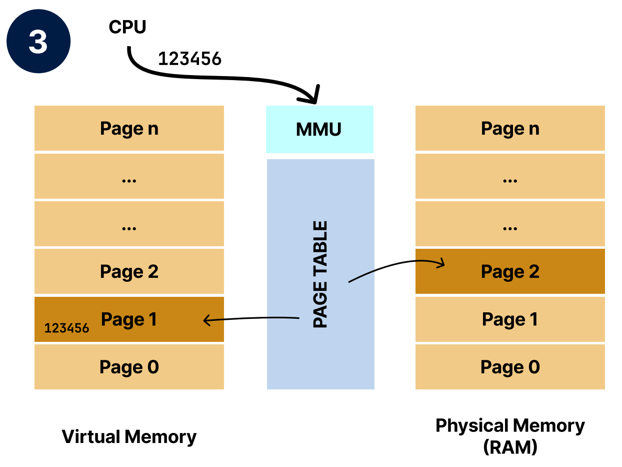


Fig. 1 CPU attempts to access virtual address 123456, but without an MMU, it's doesn't know how to locate the data in physical memory.

Fig. 2 The CPU sends virtual address 123456 to the MMU, which contains a page table for address translation.

Fig. 3 The MMU successfully translates virtual address 123456 to physical address using the page table.

The process sees a contiguous block of memory, but it might be scattered

physically across RAM.
The kernel tracks every byte, recovers unused pages, swaps to disk, and enforces limits through cgroups.
If a process exceeds its memory cgroup limit:

```bash
$ echo 104857600 > /sys/fs/cgroup/memory/myapp/memory.limit_in_bytes  # 100MB limit
```

The kernel's out-of-memory killer terminates it immediately.
Most of what we discussed so far might not be news to you:

- **Syscalls** provide controlled access to kernel functionality
- **Cgroups** enforce hard resource limits with kernel backing
- **Namespaces** create isolation boundaries for what processes can see
- **CPU context switching** happens in microseconds, allowing fair time-sharing
- **Memory** is allocated page-by-page with full kernel visibility and control

This model works beautifully: The Linux kernel sees everything, controls everything, and can enforce limits.
But it's important to recap how this works because GPUs work differently.

CUDA Fundamentals: Understanding Contexts, Kernels, and Memory

GPU applications are still regular CPU programs running on your processor like any other application.
However, they must communicate with the GPU when they need to perform parallel computation.

This communication requires managing a lot of state:

- Which GPU to use (if multiple are available)
- Memory allocations on the GPU
- Compiled GPU code (kernels)
- Synchronization between CPU and GPU
- Configuration and settings

This state gets bundled into what CUDA calls a "context."
A CUDA context is like opening a terminal session to the GPU.
When a process wants to use the GPU, it creates a context that contains:

- Memory allocations specific to this process
- Loaded kernel code (GPU programs)
- GPU state (configuration, settings)
- Execution streams and events

For the GPU to process data, that data must exist in GPU memory within the context.
It doesn't magically appear there: you have to copy it explicitly from the host ram to the GPU ram.

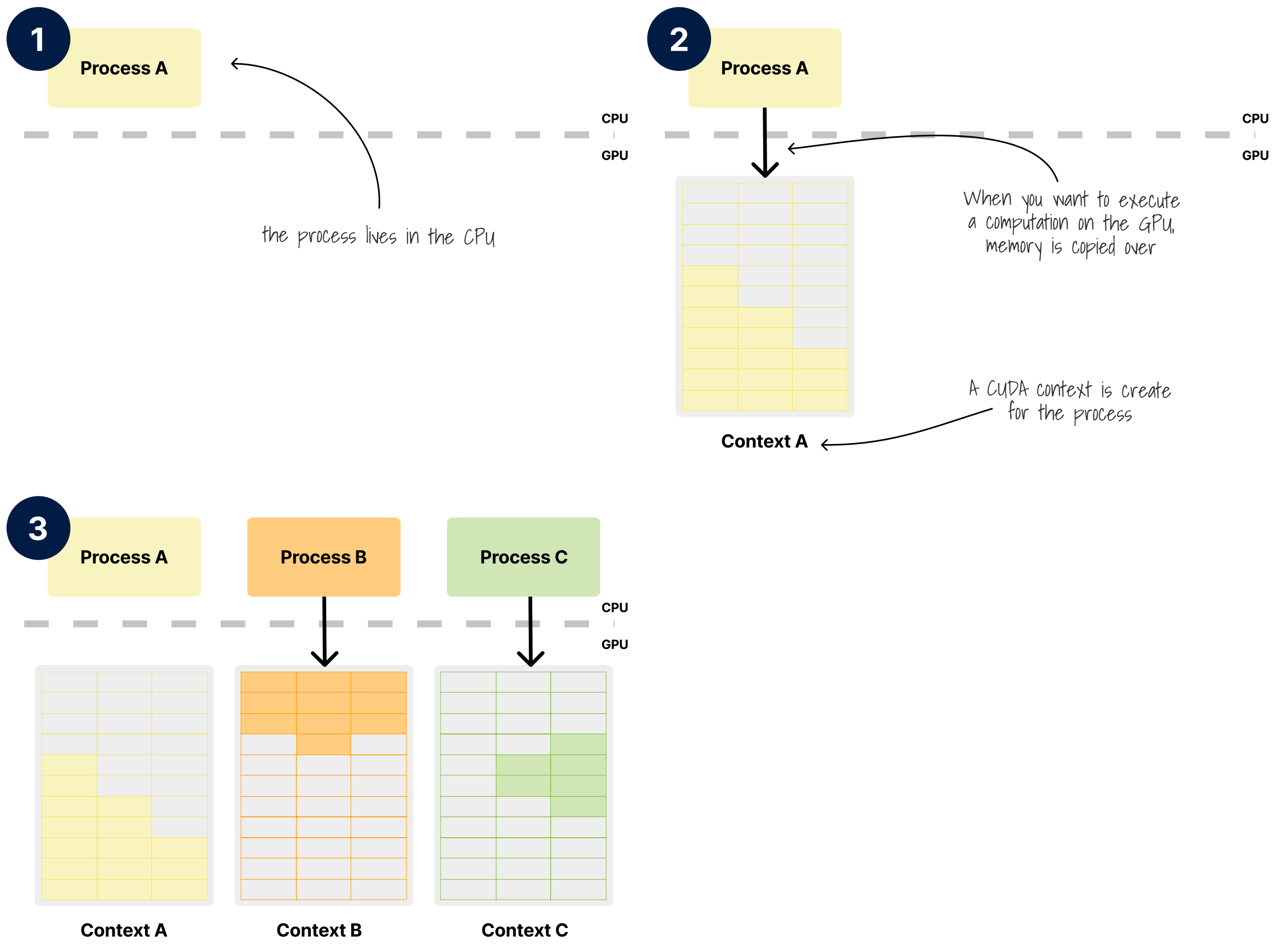

1
Process A
CPU
GPU
the process lives in the CPU
2
Process A
CPU
GPU
When you want to execute a computation on the GPU, memory is copied over
A CUDA context is create for the process
Context A
3
Process A
Process B
Process C
CPU
GPU
Context A
Context B
Context C

Fig. 1 Process A running on the CPU needs to execute computations on the GPU. The process exists in the CPU domain but requires GPU resources for parallel processing.

Fig. 2 When Process A wants to execute on the GPU, the memory is copied over and a GPU context (Context A) is created for the process.

Fig. 3 The GPU can manage multiple contexts, allowing different processes to utilize GPU resources through context switching.

Let's look at an example:

multiplication.py

```python
import cupy as cp

# This data lives in system RAM
cpu_data = [1, 2, 3, 4, 5]

# This copies the data to GPU memory
gpu_data = cp.array(cpu_data)  # Data transfer happens here

# Now the GPU can work with it
result = gpu_data * 2  # Computation happens on GPU

# To get results back, another copy
cpu_result = result.get()  # Transfer back to system RAM
```

Every time data moves between the CPU and GPU, typically it takes extra time because it has to travel across the physical connection that links them.
This is why GPU programming often focuses on minimizing data movement.

CUDA Kernels: The GPU Programs

When we talk about kernels, it's easy to get confused.
In the GPU world, a kernel is not the Linux kernel - it's a function that runs on the GPU.
When you write CUDA code:

add.py

```python
# This Python code runs on CPU
import cupy as cp
x = cp.array([1, 2, 3])
y = cp.array([4, 5, 6])

# This triggers a kernel launch on the GPU
z = x + y  # The addition happens on GPU via a kernel
```

That addition operation launches a pre-written kernel that:

1. Loads from the context's kernel cache
2. Executes on thousands of GPU cores in parallel
3. Runs to completion without interruption
4. Returns results to CPU

Now let's observe what happens when a context has more than one kernel:

add.py

```python
# This Python code runs on CPU
import cupy as cp
x = cp.array([1, 2, 3])
y = cp.array([4, 5, 6])

# This triggers a kernel launch on the GPU
z = x + y  # Kernel 1: addition kernel executes

# This triggers ANOTHER kernel launch
w = x - y  # Kernel 2: subtraction kernel executes
```

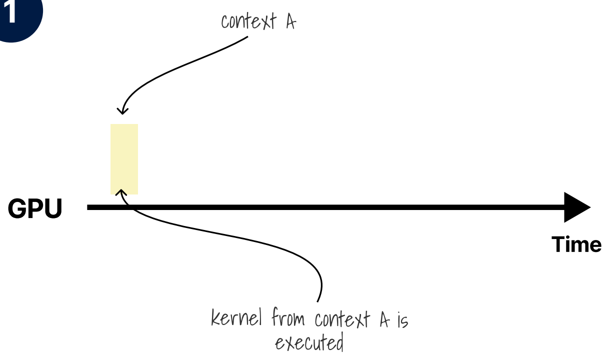
Each operation (`+` , `-` , `*` , etc.) launches a separate CUDA kernel.
They execute sequentially (one after the other) within the same CUDA context created when the Python process initializes CUDA/CuPy.
The context persists for the lifetime of your Python process.
It's like a session that holds:

- All your GPU memory allocations (`x` , `y` , `z` , `w` arrays)
- The compiled kernels (addition, subtraction)

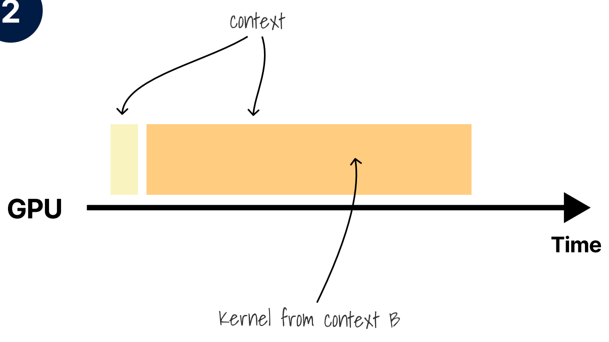
- The execution state

And, unlike CPUs, once a kernel starts, it monopolizes the GPU until it finishes.

1



2



3

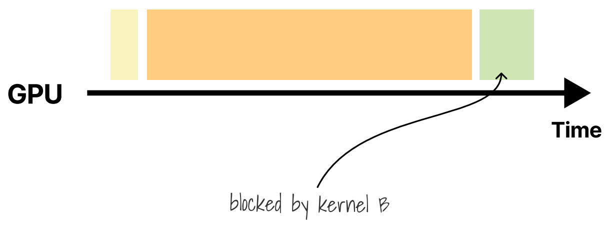


Fig. 1 Context A is actively executing a kernel on the GPU.

Fig. 2 Context switching occurs: Context A continues execution, then a kernel from Context B (orange) is scheduled and executed.

Fig. 3 Context A (yellow), Context B (orange), and Context C (green) take turns executing their kernels.

This is somewhat puzzling if you're used to CPU scheduling.

Through time-slicing, we can have hundreds of processes running simultaneously on a CPU; the kernel rapidly switches between them, giving each a few milliseconds of execution time.

But on a GPU, kernels from different processes cannot run simultaneously (there is a notable example, MPS, which we will address later).

This is serialization at the context level.

Each process creates its own CUDA context, and only one context can be active on the GPU at a time.

Kernels from different contexts are serialized: they don't truly run at the same time.

Within a single CUDA context, you can create multiple **streams** - independent queues for GPU operations.

Here's a simple example:

streams.py

```python
import cupy as cp

# Without streams - sequential execution
A = cp.random.random((4096, 4096))
B = cp.random.random((4096, 4096))
C = cp.random.random((4096, 4096))
D = cp.random.random((4096, 4096))

# These execute one after another
result1 = A + B  # Kernel 1 executes
result2 = C + D  # Kernel 2 waits for Kernel 1 to complete

# With streams - concurrent execution
stream1 = cp.cuda.Stream()
stream2 = cp.cuda.Stream()

with stream1:
    result1 = A + B  # Kernel 1 in stream1

with stream2:\
    result2 = C + D  # Kernel 2 in stream2 - runs CONCURRENTLY with Kernel 1!
```

This concurrent execution is made possible by **Hyper-Q**, a hardware feature in modern GPUs that provides multiple hardware work queues (32 in Kepler and later).

Without Hyper-Q, even kernels in different streams would often serialize due to hardware queue limitations.

But even with streams (and MPS), individual kernels still can't be preempted. *Why don't GPUs simply pause and resume kernels the way CPUs do?*

Why GPUs Don't Support Kernel Preemption

We must look at the hardware architecture and ALUs (Arithmetic Logic Units) to understand why GPUs can't pause and resume kernels.

ALUs are the circuits that do math—add, multiply, compare numbers.

Both CPUs and GPUs need them to compute, but they use them differently.

Your CPU runs everything: web browsers, databases, and operating systems.

This code contains branches (if-then-else), unpredictable memory access, and complex dependencies.

To keep even 2-4 ALUs busy, the CPU needs massive amounts of control logic to:

- Predict which branch the code will take
- Reorder instructions to avoid stalls
- Handle cache misses when data isn't ready
- Manage interrupts and context switches

Here's why CPUs can't just add more ALUs: keeping ALUs fed with practical work requires control logic that grows exponentially with ALU count.

With 2-4 ALUs, a CPU already dedicates about half its transistors to control—branch predictors, instruction decoders, reorder buffers, and caches.

To keep 8 ALUs busy on general-purpose code, you'd need to predict twice as many branches, buffer twice as many instructions, and handle twice as many cache misses.

The control logic would balloon.

That's the fundamental limit: CPUs handle unpredictable code and need sophisticated control logic that doesn't scale.

Instead, GPUs only run predictable, parallel code.

When you multiply two matrices or process pixels, you do the same operation on thousands of data points.

No branches, no unpredictable memory access.

This changes everything.

Since all 128 ALUs in a GPU streaming multiprocessor do the same operation (just on different data), you only need one control unit to feed them all.

The instruction is broadcast to all ALUs at once.

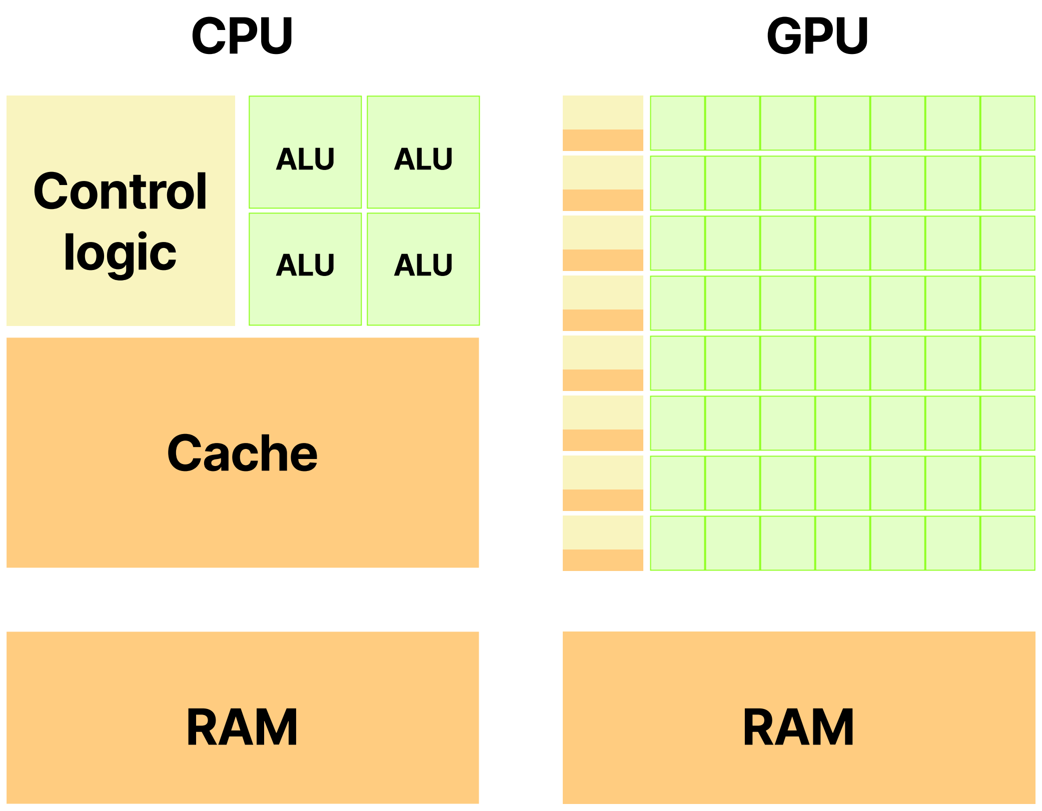


Fig. Architectural comparison between CPU and GPU

This is why a CPU core might have 2-4 ALUs while a GPU streaming multiprocessor has 64-128.

The GPU dedicates almost all its transistors to computation rather than control.

But this design comes with a cost: there are no interrupts, no preemption, and no ability to pause and resume.

Adding these features would require the complex control logic GPUs specifically gave up to pack in more ALUs.

GPU Memory: A Completely Different Model

Now that we understand how GPU execution differs from CPU execution, let's examine how GPU memory breaks all our assumptions about memory management.

GPU memory operates completely differently from system memory, creating three distinct views that often confuse operators.

First, there's the **physical GPU memory**—the actual VRAM chips on the card.

Your Tesla T4 has 16GB, an A100 has 40GB or 80GB.

This is the absolute limit, like the amount of RAM in your server.

However, unlike system memory, where the kernel carefully tracks every page, GPU memory management happens entirely in the NVIDIA driver, invisible to the Linux kernel.

Something interesting happens when a process creates a CUDA context and starts allocating memory.

Each context can see the whole GPU memory space, e.g., all 16GB on that Tesla T4.

But it can only access its own allocations.

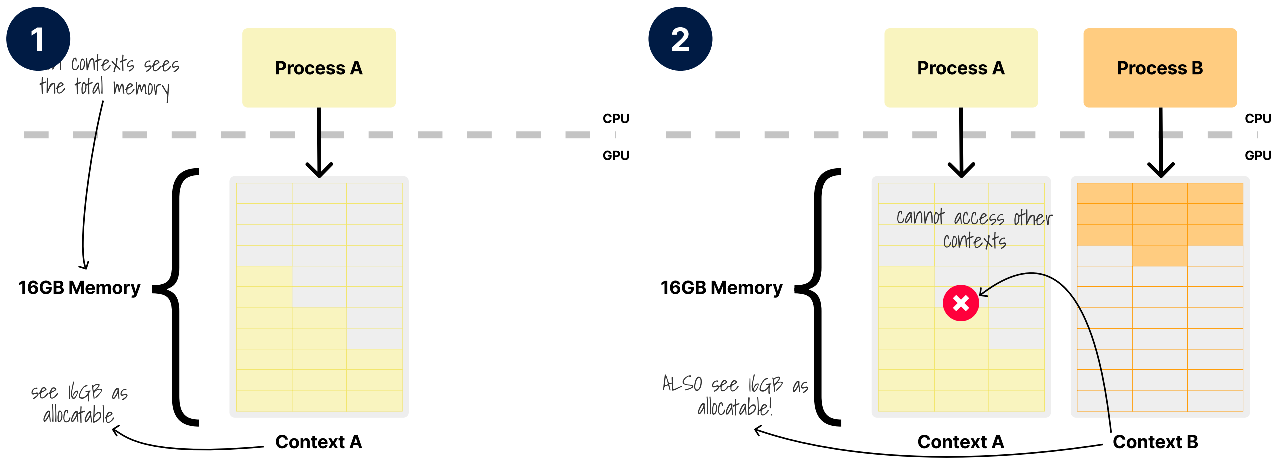


Fig. 1 Process A creates Context A on the GPU with 16GB total memory available. The context see the full 16GB as allocatable.

Fig. 2 Process B attempts to create Context B while Context A is still resident. Although both contexts see 16GB as allocatable, they cannot access other contexts' memory.

Let's look at an example:
Process A starts, creates Context A:

- Sees: 16GB total GPU memory
- Allocates: 4GB for neural network weights
- Can access: Only its own 4GB

Process B starts, creates Context B:

- Sees: 16GB total GPU memory (same view as A!)
- Allocates: 2GB for image processing
- Can access: Only its own 2GB

In reality, 7GB of memory is used and 9GB is free, but each process thinks there's 16GB available.
This is already confusing: two processes, each seeing 16GB, but together using 7GB.
The Linux kernel has no visibility into this.

There's no `/proc/meminfo` for GPU memory, no cgroup accounting, no way to enforce limits.

But it gets worse.

CUDA doesn't actually allocate memory the way you request it.

When you ask for memory, CUDA uses a pooling allocator for efficiency:

memory.py

```python
import cupy as cp

# First allocation - you ask for 8KB
arr1 = cp.zeros(1000)  # 1000 floats × 8 bytes = 8KB

# But CUDA actually reserves a 2MB pool from the GPU
# nvidia-smi now shows 2MB used, not 8KB

# Second allocation
arr2 = cp.zeros(1000)  # Another 8KB

# This uses the existing pool - no new GPU allocation
# nvidia-smi still shows 2MB used

# Much later...
huge_array = cp.zeros(10_000_000)  # 80MB needed

# Now CUDA reserves another big chunk, maybe 128MB
# nvidia-smi shows 130MB used (2MB + 128MB)
```

This pooling behavior makes sense for performance—allocating GPU memory is expensive, so CUDA grabs and manages chunks internally.

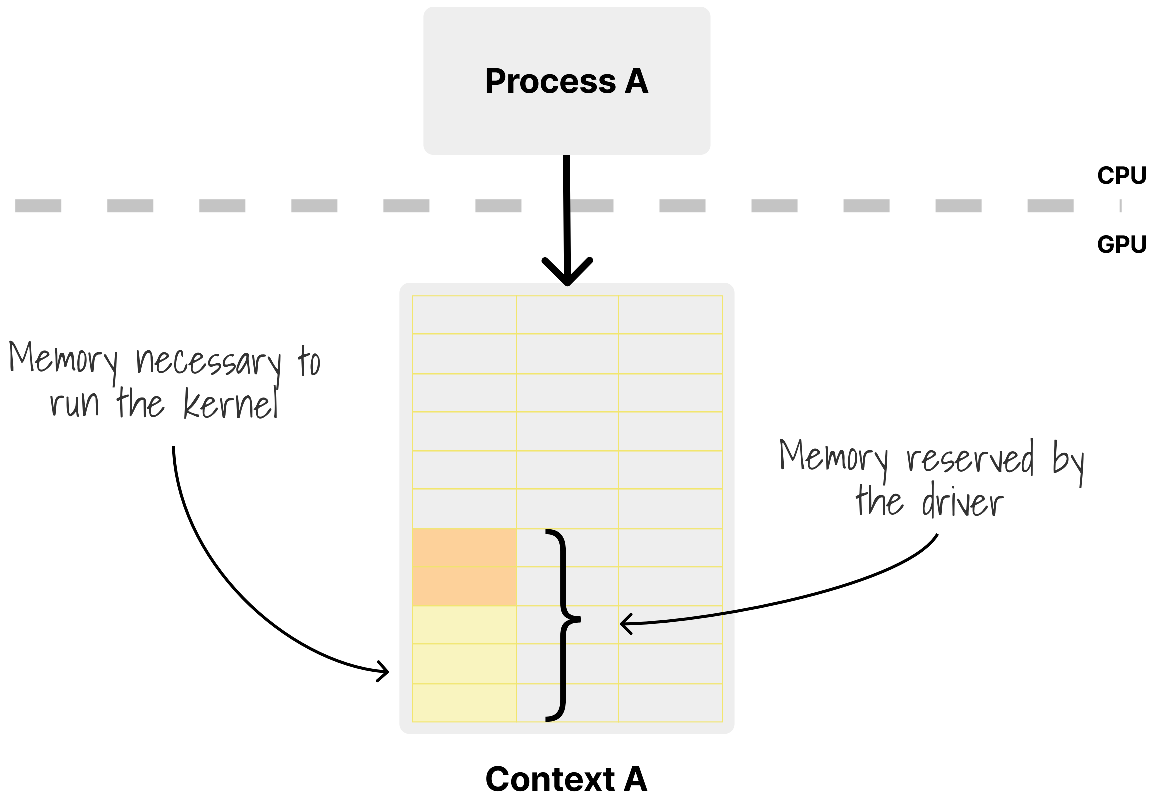

Process A
CPU
GPU
Memory necessary to run the kernel
Memory reserved by the driver
Context A

Fig. GPU memory pooling diagram showing how CUDA allocates memory in large chunks (pools) rather than exact requested sizes

But it makes monitoring and capacity planning nearly impossible.
You might see 2GB allocated in `nvidia-smi` when your application only uses 100MB of data.

The Critical Differences

Let's make the contrasts explicit:

When your CPU code runs:

- Every operation goes through syscalls to the Linux kernel
- The Linux kernel can preempt your process at any microsecond
- Context switches between processes take microseconds
- The Linux kernel sees every instruction, every memory access

When your GPU code runs:

- CUDA API calls go to the NVIDIA driver (proprietary, closed-source)
- Kernels run to completion, and no preemption is possible
- Context switches take milliseconds (1000x slower than CPU)
- The Linux kernel is utterly blind to what happens on the GPU

When the Linux kernel manages system memory:

- Allocated page by page (usually 4KB pages)
- Every allocation is tracked in page tables
- Cgroups can enforce hard limits
- Can be swapped to disk, compressed, reclaimed

In contrast, GPU memory is managed by the driver:

- Allocated in large pools (megabytes at a time)
- Driver tracks allocations, and the Linux kernel can't see them
- No cgroup enforcement possible
- The memory cannot be swapped or reclaimed

The kernel primitives we rely on for containers simply don't exist for GPUs:
There are no GPU cgroups:

bash

```bash
# This doesn't exist
echo 4G > /sys/fs/cgroup/gpu/myapp/gpu.memory_limit
```

The Linux kernel doesn't know how to limit GPU memory because it can't see GPU allocations.

There are no GPU namespaces: `/dev/nvidia0` is visible to everyone.

There's no way to hide GPUs from specific processes or create virtual GPU views.

The Linux kernel has no visibility.

When a process makes a CUDA call, the Linux kernel sees a system call (e.g., `ioctl()`) to the NVIDIA driver, and that's it.

What happens next is opaque: the Linux kernel can't track GPU utilization, can't see memory usage, can't enforce fairness.

This is the fundamental challenge: **containers rely on Linux kernel primitives that don't exist for GPUs**.

And this discrepancy transpires through Kubernetes, too.

Kubernetes and the Container Runtime Interface

Now that we understand the fundamentals of CPUs, GPUs, and containers, let's explore them in the context of Kubernetes.

When you deploy a pod, the kubelet on each node is responsible for running it.

The kubelet doesn't create containers directly; instead, it uses the Container Runtime Interface (CRI) to talk to a container runtime like containerd or CRI-O.

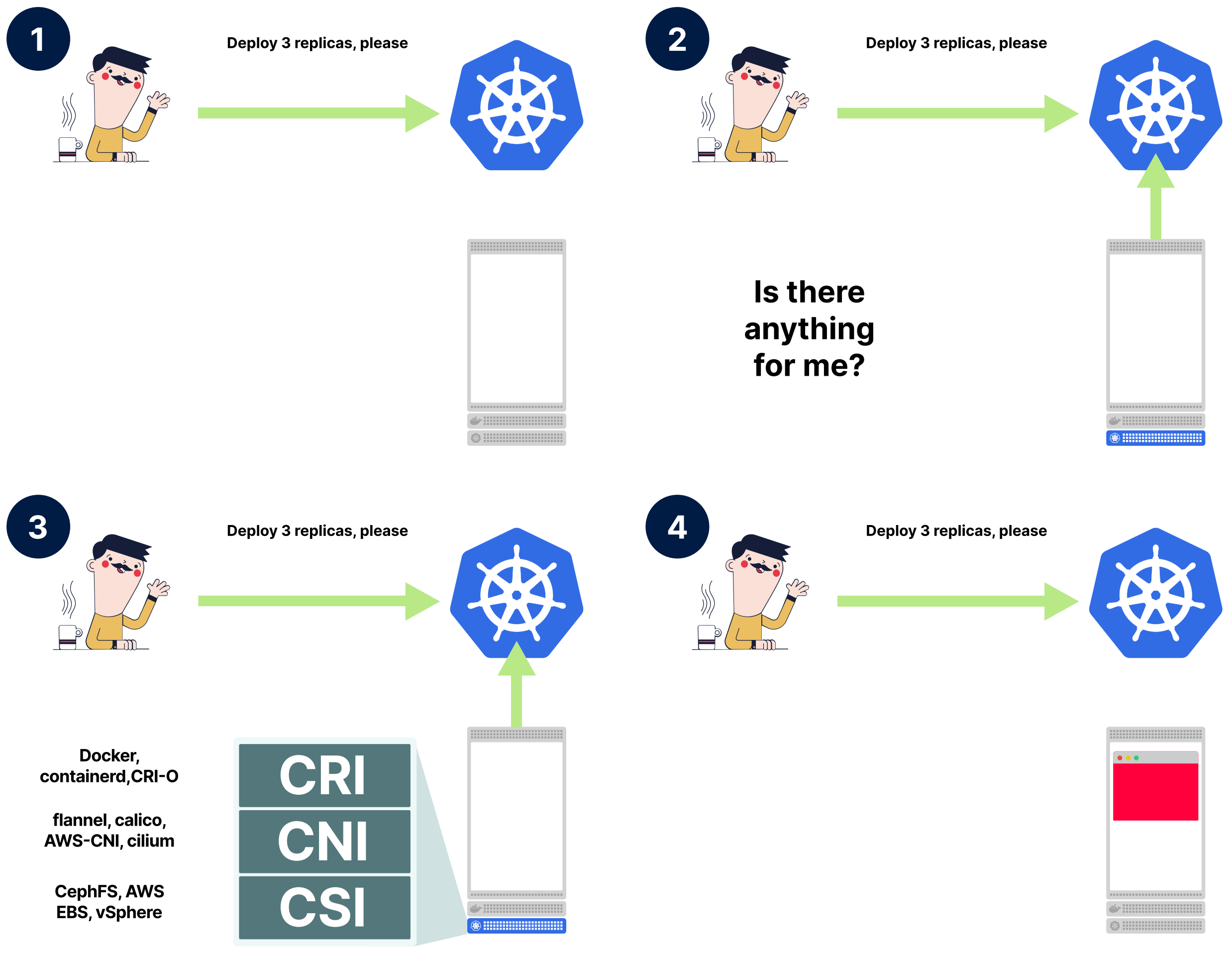

1
Deploy 3 replicas, please
2
Deploy 3 replicas, please
Is there anything for me?
3
Deploy 3 replicas, please
Docker, containerd,CRI-O
flannel, calico, AWS-CNI, cilium
CephFS, AWS EBS, vSphere
CRI
CNI
CSI
4
Deploy 3 replicas, please

Fig. 1 The deployment request is sent to the Kubernetes API (represented by the ship's wheel icon), with an empty node ready to receive workloads.

Fig. 2 The kubelet on the node polls the Kubernetes API. The API server responds with pod specifications that need to be scheduled on this node.

Fig. 3 The kubelet receives the pod specification and interacts with the CRI/CSI/CNI.

Fig. 4 The container (shown in pink/red) is now running on the node.

When a pod is scheduled to a node, here's what happens:

First, the kubelet receives the pod specification from the API server.

This spec describes what containers to run, what resources they need, and what volumes to mount.

The kubelet's job is to make this abstract specification real.

The kubelet calls the CRI to create a pod sandbox.

This sandbox is the shared environment for all containers in the pod.

The CRI creates a network namespace so all containers share the same network stack, sets up the initial hostname, and prepares shared volumes.

For each container in the pod, the kubelet orchestrates creation.

First, it checks if the image exists locally.

If not, it pulls the image from the registry and downloads the layers, extracts them, and prepares the filesystem.

The CRI translates resource requests into cgroups.

When your pod specification says:

resources.yaml

```yaml
resources:
  limits:
    cpu: '500m' # Half a CPU core
    memory: '256Mi' # 256 MiB of RAM
```

The CRI creates corresponding cgroups:

bash

```bash
# CPU cgroup - limit to 50% of one core
/sys/fs/cgroup/cpu/kubepods/pod-uuid/container-id/cpu.cfs_quota_us: 50000
/sys/fs/cgroup/cpu/kubepods/pod-uuid/container-id/cpu.cfs_period_us: 100000

# Memory cgroup - limit to 256MiB
/sys/fs/cgroup/memory/kubepods/pod-uuid/container-id/memory.limit_in_bytes: 268435456
```

The CRI sets up isolation through namespaces.

Each container gets its own mount namespace for filesystem isolation, its own PID namespace to see only its processes, and joins the pod's network namespace for shared networking.

Finally, the container process starts. The CRI executes the entrypoint, the process begins running, and the kernel enforces all the cgroups and namespace boundaries you've configured.

The CRI handles the translation from Kubernetes pod specs to actual Linux kernel primitives.

The GPU Problem in Kubernetes

But what about GPUs?

We face two fundamental problems:

1. **The scheduler needs to know which nodes have GPUs available.** But Kubernetes doesn't know what a GPU is since it only natively understands CPU and memory.
2. **Even if we schedule a pod to a GPU node, the container is isolated.** It can't see `/dev/nvidia0` or other GPU device files.

We need a way to:

1. Make GPU device files visible inside the container
2. Load the right libraries to talk to the GPU
3. Set up the CUDA context

To understand how GPUs are consumed in containers, we need to consider GPU integration as a four-layer stack, where each layer solves specific problems but introduces new complexities.

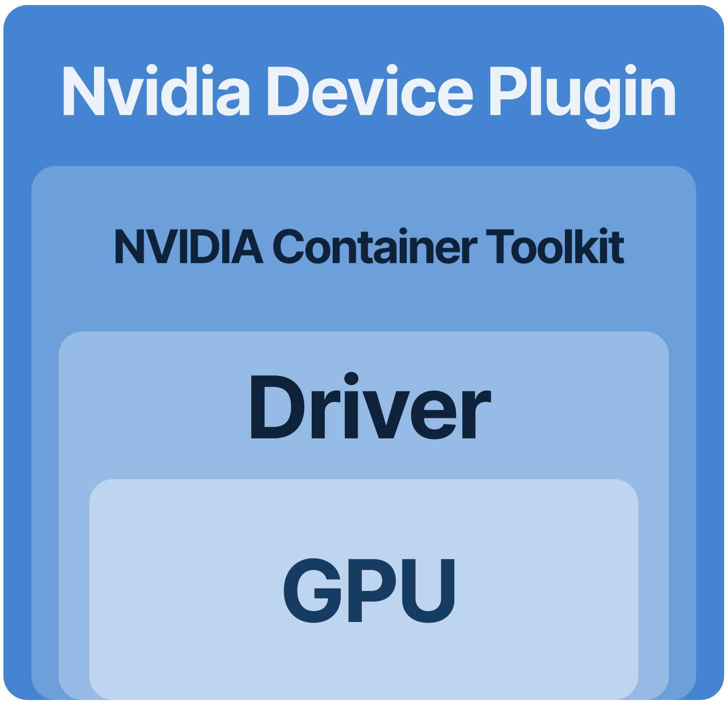


Fig. Four-layer GPU integration stack

At the foundation sits the GPU hardware and its driver.
Let's check what GPU hardware is available in our minikube environment:

bash

```
$ minikube ssh

$ nvidia-smi
Mon Aug  4 12:35:11 2025
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 550.54.15              Driver Version: 550.54.15      CUDA Version: 12.4     |
|-----------------------------------------+------------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  Tesla T4                       Off |   00000000:00:04.0 Off |                    0 |
| N/A   37C    P8              9W /   70W |       0MiB /  15360MiB |      0%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
|  No running processes found                                                           |
+---------------------------------------------------------------------------------------+

$ nvidia-container-runtime --version
NVIDIA Container Runtime version 1.13.5
commit: 6b8589dcb4dead72ab64f14a5912886e6165c079
spec: 1.1.0-rc.2

runc version 1.1.5+ds1
commit: 1.1.5+ds1-1+deb12u1
spec: 1.0.2-dev
go: go1.19.8
libseccomp: 2.5.4
```

Excellent!

We have a Tesla T4 with the full NVIDIA software stack ready and the kernel driver and container toolkit installed.

The NVIDIA driver is the bridge between the hardware and something the Linux kernel can understand:

bash

```
$ ls -la /dev/nvidia*
crw-rw-rw- 1 root root 195, 254 Aug  4 11:56 /dev/nvidia-modeset
crw-rw-rw- 1 root root 241,   0 Aug  4 11:40 /dev/nvidia-uvm
crw-rw-rw- 1 root root 241,   1 Aug  4 11:40 /dev/nvidia-uvm-tools
crw-rw-rw- 1 root root 195,   0 Aug  4 11:40 /dev/nvidia0
crw-rw-rw- 1 root root 195, 255 Aug  4 11:40 /dev/nvidiactl

/dev/nvidia-caps:
total 0
drwxr-xr-x  2 root root     80 Aug  4 11:40 .
drwxr-xr-x 15 root root   3240 Aug  4 11:56 ..
cr--------  1 root root 244, 1 Aug  4 11:40 nvidia-cap1
cr--r--r--  1 root root 244, 2 Aug  4 11:40 nvidia-cap2
```

`/dev/nvidia0` represents our Tesla T4, while `/dev/nvidiactl` handles control operations across all GPUs.

The kernel driver exposes the CUDA Driver API through these interfaces to give applications access to this processing power.

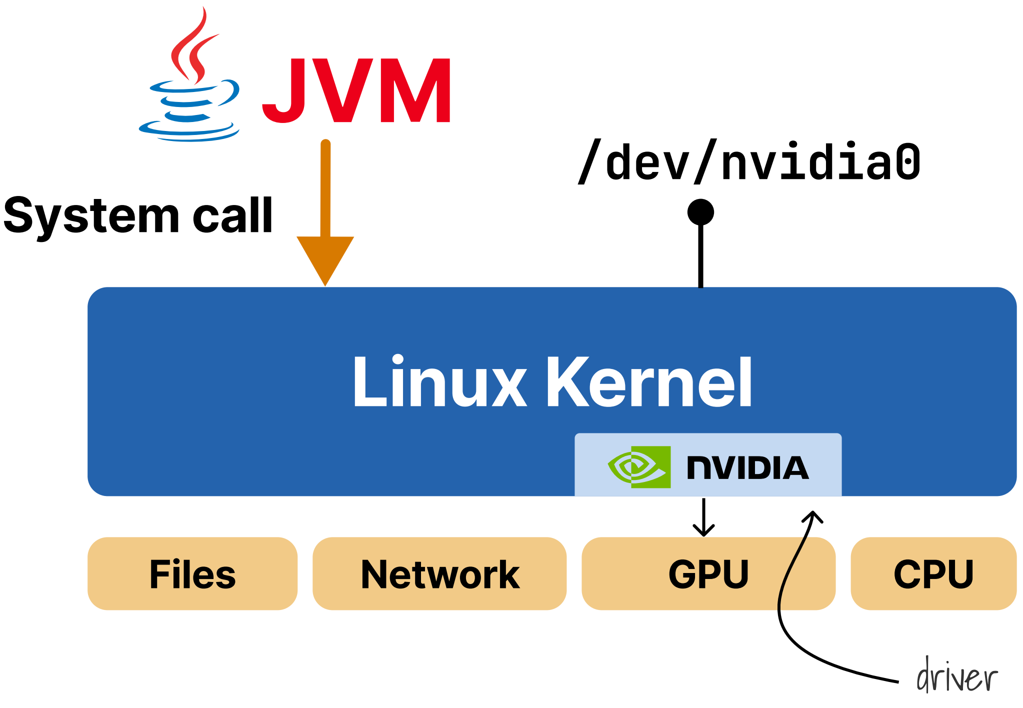


Fig. NVIDIA driver architecture diagram showing how the kernel driver exposes CUDA Driver API through device files

But here's where things get tricky.

Those device files live on the host, and container runtimes have no idea what a GPU is.

This is where the NVIDIA Container Toolkit steps in to solve the isolation puzzle.

When you launch a GPU workload, the container runtime must break its rules.

Containers are supposed to be isolated from the host, but GPUs require direct hardware access.

The NVIDIA Container Toolkit performs this controlled breach by intercepting container creation and surgically mounting exactly what's needed—the `/dev/nvidia0` device files, driver libraries, and environment variables like `NVIDIA_VISIBLE_DEVICES` .

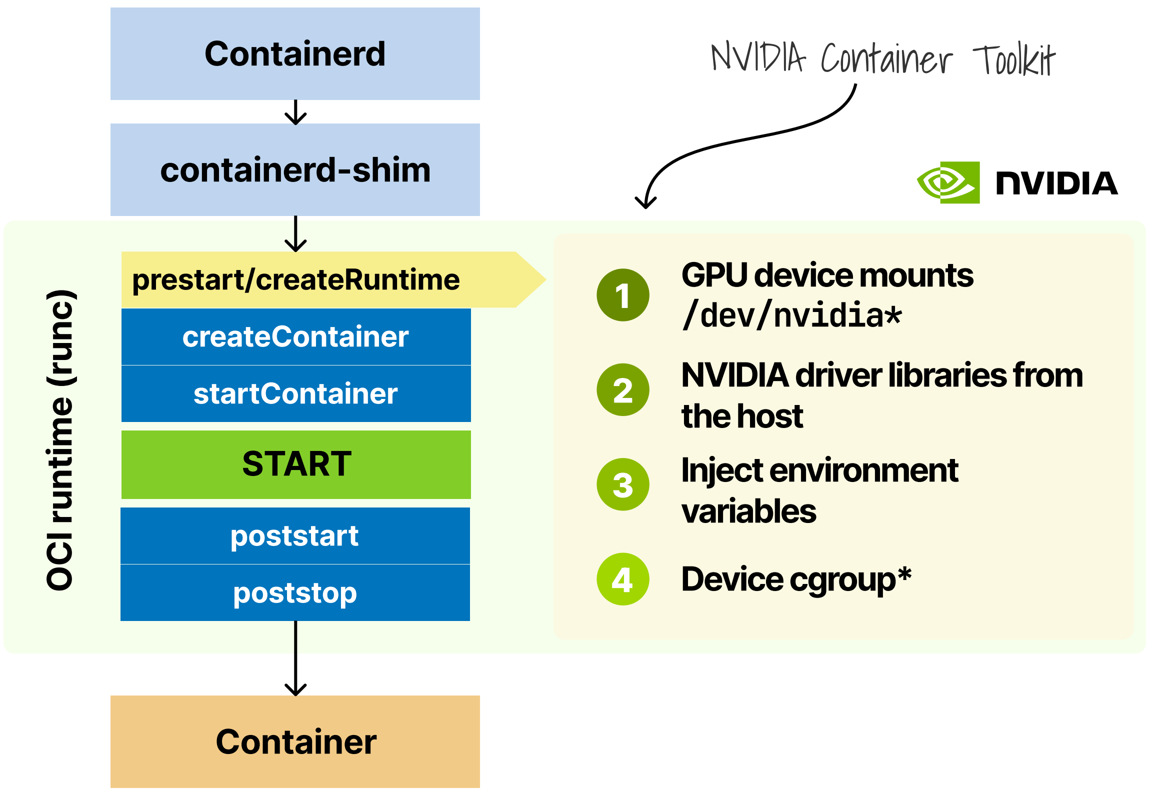


Fig. NVIDIA Container Toolkit workflow showing how it intercepts container creation to mount GPU device files, driver libraries, and set environment variables

Your application code never changes, but suddenly it can talk to that Tesla T4 sitting on the host.

The runtime handles this translation layer, but Kubernetes still doesn't know GPUs exist.

So, how does Kubernetes integrate non-native hardware devices (like NVIDIA GPUs) into its scheduling model?

Using the Device Plugin Framework.

Device plugins are Kubernetes' way of handling hardware, which it doesn't understand natively.

The NVIDIA device plugin running in your cluster implements a simple gRPC contract:

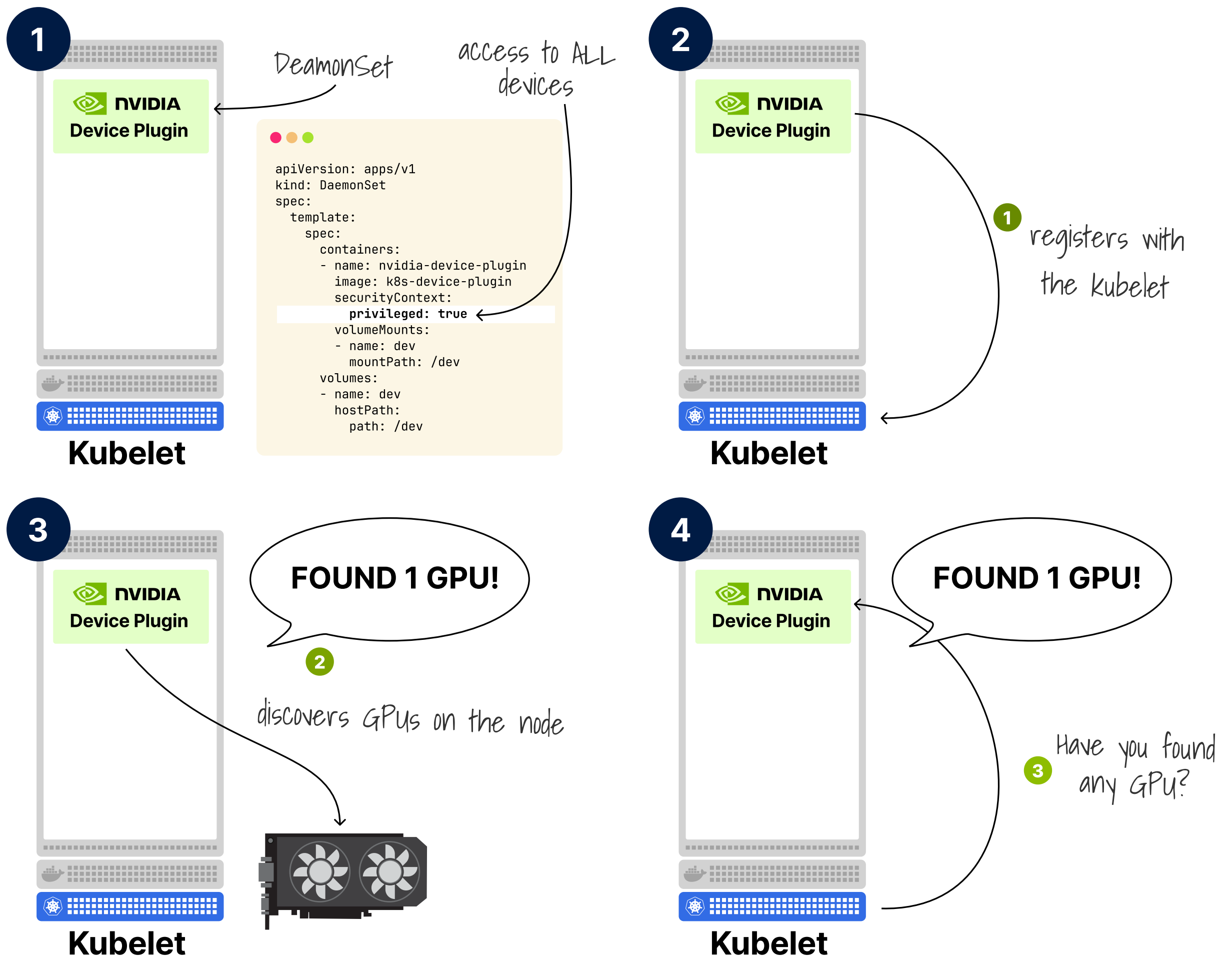
device-plugin-interface.go

```
// Device Plugin gRPC Service
service DevicePlugin {
    rpc ListAndWatch(Empty) returns (stream ListAndWatchResponse);
    rpc Allocate(AllocateRequest) returns (AllocateResponse);
}
```

This plugin acts as a translator between your Tesla T4 hardware and Kubernetes' resource model.

It discovers available GPUs through NVML library calls, then advertises them as schedulable resources.

After registration, the plugin advertises its hardware using the `ListAndWatch` stream.



```yaml
apiVersion: apps/v1
kind: DaemonSet
spec:
  template:
    spec:
      containers:
      - name: nvidia-device-plugin
        image: k8s-device-plugin
        securityContext:
          privileged: true
        volumeMounts:
        - name: dev
          mountPath: /dev
      volumes:
      - name: dev
        hostPath:
          path: /dev
```

Fig. 1 A DaemonSet deploys the NVIDIA Device Plugin to all nodes with GPU resources.

Fig. 2 The registration process establishes communication between the device plugin and kubelet to manage GPU resources.

Fig. 3 The device plugin discovers GPUs on the node but it doesn't report itt o the kubelet.

Fig. 4 The kubelet queries for GPUs from the Device Plugin.

The kubelet updates the node's capacity and allocatable resources accordingly:

resources.yaml

```yaml
status:
  capacity:
    nvidia.com/gpu: '1'
  allocatable:
    nvidia.com/gpu: '1'
```

That's how your Tesla T4 becomes `nvidia.com/gpu: 1` in the node's capacity.

Once this device is registered, a pod can request it like this:

resources.yaml

```yaml
resources:
  limits:
    nvidia.com/gpu: 1
```

Now Kubernetes can schedule GPU workloads like any other resource.

However, unlike CPU/memory, Kubernetes does not share GPUs between pods.

GPU is an integer resource (you will explore how this constraint can be worked around later).

This device plugin model hands unprecedented control to individual plugins, completely breaking Kubernetes' usual resource management patterns.

While Kubernetes manages CPU and memory through kernel cgroups, GPU plugins become the ultimate authority over their hardware.

The plugin owns three critical responsibilities that no other Kubernetes component can override.

Resource Discovery happens through continuous monitoring.

The `ListAndWatch` stream acts like a heartbeat, constantly reporting GPU health and availability to the kubelet.

Allocation Decisions give plugins surgical precision over hardware access.

When a pod requests `nvidia.com/gpu: 1`, the `Allocate` method doesn't just say "yes".

Instead, it crafts the exact container configuration:

device-plugin-interface.go

```go
// Allocate response example
&AllocateResponse{
    ContainerResponses: []*ContainerAllocateResponse{
        {
            Devices: []*DeviceSpec{
                {ContainerPath: "/dev/nvidia0", HostPath: "/dev/nvidia0"},
                {ContainerPath: "/dev/nvidiactl", HostPath: "/dev/nvidiactl"},
            },
            Envs: map[string]string{
                "NVIDIA_VISIBLE_DEVICES": "0",
            },
        },
    },
}
```

Resource Lifecycle control extends from allocation through cleanup.

Unlike CPU slices managed by the Linux kernel, your plugin orchestrates the entire GPU journey: it mounts devices, sets environment variables, and ensures clean teardown when pods terminate.

This absolute control comes with a crucial limitation that shapes everything about GPU scheduling.

Kubernetes core components like scheduler, kubelet, and resource quotas only see abstract quantities like `nvidia.com/gpu: 1`.

They cannot understand GPU memory usage or limits, enforce quality-of-service policies, implement preemption or fair sharing, or monitor actual GPU utilization.

Running a GPU Pod: End-to-End

Now let's see how all these pieces come together when you deploy a GPU workload.

We'll trace every step from pod creation to GPU access.

Here's our test pod:

pod.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: gpu-test
spec:
  containers:
    - name: cuda-container
      image: nvidia/cuda:11.0-base
      command: ['nvidia-smi']
      resources:
        limits:
          nvidia.com/gpu: 1
```

When you create this pod, the Kubernetes scheduler springs into action.

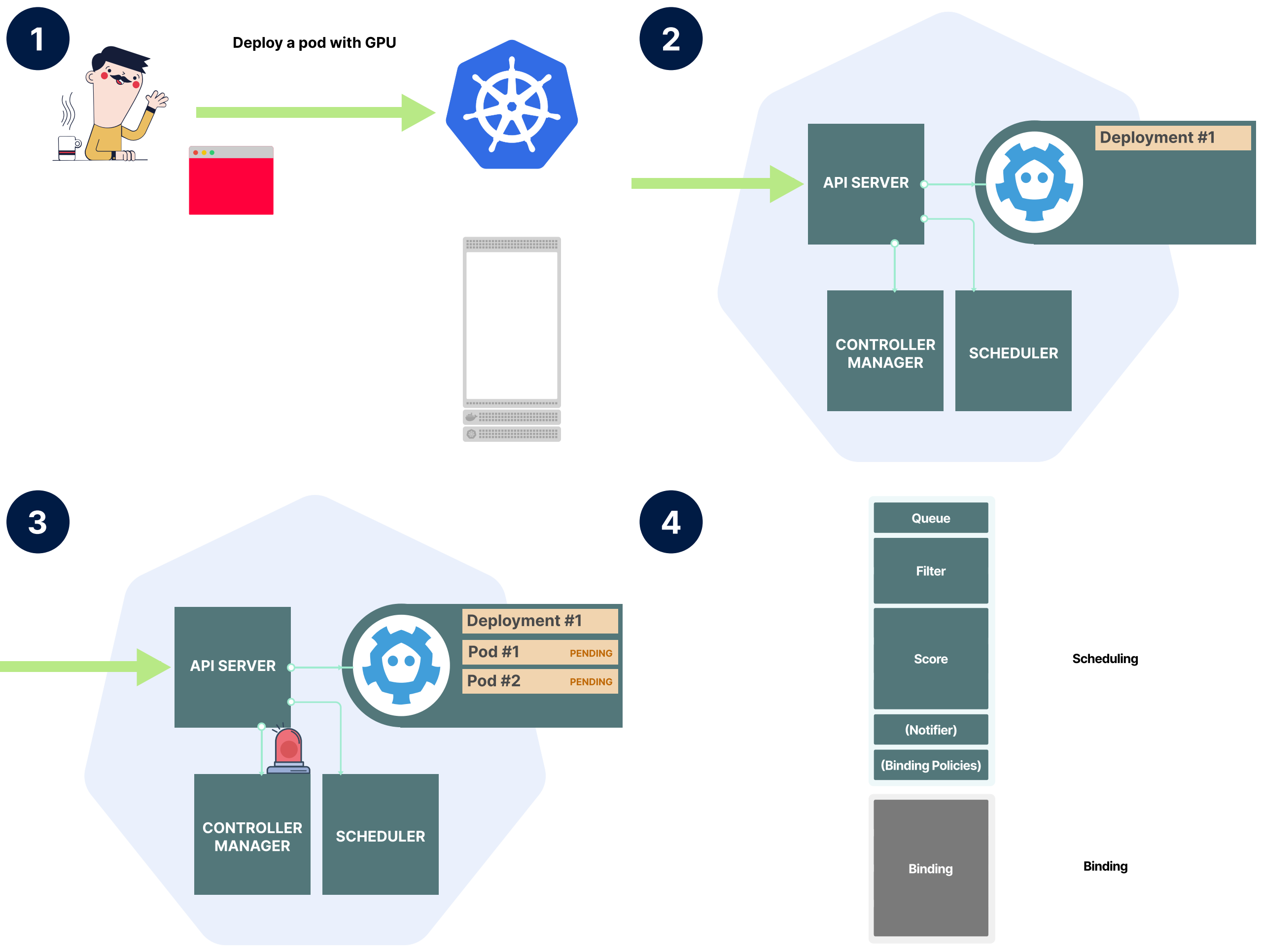


Fig. 1 A deployment request is sent to the Kubernetes API server.

Fig. 2 The API Server receives the deployment, and stores the resource in etcd.

Fig. 3 The controller manager proceeds to create the ReplicaSet and pods. The pods are pending and added to the scheduler's queue.

Fig. 4 The scheduler goes through two phases: filters predicated to decide where to allocate the node.

It sees the pod is requesting `nvidia.com/gpu: 1` and needs to find a suitable node.
The scheduler treats `nvidia.com/gpu` as an extended resource (just an integer to track).
It filters all nodes in the cluster, checking their allocatable resources for any that report at least 1 available `nvidia.com/gpu` .
It finds our minikube node advertising `nvidia.com/gpu: 1` in its allocatable resources and selects it.
The scheduler then writes a binding to the API server, assigning the pod to this node.
The kubelet on our minikube node watches the API server and sees a new pod has been bound to it.
It reads the pod specification and notices something special: this pod requests `nvidia.com/gpu: 1` .
The kubelet knows this isn't a native resource like CPU or memory.
It needs to consult the device plugin.
The kubelet calls the plugin's `Allocate` method:

device-plugin-interface.go

```
AllocateRequest{
    ContainerRequests: []*ContainerAllocateRequest{
        {DevicesIDs: ["GPU-abc123"]},  // The specific GPU identifier
    },
}
```

The NVIDIA device plugin receives this allocation request and prepares detailed instructions for how to give this container GPU access:

device-plugin-interface.go

```go
&AllocateResponse{
    ContainerResponses: []*ContainerAllocateResponse{
        {
            Devices: []*DeviceSpec{
                // Mount the GPU device file
                {ContainerPath: "/dev/nvidia0", HostPath: "/dev/nvidia0"},
                // Mount the control interface
                {ContainerPath: "/dev/nvidiactl", HostPath: "/dev/nvidiactl"},
                // Mount unified memory interface
                {ContainerPath: "/dev/nvidia-uvm", HostPath: "/dev/nvidia-uvm"},
            },
            Envs: map[string]string{
                // Tell NVIDIA runtime which GPU to use
                "NVIDIA_VISIBLE_DEVICES": "0",
                // Enable CUDA support
                "NVIDIA_DRIVER_CAPABILITIES": "compute,utility",
            },
        },
    },
}
```

The kubelet passes these instructions along with the standard pod specification to the CRI.

The CRI must create a container combining standard isolation with GPU access.

First, it creates cgroups for CPU and memory, then establishes namespaces for process and filesystem isolation.

Next, it mounts the device files from the host into the container's filesystem.

Without this, the container wouldn't even know a GPU exists.

It sets the environment variables to tell the NVIDIA runtime which GPU this container should use.

When the container process starts, something interesting happens.

The NVIDIA container runtime hook detects those special environment variables you set earlier.

This hook is a software sitting between containerd and the container itself, and it runs automatically at startup.

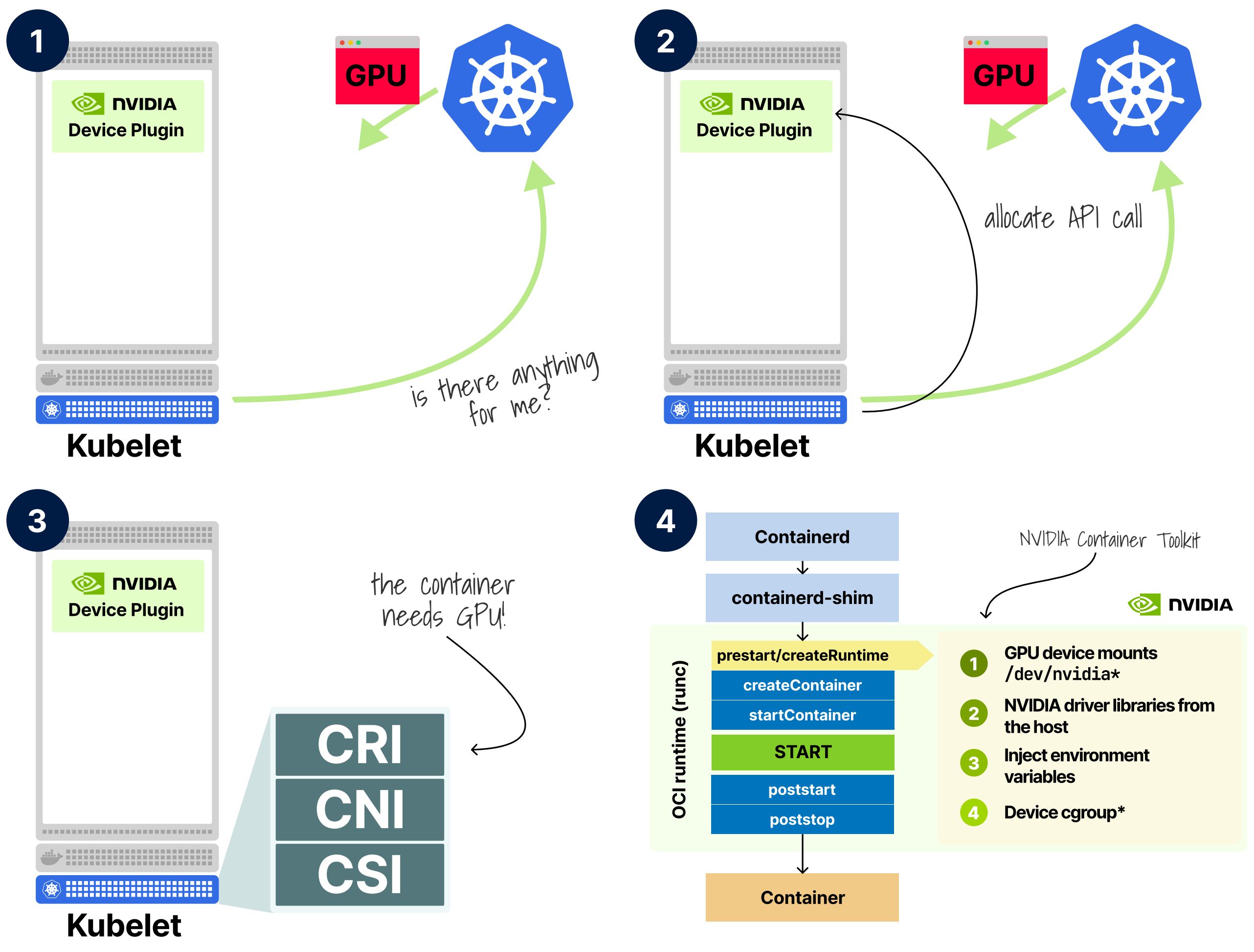


Fig. 1 The kubelet polls for new pod assignments requiring GPU resources.

Fig. 2 The NVIDIA Device Plugin on the node prepares to handle the GPU resource allocation for the incoming pod.

Fig. 3 The kubelet creates the container via the CRI

Fig. 4 The toolkit enables GPU device mounts, NVIDIA driver libraries, environment variables injection, and device cgroups management for the container.

When the container is about to start, this runtime hook:

1. **Checks for the `NVIDIA_VISIBLE_DEVICES` environment variable**. If it's not there, it does nothing and lets the container start normally
2. **If the variable exists, it springs into action**:
    - It queries the host to detect which GPU libraries are installed
    - It determines which version of CUDA libraries match your driver
    - It modifies the container specification to add mounts
3. **Injects the necessary libraries into the container.**
4. **Configures the library paths** so the container can find them

Without this hook, your container would have the `/dev/nvidia0` device file (which is what the device plugin mounted), but wouldn't have the CUDA libraries to actually use it.

It's like having a graphics card but no drivers.

The hook uses the OCI (Open Container Initiative) runtime spec's "prestart hooks" that run after the container is created but before your application starts.

NVIDIA cleverly uses this standard mechanism to inject GPU support at the last possible moment.

This is why you can use a standard Ubuntu container image and suddenly gain GPU capabilities.

The runtime hook adds all the NVIDIA-specific bits at startup time rather than requiring them to be baked into the image.

Here is an example of how it is integrated in containerd using a createContainer hook:

/etc/containerd/config.toml

```
[plugins."io.containerd.grpc.v1.cri"]
  enable_cdi = true
  cdi_spec_dirs = ["/etc/cdi", "/var/run/cdi"]
```

In Kubernetes, the NVIDIA device plugin daemonset typically configures all this automatically when you install it.

But now you know what's happening under the hood when your container magically gains GPU access.

Finally, the application starts.

When it runs `nvidia-smi`, it can see the GPU:

bash
```
$ nvidia-smi
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 550.54.15    Driver Version: 550.54.15    CUDA Version: 12.4     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla T4            Off  | 00000000:00:04.0 Off |                    0 |
| N/A   37C    P8     9W /  70W |      0MiB / 15360MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
```

The application can now:

- Create CUDA contexts (200-300ms initialization)
- Allocate GPU memory (in large pools)
- Launch kernels (non-preemptible execution)
- Process data on thousands of cores

But here's the critical point: **this pod is using the entire GPU**.

All the compute capability is allocated to one pod that only needs a fraction of these resources.

The Sharing Challenge

At this point, you have GPU workloads running in Kubernetes.

But each pod gets an entire GPU.

Your expensive Tesla T4 (or even more costly H100) is allocated to a single pod, even if that pod only uses 10% of its compute or 1GB of its memory.

You might want to share this GPU across teams or workloads, but that opens up a whole new set of challenges:

- How do you prevent one workload from monopolizing the GPU?
- What happens when multiple pods try to use the same GPU memory?
- How do you ensure fair scheduling without kernel support?
- What about security boundaries between different teams' workloads?

These questions lead us to GPU sharing mechanisms and the fundamental trust problems they create, which is the next chapter's subject.

Key Takeaways

Through this exploration, we've seen:

- **Containers rely on kernel primitives**: Syscalls provide the interface, cgroups enforce limits, and namespaces create isolation. The kernel sees and controls everything.
- **GPUs operate outside kernel control**: CUDA contexts replace processes, kernels are non-preemptible, memory is allocated in pools, and the kernel is blind to GPU operations.
- **Device plugins bridge the gap**: They discover GPUs, advertise them as integer resources, and configure containers for GPU access. But they can't provide the fine-grained control Kubernetes has for CPU and memory.
- **The integration works but with limitations**: Pods can use GPUs, but only as monolithic units. There's no native sharing, memory limits, or quality of service—just all-or-nothing allocation.

In the next part, you'll explore what happens when you try to share these expensive GPUs between workloads and why trust becomes the central challenge.

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.

bash

```
$ 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.

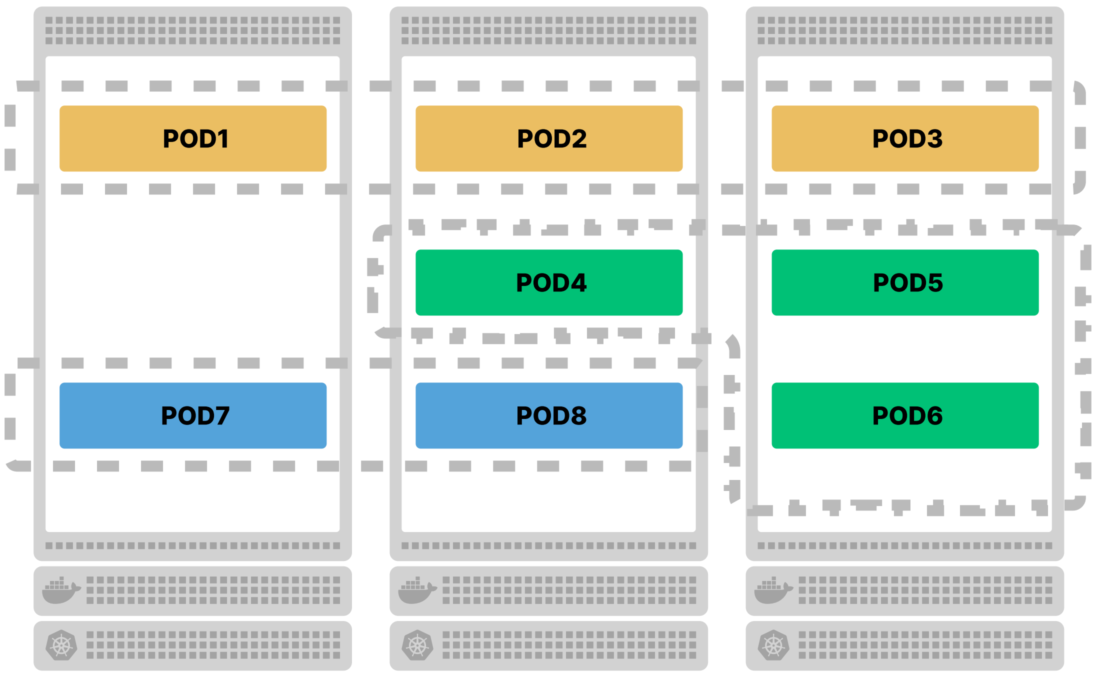


Fig. Kubernetes namespaces don't provide isolation and can span several physical nodes

To illustrate, let's deploy pods across namespaces:

```bash
$ 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.

A pod in `fraud-detection` cannot list pods in `data-analytics` through the API, but it can still reach them over the network unless you add network policies.

Namespaces don't isolate CPU, memory, or devices, they are an API-level construct only.

So, how does Kubernetes prevent them from interfering with each other?

Cgroups: The kernel is always in control

The real enforcement happens through Linux cgroups.

When you define resource limits in a pod:

resources.yaml

```yaml
resources:
  limits:
    cpu: 500m
    memory: 1Gi
```

the kubelet programs those values into the kernel:

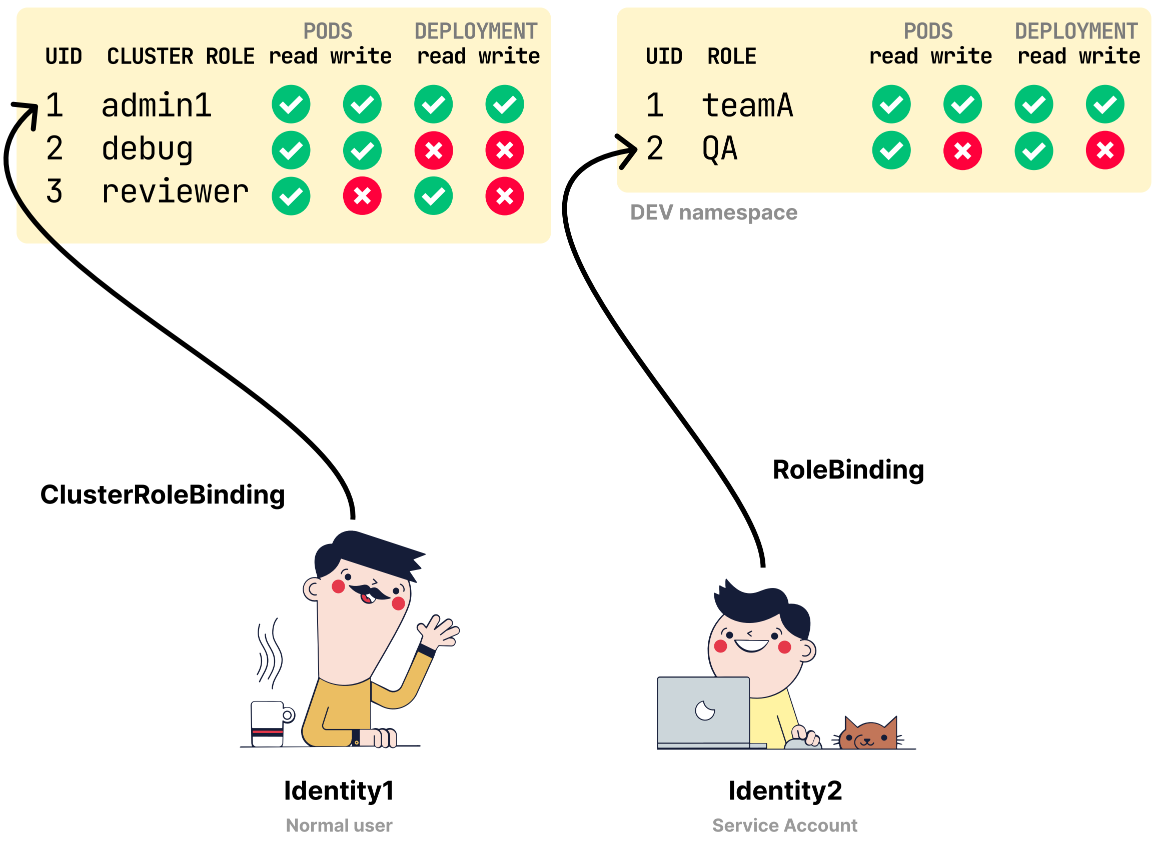
bash

```bash
$ minikube ssh
$ cd /sys/fs/cgroup/kubepods.slice
$ cat .../cpu.max
50000 100000    # 50ms of CPU per 100ms

$ cat .../memory.max
1073741824      # 1 GiB
```

These are hard limits.
If a process tries to exceed its CPU share, the kernel throttles it.
If it tries to allocate beyond 1 GiB, the kernel's OOM killer terminates it.
Every byte and every cycle is visible to the kernel, which enforces fairness across tenants.
This is why Kubernetes can safely pack multiple workloads on one node.

RBAC: Access Control Boundaries

Namespaces define runtime boundaries.
RBAC governs who can manipulate them.



UID	CLUSTER ROLE	PODS read	PODS write	DEPLOYMENT read	DEPLOYMENT write
1	admin1	✓	✓	✓	✓
2	debug	✓	✓	✗	✗
3	reviewer	✓	✗	✓	✗

UID	ROLE	PODS read	PODS write	DEPLOYMENT read	DEPLOYMENT write
1	teamA	✓	✓	✓	✓
2	QA	✓	✗	✓	✗

Fig. RBAC in Kubernetes: Users, Roles, and Bindings

Let's create a service account in `tenant-a` :

```bash
$ kubectl create sa tenant-a-user -n tenant-a
```

Now test access to another namespace:

```bash
$ kubectl auth can-i get pods --namespace=tenant-b --as=system:serviceaccount:tenant-a:tenant-a-user
```

```
no

$ kubectl auth can-i list secrets --namespace=tenant-b --as=system:serviceaccount:tenant-a:tenant-a-user
no
```

RBAC blocks cross-namespace operations through the API.

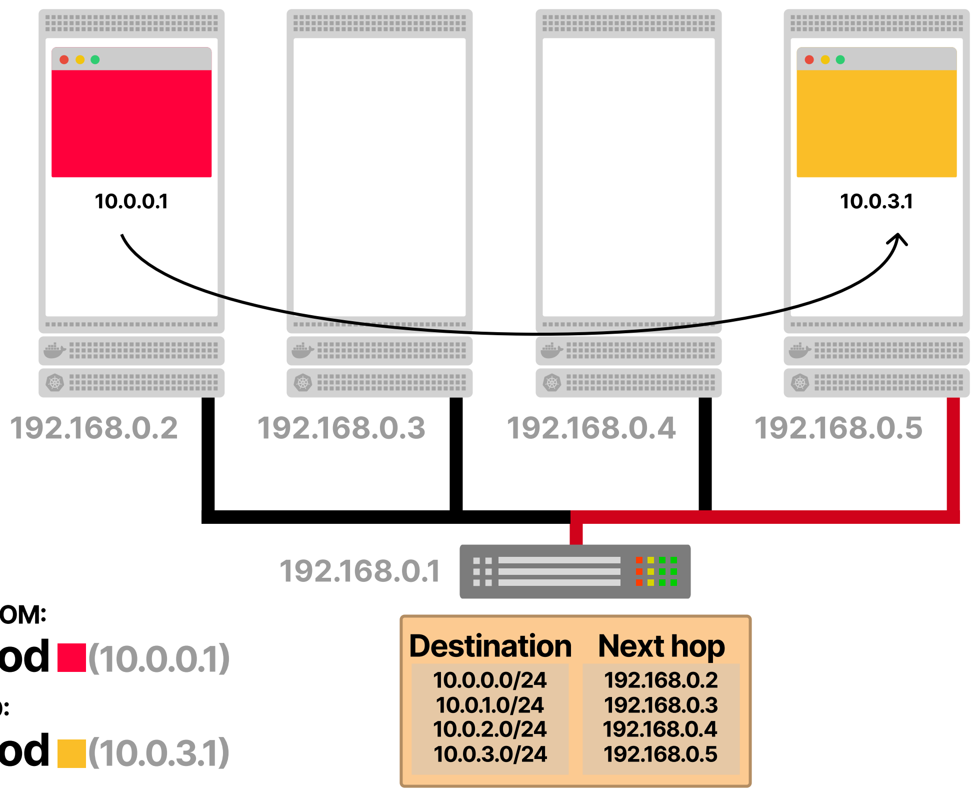
But this only applies to control-plane interactions.

At runtime, a pod in `tenant-a` can still contact a pod in `tenant-b` if it knows the IP:

```bash
# First, create a simple web service in tenant-b
$ kubectl apply -f - <<EOF
apiVersion: v1
kind: Pod
metadata:
  name: web-service
  namespace: tenant-b
spec:
  containers:
  - name: web
    image: nginx:alpine
    ports:
    - containerPort: 80
EOF

# Get the pod's IP address
$ kubectl get pod web-service -n tenant-b -o jsonpath='{.status.podIP}'
10.244.0.15

# Now from a pod in tenant-a, we can directly access tenant-b's service
$ kubectl apply -f - <<EOF
apiVersion: v1
kind: Pod
metadata:
  name: client
  namespace: tenant-a
spec:
  containers:
  - name: client
    image: curlimages/curl:latest
    command: ["sh", "-c", "curl http://10.244.0.15 && sleep 3600"]
EOF

# Check the logs - the connection succeeds
$ kubectl logs client -n tenant-a
```

```
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
...
```

The connection succeeds.



Destination	Next hop
10.0.0.0/24	192.168.0.2
10.0.1.0/24	192.168.0.3
10.0.2.0/24	192.168.0.4
10.0.3.0/24	192.168.0.5

Fig. In Kubernetes, any pod can talk to any pod as long as it knows it IP address

RBAC did not stop it.

This demonstrates the fundamental limitation: namespaces + RBAC secure the API, while cgroups secure CPU and memory.

Network traffic requires additional policies.

Still, with these three layers (e.g., namespaces for logical separation, RBAC for control-plane permissions, and cgroups for hardware limits) Kubernetes achieves effective multi-tenancy for CPU and memory.

This works because the kernel is always in the loop.

Why This Multi-Tenancy Model Works

Every process in Linux belongs to a cgroup, and the kernel maintains that relationship directly.

When a process from tenant-a requests CPU time or allocates memory, the kernel checks its counters and enforces the configured limits.

If the tenant consumes more CPU than allocated, the scheduler throttles it.

If it exceeds its memory limit, the out-of-memory killer terminates it.

The important detail is visibility: the kernel sees every instruction and every byte.

That visibility is why Kubernetes can safely pack multiple workloads onto the same node.

But what happens when tenants share a GPU?

The model collapses, because the kernel is blind.

Why This Model Collapses for GPUs

The first breakdown appears with GPU memory.

When you allocate system memory, the kernel tracks it page by page.

Cgroups immediately reflect the change and enforce hard limits.

When you allocate GPU memory, the request never passes through the kernel.
Instead, it is handled inside the NVIDIA driver in user space.
Let's prove this with a pod that allocates both system RAM and GPU RAM:

bash

```
$ kubectl apply -f - <<EOF
apiVersion: v1
kind: Pod
metadata:
  name: gpu-memory-test
spec:
  containers:
  - name: tester
    image: pytorch/pytorch:2.0.0-cuda11.7-cudnn8-runtime
    command: ["python", "-c"]
    args:
    - |
      import torch, os

      def cgroup_mem():
          try:
              with open("/sys/fs/cgroup/memory.current") as f:
                  return int(f.read().strip()) / 10242
          except:
              return -1

      print("System memory before:", cgroup_mem(), "MB")
      system_data = bytearray(100 * 1024 * 1024)
      print("System memory after 100MB allocation:", cgroup_mem(), "MB")

      print("GPU memory before:", torch.cuda.memory_allocated() / 10242, "MB")
      gpu_data = torch.zeros(25 * 1024 * 1024, device="cuda")  # ~100MB
      print("GPU memory after 100MB allocation:", torch.cuda.memory_allocated() / 10242, "MB")

      print("System memory after GPU allocation:", cgroup_mem(), "MB")
EOF
```

When this pod runs, the cgroup counters increase after allocating 100MB of system RAM, but they do not change after allocating 100MB of GPU RAM.
Yet `torch.cuda.memory_allocated()` shows the GPU memory is consumed.
The kernel has no visibility into GPU memory.
The same problem extends to scheduling.

GPU Scheduling Happens Outside the Kernel

On CPUs, the Linux kernel preempts processes in microseconds, saving and restoring their state to enforce fair sharing.

On GPUs, kernels launch through the NVIDIA driver.

Once a kernel begins, it runs to completion.

The Linux kernel cannot interrupt it, cannot time-slice it, and cannot enforce fairness across tenants.

This means that when two tenants share a GPU, one can invisibly consume memory, launch long-running kernels, and starve the other.

The enforcement model that worked for CPU and memory no longer applies.

Let's demonstrate this difference with a pod that runs both a CPU and a GPU-bound task in the same container:

bash

```bash
$ kubectl apply -f - <<EOF
apiVersion: v1
kind: Pod
metadata:
  name: scheduler-demo
spec:
  containers:
  - name: tasks
    image: pytorch/pytorch:2.0.0-cuda11.7-cudnn8-runtime
    command: ["python", "-c"]
    args:
    - |
      import torch, threading, time

      def cpu_task():
          print("Starting CPU task...")
          start = time.time()
          sum(i*i for i in range(30_000_000))
          print(f"CPU task finished in {time.time() - start:.2f}s")
```

```
$       def gpu_task():
            print("Starting GPU task...")
            start = time.time()
            x = torch.randn(2000, 2000, device="cuda")
            for _ in range(50):
                torch.matmul(x, x.T)
                torch.cuda.synchronize()
            print(f"GPU task finished in {time.time() - start:.2f}s")

        t1 = threading.Thread(target=cpu_task)
        t2 = threading.Thread(target=gpu_task)
        t1.start(); t2.start()
        t1.join(); t2.join()
      resources:
        limits:
          cpu: "200m"
          memory: "512Mi"
          nvidia.com/gpu: 1
    restartPolicy: Never
  EOF
```

When you check the logs, something interesting happens:

- **The CPU task runs under the kernel's control.** Because the container is limited to `200m` (20% of a core), the task takes several seconds to complete. The kernel throttles it exactly as configured.
- **The GPU task completes in under a second**, unaffected by the CPU quota or memory limit.

Why?
Because GPU scheduling lives in a completely separate universe.

CUDA Contexts Span Containers

The third breakdown comes from the way CUDA contexts operate.
A CUDA context is the driver's private environment for a single process.

It keeps track of the GPU memory the process has allocated, the kernels it has loaded, and the streams it executes.

Every process that uses CUDA creates its own context.

The driver then switches between contexts as needed.

When a container requests `nvidia.com/gpu: 1`, Kubernetes mounts the entire GPU device into that pod.

Inside the pod, frameworks such as CUDA or PyTorch communicate directly with the GPU driver.

The driver is the layer responsible for all interaction with the physical device, and it is also the component that creates and manages contexts.

On Linux, when you install the NVIDIA driver, it registers GPUs with the kernel and exposes them as device files under `/dev/`:

```bash
$ ls -l /dev/nvidia*
crw-rw-rw- 1 root root 195,   0 Aug 25 12:00 /dev/nvidia0
crw-rw-rw- 1 root root 195,   1 Aug 25 12:00 /dev/nvidia1
crw-rw-rw- 1 root root 195, 255 Aug 25 12:00 /dev/nvidiactl
```

The key entry point for applications is `/dev/nvidia0`.

It is just a file representing the first GPU.

From the kernel's perspective, this is simply a file that a process can open.

When a process (such as PyTorch or TensorFlow) starts using CUDA, it opens the file descriptor as it would any file.

This means that if two containers can both access `/dev/nvidia0`, they can create CUDA contexts.

Everything else such as allocation of GPU memory, loading of kernels, scheduling of execution, happens inside the NVIDIA driver.

When this pod runs, both containers allocate GPU memory and create their own CUDA context:

```bash
$ kubectl apply -f - <<
$ EOF
  apiVersion: v1
  kind: Pod
  metadata:
    name: context-demo
  spec:
    containers:
    - name: container-a
      image: pytorch/pytorch:2.0.0-cuda11.7-cudnn8-runtime
      command: ["python", "-c"]
      args:
      - |
        import torch, os
        print(f"Container A PID: {os.getpid()}")
        a = torch.full((1000, 1000), 1.0, device="cuda")
        print(f"Container A allocated {torch.cuda.memory_allocated()} bytes of GPU memory")

    - name: container-b
      image: pytorch/pytorch:2.0.0-cuda11.7-cudnn8-runtime
      command: ["python", "-c"]
      args:
      - |
        import torch, os
        print(f"Container B PID: {os.getpid()}")
        b = torch.full((1000, 1000), 2.0, device="cuda")
        print(f"Container B allocated {torch.cuda.memory_allocated()} bytes of GPU memory")

    restartPolicy: Never
  EOF
```

These are just two processes sharing the same physical device at the GPU driver level.

The driver doesn't know or care that they are in different containers.

It only sees two file descriptors opened to `/dev/nvidia0`.

Kubernetes Enforces Scheduling Only at the API Layer

When you request a GPU in Kubernetes, the system doesn't manage the hardware directly.

It only enforces the request through scheduling rules and local bookkeeping.

When a pod requests `nvidia.com/gpu: 1`, here's what actually happens:

1. The scheduler sees `nvidia.com/gpu` as just a number.
2. It finds a node that advertises at least one available GPU.
3. It assigns the pod to that node using its normal scoring rules.
4. The kubelet on that node receives the pod specification.
5. The kubelet calls the NVIDIA device plugin, since this is not a native resource.
6. The plugin picks a GPU and returns the device files to mount.
7. The kubelet mounts those files into the container and marks the GPU as taken.

Notice what did not happen: at no point did Kubernetes check the GPU hardware.

The scheduler has no GPU-specific logic.

To it, `nvidia.com/gpu: 1` is no different than `example.com/device: 1`.

The actual allocation happens only on the node, and even there it is managed by the plugin, not the Linux kernel.

Kubernetes then prevents other pods from requesting the same GPU by updating its internal counters.

In other words, Kubernetes enforces a simple rule: one pod per GPU.

But this rule lives only in the control plane.

At the hardware level, the GPU is still a shared physical device.

Any process that can open `/dev/nvidia0` can use it.

This gap between what Kubernetes enforces and what the hardware allows is where the security and performance problems begin.

Threat Scenarios in Practice

These architectural gaps create security vulnerabilities that traditional Kubernetes boundaries cannot prevent.

Let's examine three common attack patterns.

In a shared GPU environment, one tenant can easily starve others of resources. First, `tenant-a` starts a stable training workload:

```bash
$ kubectl apply -f - <<EOF
apiVersion: v1
kind: Pod
metadata:
  name: stable-training
  namespace: tenant-a
spec:
  containers:
  - name: training
    image: nvidia/cuda:11.0-base
    command: ["python3", "-c"]
    args:
      - |
        import time
        for epoch in range(100):
          print(f"Epoch {epoch+1}/100: loss=0.{234-epoch*2}, memory=2.1GB")
          print("Training stable at 45ms per batch")
          time.sleep(2)
    resources:
      limits:
        nvidia.com/gpu: 1
EOF
```

The training runs smoothly:

```bash
$ kubectl logs -f stable-training -n tenant-a
Epoch 1/100: loss=0.234, memory=2.1GB
Training stable at 45ms per batch
Epoch 2/100: loss=0.232, memory=2.1GB
Training stable at 45ms per batch
```

Now, `tenant-b` deploys a memory hog:

```bash
```

```
$ kubectl apply -f - <<EOF
apiVersion: v1
kind: Pod
metadata:
  name: memory-hog
  namespace: tenant-b
spec:
  containers:
  - name: hog
    image: nvidia/cuda:11.0-base
    command: ["python3", "-c"]
    args:
      - |
        print("Allocating 8GB GPU memory...")
        # In reality, this would allocate massive GPU arrays
        print("Allocation complete. Holding memory.")
        import time
        time.sleep(3600)
    resources:
      limits:
        nvidia.com/gpu: 1
EOF
```

If both pods were actually sharing the same GPU, `tenant-a`'s training would crash:

```bash
# CUDA out of memory error: tried to allocate 2.10GB
# Training failed.
```

Namespace boundaries and RBAC did nothing to prevent this.

The GPU is a shared resource outside Kubernetes' control.

But there's more.

Sensitive data can leak across tenants when GPU memory persists after pods exit.

A financial services company runs proprietary models:

```bash
$ kubectl apply -f - <<EOF
apiVersion: v1
kind: Pod
metadata:
  name: financial-model
  namespace: tenant-a
spec:
  containers:
  - name: model
    image: pytorch/pytorch
    command: ["python3", "-c"]
    args:
      - |
        import torch
        import os

        # Load sensitive data into GPU
        secret_weights = torch.tensor([0.7234, -0.3456, 0.8891], device='cuda')
        customer_ssn = torch.tensor([123456789, 987654321], device='cuda')

        print("Processing sensitive financial data...")

        # Simulate crash - no cleanup!
        os._exit(139)  # Segmentation fault

        # This never runs:
        # torch.cuda.empty_cache()
    resources:
      limits:
        nvidia.com/gpu: 1
EOF
```

The pod crashes due to segfault.
Another tenant can potentially recover this data:

```bash
$ kubectl apply -f - <<EOF
apiVersion: v1
kind: Pod
metadata:
  name: data-scanner
  namespace: tenant-b
spec:
  containers:
  - name: scanner
    image: pytorch/pytorch
    command: ["python3", "-c"]
    args:
      - |
$         import torch
          import numpy as np

          # Allocate memory WITHOUT initialization
          # torch.empty() doesn't zero memory (unlike torch.zeros())
          dirty_memory = torch.empty(1000000, dtype=torch.float32, device='cuda')

          # Check what's in this "empty" memory
          cpu_copy = dirty_memory.cpu().numpy()
          non_zero = cpu_copy[cpu_copy ≠ 0]

          if len(non_zero) > 0:
              print(f"Found {len(non_zero)} non-zero values in 'empty' memory!")
              print(f"Sample values: {non_zero[:10]}")
              # Could contain previous tenant's model weights or data
      resources:
        limits:
          nvidia.com/gpu: 1
  EOF
```

The scanner, which is running in a completely different namespace, recovered sensitive financial data.

And a last very common example.

Long-running kernels can block time-sensitive workloads indefinitely.

A batch processing job locks the GPU:

bash

```
$ kubectl apply -f - <<EOF
apiVersion: v1
kind: Pod
metadata:
  name: batch-processor
  namespace: engineering
spec:
  containers:
  - name: batch
    image: nvidia/cuda:11.0-base
    command: ["python3", "-c"]
    args:
      - |
        import time
        print("Starting 60-minute matrix computation...")
        for i in range(60):
          print(f"Progress: {i+1}/60 minutes completed")
```

```
$           time.sleep(60)
      resources:
        limits:
          nvidia.com/gpu: 1
  EOF
```

A real-time inference service tries to run:

```bash
$ kubectl apply -f - <<EOF
apiVersion: v1
kind: Pod
metadata:
  name: realtime-inference
  namespace: marketing
spec:
  containers:
  - name: inference
    image: nvidia/cuda:11.0-base
    command: ["nvidia-smi"]
    resources:
      limits:
        nvidia.com/gpu: 1
EOF
$ kubectl get pods -A | grep -E "batch|realtime"
engineering   batch-processor      1/1   Running   0    10m
marketing     realtime-inference   0/1   Pending   0    5m
```

The inference service remains pending for the full hour.

No SLA can be met, no guarantees can be made.

##GPU Sharing Options: Finding the Right Balance

Given these fundamental challenges, how can we share GPUs safely?

NVIDIA provides several approaches, each making different trade-offs between utilization and isolation.

Multi-Process Service (MPS) is the most straightforward way to share GPUs.

It doesn't require any special configuration in Kubernetes.

You just enable it on the GPU node, and multiple processes can use the same

GPU simultaneously.

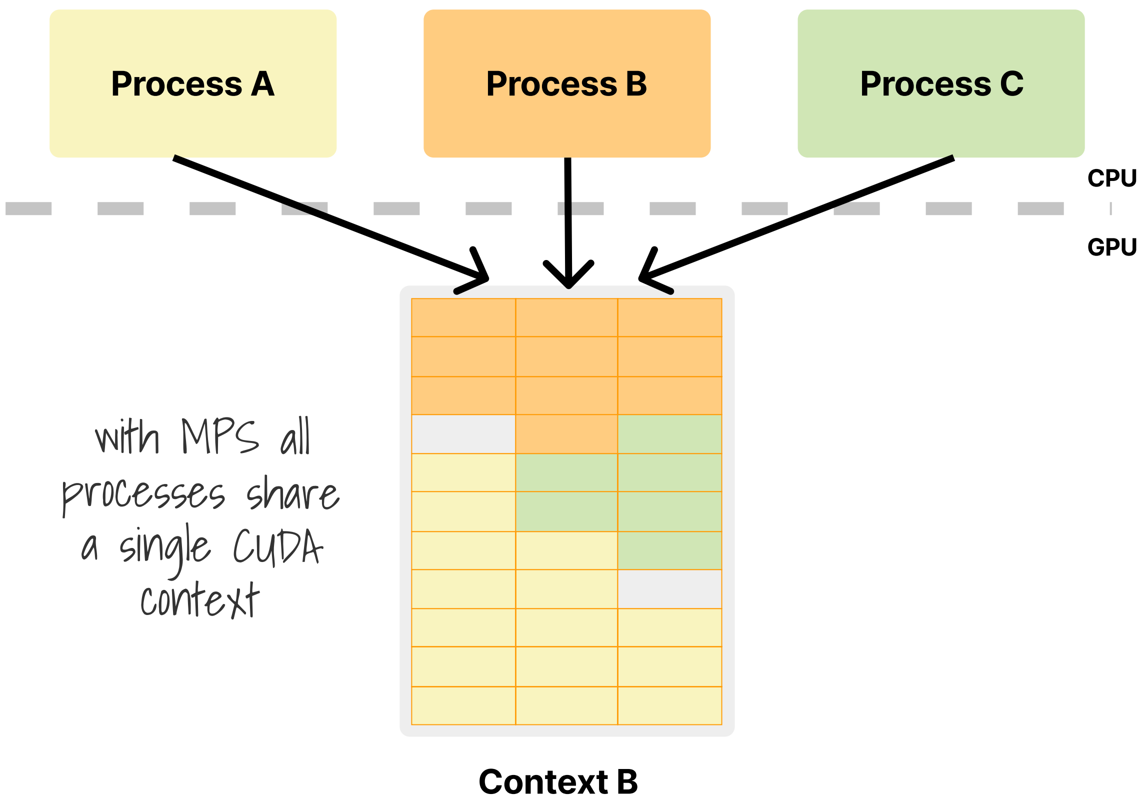


Fig. Multi-Process Service (MPS) is the easiest way to share a GPU.

MPS creates a single shared CUDA context that all processes use together.

Usually, each CUDA process creates its own context—essentially a private GPU session with its memory allocations, loaded kernels, and execution state.

The GPU driver has to switch between these contexts constantly, just like an operating system switches between processes.

Each switch takes time and creates overhead.

MPS eliminates this overhead by having all processes share one context.

Instead of each process having its own GPU session, they all submit work through a shared MPS server process.

This shared context means no switching overhead, resulting in better performance:

```bash
# Without MPS - high variance, slower due to context switching
App1 Average: 0.0847s
App1 StdDev: 0.0234s    # High variance from context switching

App2 Average: 0.0891s
App2 StdDev: 0.0267s    # Also high variance
```

When you enable MPS, the performance improves dramatically:

```bash
# Enable MPS
$ nvidia-smi -c EXCLUSIVE_PROCESS
$ nvidia-cuda-mps-control -d

# With MPS - consistent, faster
App1 Average: 0.0623s   # 26% faster
App1 StdDev: 0.0031s    # 87% less variance

App2 Average: 0.0629s   # 29% faster
App2 StdDev: 0.0028s    # 90% less variance
```

The performance improvement is significant, but MPS has severe limitations that make it unsuitable for multi-tenant environments.

First, there's no memory tracking or enforcement.

When multiple processes share the MPS context, they all draw from the same GPU memory pool without accounting for who uses what.

If Process A allocates 8GB and Process B tries to allocate 10GB on a 16GB GPU, Process B will simply fail with an out-of-memory error.

There's no way to know Process A was the culprit, and no way to enforce limits.

Second, there's no visibility into individual process behavior.

Tools like `nvidia-smi` show aggregate GPU usage, not per-process breakdowns.

You can't tell which process is consuming memory or computing resources.

This makes debugging and capacity planning nearly impossible.

Third, and most critically, all processes share the same memory space.

This creates a massive security vulnerability:

```bash
# One bad actor affects everyone
$ kubectl logs memory-corruption-test
Process A: Running normally...
Process B: Corrupting shared memory...
Process A: Segmentation fault
Process B: Segmentation fault
MPS Server: Fatal error, restarting...
```

If one process crashes or corrupts memory, it can bring down all other processes using that GPU.

MPS is perfect for trusted workloads from the same team where performance matters more than isolation.

But what if you need some level of resource accounting?

What if you want Kubernetes to at least track which pods are using the GPU?

This is where time-slicing becomes useful.

Time-Slicing: Multiplying GPU Count

Time-slicing is the next step up in complexity.

Unlike MPS, which lets multiple processes share, time-slicing makes one physical GPU appear as multiple logical GPUs to Kubernetes.

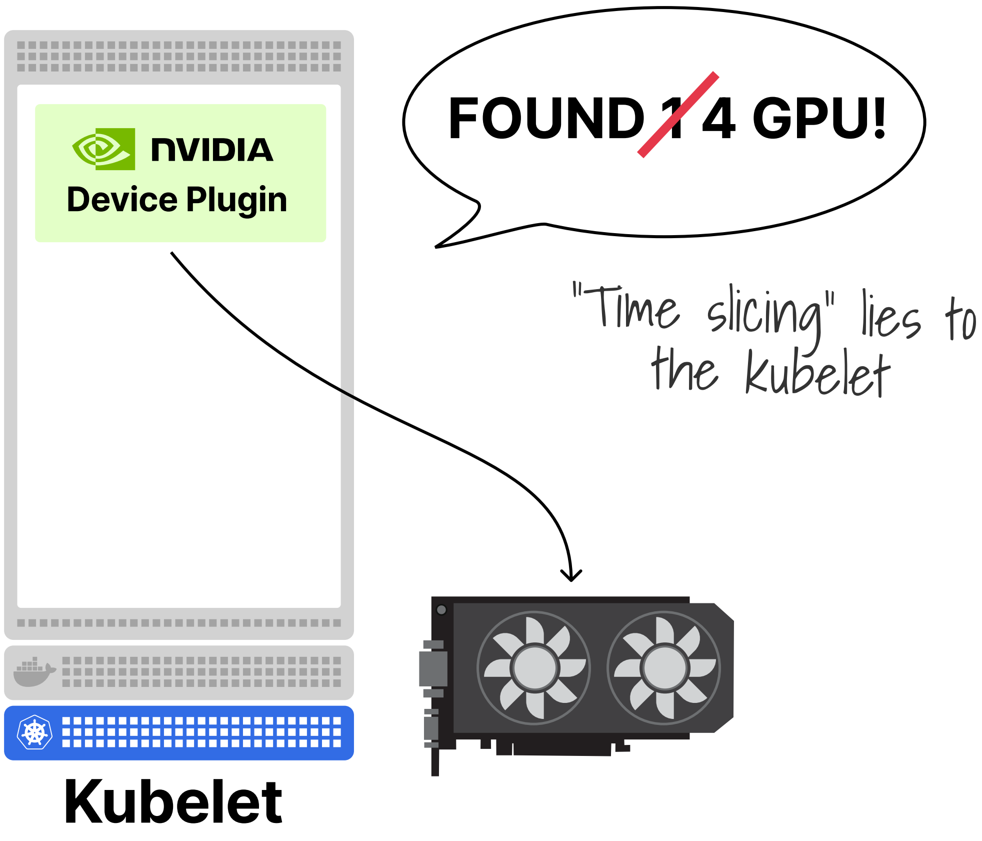


Fig. "Time slicing" makes one physical GPU appear as multiple logical GPUs

You can configure the NVIDIA device plugin to advertise multiple replicas:

configmap.yaml

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: nvidia-device-plugin-config
data:
  config.yaml: |
    version: v1
    sharing:
      timeSlicing:
        replicas: 4  # One GPU becomes four
```

After applying this configuration:

```bash
$ kubectl describe node minikube | grep nvidia.com/gpu
nvidia.com/gpu:  4
nvidia.com/gpu:  4
```

Your single Tesla T4 now appears as four schedulable GPUs.
Multiple pods can be scheduled:

```bash
$ kubectl get pods
NAME        READY   STATUS    GPU-REQUEST
workload-1  1/1     Running   nvidia.com/gpu: 1
workload-2  1/1     Running   nvidia.com/gpu: 1
workload-3  1/1     Running   nvidia.com/gpu: 1
workload-4  1/1     Running   nvidia.com/gpu: 1
```

Each pod gets its own CUDA context, and with time-slicing, the GPU driver coordinates switching between them.

But here's the crucial part: these contexts belong to running processes (pods) that are all active simultaneously in Kubernetes.

When workload-1's pod runs, it creates a CUDA context and might allocate 12GB.

When workload-2's pod starts running, it creates its own context.

Both pods are "Running" in Kubernetes - they're both active processes holding their CUDA contexts, even if only one can execute on the GPU at a given moment.

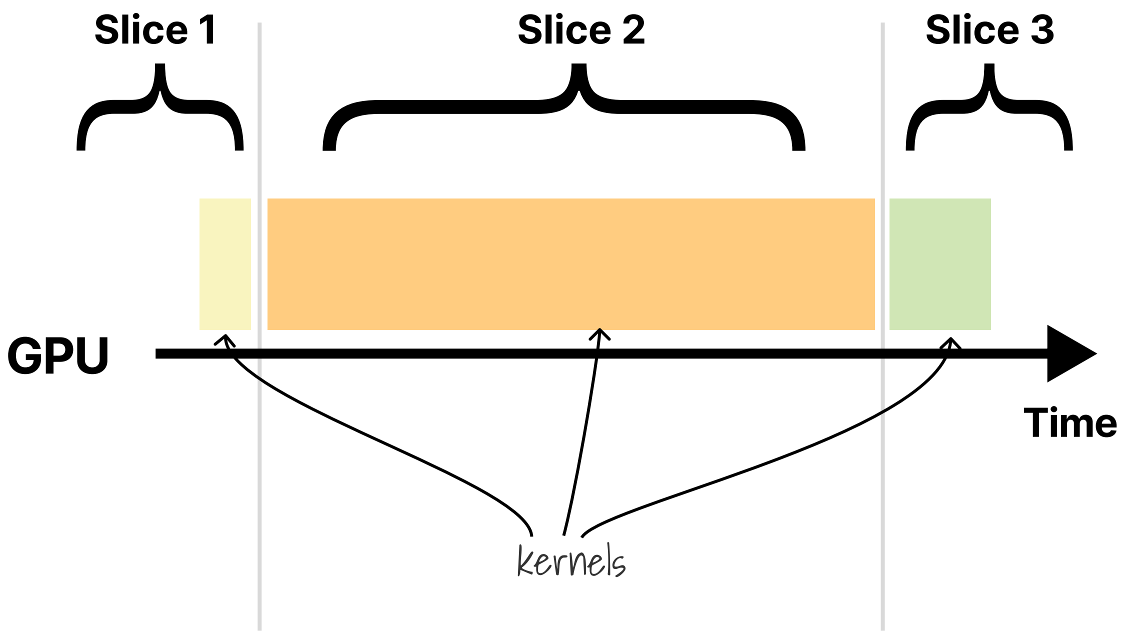


Fig. Processes take turns to execute Kernels on the GPU

The GPU driver time-slices execution between these contexts and gives each one a turn to run kernels.

But the contexts themselves persist because the processes are still alive.

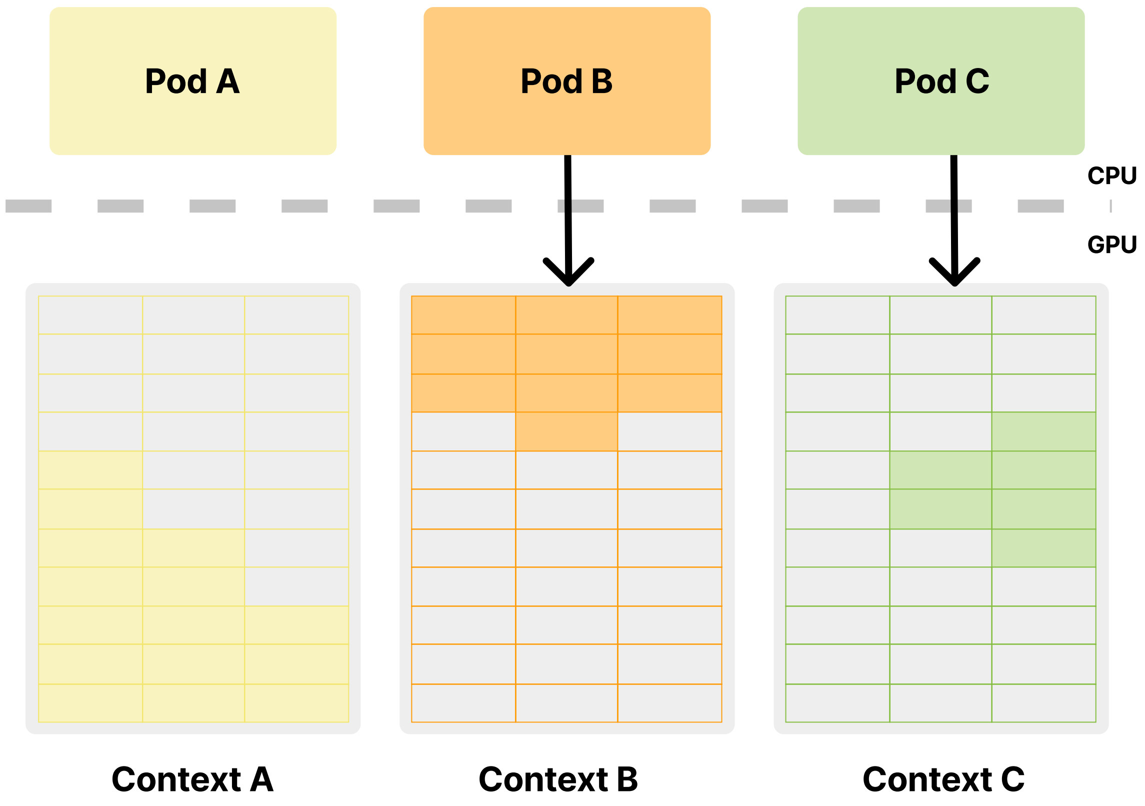


Fig. Each process has a dedicated Context for a long as it needs.

When it's Context B's turn to execute, Context A doesn't get unloaded; it just gets paused.

Its memory allocations remain because the process is still running, and it expects its data to be there when it gets its next slice.

With time-slicing, all four workloads are running simultaneously in Kubernetes—they're just taking turns on the GPU.

The key difference from MPS is that contexts are separate.

When Context A allocates memory, Context B can't read or write to that memory.

This provides a basic security boundary: processes can't spy on each other's data.

But they still compete for the same physical memory pool.

If workload-1 allocates 12GB for its model, that memory stays allocated as long as the pod runs.
The other three workloads must share the remaining 4GB, potentially failing if they need more.

MIG: Hardware Partitioning

Multi-Instance GPU (MIG) provides true hardware-level isolation, but only on A100 and H100 GPUs.
MIG physically splits the GPU into isolated instances:

bash

```bash
# On an A100:
$ nvidia-smi mig -cgi 1g.5gb,2g.10gb,3g.20gb

$ nvidia-smi -L
GPU 0: NVIDIA A100-SXM4-40GB (UUID: GPU-0dcf3380)
  MIG 3g.20gb  Device 0: (UUID: MIG-fb9af708)
  MIG 2g.10gb  Device 1: (UUID: MIG-a0d4e8e4)
  MIG 1g.5gb   Device 2: (UUID: MIG-52a84557)
```

Each MIG instance has:

- Dedicated compute units (SMs)
- Isolated memory (5GB, 10GB, or 20GB)
- Separate memory bandwidth
- Independent fault domains

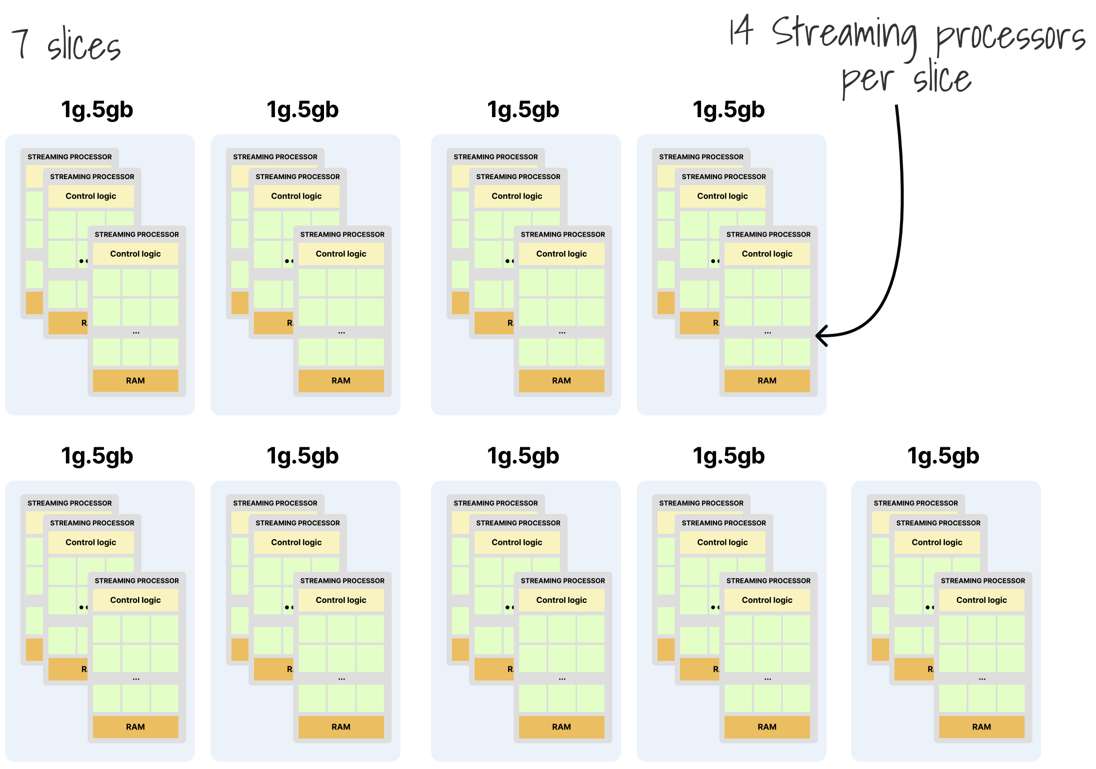


Fig. MIG physically splits the GPU into isolated instances

Kubernetes sees these as distinct GPU types:

```bash
$ kubectl describe node | grep nvidia.com/mig
nvidia.com/mig-1g.5gb:   1
nvidia.com/mig-2g.10gb:  1
nvidia.com/mig-3g.20gb:  1
```

Pods request specific slices:

resources.yaml

```yaml
resources:
  limits:
    nvidia.com/mig-2g.10gb: 1
```

MIG provides true isolation: one slice cannot access another's memory or interfere with its computation.
But you need A100/H100 hardware, and you're limited to seven slices maximum.

vGPU: The Enterprise Solution

Virtual GPU (vGPU) takes GPU sharing to the hypervisor level, providing the same isolation guarantees that VMs have delivered for decades.
Instead of sharing at the process level, like MPS, or the scheduling level, like time-slicing, vGPU creates virtual machines that each believe have their own dedicated GPU.
The hypervisor manages the hardware, parceling out time and memory to each VM according to configured profiles.

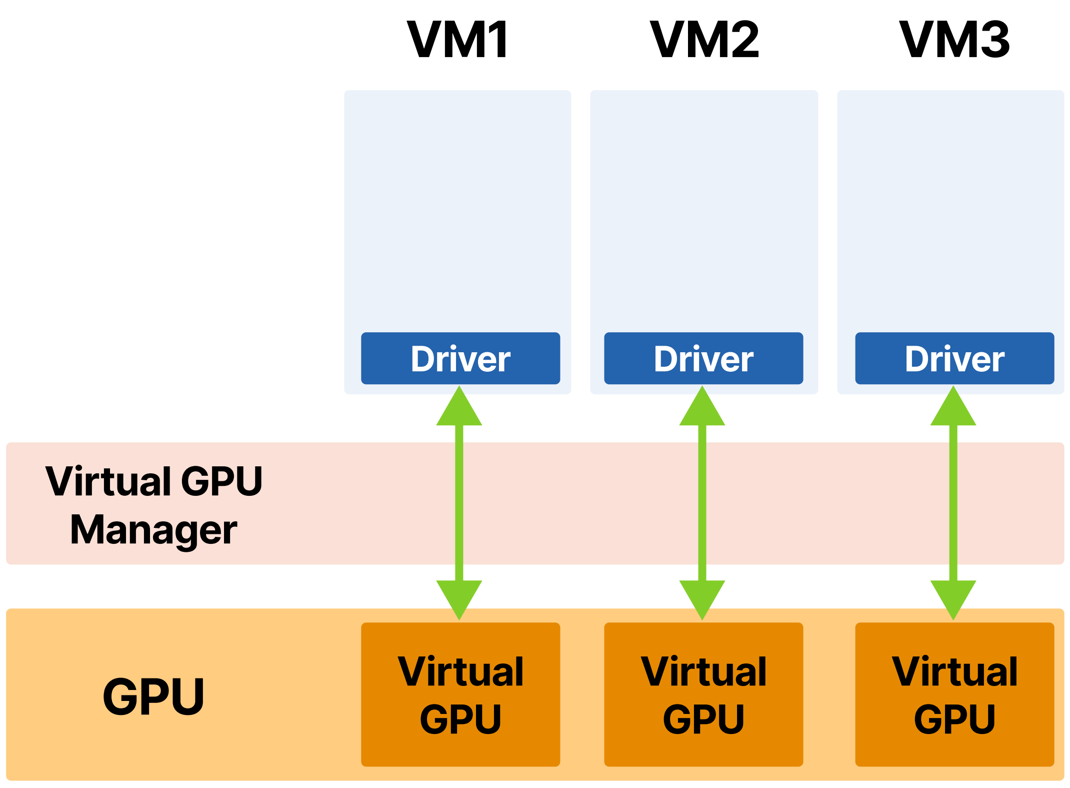


Fig. vGPU creates virtual machines that each believe have their own dedicated GPU

This approach requires significant infrastructure investment.

You need NVIDIA vGPU software licenses, which are sold per GPU per year and can cost as much as the hardware.

You also need a supported hypervisor with specific patches, such as VMware vSphere, Citrix XenServer, or KVM.

Not all GPUs support vGPU either; you need datacenter-grade cards like the A40 or A100.

When you deploy vGPU, each virtual machine gets what appears to be a dedicated GPU.

From inside the VM, applications see a standard NVIDIA device with a fixed amount of memory.

They can install drivers, run CUDA applications, and even use tools like `nvidia-smi` without knowing they're on shared hardware.

The hypervisor enforces strict boundaries between VMs.

If one VM tries to allocate more GPU memory than its profile allows, it gets an out-of-memory error.

If one VM crashes its GPU driver, other VMs continue running unaffected: this is true isolation, enforced below the operating system level.

vGPU also enables capabilities that bare-metal GPUs lack:

- You can live-migrate VMs between hosts without disrupting GPU workloads
- You can snapshot GPU state for backup and recovery
- and dynamically adjust GPU profiles based on workload demands.

For enterprises running critical workloads, these operational benefits often justify the complexity.

But this isolation comes at a cost.

The hypervisor adds overhead—typically 20-30% performance degradation compared to bare metal.

The licensing model is complex, with different licenses for different use cases (virtual workstation vs. virtual compute).

And you need an entire virtualization stack, including shared storage, vCenter or equivalent management tools, and staff trained in virtualization and GPU technologies.

The Trust Spectrum: Choosing Your Strategy

GPU sharing only works when you understand your trust boundaries.

The weaker the trust, the stronger the isolation must be.

When everyone works together with shared goals, Time-slicing or MPS works fine.

People coordinate informally, share resources willingly, and tolerate occasional interference.

You might need MIG or dedicated GPUs for each department if you have different departments with competing priorities.

Accidental interference is likely without hardware boundaries.

Paying customers with SLA requirements might benefit from VM isolation or dedicated nodes, which are essential since they expect guaranteed resources and complete isolation.

And finally, regulated industries or adversarial workloads might not want to share anything.

Each tenant gets dedicated physical GPUs with no possibility of cross-contamination.

Comparing All Options

Here's how all GPU sharing mechanisms stack up:

Approach	Isolation	Performance	Complexity	Hardware Requirement
MPS	None	Excellent	Low	Any GPU
Time-slicing	None	Good	Low	Any GPU
MIG	Hardware	Good	Medium	A100/H100 only
vGPU	VM-level	Fair (70–80%)	High	License + specific GPUs
Dedicated	Complete	Perfect	Low	Any GPU

Key Takeaways

GPU multi-tenancy challenges stem from fundamental architectural differences:

Traditional Kubernetes isolation works because the kernel controls everything. Cgroups enforce limits, namespaces create boundaries, and RBAC controls access. These mechanisms have proven reliable for CPU and memory over decades.

GPUs break every assumption about resource isolation. There are no GPU cgroups, no preemptible execution, and no kernel visibility. The driver manages everything, and Kubernetes can only watch from the sidelines.

Trust determines architecture. In high-trust environments, you can share GPUs with minimal isolation and rely on human coordination. In low-trust environments, you need hardware boundaries or physical separation.

Every sharing mechanism trades something. Time-slicing trades isolation for simplicity. MPS trades safety for performance. MIG trades flexibility for security. vGPU trades performance for isolation. There's no perfect solution—only the right solution for your trust model.

In the next part, we'll explore how to implement these sharing mechanisms, starting with the simplest approaches and building toward production-grade orchestration.

Chapter 3

Orchestrating GPU Sharing - How Kubernetes Manages Turn-Taking

In this chapter, you will learn:

- How GPUs have always supported multiple processes through automatic context switching
- Why GPU memory isn't partitioned between processes like CPU memory
- How KAI-Scheduler uses "reservation pods" to trick Kubernetes into GPU sharing
- What NVIDIA "time-slicing" actually does (spoiler: it's not time-slicing)
- Why neither approach provides true isolation or fairness guarantees
- The critical difference between orchestrating sharing and enforcing it

Let's start by revealing something that might surprise you about GPU capabilities.

The Hidden Truth: GPUs Already Support Sharing

GPUs have always supported multiple processes.
Let me demonstrate this on our test machine with a Tesla T4:

bash

```
$ python matrix_multiply.py
Starting matrix operations on GPU 0...
Iteration 1: 1.23s
Iteration 2: 1.21s
Iteration 3: 1.24s
```

While that's running, open another terminal:

bash

```
$ python vector_add.py
Starting vector addition on GPU 0...
Iteration 1: 0.89s
Iteration 2: 1.45s  # Slower - competing with matrix multiply
Iteration 3: 0.91s
```

Both processes are using the same GPU. Check nvidia-smi:

```bash
$ nvidia-smi
+-----------------------------------------------------------------------------------------+
| Processes:                                                                              |
|  GPU   GI   CI        PID   Type   Process name                              GPU Memory |
|=========================================================================================|
|    0   N/A  N/A      4521    C     python matrix_multiply.py                   2341MiB |
|    0   N/A  N/A      4627    C     python vector_add.py                        1122MiB |
+-----------------------------------------------------------------------------------------+
```

Two processes, one GPU, both running.
No special configuration required.
This has worked since CUDA was invented.
But how does this actually work under the hood?

Understanding GPU Context Switching

When multiple processes use the same GPU, they don't run simultaneously in the way you might expect.
GPUs operate fundamentally differently from CPUs.
When a CUDA kernel launches, it monopolizes the GPU's compute units until completion.
There's no way to pause it, save its state, and switch to another kernel

mid-execution.

So how can two processes both use the GPU?

They take turns.

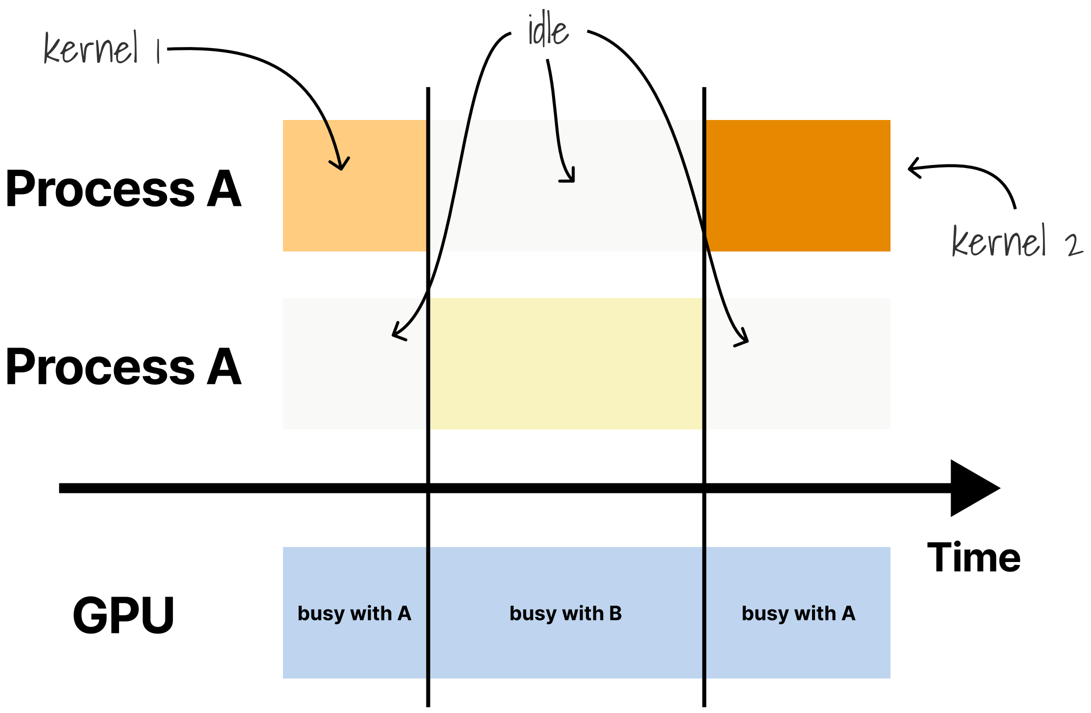


Fig. Kernel on the GPU take turns

Each kernel runs to completion without interruption.

When Process A's kernel finishes, Process B's kernel can start.

The NVIDIA driver manages this queue of kernels from different processes, scheduling them one after another.

This is context switching at the driver level, and it happens automatically.

But an important detail about memory breaks most people's mental model.

The Memory Illusion

Here's where things get counterintuitive.
Let's say you have a 24GB GPU, and you want to run:

- Application A that needs 16GB
- Application B that needs 8GB

You might think: "Perfect! 16 + 8 = 24GB, they'll fit together."
It's more complex than that.
Watch what actually happens:

bash

```bash
# Start Application A
$ python app_a.py
Allocating 16GB of GPU memory...
Allocated successfully. GPU memory available: 8GB remaining

# Start Application B
$ python app_b.py
Allocating 8GB of GPU memory...
Error: CUDA out of memory. Tried to allocate 8GB (8GB free)
```

Wait, there's 8GB free. Why did it fail?
Because Application B doesn't just need 8GB of data.
It needs memory for:

- CUDA context overhead (~300-500MB)
- Kernel code
- Temporary buffers
- Driver allocations

But even if we had a bit more memory, there's a bigger issue.
Each application maintains its own CUDA context with its own memory

allocations.

The critical point is this: while the contexts are separate, they all draw from the same physical memory pool.

When Application A allocates 16GB, that memory is reserved for its context and stays allocated as long as the context exists.

Application B can't use that 16GB since it's locked to Application A.

This is why your 16GB + 8GB scenario fails: the sum of all active contexts' memory must fit within the physical GPU memory.

But here's a subtle detail that trips people up.

Each application's memory allocation is static from when it allocated its memory when it started.

But during kernel execution, they might use temporary memory for intermediate results:

bash

```bash
# Both apps allocated their memory upfront
App A: 16GB allocated
App B: 7GB allocated
Total: 23GB used, 1GB free

# During kernel execution
Time T1: App A kernel running - needs 500MB temporary space (fails - not enough free)
Time T2: App B kernel running - runs successfully within its 7GB
```

The memory allocations are fixed per context, but kernels might need additional temporary space during execution.

If that space isn't available, the kernel fails even though the context "owns" its allocated memory.

To understand this better, let's look at a concrete example.

Consider this simple computation from Chapter 1:

two-kernels.py

```
# This Python code runs on CPU
import cupy as cp

# Create arrays on GPU - each allocates memory
x = cp.array([1, 2, 3])      # Allocates memory for x
y = cp.array([4, 5, 6])      # Allocates memory for y

# This triggers a kernel launch on the GPU
z = x + y  # Kernel 1: addition kernel executes

# This triggers ANOTHER kernel launch
w = x - z  # Kernel 2: subtraction kernel executes
```

Here's what's happening with memory:

1. When we create `x` and `y`, CuPy allocates GPU memory for each array
2. When we compute `z = x + y`, we need memory to store the result
3. The variable `z` isn't just a temporary computation—it's actual data that needs GPU memory
4. When we compute `w = x - z`, we need even more memory for the result

Let's see this in action with actual memory tracking:

memory.py

```
import cupy as cp

# Check initial GPU memory
print(f"Initial GPU memory: {cp.cuda.MemoryPool().used_bytes()} bytes")

# Allocate some large arrays
x = cp.ones((1000, 1000), dtype=cp.float32)  # 4MB
print(f"After x: {cp.cuda.MemoryPool().used_bytes() / 1024**2:.2f} MB")

y = cp.ones((1000, 1000), dtype=cp.float32)  # Another 4MB
print(f"After y: {cp.cuda.MemoryPool().used_bytes() / 1024**2:.2f} MB")

# Compute z = x + y (needs memory for result)
z = x + y
print(f"After z = x + y: {cp.cuda.MemoryPool().used_bytes() / 1024**2:.2f} MB")

# Compute w = x * z (needs MORE memory)
w = x * z
```

```
print(f"After w = x * z: {cp.cuda.MemoryPool().used_bytes() / 1024**2:.2f} MB")

# Output:
# Initial GPU memory: 0 bytes
# After x: 4.00 MB
# After y: 8.00 MB
# After z = x + y: 12.00 MB
# After w = x * z: 16.00 MB
```

Every intermediate result consumes GPU memory that stays allocated as long as the variable exists.

Now imagine this scenario with multiple applications:

```bash
# Application A starts
App A: Allocates 8GB for model weights
App A: Allocates 2GB for input data
App A: Needs 3GB for intermediate computations
Total for App A: 13GB

# Application B tries to start on a 16GB GPU
App B: Allocates 2GB for its model
App B: Allocates 1GB for data
App B: Tries to allocate 2GB for intermediate results
# CRASH: Out of memory (13GB + 2GB + 1GB + 2GB = 18GB > 16GB)
```

Even though App B only "permanently" needs 3GB, the temporary space for intermediate results pushes the total over the limit.

This is why the memory situation is so complex: applications don't just need memory for their data; they need working space for computations.

And GPU memory is all-or-nothing, unlike CPU memory, where the kernel can swap pages to disk or compress them.

Either you have enough space for all allocations (permanent and temporary), or your kernel fails.

Now let's see how Kubernetes tries to manage this chaos.

The Kubernetes Orchestration Problem

Kubernetes elegantly handles CPU and memory sharing.

When you write:

resources.yaml

```yaml
resources:
  requests:
    memory: "2Gi"
    cpu: "500m"       # 0.5 CPU cores
  limits:
    memory: "4Gi"
    cpu: "1"          # 1 CPU core
```

Kubernetes understands fractions, enforces limits through cgroups, and schedules pods based on available resources.

But GPUs break this model entirely.

When you write:

resources.yaml

```yaml
resources:
  limits:
    nvidia.com/gpu: 1
```

You're requesting one entire GPU.

You can't write:

resources.yaml

```yaml
resources:
  limits:
    nvidia.com/gpu: 0.5        # ERROR: must be an integer
    nvidia.com/gpu-memory: 2Gi  # ERROR: no such resource
```

Kubernetes has no native concept of GPU sharing.
So if you want two pods to share a GPU, you need to somehow:

1. Get both pods scheduled to the same node
2. Make both pods see the same physical GPU
3. Prevent Kubernetes from giving that GPU to other pods
4. Track who's using what (even if you can't enforce it)

Doing this manually would be a nightmare.
Let's examine two solutions that automate this orchestration.

KAI-Scheduler's Reservation Pod Strategy

KAI-Scheduler solves GPU sharing through a clever workaround.
The problem it faces: Kubernetes only understands whole GPUs, but you want fractions.
The solution is to use a placeholder to claim the whole GPU, then secretly share it.

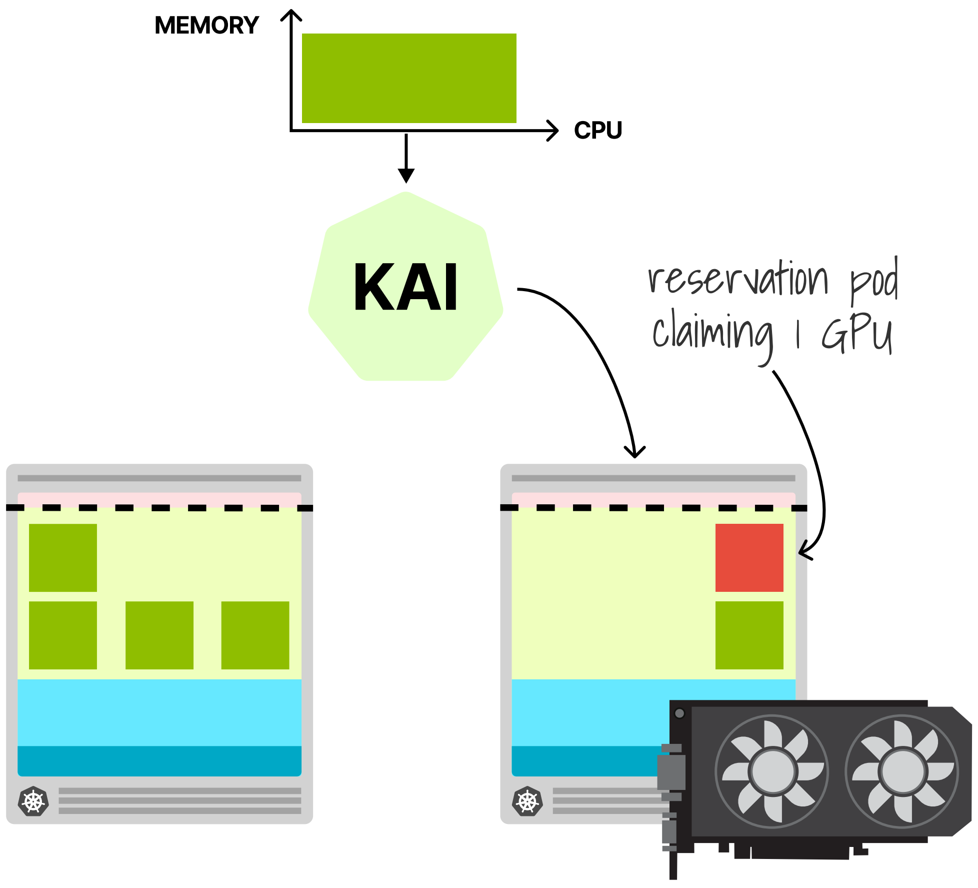


Fig. The KAI scheduler deploys a placeholder pod to reserve the full GPU.

Let's say you want to run an inference service with just 2GB of GPU memory. Since Kubernetes doesn't understand GPU memory requests, KAI-Scheduler invents its own annotation system:

pod.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: inference-service
  annotations:
    gpu-memory: "2000"  # In MiB - KAI's custom annotation
spec:
  schedulerName: kai-scheduler  # Must use KAI, not default scheduler
  containers:
  - name: inference
    image: inference:latest
    # Note: No nvidia.com/gpu in resources!
```

This `gpu-memory` annotation (measured in MiB) is KAI's workaround for Kubernetes' lack of GPU memory awareness.

When you submit this pod, here's what happens behind the scenes.

KAI-Scheduler looks for a node with a GPU with at least 2GB free.

It finds node-1 with a Tesla T4 (16GB total).

KAI-Scheduler creates a special "reservation" pod:

reservation-pod.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: gpu-reservation-node1-abc123
  namespace: kai-resource-reservation
  labels:
    runai-gpu-group: group-xyz-789  # Links this GPU to user pods
spec:
  nodeName: node-1  # Explicitly placed on node-1
  runtimeClassName: nvidia  # Required by KAI
  containers:
  - name: resource-reservation
    image: kai-resource-reservation:latest  # KAI's discovery component
    resources:
      limits:
        nvidia.com/gpu: 1  # Claims the ENTIRE GPU
```

This reservation pod claims the whole GPU from Kubernetes and runs a controller that actively discovers which GPU it received.

The default Kubernetes scheduler sees this GPU as fully occupied and won't schedule anything else.

The next step involves discovering which GPU was assigned.

Here's where it gets clever.

When the reservation pod starts, the kubelet and device plugin assign it a

specific GPU, say `GPU-abc-def-123`.
But KAI-Scheduler doesn't know which physical GPU it got.
The reservation pod runs a KAI controller that:

1. Uses NVML to discover which GPU it was assigned
2. Updates its own pod annotation with the GPU index

```bash
$ nvidia-smi -L Inside the reservation pod's controller
GPU 0: Tesla T4 (UUID: GPU-abc-def-123)

# The controller updates the pod annotation
$ $ kubectl annotate pod gpu-reservation-node1-abc123 \
  run.ai/reserve_for_gpu_index=GPU-abc-def-123
```

Now KAI-Scheduler:

1. Schedules your inference-service pod to node-1 (same node as the reservation)
2. Adds the `runai-gpu-group` label to link it to the reservation pod
3. Sets the environment variable `NVIDIA_VISIBLE_DEVICES=GPU-abc-def-123`
4. Updates its internal tracking: "GPU-abc-def-123 has 2GB allocated, 13GB free"

What happens when another pod requests 4GB of GPU memory?

```yaml
metadata:
  annotations:
    gpu-memory: "4000"  # In MiB
```

KAI-Scheduler:

1. Sees that GPU-abc-def-123 has 13GB free

2. Schedules this pod to node-1
3. Adds the same `runai-gpu-group` label to link it to the existing reservation
4. Sets `NVIDIA_VISIBLE_DEVICES=GPU-abc-def-123` (same GPU!)
5. Updates tracking: "GPU-abc-def-123 has 6GB allocated, 10GB free"

Both pods now see the same GPU.

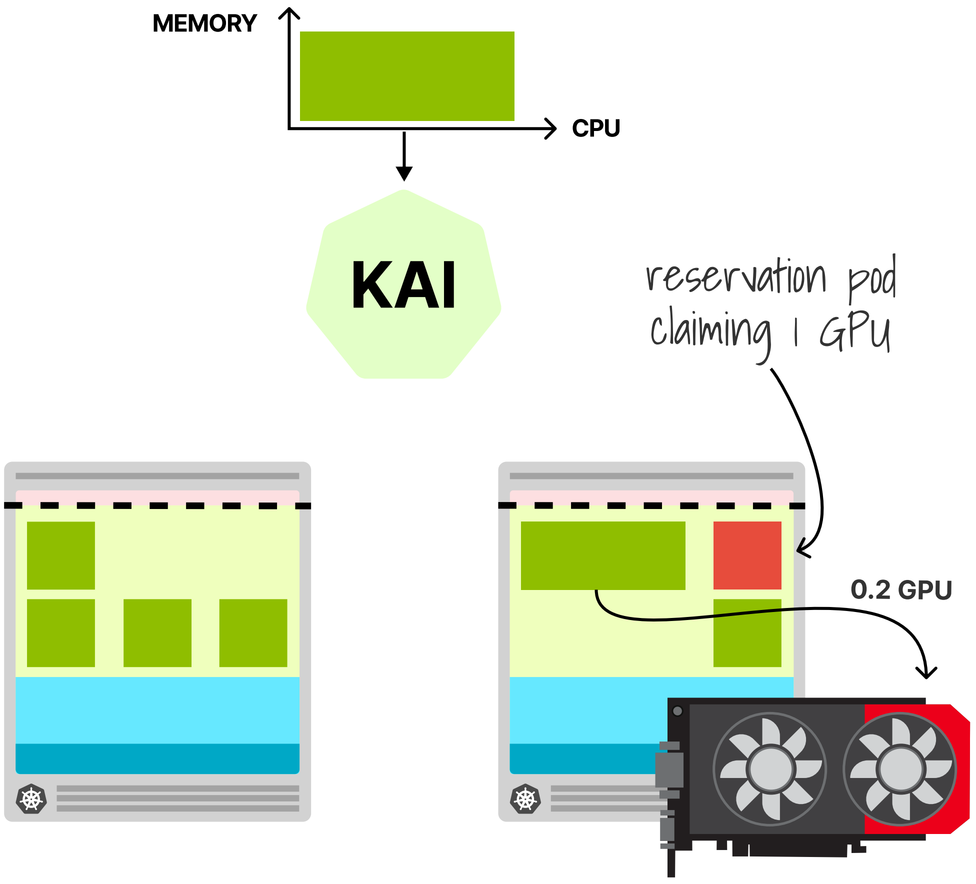


Fig. The KAI scheduler will keep track of the allocations and schedule the pod to the right node.

They take turns running kernels, just like the previous manual example.

KAI: The Clever Parts and the Limitations

What makes KAI-Scheduler clever is how it solves the fundamental mismatch between what users want and what Kubernetes provides.

Users want to request specific amounts of GPU memory, such as 2GB for inference and 8GB for training.

But Kubernetes only understands whole GPUs.

KAI bridges this gap through its reservation pod mechanism.

When you submit a pod requesting 4GB of GPU memory, KAI automatically finds a GPU with sufficient free memory and ensures your pod lands on the same node.

You don't have to think about node affinity or pod placement—KAI handles it all.

KAI tracks how much memory each GPU has allocated and makes smart decisions about where to place new pods.

It can pack many small inference workloads onto one GPU or dedicate most of a GPU to a large training job.

This dynamic sharing adapts to your workload patterns.

But here's what KAI doesn't do, which is crucial to understand.

There's no memory enforcement.

When a pod that requested 2GB decides to allocate 15GB, nothing stops it.

The pod will happily grab all available memory, potentially crashing other workloads.

KAI tracks the allocation in its database but can't prevent overuse.

Similarly, there's no compute isolation.

One pod can launch a kernel that runs for 30 seconds, blocking all other pods from executing.

KAI has no mechanism to interrupt that kernel or ensure fair time-sharing.

And there's no fairness guarantee.

Pods compete for GPU time based on who launches kernels first.

A pod constantly launching kernels will dominate GPU time, starving others regardless of their allocated "share."

KAI-Scheduler is orchestrating sharing, not enforcing it.

It's worth noting that KAI also supports requesting GPU fractions (e.g., `gpu-fraction: "0.5"`) and even multiple fractional GPUs through the `gpu-fractions-num-devices` annotation.

But regardless of how you request resources, the underlying behavior remains the same: orchestration without enforcement.

NVIDIA "Time-Slicing" - The Misleading Name

NVIDIA's device plugin offers something called "time-slicing."

The name suggests that the GPU divides time fairly between users, like CPU time slices, where each process gets 10ms turns.

This is entirely wrong.

Let me show you what it actually does.

You configure time-slicing through the device plugin:

configmap.yaml

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: nvidia-device-plugin-config
  namespace: kube-system
data:
  config.yaml: |
    version: v1
    sharing:
      timeSlicing:
        resources:
```

```yaml
      - name: nvidia.com/gpu
        replicas: 4  # Make 1 GPU look like 4
```

After applying this and restarting the device plugin:

```bash
$ kubectl describe node node-1
Capacity:
  nvidia.com/gpu: 4  # Was 1, now reports 4!
Allocatable:
  nvidia.com/gpu: 4
```

One physical GPU now appears as four GPUs to Kubernetes.
What really happens is that the device plugin is lying to Kubernetes.
There's still one physical GPU, but Kubernetes thinks there are four.
When you deploy four pods, each requesting one GPU:

```yaml
# Pod 1
resources:
  limits:
    nvidia.com/gpu: 1  # Thinks it's getting a whole GPU

# Pod 2
resources:
  limits:
    nvidia.com/gpu: 1  # Also thinks it's getting a whole GPU

# Pods 3 & 4: Same thing
```

Kubernetes happily schedules all four pods to the node.
Each pod believes it has exclusive GPU access.
But check what actually happens:

```bash
$ kubectl exec pod-1 -- nvidia-smi -L
GPU 0: Tesla T4 (UUID: GPU-abc-def-123)

$ kubectl exec pod-2 -- nvidia-smi -L
GPU 0: Tesla T4 (UUID: GPU-abc-def-123)  # Same GPU!

$ kubectl exec pod-3 -- nvidia-smi -L
GPU 0: Tesla T4 (UUID: GPU-abc-def-123)  # Same GPU!

$ kubectl exec pod-4 -- nvidia-smi -L
GPU 0: Tesla T4 (UUID: GPU-abc-def-123)  # Same GPU!
```

All four pods see the same physical GPU.

They're taking turns running kernels, just like in the earlier manual example where `matrix_multiply.py` and `vector_add.py` both ran on the same GPU.

Why "Time-Slicing" Is Misleading

The name "time-slicing" is profoundly misleading and creates dangerous false expectations.

In CPU scheduling, time-slicing means something very specific.

Each process gets a fixed time to run.

When that time expires, the Linux kernel forcibly pauses the process and switches to another.

This ensures fairness: even if one process wants to monopolize the CPU, it can't.

Everyone gets their turn.

GPU "time-slicing" does none of this.

You might expect that with four replicas, each pod gets 250ms time slices in a round-robin fashion.

That's not what happens.

Pods take turns when their kernels complete.

This is the same behavior GPUs have always had.

There's no fair scheduling either.

If Pod A launches a kernel that runs for 5 seconds, Pods B, C, and D wait the full 5 seconds.

Pod A isn't limited to a "fair share" of time, and it can dominate the GPU by launching long-running kernels.

And there's absolutely no preemption.

Once a kernel starts, it runs to completion.

The GPU can't pause it mid-execution to give another pod a turn.

This is a fundamental limitation of GPU architecture, which NVIDIA cannot fix in software.

What NVIDIA calls "time-slicing" should really be called "GPU multiplexing" or "replica mode."

It's simply making one GPU appear as multiple GPUs to Kubernetes, with no actual time management involved.

The misleading name creates false expectations about isolation and fairness that the technology doesn't deliver.

But there's more.

Time-slicing Critical Limitations

With NVIDIA time-slicing, you still can't request specific amounts:

resources.yaml

```
```

```
# You CAN'T do this:
annotations:
  gpu-memory: "4000"  # Time-slicing doesn't support this

# You can only do:
resources:
  limits:
    nvidia.com/gpu: 1  # Still just "one GPU" (really 1/4)
```

Every pod gets the same 1/N share conceptually, though they all see the full GPU in practice.

But here's the worst part: time-slicing has zero memory tracking.

The lack of memory awareness makes time-slicing particularly problematic for production use.

KAI-Scheduler at least maintains an internal state regarding memory usage.

It knows that GPU-abc has 6GB allocated and 9GB free.

Based on this information, it can make intelligent scheduling decisions, avoiding placements exceeding available memory.

Operators gain visibility into usage patterns.

They can query KAI's state to understand how memory is distributed across workloads.

NVIDIA time-slicing provides none of this intelligence.

It has no idea how much memory each pod actually needs.

A pod requesting one of four time-sliced GPUs might need 100MB or 10GB—time-slicing doesn't know or care.

There's no tracking of current usage either.

You can't ask time-slicing "how much memory is Pod A using?" because it doesn't track that information.

All you know is "4 slots occupied" with no further detail about actual resource consumption.

You're flying blind, unable to capacity plan or troubleshoot memory issues until pods start crashing.

This lack of accounting makes time-slicing particularly dangerous in production.

Comparing the Approaches: Same Reality, Different Tricks

Let's be clear about what both solutions actually provide:

Aspect	KAI-Scheduler	NVIDIA "Time-Slicing"
How it tricks Kubernetes	Reservation pod claims whole GPU	Device plugin reports N fake GPUs
Node placement	Automatic (KAI handles it)	Manual (you ensure pods co-locate)
Memory awareness	Yes (tracks via annotations)	No (fixed slots)
Flexible packing	Yes (2GB + 4GB + 8GB pods)	No (N equal slots)
Actual GPU behavior	Context switching	Context switching
Memory isolation	None	None
Compute isolation	None	None
Fair scheduling	None	None

Both enable the same underlying behavior: multiple processes taking turns on one GPU.

Neither provides isolation nor guarantees.

They're just different ways to orchestrate sharing.

A Practical Example: The Scheduler Showdown

Let's see how each approach handles a real scenario.

You have a Tesla T4 (16GB memory) and need to run:

- 2 inference services (2GB each)
- 1 training job (8GB)
- 1 data preprocessing job (3GB)

Total requested: 16GB (looks like it fits!)

Let's deploy those workloads with the KAI scheduler:

```bash
# Deploy all four pods with memory annotations (in MiB)
$ kubectl apply -f inference-1.yaml  # gpu-memory: "2000"
$ kubectl apply -f inference-2.yaml  # gpu-memory: "2000"
$ kubectl apply -f training.yaml       # gpu-memory: "8000"
$ kubectl apply -f preprocessing.yaml # gpu-memory: "3000"

$ kubectl get pods
NAME                READY   STATUS     NODE
inference-1         1/1     Running    node-1
inference-2         1/1     Running    node-1
training            1/1     Running    node-1
preprocessing       1/1     Running    node-1

# All scheduled to the same node, sharing one GPU via the same reservation pod
```

KAI tracked the memory requests and packed them efficiently.

But watch what happens when they run:

```bash
$ kubectl logs training
Epoch 1: Training... Allocated 11GB for gradients
```

```
# Uses more than the 8GB "requested"!

$ kubectl logs preprocessing
ERROR: CUDA out of memory
# Crashed because training used extra
```

KAI orchestrated the sharing but couldn't enforce limits.

Let's repeat the same experiment with time-slicing:

```bash
# With time-slicing configured for four replicas on node-1
# Each pod requests one "GPU"
$ kubectl apply -f inference-1.yaml  # nvidia.com/gpu: 1
$ kubectl apply -f inference-2.yaml  # nvidia.com/gpu: 1
$ kubectl apply -f training.yaml      # nvidia.com/gpu: 1
$ kubectl apply -f preprocessing.yaml # nvidia.com/gpu: 1

$ kubectl get pods -o wide
NAME             READY   STATUS    NODE
inference-1      1/1     Running   node-1
inference-2      1/1     Running   node-1
training         1/1     Running   node-1
preprocessing    1/1     Running   node-1
```

All four pods scheduled successfully on node-1 because time-slicing made it advertise 4 GPUs.

But here's the critical difference: there's no memory awareness whatsoever.

Unlike KAI, which at least attempts to track memory requests through annotations, time-slicing does not have a concept of GPU memory.

Each pod thinks it has exclusive access to a whole GPU.

The training job will happily try to allocate all 15GB of memory, potentially causing the other pods to crash with out-of-memory errors.

```bash
```

```
$ kubectl logs training
Loading model... Allocated 12GB for weights and gradients
Training epoch 1/100...

$ kubectl logs preprocessing
ERROR: CUDA out of memory
# Crashed because training consumed most of the GPU memory
```

Orchestrating sharing

Both solutions are orchestration layers, not isolation mechanisms.

They solve the scheduling problem: "How do we trick Kubernetes into allowing multiple pods on one GPU?"

They don't solve the sharing problem: "How do we ensure pods play nicely together?"

This is why both approaches work best in high-trust environments where:

- Applications are well-behaved
- Developers coordinate resource usage
- Memory requirements are predictable
- Kernel execution times are reasonable

In hostile or multi-tenant environments, you need actual enforcement.

That's where solutions like HAMi (with CUDA API interception) or MIG (with hardware partitioning) come in.

But those are stories for the next chapter.

Key Takeaways

Understanding GPU sharing in Kubernetes requires separating orchestration from execution:

GPUs have always supported multiple processes through context switching. No special features needed—the NVIDIA driver handles kernel queueing automatically.

Memory is not partitioned. When pods share a GPU, they share the entire memory pool. Each can use all available memory when their kernels run.

Neither KAI nor "time-slicing" provides isolation. They orchestrate which pods share which GPUs. The actual sharing behavior—taking turns without guarantees—is identical.

Names can deceive. NVIDIA's "time-slicing" doesn't slice time. It multiplies the GPU count. Understanding what technologies actually do, versus what their names suggest, is crucial for setting correct expectations.

The next time someone asks if you can run multiple workloads on one GPU, the answer is yes.

The question is whether you can do it safely and fairly.

For that, we need to look beyond orchestration to actual enforcement mechanisms.

Chapter 4

Hardware Isolation and Enforcement

In this chapter, you will learn:

- How to achieve true parallel GPU execution instead of time-sharing with MIG (Multi-Instance GPU)
- Why software enforcement through HAMi works on any GPU, not just expensive ones
- When hardware isolation justifies the cost versus software alternatives
- How to implement both MIG and HAMi
- The operational trade-offs between hardware partitioning and API interception

Your ML team's training job just crashed. Again.

The inference team swears their service only needs 2GB of GPU memory, but somehow it grabbed all 16GB and killed everything else.

KAI-Scheduler dutifully tracked that allocation. Time-slicing divided the GPU into neat slots.

Neither prevented the crash.

This is the trust problem: orchestration without enforcement isn't enough for everyone.

When "please play nice" doesn't work, you need isolation.

Let's explore two radically different approaches—one blessed by NVIDIA's hardware and the other a magnificent hack that works on anything.

The Fundamental Difference: Parallel vs Sequential

Before diving into solutions, we need to understand a critical distinction.

Everything we've discussed so far, e.g., manual sharing, time-slicing, and KAI, involves sequential execution:

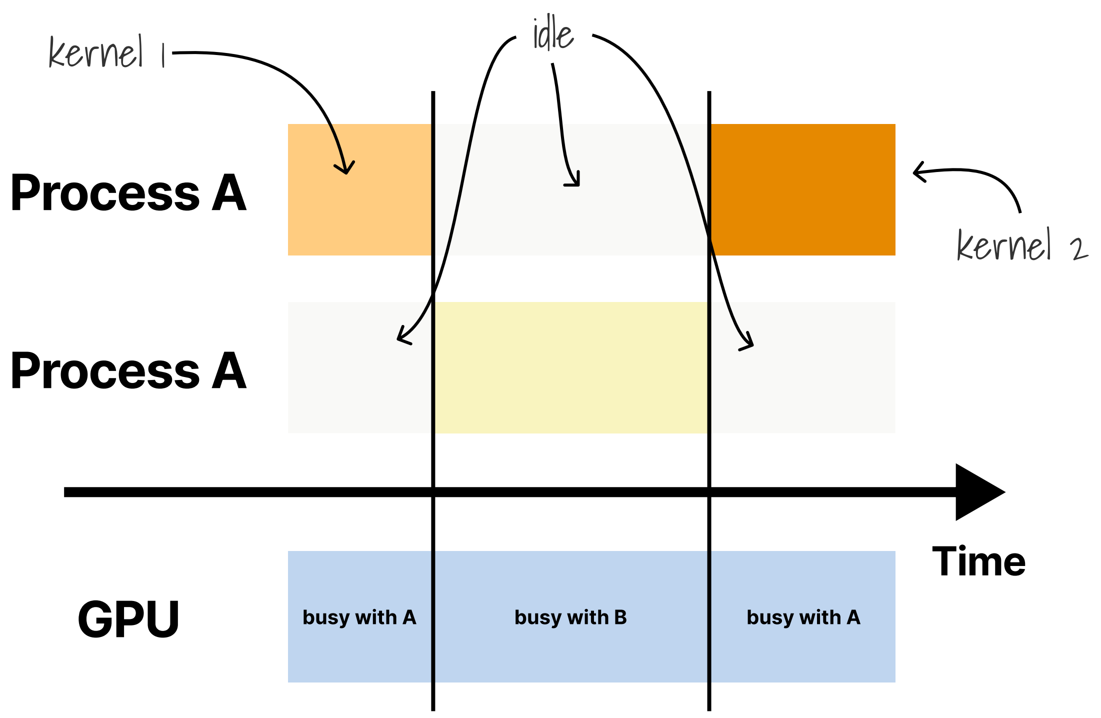


Fig. Kernel on the GPU take turns

Pods take turns.
When Pod A's kernel runs, Pod B waits.
This creates a fundamental problem: one badly behaved pod can monopolize the GPU.
Watch what happens with a long-running kernel:

```bash
# Pod A launches a 30-second matrix operation
kubectl logs pod-a
Starting 30-second computation...

# Pod B tries to run inference
```

```
$ kubectl logs pod-b
Waiting for GPU...
Waiting for GPU...
Waiting for GPU...
[Finally runs after 30 seconds]
```

Even with perfect memory isolation, Pod B is stuck waiting.
But what if pods could run simultaneously?

MIG: When NVIDIA Decided to Fix This in Silicon

Multi-Instance GPU (MIG) represents NVIDIA's hardware solution to the sharing problem.
Instead of time-sharing one GPU, MIG physically divides it into separate, parallel processors.
Think of a GPU as a large office building.
Time-slicing is like having one conference room that teams book in shifts.
Everyone waits for their turn.
MIG is like putting up permanent walls to create seven separate offices, each with a door, desk, and phone.
Teams work simultaneously, not sequentially.
Let me show you what this means with our Tesla T4:

```bash
$ nvidia-smi -L
GPU 0: Tesla T4 (UUID: GPU-abc-123-def)

$ nvidia-smi mig -lgip
```

```
No MIG-capable GPUs found.
```

Our T4 doesn't support MIG.
This is the first harsh reality: MIG only works on specific, expensive GPUs.
Let's switch to an H100 to see MIG in action:

bash

```
# On an H100-equipped node
$ nvidia-smi -L
GPU 0: NVIDIA H100 80GB HBM3 (UUID: GPU-xyz-789)

# Check MIG capability
$ nvidia-smi mig -lgip
+-----------------------------------------------------------------------------+
| GPU Instance Profiles:                                                      |
| GPU   Name             ID    Instances   Memory     P2P    SM    DEC   ENC  |
|                              Free/Total   GiB              CE    JPEG  OFA  |
|=============================================================================|
|   0  MIG 1g.10gb       19     7/7        9.75       No     16     0     0   |
|      MIG 2g.20gb       14     3/3        19.62      No     32     1     0   |
|      MIG 3g.40gb        9     2/2        39.50      No     48     2     0   |
|      MIG 4g.40gb        5     1/1        39.50      No     64     2     0   |
|      MIG 7g.80gb        0     1/1        79.00      No    112     7     0   |
+-----------------------------------------------------------------------------+
```

The H100 can be split multiple ways.
Notice the 1g.10gb profile shows "7/7" instances available—the GPU can be divided into seven independent slices.

Why Seven Slices?

The number seven isn't arbitrary.
The H100 SXM5 has 132 Streaming Multiprocessors (SMs).
To understand this, we need to clarify what SMs actually are.

Part I discussed how GPUs pack thousands of ALUs (Arithmetic Logic Units) to achieve massive parallelism.

But ALUs don't exist in isolation—they're organized into clusters called Streaming Multiprocessors.

Think of an SM as a mini-processor within the GPU.

Each SM contains its resources: CUDA cores (the actual ALUs), shared memory, registers, and scheduling units.

On the H100 SXM5, each SM contains 128 CUDA cores for single-precision operations, meaning the GPU has 132 × 128 = 16,896 total ALUs for FP32 math.

When we say MIG divides the H100 into seven slices, each slice gets a portion of these SMs.

The smallest slice (1g.10gb) gets 16 SMs on the H100 PCIe variant. This means 16 × 128 = 2048 CUDA cores per slice—still plenty of parallel compute power.

NVIDIA's architecture allows these to be cleanly divided into seven isolated units:

- Each instance gets a dedicated portion of SMs
- Each instance gets dedicated memory controllers
- Each instance gets separate cache partitions

The hardware can support the smallest viable independent compute unit while maintaining complete isolation.

The seven base units can be combined flexibly, as long as the total doesn't exceed seven:

Valid combinations on H100:

- 7× 1g.10gb (7 small instances)
- 3× 2g.20gb + 1× 1g.10gb (3 medium + 1 small = 7 units)
- 2× 3g.40gb + 1× 1g.10gb (2 large + 1 small = 7 units)
- 1× 7g.80gb (1 full GPU = 7 units)
- 1× 4g.40gb + 1× 3g.40gb (4 units + 3 units = 7 units)

Invalid combination:

- 4× 2g.20gb (Would need 8 units, only 7 available)

Each profile consumes a specific number of base units:

- 1g.10gb = 1 unit
- 2g.20gb = 2 units
- 3g.40gb = 3 units
- 4g.40gb = 4 units
- 7g.80gb = 7 units

Think of it like dividing a pizza into sevenths.
You can group the slices however you want, but you can't create an eighth slice.

Creating MIG Instances

Let's partition an H100 into multiple instances:

bash

```bash
# Enable MIG mode (requires GPU reset)
$ nvidia-smi -mig 1
Enabled MIG Mode for GPU 00000000:00:04.0
All done.

# Create instances: 2 small (1g.10gb) and 1 large (3g.40gb)
$ nvidia-smi mig -cgi 19,19,9 -C
Successfully created GPU instance ID  1 on GPU  0 using profile MIG 1g.10gb (ID 19)
Successfully created GPU instance ID  2 on GPU  0 using profile MIG 1g.10gb (ID 19)
Successfully created GPU instance ID  3 on GPU  0 using profile MIG 3g.40gb (ID  9)
```

Now check what we've created:

bash

```bash
$ nvidia-smi -L
GPU 0: NVIDIA H100 80GB HBM3 (UUID: GPU-xyz-789)
  MIG 1g.10gb  Device 0: (UUID: MIG-aaa-111)
  MIG 1g.10gb  Device 1: (UUID: MIG-bbb-222)
  MIG 3g.40gb Device 2: (UUID: MIG-ccc-333)
```

Three completely independent GPUs from one physical card.
Each has its own UUID, memory, and compute resources.
Let's run workloads on different MIG instances:

```bash
# Terminal 1: Run on MIG instance 0
$ CUDA_VISIBLE_DEVICES=MIG-aaa-111 python matrix_multiply.py
Starting matrix operations...
Consistent performance: 1.20s, 1.21s, 1.20s, 1.21s

# Terminal 2: Run on MIG instance 1 simultaneously
$ CUDA_VISIBLE_DEVICES=MIG-bbb-222 python inference.py
Running inference...
Consistent latency: 45ms, 44ms, 46ms, 45ms

# Terminal 3: Run on MIG instance 2
$ CUDA_VISIBLE_DEVICES=MIG-ccc-333 python training.py
Training model...
Stable throughput: 1000 samples/sec
```

All three run in parallel with zero interference.
No waiting.
No performance degradation.
Check GPU utilization:

```bash
$ nvidia-smi
+-----------------------------------------------------------------------------+
| MIG devices:                                                                |
|=============================================================================|
| GPU  GI  CI  MIG |         Memory-Usage |        Vol|      Shared         |
|      ID  ID  Dev |           BAR1-Usage | SM     Unc|     CE ENC DEC OFA  |
|                  |                      |        Err|                     |
|=============================================================================|
|  0    1   0   0  |   3847MiB /  4864MiB | 14      0 |  0   0    0    0   |
|  0    2   0   1  |   1923MiB /  4864MiB | 14      0 |  0   0    0    0   |
|  0    3   0   2  |  15234MiB / 20096MiB | 42      0 |  2   0    2    0   |
+-----------------------------------------------------------------------------+
```

Each instance shows independent utilization.
This is true parallel execution, not turn-taking.
MIG integrates beautifully with Kubernetes because each instance appears as a distinct resource.
You can configure MIG manually without any operator, but it requires careful coordination:

bash

```bash
# Manual MIG setup (on the node)
$ nvidia-smi -mig 1  # Enable MIG mode
$ nvidia-smi mig -cgi 19,14,9 -C  # Create profiles
# Then manually configure device plugin, update node labels, etc.
```

The GPU Operator automates this entire process:

gpu-cluster-policy.yaml

```yaml
apiVersion: nvidia.com/v1
kind: ClusterPolicy
metadata:
  name: gpu-cluster-policy
spec:
  mig:
    strategy: mixed  # Allows different MIG profiles per node
  devicePlugin:
    config:
      default: default
      name: device-plugin-config
```

Configure MIG profiles per node:

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: device-plugin-config
  namespace: gpu-operator
data:
  default: |
    version: v1
    mig-strategy: none
  mig-mixed: |
    version: v1
    mig-strategy: mixed
    mig-devices:
      "1g.5gb": 2
      "3g.20gb": 1
```

Label nodes to apply MIG configuration:

```bash
$ kubectl label node gpu-node-1 nvidia.com/mig.config=mig-mixed
```

The GPU Operator automatically:

1. Enables MIG mode
2. Creates the specified instances
3. Configures the device plugin to expose them

Using MIG Instances in Pods

Each MIG profile appears as a native Kubernetes resource:

bash
```bash
$ kubectl describe node gpu-node-1
Allocatable:
  nvidia.com/mig-1g.5gb:   2
  nvidia.com/mig-3g.20gb:  1
```

Request them like any other resource:

inference-pod.yaml
```yaml
apiVersion: v1
kind: Pod
metadata:
  name: inference-pod
spec:
  containers:
  - name: inference
    image: inference:latest
    resources:
      limits:
        nvidia.com/mig-1g.5gb: 1  # Specific MIG profile
```

No annotations.
No scheduler tricks.
Just native Kubernetes resources with hardware isolation.
But notice the crucial difference from KAI-Scheduler:

kai-pod.yaml
```yaml
# KAI-Scheduler - flexible but software-only
annotations:
  gpu-memory: "3500"  # Can request any amount

# MIG - hardware-enforced but fixed profiles
resources:
  limits:
    nvidia.com/mig-1g.5gb: 1   # 5GB memory, 14 SMs
    # Can't request mig-1g.3.5gb - doesn't exist!
```

With MIG, you're limited to predefined profiles.
Want 3.5GB of memory?
Too bad: choose between 1g.5gb (too small) or 2g.10gb (too large).
The hardware dictates your options.
Let's prove MIG actually enforces limits:

memory-test-pod.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: memory-test
spec:
  containers:
  - name: test
    image: cuda:latest
    resources:
      limits:
        nvidia.com/mig-1g.5gb: 1  # Gets ~5GB limit
    command: ["python"]
    args: ["-c", "
import cupy as cp
print('Attempting to allocate 8GB...')
try:
    x = cp.zeros((1024, 1024, 1024), dtype=cp.float64)  # 8GB
    print('Success?!')
except Exception as e:
    print(f'Failed as expected: {e}')
"]
```

Run it:

bash

```bash
$ kubectl apply -f memory-test-pod.yaml
$ kubectl logs memory-test
```

```
Attempting to allocate 8GB...
Failed as expected: cupy.cuda.memory.OutOfMemoryError: Out of memory allocating 8,589,934,592 bytes
```

The hardware enforced the 10GB boundary.
No API interception needed.
No trust required.

MIG's Achilles' Heel: Cost and Architecture Trade-offs

MIG is nearly perfect, except for several critical considerations:

```
# H100 80GB on Azure (as of 2024)
Standard_NC40ads_H100_v5: ~$6.98/hour = ~$5,026/month

# Tesla T4 for comparison
Standard_NC4as_T4_v3: ~$0.53/hour = ~$382/month
```

An H100 costs over 13× more than a T4.
But there's a deeper architectural question: should you share one H100 or buy seven Tesla T4s?
Consider this comparison:

Approach	One H100 with MIG	Seven T4 GPUs
Cost	$5,026/month	$2,674/month
Total Memory	80GB shared	112GB total (7×16GB)
Pod Placement	All on same node	Across 7 nodes
Network Bandwidth	3.35TB/s (shared)	320GB/s each
Failure Domain	One node failure affects all	Isolated failures
Scheduling Flexibility	Fixed to one node	Can spread across zones

The H100 looks impressive on paper with its massive memory bandwidth, but this comparison reveals a counterintuitive truth.

When you enable MIG on an H100, you're essentially paying premium prices to turn it into seven smaller GPUs.

Each MIG slice gets hardware-enforced isolation—a genuine advantage for untrusted workloads.

But consider what else the H100 offers: simpler management (one GPU to maintain), better performance for tightly-coupled workloads that need inter-GPU communication, and impressive 3.35TB/s memory bandwidth (though remember, it's shared across all instances).

Now look at what seven T4s give you.

First, the economics are compelling—with seven T4s at $2,674/month versus an H100 at $5,026/month, you save $2,352.

But the advantages go beyond cost.

With seven separate nodes, you gain true failure isolation.

When one T4 node crashes, six workloads continue running.

When your H100 node fails, everything stops.

You also get geographic flexibility.

Those seven T4s can span availability zones, giving you real disaster recovery.

Your H100 sits in one rack, one zone, one failure domain.

The scheduling flexibility matters too.

With distributed T4s, Kubernetes can bin-pack workloads across nodes, potentially achieving better overall utilization.

The H100 forces all workloads onto one node, creating hotspots and limiting your options.

There's also the uncomfortable fact about hardware support.

MIG only works on NVIDIA's most expensive GPUs: the A100 and H100 families, plus the A30.

Your existing fleet of T4s?

None supports MIG.

This creates a stark choice: investing in H100s for hardware isolation, or finding another way.

For many organizations, the answer is clear.

They already have T4, can't justify H100 prices, and still need some form of isolation.

HAMi: The Software Enforcement Revolution

HAMi takes an entirely different approach: when hardware cannot enforce isolation, it polices every GPU call in software and applies its own rules.

To see how this works, we first need to understand how Linux actually loads and executes GPU programs.

When you run a CUDA application, it doesn't talk directly to the GPU.

It calls functions in the CUDA library, which then talks to the kernel driver:

```
Your Python Code → CUDA Runtime (libcudart.so) → CUDA Driver (libcuda.so) → Kernel Driver → GPU
```

Linux provides a powerful mechanism called `LD_PRELOAD` that lets you inject your library before any other libraries load.
Here's how it works:

1. Linux loads your program
2. Before loading CUDA libraries, it checks `LD_PRELOAD`
3. If set, it loads that library first
4. That library can replace (intercept) any function from libraries loaded later

Think of it this way: When your program says, "call `cuMemAlloc`," Linux normally routes that to the real CUDA library.
HAMi uses Linux's `LD_PRELOAD` mechanism to inject its library (`libvgpu.so`) that intercepts CUDA Driver API calls directly:

```
Your Code → CUDA Runtime → [HAMi intercepts here] → CUDA Driver → GPU
```

This is similar to how cgroups work with system calls.
When a process tries to allocate memory, the kernel checks cgroups to see if it's allowed.
HAMi does the same thing for GPU calls instead of system calls.
Let me show you this in action:

```bash
# Without HAMi - direct CUDA access
$ python -c "import torch; x = torch.zeros(2048, 2048, 512).cuda()"  # 8GB allocation
# Success - grabbed 8GB

# With HAMi - intercepted and limited
```

```
$ export LD_PRELOAD=/path/to/libvgpu.so
$ export CUDA_DEVICE_MEMORY_LIMIT=4g  # 4GB limit
$ python -c "import torch; x = torch.zeros(2048, 2048, 512).cuda()"
# RuntimeError: CUDA out of memory
```

HAMi tracks every memory allocation like an accountant tracking expenses. When your application tries to allocate GPU memory, here's what happens:

1. **Application requests memory**: "I need 8GB for this tensor"
2. **HAMi intercepts**: Before the request reaches CUDA Driver, HAMi checks:
   - What's this pod's memory limit? (4GB)
   - How much has it already been used? (1GB)
   - Would this allocation exceed the limit? (1GB + 8GB > 4GB)
3. **HAMi decides**:
   - If within limits: Pass the request to the CUDA driver
   - If over limits: Return an out-of-memory error immediately
4. **Tracking**: If allocation succeeds, HAMi records it in a table:
   - Memory address → Size allocated
   - Running total for this process
5. **Cleanup**: When memory is freed, HAMi updates its records

This is precisely how cgroups work for CPU and memory.
When a process calls `malloc()` to request system memory, the kernel intercepts this request and checks the process's cgroup limits.
If the allocation would exceed the limit, the kernel denies it immediately.
Similarly, when a process calls `cuMemAlloc()` to request GPU memory, HAMi intercepts this call and checks the pod's configured limit.
If the request exceeds that limit, HAMi returns an out-of-memory error without calling the real CUDA function.
The parallel is deliberate, HAMi brings cgroup-like enforcement to GPUs.
But there's a clever trick for applications that check available memory:
When an application asks "How much GPU memory is available?", HAMi lies:

- Real GPU: "16GB total, 12GB free"

- HAMi responds: "4GB total, 3GB free" (based on pod's limit)

The application thinks it's running on a smaller GPU and adjusts accordingly. This prevents well-behaved applications from even trying to exceed their limits.

Compute Throttling: Token Bucket in Action

Memory is straightforward—allow or deny allocations.

Compute is trickier.

You can't stop a running kernel and limit how many SMs it uses once launched.

HAMi sets the `CUDA_DEVICE_SM_LIMIT` environment variable when you configure `nvidia.com/gpucores: 30`, representing the target SM utilization percentage (compute throughput), not a limit on the number of SMs used.

But unlike memory enforcement, compute throttling requires a more sophisticated approach.

HAMi implements compute throttling through a token bucket algorithm, similar to network rate limiting.

When HAMi intercepts a kernel launch through `cuLaunchKernel`, it doesn't just pass it through, it implements a gating mechanism.

Here's how it actually works:

1. **Token Calculation**: Each kernel launch consumes tokens based on the number of grid blocks (gridDimX × gridDimY × gridDimZ)
2. **Token Check**: Before launching, HAMi checks if enough tokens are available
3. **Blocking Wait**: If insufficient tokens, the thread blocks until tokens accumulate
4. **Token Consumption**: Once available, tokens are deducted, and the kernel

4. launches
5. **Continuous Replenishment**: A background thread replenishes tokens based on actual GPU utilization

In simplified form, the implementation code looks like this:

```
// When a kernel launch is requested
function intercepted_kernel_launch(kernel, grid_dimensions, block_dimensions):
    // Calculate how many tokens this kernel needs (based on grid size)
    token_cost = grid_dimensions  // Actually just grid, not grid × block

    // Wait until we have enough tokens
    while (available_tokens < token_cost):
        wait()  // Block the thread

    // Consume the tokens atomically (using CAS in real implementation)
    available_tokens = available_tokens - token_cost

    // Launch the actual kernel
    return real_cuda_kernel_launch(kernel, grid_dimensions, block_dimensions)

// Background thread continuously adjusting tokens
function token_manager():
    while (true):
        current_gpu_utilization = measure_gpu_utilization()
        target_utilization = CUDA_DEVICE_SM_LIMIT  // e.g., 30%

        if (current_gpu_utilization < target_utilization):
            // We're under quota, add more tokens
            tokens_to_add = calculate_increase(target - current)
            available_tokens = available_tokens + tokens_to_add
        else:
            // We're over quota, reduce tokens
            tokens_to_remove = calculate_decrease(current - target)
            available_tokens = max(0, available_tokens - tokens_to_remove)

        sleep(interval)  // Configurable check interval
```

The clever part is the feedback loop.

A background thread continuously monitors actual GPU utilization through NVML (NVIDIA Management Library) and adjusts token generation.

This creates a self-regulating system where pods naturally converge toward their allocated compute percentage over time.

Despite this sophistication, the core limitation persists: if a single kernel runs for 30 seconds, it still runs for 30 seconds.

HAMi can't interrupt it mid-execution.

However, the token bucket ensures that after consuming tokens for that expensive kernel, the pod must wait for tokens to replenish before launching the next one.

This works well for workloads with many small kernels but struggles with applications that launch occasional massive kernels.

HAMi provides configuration options to tune this behavior:

- **Default Policy**: Normal token bucket with adaptive replenishment
- **Force Policy**: Strictly limit utilization below the configured percentage
- **Disable Policy**: Turn off compute limiting entirely

> Note: If `nvidia.disablecorelimit` is set to true in the HAMi configuration, compute limiting is disabled globally regardless of pod settings.

The enforcement remains "best effort" since it can't violate the laws of GPU physics, but it's more sophisticated than simple delays between kernel launches.

HAMi installation is straightforward since it works on any GPU:

bash

```bash
# Add HAMi helm repository
$ helm repo add hami-charts https://project-hami.github.io/HAMi/

# Install HAMi
$ helm install hami hami-charts/hami \
  --namespace kube-system \
  --set scheduler.enabled=true \
  --set devicePlugin.enabled=true
```

Verify installation:

bash

```
$ kubectl get pods -n kube-system | grep hami
hami-device-plugin-ds-xvnbg   1/1     Running   0          2m
hami-scheduler-7b9d8954-kjplw  1/1     Running   0          2m
```

Once installed, HAMi extends Kubernetes with new resource types:

```yaml
# pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: inference-with-limits
spec:
  containers:
  - name: inference
    image: inference:latest
    resources:
      limits:
        nvidia.com/gpu: 1             # Need GPU access
        nvidia.com/gpumem: 4000       # 4GB memory limit (enforced!)
        nvidia.com/gpucores: 30       # 30% compute (best effort)
```

Unlike KAI's annotations, these are actual Kubernetes resources.
HAMi strictly enforces memory limits (`nvidia.com/gpumem`) but provides best-effort throttling for compute (`nvidia.com/gpucores`).
Let's prove HAMi actually blocks overallocation:

```yaml
# hami-test.yaml
apiVersion: v1
kind: Pod
metadata:
  name: hami-test-1
spec:
  containers:
  - name: memory-limited
    image: tensorflow/tensorflow:latest-gpu
    resources:
      limits:
        nvidia.com/gpu: 1
        nvidia.com/gpumem: 2048  # 2GB limit
    command: ["python", "-c"]
    args: ["
import tensorflow as tf
print('Allocated memory limit: 2048MB')
print('Attempting to allocate 3GB...')
try:
    # Try to allocate 3GB
    x = tf.random.normal([1024, 768, 1024])
    print('Should not see this message!')
except Exception as e:
    print(f'Blocked by HAMi: {e}')
"]
---
apiVersion: v1
kind: Pod
metadata:
  name: hami-test-2
spec:
  containers:
  - name: compute-limited
    image: tensorflow/tensorflow:latest-gpu
    resources:
      limits:
        nvidia.com/gpu: 1
        nvidia.com/gpumem: 4096
        nvidia.com/gpucores: 20  # 20% compute limit
    command: ["python", "-c"]
    args: ["
import time
import tensorflow as tf
print('Running with 20% compute limit')
start = time.time()
# Heavy computation
for i in range(100):
    x = tf.random.normal([2048, 2048])
    y = tf.matmul(x, x)
duration = time.time() - start
print(f'Duration: {duration:.2f}s (throttled by HAMi)')
"]
```

Deploy and check:

```bash
$ kubectl apply -f hami-test.yaml

$ kubectl logs hami-test-1
Allocated memory limit: 2048MB
Attempting to allocate 3GB...
Blocked by HAMi: ResourceExhaustedError: OOM when allocating tensor

$ kubectl logs hami-test-2
Running with 20% compute limit
Duration: 45.23s (throttled by HAMi)
# Without throttling, this would take ~10s
```

HAMi successfully enforced both memory and compute limits on our Tesla T4.

The Reality of Software Enforcement

HAMi is impressive, but it's still a hack.

Here's what can go wrong.

The most immediate problem is version sensitivity.

CUDA evolves constantly, adding new APIs and deprecating old ones.

When NVIDIA releases CUDA 12.5 with a new memory allocation function like `cuMemAllocAsync_v2`, HAMi might not recognize it:

```bash
# New CUDA version breaks interception
$ nvidia-smi
Driver Version: 535.104.12    CUDA Version: 12.5
```

```
$ kubectl logs hami-device-plugin
ERROR: Unknown CUDA function cuMemAllocAsync_v2
WARNING: Cannot intercept new allocation API
```

Suddenly, applications using the new API bypass HAMi's controls entirely.
`LD_PRELOAD` is a convenience, not a security boundary.
A malicious user with enough knowledge can unset the environment variable:

```bash
# Malicious user bypasses LD_PRELOAD
$ unset LD_PRELOAD
$ python gpu_memory_grab.py
# Successfully grabbed all GPU memory
```

Or they could use static linking to avoid the dynamic loader entirely.
Or call system calls directly without going through libraries.
The interception model assumes cooperation, and it falls apart against deliberate circumvention.
Though HAMi implements additional PID tracking and shared memory regions to make bypass more difficult than simply unsetting LD_PRELOAD.
Performance overhead is another reality.
Every CUDA call now goes through an extra layer of inspection:

```
# Benchmark without HAMi
Inference latency: 12.3ms

# Benchmark with HAMi
Inference latency: 14.1ms
# ~15% overhead from interception
```

For training workloads that make thousands of CUDA calls per second, this overhead compounds.

But here's the thing: despite these limitations, HAMi provides real enforcement where hardware solutions don't exist.

For many teams, that's good enough.

Production Considerations

Running MIG or HAMi in production reveals operational challenges that don't appear in proof-of-concepts.

The most frustrating MIG limitation is profile lock-in.

Once you've configured MIG profiles, changing them requires a complete node drain:

```bash
# Cannot change MIG profiles without draining node
$ kubectl drain gpu-node-1
$ nvidia-smi mig -dci
$ nvidia-smi mig -dgi
# Recreate with new profiles
# Redeploy workloads
```

Imagine you've configured your H100 with seven 1g.10gb instances for inference workloads.

Six months later, your team must run larger models requiring 3g.40gb profiles.

You can't just reconfigure.

You must evict all running workloads, destroy the existing MIG instances, create new ones, and redeploy everything.

This means scheduling maintenance windows, migrating workloads, and accepting downtime in a production cluster.

The rigidity extends to resource utilization.

MIG profiles must divide evenly into seven units, leading to stranded resources:

```yaml
# This works perfectly
mig-devices:
  "1g.10gb": 7  # Uses all 7/7 units

# This leaves resources stranded
mig-devices:
  "3g.40gb": 2  # Only uses 6/7 units
  # 1/7 of your GPU (14% of an H100!) sits idle
```

The last seventh of your GPU—representing about $718/month of an H100's cost—can not be allocated to any workload.

You're paying for resources you literally cannot use.

HAMi's software approach brings different operational headaches.

Driver updates become minefields:

```bash
# After driver update
$ kubectl logs hami-device-plugin
Warning: CUDA 12.5 detected, HAMi tested up to 12.2
Some interceptions may fail

# Solution: Wait for HAMi update or pin driver version
```

Every NVIDIA driver update potentially breaks HAMi's interception.

You're forced to choose between staying on old drivers (which lack performance improvements and bug fixes) and risking HAMi failures.

Many teams maintain a compatibility matrix, testing each driver version extensively before rolling it out.

Monitoring becomes problematic.

Your traditional tools show one reality, HAMi shows another:

```bash
# nvidia-smi shows physical usage
$ nvidia-smi
Memory-Usage: 15GB / 16GB  # Looks nearly full!

# But HAMi's view is different
$ kubectl exec hami-monitor -- hami-status
Pod A: 2GB used of 4GB limit
Pod B: 3GB used of 4GB limit
Pod C: 4GB used of 4GB limit
Pod D: 6GB used of 8GB limit  # Over-provisioned!
# Total: 15GB allocated, but 20GB of limits!
```

This over-provisioning is intentional when `nvidia.deviceMemoryScaling` > 1, allowing memory overcommit similar to Linux RAM.

Which monitoring system do you trust?

How do you set alerts?

These aren't just technical problems; they're organizational challenges that require training, documentation, and careful operational procedures.

The Evolution of GPU Sharing

We've come far from "one pod, one GPU."

The journey started with manual sharing—multiple containers accessing the same GPU without coordination.

Then came orchestration through MPS, time-slicing, and tools like KAI-Scheduler.

HAMi introduced software enforcement, bringing controlled sharing to any GPU.

MIG delivered true hardware isolation with parallel execution.

But whether you choose MIG's hardware isolation or HAMi's software enforcement, you face the same operational challenge: how do you know if it's working?

How do you monitor GPU utilization across different sharing mechanisms?

How do you optimize allocation when every approach measures resources differently?

That's where our journey continues—into GPU observability and optimization.

Key Takeaways

MIG provides true hardware isolation with parallel execution, but only on expensive GPUs. It's the gold standard, offering guaranteed resources and zero interference.

HAMi offers software enforcement through clever API interception, working on any GPU. It's the practical choice for existing infrastructure, providing real protection with acceptable overhead.

The trust level drives the decision. High-trust environments might not need any enforcement, while low-trust environments require hardware isolation. Most fall somewhere in between.

Performance isn't everything. MIG gives you better utilization through parallel execution, but if your workloads are small and sequential anyway, the expensive hardware might not pay off.

The next time someone says you need H100s for multi-tenancy, ask them about their trust model first.

A simpler solution that is good enough for your actual requirements might save you hundreds of thousands of dollars.

Chapter 5

Monitoring GPU Clusters

In this chapter, you will learn:

- Why nvidia-smi, Kubernetes metrics, and actual GPU usage tell three different stories about the same hardware
- How GPU utilization percentages mislead you into thinking busy GPUs are productive GPUs
- Why zombie processes can hold GPU memory hostage even after pods are deleted
- How to use DCGM to bridge the gap between what you think is happening and what's really happening
- The patterns that predict GPU failures before they crash your production workloads
- Why teams waste 60-70% of their GPU budget on resources that sit idle or underutilized

Imagine your monitoring dashboard shows all green.
GPU nodes are healthy.
Pods are running.
Yet your ML team is screaming that nothing works.

```bash
$ kubectl get pods
NAME                READY   STATUS     RESTARTS   AGE
training-job-42     1/1     Running    0          3h
inference-svc-1     1/1     Running    0          5h
inference-svc-2     1/1     Running    0          5h

$ nvidia-smi
GPU 0: Tesla T4, 15360MiB, 42°C
  2156MiB / 15360MiB used
  GPU-Util: 78%
```

Everything looks fine.
Except that the training job hasn't progressed in two hours.
The inference service is returning errors.
And there are 47 pods stuck in the Pending state that your monitoring doesn't

show.

Welcome to GPU monitoring, where what you see is rarely what's happening.

The Three Views of GPU Reality

GPU monitoring isn't like CPU monitoring.

The kernel knows everything about CPUs—usage, allocation, and scheduling.

With GPUs, you have three completely disconnected views of reality.

The first layer is what nvidia-smi Shows: the Physical Reality.

Let's start with what everyone checks first:

```bash
$ nvidia-smi
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 535.104.12             Driver Version: 535.104.12   CUDA Version: 12.4     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name            Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf     Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. | MIG M.   |
|=================================+======================+======================+===========|
|   0  Tesla T4                Off  | 00000000:00:04.0 Off |                    0 | N/A   N/A |
| N/A   42C    P0     28W /   70W |   8234MiB / 15360MiB |     78%      Default | N/A      |
+---------------------------------+----------------------+----------------------+-----------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                         GPU Memory Usage |
|=======================================================================================|
|    0   N/A  N/A      2847    C     python train.py                           4234MiB   |
|    0   N/A  N/A      3923    C     python inference.py                       2896MiB   |
|    0   N/A  N/A      4102    C     python notebook.py                        1104MiB   |
+---------------------------------------------------------------------------------------+
```

This looks informative.

It's lying to you.

That 78% GPU utilization? It's not what you think.

GPU utilization in nvidia-smi measures the percentage of time at least one kernel was executing.

Not how much work got done.

Watch this deceptive behavior in action:

terrible_gpu_code.py

```python
import cupy as cp
import time

while True:
    # Create a tiny array with just one element
 x = cp.array([1.0])

    # Perform a trivial GPU operation: add 1 to the single element
 y = x + 1  # This launches a GPU kernel that takes ~1 microsecond

    # The CPU thread sleeps for 999 microseconds
    # The GPU sits idle during this time
 time.sleep(0.000999)
```

Here's what's happening in this example.

Every loop iteration launches a GPU kernel to add 1 to a single floating-point number.

In each millisecond, the GPU works for 1 microsecond and idles for 999 microseconds.

That's 0.1% actual utilization.

But watch what nvidia-smi reports when you run this:

bash

```bash
$ python terrible_gpu_code.py &
[1] 2847

$ nvidia-smi --query-gpu=utilization.gpu --format=csv,noheader
87

# 87% utilization!
# But we're doing almost nothing
```

In this example:

According to `nvidia-smi`, the GPU is "busy" 87% of the time but

accomplishes virtually nothing.
But what actually happened?
How can 0.1% real work show as 87% utilization?

How nvidia-smi Actually Measures

To understand the lie, we must realize how `nvidia-smi` works.
The GPU driver doesn't continuously monitor GPU activity.
Instead, it samples at intervals—typically every 1/6 second (about 166ms).
During each sampling window, it asks a simple binary question: Did any kernel run during this period?

```
Time:     0ms                            166ms                          332ms
Kernels:  [50μs kernel]----wait----[2μs]----wait----[100μs]----wait----
Sample:   <-------- Sample 1 -------->  <-------- Sample 2 -------->
Result:   "Activity detected"            "Activity detected"
```

In this example:

- Sample 1: A 50-microsecond kernel ran → Entire 166ms window marked as "active"
- Sample 2: A 100-microsecond kernel ran → Entire 166ms window marked as "active"
- nvidia-smi reports: 100% utilization (2 active samples / 2 total samples)

But what actually happened?

- Total kernel runtime: 50μs + 2μs + 100μs = 152μs
- Total time elapsed: 332ms
- Real utilization: 0.152ms / 332ms = 0.046%

nvidia-smi reported 100% utilization.

The GPU actually computed for 0.046% of the time.

This is why GPU utilization is misleading—it measures whether the GPU was used during each sample, not how much work actually got done.

A program launching tiny kernels continuously shows 100% utilization while accomplishing almost nothing.

The GPU is "busy" but not productive.

The Three-Layer Memory

GPU memory is more complex than it appears.

There are actually three levels:

1. **Free memory**: Not claimed by any process
2. **Reserved memory**: Claimed by CUDA but not yet used
3. **Allocated memory**: Actually storing data

Here's what happens when you run a CUDA program:

memory.py

```python
import cupy as cp

# Before any GPU operations
# GPU has 16GB total, all free

array1 = cp.zeros((512, 512, 512), dtype=cp.float32)  # 512MB
# CUDA reserves a 2GB pool (default behavior)
# But only allocates 512MB for actual data
```

Check nvidia-smi:

bash

```
$ nvidia-smi --query-gpu=memory.used,memory.free --format=csv,noheader
512 MiB, 15848 MiB  # Shows only allocated, not reserved
```

But try to allocate 15GB:

memory.py

```
>>> array2 = cp.zeros((1920, 1024, 1024), dtype=cp.float64)  # 15GB
MemoryError: out of memory
```

Why? Because CUDA already reserved 2GB.
The available memory is actually:

- Total: 16GB
- Reserved by CUDA: 2GB
- Available: 14GB
- You tried to allocate: 15GB
- Result: Failure

Unfortunately, `nvidia-smi` can't show reserved memory directly.
You need to check inside the process:

memory.py

```
pool = cp.get_default_memory_pool()
print(f"Allocated: {pool.used_bytes() / 1e9:.2f} GB")   # 0.51 GB
print(f"Reserved: {pool.total_bytes() / 1e9:.2f} GB")   # 2.00 GB
```

The Zombie Process Problem

Here's the worst part:

```bash
$ kubectl delete pod training-job-42
pod "training-job-42" deleted

$ nvidia-smi
| No running processes found                                                  |

# But...
$ nvidia-smi --query-gpu=memory.used --format=csv,noheader
4234 MiB

# Memory still occupied!
```

The process is gone.
The pod is deleted.
But 4GB of GPU memory is still allocated.
Check deeper:

```bash
$ fuser -v /dev/nvidia0
                     USER        PID ACCESS COMMAND
/dev/nvidia0:        root     kernel mount /dev
                     65534     27341 F...m python <defunct>
```

A zombie process holding GPU memory.

Why Zombie Processes Hold GPU Memory

When a process using GPU resources terminates, the cleanup dance between CUDA and the kernel matters more than you might think.

In the normal flow, your Python script finishes its work and exits gracefully.

The Python runtime calls destructors, CUDA cleanup routines run, and the driver receives proper deallocation requests.

The GPU memory returns to the pool, ready for the next workload.

But containers rarely exit gracefully in production.

A pod exceeds its CPU limit and gets OOMKilled.

A node experiences memory pressure and evicts pods.

An impatient developer runs `kubectl delete pod --force --grace-period=0`.

In all these scenarios, the kernel sends a SIGKILL signal, which doesn't wait for cleanup.

It terminates the process immediately, ripping it out of memory without giving it a chance to clean up.

From the kernel's perspective, the process is gone.

Its CPU time is reclaimed, its system memory is freed, and its file descriptors are closed.

But the GPU driver lives in a different world.

It's still waiting for that process to make cleanup calls that will never come.

The driver maintains its own process tracking table, separate from the kernel's process table.

When Process ID 27341 allocated 4GB of GPU memory, the driver recorded that allocation.

Now Process 27341 is dead, but the driver doesn't know.

It's waiting for an explicit deallocation call.

This is fundamentally different from how CPU memory works.

The GPU driver operates in a different space and doesn't have the exact cleanup mechanisms.

The fixes for zombie GPU memory are all painful, which is why this problem persists in production clusters.

Your first option is to restart the GPU driver entirely.

You unload the kernel module with `rmmod nvidia`, then reload it with `modprobe nvidia`.

This nuclear option works, forcing the driver to release all memory allocations and start fresh.

But it also terminates every GPU workload on the node.

Every pod using that GPU crashes.

Your second option is even more drastic: reboot the entire node.

This guarantees a clean slate, but at the cost of evacuating all pods, losing all local state, and causing minutes of downtime while the node comes back up and pods are rescheduled.

A node with 8 GPUs running critical workloads could mean thousands of dollars in lost compute time and angry users.

Sometimes you get lucky with `nvidia-smi --gpu-reset`.

This command attempts to reset a specific GPU without affecting others.

But it only works if no processes currently use that GPU, including the zombie process the driver still thinks is alive.

The error message "GPU is in use" mocks you as you stare at the empty process list.

This is why GPU memory leaks are so insidious in Kubernetes.

They accumulate slowly, eating away at your available memory until pods start failing with out-of-memory errors on GPUs that appear to be memory-free.

While you observe this from the hardware side, what Kubernetes thinks differs quite a bit.

Kubernetes has its own view of GPU resources:

```bash
$ kubectl describe node gpu-node-1
```

```
Capacity:
  nvidia.com/gpu:  4

Allocatable:
  nvidia.com/gpu:  4

Allocated resources:
  nvidia.com/gpu:  4
```

Four GPUs allocated.

But wait, this is a single T4 GPU with time-slicing configured for 4 replicas.

However, Kubernetes thinks there are four independent GPUs.

Check what's really allocated:

```bash
$ kubectl get pods -A -o custom-columns=\
  NAME:.metadata.name,\
  GPU_REQUEST:.spec.containers[*].resources.limits.nvidia\\.com/gpu
NAME                   GPU_REQUEST
training-job-1         1
training-job-2         1
inference-svc          1
notebook-3             1
```

Four pods each think they have a dedicated GPU.

They're all sharing the same physical T4.

Kubernetes has no idea.

With MIG, It Gets Worse.

On a node with MIG configured:

```bash
$ kubectl describe node gpu-node-2
Capacity:
  nvidia.com/mig-1g.5gb:   2
  nvidia.com/mig-2g.10gb:  1
  nvidia.com/mig-3g.20gb:  1

Allocated resources:
  nvidia.com/mig-1g.5gb:   2
  nvidia.com/mig-2g.10gb:  1
  nvidia.com/mig-3g.20gb:  0
```

Kubernetes sees four different resource types.

But here's where MIG's promise of isolation collides with Kubernetes' simplistic resource model.

When you request a MIG slice, Kubernetes knows you want "one unit of nvidia.com/mig-1g.5gb" resource.

It schedules your pod to a node that has one available.

The device plugin allocates one of the MIG instances to your pod.

But which one?

Let's trace what actually happens:

```bash
$ kubectl get pod inference-1 -o yaml | grep gpu
  nvidia.com/mig-1g.5gb: "1"

# OK, but which physical MIG instance?
$ nvidia-smi -L
MIG 1g.5gb Device 0: (UUID: MIG-aaa-111)
MIG 1g.5gb Device 1: (UUID: MIG-bbb-222)

# Kubernetes doesn't track the UUID mapping
```

You know pod `inference-1` has been allocated a 1g.5gb MIG slice.

But Kubernetes doesn't record which specific MIG instance—MIG-aaa-111 or MIG-bbb-222—was assigned to that pod.

This mapping exists only in the device plugin's internal state, which isn't exposed through the Kubernetes API.

The missing link between Kubernetes pods and MIG UUIDs creates real operational problems that surface during incidents.

Consider this scenario: A user opens a ticket complaining that their inference service has degraded from 50ms to 200ms response times.

You start investigating:

```bash
# User complains: "My inference is slow!"
# You check their pod:
$ kubectl get pod slow-inference -o yaml | grep gpu
  nvidia.com/mig-1g.5gb: "1"

# You check GPU metrics:
$ nvidia-smi
MIG 1g.5gb Device 0: Utilization 15%, Temperature 45°C
MIG 1g.5gb Device 1: Utilization 99%, Temperature 85°C
```

Now you're stuck.

The pod has a MIG-1g.5gb slice, and you can see two such slices on the node.

One is barely being used—15% utilization, running cool.

The other is completely saturated—99% utilization, approaching thermal limits.

If the pod is on Device 0, the problem isn't GPU saturation.

You must investigate the application code, network latency, or data loading bottlenecks.

If the pod is on Device 1, you've found your smoking gun.

The GPU is maxed out and probably thermal throttling, which explains the 4x latency increase.

But without knowing which MIG instance the pod uses, you can't distinguish between these completely different root causes.

The abstraction that Kubernetes provides (i.e., treating all MIG-1g.5gb instances as identical) breaks down the moment you need to understand what's happening on your hardware.

The third view reveals the biggest lie in GPU resource management: the massive gap between what pods request and what they actually use.

This isn't just a monitoring problem; it's a fundamental mismatch between how developers think about resources and how GPUs consume them.

Let's peek behind the curtain of a typical machine learning pod:

resources.yaml

```yaml
# Pod requests 8GB of GPU memory
resources:
  limits:
    nvidia.com/gpumem: 8192
```

But check actual usage:

bash

```bash
$ kubectl exec training-job -- python -c "
import cupy as cp
print(f'Allocated: {cp.get_default_memory_pool().used_bytes() / 1e9:.2f} GB')
print(f'Reserved: {cp.get_default_memory_pool().total_bytes() / 1e9:.2f} GB')
"

Allocated: 2.34 GB
Reserved: 4.00 GB
```

Three different numbers for the same pod, each telling a different story.

The pod requested 8GB from Kubernetes because the developer remembered that six months ago, their model needed 7.5GB.

They added a buffer "just in case" and never looked back.

The KAI scheduler dutifully reserves 8GB of the GPU's memory capacity for this pod, marking it as unavailable to others.

As far as the scheduler is concerned, this pod owns 8GB.

However, CUDA has its own memory management strategy.

When the application starts, CUDA's memory pool allocator reserves 4GB from the GPU—half of what the KAI scheduler thinks the pod is using.

CUDA is smart about performance, grabbing a chunk of memory upfront to avoid the overhead of frequent allocations.

Then there's what the application actually uses: 2.34GB.

This is the real data (e.g., the model weights, gradients, and intermediate tensors) that the training job actually needs.

The rest sits empty.

This three-layer discrepancy creates a cascade of waste.

Other pods that need 3GB of GPU memory can't be scheduled to this node because KAI thinks 8GB is taken.

The GPU has 4GB of free memory that no one can use.

The developer thinks they're being responsible by requesting what they might need.

Everyone loses.

When you multiply this waste across dozens of GPUs, you're burning hundreds of thousands of dollars on unused capacity that looks fully allocated.

The Cascade Effect of GPU Hoarding

GPU hoarding doesn't just waste GPU resources.

It creates a domino effect that wastes everything else on the node.

This is the hidden cost that makes GPU clusters so expensive to operate.

Consider a typical GPU node in your cluster:

- 64 CPU cores
- 256GB RAM
- 4 Tesla T4 GPUs

This is a balanced configuration, designed to support GPU workloads that also need CPU for data preprocessing and memory for dataset caching.

Then a small pod arrives:

resources.yaml

```yaml
resources:
  requests:
    cpu: "2"
    memory: "8Gi"
  limits:
    nvidia.com/gpu: 1
```

A lightweight pod that needs minimal CPU and memory but requires GPU acceleration.

But this tiny pod just made 25% of your GPU capacity unavailable.

And here's where the cascade begins.

Let's examine the damage:

```bash
$ kubectl describe node gpu-node-1
Allocatable:
  cpu:      64
  memory:   256Gi
  gpu:      4

Allocated:
  cpu:      2      # 3% used
  memory:   8Gi    # 3% used
  gpu:      1      # 25% used

Remaining:
  cpu:      62     # 97% wasted
  memory:   248Gi  # 97% wasted
  gpu:      3      # Can only fit 3 more GPU pods
```

The pod uses 3% of the CPU and 3% of the memory, but 25% of the GPU.

This massive imbalance means the remaining resources can't be efficiently utilized.

Sure, you could schedule CPU-only pods to use those 62 remaining cores.

However, most organizations separate GPU nodes from CPU nodes to simplify management and cost tracking.

GPU nodes are expensive: you want them running GPU workloads, not generic applications.

The situation gets worse as more small pods arrive.

If similar lightweight pods take all 4 GPUs:

```bash
Total used:      8 CPU, 32GB RAM, 4 GPU
Total available: 64 CPU, 256GB RAM, 4 GPU
Waste:           56 CPU (87%), 224GB RAM (87%)
```

You're now wasting 87% of your CPU and RAM capacity.

These resources are allocated to the node, paid for, and powered on, but they sit idle because the GPUs are the bottleneck.

At $2,000/month for the node, you're effectively burning $1,740/month on compute capacity that can't be used.

Not because the hardware is broken.

Not because the scheduler is inefficient.

But because GPUs are treated as monolithic, unshareable resources.

This is why GPU utilization metrics can be so misleading.

Your dashboards might show "100% GPU allocation," and you think you need more GPUs.

But what you really have is an allocation problem: small pods hoarding entire GPUs while leaving massive amounts of CPU and memory stranded.

DCGM: Bridging the Reality Gap

NVIDIA's Data Center GPU Manager (DCGM) is the missing link between these three views.

It provides per-process, per-container GPU metrics that actually reflect usage.

First, deploy the DCGM exporter:

```bash
$ helm repo add nvidia https://nvidia.github.io/dcgm-exporter/helm-charts
$ helm install dcgm-exporter nvidia/dcgm-exporter \
  --namespace monitoring \
  --set serviceMonitor.enabled=true
```

Verify it's collecting metrics:

```bash
$ kubectl port-forward -n monitoring svc/dcgm-exporter 9400:9400 &
$ curl -s localhost:9400/metrics | grep DCGM_FI_DEV_GPU_UTIL
DCGM_FI_DEV_GPU_UTIL{gpu="0",UUID="GPU-abc-123-def",pod="training-job-1"} 45.2
DCGM_FI_DEV_GPU_UTIL{gpu="0",UUID="GPU-abc-123-def",pod="inference-svc"} 12.8
DCGM_FI_DEV_GPU_UTIL{gpu="0",UUID="GPU-abc-123-def",pod="notebook-3"} 0.0
```

Now we can see per-pod GPU utilization.
DCGM exposes metrics that `nvidia-smi` can't provide.
These are Prometheus-format metrics that your monitoring stack can scrape and visualize:

```
# Real memory usage per container (in bytes)
DCGM_FI_DEV_FB_USED{pod="training-job-1"} 4234567890

# Memory bandwidth utilization (percentage of theoretical max)
DCGM_FI_DEV_MEM_COPY_UTIL{pod="training-job-1"} 67

# Tensor Core utilization for AI workloads
DCGM_FI_PROF_PIPE_TENSOR_ACTIVE{pod="training-job-1"} 23

# SM (Streaming Multiprocessor) activity - actual compute usage
DCGM_FI_PROF_SM_ACTIVE{pod="training-job-1"} 82
```

Each metric tells a different part of the performance story.
Memory bandwidth utilization reveals if you're bottlenecked on data

movement.

Tensor Core activity shows whether your AI workloads use the specialized hardware efficiently.

However, the key insight comes from comparing SM activity to GPU utilization.

SM activity shows what percentage of the GPU's compute units are doing work, not just whether a kernel is resident.

A pod might show 90% GPU utilization in nvidia-smi but only 20% SM activity in DCGM.

This means the GPU is "busy" running kernels, but those kernels aren't using the compute resources efficiently.

For MIG instances, DCGM finally provides the missing link—per-slice metrics with UUID attribution:

```bash
$ curl -s localhost:9400/metrics | grep "MIG-"
DCGM_FI_DEV_GPU_UTIL{UUID="MIG-aaa-111",pod="inference-1"} 67
DCGM_FI_DEV_GPU_UTIL{UUID="MIG-bbb-222",pod="inference-2"} 89
DCGM_FI_DEV_GPU_UTIL{UUID="MIG-ccc-333",pod="training"} 34

DCGM_FI_DEV_FB_USED{UUID="MIG-aaa-111",pod="inference-1"} 2147483648
DCGM_FI_DEV_FB_USED{UUID="MIG-bbb-222",pod="inference-2"} 3221225472
DCGM_FI_DEV_FB_USED{UUID="MIG-ccc-333",pod="training"} 8589934592
```

Finally, we can see which pod uses which MIG instance and how much they consume.

The pod `inference-1` is on MIG-aaa-111, using 67% of its compute and 2GB of memory.

The pod `inference-2` is on MIG-bbb-222, nearly saturated at 89% with 3GB of memory used.

The training job is surprisingly light at 34% utilization but memory-heavy at 8GB.

When someone complains about performance, you can immediately identify which MIG instance they're on and whether it's a GPU problem.

The Allocation vs Utilization Gap

Now that we can see reality, let's measure the waste.

Finding GPU Hoarders

Query Prometheus to find pods holding GPUs but not using them:

Pods

```
(
  sum by (pod, namespace) (
    DCGM_FI_DEV_GPU_UTIL
  ) < 10
)
and
(
  sum by (pod, namespace) (
    kube_pod_container_resource_limits{resource="nvidia.com/gpu"}
  ) > 0
)
```

Results:

```
{namespace="team-ml", pod="notebook-23"} 0
{namespace="team-ml", pod="notebook-24"} 3
{namespace="research", pod="experiment-old"} 0
{namespace="research", pod="debug-session"} 0
```

Four pods holding GPUs are doing almost nothing.
Let's quantify the waste:

GPU

```
sum by (namespace) (
  # GPUs allocated
  kube_pod_container_resource_limits{resource="nvidia.com/gpu"}
  *
  # Waste percentage (100% - utilization)
  (100 - clamp_min(DCGM_FI_DEV_GPU_UTIL, 0.01))
  / 100
  *
  # Cost per GPU per hour
  1.45  # T4 cost on AWS
)
```

Results:

```
{namespace="team-ml"} $67.20/hour
{namespace="research"} $43.50/hour
{namespace="inference"} $8.70/hour
```

The memory gap is often worse than compute, and DCGM gives us the tools to measure it precisely.
Let's compare what pods request versus what they actually use:

bash

```
# Compare requested vs actual memory usage
$ join -t$'\t' \
  <(kubectl get pods -A -o json | jq -r '.items[] |
    select(.spec.containers[].resources.limits."nvidia.com/gpumem" != null) |
    "\(.metadata.namespace)/\(.metadata.name)\t\(.spec.containers[].resources.limits."nvidia.com/gpumem")"' | sort) \
  <(curl -s localhost:9400/metrics | grep DCGM_FI_DEV_FB_USED |
    awk '{print $1}' | sed 's/.*pod="//' | sed 's/".*//' |
    awk '{getline val; print $0 "\t" int(val/1048576)}' | sort)

team-ml/training-1     8192    2340
team-ml/training-2     8192    1890
research/model-test    16384   4530
inference/service-1    4096    1200
```

This shell pipeline joins two data sources:

- Kubernetes API showing what each pod requested (in MB)
- DCGM metrics showing what each pod actually uses (converted from bytes to MB)

The pattern is damning and consistent across every team:

The `training-1` pod requested 8GB but uses only 2.3GB—an overprovisioning factor of 3.5x.

The `training-2` pod also requested 8GB but uses even less at 1.9GB—a 4.3x overprovisioning.

The `model-test` pod went big with a 16GB request but only consumes 4.5GB—3.6x overprovisioned.

Even the "conservative" inference service requested 4GB but uses just 1.2GB—3.4x overprovisioned.

This isn't malicious or incompetent—it's rational behavior in an environment with no memory elasticity.

Developers have been burned before.

They ran a training job that worked fine for 20 epochs, then crashed with an OOM error on epoch 21 when the batch size increased slightly.

They submitted an inference service that handled normal traffic fine, then died during a traffic spike when the batching logic accumulated more requests.

So they learned to overprovision.

Request double or triple what you think you need.

Add a safety margin on top of that.

The result?

Your cluster shows 90% GPU memory allocated, but only 25-30% is being used.

You think you need more GPUs when what you really need is better memory management.

This waste compounds because GPU memory can't be overcommitted like CPU.

If everyone requests 2x CPU cores beyond what they need, Kubernetes can oversubscribe and let the kernel sort it out through time-slicing.

However, with GPU memory, that 8GB request blocks 8GB of physical memory, whether you use it or not.

Monitoring Patterns That Actually Matter

After implementing DCGM, you can finally detect and prevent the failure patterns that plague GPU clusters.

Let's have a look at a few examples.

GPU memory leaks are insidious because GPUs don't have garbage collection like CPU environments.

In Python or Java, unused memory eventually gets reclaimed by the garbage collector.

On GPUs, allocated memory stays allocated until explicitly freed.

A tiny leak compounds over hours or days until the inevitable OOM crash:

Detect

```
rate(DCGM_FI_DEV_FB_USED[1h]) > 0
and
DCGM_FI_DEV_FB_USED / DCGM_FI_DEV_FB_FREE > 0.8
```

This query identifies pods where memory usage is steadily increasing AND they already use 80% of available memory.

The rate function calculates the per-second increase over the past hour.

A positive rate means memory is growing.

Combined with high utilization, you're looking at an imminent OOM.

Convert this insight into an alert that fires before the crash:

gpu-memory-leak.yaml

```yaml
apiVersion: monitoring.coreos.com/v1
kind: PrometheusRule
metadata:
  name: gpu-memory-leak
```

```
spec:
  groups:
  - name: gpu
    rules:
    - alert: GPUMemoryLeak
      expr: |
        rate(DCGM_FI_DEV_FB_USED[1h]) > 1048576
        and
        DCGM_FI_DEV_FB_USED / (DCGM_FI_DEV_FB_USED + DCGM_FI_DEV_FB_FREE) > 0.8
      annotations:
        summary: "GPU memory leak detected on {{ $labels.pod }}"
        description: "Memory usage growing at {{ $value | humanize }}B/hour"
```

This alert triggers when memory grows by more than 1MB/hour and utilization exceeds 80%.

You are warned hours before the crash, which is enough time to restart the pod gracefully or debug the leak.

Without this monitoring, the first sign of a memory leak is a crashed pod and an angry data scientist who lost 12 hours of training progress.

Another example involves Thermal Throttling.

GPUs are designed to protect themselves from heat damage by reducing clock speeds when temperatures get too high.

This throttling can cut performance by 30-50%, making your expensive GPU perform like a cheaper model:

```bash
$ nvidia-smi -q -d TEMPERATURE
GPU Current Temp                   : 83 C
GPU Shutdown Temp                  : 96 C
GPU Slowdown Temp                  : 93 C
GPU Max Operating Temp              : 85 C
```

These thresholds vary by GPU model, but the pattern is consistent.

At 85°C, the GPU starts reducing clock speeds.

At 93°C, it aggressively throttles.

At 96°C, it shuts down entirely to prevent damage.

Your training job, which usually takes 2 hours, now takes 3 hours.

Monitor temperature trends to catch throttling before it impacts production:

```Thermal
DCGM_FI_DEV_GPU_TEMP > 80
and
rate(DCGM_FI_DEV_GPU_TEMP[5m]) > 0
```

This query identifies GPUs that are both hot (>80°C) AND getting hotter (positive rate of change).
These are about to throttle.
The 5-minute rate window catches rapid temperature increases that suggest a cooling problem—maybe a fan failed, or someone blocked an air vent.
When this alert fires, you can migrate workloads to cooler GPUs, investigate cooling issues, or reduce the workload intensity before performance degrades.
Lastly, it's good practice to check for Context Switching.
When too many processes share a GPU through time-slicing, the overhead of switching between them can consume more time than actual computation:

```bash
# Count processes per GPU
$ nvidia-smi --query-compute-apps=pid --format=csv,noheader | wc -l
12

# Check context switch overhead
$ nvidia-smi dmon -s u -c 10
# gpu    sm    mem    enc    dec
    0    45    23      0      0
    0     0     0      0      0  # Context switch
    0    67    45      0      0
    0     0     0      0      0  # Context switch
    0    23    12      0      0
```

The `nvidia-smi dmon` command shows GPU utilization in real-time.
Those rows full of zeros aren't idle time; they're context switches.
The GPU stopped all work, saved the state of one process, loaded the state of

another, then resumed.

Each switch takes milliseconds, but with 12 processes competing, those milliseconds add up.

In this example, the GPU spends 40% of its time switching contexts.

You're getting 60% of the performance you paid for.

This pattern emerges when teams enable time-slicing to improve utilization but don't limit how many pods can share a GPU.

Each pod thinks it has a full GPU, but they're all fighting for time slices like processes on an overloaded CPU.

The solution isn't to turn off sharing, but to monitor and limit concurrent processes:

Detect

```
count by (gpu_id) (
  DCGM_FI_DEV_GPU_UTIL{} > 0
) > 8
```

When more than 8 processes actively use the same GPU, context switching overhead typically exceeds 20%.

That's your signal to either migrate workloads or adjust your time-slicing configuration.

The Reality Check

After implementing proper monitoring, here's what you typically find:

1. **30-40% of GPU allocations are completely idle** (0% utilization)
2. **Another 30% are underutilized** (<20% utilization)
3. **Memory is overprovisioned by 3-4x** (request 16GB, use 4GB)
4. **Interactive notebooks hold GPUs for days** doing minutes of work

5. **Failed experiments continue running** because nobody checks

The monitoring doesn't fix these problems.
But it makes them impossible to ignore.

Key Takeaways

GPU monitoring requires three views: Physical (nvidia-smi), logical (Kubernetes), and actual usage (DCGM). Any single view lies by omission.

DCGM is essential for container-level GPU metrics. Without it, you can't attribute usage to specific pods or namespaces.

Utilization != efficiency. A GPU can be "100% utilized" while doing almost no useful work. SM activity and memory bandwidth matter more.

Memory is chronically overprovisioned. Teams request 3-4x what they need "just in case," wasting expensive resources.

Waste becomes visible only when measured in dollars. Abstract percentages don't motivate change, but weekly cost reports do.

Chapter 6

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.

In fact, many Kubernetes-native APIs (like Deployments, Services, ConfigMaps, etc.) are scoped at the namespace level.

You think, "Perfect, I'll create one namespace per team!"

You then:

- Provision a single Kubernetes cluster
- Add a few CPU and GPU nodes
- Create 10 namespaces, one for each team

This seems to work at first glance:

- You add NetworkPolicies to prevent cross-team communication
- You set ResourceQuotas to avoid resource hogging
- You even add LimitRanges for baseline governance

But here's the catch.

While namespaces work for basic isolation, they come with critical limitations for real-world AI multi-tenancy, e.g.,

1. Limited Tenant Isolation

Namespaces provide weak isolation, leading to risks associated with "noisy neighbors," where workloads from one tenant can negatively impact others.

For example, a team in namespace A created 3000 secrets or overutilised the GPU; it can degrade performance for others because they share the same control plane.

This means that the control plane-level isolation is missing when using Kubernetes namespaces for multi-tenancy.

2. Restricted Tenant Autonomy

Teams can't:

- Define or install their own Custom Resource Definitions (CRDs)
- Deploy isolated GPU operators or custom schedulers
- Independently manage API extensions or admission controllers

You often need controllers tied to different Kubernetes versions, but operators are cluster-scoped, not namespace-scoped.

That means one cluster, one operator version, and every team bound to it, regardless of their requirements.

3. Compliance and Regulatory Challenges

Weak isolation is a non-starter for enterprises with strict compliance, security, or data residency needs.

Namespaces don't allow for:

- Separate audit trails
- Namespace-specific API server policies
- Per-tenant RBAC beyond a certain depth

Even if you can achieve this, it's a complex operation.

1. **Operational Overhead**

You (as the infra team) still hold the keys:

- Every tenant has to route through you for CRD installs, controller upgrades, etc.
- You become the bottleneck for scaling and updates
- Namespaces scale up friction, not flexibility.

TL;DR: Namespaces are necessary but not sufficient.

They're a starting point but not an enterprise-ready solution for AI workloads at scale, especially when:

- GPU isolation is crucial
- CRDs need to be tenant-specific
- Operational independence is a goal
- Compliance is a priority
- Autonomy needs to be at the front

Should we return to the dedicated cluster if we cannot go the namespaces route?

The industry's traditional solution to this problem has been providing dedicated, single-tenant clusters for each team.

While providing the highest level of isolation with separate control and data planes, this approach comes with its own set of significant drawbacks.

Managing a fleet of clusters at scale introduces immense operational overhead, leads to resource fragmentation, and requires complex, third-party multi-cluster tools to orchestrate deployments and policies.

This model forces platform teams to choose between strong isolation and operational efficiency.

That's where vCluster comes in.

vCluster brings lightweight, isolated control planes without the overhead of full clusters.

A virtual cluster is a fully functional, independent Kubernetes cluster that runs as a lightweight workload inside a single host cluster's namespace.

This abstraction solves the core multi-tenancy challenge by providing each tenant with a dedicated control plane, including a private API server, controller manager, and datastore.

From the tenant's perspective, they interact with their own, private Kubernetes environment.

They can deploy their own CRDs, install custom operators, and manage their RBAC rules without any risk of affecting other tenants or the underlying host cluster.

This simple but powerful concept shifts the operational model.

Instead of managing a sprawling fleet of dedicated clusters, platform teams can now manage a single, centralized host cluster, empowering tenant teams to provision their own isolated virtual clusters in seconds.

This shift greatly improves agility, reduces management overhead, and addresses the **too many clusters** problem head-on.

Understanding the vCluster Architecture: A Primer

vCluster is an open source tool that spans the entire multi-tenancy spectrum, giving enterprises and teams the flexibility to implement multi-tenancy in the way that best suits their needs.

Think of it as dividing a Kubernetes cluster into multiple virtual clusters.

Each team can be assigned a virtual cluster with a dedicated kubeconfig file.
This ensures that every team only has access to their own environment.
They cannot see or interfere with other teams' clusters or the host cluster itself.

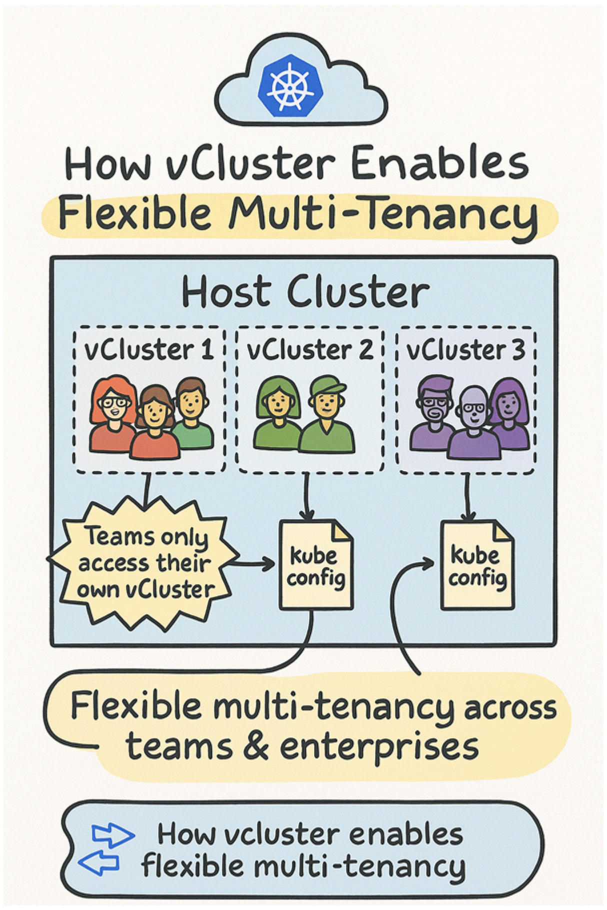

How vCluster Enables Flexible Multi-Tenancy
Host Cluster
vCluster 1
vCluster 2
vCluster 3
Teams only access their own vCluster
kube config
kube config
Flexible multi-tenancy across teams & enterprises
How vcluster enables flexible multi-tenancy

Fig. Diagram showing how vCluster enables flexible multi-tenancy with a host cluster containing three virtual clusters, each with isolated teams accessing their own kubeconfig, demonstrating flexible multi-tenancy across teams and enterprises

The architectural design of a virtual cluster lies in its two primary components, deployed as a single StatefulSet or Deployment inside a host cluster's namespace.
First, there is the **Virtual Control Plane**.
This is the core of the vCluster.
It's a lightweight, fully functional Kubernetes control plane entirely isolated

from the host cluster's control plane.

This virtual control plane comprises its Kubernetes API server, controller manager, and a data store.

By default, it uses an embedded SQLite database for a minimal footprint, but it can be configured to use external data stores like etcd or PostgreSQL for production workloads.

The key benefit is that this isolated control plane and its dedicated data store handle all API requests from a tenant.

This separation provides a strong boundary, ensuring that a tenant's actions, even if they are potentially malicious, cannot impact the host cluster's stability or the workloads of other tenants.

Second, and equally important, is the **Syncer**.

A virtual cluster does not have its own worker nodes or a network.

The **Syncer** is the bridge that emulates a fully functional cluster by replicating resources from the virtual cluster to the host cluster and vice versa.

When a tenant creates a resource like a Pod or a PersistentVolumeClaim inside their vCluster, the **Syncer** component intercepts the request and replicates a transformed version of that resource to the host cluster's namespace.

The host cluster then performs the actual scheduling and workload execution.

The **Syncer** is crucial in preventing resource conflicts on the host by rewriting resource names and namespaces.

For example, a pod named `ollama` in the `ai-team` namespace of a virtual cluster might be renamed on the host to a unique identifier like `ollama-x-ai-team-x-my-vcluster`.

The **Syncer** is more than just a simple copier; it acts as a sophisticated engine that allows platform teams to enforce standards and policies in a decentralized, tenant-specific way.

In traditional multi-tenancy, platform teams often rely on global admission controllers or webhooks to enforce policies, which can be complex to manage and risky to the entire cluster.

The **Syncer**, however, provides a safer and more flexible alternative through its patching capabilities.

A platform administrator can configure a vCluster template to automatically apply "patches" to resources synced from the virtual cluster to the host.

For example, a patch can automatically add a **runtimeClassName** to any pod that requests a GPU resource, ensuring it uses the correct runtime on the host nodes.

This approach empowers the platform team to maintain control over the underlying infrastructure, while simultaneously giving tenants the autonomy to manage their environments and resources without being burdened by global policies they don't need or understand.

The tenant has complete freedom inside their vCluster, but the platform's rules are enforced transparently and securely on the host.

The table below summarizes the core components of a virtual cluster and their functions:

Component	Function	Key Benefit
Virtual Control Plane	Handles all tenant API requests and stores a dedicated state for each vCluster.	Provides strong API-level and CRD isolation, preventing cross-tenant interference.
Syncer	Replicates pods and key resources from the virtual cluster to the host cluster.	Emulates a full cluster without requiring a separate data plane; enables powerful policy enforcement via patches.
Datastore	Stores all Kubernetes resource states for the virtual cluster.	Can be embedded (SQLite) for a lightweight footprint or externalized for high availability and scale.
Scheduler (Optional)	Schedules workloads within the virtual cluster.	Reuses the host cluster scheduler by default to save resources, but can be enabled for custom scheduling logic.

Let's try understanding this differently, using the current architectures and how some modern tooling would help.

Along with multi-tenancy, this enables efficient GPU usage for teams at scale with isolation and low overhead of managing them.

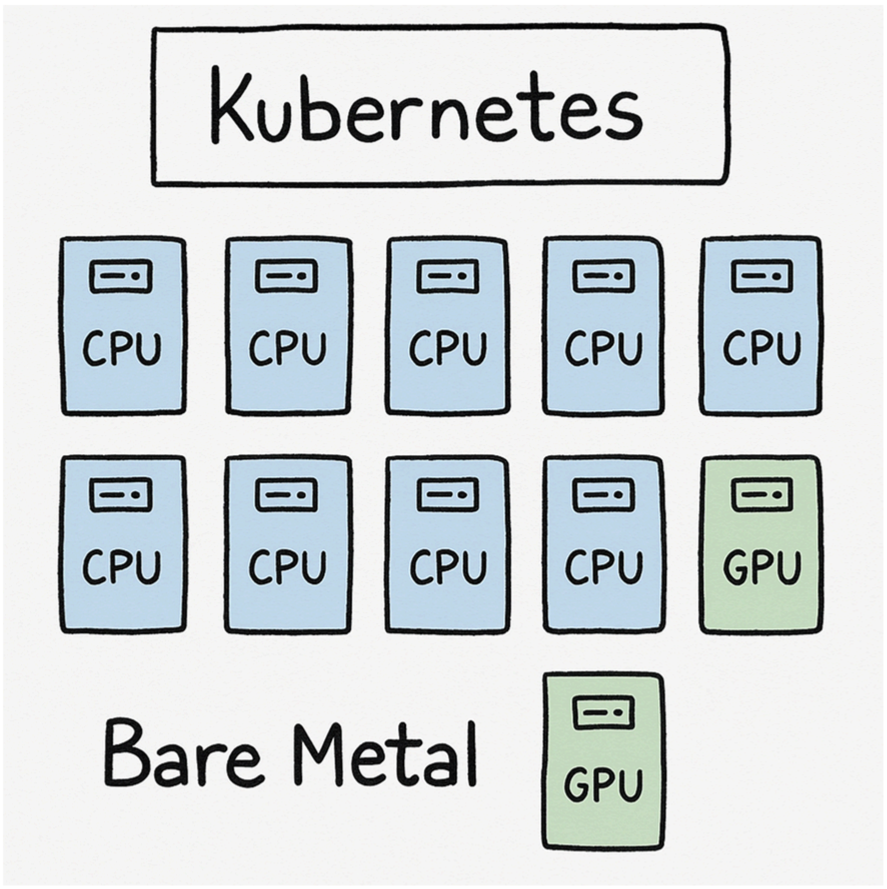


Fig. Infrastructure diagram showing a Kubernetes cluster with 10 bare metal nodes - 9 CPU nodes and 1 GPU node, illustrating the available hardware resources

Above is the bare metal hardware that you have for your teams. The obvious choice is that you would create a Kubernetes cluster from the available nodes.

Since you need to give it to separate teams, you will create multiple Kubernetes clusters (as a first simple pattern).

Even if you are on the cloud, you will create separate clusters and then add **nodepools** to give to each team, which can then deploy their applications.

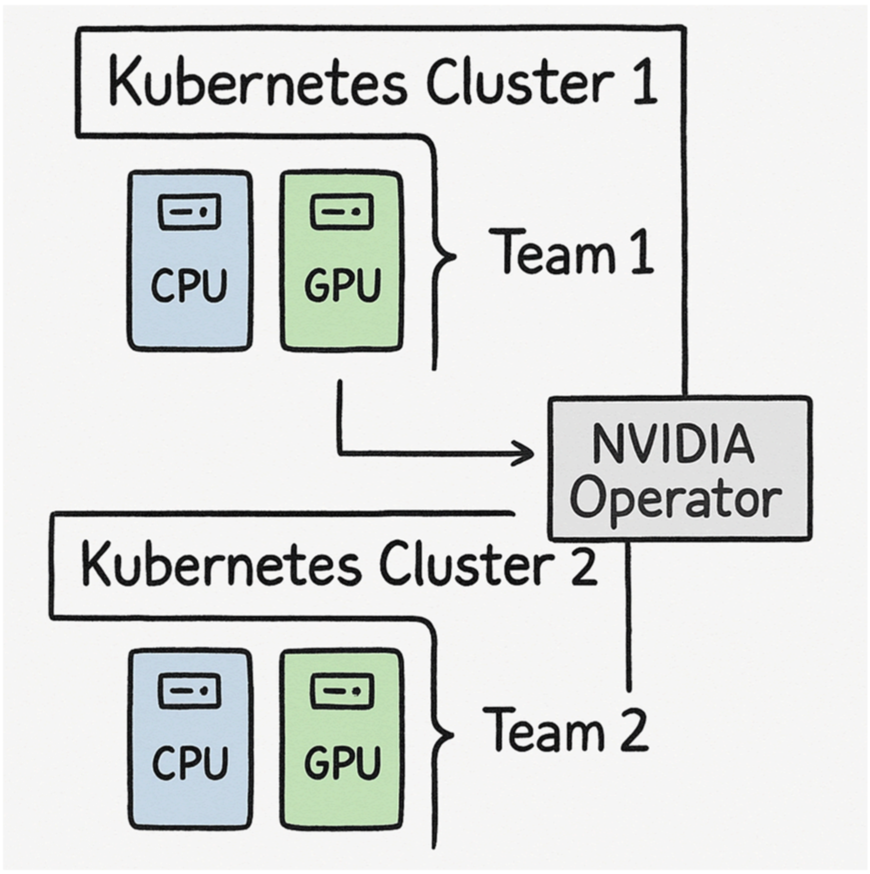


Fig. Architecture diagram showing two separate Kubernetes clusters for Team 1 and Team 2, each with CPU and GPU nodes, where Team 1's cluster includes an NVIDIA Operator

Soon, you realize that your bare metal servers are quickly running out.

To set up just one Kubernetes cluster, you need at least three nodes: one for the control plane, one for CPU workloads, and one for GPU workloads.

With limited capacity, you need a smarter way to effectively divide your existing infrastructure to serve multiple teams.

Before moving on to the namespace-based multi-tenancy model, you'll likely encounter another problem: the default Kubernetes scheduler isn't optimized for AI/ML workloads.

AI jobs aren't your typical web apps.

They need the right GPUs (specific types, counts, MIG or time-sliced), drivers and runtimes (like the NVIDIA Operator and CUDA), and sufficient GPU memory.

They perform best when backed by fast storage like NVMe and high-bandwidth networking.

Efficiency improves with orchestration features such as gang scheduling to start pods together, checkpointing to pause and resume, and fair queues to prevent teams from blocking each other.

In short: AI workloads demand the right hardware, software, infrastructure plumbing, and intelligent scheduling.

But the default Kubernetes scheduler is general-purpose.

It simply sees "1 GPU" as a number and not as a slice, or memory range, or with topology awareness.

This leads to GPU fragmentation and idle silicon.

It also schedules pods one-by-one, not as a gang, so distributed training jobs often half-start and then stall.

Fair sharing across teams is lacking, and the basic preemption knobs are too coarse.

It's not topology or data-aware either (PCIe, NUMA, data locality), so even correctly configured pods can end up on suboptimal nodes and stay "Pending".

The result?

Long queues, poor GPU utilization, and many frustrated engineers wondering, "Why is my job still pending?"

This is where more innovative scheduling projects like Kai Scheduler, Hami, and Kueue come in.

For this book, we'll focus on Kai Scheduler and leave the others for your exploration.

Before diving into the namespace-based multi-tenancy model, let's try to schedule your AI workloads efficiently.

To do so, you'll install Kai Scheduler.

Locate the latest release version on the releases page.

Run the following command after replacing "

" with the desired release version:

bash

```
$ helm upgrade oci://ghcr.io/nvidia/kai-scheduler/kai-scheduler \
  -i kai-scheduler \
  -n kai-scheduler \
  --create-namespace --version <VERSION>
```

What is Kai-Scheduler?

Kai-Scheduler is an open-source, Kubernetes-native scheduler built (and now maintained in the open) by NVIDIA to make AI/ML clusters use GPUs smarter.

Think of it as a companion to the default scheduler, only for workloads that require extra intelligence for GPU queues, fairness, and packing jobs so fewer GPU cycles sit idle.

It's designed for large clusters and lots of jobs/users.

Key things Kai adds on top of default K8s scheduling:

- **Hierarchical queues with fair-share (DRF):** Resources are divided into `org/team/user` buckets with quotas and priorities so no team can overuse GPUs. **Gang/batch scheduling:** This ensures that N pods start together or not at all, which is critical for distributed training.
- **GPU sharing (fractional/time-sliced):** multiple small jobs packed onto one GPU, reducing underutilization and leveraging NVIDIA's time-slicing.
- **Preemption & consolidation:** evicts lower-priority jobs and defragments GPUs to free capacity for high-priority workloads.
- **DRA/ResourceClaims with topology awareness:** better handling of vendor devices and placement constraints.

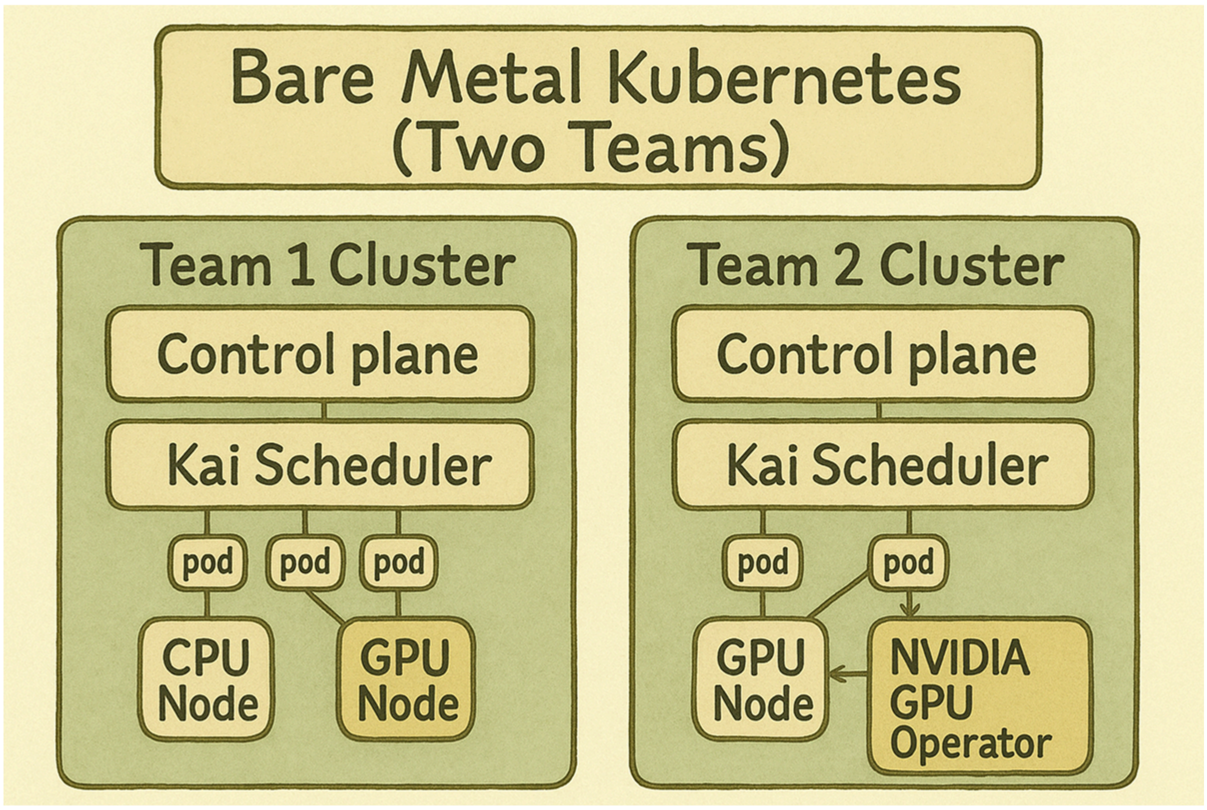


Fig. Bare metal Kubernetes setup for two teams showing Team 1 and Team 2 clusters, each with control plane, Kai Scheduler, pods, and dedicated CPU/GPU nodes, with Team 2 also having NVIDIA GPU Operator

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.

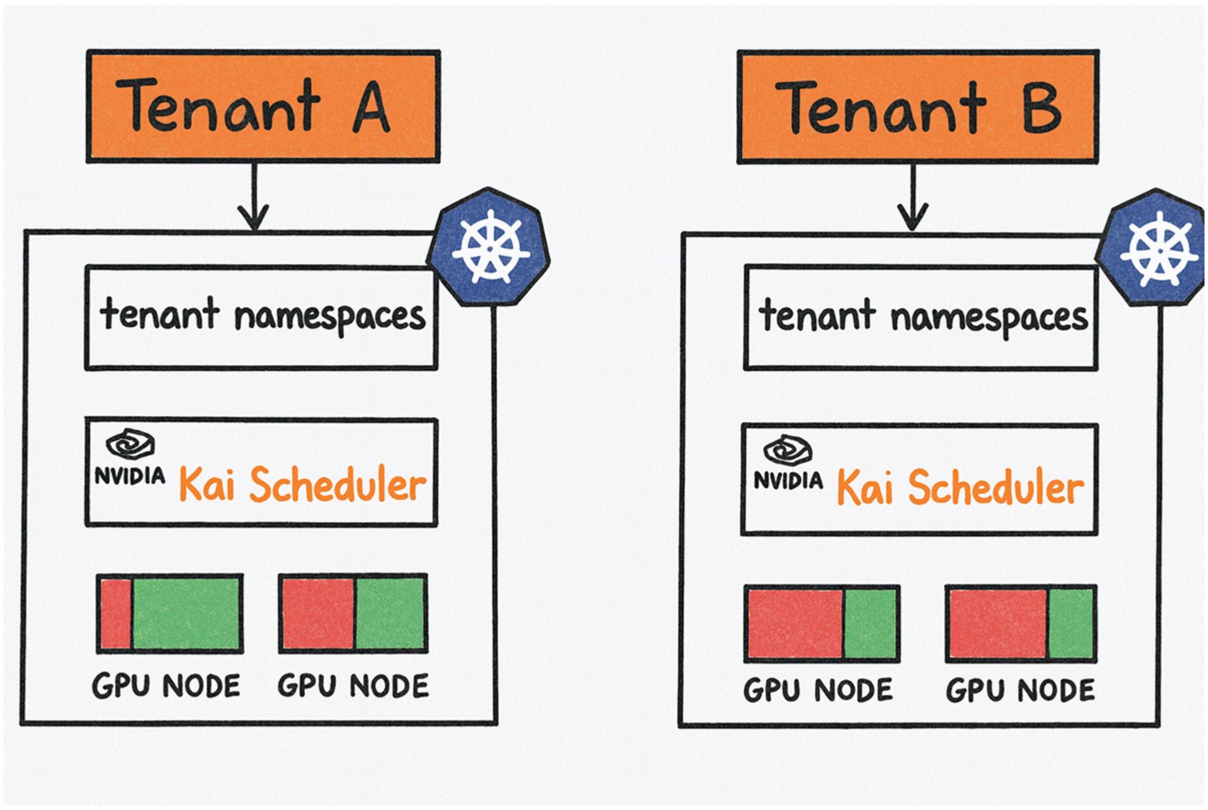


Fig. Multi-tenant architecture showing Tenant A and Tenant B each with separate namespaces, Kai Scheduler, and shared GPU nodes displaying utilization with red (used) and green (available) portions

Even the clusters you have now aren't properly utilizing GPUs; one is overutilized, while the other is underutilized.

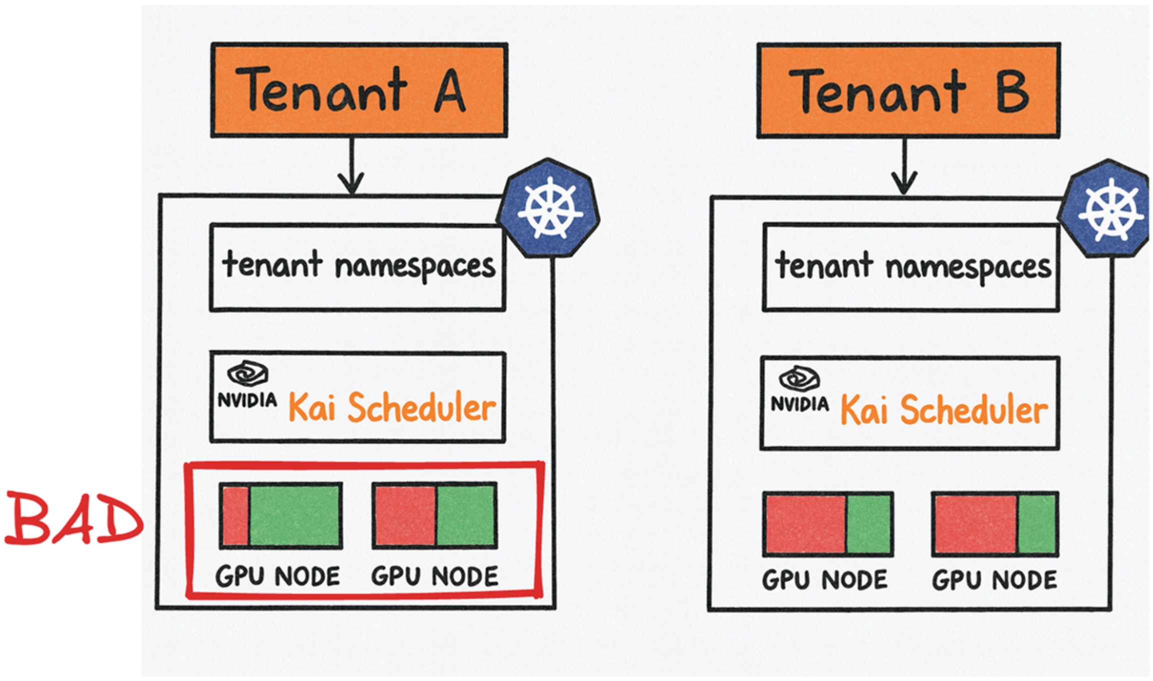


Fig. GPU utilization diagram showing Tenant A with "BAD" GPU node utilization (partial red/green usage) and Tenant B with better distributed GPU usage across two nodes

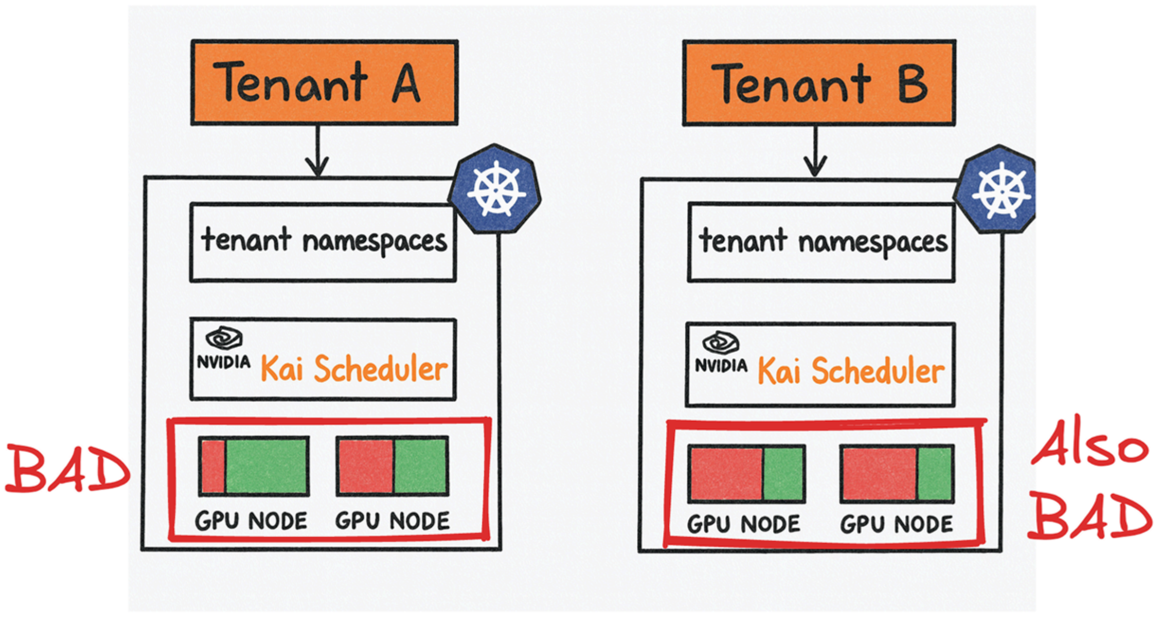


Fig. Similar multi-tenant setup showing inefficient GPU utilization patterns across their dedicated GPU nodes

After this, you will still have teams waiting to get their hands on GPUs since you can only create a finite number of clusters backed by GPUs for the teams due to limited resource availability (and again, the same principles apply to the cloud).

So, should you go back to shared clusters using namespace now?

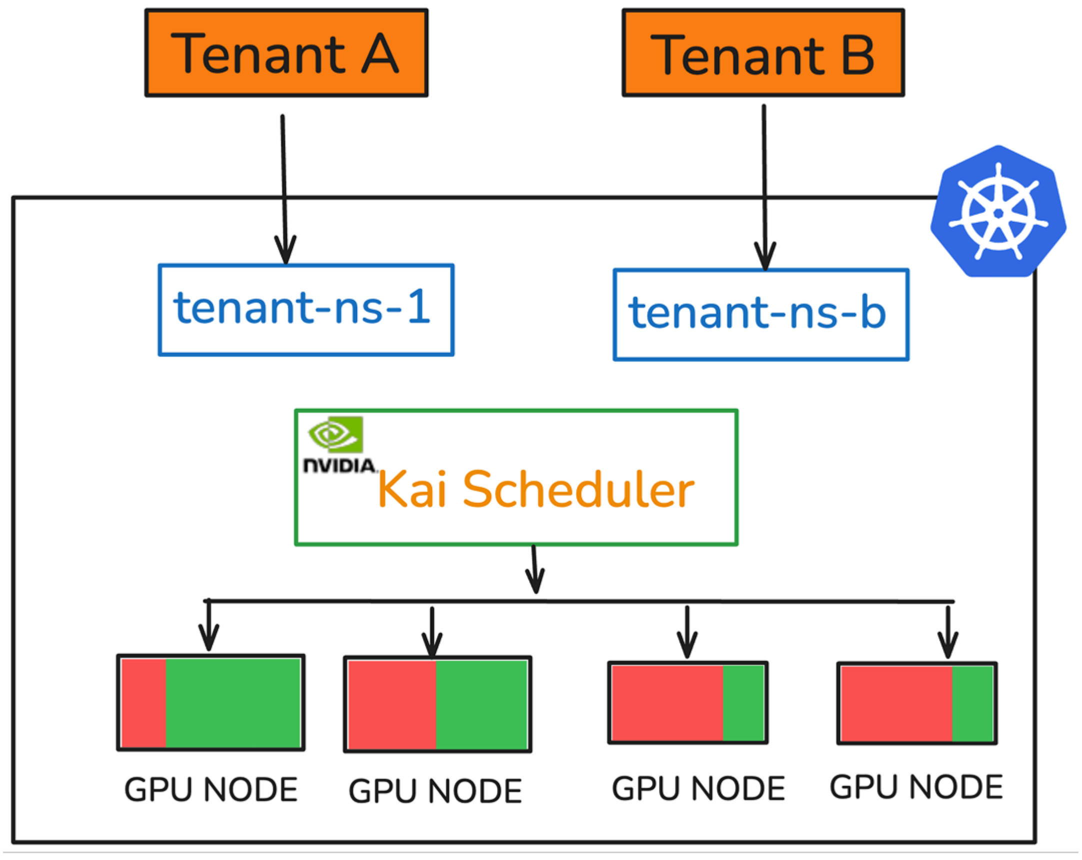


Fig. Namespace-based multi-tenancy architecture showing Tenant A and Tenant B using tenant-ns-1 and tenant-ns-b namespaces respectively, with Kai Scheduler managing GPU node allocation

If we pursue this route, where there is a namespace for everyone, it will lead back to all the namespace-related issues we discussed earlier.

Instead, let's review how vCluster solves this.

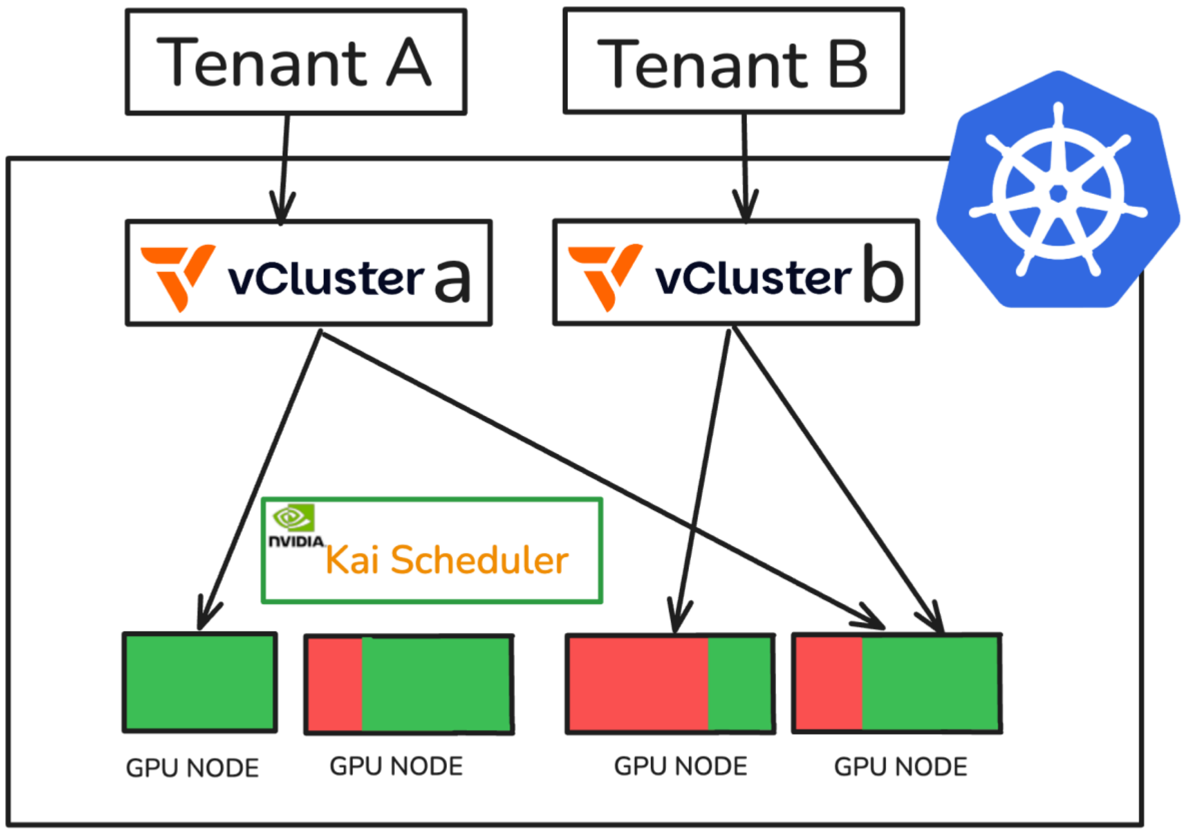


Fig. vCluster architecture showing Tenant A and Tenant B each with their own vCluster (a and b), managed by Kai Scheduler with NVIDIA integration, distributing workloads across four GPU nodes

Here, you have a bare metal infrastructure or cloud nodes, so create a single Kubernetes cluster with all required resources.

After this, you create a virtual cluster using vCluster for each team.

This gives every team its own isolated control plane with access limited to that particular cluster. You can also make this a part of your internal developer platform.

And if there's a new tenant, say Tenant C, there's no need to wait; create another virtual cluster.

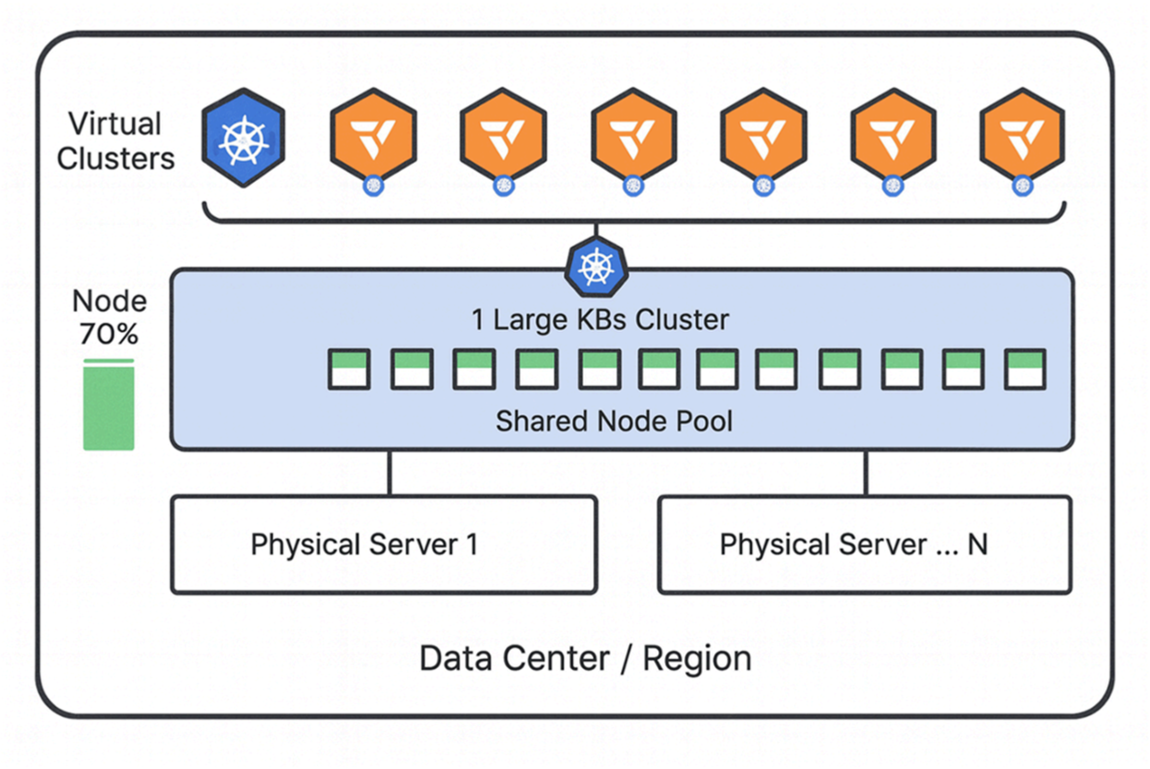


Fig. Data center architecture showing multiple virtual clusters, a single large Kubernetes cluster with shared node pool, and physical servers with 70% node utilization

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)

Imagine you're managing a shared GPU environment where different teams want to deploy RAG-based applications using Ollama, each with different datasets, model choices, and scaling needs.

How do you isolate GPU usage, ensure fair scheduling, and support GPU sharing?

This section walks you through deploying Kai Scheduler on a Kubernetes cluster with a GPU node and then launching isolated RAG workloads using vCluster.

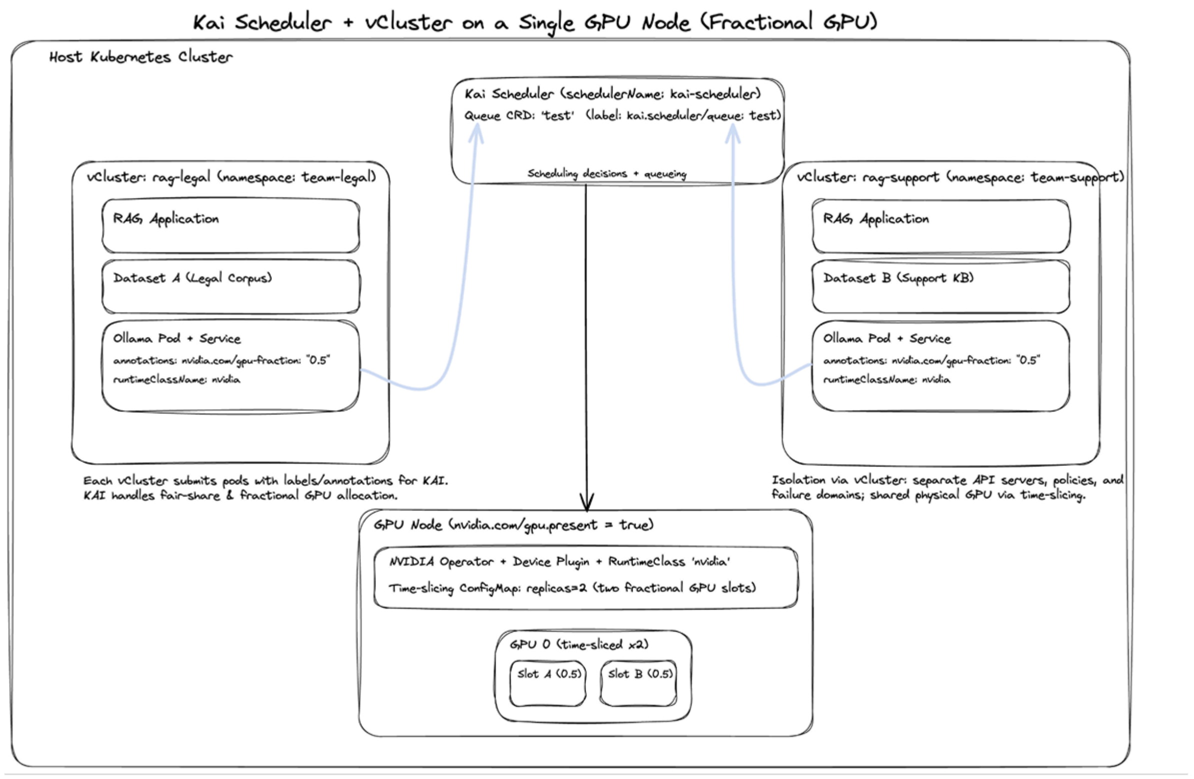

Kai Scheduler + vCluster on a Single GPU Node (Fractional GPU)
Host Kubernetes Cluster
Kai Scheduler (schedulerName: kai-scheduler)
Queue CRD: 'test' (label: kai.scheduler/queue: test)
Scheduling decisions + queueing
vCluster: rag-legal (namespace: team-legal)
RAG Application
Dataset A (Legal Corpus)
Ollama Pod + Service
annotations: nvidia.com/gpu-fraction: "0.5"
runtimeClassName: nvidia
vCluster: rag-support (namespace: team-support)
RAG Application
Dataset B (Support KB)
Ollama Pod + Service
annotations: nvidia.com/gpu-fraction: "0.5"
runtimeClassName: nvidia
Each vCluster submits pods with labels/annotations for KAI.
KAI handles fair-share & fractional GPU allocation.
Isolation via vCluster: separate API servers, policies, and failure domains; shared physical GPU via time-slicing.
GPU Node (nvidia.com/gpu.present = true)
NVIDIA Operator + Device Plugin + RuntimeClass 'nvidia'
Time-slicing ConfigMap: replicas=2 (two fractional GPU slots)
GPU 0 (time-sliced x2)
Slot A (0.5)
Slot B (0.5)

Fig. Architecture diagram of Kai Scheduler with vCluster on a single GPU node showing fractional GPU allocation, with two vClusters

1. Prerequisites

- Kubernetes cluster with NVIDIA GPUs (bare metal or GKE/AKS/EKS with GPU nodes)
- kubectl (v1.28+) installed and context set
- helm (v3.10+) installed
- vcluster CLI installed (brew install vcluster or docs)
- One Kubernetes cluster with at least 1 GPU node (e.g., g4dn.xlarge)

In this example, we are using an EKS cluster with g4dn.xlarge single node GPU.

For the same EKS setup:

Start by creating a Kubernetes cluster with a GPU node pool.

To simplify this, we are using a single GPU node.

config.yaml

```yaml
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig
metadata:
  name: vcluster-gpu
  region: us-east-2
  version: '1.32'
  tags:
    usage: sandbox
    owner: vcluster
vpc:
  cidr: 10.1.0.0/16
  autoAllocateIPv6: false
  hostnameType: resource-name
  clusterEndpoints:
    publicAccess: true
    privateAccess: true
managedNodeGroups:
  - name: vcluster-gpu
    amiFamily: Ubuntu2404
    desiredCapacity: 1
    instanceTypes:
      - g4dn.xlarge
    ssh:
      allow: true
      publicKeyName: vcluster
```

Run the following command to create an EKS cluster using eksctl:

```bash
$ eksctl create cluster --config-file eks-config-gpu.yaml
```

Now, get the **kubeconfig** file and export the **KUBECONFIG** variable:

~~~bash
$ aws eks update-kubeconfig --region us-east-2 --name vcluster-gpu --kubeconfig ./vcluster-kubeconfig

You'll see your cluster is ready and you're connected.

```terminal|title=bash|command=1
kubectl get nodes
NAME                                        STATUS    ROLES     AGE
VERSION
ip-10-1-92-3.us-east-2.compute.internal     Ready     <none>    19h
v1.32.5
~~~

2. NVIDIA GPU Operator Setup

Install the GPU operator and configure it to allow fractional GPU sharing using a time-slicing config:

Install NVIDIA Operator:

```bash
$ helm repo add nvidia https://helm.ngc.nvidia.com/nvidia
$ helm repo update
$ helm install gpu-operator nvidia/gpu-operator -n gpu-operator --create-namespace
NAME: gpu-operator
LAST DEPLOYED: Tue Sep  2 12:07:02 2025
NAMESPACE: gpu-operator
STATUS: deployed
REVISION: 1
TEST SUITE: None
```

Now verify that all GPU operator components are running in the `gpu-operator` namespace:

```bash
$ kubectl get pods -n gpu-operator
NAME                                                          READY   STATUS
gpu-feature-discovery-svqzb                                   1/1     Running
gpu-operator-5798b5b564-rw57d                                 1/1     Running
gpu-operator-node-feature-discovery-gc-86f6495b55-dsvfk       1/1     Running
gpu-operator-node-feature-discovery-master-694467d5db-cc2zj   1/1     Running
gpu-operator-node-feature-discovery-worker-p8k7n              1/1     Running
nvidia-container-toolkit-daemonset-s4vg4                      1/1     Running
nvidia-cuda-validator-sqfp2                                   0/1     Completed
nvidia-dcgm-exporter-2bx94                                    1/1     Running
nvidia-device-plugin-daemonset-tvvpr                          1/1     Running
nvidia-driver-daemonset-6wjnt                                 1/1     Running
nvidia-operator-validator-s4wxv                               1/1     Running
```

Create the time-slicing config in the `gpu-operator` namespace:

```bash
$ kubectl apply -n gpu-operator -f - <<EOF
apiVersion: v1
kind: ConfigMap
metadata:
  name: time-slicing-config
  namespace: gpu-operator
  labels:
    nvidia.com/device-plugin.config: "true"
data:
```

```
$   time-slicing-config.yaml: |
      version: v1
      flags:
        migStrategy: none
      sharing:
        timeSlicing:
          resources:
          - name: nvidia.com/gpu
            replicas: 2
  EOF
```

Restart the NVIDIA device plugin to apply the new configuration:

```bash
$ kubectl rollout restart daemonset nvidia-device-plugin-daemonset -n gpu-operator
```

Finally, confirm that the GPU capacity is exposed as two fractional devices:

```bash
$ kubectl get nodes -o json | jq '.items[].status.capacity."nvidia.com/gpu"'
"2"
```

The single physical GPU was split into two logical devices.

3. Install KAI-Scheduler

Install KAIi-Scheduler on the host Kubernetes cluster and make sure the Global sharing is true.

```bash
$ helm upgrade -i kai-scheduler \
  oci://ghcr.io/nvidia/kai-scheduler/kai-scheduler \
  -n kai-scheduler --create-namespace \
  --version v0.8.4 \
  --set gpuSharing=true
```

List the pods in the `kai-scheduler` namespace:

```bash
$ kubectl get pods -n kai-scheduler
NAME                                   READY   STATUS
binder-5dbbb78b47-75hbq                1/1     Running
kai-admission-649bf8cbd7-gjs4t         1/1     Running
podgroup-controller-8ff79c87f-8p7wr    1/1     Running
podgrouper-85cfbb79b5-dgjx4            1/1     Running
queuecontroller-658499d99c-v6wjt       1/1     Running
scheduler-56f6f568b9-6gpgl             1/1     Running
```

Once all the pods in the `kai-scheduler` namespace are up and running, we must create a queue.

This queue is where vCluster workloads will submit jobs via the `kai.scheduler/queue` label:

```bash
$ kubectl apply -f - <<EOF
apiVersion: scheduling.run.ai/v2
kind: Queue
metadata:
  name: test
spec:
  parentQueue: default
  resources:
    cpu:
      quota: -1
      limit: -1
```

```
$       overQuotaWeight: 1
      gpu:
        quota: -1
        limit: -1
        overQuotaWeight: 1
      memory:
        quota: -1
        limit: -1
        overQuotaWeight: 1
  EOF
```

We have a `test` queue successfully created as a child of the default queue:

```bash
$ kubectl get queue
NAME   PRIORITY   PARENT    CHILDREN   DISPLAYNAME
test              default
```

The queue is ready to use.

4. Create a vCluster with Kai Integration

First, define the sync configuration that disables owner references and enables syncing of runtime classes:

```bash
$ cat > values.yaml <<EOF
experimental:
```

```
$   syncSettings:
      setOwner: false

  sync:
    fromHost:
      runtimeClasses:
        enabled: true
  EOF
```

Install the `vcluster` binary if not already present, then create a new virtual cluster named `rag-legal` inside the `team-legal` namespace:

```bash
$ curl -L -o vcluster "https://github.com/loft-sh/vcluster/releases/latest/download/vcluster-linux-amd64" \
  && sudo install -c -m 0755 vcluster /usr/local/bin \
  && rm -f vcluster

$ vcluster create rag-legal --namespace team-legal -f values.yaml

11:32:16 done Successfully created virtual cluster rag-legal in namespace team-legal
11:32:15 info Waiting for vcluster to come up...
11:34:45 done Switched active kube context to vcluster_rag-legal_team-legal_minikube
```

Verify the vCluster is running:

```bash
$ vcluster list
    NAME     | NAMESPACE   | STATUS   | VERSION | CONNECTED |   AGE
-------------+-------------+----------+---------+-----------+--------
  rag-legal | team-legal | Running | 0.27.0  | True       | 3m25s
```

5. Deploy Application

Now, create a simple GPU workload inside the vCluster.

This pod requests half of a GPU via Kai Scheduler annotations and will run using the NVIDIA runtime class.
Paste the following content to `podkai2.yaml` :

podkai2.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: gpu-sharing-pod
  labels:
    kai.scheduler/queue: test
  annotations:
    nvidia.com/gpu-fraction: "0.5"
spec:
  schedulerName: kai-scheduler
  runtimeClassName: nvidia
  nodeSelector:
    nvidia.com/gpu.present: "true"
  containers:
  - name: ollama-rag
    image: ollama/ollama:latest
    command: ["ollama", "serve"]
    resources:
      limits:
        nvidia.com/gpu: 1
```

Then apply the file to the vCluster:

bash

```bash
$ vcluster connect rag-legal --namespace team-legal -- kubectl apply -f podkai2.yaml
```

Once the pod is running, pull a model into it:

bash

```bash
$ kubectl exec -it -n team-legal gpu-sharing-pod-x-default-x-rag-legal -- ollama pull mistral
```

```
pulling f5074b1221da: 100% ████████████████████  4.4 GB
pulling 43070e2d4e53: 100% ████████████████████   11 KB
pulling 1ff5b64b61b9: 100% ████████████████████  799 B
pulling ed11eda7790d: 100% ████████████████████   30 B
pulling 1064e17101bd: 100% ████████████████████  487 B
verifying sha256 digest
writing manifest
success
```

Forward the container port to your local machine:

```bash
$ kubectl port-forward pod/gpu-sharing-pod-x-default-x-rag-legal -n team-legal  11434:11434
Forwarding from 127.0.0.1:11434 → 11434
```

Then send a test request:

```bash
$ curl -s http://localhost:11434/api/chat \
  -H "Content-Type: application/json" \
  -d '{
    "model": "mistral",
    "stream": false,
    "messages": [
      {"role": "user", "content": "Summarize Kubernetes in one line"}
    ]
  }' | jq -r '.message.content'

Kubernetes is an open source platform for automating
deployment, scaling, and management of containerized applications.
```

Repeat the same for other departments.

5. Final Outcome

You now have:

Two vClusters, one for legal research, one for customer support

Each runs an isolated RAG application with Ollama

Each workload gets fractional GPU access (0.5) using Kai Scheduler

Complete failure isolation, scaling independence, and compliance separation

This demo shows the power of combining:

- vCluster → for multi-tenancy, custom policies, versioned Kubernetes APIs
- Kai Scheduler → for fair GPU sharing, queuing, and priority handling
- NVIDIA Operator → for GPU resource visibility and management
- Ollama + RAG → for LLM-based tasks like legal, support, finance, etc.

Together, they enable multi-tenant AI platforms to run efficiently and securely on shared GPU clusters.

Conclusion

This chapter explores the challenge of scaling GPU resources for AI workloads in multi-tenant Kubernetes environments.

While namespaces provide a lightweight approach to multi-tenancy, they fall short in isolation, autonomy, compliance, and operational flexibility, problems that grow acute when teams require GPU operators, custom CRDs, or independent control planes.

Dedicated clusters solve isolation, but at the cost of fragmentation and management overhead. vCluster bridges this gap by providing lightweight, fully isolated virtual control planes inside a shared host cluster, allowing each team to run its own Kubernetes environment with autonomy and security.

Combined with advanced GPU scheduling via Kai Scheduler and fractional GPU support from the NVIDIA Operator, vCluster enables organizations to maximize GPU utilization while ensuring fairness, compliance, and scalability.

It delivers strong isolation without the inefficiency of managing dozens of clusters.