GPUs on Kubernetes Unlocked

Understanding GPU scheduling and resource management

Roland Huß
Distinguished Engineer, Red Hat

Generative AI on Kubernetes

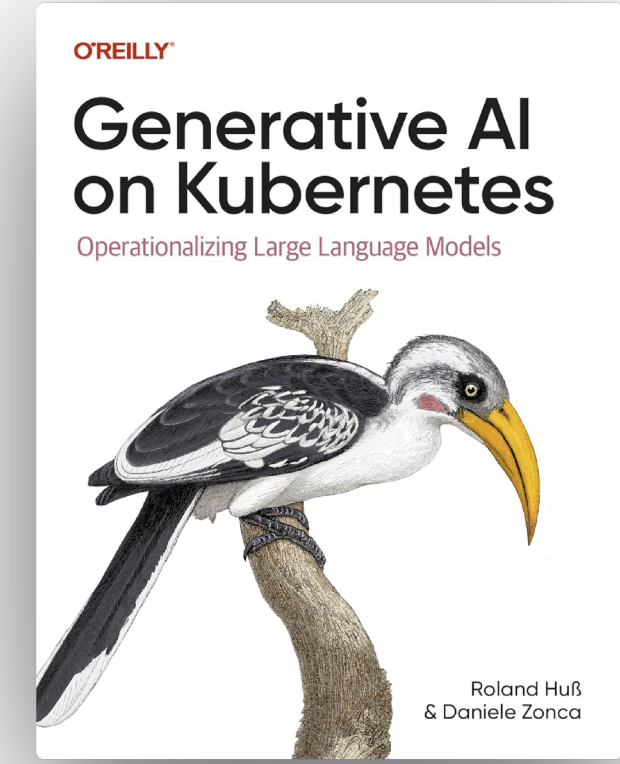
Roland Huß · Daniele Zonca

O'Reilly Media · March 2026

Chapter 5: GPU Scheduling on Kubernetes

📖 **Early Release: Raw & Unedited**

bit.ly/genaik8s-raw



Kubernetes Lacks Native GPU Support

Kubernetes Native

- CPU & Memory
- Standard Resources

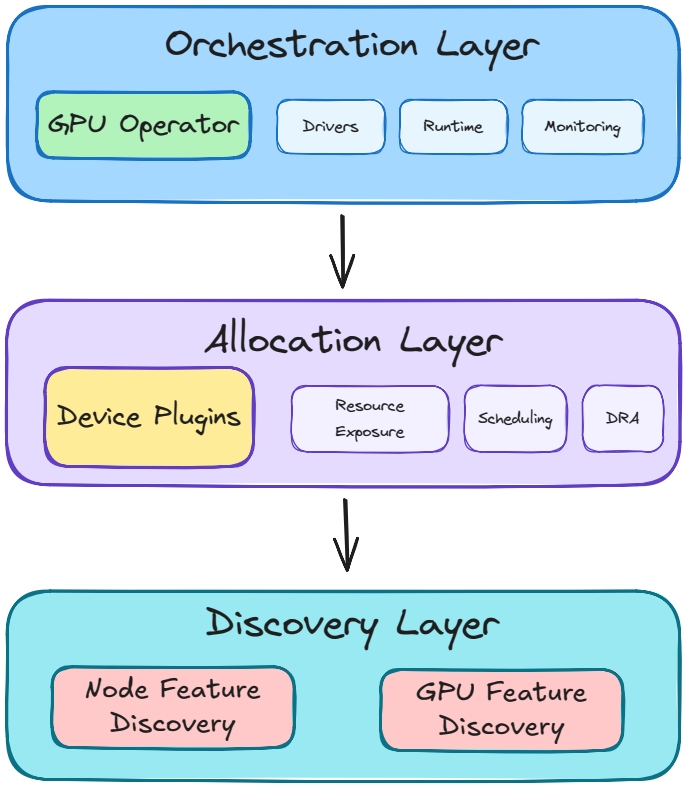
✓ Built-in

LLM Requirements

- GPU Compute
- VRAM & Interconnects

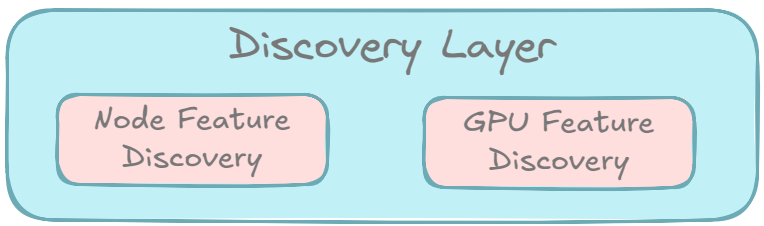
🔧 Extensions Needed

Three Layers Enable GPU Scheduling



Discovery Layer

Finding GPUs in Your Cluster



Hardware Labels Enable Intelligent GPU Scheduling

The Challenge:

- Different GPU models per node
- T4s with 16GB, A100s with 40GB, H100s with 80GB
- Scheduler picks randomly without labels

What Scheduler Needs:

- Architecture (Ampere, Hopper)
- Memory and capabilities (MIG, NVLink)

```yaml
apiVersion: v1
kind: Node
metadata:
  name: gpu-node-01
  labels:
    # NFD
    feature.node.kubernetes.io/pci-10de.present: "true"

    # GFD
    nvidia.com/gpu.product: A100-SXM4-40GB
    nvidia.com/gpu.memory: "40537"
    nvidia.com/gpu.family: ampere
    nvidia.com/gpu.count: "8"
    nvidia.com/cuda.runtime.major: "12"
    nvidia.com/mig.capable: "true"
...
```

The Solution: Automatic hardware detection → Node labels

NFD Provides Foundation for Hardware Discovery

- **General-purpose** hardware discovery
- Detects CPU, PCI devices, network
- Runs as DaemonSet and labels nodes automatically

```
feature.node.kubernetes.io/pci-0302_10de.present: "true"
feature.node.kubernetes.io/pci-0302_1002.present: "true"
feature.node.kubernetes.io/kernel-version.full: "6.6.8"
feature.node.kubernetes.io/system-os_release.ID: "fedora"
```

PCI Label format: `pci-<class>_<vendor>`

- **0302** = PCI class (3D graphics controller)
- **10de** = NVIDIA vendor ID | **1002** = AMD vendor ID

Detects GPU presence, not model or memory

GFD Delivers Model-Specific GPU Information

- **GPU Feature Discovery (GFD)** - NVIDIA graphics cards only
- AMD alternative: AMD GPU Node Labeler

Features:

- GPU model name
- Memory capacity (MiB)
- Architecture family
- CUDA driver version
- MIG and time-slicing support

```
nvidia.com/gpu.count: "4"
nvidia.com/gpu.product: "A100-SXM4-40GB"
nvidia.com/gpu.memory: "40537"
nvidia.com/gpu.family: "ampere"
nvidia.com/gpu.compute.major: "8"
nvidia.com/gpu.compute.minor: "0"
nvidia.com/cuda.driver-version.full: "525.105.17"
nvidia.com/cuda.driver.major: "525"
nvidia.com/cuda.runtime-version.major: "12"
nvidia.com/cuda.runtime-version.minor: "0"
nvidia.com/mig.capable: "true"
```

GFD is part of the NVIDIA GPU Operator

NFD Detects Presence, GFD Provides Specifications

Aspect	**NFD**	**GFD**
Scope	General hardware	NVIDIA GPUs
Detail	Basic presence	Detailed specs
Labels	`feature.node.k8s.io/`	`nvidia.com/`
Use Case	"Has GPU"	"Has A100 40GB"
Install	Standalone	Via GPU Operator

Allocation Layer

Making GPUs Schedulable



Device Plugins Turn GPUs Into Schedulable Resources

Four Core Functions:

1. **Device Discovery** – Detect GPUs, report to Kubelet
2. **Resource Allocation** – Exclusive access by default
3. **Health Monitoring** – Report unhealthy devices
4. **Scheduler Integration** – Expose as `nvidia.com/gpu`

The magic moment: GPUs go from invisible to schedulable

Resource-Based Scheduling: Simple and Direct

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: gpu-pod
spec:
  containers:
  - name: cuda-app
    image: nvidia/cuda:12.2-base
    resources:
      limits:  # ⟵ GPU resource request
        nvidia.com/gpu: 1
```

How It Works

1. Device plugin advertises `nvidia.com/gpu: 4`
2. Pod requests `nvidia.com/gpu: 1`
3. Scheduler finds node with available GPU
4. Kubelet allocates exclusive access

Simple But Limited

Treats all GPUs equally - A100 and T4 both count as "1 GPU"

Resource Requests Trade Simplicity for GPU Precision

Strengths

- Simple syntax
- Native K8s integration
- No extra labels needed
- Works with familiar resource model
- Good for homogeneous fleets

Limitations

- All GPUs look identical
- Can’t request "A100" specifically
- Can’t specify memory size
- Can’t request compute capability
- GPU-blind scheduling

Best For: Dev/test environments with single GPU type

Request Specific GPU Models with nodeSelector

```
spec:
  ...
  nodeSelector:
    nvidia.com/gpu.product: Tesla-T4  # ⟵ Exact match required
  ...
```

🌟 Strengths

- Simple, explicit syntax
- Direct model specification
- Easy to debug and troubleshoot

⛔ Limitations

- All-or-nothing matching
- No alternatives ("T4 OR A10")
- Can’t express ranges

Best For: Workloads tested on specific GPU model

Match GPUs by Memory Size with Node Affinity

```yaml
affinity:
  nodeAffinity:
    ...
    matchExpressions:
    - key: nvidia.com/gpu.memory
      operator: Gt  # Greater than
      values: ["40000"]
```

🌟 Strengths

- Rich operators (Gt, Lt, In, NotIn)
- Express minimum requirements
- Allow multiple GPU options

⛔ Limitations

- More verbose than nodeSelector
- Complex syntax to learn
- Harder to debug failures

Best For: Flexible GPU requirements (>40GB memory, any Ampere/Hopper)

Taints Protect GPU Nodes From CPU Workloads

```yaml
kind: Node
spec:
  taints:  # ⟵ Reject non-GPU pods
  - key: nvidia.com/gpu
    value: "true"
    effect: NoSchedule
  ...
```

```yaml
kind: Pod
spec:
  tolerations:  # ⟵ Opt-in to GPU nodes
  - key: nvidia.com/gpu
    operator: Exists
    effect: NoSchedule
  ...
```

🌟 Strengths

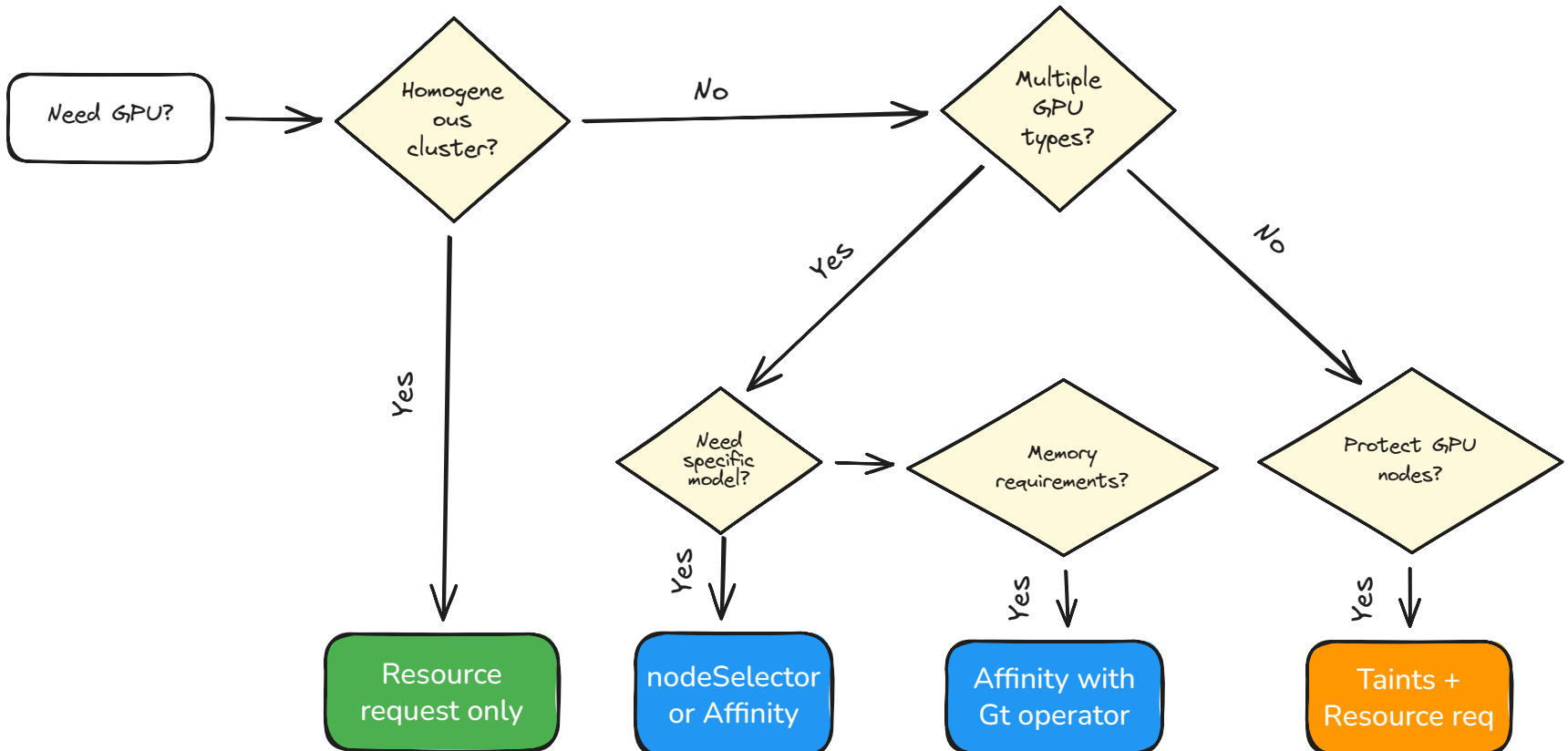
- Protects expensive GPU resources
- Explicit opt-in model
- Complements resource requests
- Prevents accidental placement

⛔ Limitations

- Requires manual toleration setup
- Extra YAML complexity
- Easy to misconfigure
- No warning on missing toleration

Best For: Heterogeneous clusters with expensive GPU nodes

When to Use Each Scheduling Approach



Combined Approach Is Production Best Practice

```yaml
kind: Pod
spec:
  containers:
  - name: inference
    resources:
      limits:
        nvidia.com/gpu: 1  # ⟵ Ensures GPU availability
    ...
  nodeSelector:
    nvidia.com/gpu.family: ampere  # ⟵ Ensures correct GPU type
```

🌟 Double Protection

- Resource request finds available GPU
- Label selector ensures correct type
- Scheduler validates both constraints
- Prevents wrong GPU placement

🎯 How It Works

- Scheduler first filters by available GPUs
- Then applies nodeSelector constraints
- Result: Right GPU type AND available
- Add tolerations if nodes are tainted

DRA Requests GPUs By Intent Not Model

💡 Dynamic Resource Allocation (DRA)

- Intent-based API: request by capability, not model
- Like PVC: request by class, not device
- DeviceClass resources defined by vendors
- Scheduler resolves requirements to actual GPUs

```yaml
apiVersion: resource.k8s.io/v1beta1
kind: ResourceClaimTemplate
metadata:
  name: high-memory-gpu
spec:
  devices:
    requests:
      - deviceClassName: gpu.nvidia.com/a100  # ⟵ Device class defines available GPU types
        minMemory: "40Gi"  # ⟵ Request by capability
        migMode: "disabled"
```

Pods Claim Resources, Scheduler Allocates Devices

🔄 Template-Based Flow

1. Create ResourceClaimTemplate defining GPU requirements
2. Reference template in Pod spec

```yaml
apiVersion: resource.k8s.io/v1beta1
kind: ResourceClaimTemplate
metadata:
  name: high-memory-gpu
spec:
  devices:
    requests:
      - deviceClassName: gpu.nvidia.com/a100
        minMemory: "40Gi"
        allocationMode: "exclusive"
```

```yaml
kind: Pod
spec:
  resourceClaims:  # ⟵ Reference template
  - name: gpu-claim
    resourceClaimTemplateName: high-memory-gpu
  containers:
  - name: inference
    resources:
      claims:  # ⟵ Use gpu-claim defined above
      - name: gpu-claim
```

- Scheduler allocates matching device automatically
- Pattern mirrors PVC/PV binding

DRA Is Promising But Not Production-Ready

🌟 Benefits

- Precise requirements
- Dynamic allocation
- No manual labels
- Future-proof architecture

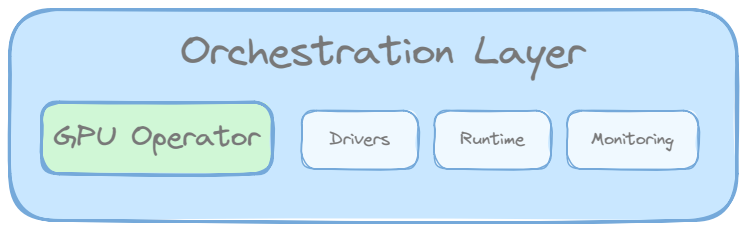
⛔ Current Limitations

- Beta API
- Limited adoption
- Experimental driver
- Feature gate required

Wait for production, but start learning now

Orchestration Layer

NVIDIA GPU Operator



Scheduling GPUs Is Only Half the Battle

🌟 What We Have So Far

- Discovery labels nodes with GPU capabilities
- Device plugins expose GPUs as schedulable resources
- Scheduler can place GPU workloads on correct nodes

⛔ What's Still Missing for Production

- Driver installation and updates
- Container runtime configuration
- GPU health monitoring
- Lifecycle management across cluster

GPU Operator Automates Six Critical Components

Managed Components

✅	**Device Plugin**	*Exposes nvidia.com/gpu resources*
✅	**GPU Feature Discovery**	*Adds detailed GPU labels*
⬜	**MIG Manager**	*Configures GPU partitioning*
⬜	**Driver Container**	*Installs NVIDIA drivers on host nodes*
⬜	**Container Toolkit**	*Configures runtime to expose GPUs to containers*
⬜	**DCGM Exporter**	*Prometheus metrics for monitoring*

```yaml
apiVersion: nvidia.com/v1
kind: ClusterPolicy
metadata:
  name: gpu-cluster-policy
spec:
  driver:
    enabled: true
  toolkit:
    enabled: true
  devicePlugin:
    enabled: true
    config:
      name: gpu-sharing-config
  dcgmExporter:
    enabled: true
  mig:
    strategy: mixed
```

Change one YAML → operator reconfigures entire cluster

GPU Sharing

Sub-GPU Allocation

NVIDIA GPU Sharing Maximizes Utilization

The Challenge:

- GPUs expensive, often <30% utilized
- Small models waste full GPU capacity
- Dev/test environments need flexibility

Time-Slicing

- 🌟 Works on any GPU
- 🌟 Flexible for dev/bursty workloads
- ⛔ No isolation between pods

MIG (Multi-Instance GPU)

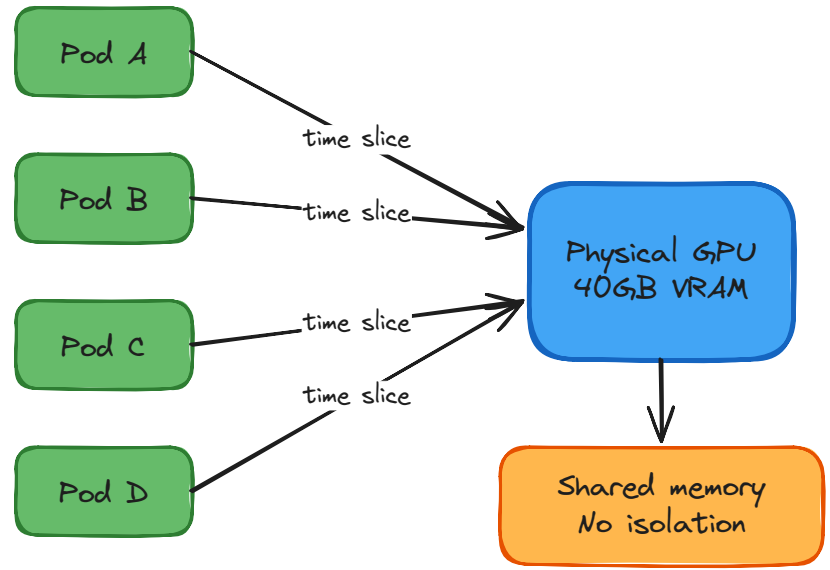
- 🌟 Hardware isolation
- 🌟 Production-ready for multi-tenant
- ⛔ A100/H100 only

Pick sharing strategy based on isolation needs and GPU model

Time-Slicing Oversubscribes GPUs

- 1 physical GPU → N virtual GPUs
- Processes interleave (like CPU time-sharing)
- No memory isolation
- Node advertises N GPUs to scheduler

```yaml
# ConfigMap
sharing:
  timeslicing:
    resources:
    - name: nvidia.com/gpu
      replicas: 4  # 1 physical → 4 virtual
---
# Referenced in ClusterPolicy
devicePlugin:
  config:
    name: time-slicing-config
```



Trade memory isolation for higher density – use for dev/test and bursty workloads

Time-Slicing Trades Isolation for Density

🌟 Good For

- Dev/test & interactive notebooks
- Bursty lightweight inference
- Many small models (<5GB each)
- GPUs without MIG (T4, V100)

⛔ Watch Out For

- No memory isolation
- No fault isolation
- Unpredictable performance
- Multi-GPU requests ineffective

💚 8 small models (2GB each) share one GPU with bursty usage

⭐ Dev notebooks sitting idle 80% of time

⛔ Single large LLM (40GB) blocks all other workloads

For isolation guarantees → use MIG instead

NVIDIA MIG Provides Hardware-Isolated Partitions

Fixed Profiles (A100, H100, B100/B200 only) – Examples: 1g.5gb, 2g.10gb, 3g.20gb, 7g.40gb

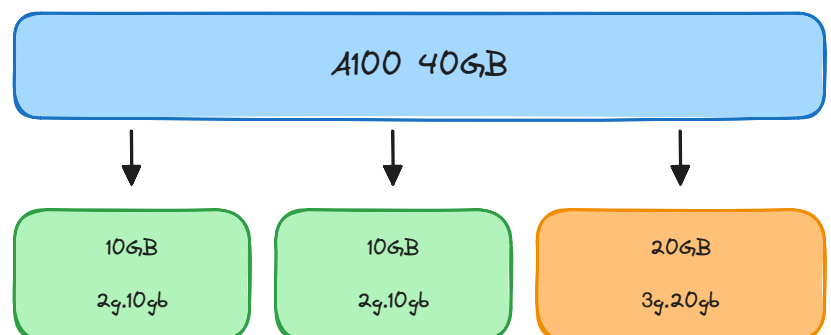
Single Strategy

- Identical GPU partitioning, simple to manage
- Easy to scale uniformly

Mixed Strategy

- Per-GPU partitioning, flexible placement
- More complex configuration

```
# ClusterPolicy
devicePlugin:
  config:
    name: mig-config        # ConfigMap name
    default: mixed-profile  # Profile name
---
# ConfigMap: mig-config
data:
  mixed-profile: |          # Profile referenced above
    mig-configs:
      ... :
        mig-devices:
          "2g.10gb": 2      # 2× 10GB (4/7 slices)
          "3g.20gb": 1      # 1× 20GB (3/7 slices)
```



MIG Provides Isolation at Cost of Flexibility

Good For

- Production multi-tenancy with isolation
- Multiple smaller models (7B params)
- Guaranteed resources required
- Ampere+ hardware (A100, H100)

Limitations

- Fixed partition sizes (min 5GB)
- No bursting beyond partition
- Ampere+ GPUs only
- More operational complexity

- Multi-tenant: 3 customers get 10GB slices - one crashes, others unaffected
- 7 models (5GB each) share A100 with complete isolation
- Dev needs 8GB, could burst to 15GB - locked at 10GB
- 70B model (35GB) won't fit 20GB partition - needs full GPU

For flexible sharing → use time-slicing instead

Choose Time-Slicing for Flexibility, MIG for Isolation

	Time-Slicing	MIG
Isolation & Faults	Shared, cascade failures	Dedicated, isolated
Resource Guarantees	Time-sharing, variable	Memory + compute, predictable
Flexibility	Burst to full GPU	Fixed partition
Complexity	Simple replica count	Fixed profiles, partition mgmt
Hardware Support	All NVIDIA GPUs	Ampere+ only
Use Case	Dev, bursty workloads	Production, multi-tenant

Trade isolation for flexibility - choose based on your workload requirements

Can Combine Time-Slicing and MIG for Maximum Density

```
mig:
  strategy: single  # Create 2× 20GB MIG instances
sharing:
  replicas: 2       # Time-slice each MIG instance 2×
# Result: 4 schedulable units per physical GPU
```

Pattern:

- Create MIG instances (memory isolation)
- Time-slice each instance (increase density)
- Balance: Some isolation + Higher utilization

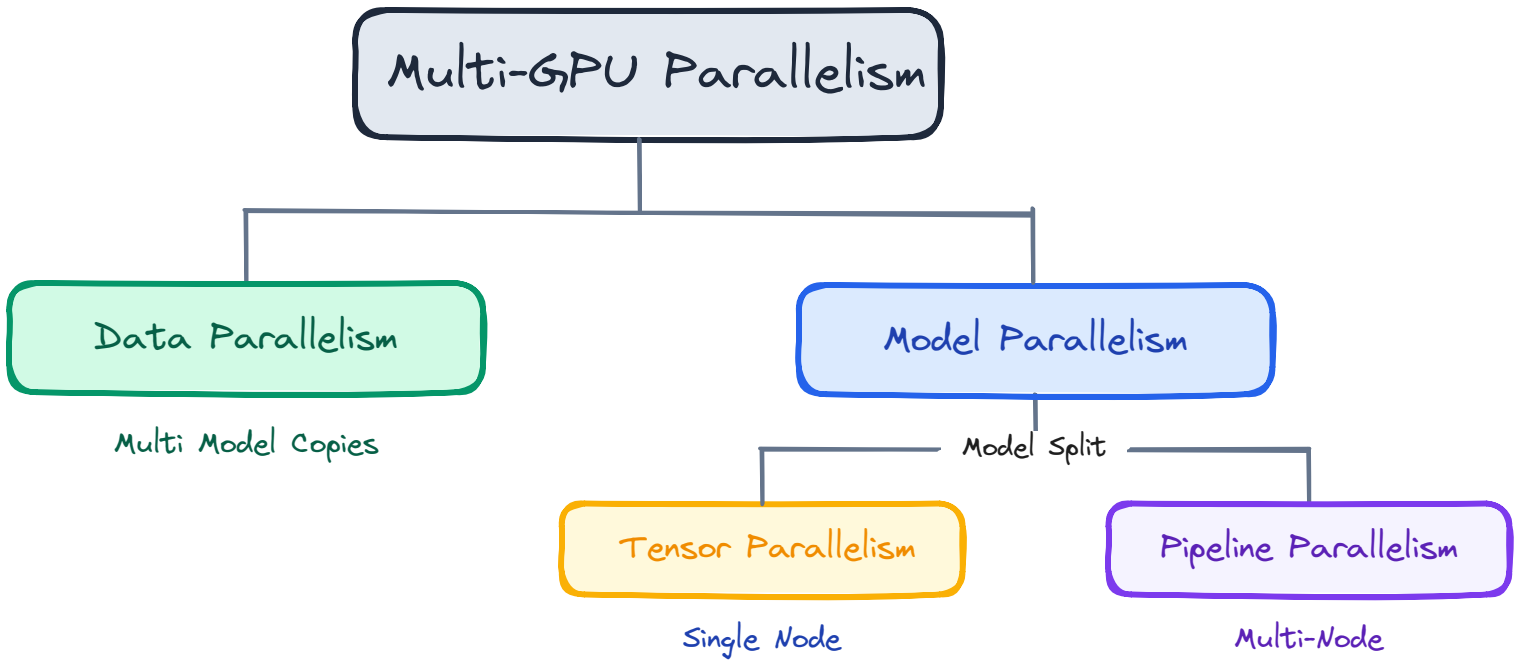
Example:

- 7g.80gb MIG instance × 4 time-slicing replicas
- 4 pods share 80GB MIG partition
- Memory isolation + Compute sharing

Multi-GPU Patterns

Scaling Out

Two Dimensions Define Three GPU Strategies



Data Parallelism Trades Memory for Throughput

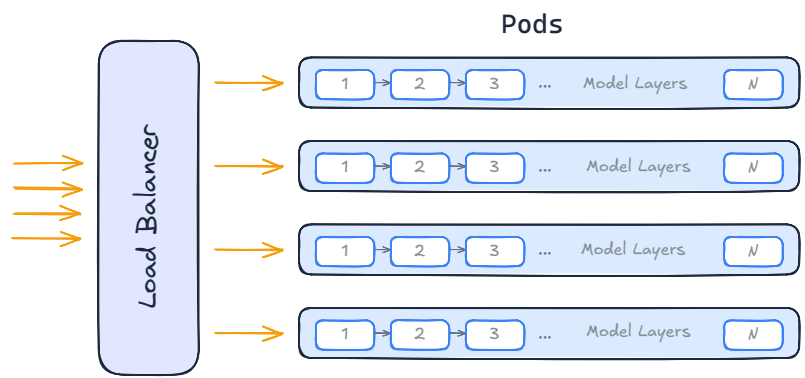
⭐ Strengths

- Model fits single GPU
- High request volume (QPS)
- Simple autoscaling needed
- Independent requests

⛔ Limitations

- N replicas = N × memory cost
- Doesn't reduce latency
- Won't help if model > 1 GPU
- Each GPU loads full model

```yaml
apiVersion: apps/v1
kind: Deployment
spec:
  replicas: 8  # ⟵ Linear scaling
  template:
    spec:
      containers:
      - name: model
        resources:
          limits:
            nvidia.com/gpu: 1
```



Tensor Parallelism Splits Layers Horizontally

How it works

- Single server, all GPUs cooperate
- NVLink: 600 GB/s (50× faster than network)
- All GPUs process same layer in parallel

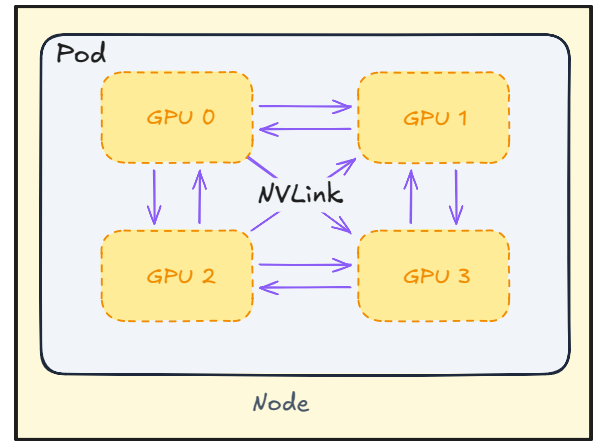
Trade-offs

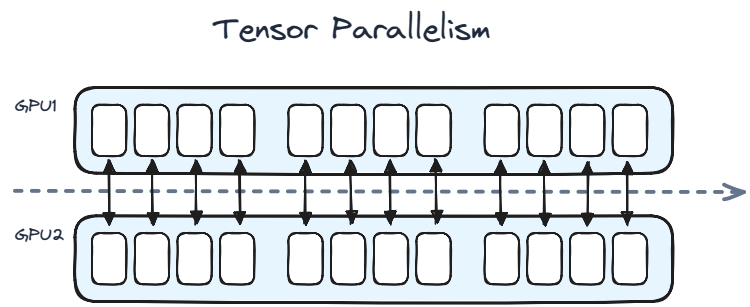
Advantages:

- Lowest latency - parallel processing
- Simplest setup - single Kubernetes pod

Limitations:

- Limited scale - max 8 GPUs per server

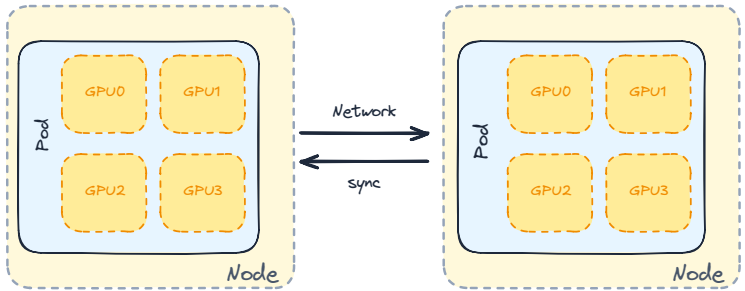




Pipeline Parallelism Splits Layers Vertically

How it works

- Model layers distributed across multiple nodes
- Network: 12-25 GB/s (24-50× slower than NVLink)
- Sequential processing with micro-batching



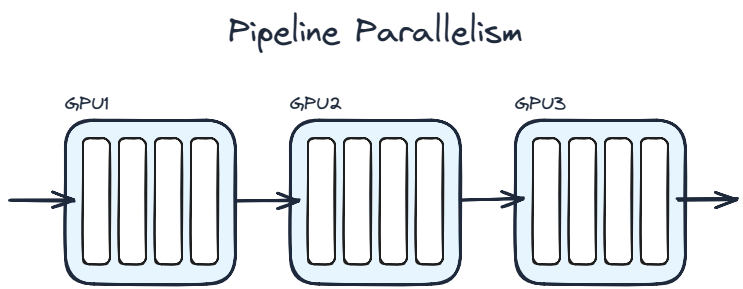
Trade-offs

Advantages:

- Unlimited scale - beyond single server
- Huge models - 100B+ parameters

Limitations:

- More complex - Ray.io, gang scheduling needed

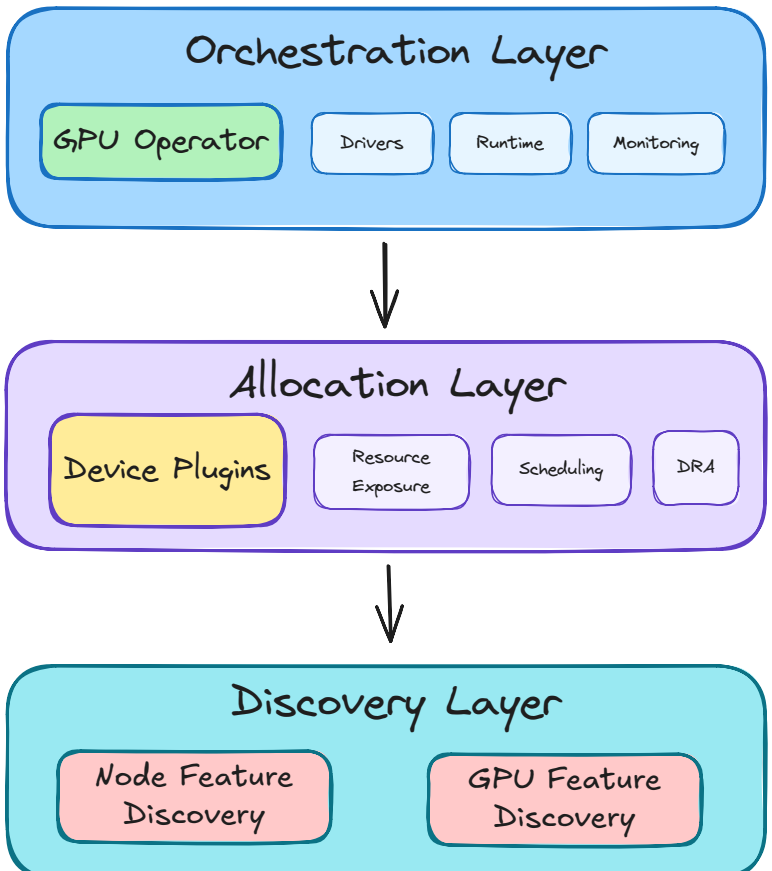


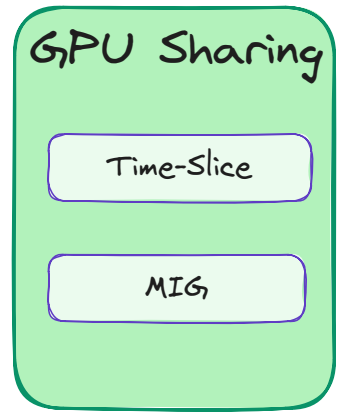
Tensor within nodes, pipeline across nodes

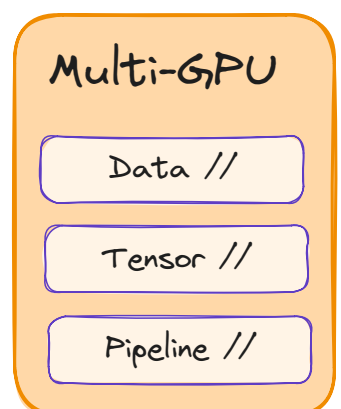
Aspect	Tensor Parallelism	Pipeline Parallelism
Splits	Within layers (horizontal)	Across layers (vertical)
Communication	After every layer	Between stages
Bandwidth	Very high (600 GB/s NVLink)	Network (12-25 GB/s, 24-50× slower)
Latency	Lower (parallel processing)	Higher (sequential processing)
Complexity	Simple (single pod)	Complex (distributed orchestration)

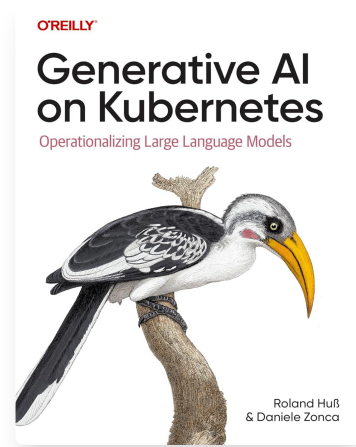
Best practice: Tensor parallelism within nodes + Pipeline across nodes for models exceeding single-server capacity

GPUs on Kubernetes: Unlocked











bit.ly/genaik8s-raw