Cluster Management & Orchestration

SLURM, Kubernetes, Gang Scheduling, and the Art of Operating 10,000-GPU Clusters

Chapter 16 ~42 pages Prerequisite: Ch 14 (Networking), Ch 15 (Storage)

Next: Ch 17 (Telemetry)

> *"A 10,000-GPU cluster running at 60% utilization costs the same as one running at 85% utilization. The difference is the scheduler. Getting scheduling right is worth more than the next GPU generation."*
>
> — Production AI Infrastructure Maxim

A Principal Performance Architect who can design the perfect GPU kernel but cannot schedule, monitor, and operate a GPU cluster at scale is only half-equipped. The orchestration layer — the software that decides which workload runs on which GPU, when, and in what configuration — is where the economics of AI infrastructure are won or lost. A poorly-configured scheduler on a 1,000-GPU cluster at $3/GPU-hr wastes **$12,600 per day** for every 10% of avoidable idle time. This chapter provides a complete, hands-on guide to the two dominant cluster management systems in AI infrastructure: **SLURM** for large-scale HPC and training, and **Kubernetes** for cloud-native serving and multi-tenant platforms.

LEARNING OBJECTIVES BY LEVEL

- **Foundational:** Understand why cluster schedulers exist, what SLURM and Kubernetes do, and how distributed training jobs are different from normal batch jobs.
- **Intermediate:** Write production SLURM job scripts, configure QOS/fair-share policies, deploy GPU workloads on Kubernetes, and diagnose pending-job failures.
- **Principal:** Design multi-tenant scheduling architectures, reason about MTBF/cost tradeoffs, choose between SLURM and Kubernetes for any given workload, and build fault-tolerant 1,000+ GPU training pipelines with quantified failure-recovery economics.

16.1 The Cluster Management Problem: Utilization, Fairness, and Efficiency

Why do we need specialized cluster schedulers at all? Consider a 512-GPU cluster with 20 teams submitting jobs. Without a scheduler, every team would SSH to nodes directly, GPU contention would be resolved by whoever got there first, and

there would be no accounting of who used what. With a scheduler, the cluster becomes a *shared resource with policies*: fairness, priority, topology placement, accounting, and fault recovery are all encoded as configuration rather than convention.

The economics are stark. An H100 SXM node with 8 GPUs costs approximately $300,000 capital. At $0.40/GPU-hr (amortized TCO for on-premise), a 512-GPU cluster costs $204 per hour to operate. Improving cluster utilization from 65% to 85% recovers **$40.8/hr** — over $357,000 per year from scheduling improvements alone, without purchasing a single additional GPU. (See Chapter 5 for full TCO derivations.)

16.1.1 The Three Scheduling Problems

Cluster management for AI workloads involves three distinct, sometimes conflicting, optimization problems:

PROBLEM 1: GANG SCHEDULING (ALL-OR-NOTHING)

A 512-GPU distributed training job requires ALL 512 GPUs to start simultaneously. If even one is unavailable, the entire job stalls at the NCCL initialization barrier. Unlike traditional HPC, where a job can start with partial resources and grow, LLM training requires perfect collective synchronization from step 0. The scheduler must either allocate all requested resources or allocate none — this is the "gang" constraint.

PROBLEM 2: TOPOLOGY AWARENESS

Not all GPUs are equal from a communication perspective. Two GPUs connected by NVLink (900 GB/s) can AllReduce a 1 GB gradient in 2.2 ms. Two GPUs communicating via InfiniBand NDR across two spine hops need ~0.8 ms per message, making large AllReduces 5–10× slower. The scheduler must place tensor-parallel groups within a single node (NVLink domain) and pipeline-parallel stages on the same ToR switch — or pay the bandwidth penalty in every training step for the entire run.

PROBLEM 3: MULTI-TENANCY AND FAIRNESS

A research cluster serves multiple teams with different workload types: a team running a 7-day 1,024-GPU pre-training job (high priority, needs guaranteed allocation), a team running hyperparameter search (medium priority, many small jobs, tolerates preemption), and a team running CI/CD evaluation (low priority, short jobs, needs fast queue turnaround). The scheduler must enforce fairness without creating priority inversions, and must preempt lower-priority jobs gracefully when higher-priority ones arrive.

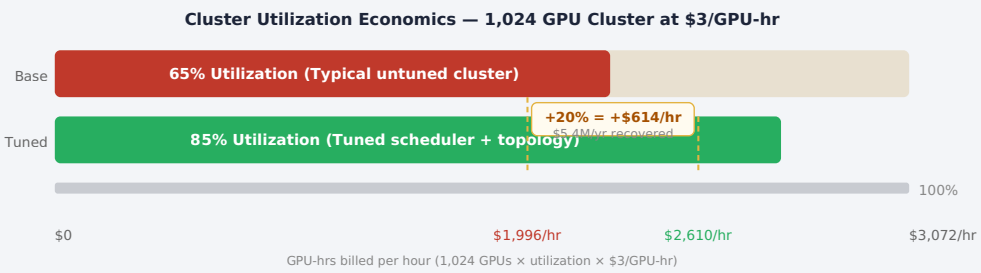


Fig 16.1 — Utilization economics on a 1,024 GPU cluster. Each 1% of utilization improvement at $3/GPU-hr is worth $30.7/hr or $269K/yr. Scheduling improvements from 65%→85% recover $614/hr without any hardware investment.

16.2 SLURM Architecture Deep Dive

SLURM (Simple Linux Utility for Resource Management) is the dominant scheduler for HPC and large-scale AI training clusters. Understanding its internal architecture is prerequisite to tuning it — performance issues almost always trace back to a misconfigured daemon, plugin, or account hierarchy.

16.2.1 The Three Core Daemons

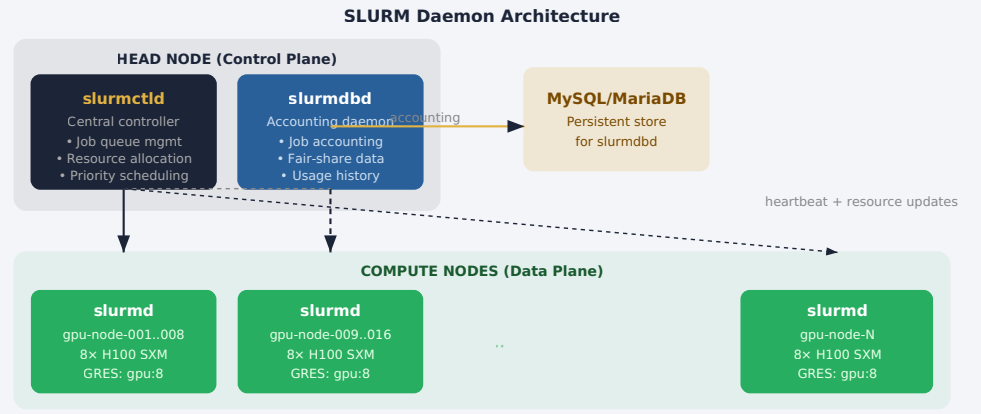


Fig 16.2 — SLURM's three-daemon architecture. slurmctld is the central brain — if it fails, no new jobs start but running jobs continue. slurmdbd persists all accounting data for fair-share calculation. slurmd on each compute node manages local job execution and reports GPU health.

16.2.2 SLURM Configuration Essentials

The `/etc/slurm/slurm.conf` file controls all scheduling behavior. For AI workloads, three parameters dominate performance:

```
# /etc/slurm/slurm.conf — Critical AI cluster settings

SchedulerType=sched/backfill       # REQUIRED: enables backfill for smaller jobs to fill gaps
SchedulerParameters=bf_max_job_test=1000,bf_resolution=30,bf_continue=true
# bf_max_job_test: check up to 1000 pending jobs for backfill opportunities
# bf_resolution=30: evaluate in 30-second time windows
# bf_continue: don't stop backfill when a reservation is encountered

PriorityType=priority/multifactor   # Multi-factor priority (fair-share + QOS + age)
PriorityWeightAge=1000              # Weight for job age (prevents starvation)
PriorityWeightFairshare=10000       # Weight for fair-share (most important factor)
PriorityWeightJobSize=500           # Slight bonus for larger jobs (fills cluster efficiently)
PriorityWeightQOS=5000              # QOS priority override weight
PriorityDecayHalfLife=14-00:00:00   # Fair-share usage decays over 14 days

GresTypes=gpu                       # Register GPU as a generic resource
SelectType=select/cons_tres          # Trackable resources (CPUs, mem, GPUs)
SelectTypeParameters=CR_Core_Memory  # Allocate at core+memory granularity

TaskPlugin=task/cgroup             # REQUIRED: isolate jobs via Linux cgroups
TaskPluginParam=Cores,Sockets,RAM    # Enforce CPU affinity and memory isolation

PreemptType=preempt/qos             # Enable QOS-based preemption
PreemptMode=REQUEUE                  # Preempted jobs are requeued (not cancelled)
PreemptExemptTime=300               # Grace period: 5 min before preemption kicks in

# Node definition with GPU GRES
NodeName=gpu[001-512] Gres=gpu:h100:8 CPUs=96 RealMemory=2048000 State=UNKNOWN
# Gres=gpu:h100:8 → 8 H100 GPUs per node (type-qualified GRES)
```

16.2.3 GRES Configuration — The GPU Registration Chain

The GPU resource registration chain spans three files. Getting any one wrong causes GPUs to be invisible to SLURM. This is the #1 cause of "why can't I request more than X GPUs" support tickets.

```
# Step 1: /etc/slurm/gres.conf (on EACH compute node — must match NodeName Gres= in slurm.conf)
AutoDetect=nvml                    # Auto-detect H100s via NVML (recommended)
Name=gpu Type=h100 File=/dev/nvidia0
Name=gpu Type=h100 File=/dev/nvidia1
Name=gpu Type=h100 File=/dev/nvidia2
Name=gpu Type=h100 File=/dev/nvidia3
Name=gpu Type=h100 File=/dev/nvidia4
Name=gpu Type=h100 File=/dev/nvidia5
Name=gpu Type=h100 File=/dev/nvidia6
Name=gpu Type=h100 File=/dev/nvidia7

# Step 2: Verify GRES registration after slurmd restart
scontrol show node gpu001 | grep -E "Gres=|GresUsed="
# Expected: Gres=gpu:h100:8(IDX:0,1,2,3,4,5,6,7)
# If Gres=gpu:0 → gres.conf mismatch or slurmd not restarted

# Step 3: Test GPU allocation
srun --nodes=1 --gres=gpu:2 nvidia-smi
# Should show exactly 2 GPUs visible via CUDA_VISIBLE_DEVICES
```

16.3 SLURM Job Scripting for Multi-Node GPU Training

The job script is the contract between the researcher and the scheduler. A well-written script encodes resource requirements precisely, sets up the distributed environment correctly, handles signals for graceful preemption, and always exits with a meaningful code. A poorly written script wastes GPUs, causes NCCL hangs, and leaves zombie processes on nodes.

16.3.1 Production Job Script Anatomy

```
#!/bin/bash
#SBATCH --job-name=llama3-70b-pretrain
#SBATCH --nodes=64                  # 64 nodes × 8 GPUs = 512 GPUs total
#SBATCH --ntasks-per-node=8         # One rank per GPU (standard for PyTorch DDP)
#SBATCH --gres=gpu:h100:8           # 8 H100s per node (type-qualified)
#SBATCH --cpus-per-task=12          # 12 CPU cores per rank (96 CPUs / 8 GPUs)
#SBATCH --mem=240G                  # 30 GB per rank (240 GB / 8 ranks per node)
#SBATCH --time=168:00:00            # 7-day wall time limit
#SBATCH --partition=gpu-train       # Training partition (not inference)
#SBATCH --qos=high-priority         # QOS tier — affects scheduling priority
#SBATCH --account=team-research     # Accounting account for fair-share
#SBATCH --output=/checkpoint/slurm/llama3-%j.out
#SBATCH --error=/checkpoint/slurm/llama3-%j.err
#SBATCH --switches=8@00:30:00       # Prefer ≤8 ToR switches; relax after 30 min
#SBATCH --requeue                   # Requeue on node failure or preemption

set -euo pipefail

# ── Distributed training environment ──────────────────
export MASTER_ADDR=$(scontrol show hostnames "$SLURM_JOB_NODELIST" | head -1)
export MASTER_PORT=29500
export WORLD_SIZE=$SLURM_NTASKS
export RANK=$SLURM_PROCID              # Set by srun per rank
export LOCAL_RANK=$SLURM_LOCALID      # GPU index within node

# ── NCCL performance tuning ──────────────────────────
export NCCL_IB_DISABLE=0              # Use InfiniBand (not TCP fallback)
export NCCL_IB_HCA="mlx5_0,mlx5_1"  # Specific HCAs for IB (match topology)
export NCCL_SOCKET_IFNAME=eth0        # Fallback interface
export NCCL_TREE_THRESHOLD=0          # Force ring AllReduce for large messages
export NCCL_DEBUG=WARN                # WARN in production; INFO for debugging

# ── Prolog health check ──────────────────────────────
srun --ntasks=$SLURM_NNODES --ntasks-per-node=1 \
     /opt/slurm/bin/gpu_health_check.sh || exit 1
# Runs 1 health check per node: verifies ECC, fabric, DCGM diagnostics

# ── Launch training ──────────────────────────────────
srun --kill-on-bad-exit=1 \          # If ANY rank fails → kill all ranks
     python3 train.py \
       --config llama3_70b.yaml \
       --checkpoint-dir /checkpoint/llama3-70b/ \
       --resume-from-checkpoint auto   # Auto-detect latest checkpoint
```

16.3.2 SIGTERM Handler for Graceful Preemption

When SLURM preempts a job, it sends `SIGTERM` to all processes (followed by `SIGKILL` after the configured grace period). Without a signal handler, the job dies immediately, losing the current checkpoint. With a handler, the job checkpoints at the current step and exits cleanly — SLURM then requeues it to resume from the checkpoint. On a 7-day run, this difference is **$4,608 of wasted compute vs. $0** for each preemption event at $3/GPU-hr on 512 GPUs.

```
# In train.py — SIGTERM/SIGUSR1 handler for SLURM preemption
import signal, sys, torch.distributed as dist

CHECKPOINT_DIR = "/checkpoint/llama3-70b/"

def checkpoint_and_exit(signum, frame):
    if dist.get_rank() == 0:
        print(f"[RANK 0] Signal {signum} received — saving checkpoint...")
        save_checkpoint(model, optimizer, global_step, CHECKPOINT_DIR)
    dist.barrier()    # All ranks wait for rank 0 to finish writing
    sys.exit(0)       # Exit with code 0 → SLURM interprets as clean exit → requeue

signal.signal(signal.SIGTERM, checkpoint_and_exit)
signal.signal(signal.SIGUSR1, checkpoint_and_exit)  # SLURM sends SIGUSR1 before SIGTERM

# SLURM requeue config in slurm.conf:
# KillWait=120   → 120s between SIGTERM and SIGKILL (checkpoint window)
# RequeueExit=0  → Requeue jobs that exit with code 0
```

16.4 Topology-Aware Scheduling and Gang Scheduling

Topology awareness is the scheduling feature with the highest direct impact on distributed training throughput. Without it, a 512-GPU job scattered across maximum fabric hops can spend 35% of step time in AllReduce instead of 8% — a 27-percentage-point MFU loss that persists for the entire run.

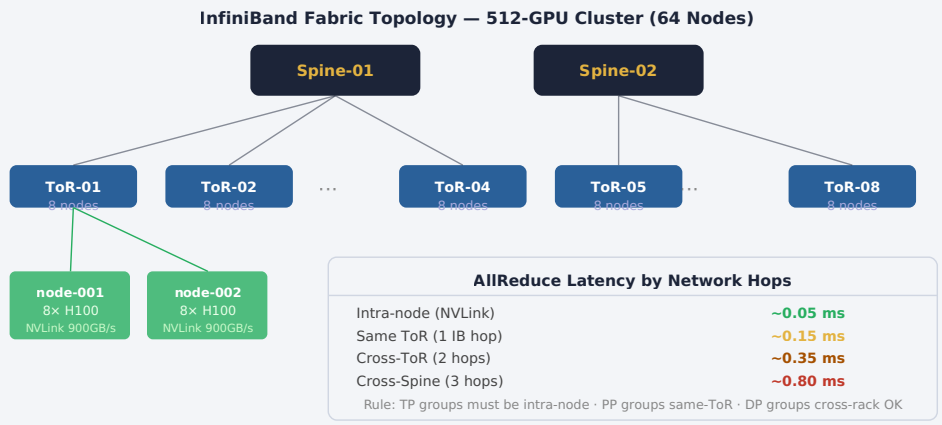


Fig 16.3 — InfiniBand fabric topology for a 512-GPU cluster. Communication cost increases 16× from intra-node (NVLink, 0.05 ms) to cross-spine (3 hops, 0.80 ms). Tensor-parallel groups must stay within the 900 GB/s NVLink domain; only data-parallel gradient syncs can tolerate cross-rack placement.

16.4.1 Gang Scheduling for Distributed Training

Gang scheduling ensures all tasks of a distributed job start simultaneously. Without it, Rank 0 might launch and begin NCCL initialization while Rank 127 is still waiting for its allocation — causing a timeout that kills the entire job. SLURM implements gang scheduling natively through `srun`: it launches all tasks but holds them at a synchronization barrier until all are ready, then releases the entire cohort simultaneously.

> **SLURM GANG SCHEDULING — WHY THE DEFAULT WORKS**
>
> SLURM's `srun` command implements all-or-nothing launch semantics: all task slots are allocated before any task receives CPU time. The scheduler won't start a job until ALL nodes in the allocation are available (no partial start). This is different from MPI's non-gang-safe mode where processes can start individually. For NCCL-based training, this is the correct behavior — never configure `--ntasks-per-node` less than the number of GPUs you intend to use, or you will lose gang semantics.

```
# Verifying gang start — all ranks should show identical start timestamps
sacct -j ${SLURM_JOB_ID} --format=JobID,Start,Elapsed --parsable2
# Example output (correct gang start):
# JobID|Start|Elapsed
# 12345.0|2024-11-15T14:30:00|00:00:01
# 12345.1|2024-11-15T14:30:00|00:00:01  ← same second as .0
# 12345.2|2024-11-15T14:30:00|00:00:01  ← correct

# topology.conf for SLURM topological scheduling
SwitchName=spine01 Switches=tor[01-16]
SwitchName=spine02 Switches=tor[17-32]
SwitchName=tor01   Nodes=gpu[001-016]
SwitchName=tor02   Nodes=gpu[017-032]
# ... define all switches

# Job script: request topology constraint
#SBATCH --switches=8@00:30:00   # Up to 8 ToR switches, wait up to 30 min for better placement
# For a 64-node job on 8-node ToRs: 8 switches = all nodes within 8 racks
# This guarantees max 2-hop IB communication (intra-rail)
```

16.5 QOS, Fair-Share, and Multi-Tenant Policies

Fair-share scheduling solves the problem of preventing any single team from monopolizing a shared cluster. The algorithm is conceptually simple: track how much each account *should* have received (normalized share) vs. how much it *did* receive (usage over a decay window). Accounts that have used less than their share get priority; accounts that have over-consumed get deprioritized. This automatically corrects for bursty usage patterns without requiring manual intervention.

```
# Fair-share priority formula (SLURM multifactor plugin):
Priority = PriorityWeightAge × age_factor
         + PriorityWeightFairshare × fairshare_factor
         + PriorityWeightQOS × qos_factor
         + PriorityWeightJobSize × jobsize_factor

# fairshare_factor = 2^(-effective_usage / normalized_share)
# If team used 0% of their share → factor = 1.0 (max priority)
# If team used 100% of share  → factor = 0.5
# If team used 200% of share  → factor = 0.25
# Decay half-life (14 days) means old usage counts less over time

# ── Setting up a 5-team multi-tenant cluster ──────────────────
sacctmgr add cluster gpu-cluster

# Create top-level accounts with share allocations
sacctmgr add account research    Share=40 Description="Research Division"
sacctmgr add account inference   Share=30 Description="Inference Serving"
sacctmgr add account evaluation  Share=20 Description="Model Evaluation"
sacctmgr add account dev         Share=10 Description="Development/CI"

# Create users under accounts
sacctmgr add user alice  Account=research   Partition=gpu-train
sacctmgr add user bob    Account=inference  Partition=gpu-infer
sacctmgr add user ci_bot Account=evaluation Partition=gpu-eval DefaultQOS=ci

# Create QOS tiers with resource limits
sacctmgr add qos high-priority MaxWall=336:00:00 Priority=1000 GrpTRES=gpu=512
sacctmgr add qos standard      MaxWall=168:00:00 Priority=500  GrpTRES=gpu=256
sacctmgr add qos batch         MaxWall=72:00:00  Priority=100  GrpTRES=gpu=64
sacctmgr add qos ci            MaxWall=4:00:00   Priority=200  GrpTRES=gpu=16 \
    MaxJobsPerUser=5 MaxSubmitJobsPerUser=20

# View fair-share status (sshare shows actual vs. effective usage)
sshare -l -u alice,bob
# Account   User   RawShares RawUsage NormShares NormUsage FairShare
# research  alice  40        2847.2   0.40       0.12      0.97  ← under-utilized, gets boost
# inference bob    30        8432.1   0.30       0.52      0.71  ← over-utilized, deprioritized
```

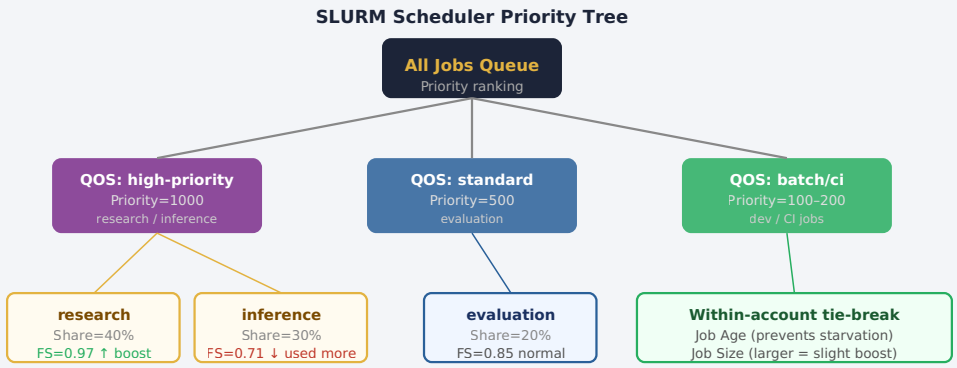


Fig 16.4 — SLURM scheduler priority tree. Jobs compete first by QOS tier (1000 vs. 500 vs. 100), then by fair-share within their QOS tier, then by age. The research team (fair-share=0.97) wins over inference (0.71) at the same QOS level because it has used less of its allocated share.

16.6 Kubernetes for AI — Architecture and GPU Scheduling

Kubernetes has become the dominant platform for AI inference serving, multi-tenant research platforms, and increasingly, large-scale training. Its containerized resource model, rich RBAC, and native integration with cloud auto-scaling make it the right choice wherever workloads look more like microservices than batch HPC jobs.

GPUs enter Kubernetes through the **NVIDIA device plugin**, which runs as a DaemonSet and advertises GPU resources to the Kubernetes API server. From the scheduler's perspective, GPUs are just another extended resource: `nvidia.com/gpu: 8` in a pod spec is equivalent to requesting 8 CPU cores — the scheduler finds a node with 8 available GPUs and places the pod there.

```yaml
# GPU deployment: PyTorch training pod with resource requests
apiVersion: v1
kind: Pod
metadata:
  name: llama-finetune-worker
  labels:
    app: llama-finetune
    task-type: training
spec:
  containers:
  - name: trainer
    image: nvcr.io/nvidia/pytorch:24.05-py3
    resources:
      requests:
        nvidia.com/gpu: "8"        # Request 8 GPUs
        cpu: "96"                  # Match CPU count to GPUs
        memory: "512Gi"
      limits:
        nvidia.com/gpu: "8"        # Limits must equal requests for GPUs
        cpu: "96"
        memory: "512Gi"
    env:
    - name: NCCL_IB_DISABLE
      value: "0"
    - name: NVIDIA_VISIBLE_DEVICES
      value: all                   # Device plugin sets CUDA_VISIBLE_DEVICES
  tolerations:
  - key: nvidia.com/gpu
    operator: Exists
    effect: NoSchedule             # Required: GPU nodes tainted to prevent CPU-only workloads
  nodeSelector:
    nvidia.com/gpu.product: "NVIDIA-H100-80GB-HBM3"   # Match specific GPU model

---
# Multi-tenant namespace isolation with ResourceQuota
apiVersion: v1
kind: ResourceQuota
metadata:
  name: research-team-quota
  namespace: research
spec:
  hard:
    requests.nvidia.com/gpu: "256"  # Max 256 GPUs for research namespace
    limits.nvidia.com/gpu: "256"
    requests.cpu: "3072"
    requests.memory: "12Ti"
```

```
    pods: "500"
    count/deployments.apps: "50"
```

16.6.1 Diagnosing Pending GPU Pods

The most common Kubernetes support request is "my GPU pod is stuck in Pending." There are exactly five root causes, and they can be diagnosed in under two minutes with the right command sequence:

```
# Step 1: Get the exact scheduling failure reason
kubectl describe pod <pod-name> -n <namespace>
# Look for Events section at the bottom — this is the diagnosis:

# Cause 1: Insufficient GPU resources
# Events: 0/8 nodes are available: 8 Insufficient nvidia.com/gpu
# Fix: Check GPU availability (kubectl describe nodes | grep nvidia.com/gpu)
#      or reduce request

# Cause 2: ResourceQuota exceeded
# Events: exceeded quota: research, requested: requests.nvidia.com/gpu=8,
#         used: requests.nvidia.com/gpu=252, limited: requests.nvidia.com/gpu=256
# Fix: Delete idle pods or increase quota

# Cause 3: Affinity/taint mismatch
# Events: 0/8 nodes are available: 8 node(s) had untolerated taint
# Fix: Add toleration for nvidia.com/gpu NoSchedule

# Cause 4: Node selector mismatch
# Events: 0/8 nodes are available: 8 node(s) didn't match Pod's node affinity
# Fix: kubectl get nodes --show-labels | grep gpu.product

# Cause 5: Device plugin not running
kubectl get pods -n kube-system | grep device-plugin
# If no nvidia-device-plugin DaemonSet pods → GPUs not registered with k8s
# Fix: Helm install nvidia-device-plugin or re-register
```

16.7 Gang Scheduling on Kubernetes: Volcano and Kueue

Kubernetes' default scheduler is not gang-scheduler aware. Without a gang scheduling plugin, a 512-GPU distributed training job (512 pods) can partially start: pods 1–400 might acquire their GPUs while pods 401–512 wait in a resource contention queue. The running pods spin at the NCCL barrier, burning 400 GPUs worth of cost ($1,200/hr) producing zero useful work until the remaining pods are scheduled — or a 10-minute NCCL timeout kills everything. This is why Volcano and Kueue exist.



Volcano Job YAML — 512-GPU Training with PodGroup

```
apiVersion: batch.volcano.sh/v1alpha1
kind: Job
minAvailable: 512 # Gang constraint: ALL 512 or NONE
policies: # Restart policy
- event: PodFailed action: RestartJob # Whole job restart on failure
queue: research-high-pri # Volcano queue
priorityClassName: high # K8s PriorityClass
plugins:
pytorch: [] # PyTorch plugin: sets MASTER_ADDR etc.
svc: [] # Service discovery for ranks
```

Fig 16.5 — Volcano gang scheduling lifecycle. The critical difference from default K8s: pods are held in "Pending" until ALL minAvailable=512 pods can be simultaneously placed. Without this, partial starts waste GPU-hours at the NCCL synchronization barrier.

16.8 Fault Tolerance: MTBF, SDC, and Node Quarantine

At scale, hardware failure is not an exception — it is the steady state. A cluster of 1,024 H100 GPUs has approximately 8,192 individual components (GPU dies, NVLink connections, memory stacks, PCIe lanes). With a mean time between failures (MTBF) of ~100,000 hours per component, the expected time to the *first* failure across the cluster is just 12 hours. Meta's production Llama 3 training runs on 24,000 GPUs experienced ~50% fewer cluster interruptions after systematically investing in automated failure detection and recovery — demonstrating that engineering the recovery path is as important as engineering the training algorithm.

16.8.1 The Three-Layer Fault Detection Hierarchy

> **LAYER 1: HARDWARE TELEMETRY (DCGM)**
>
> NVIDIA DCGM runs as a service on every GPU node and exposes hardware health metrics at 10-second intervals. Key metrics to monitor: GPU ECC single-bit errors (DCGM_FI_DEV_ECC_SBE_AGG_TOTAL — more than 10/hour predicts imminent failure), temperature (throttle at 83°C, thermal runaway above 87°C), PCIe replay counter (non-zero = signal integrity issue), and NVLink error counter (any errors = degraded bandwidth). See Chapter 17 for complete DCGM metric catalog.

> **LAYER 2: SLURM PROLOG/EPILOG HEALTH CHECKS**
>
> Prolog scripts run on every node before a job starts; epilog scripts run after it ends. A prolog that exits non-zero causes SLURM to mark the node as drained and move the job to the next available set of nodes. This is the first line of defense against placing a training job on a degraded node.

LAYER 3: SILENT DATA CORRUPTION (SDC) DETECTION

SDC is the worst failure mode: the GPU appears healthy (no ECC errors), but produces wrong computation results. SDC manifests as training loss divergence or incorrect model outputs and is notoriously hard to detect. The state-of-the-art approach is to run "golden inputs" tests — pre-computed tensors with known correct outputs — and verify exact match. Meta, Google, and NVIDIA all run SDC screening tests before assigning nodes to production training jobs.

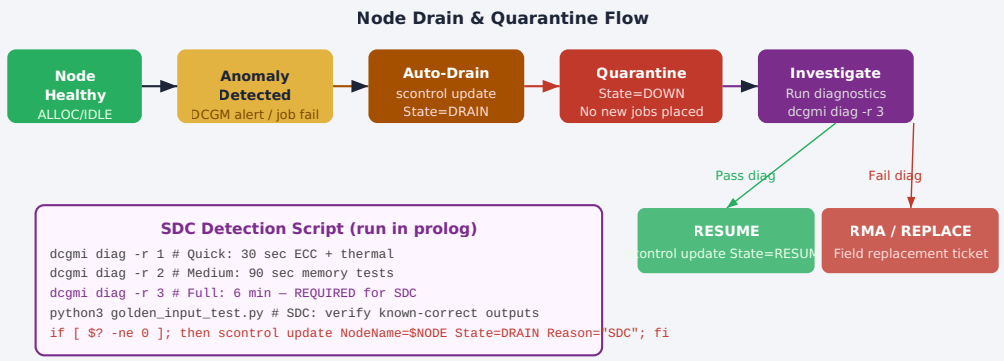


Fig 16.6 — Node drain and quarantine flow. Automated drain on anomaly detection prevents a degraded node from silently corrupting a multi-week training run. The SDC prolog test (dcgmi diag -r 3) takes 6 minutes but saves days of lost training on a node with a defective GPU that appears ECC-clean.

```
# ── SLURM prolog: node health check (runs before each job) ──────────────
#!/bin/bash
# /opt/slurm/prolog.d/01-gpu-health.sh

NODE_PASS=1

# Check 1: GPU count (should be 8 on all nodes)
GPU_COUNT=$(nvidia-smi --query-gpu=name --format=csv,noheader | wc -l)
if [ "$GPU_COUNT" -ne 8 ]; then
    scontrol update NodeName=${SLURM_JOB_NODELIST} State=DRAIN \
        Reason="GPU count mismatch: expected 8, got ${GPU_COUNT}"
    exit 1
fi

# Check 2: ECC error rate (accumulated since last reset)
ECC_ERRORS=$(dcgmi dmon -e 160 -c 1 | awk 'NR>1{sum+=$2}END{print sum+0}')
if [ "$ECC_ERRORS" -gt 100 ]; then
    scontrol update NodeName=${SLURM_JOB_NODELIST} State=DRAIN \
        Reason="ECC errors exceed threshold: ${ECC_ERRORS} SBE"
    exit 1
fi

# Check 3: NVLink errors (any = fabric degradation)
NVLINK_ERR=$(nvidia-smi nvlink -e -i 0 | grep -c "Error Count" || true)
if [ "$NVLINK_ERR" -gt 0 ]; then
    scontrol update NodeName=${SLURM_JOB_NODELIST} State=DRAIN \
        Reason="NVLink errors detected"
    exit 1
fi

echo "[PROLOG] GPU health check PASSED on ${SLURM_JOB_NODELIST}"
exit 0
```

16.9 Multi-Tenant GPU Sharing: MIG, MPS, and Time-Slicing

Not every workload needs a full GPU. A CI/CD model evaluation job that runs 1,000 forward passes needs perhaps 2 GB of VRAM and 20% of SM utilization — assigning it a whole H100 wastes 80 GB of HBM and 80% of compute. The three GPU sharing mechanisms — MIG, MPS, and time-slicing — address this at different granularities, with different isolation and performance guarantees.

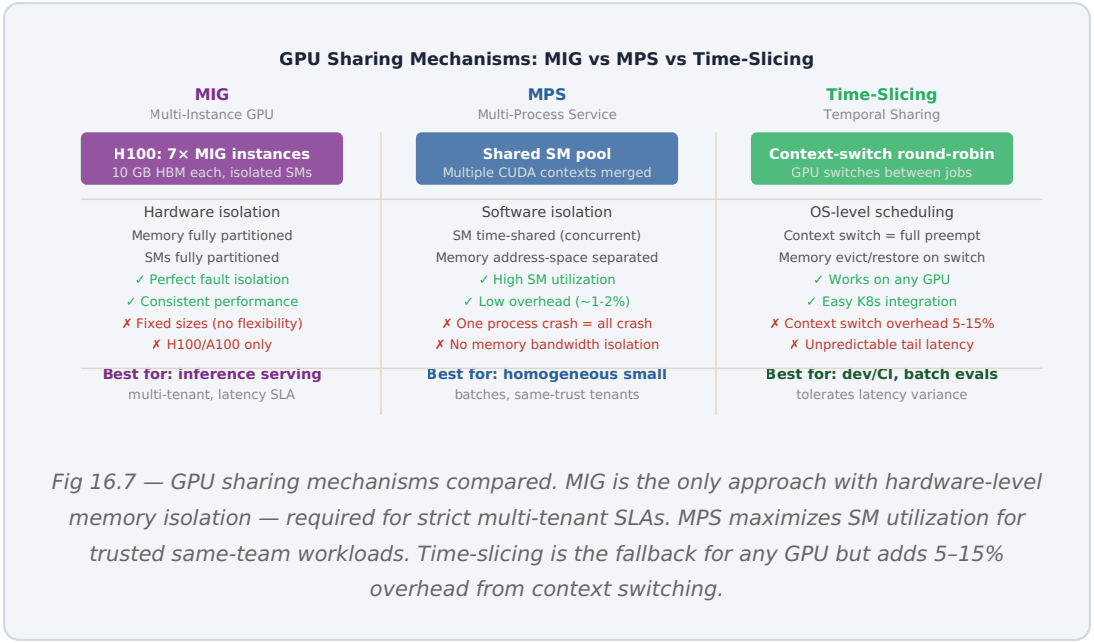


Fig 16.7 — GPU sharing mechanisms compared. MIG is the only approach with hardware-level memory isolation — required for strict multi-tenant SLAs. MPS maximizes SM utilization for trusted same-team workloads. Time-slicing is the fallback for any GPU but adds 5–15% overhead from context switching.

```
# MIG configuration on H100 (7× 1g.10gb instances)
nvidia-smi mig -cgi 1g.10gb,1g.10gb,1g.10gb,1g.10gb,1g.10gb,1g.10gb,1g.10gb -C

# Verify MIG instances
nvidia-smi mig -lgi
# Expected: 7 GPU instances, each with 1 GPC, 10 GB memory

# Kubernetes: expose MIG instances to pods
# With MIG strategy="mixed" in device plugin config:
resources:
  limits:
    nvidia.com/mig-1g.10gb: "1"  # Request one 10GB MIG slice

# MPS: enable for all processes on GPU 0
export CUDA_MPS_PIPE_DIRECTORY=/tmp/nvidia-mps
export CUDA_MPS_LOG_DIRECTORY=/tmp/nvidia-log
nvidia-cuda-mps-control -d          # Start MPS daemon
echo "set_active_thread_percentage 50" | nvidia-cuda-mps-control
# Limits each MPS client to 50% of SMs (prevents one job starving others)

# K8s time-slicing config (device plugin configmap)
version: v1
flags:
  migStrategy: none
sharing:
  timeSlicing:
    replicas: 4         # Present each physical GPU as 4 virtual GPUs
# Pods can now request nvidia.com/gpu: 1 and share a physical GPU
# WARNING: No memory isolation — use only for trusted workloads
```

16.10 SLURM vs. Kubernetes — Decision Framework

The choice between SLURM and Kubernetes is not a matter of one being better — it's a matter of fit to workload characteristics. The correct answer for most large organizations is both: SLURM for training, Kubernetes for inference, with a lightweight bridge (MLFlow, Kubeflow, Weights & Biases) for experiment tracking across both. Meta's production architecture is precisely this: SLURM-like gang scheduling semantics for physical training clusters, Kubernetes-style container management for the inference serving layer.

> **DECISION RULE FOR NEW CLUSTERS**
>
> If your primary workload is multi-node distributed training (>16 nodes) and you have InfiniBand fabric and a dedicated HPC team: use SLURM. If your primary workload is inference serving, multi-tenant research with container images, or you're deploying on cloud (EKS/GKE/AKS): use Kubernetes. If you have both: use both, with SLURM for training and K8s for serving.

Dimension	SLURM	Kubernetes	Winner
Gang scheduling	Native (built-in)	Requires Volcano/Kueue	**SLURM**
Multi-node training	Excellent (native MPI/srun)	Good (with PyTorch operator)	**SLURM**
Topology awareness	Native (topology.conf)	Limited (topology manager)	**SLURM**
Inference serving	Not designed for it	Excellent (native autoscale)	**Kubernetes**
Container support	Singularity/Apptainer	Docker/containerd (native)	**Kubernetes**
Auto-scaling nodes	Manual add	Cluster autoscaler (native)	**Kubernetes**
Multi-tenancy	Accounts + QOS	Namespaces + RBAC + quotas	**Tie**
Fair-share accounting	Native (sacctmgr)	Requires Kueue/custom	**SLURM**
Job preemption	Mature (QOS-based)	PriorityClasses (works well)	**SLURM**
Monitoring	sacct + DCGM	Prometheus/Grafana (native)	**Kubernetes**
Rolling updates	Not applicable	Native deployment strategy	**Kubernetes**
Production examples	Meta training, NERSC, NVIDIA	Google serving, OpenAI, AWS	**Both**

16.11 Production Operations Playbook

16.11.1 Daily Cluster Health Checks

```
#!/bin/bash
# cluster_health_check.sh — Run daily via cron
echo "=== SLURM CLUSTER HEALTH REPORT $(date) ==="

# Node state summary
sinfo --format="%P %n %T %C %G" | column -t
echo ""

# Drained nodes (require investigation)
echo "DRAINED NODES:"
sinfo --states=drained,down --format="%n %T %E" | column -t

# GPU utilization (DCGM)
echo "CLUSTER GPU UTILIZATION:"
dcgmi dmon -e 203,1001,1002 -c 1 | \
    awk 'NR>1{util+=$2;mem+=$3;count++} END \
    {printf "Avg GPU: %.1f%%  Avg HBM: %.1f%%\n", util/count, mem/count}'

# Long-pending job alert (pending > 60 min → possible gang deadlock)
echo "LONG-PENDING JOBS (>60 min):"
squeue -t PENDING --format="%i %u %D %G %V" | \
    awk -v now="$(date +%s)" '{
      cmd="date -d "$5" +%s"; cmd | getline sub; close(cmd);
      if(now - sub > 3600) print $0, "("int((now-sub)/60)"min)"
    }'
```

16.11.2 Capacity Planning Formula

Capacity planning for AI clusters requires reasoning from the training roadmap backward to GPU-hours, then forward to fleet size. The key insight: you need to provision for *peak demand with N+1 fault tolerance*, not average utilization.

```
# Capacity planning: GPU-hours → fleet size

Fleet_GPUs = (Weekly_GPU_Hours_Demand × Safety_Factor) / (168 hrs × Target_Utilization)

Example: Team trains 3 models/week, each needing 1M GPU-hours
Weekly_demand = 3 × 1,000,000 = 3,000,000 GPU-hours/week
Safety_factor = 1.3  # 30% headroom for failures, reruns, experiments
Target_util   = 0.80  # 80% realistic sustained utilization
Hours_per_week = 168

Fleet_GPUs = (3,000,000 × 1.3) / (168 × 0.80) = 29,018 GPUs ≈ 30,000 GPUs

# Annual cost at $3/GPU-hr (amortized TCO):
Annual_cost = 30,000 × $3 × 8,760 = $789M/year

# Impact of utilization improvement (80% → 85%): same output, fewer GPUs
Fleet_at_85% = 30,000 × (0.80/0.85) = 28,235 GPUs
GPUs_freed = 1,765  →  Savings = 1,765 × $3 × 8,760 = $46.4M/year
```

WORKED EXAMPLE — FAULT TOLERANCE COST ANALYSIS: 1,024-GPU LLM PRE-TRAINING

Scenario: You're planning a 7-day, 1,024-GPU LLM pre-training job on H100s at $3/GPU-hr. Cluster MTBF is 50 hours per node (8 GPUs per node). Assume no fault tolerance vs. a well-engineered checkpoint/requeue pipeline. Quantify the GPU-hour cost of each approach.

Given:

- Duration: 7 days = 168 hours
- GPUs: 1,024 (128 nodes × 8 GPUs)
- Node MTBF: 50 hours
- Cost: $3/GPU-hr
- Checkpoint time: 3 minutes
- Recovery time (no fault tolerance): full restart from hour 0
- Recovery time (with fault tolerance): resume from last checkpoint (every 30 min)

Step 1: Expected number of failures

```
Nodes = 128
Run_duration = 168 hours
Expected_failures = Nodes × (Run_duration / MTBF)
                  = 128 × (168 / 50)
                  = 128 × 3.36 = 430 expected failures over the run

# But failures follow Poisson: time to FIRST failure is the critical metric
Time_to_first_failure = MTBF / Nodes = 50 / 128 = 0.39 hours ≈ 23 minutes
# A 1,024-GPU cluster will see its first failure within 23 minutes on average.
# Without fault tolerance, a job that restarts from scratch loses ALL prior work.
```

Step 2: GPU-hours wasted without fault tolerance

```
# Assume failures are detected and job restarts from hour 0 each time
# Poisson process: Nth failure follows Gamma distribution
# Simplified: model as sequential → each failure wastes hours since last successful start

# In practice, at 0.39 hr between failures, the job can never complete
# Without checkpointing: infinite loop (job never reaches hour 168)

# Conservative model: assume 1 failure per run at random point (expected: hour 84)
# Wasted GPU-hours (without FT) = GPUs × hours_worked_before_failure
Wasted_GPU_hours = 1024 × 84 = 86,016 GPU-hours wasted per failure
Wasted_cost = 86,016 × $3 = $258,048 wasted per failure

# With 430 failures expected: effectively the job never completes
```

Step 3: GPU-hours wasted WITH fault tolerance (30-min checkpoints)

```
# Maximum work lost = work since last checkpoint = 30 minutes
Checkpoint_interval = 0.5 hours
Checkpoint_time = 0.05 hours (3 min)

# Per failure: wasted compute = replay from checkpoint + checkpoint overhead
Wasted_per_failure = 1024 GPUs × (0.5 + 0.05) hrs = 563 GPU-hours
Cost_per_failure = 563 × $3 = $1,690

# Over full 168-hour run with 430 expected failures:
Total_wasted = 430 × 563 = 242,090 GPU-hours wasted
Total_wasted_cost = 430 × $1,690 = $726,700 over the 7-day run

# vs. without FT: job never completes — effectively infinite cost
# Checkpoint overhead (writing 30-min interval): 3 min / 30 min = 10% overhead
# Tuning: use async checkpointing → reduce to <2% overhead
# See Chapter 15 for checkpoint I/O optimization (async, distributed, incremental)
```

Conclusion: Without fault tolerance, a 1,024-GPU training job on a realistic cluster *cannot complete* a 7-day run — it will fail before reaching the end. With a checkpoint-requeue pipeline, the run completes with ~10% overhead from checkpoint writes and failure recovery, costing $726K in wasted GPU-hours over the full run. The engineering investment to build this pipeline (typically 1–2 weeks for a production-grade implementation) pays back in the first training run alone.

16.12 Profiling & Diagnostic Cookbook

When a 512-GPU Training Job is Slow — Systematic Diagnosis

> **DIAGNOSTIC DECISION TREE**
>
> **STEP 1: CHECK MFU**
>
> — compute `achieved_TFLOPS / peak_TFLOPS`. If <40%: scheduling or network issue. If 40–70%: likely communication overhead. If >70%: GPU-side issue (memory, kernel).
>
> **STEP 2: CHECK ALLREDUCE TIME**
>
> — add NCCL timing: `NCCL_TIMING=1`. If AllReduce >15% of step time: topology placement issue or fabric congestion. Check `sinfo --format=%f %n` to verify nodes are on same ToR switches.
>
> **STEP 3: CHECK STRAGGLERS**
>
> — if AllReduce time is normal but throughput varies, one rank is slow. Use `torch.distributed.monitored_barrier()` to identify slow ranks. Check DCGM for thermal throttling on that node's GPUs.
>
> **STEP 4: CHECK CHECKPOINT I/O**
>
> — if training pauses periodically, checkpoint writes are synchronous and blocking. Switch to async checkpointing (Torch Distributed Checkpoint) or reduce checkpoint frequency. See Chapter 15.
>
> **STEP 5: CHECK GANG START**
>
> — run `sacct -j $JOBID --format=JobID,Start --parsable2`. If any rank started more than 5 seconds after the first: partial gang start, scheduler configuration issue.

```
# SLURM job diagnostic commands
squeue -j $JOBID -o "%i %T %D %l %S %V %P %q %u"
# Shows: job state, nodes, time limit, start time, submit time, partition, QOS, user

sacct -j $JOBID --format=JobID,State,ExitCode,CPUTimeRAW,MaxRSS --parsable2
# Post-mortem: exit code, CPU time consumed, max memory used

# Check fair-share balance (why is a job low priority?)
sprio -l -j $JOBID
# Shows: priority breakdown by component (age, fairshare, QOS, jobsize)

# Check node topology placement
scontrol show job $JOBID | grep NodeList
srun --ntasks=1 bash -c "hostname; ibnetdiscover 2>/dev/null | grep $(hostname) | head -5"
# Verify nodes are within expected switch domain

# Kubernetes: GPU utilization across all pods in namespace
kubectl top pods -n research --sort-by=cpu
kubectl describe nodes | grep -A5 "Allocated resources"
```

Key Takeaways

CORE CONCEPTS

- SLURM and Kubernetes solve different problems: SLURM excels at gang-scheduled, topology-aware HPC training; Kubernetes excels at containerized, auto-scaling inference serving. Most large organizations need both.
- Gang scheduling is the non-negotiable constraint for distributed training: all ranks must start simultaneously, or NCCL initialization hangs and GPUs burn money doing nothing. SLURM provides this natively; Kubernetes requires Volcano or Kueue.
- Topology-aware scheduling can recover 5–27 percentage points of MFU by ensuring TP groups stay within the NVLink domain (0.05 ms) and PP groups stay within the same ToR switch (0.15 ms), not scattered across spine switches (0.80 ms).
- Fair-share scheduling automatically corrects cluster equity: teams that have used less of their allocation get boosted priority; over-consumers are deprioritized. The 14-day decay half-life means even heavy users return to fair standing over time.

COST & PRODUCTIVITY IMPACT

- Improving cluster utilization from 65%→85% on a 1,024-GPU cluster at $3/GPU-hr saves $614/hr = $5.4M/year — entirely from scheduling improvements, zero additional hardware.
- A fault-tolerant checkpoint-requeue pipeline is mandatory at scale: on a 1,024-GPU cluster with 50-hour node MTBF, the first failure arrives within 23 minutes on average. Without checkpointing, a 7-day training run cannot complete. With 30-minute checkpointing, recovery cost per failure drops from hundreds of thousands to $1,690.
- Multi-tenant GPU sharing via MIG on H100s allows 7 independent CI/CD evaluation jobs to run concurrently on a single GPU, recovering $0.43/hr from GPUs that would otherwise sit idle between training runs.

PRINCIPAL-LEVEL INSIGHT

- Silent Data Corruption (SDC) is the most dangerous failure mode: the GPU appears healthy by all hardware metrics (no ECC errors, nominal temperature) but produces wrong computation results. The only reliable detection is running known-correct "golden input" validation tests in the prolog before every large training job. No amount of post-hoc analysis recovers a multi-week training run corrupted by SDC.
- The SLURM `--switches` constraint is the highest-ROI single configuration change for multi-node training: it prevents the scheduler from scattering a 512-GPU job across maximum fabric hops. The tradeoff — up to 30 minutes additional queue wait for better placement — pays back in reduced AllReduce latency for the entire multi-day run. At a step time of 1 second, 10% latency improvement = 0.1 GPU-seconds saved per step = hours of compute recovered on million-step runs.

- Kubernetes PriorityClasses and SLURM QOS tiers solve different parts of the same problem: K8s priorities are about preemption order within a namespace; SLURM QOS is about preemption, resource caps, and scheduling boosters simultaneously. For a hybrid cluster, the policy must be consistent across both systems or teams will route jobs to whichever system gives them an advantage.

Review Questions

FOUNDATIONAL Build the mental model — no math required

1. Explain why a distributed training job requires gang scheduling. What happens if Rank 0 starts on GPU-A while Rank 127 is still waiting for its allocation? Why is this worse than simply being slow?
2. What are the three SLURM daemons and what does each one do? If `slurmctld` crashes on the head node, what happens to jobs that are currently running on compute nodes? What happens to new job submissions?
3. A researcher on your team says "just use MPS everywhere — it maximizes GPU utilization." What is the key failure-isolation risk they are ignoring, and when would MIG be the right choice instead?

INTERMEDIATE Apply and calculate — production context

1. Your 256-GPU SLURM cluster has 32 nodes (8 GPUs each) on 4 ToR switches (8 nodes per ToR). A team submits a training job requesting all 256 GPUs without a `--switches` constraint. The scheduler places 64 nodes across all 4 switches. Calculate the AllReduce latency for the tensor-parallel group (8 GPUs on the same node) vs. the data-parallel gradient sync (64 nodes across 4 ToRs, 3-hop worst case). Using the latency values from §16.4, what is the additional time spent in cross-spine AllReduce per step if gradient sync takes 20% of step time at optimal placement?
2. You are configuring fair-share for 4 teams: research (Share=40), inference (Share=30), evaluation (Share=20), dev (Share=10). After one week, the raw usage is: research=1,200 GPU-hrs, inference=3,600 GPU-hrs, evaluation=800 GPU-hrs, dev=200 GPU-hrs (total=5,800). Calculate the normalized share and normalized usage for each team. Which team gets the highest fair-share factor (is most under-utilized relative to their allocation)? Write the `sacctmgr` command to create the account hierarchy.
3. A Kubernetes GPU pod for a batch inference job is stuck in Pending. The describe output shows: "0/12 nodes are available: 12 node(s) had untolerated taint {nvidia.com/gpu: NoSchedule}." Write the YAML stanza to fix this pod spec. Then explain: why is this taint applied to GPU nodes, and what would happen if you removed the taint entirely instead of adding the toleration?

PRINCIPAL LEVEL Design, decide, and defend — architecture reviews

1. Design a multi-tenant GPU cluster for 5 ML teams with 1,024 H100 GPUs. Team A runs large-scale pre-training (1–2 jobs, 512–1,024 GPUs each, 7-day walls, can tolerate preemption with

checkpointing). Team B runs production inference serving (SLA: <50ms p99, 24/7, never preemptible, needs 64 GPUs). Team C runs hyperparameter search (200 concurrent 8-GPU jobs, each 4 hours, tolerates preemption). Teams D and E do development (small jobs, 1–8 GPUs, low priority). Design: SLURM QOS tiers, fair-share allocations, partition layout, and the MIG/MPS/full-GPU assignment policy. Justify every decision quantitatively. What is the expected overall cluster utilization?

2. A 7-day, 512-GPU LLM training run is failing due to node failures at a rate of 1 failure per 8 hours. Current setup: no checkpointing, manual restart from epoch 0. Design a complete fault-tolerant pipeline. Specify: checkpoint interval (justify the math, considering both checkpoint I/O overhead and expected work lost per failure), SLURM requeue configuration, SIGTERM handler code pattern, prolog health check contents, and async vs. synchronous checkpoint strategy. Quantify GPU-hours wasted and money saved versus the current approach at $3/GPU-hr.

3. Your organization is evaluating whether to use SLURM, Kubernetes, or a hybrid for a new 4,000-GPU cluster that will run: (a) 2,000 GPUs dedicated to LLM pre-training jobs (128–512 GPUs each), (b) 1,500 GPUs for 24/7 inference serving with auto-scaling, (c) 500 GPUs for research experiments (mixed sizes, frequent iteration). Design the cluster architecture, justifying the orchestration system choice for each workload segment. Describe the "bridge" layer between systems. What are the three biggest operational risks of a hybrid architecture, and how do you mitigate each?