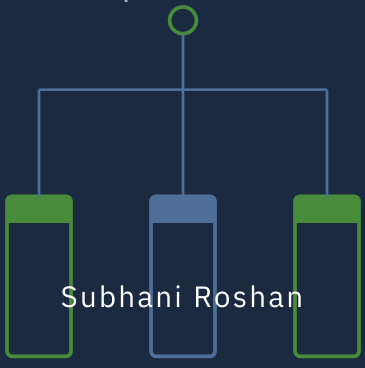
HPC

Infrastructure & Troubleshooting Handbook

A Practical Guide to Linux, Slurm, InfiniBand, NVIDIA GPUs, NCCL and Cluster Operations



Subhani Roshan

Copyright

First Edition, 2026.

Disclaimer

This book is intended as educational and reference material for infrastructure engineers, system administrators, and technical professionals working with High Performance Computing (HPC) and AI/HPC infrastructure. It is not a substitute for official vendor documentation, formal training, or your organization's own operational procedures.

Commands, configuration examples, and troubleshooting procedures presented in this book should be tested in an appropriate non-production environment before being applied to production systems. Any changes to production infrastructure should follow your organization's established change-control procedures.

Hardware, firmware, driver, and software behavior can vary significantly by vendor, product generation, version, and site-specific configuration. Where this book identifies a claim as version-sensitive, configuration-dependent, or requiring verification against current documentation, readers should confirm the current behavior for their specific environment before relying on it.

The author has made a good-faith effort to ensure technical accuracy at the time of writing, drawing on official documentation and established operational practice. However, the author assumes no responsibility or liability for any loss, damage, or disruption resulting from the application of any command, configuration, or procedure described in this book without appropriate testing, validation, and adherence to the reader's own organizational policies.

How to Use This Book

This book is organized as a practical reference, not a linear textbook — you're not expected to read it start to finish before you can use it, though the chapters are sequenced so that each one builds on vocabulary the earlier ones established.

Foundations first. Part 1 (Chapters 1–2) sets up the book's core philosophy and the shape of an HPC cluster's architecture. Part 2 (Chapters 3–4) covers the Linux fundamentals and first troubleshooting methodology every later chapter assumes. If you're already comfortable with HPC basics and Linux operations, you can skim these — but Chapter 1's troubleshooting philosophy and Chapter 4's ten-section scenario format are worth reading even if the surrounding material is familiar, since every later troubleshooting chapter builds on them without re-explaining them.

Conceptual chapters vs. troubleshooting chapters. Most Parts pair a "fundamentals" or "architecture" chapter with a "troubleshooting" chapter — Linux (3/4), Storage (7/8), InfiniBand (9–12), GPU (13–15), Slurm (16–20), and NCCL (23/24) all follow this pattern. Read the conceptual chapter first if a domain is new to you; if you already know the concepts and have a live problem in front of you, you can go straight to the troubleshooting chapter — it names what it assumes and points back to where that's covered.

The troubleshooting philosophy. Every diagnostic chapter in this book applies the same underlying discipline:

OBSERVE → COLLECT EVIDENCE → ISOLATE → ROOT CAUSE → FIX → VALIDATE → PREVENT

This is the mindset. Chapter 1 introduces it in full and explains why the order matters — the single most common way engineers

waste time in production is jumping from Observe straight to Fix, applying a remembered solution before evidence supports it.

The scenario format. Every worked troubleshooting scenario documents that mindset using a consistent ten-section structure:

SYMPTOM → INITIAL CHECKS → COMMANDS → OUTPUT INTERPRETATION → POSSIBLE CAUSES → FAULT ISOLATION → ROOT CAUSE → RESOLUTION → VALIDATION → PREVENTION

Once you've read one or two of these closely (Chapter 4 has the first full set), you'll be able to scan the rest for the section you actually need rather than reading linearly.

Scenario labels mean something. Every scenario in this book is explicitly labeled **[Illustrative]** — a realistic, constructed teaching example built to demonstrate a specific troubleshooting pattern, not a claim that this exact incident happened as written. Treat each one as a pattern to recognize, not a script to follow verbatim: the value is in the reasoning path (Symptom → ... → Prevention), not in matching your own situation to it detail-for-detail.

Commands are explained, not just listed. Every command in this book is introduced with what it shows, what healthy output looks like, and what to check next — and classified as SAFE/READ-ONLY, CHANGES STATE, POTENTIALLY DISRUPTIVE, or (where a command's safety depends entirely on what it's used to run) CONTEXT-DEPENDENT. Read the classification before running anything, especially in a production environment. No command in this book should be copied and run blindly — the interpretation guidance matters as much as the syntax, and real systems vary enough in version and configuration that syntax alone isn't a safe basis for action.

Part 11's playbooks (Chapters 26–27) are where the book's domains meet — cross-layer, intermittent, and configuration-drift scenarios that don't fit neatly into one earlier chapter. Use them once you're

comfortable with the individual domains; they assume that vocabulary rather than re-teaching it.

Chapter 28 (the command cheat sheet) is a lookup tool, not a teaching chapter — every entry traces back to the chapter that actually explains it. Use it once you already know the material and just need the syntax and safety classification quickly.

Chapter 29 (interview and scenario questions) is built for two audiences: engineers preparing for an HPC infrastructure interview, and anyone who wants to test their own troubleshooting reasoning against a domain they've just studied. The questions are deliberately reasoning-based rather than fact-recall — each includes a reasoning path, not a single "correct" answer to memorize.

Who This Book Is For

This book is written for:

- HPC infrastructure engineers
- HPC system administrators
- Linux administrators moving into HPC
- Data center and infrastructure engineers working with HPC or AI/HPC systems
- AI/GPU infrastructure engineers
- Junior and mid-level HPC engineers building operational depth
- Engineers preparing for HPC infrastructure interviews

It assumes basic Linux familiarity (you can open a terminal and edit a file) but not prior HPC-specific experience. Later chapters — InfiniBand administration and troubleshooting, GPU topology, Slurm scheduling internals, NCCL diagnostics — go deep enough to be useful to more experienced engineers as well, not just newcomers to the field.

This is a practical operator's handbook, not an academic HPC textbook. The core positioning is simple: don't just learn what HPC is — learn how to operate, troubleshoot, diagnose, and maintain it.

Table of Contents

PART 1

HPC Foundations

Chapter 1, Chapter 2

CHAPTER 1

FOUNDATIONS

Introduction to HPC & the Modern AI/HPC Landscape

1.1 What HPC Actually Means to the Person Running It

Most introductions to High Performance Computing start with a definition: HPC is the use of aggregated computing resources to solve problems too large for a single machine. That definition is correct, and it is also not particularly useful if your job is to keep a cluster running.

This book is written from the other side of that definition. It assumes you are not trying to design a parallel algorithm — you are trying to figure out why a job has been pending for six hours, why a GPU disappeared from `nvidia-smi`, or why a fabric that worked yesterday is dropping links today. HPC, from an operations seat, is not a subject you study once. It's a set of systems — compute, storage, networking, scheduling — that fail in specific, recognizable ways, and an engineer's job is to recognize those ways faster than the business feels the impact.

That is the core philosophy behind this book: don't just learn what HPC is — learn how to operate it, troubleshoot it, diagnose it, and keep it running. Every chapter after this one is built around that idea. Where a concept is introduced, it's introduced because you'll need it to understand a failure mode later, not because it's interesting in isolation.

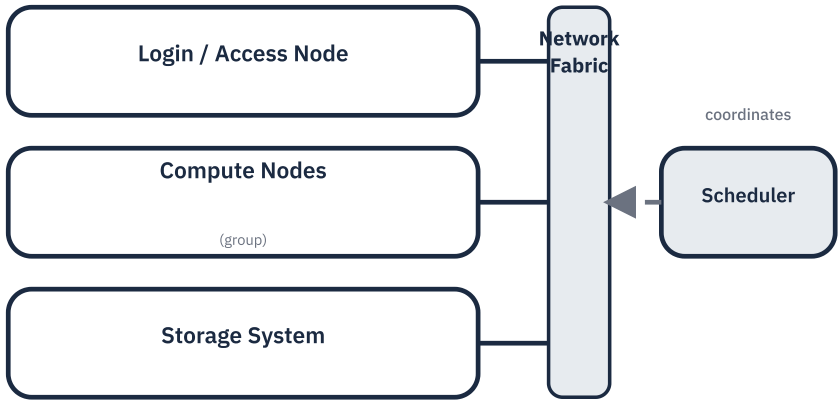


Figure 1.1 — Cluster Building Blocks (Conceptual). Simplified for orientation — see Chapter 2 for the full architecture, including management vs. data network separation. *Components: a Login/Access Node, a group of Compute Nodes, and a Storage System, all connected by a Network Fabric shown as a single bar; a Scheduler sits above the Compute Nodes, linked to them by a dashed connector labeled "coordinates" rather than a solid data-path line.*

1.2 Where HPC Shows Up in Practice

HPC clusters exist because some workloads exceed what a single machine can reasonably provide — in compute performance, memory capacity, storage throughput, or time to completion at the scale the work requires. Aggregating many machines under a shared fabric and scheduler makes it possible to apply distributed compute, memory, and I/O capacity to a single large problem, or to run many smaller jobs in parallel against shared infrastructure. In practice, the clusters you're likely to operate fall into a small number of overlapping categories:

- **Scientific simulation and modeling** — computational fluid dynamics, weather and climate modeling, molecular dynamics, and similar workloads that have driven HPC cluster design for decades.

- **Engineering and CAE (computer-aided engineering)** — structural analysis, crash simulation, and other design-validation workloads run by manufacturing and engineering organizations.
- **Research computing** — university and national-lab clusters serving many research groups with heterogeneous, often unpredictable workloads.
- **Large-scale AI/ML training and inference** — the fastest-growing category for the audience of this book. Large-scale AI/ML training shares many infrastructure characteristics with traditional HPC: distributed compute across many nodes, hardware accelerators, high-performance networking, shared storage, and coordinated resource scheduling.

This book does not assume you work in any one of these domains. It assumes you operate the infrastructure underneath them — and that infrastructure looks remarkably similar regardless of what workload is running on top of it. An InfiniBand link-flap, a NUMA misbinding, or a Slurm job stuck in `PENDING` looks the same whether the job is a climate model or a language-model training run. That similarity is what makes an infrastructure-focused book possible in the first place.

1.3 Cluster vs. Supercomputer vs. Cloud HPC

The terms "cluster," "supercomputer," and "cloud HPC" get used loosely, often interchangeably, in casual conversation. For the purposes of this book, it's worth drawing a practical (not formal) line between them — not because the industry has a single agreed taxonomy, but because the distinction changes what you, as the engineer, actually control.

Table 1.1 — A Practical Comparison

Dimension	On-Premises Cluster	Leadership-Class / Large-Scale Supercomputer	Cloud HPC
Ownership	Owned/operated by your organization	Owned/operated by a national lab or research consortium	Rented from a cloud provider
Elasticity	Fixed capacity; growth requires procurement	Fixed capacity; among the largest single systems in existence	Elastic; capacity requested and released on demand
Typical scheduler	Slurm, PBS, LSF, or other schedulers	Slurm or a site-specific scheduler	Slurm-on-cloud, or a cloud-native/Kubernetes-based orchestrator
Typical workload pattern	Mixed, multi-tenant, continuous	Large, allocation-based, scheduled in advance	Bursty; scaled to match a specific job or training run
Typical engineer's concern	Day-to-day operations, hardware lifecycle, fabric health	Extreme scale, allocation policy, system-level reliability	Cost management, provisioning automation, ephemeral infrastructure

This is a working distinction for this book, not a formal industry standard — the boundaries blur in practice (a cloud provider's GPU cluster is, physically, still a cluster with InfiniBand and Slurm-like

scheduling underneath it). The infrastructure concepts and troubleshooting principles in this book transfer across all three columns, but how you apply them — including which specific commands and diagnostics are available to you — differs: your physical or administrative access, ownership boundaries, and remediation paths all vary between operating your own hardware, working within a cloud provider's support boundary, and following a leadership-class facility's allocation and change-control process. This book focuses on the operational concepts that hold across all three, and flags where access- or ownership-driven differences matter rather than assuming them away.

1.4 The Modern AI/HPC Ecosystem — Where This Book's Vocabulary Fits

If you're arriving at HPC from cloud infrastructure, DevOps, or machine learning rather than from a traditional HPC background, a few terms will come up early and often in this book that are worth placing in context now — not explaining fully, just placing, so later chapters don't have to stop and re-orient you.

Slurm and Kubernetes. Both are used to run large-scale AI training workloads today, and both are covered — or referenced — in this book, but they are not interchangeable, and this book takes no position on which you should use. Slurm is a batch scheduler built for HPC: it excels at tightly-coupled, multi-node jobs with well-defined resource requests, and it is the dominant scheduler in traditional HPC environments. Kubernetes is a container orchestration platform built for services and, increasingly, adapted for ML training workloads through additional tooling. Which one you encounter depends entirely on the environment you're operating — this book's scheduling chapters (Part 7) are Slurm-specific because Slurm is what a reader in a traditional HPC or GPU-cluster operations role will most often be responsible for.

MPI and NCCL. These solve related but distinct problems, and it's worth being precise about the difference early, because getting it wrong is a common source of confusion. MPI — the Message Passing Interface — is a long-standing, general-purpose standard for message passing in parallel and distributed computing, implemented by libraries such as Open MPI and MPICH, and used across HPC applications generally, not just AI workloads. NCCL — NVIDIA's Collective Communications Library — is a GPU-specific library that implements collective communication operations (operations like AllReduce and Broadcast, covered in Part 9) for the way GPUs are actually connected to each other, whether by NVLink, PCIe, or InfiniBand. NCCL's own documentation describes it as closely following the collective communication API that MPI popularized, specifically so that engineers already familiar with MPI find NCCL's interface natural — but NCCL is not "MPI for GPUs," and it does not replace MPI. Distributed GPU training software isn't built one single way: some stacks use MPI for process launch and general message passing, some use NCCL for GPU-to-GPU collective communication, and some use both together alongside other process-management components, depending on the framework and deployment. This book covers NCCL in Part 9 because it's the library most directly relevant to the GPU-to-GPU and GPU-to-NIC communication covered in Part 6 — not because it's the only way distributed training is built. It does not attempt to be an MPI reference.

On-prem vs. cloud HPC. As established in §1.3, this book does not treat these as fundamentally different subjects — the troubleshooting principles it teaches (the philosophy in §1.7, the fault-isolation logic, the command patterns) transfer between them. What can differ is what's available to you when you apply those principles: on hardware you operate directly, you typically have full diagnostic access and can reseat a card or swap a cable yourself; in a cloud environment, some lower-level diagnostics may be restricted, and remediation runs through the provider's support and access boundaries instead. Where cloud-specific

concerns exist — provisioning automation, cost, ephemeral infrastructure — they are outside this book's scope, noted explicitly in §1.5.

1.5 What This Book Covers — and What It Deliberately Doesn't

This book is a practical operations handbook for engineers responsible for HPC and AI/HPC infrastructure — Linux, storage, InfiniBand/RDMA, NVIDIA GPUs, Slurm, and NCCL, with an emphasis throughout on troubleshooting methodology over definitions. It is written for readers with roughly one to five years of infrastructure experience, though later chapters (InfiniBand administration, GPU topology, Slurm scheduling internals, NCCL diagnostics) go deep enough to be useful to more experienced engineers as well.

Table 1.2 — Scope

Topic Area	In Scope	Mentioned for Context Only	Out of Scope
Linux administration & troubleshooting	Yes — Part 2	—	General Linux certification-level material
HPC storage (NFS, Lustre, GPFS)	Yes — Part 4	—	Storage vendor product comparisons
InfiniBand, RDMA, RoCE	Yes — Part 5	—	Low-level RDMA application programming

NVIDIA GPU infrastructure & NVLink/NVSwitch	Yes — Part 6	—	CUDA application programming
Slurm administration, scheduling, and troubleshooting	Yes — Part 7	—	PBS, LSF, and other schedulers
Fabric monitoring, NVIDIA UFM	Yes — Part 8	—	General-purpose monitoring tool tutorials (Prometheus/Grafana, etc.)
NCCL fundamentals & troubleshooting	Yes — Part 9	—	MPI as a standalone subject; distributed training algorithms
Data center operations & hardware lifecycle	Yes — Part 10	—	Electrical/mechanical facilities engineering
Kubernetes for AI training	—	Yes — §1.4 only	Kubernetes administration
Cloud-specific provisioning & cost management	—	Yes — §1.3, §1.4	Cloud platform administration

If a topic isn't in one of these Parts, it isn't in this book — and that's intentional. A book that tries to cover everything adjacent to HPC ends up covering nothing in the depth an operator actually needs.

1.6 How This Book Is Organized

The twelve Parts of this book follow a deliberate order: foundations, then Linux, then hardware (CPU/NUMA), then storage, then the InfiniBand fabric, then GPUs, then Slurm (which depends on understanding GPUs first), then monitoring, then NCCL (which depends on both InfiniBand and GPU topology), then data center and hardware lifecycle operations, then a dedicated troubleshooting playbook, then reference material. Each Part builds on vocabulary the previous ones established — you shouldn't need to skip ahead to understand a later chapter.

Two formatting conventions appear throughout the book, starting in Chapter 4:

- **Troubleshooting scenarios** are written in a consistent structure — SYMPTOM, INITIAL CHECKS, COMMANDS, OUTPUT INTERPRETATION, POSSIBLE CAUSES, FAULT ISOLATION, ROOT CAUSE, RESOLUTION, VALIDATION, PREVENTION — so you can scan for the section you need rather than reading linearly. Section 1.7 explains the thinking this structure is built on.
- **Every command** is marked as either read-only (safe to run without changing system state) or state-changing (alters configuration, cancels a job, restarts a service, and so on). The marker appears the first time a command is introduced and is repeated in the Chapter 28 command cheat sheet.

KEY TAKEAWAY — HOW SCENARIO BOXES WORK

From Chapter 4 onward, troubleshooting content is presented in a labeled box with ten fixed sections, in order: what you'd observe (SYMPTOM), what to check first (INITIAL CHECKS), the exact commands to run (COMMANDS), how to read their output (OUTPUT INTERPRETATION), what could be causing it (POSSIBLE CAUSES), how to narrow it down (FAULT ISOLATION), what's actually wrong (ROOT CAUSE), how to fix it (RESOLUTION), how to confirm the fix worked (VALIDATION), and how to stop it recurring (PREVENTION). The first worked example appears in Chapter 4.

1.7 The HPC Troubleshooting Philosophy

Every troubleshooting scenario in this book is written using the ten-section format previewed above. That format is a way of *documenting* a diagnosis. It isn't, by itself, a way of *thinking through* one — and the difference matters, because the most common way engineers waste time in production isn't a lack of commands, it's skipping straight from noticing a problem to trying a remembered fix, without doing the steps in between.

The mental model underneath every scenario in this book is seven stages:

- **Observe** — Notice the symptom as it's actually reported or detected. Not "the GPU is broken" — "the job failed with this specific error, at this specific time."
- **Collect Evidence** — Gather command output, logs, and metrics *before* forming a theory. This is where most of the commands in this book get used.

- **Isolate** — Narrow the problem to a specific layer or component. Is it the node? The fabric? The scheduler? The application? Isolation is what turns "something is wrong" into "this specific thing is wrong."
- **Root Cause** — Identify the actual underlying failure — not the first plausible explanation, and not the symptom itself.
- **Fix** — Apply a correction targeted at the root cause. A fix aimed at a symptom tends to resurface later as the same ticket with a different timestamp.
- **Validate** — Confirm the fix actually resolved the issue under real conditions — not just that the error message stopped appearing.
- **Prevent** — Decide what stops this from happening again: a monitoring check, a configuration change, a process change.

These seven stages map directly onto the ten-section scenario format used from Chapter 4 forward:

Table 1.3 — Philosophy to Format Mapping

Philosophy Stage	Corresponding Scenario Section(s)
Observe	SYMPTOM
Collect Evidence	INITIAL CHECKS, COMMANDS, OUTPUT INTERPRETATION
Isolate	POSSIBLE CAUSES, FAULT ISOLATION
Root Cause	ROOT CAUSE
Fix	RESOLUTION
Validate	VALIDATION
Prevent	PREVENTION

The order matters more than it looks like it should. The single most common failure mode this book is written to correct is jumping from Observe directly to Fix — trying something that worked on a similar-sounding problem before, without evidence that it applies here. That approach sometimes works, which is exactly why it's dangerous: it doesn't fail loudly, it fails by leaving the real cause in place, ready to resurface under slightly different conditions. Every scenario chapter in this book is written to make the isolation and root-cause steps explicit, not skippable.

Chapter 1 has given you the mindset. Chapter 4 is where it's first applied to a real symptom, with real commands and real output to interpret.

CHAPTER 2

FOUNDATIONS

HPC Cluster Architecture

2.1 From the Big Picture to the Components

Chapter 1 sketched a cluster as five blocks — a login node, a group of compute nodes, a storage system, a network fabric connecting them, and a scheduler coordinating it all (Figure 1.1) — deliberately coarse, meant only to establish that these pieces exist. This chapter names each of those pieces properly, adds two the earlier sketch left out for simplicity (GPU nodes and the management/control layer), and gives you enough of a map to recognize any of them on a real cluster. Nothing here is administration or troubleshooting — the detailed operational and troubleshooting material begins in the chapters that follow. What follows is the shape of a cluster, not yet how to run one.

2.2 Node Roles

An HPC cluster is organized around a small number of node roles. Understanding what each is *for* — not yet how each is configured — is the foundation for everything else in this book.

Login (access) nodes are where users land. This is where you edit files, prepare a job submission, and hand work off to the scheduler. A login node is not where computation happens; if a job's actual work is running on the login node, something has gone wrong with the workflow, not the architecture.

Compute nodes are where jobs actually run. They accept work assigned by the scheduler, execute it, and report back. This is typically where most of a cluster's raw capacity resides, simply because compute is the role every other role exists to support.

GPU nodes are compute nodes built around one or more accelerators, rather than — or in addition to — general-purpose CPU cores alone. They deserve their own category instead of being folded into "compute nodes" because the accelerators change more than just the workload: they change the node's internal topology (how CPUs, GPUs, and the network interface connect to each other), its power draw, and its cooling requirements. A GPU node might, for example, be built around a system like an NVIDIA DGX-class server — mentioned here only to make the category concrete, not as a recommendation. This book returns to GPU node topology in depth in Part 6; for now, the point to fix in mind is that GPU nodes are a distinct role, not a compute node with an accessory attached.

Management (or head/control) nodes run the software that keeps the cluster operating: the scheduler's controller process, provisioning services, monitoring collection, and similar control-plane functions. A management node's job is to coordinate the cluster, not to do the cluster's work — it typically doesn't run user jobs at all.

Storage, in this context, is less a single node than a system — one or more servers presenting shared storage to every other node role, so that a job can start on any compute or GPU node and see the same files. Part 4 covers what that system is actually built from.

Table 2.1 — Node Role Responsibility

Node Role	Primary Responsibility	Talks Directly To	Full Treatment In

Login/Access	User entry point; job preparation and submission	Management node (scheduler), shared storage	This chapter only
Compute	Executes general-purpose job workloads	Management node (scheduler), fabric, storage	Part 7 (scheduling), Parts 2–3 (Linux/hardware)
GPU	Executes accelerator-bound workloads	Management node (scheduler), fabric, storage, other GPU nodes	Part 6
Management/Control	Hosts or supports scheduler, provisioning, monitoring, and other cluster control services	Every other node role	Part 7 (scheduler), Part 8 (monitoring)
Storage	Provides shared, consistent storage to all other roles	Compute, GPU, and login nodes, via the fabric	Part 4

On a small cluster, these roles don't always map to five separate physical machines — a small deployment might, for instance, combine login and management functions on one machine. The roles themselves don't change; what changes is how many physical boxes they're spread across. Later chapters describe each role assuming it's dedicated, since that's the pattern on any cluster large enough to need a book like this one — but recognizing a collapsed role on a smaller system is worth being able to do.

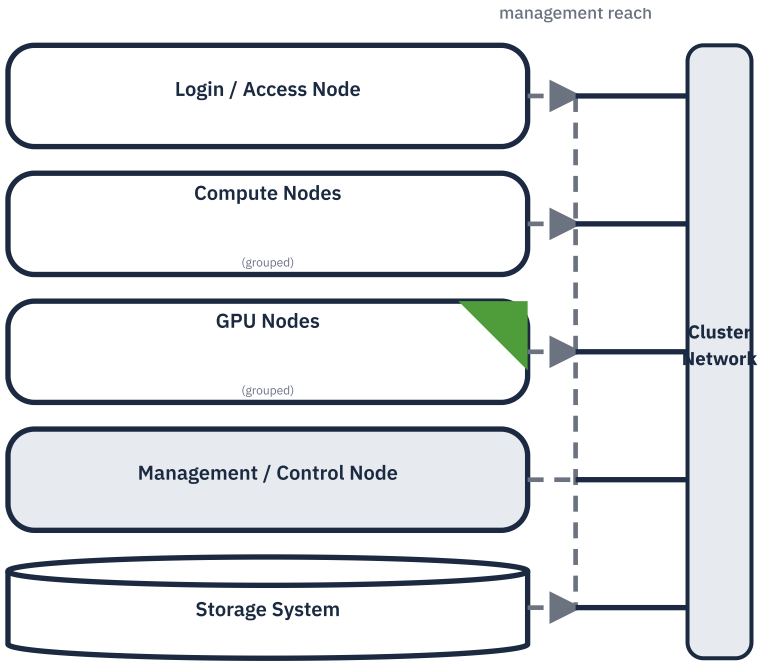


Figure 2.1 — Cluster Architecture (Node Roles). Every node role named in this chapter, shown together for the first time. The network shown here as one bar is split into management and data/fabric networks in Figure 2.2. *Components: Login/Access Node · Compute Nodes (grouped) · GPU Nodes (grouped, shown as distinct from Compute Nodes) · Management/Control Node · Storage System. Connections: all node types connect to a single labeled "Cluster Network" bar, plus a dashed Management/Control → all-nodes connector representing provisioning/monitoring reach.*

2.3 The Scheduler's Place in the Architecture

The scheduler doesn't fit neatly into the node-role list above because it isn't a node role — it's a control-plane service that happens to run on a management node. Its job is to decide which job runs on which compute or GPU node, and when, based on what's requested and what's available.

This matters architecturally for one reason: the scheduler coordinates the compute/GPU nodes without being part of the path data actually flows through. A job's input and output normally travel between compute/GPU nodes and storage over the appropriate data network or fabric; the scheduler is not normally in that workload data path. It decides where the job runs and tracks its state, then gets out of the way. Figure 1.1 showed this with a dashed connector for exactly this reason: the relationship is one of coordination, not data flow.

Part 7 covers Slurm's architecture, commands, and scheduling policy in depth. At this stage, the point worth fixing in mind is simply where the scheduler sits relative to everything else: logically above the compute/GPU nodes, often hosted on a management/control node, and outside the data path entirely.

2.4 Management Network vs. Data/Fabric Network

Every node role above needs to communicate with the others, but not all of that communication has the same character. Provisioning a new node, collecting monitoring data, and reaching a node for administrative purposes are all comparatively low-bandwidth, latency-tolerant operations. A running job reading its input from

shared storage, or a multi-node job exchanging data between compute or GPU nodes, is neither — it needs bandwidth and low latency, and it needs them consistently.

Because of that difference, clusters commonly separate these into two networks: a **management network**, carrying provisioning, monitoring, and control traffic, and a **data (or fabric) network**, carrying job I/O and storage traffic. This separation is a design principle, not a specific product — some sites build it as genuinely separate physical networks, and the details of how it's implemented vary by site. What holds regardless of implementation is the underlying idea: control traffic and job traffic have different requirements, and keeping them apart prevents one from starving the other.

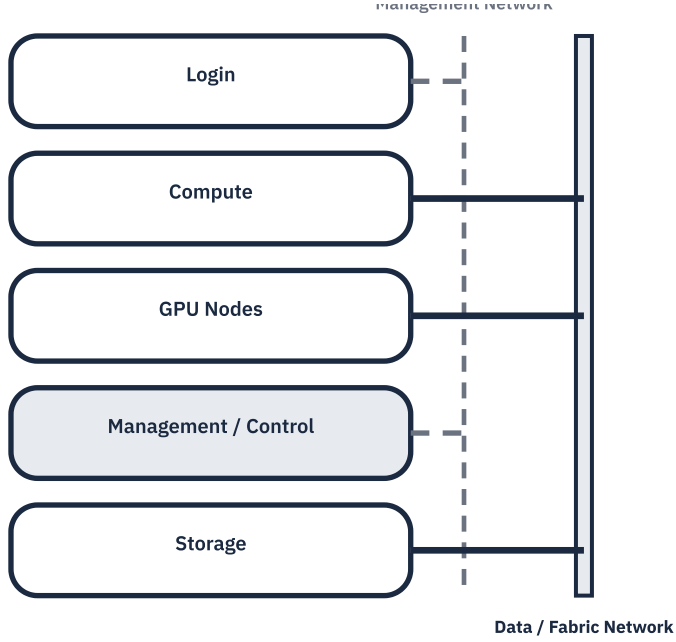


Figure 2.2 — Management Network vs. Data/Fabric Network Segmentation. Two logically — and usually physically — separate networks carry different traffic. Section 2.5 covers what the data/fabric network is built from; Part 5 covers it in depth. *Same node set as Figure 2.1, now with two distinct connecting planes: a "Management Network" (thinner/dashed, connecting the Management/Control Node*

to every other node) and a "Data/Fabric Network" (thicker/solid, connecting Compute, GPU, and Storage to each other — not the Login node in the same way). Labels: "Management Network (provisioning, monitoring, control)" and "Data/Fabric Network (job I/O, storage traffic).".

One practical consequence worth internalizing now: if a node is reachable on the management network but a job on it can't reach storage, that's not a contradiction — it's exactly what the separation predicts. Section 2.9 returns to this as this chapter's one illustrative example.

Part 5 covers what actually rides on the data/fabric network — InfiniBand, RDMA, RoCE — in depth. This chapter stops at recognizing that the split exists and why.

2.5 The Cluster Interconnect

The data/fabric network described above is carried by physical hardware: a network interface in each node, cables, and switches connecting them into a fabric. This physical and logical structure — collectively, the cluster interconnect — is what compute, GPU, and storage nodes actually use to reach each other.

Two things are worth fixing in mind at this level, before Part 5 goes deep on the technology itself. First, the interconnect is a shared resource: many nodes' traffic crosses the same switches and links, so how it's built — its topology — affects how well it holds up under load from many nodes at once, a concern Part 5 returns to directly. Second, InfiniBand is a common choice for the data/fabric network in traditional HPC environments, though Ethernet-based fabrics — including RoCE, covered in Part 5 — are also widely used; this book doesn't assume one over the other going in, and neither should you.

This chapter doesn't cover switch topology, cabling, or fabric administration — that's Part 5's job, in detail. What matters here is only that "the fabric" is physical hardware with real capacity limits,

not an abstraction that scales for free.

2.6 Storage's Place in the Architecture

Section 2.2 introduced storage as a system rather than a single server; this section says a little more about where that system sits.

Shared storage connects to the rest of the cluster through an appropriate storage/data network, which may be the same fabric used for workload communication or may be separately designed — which means storage performance and interconnect performance can be linked. A job's read or write to shared storage consumes network and storage resources along the path between the compute/GPU node and the storage system.

Many HPC storage architectures distinguish between metadata operations and file-data operations, although the way those functions are implemented varies considerably between storage systems. That separation exists so that operations that only need metadata — listing a directory, checking a file's size — don't have to compete with large data transfers for the same resource. NFS, Lustre, and GPFS/IBM Storage Scale — the systems Part 4 covers in depth — implement this idea differently, but the underlying shape is similar enough to introduce once, here.

For now: storage is reachable from every compute and GPU node over the storage/data network, and it's typically built as a system with more than one kind of internal component. Part 4 is where the differences between specific storage architectures, and how to operate them, actually matter.

2.7 Provisioning and the Software Environment

Every node role described so far assumes the node is already a working member of the cluster — running the right operating system, with the right services active, reachable on the right

networks. Getting a node from newly-racked hardware to that state is provisioning, and keeping many users productive on shared nodes afterward is what environment modules solve. They're different problems, and it's worth keeping them separate in your head.

Provisioning is the process of turning bare hardware into a cluster member: deploying an operating system image, applying the cluster's standard configuration, and joining it to the management network so it can be monitored and scheduled against. Two broad conceptual approaches show up across sites: network boot and image deployment (a node boots over the network — commonly using mechanisms such as PXE — and receives an operating-system image or stateless runtime environment prepared for it), and configuration-management-driven provisioning (a node starts from a more generic base, and a tool applies the cluster's specific configuration afterward). Tools associated with these approaches — stateless-provisioning systems such as Warewulf, or general configuration-management tools — exist and change over time; this book names the concepts rather than endorsing a specific tool, since the right choice is a site decision, not a universal one.

Environment modules solve a different problem: once a node is working, different users and different jobs often need different, sometimes conflicting, versions of compilers, libraries, or applications on the same shared nodes. Rather than installing one fixed software stack, a cluster installs many, and lets each user select what they need for a given job through a module system — commonly Lmod, sometimes paired with Spack for building the software itself. The mechanism for doing this — `module avail`, `module load`, and similar commands — is covered in Chapter 3; this chapter only needs you to recognize that the mechanism exists and why: it's what makes a shared, multi-tenant cluster usable by people who need different, sometimes incompatible, software at the same time.

Both provisioning and environment modules are part of "the software environment" in a loose sense, but they operate at different points in a node's life: provisioning happens once, or is repeated when a node is rebuilt, while environment modules are used continuously by every job that runs afterward.

2.8 Facility Relationship: Power, Cooling, and Physical Placement

Everything described so far has a physical cost. Nodes draw power and generate heat, and both scale with what's installed in them — a GPU node with several accelerators can have substantially higher power and cooling requirements than a general-purpose compute node occupying similar rack space. That difference has real architectural consequences: how many nodes fit in a rack, how power is distributed to them, and how the room cools them are decisions that follow directly from what kind of nodes are being deployed, not decisions made independently of cluster design.

This book treats that connection as something an infrastructure engineer needs to be aware of at this stage, not something this chapter teaches you to design. Rack layout, power distribution, cabling discipline, and the hardware lifecycle that follows — burn-in, validation, node recovery, RMA — are covered in Part 10. What matters here is the link itself: the architecture decisions described in this chapter, such as how many GPU nodes and how densely they're packed, are not free of facility consequences, and treating them as if they were is a common source of surprises later.

2.9 How the Pieces Fit Together

Put together, a cluster looks like this: users reach it through login nodes; the scheduler, running on a management node, assigns their jobs to compute or GPU nodes; those nodes execute the work, reading and writing shared storage over the data/fabric network,

while the management network handles provisioning and monitoring in the background; and all of it runs on physical hardware with real power and cooling limits. Figure 2.1 draws this map explicitly, with every role from Section 2.2 shown together for the first time.

[Illustrative] Suppose a new compute node is added to the cluster with the wrong software image. If the site's provisioning and node-validation process includes management-plane checks, the mismatch can be detected before the node is placed into service. The node can then be held out of scheduling until the software baseline is corrected, rather than allowing the mismatch to surface later as a confusing in-job failure that looks like an application problem. The architecture doesn't just organize the cluster — it shapes how failures are caught, and that idea recurs throughout the rest of this book.

Chapter 3 picks up where this chapter's software-environment section left off, with Linux fundamentals for the nodes this chapter has just mapped out.

PART 2

Linux for HPC

Chapter 3, Chapter 4

CHAPTER 3

LINUX

Linux Fundamentals for HPC Engineers

3.1 Where This Fits

Chapter 2 mapped the shape of a cluster — the node roles, the networks, the fabric. This chapter is about the Linux underneath any one of those nodes: what's on it, who can touch what, and how to look around safely before you change anything. Nothing here is performance troubleshooting — that starts in Chapter 4.

3.2 Filesystem Layout and Navigation

An HPC node's filesystem follows the same general layout as any Linux system, but a few directories matter more here than they do on a laptop. `/etc` holds system and service configuration. `/var/log` holds the logs Section 3.5 covers. `/proc` and `/sys` are virtual filesystems exposing kernel and hardware state — you'll read from them constantly (often indirectly, through commands that parse them for you) but rarely write to them directly as a matter of routine operations.

Beyond the standard system directories, HPC nodes typically expose one or more shared paths — often something like a home directory and a scratch or shared work area — mounted from the storage system described conceptually in Chapter 2 and in depth in

Part 4. This chapter doesn't cover how those mounts are designed; it only assumes you can recognize one when `df` or `mount` shows it to you.

`ls`, `pwd`, and `cd` are how you move around and see what's there — worth naming explicitly only because everything else in this chapter assumes you're already comfortable with them.

3.3 Processes, Users, and Permissions

Every running program on a node is a process, identified by a process ID (PID) and owned by a user. `ps` lists them; Chapter 4 covers using `ps` to diagnose a problem, but the concept — a process has an owner, and that ownership determines what it can touch — belongs here. Two small commands are worth knowing alongside this: `whoami` prints the username you're currently operating as, and `id` prints that same user's UID, GID, and group memberships in full. On a shared HPC node, confirming which identity you're actually acting under — before assuming a permission error is a bug rather than a group-membership gap — is a quick, first-step habit worth having.

HPC nodes are shared. Multiple users' jobs run on the same compute nodes, often concurrently, which makes the Linux permission model more operationally significant than it is on a single-user machine. Every file and directory has an owner (a user) and a group, plus permission bits controlling read, write, and execute access for the owner, the group, and everyone else. `chmod` changes those bits; `chown` changes ownership. Both change system state and are worth pausing before running, especially with a recursive flag, on a shared filesystem where other users' files might be affected by a mistake.

`sudo` grants a user temporary elevated privileges for specific commands, configured by an administrator rather than granted freely. On a shared HPC node, reaching for `sudo` to work around a

permission error is usually the wrong instinct — permission errors on shared storage are common precisely because the system is multi-tenant by design, and the more common causes are group membership or a misunderstanding of which directory you're actually writing to, not a missing privilege that `sudo` is meant to solve.

3.4 Services and systemd

Long-running background functionality on a Linux node — networking, logging, the services that make the node a working cluster member — is managed by systemd, and `systemctl` is how you interact with it. `systemctl status <service>` is read-only and tells you whether a service is active, when it last started, and its recent log lines. `systemctl start`, `stop`, and `restart` change state; `enable`/`disable` change whether a service starts automatically at boot, which is a persistent configuration change rather than a one-time action.

HPC nodes run services specific to their role — a scheduler-related daemon on nodes that need to accept jobs, an authentication service supporting multi-node trust, among others. This chapter doesn't configure or administer any of them by name; that's Part 7's job for the scheduler specifically. What matters here is the general skill: knowing how to check whether a service is running and reading what it reports, before assuming a problem is bigger than "a service isn't up."

Restarting a service is not automatically safe. On a node currently running jobs, restarting the wrong service — or even the right one, at the wrong time — can disrupt work that has nothing to do with why you're looking at it. Confirm what a service is actually responsible for before restarting it on a production node.

3.5 Logs

Logs are usually the first evidence you collect when something looks wrong — a direct application of Chapter 1's "Collect Evidence" stage, which Chapter 4 puts into full practice. `journalctl` reads the systemd journal, and is the modern, read-only way to see a service's recent output and system-level events together. Traditional log files under `/var/log` still exist for some services and are worth knowing how to find, even where `journalctl` covers most day-to-day needs.

Reading logs is a habit worth building before you need it under pressure: check what a service or the system itself has already recorded before running new diagnostic commands, since the evidence you need is often already sitting there.

3.6 Basic Networking on a Node

This section is about confirming a single node's own network state, not the fabric it's connected to — that's Part 5's territory entirely. `ip a` (or `ip addr`) shows a node's network interfaces and their addresses; `ip link` shows interface state without address detail. `ss` shows active sockets and listening ports, useful for confirming whether a service is actually listening where it's supposed to be.

All three are read-only. None of them tell you anything about fabric health, link state, or InfiniBand-specific concepts — a node's Ethernet management interface being up says nothing about the data/fabric network Chapter 2 described, and conflating the two is a common early mistake.

3.7 Disks and Filesystems at an Operator Level

`lsblk` lists block devices — local disks and their partitions — read-only. `df` shows how full each mounted filesystem is, also read-only, and is the first command to reach for when a "disk full" symptom shows up (Chapter 8 covers that symptom in depth for shared storage specifically). `mount`, run without arguments, lists currently mounted filesystems read-only; run with arguments to actually mount something, it changes system state.

At this level, the goal is recognition: can you tell a local disk from a shared mount, and can you tell whether something is mounted at all. The architecture behind a shared mount — what's actually serving it — is Part 4's subject.

3.8 SSH and Multi-Node Access

HPC clusters have many nodes, and a person or a script often needs to reach several of them without typing a password each time. Key-based SSH authentication — a public/private key pair, with the public key placed on the systems you need to reach — is the common mechanism for this. `ssh-keygen` generates a key pair; `ssh-copy-id` places your public key in the remote system's `authorized_keys` file (a state-changing action on the remote side). Once set up, `ssh` itself is how you reach any node you have access to.

This matters more in HPC than on a single workstation because the pattern repeats across every node in the cluster, and because scheduler-launched work often depends on the same non-interactive access working consistently everywhere.

[Illustrative] A user can reach the login node over SSH without any trouble, but SSH to a specific compute node they've just been assigned fails. Before assuming the compute node itself is broken, the more common explanation is worth checking first: has this user's key actually been propagated to that node, and is the node reachable on the network path SSH uses at all? This chapter doesn't work the scenario in full — Chapter 4 introduces the format for that — but the instinct to check propagation and reachability before assuming node failure is worth having early.

3.9 Package Management

HPC nodes in this book's context are RHEL-family systems, and `dnf` (or the older `yum`, still common on some deployed systems) is the package manager. `dnf list installed` and `dnf info <package>` are read-only ways to check what's on a node and which version. `dnf install` changes state and, on a production compute node, is usually something done through the provisioning and configuration-management process described conceptually in Chapter 2 — not as an ad hoc action on one running node — precisely because an ad hoc install on a single node breaks the fleet-wide consistency every later chapter's assumptions depend on.

3.10 Shell Fundamentals, Environment Variables, and Bash Basics

Your shell session has an environment: a set of variables like `PATH` (where the shell looks for commands) that shape how commands behave. `env` lists them read-only; `export VAR=value` sets one, changing your current session's state (not the system's). Recognizing environment variables matters beyond convenience —

HPC job scripts (covered in Part 7) rely heavily on environment variables to pass configuration into a job, and a scheduler-launched job's environment isn't always identical to your interactive shell's.

Bash is widely used in HPC environments, although the default shell depends on the operating system and site configuration. Bash itself supports variables, loops, and conditionals for scripting. This chapter doesn't teach bash scripting; it assumes enough recognition to read a short job script later without being confused by `$VARIABLE` syntax or a simple `for` loop, which is as far as this book goes on the subject.

3.11 Environment Modules in Practice

Chapter 2 introduced environment modules conceptually: a mechanism letting different users select different, sometimes conflicting, software versions on the same shared nodes. In practice, that mechanism is a small set of commands — commonly provided by Lmod. `module avail` lists what's offered (read-only); `module list` shows what's currently loaded in your session (read-only); `module load <name>` and `module unload <name>` change your session's environment (state-changing, but only for your session, not the system).

This is the one place in the chapter where "changes state" means something narrower than elsewhere — loading a module changes your shell's environment variables, nothing more permanent.

3.12 Containers on HPC Nodes, Briefly

Multi-tenant HPC nodes generally don't run Docker the way a single-user development machine might, because Docker's daemon-based model typically requires elevated privileges that don't fit a shared, many-users environment well. Apptainer (formerly Singularity) is the container runtime more commonly

seen on HPC clusters instead, largely because it's designed to run without a persistent privileged daemon and without granting a user root access to do it. The practical takeaway at this level is narrow: if you see a `.sif` file or an `apptainer run` command in someone's job script, that's what it is — a portable, self-contained software environment, running under the invoking user's own permissions rather than an elevated one.

This book doesn't teach building or authoring container images; the goal here is recognizing containers as one more layer in the software stack Figure 3.1 shows, alongside packages and modules.

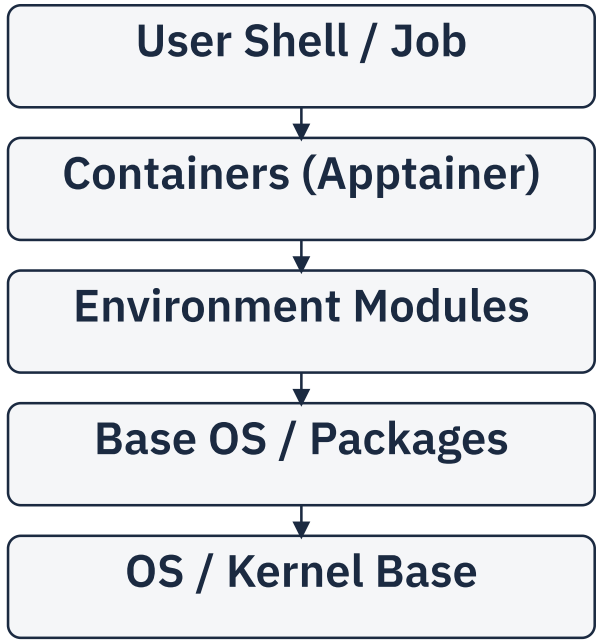


Figure 3.1 — The Software Stack on an HPC Node. Software on an HPC node is layered: a common base, selectable module environments on top, and optionally a container runtime for full environment portability. *Components: OS/Kernel Base → Package-Managed System Software → Environment Modules Layer → Container Runtime Layer (optional) → User Shell/Job, shown as a bottom-up layered stack. Labels: "Base OS/Packages," "Environment Modules," "Containers (Apptainer)," "User Shell/Job.".*

3.13 Command Safety at a Glance

Table 3.1 — Command Safety Classification

Category	Commands
● SAFE / READ-ONLY	`ls`, `pwd`, `cd`, `ps`, `whoami`, `id`, `systemctl status`, `journalctl`, `ip a`, `ip link`, `ss`, `lsblk`, `df`, `mount` (no arguments), `dnf list installed`, `dnf info`, `env`, `module avail`, `module list`, `ssh` (establishing/inspecting a session)
■ CHANGES STATE	`chmod`, `chown`, `systemctl start/stop/enable/disable`, `mount <device> <path>`, `export`, `module load` / `unload`, `ssh-copy-id`, `ssh-keygen`
▲ POTENTIALLY DISRUPTIVE	`systemctl restart` on a service other jobs or users depend on, recursive `chmod` / `chown` on shared directories, `dnf install` on a production node outside change control
◆ CONTEXT-DEPENDENT	`sudo`, `apptainer run`

`sudo` is not itself read-only or state-changing — it's a privilege-escalation mechanism, and its effect is entirely determined by the command run through it. `ssh`'s SAFE/READ-ONLY classification above covers only establishing a session and inspecting a remote system; commands executed once connected carry their own classification, not `ssh`'s. `ssh-keygen` changes state because it creates or can overwrite local key files. `apptainer run`'s effect

depends entirely on the container image and how it's invoked — it should not be assumed read-only merely because it's "just running a container."

Table 3.2 — FHS-Relevant Directories, Quick Reference

Path	Typical Contents
`/etc`	System and service configuration
`/var/log`	Traditional log files
`/proc`	Virtual filesystem exposing process and kernel state
`/sys`	Virtual filesystem exposing kernel/hardware state
`/home` (or equivalent shared path)	User home directories, typically on shared storage

A node's shared work/scratch path varies by site and is introduced conceptually in Chapter 2; Part 4 covers what serves it.

CHAPTER 4

LINUX

Linux Performance Troubleshooting

4.1 The Triage Method

Chapter 1 introduced a seven-stage way of thinking about any problem: Observe, Collect Evidence, Isolate, Root Cause, Fix, Validate, Prevent. This chapter is where that mindset meets its first real commands. The practical question every Linux-level performance symptom starts with is simple to ask and easy to answer badly if you skip evidence: is this CPU, memory, storage I/O, network, or a specific process — or is it not actually a node problem at all?

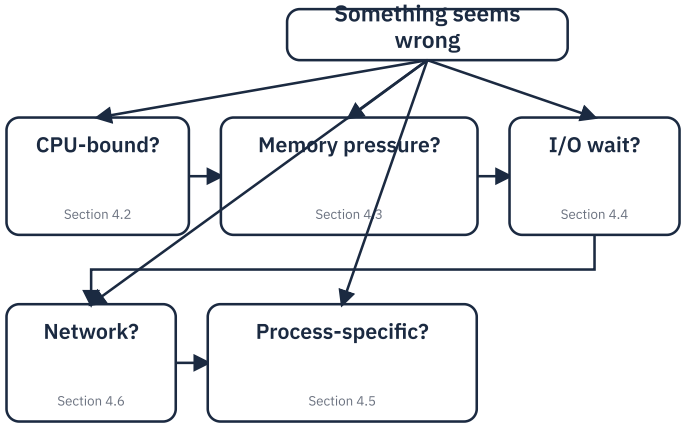


Figure 4.1 — Linux Triage Decision Flow. Start here. Each branch points to the section with the commands and interpretation for that category. *Components: a*

starting point ("something seems wrong") branching to five checks — CPU-bound? / Memory pressure? / I/O wait? / Network? / Process-specific? — each branch pointing to the section that covers it (4.2–4.6).

Every scenario later in this chapter, and every troubleshooting chapter from here on, follows the ten-section write-up format previewed in Chapter 1: SYMPTOM, INITIAL CHECKS, COMMANDS, OUTPUT INTERPRETATION, POSSIBLE CAUSES, FAULT ISOLATION, ROOT CAUSE, RESOLUTION, VALIDATION, PREVENTION. This chapter is the first place that format is actually used, and it's worth reading the first scenario slowly for the pattern before skimming the rest.

4.2 CPU Utilization and Load Average

`top` and `htop` (read-only) show per-process CPU usage and a running summary, updated continuously. `ps` (read-only), typically with options to show all processes and their resource usage, gives a point-in-time snapshot instead — useful in scripts or when you want a single capture rather than a live view.

Load average — the three numbers most of these tools show alongside CPU stats — represents the average number of processes wanting to run (or waiting on I/O, depending on the system) over the last one, five, and fifteen minutes. A load average higher than the number of CPU cores doesn't automatically mean the CPU itself is the bottleneck: on Linux, processes waiting on disk I/O count toward load average too, which is why load average and CPU utilization need to be read together, not interchangeably. A node showing high load but low actual CPU utilization in `top` is a strong hint to look at I/O next (Section 4.4), not to conclude the CPU is overloaded.

4.3 Memory Pressure and Swap

`free` (read-only) shows total, used, and available memory, plus swap usage. "Available" is the number worth trusting over "free" — Linux uses spare memory for caching, which `free`'s "free" column doesn't account for as reclaimable, while "available" does.

When a system runs out of memory and swap can't (or won't) absorb the pressure, the kernel's out-of-memory (OOM) killer terminates a process to recover memory — usually the one it judges to be using the most, not necessarily the one that caused the problem. This shows up in `dmesg` and `journalctl` output as an OOM-related kernel message naming the killed process. Finding that message is often the fastest way to confirm a job failure was memory-related rather than an application bug.

4.4 I/O Wait and Disk-Level Symptoms

`iostat` (read-only, from the `sysstat` package) reports per-device I/O statistics — how busy a disk is, how long requests are taking. `vmstat` (read-only) gives a broader system snapshot including a column for I/O wait, the percentage of time the CPU spent idle specifically because it was waiting on outstanding disk I/O.

The distinction that matters: a CPU showing high "wait" time is not a CPU problem — the CPU has nothing to do because storage hasn't responded yet. This is the single most common source of a load-average number that looks alarming next to a CPU-utilization number that doesn't. Note that exact column names and defaults for `iostat`/`vmstat` can differ slightly by `sysstat` version — the concept (idle-waiting-on-I/O vs. genuinely busy) is what to rely on, not a specific column position.

4.5 Process-Level Problems

`ps` shows a process state code alongside its PID — running, sleeping, or, notably, a "zombie" (defunct) state, where a process has finished but its exit status hasn't yet been collected by its parent. A small number of short-lived zombies is normal; a large or growing number usually points to a parent process that isn't cleaning up after its children, which is worth investigating rather than repeatedly working around.

A hung process — one that's running but not making progress — can sometimes be recovered with a `SIGTERM` (a normal `kill <pid>`), which asks it to exit cleanly. `kill -9` (`SIGKILL`) forcibly terminates it, bypassing any cleanup the process would otherwise do — necessary sometimes, but genuinely disruptive: it can leave shared resources, lock files, or job accounting in an inconsistent state. Reach for `SIGTERM` first; treat `SIGKILL` as a deliberate escalation, not a default.

4.6 Network Symptoms at the Host Level

`ip a` and `ss` (both read-only, introduced in Chapter 3) are what you reach for to confirm a node's own interface is up and that a service is listening where expected. This section stops there deliberately: a host-level network check tells you about this one node's management-facing interface, not about the data/fabric network's health, which is Part 5's subject in full. A symptom that looks network-related but doesn't resolve at the host level is a hand-off point to Part 5, not something to keep chasing with host-level tools alone.

4.7 Kernel and System Messages

`dmesg` (read-only) shows the kernel's ring buffer — recent kernel-level messages, including hardware events, driver messages, and OOM notices. Access to kernel messages may be restricted: on systems with `kernel.dmesg_restrict` enabled, unprivileged users may be unable to read the kernel log, and the exact privileges or capabilities required depend on the operating system and its configuration — don't assume `dmesg` is always readable without elevated access. `journalctl` (read-only) covers the same ground plus every service's own logging, searchable and filterable by time and unit.

These are evidence sources, not conclusions. A `dmesg` entry mentioning a hardware-adjacent term doesn't automatically mean a hardware failure — plenty of kernel messages are informational or reflect transient, self-recovered conditions. Reading the message carefully, and checking whether it recurs, is part of the "Collect Evidence" stage — not something to skip past on the way to a theory.

4.8 Worked Scenarios

The five scenarios below apply Sections 4.2–4.7 using the full ten-section format. All are **[Illustrative]** — realistic, but not claims of a specific real incident.

[Illustrative] **Scenario 1 — High CPU With No Job Assigned**

SYMPTOM A compute node shows sustained high CPU usage in monitoring, but the scheduler reports no job currently assigned to it.

INITIAL CHECKS	Confirm the node's actual state with `top`, not just the monitoring dashboard, in case of a reporting lag.
COMMANDS	`top`, `ps` (to identify the specific process by PID and owner).
OUTPUT INTERPRETATION	`top` shows one or more processes consuming significant CPU; `ps` reveals their owner and start time.
POSSIBLE CAUSES	A leftover process from a previous job that didn't terminate cleanly; a system-level process (e.g., a health-check script) running unexpectedly often; a zombie-accumulation pattern from Section 4.5.
FAULT ISOLATION	Check the process owner and start time against the scheduler's job history for that node — does it correspond to a job that should have ended?
ROOT CAUSE	**A prior job's process was not fully terminated when the job completed, leaving it running outside the scheduler's accounting.**
RESOLUTION	Terminate the orphaned process (`SIGTERM` first), and confirm the node returns to an idle CPU baseline.
VALIDATION	Re-check `top` after resolution; confirm CPU usage matches an idle node's expected baseline.
PREVENTION	Investigate why the job's cleanup didn't run — often a scheduler epilog/cleanup step (Part 7) worth reviewing, not just a one-off oddity.

[Illustrative]

Scenario 2 — A Job Is Killed by the OOM Killer

SYMPTOM	A user's job terminates unexpectedly partway through, with no application-level error message.
INITIAL CHECKS	Check whether the job's expected memory usage was close to or exceeded the node's available memory.
COMMANDS	`dmesg`, `journalctl` (searching around the job's failure time), `free` (for current state, though the event has already passed).
OUTPUT INTERPRETATION	A kernel message naming the job's process as OOM-killed, with a timestamp matching the failure.
POSSIBLE CAUSES	The job genuinely needs more memory than requested/available; another process on the same node was also consuming memory concurrently; a memory leak within the job itself.
FAULT ISOLATION	Compare the job's requested resources (Part 7 covers how jobs request memory) against what the node actually had available at the time.
ROOT CAUSE	**The job's actual memory usage exceeded what was available on the node, triggering the kernel's OOM killer.**
RESOLUTION	Resubmit with an appropriately larger memory request, or reduce the job's memory footprint if that's feasible.

VALIDATION	Confirm the rerun job completes without triggering an OOM event, checking `dmesg` / `journalctl` again after.
PREVENTION	If this recurs across many jobs, it may indicate memory requests are systematically underestimated for this workload type — worth flagging beyond a single job.

[Illustrative]

Scenario 3 — I/O Wait Spike Blocking a Job

SYMPTOM	A job appears "stuck" — low CPU usage, no progress, but not exited or erroring.
INITIAL CHECKS	Check load average against actual CPU utilization; a mismatch is the first clue.
COMMANDS	`vmstat`, `iostat`.
OUTPUT INTERPRETATION	`vmstat`'s wait column is elevated; `iostat` shows a specific device with high utilization and long request times.
POSSIBLE CAUSES	The job's own I/O pattern (many small reads/writes); contention from another job's I/O on the same shared storage; a genuinely degraded storage path.
FAULT ISOLATION	Check whether other jobs on other nodes accessing the same storage are also affected — a shared-resource contention pattern points away from this node specifically.
ROOT CAUSE	Concurrent heavy I/O from another job sharing the same storage backend was saturating available throughput.

RESOLUTION	No node-level fix required; the underlying storage-contention question is handed off to Chapter 8's methodology if it recurs.
VALIDATION	Confirm the job's I/O wait drops and progress resumes once contention eases.
PREVENTION	If this is a recurring pattern, it's a capacity/scheduling conversation (Part 4/7), not a per-incident node fix.

[Illustrative]

Scenario 4 — A Hung Process Ignores Normal Termination

SYMPTOM	A process is consuming resources and not responding to a normal termination request.
INITIAL CHECKS	Confirm the process's state via `ps` — is it actually running, or blocked on something (e.g., uninterruptible sleep, waiting on I/O)?
COMMANDS	`ps` (state column), `kill` (SIGTERM), then `kill -9` (SIGKILL) only if SIGTERM has no effect after a reasonable wait.
OUTPUT INTERPRETATION	`ps` shows the process in a state that doesn't respond to signals cleanly (e.g., waiting on an unresponsive resource).
POSSIBLE CAUSES	The process is blocked on an unresponsive storage or network resource; a driver- or kernel-level issue is holding it in an uninterruptible state.
FAULT ISOLATION	Check what resource the process is blocked on, if determinable, before escalating to a forceful kill.

ROOT CAUSE	The process was blocked waiting on a resource that itself wasn't responding — the process was a symptom, not the cause.
RESOLUTION	SIGKILL the process once SIGTERM is confirmed ineffective, understanding it may leave the underlying resource issue unresolved.
VALIDATION	Confirm the process is actually gone (a SIGKILL doesn't always work instantly against a process stuck in an uninterruptible kernel wait).
PREVENTION	Investigate the underlying resource the process was blocked on — killing the symptom without addressing that risks a repeat.

[Illustrative]

Scenario 5 — A Kernel Message Is Mistaken for an Application Bug

SYMPTOM	An application reports an unusual error, and the user assumes it's a bug in their code.
INITIAL CHECKS	Check `dmesg` / `journalctl` around the failure time before accepting the application's own error message as the full picture.
COMMANDS	`dmesg`, `journalctl`.
OUTPUT INTERPRETATION	A kernel-level message at the same timestamp points to a lower-level cause the application was simply reporting the symptom of.

POSSIBLE CAUSES	A hardware-adjacent kernel event; a resource limit being enforced at the kernel level; genuinely unrelated timing coincidence (worth ruling out, not assuming).
FAULT ISOLATION	Confirm the timestamps actually correlate, and check whether the kernel message recurs independently of this specific application.
ROOT CAUSE	**In this illustrative case, a lower-level kernel event (not the application's own logic) was the actual cause the application surfaced only indirectly.**
RESOLUTION	Address the underlying condition the kernel message pointed to, rather than debugging application code that was working as intended.
VALIDATION	Re-run and confirm the kernel message doesn't recur alongside a clean application run.
PREVENTION	This is the chapter's core lesson restated as a habit: collect evidence — including kernel/system messages — before assuming which layer actually failed.

PART 3

Compute & Hardware

Chapter 5, Chapter 6

CHAPTER 5

HARDWARE

CPU, Memory and NUMA

5.1 Where This Fits

Chapter 4 gave you a method for asking "is something wrong." This chapter is different in kind: it's not about diagnosing a problem, but about reading a node's CPU and memory topology so that later diagnoses — and later chapters on GPU and Slurm binding — have somewhere to attach. Nothing here is a general performance-triage method; that stays in Chapter 4.

5.2 CPU Building Blocks

A server has one or more sockets, each holding a physical CPU package. Each CPU package has multiple cores, each capable of independent execution. Many CPUs also support simultaneous multithreading (SMT, often called hyperthreading on some platforms), which allows a physical core to expose multiple logical CPUs to the operating system, sharing the core's execution resources rather than duplicating them outright. The number of logical threads per core depends on the processor architecture and configuration — two-way SMT is common, but it is not universal.

This matters practically because "thread count" and "core count" are not the same number, and a workload's performance doesn't scale linearly just because more logical threads are available — logical threads sharing one SMT-enabled core are competing for that core's actual execution resources, not running fully independently.

5.3 Memory and NUMA Nodes

On a multi-socket server, memory is physically attached in banks associated with each socket, not shared uniformly. A CPU accessing memory attached to its own socket ("local" access) is faster than accessing memory attached to a different socket ("remote" access), which has to cross an inter-socket interconnect. This is Non-Uniform Memory Access (NUMA): the cost of a memory access depends on where, relative to the requesting CPU, that memory physically lives.

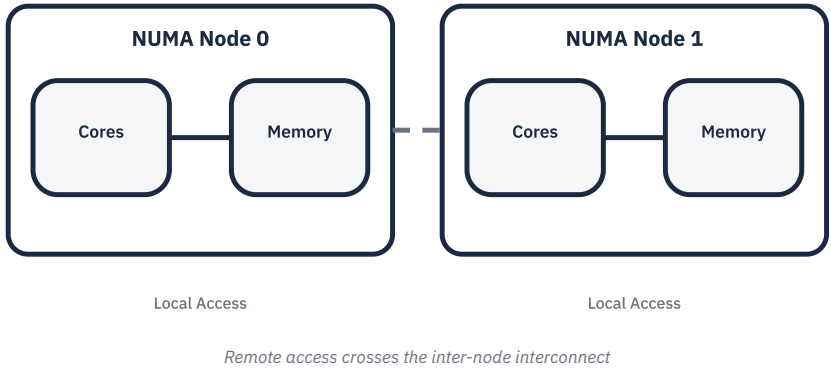


Figure 5.1 — NUMA Topology. Memory access cost depends on which NUMA node a process's memory lives on relative to the core it's running on. *Components: two (or more) NUMA nodes, each with CPU cores and locally attached memory, connected by an inter-node interconnect. Labels: "NUMA Node 0," "NUMA Node 1," "Local Access," "Remote Access (crosses interconnect).".*

A process isn't automatically kept on the NUMA node closest to its own memory — without guidance, the kernel scheduler and memory allocator make reasonable but not always optimal choices, especially for long-running, memory-intensive HPC workloads where the cost of remote access, paid repeatedly over a long run, adds up.

5.4 CPU Affinity and Pinning

CPU affinity is the practice of constraining a process to run only on a specified set of CPU cores, rather than letting the kernel scheduler move it freely. Pinning is affinity applied deliberately, usually to keep a process on the same NUMA node as the memory it's using, avoiding the remote-access cost described above.

Pinning is a tool, not a universal improvement. A process pinned to a NUMA node that doesn't actually hold its memory — because of a mismatch between how it was launched and how its memory was allocated — can perform worse than an unpinned process the kernel was free to place sensibly. Binding correctly requires knowing the actual topology, not just applying a fixed recipe to every job.

5.5 Cache Awareness, Practically

Beyond NUMA, each core also has its own cache hierarchy, and cores on the same socket typically share a level of cache with each other. At a practical level, this matters mainly as a reason why keeping related work on cores that are "close" to each other (same socket, ideally same NUMA node) tends to help more than spreading it arbitrarily — the same locality principle as NUMA, one level closer to the core. This book doesn't go deeper into cache theory than that.

5.6 Reading Topology Information

`lscpu` (read-only) reports a node's CPU topology: socket count, cores per socket, threads per core, and NUMA node layout. `numactl --hardware` (read-only) reports NUMA topology specifically — how many NUMA nodes exist and how much memory each holds. `numastat` (read-only) reports NUMA-related

memory statistics per process or system-wide, including how much of a process's memory access has been local versus remote — the direct evidence for whether a NUMA problem is actually occurring.

`numactl` can also be used to launch a process with specific binding (e.g., constraining it to a NUMA node's CPUs and memory) — this changes how the launched process runs, though it doesn't alter any persistent system configuration.

Table 5.1 — Topology Commands, Quick Reference

Command	Shows	Safety
`lscpu`	Socket/core/thread/NUMA layout	SAFE / READ-ONLY
`numactl --hardware`	NUMA node count and memory per node	SAFE / READ-ONLY
`numastat`	Local vs. remote memory access per process	SAFE / READ-ONLY
`numactl <bind-flags> <command>`	Launches a process with specified binding	CHANGES STATE (for the launched process only)

5.7 The CPU/Memory/Accelerator Relationship

The same locality principle extends beyond CPU and memory. A GPU or a network interface is also physically attached "close" to one NUMA node or another, and a job moving data between a CPU, a GPU, and a NIC pays a locality cost analogous to remote memory access if those components aren't well-matched. Part 6 covers this in full for GPUs specifically, and Part 7 covers how Slurm exposes binding controls for it.

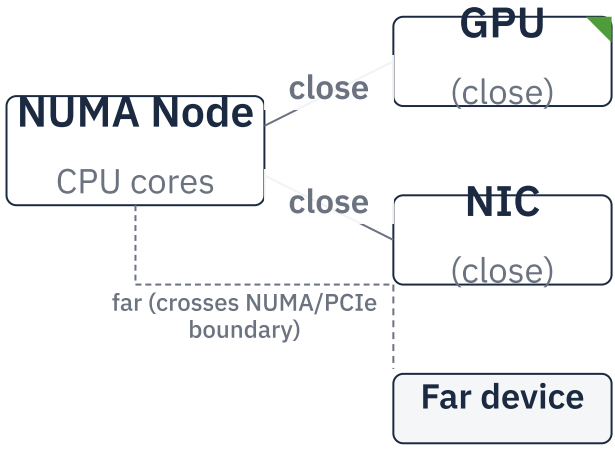


Figure 5.2 — CPU–GPU–NIC Affinity Preview. The same locality principle applies to GPUs and network interfaces — Part 6 covers this in depth. *Components: one NUMA node's CPU cores, paired with a GPU and a NIC attached "close" to that node versus "far" from it, mirroring Figure 5.1's style. Labels: "Close (same NUMA node)," "Far (crosses NUMA/PCIe boundary).".*

5.8 Misconceptions and Illustrative Scenarios

A few habits of mind are worth correcting early. More logical threads doesn't automatically mean more throughput — SMT contention on a busy core can make those threads collectively slower than running fewer of them would be. NUMA effects aren't limited to unusually large jobs; any latency-sensitive workload can be affected. And binding is not "set and forget" — a binding that doesn't match a job's actual memory allocation pattern can hurt more than leaving the kernel to decide.

[Illustrative] Two otherwise identical compute nodes run the same job, and one consistently finishes slower. `numastat` on the slower node shows a high proportion of remote memory accesses, while the faster node's shows mostly local access — the job's process landed on a different NUMA node than most of its memory, likely due to how it was launched rather than any hardware difference between the two nodes.

[Illustrative] A user suspects "the node is slow" based on a vague sense that a job usually finishes faster. Running `numastat` during the job confirms — rather than assumes — that remote memory access is elevated, turning a vague impression into evidence that supports adjusting how the job is bound, consistent with Chapter 1's "Collect Evidence" stage.

CHAPTER 6

HARDWARE

HPC Server Hardware

6.1 Where This Fits

Chapter 5 covered CPU and memory topology as something you read. This chapter zooms out one level further, to the physical server itself — what's inside it, how its health is signaled, and what happens across its lifecycle from arrival to eventual replacement. None of this is a vendor manual; the goal is vendor-neutral recognition, not brand-specific procedure.

6.2 Server Building Blocks

An HPC server, general-purpose or GPU-equipped, is built from a recognizable set of components, each with its own typical failure signature.

Table 6.1 — Server Component Inventory

Component	Role	Typical Failure Signal
CPU	Executes general-purpose instructions	Overheating alerts, unexpected throttling, correctable/uncorrectable errors reported by firmware
DIMMs (memory modules)	Provide system memory	ECC-correctable error counts rising over time; uncorrectable errors causing a crash

GPU/accelerator (GPU nodes only)	Executes accelerator-bound workloads	Not detected by the OS, thermal throttling, ECC errors — Part 6 covers this in depth
NIC/HCA	Network/fabric connectivity	Link down, not detected, degraded negotiated speed
Storage controller and local disks	Local (non-shared) storage, boot devices	SMART-reported predictive failure, I/O errors
PCIe	Interconnects CPU to GPU, NIC/HCA, and storage controllers	Link trained at a lower width/speed than expected
Power supply units (PSUs)	Convert and deliver power, often in a redundant pair	One PSU reporting a fault while the node keeps running on the other
Fans	Cooling	Reported RPM drop, thermal alerts as an indirect signal

On a GPU node (Chapter 2), the accelerator block is present and materially changes the node's power and cooling profile relative to a general-purpose compute node — a point Section 6.7 returns to.

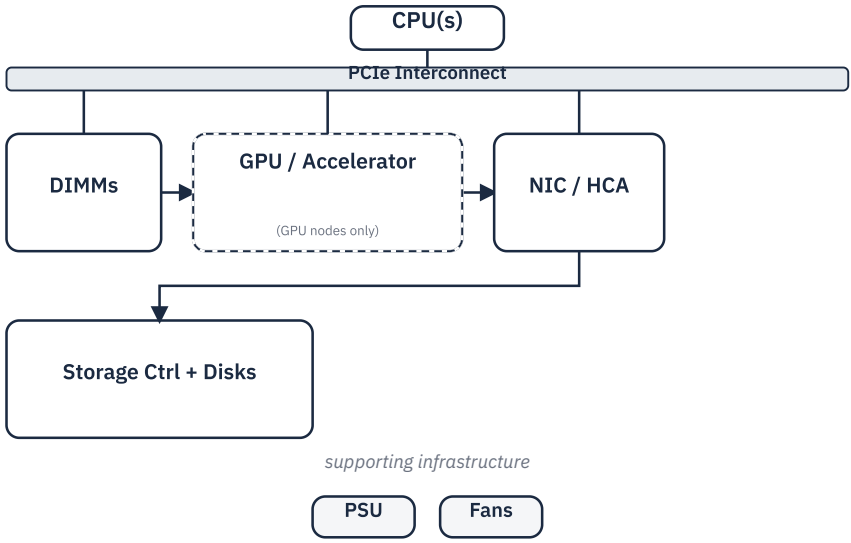


Figure 6.1 — HPC Server Block Diagram. A generic HPC server's major components. GPU nodes (Chapter 2) add the accelerator block; general-purpose compute nodes may not have one. *Components: CPU(s), DIMMs, GPU/accelerator (dashed block, present on GPU nodes), NIC/HCA, storage controller with local disks, PCIe interconnect linking them, PSU, fans. Connections: PCIe bus linking CPU to GPU/NIC/storage controller; PSU/fans shown as supporting infrastructure.*

6.3 Firmware/BIOS Baseline Concepts

Every component above runs its own firmware, and the server's BIOS/UEFI settings govern how those components initialize and interact. A firmware baseline is the set of firmware and BIOS versions a site standardizes on across a fleet of otherwise-identical nodes. The point of maintaining one isn't that older or newer firmware is inherently wrong — it's that inconsistency across a fleet turns "this node behaves oddly" into a harder problem than it needs to be, since two nodes that should be identical no longer are.

6.4 Hardware Health Signals: OS-Visible vs. BMC-Visible

Most servers include a baseboard management controller (BMC) — a small, independent management processor that can report on hardware health even when the main operating system is unresponsive or hasn't booted. This creates two distinct visibility layers, and they don't always agree.

Table 6.2 — OS-Visible vs. BMC-Visible Signals

Symptom Type	Visible to the OS?	Visible to the BMC?
A correctable memory error, occurring occasionally	Sometimes, via kernel logs	Yes, typically logged to the System Event Log (SEL)
A fan running below expected RPM	Rarely, unless a monitoring agent reads sensor data	Yes, directly
A failed PSU in a redundant pair (node still running)	Rarely, if the OS never loses power	Yes, directly
A PCIe link training at reduced width	Sometimes, via kernel boot messages	Sometimes, depending on platform
A GPU not initializing	Yes, typically absent from `nvidia-smi`'s device list	Sometimes, depending on platform

The operational implication: relying on OS-visible signals alone misses a category of hardware degradation the BMC would have caught earlier. Vendor-specific BMC tools exist for reading this out-of-band data — this book names the category rather than teaching a specific tool, since interfaces vary meaningfully by vendor.

6.5 Logs and Events as Hardware Evidence

`dmesg` (Chapter 4; note there the caveat about restricted read access on some systems) surfaces hardware-relevant kernel messages — PCIe link-training results, memory errors the kernel itself detected, and similar events — and remains one of the first places to look. The BMC's own System Event Log (SEL) is the out-of-band equivalent, capturing events the OS-level log may never see at all, particularly anything occurring before the OS has booted or after it has crashed.

6.6 Burn-In and Validation Concepts

Burn-in is a period of sustained, deliberately demanding workload run against a node before it's accepted into production — the goal is surfacing marginal hardware that passes a quick power-on test but fails under real, sustained load. Validation is the broader practice of confirming a node meets its expected baseline (firmware versions, component inventory, and passing a burn-in workload) before it's trusted with real jobs. This book treats both at the level of what they're for; Part 10 covers running them operationally.

6.7 Node Recovery and RMA Concepts

Node recovery is the process of returning a node to service after a fault — which might mean a simple reboot, a firmware update, a component reseat, or, when a component is genuinely defective, a Return Merchandise Authorization (RMA): formally returning a

failed part to the vendor for replacement or repair. RMA processes generally require documentation — what failed, what evidence supports that conclusion, and when — both to satisfy the vendor's process and to build a fleet-wide record of what tends to fail and why. This chapter introduces the concept; Part 10 covers the operational procedure and documentation discipline in depth.

6.8 Misconceptions and Scenarios

If the OS doesn't see a problem, it's tempting to conclude there isn't one — but Section 6.4 showed that's not reliable. A reboot resolving an intermittent issue doesn't mean the issue is fixed; it often means the underlying condition was temporarily cleared, not addressed. And firmware versions "not mattering once installed" ignores that drift across a fleet is itself a source of inconsistent behavior.

[Illustrative] A node passes every OS-level health check, but its BMC's SEL shows a slowly rising count of correctable memory errors on one DIMM. Left alone, correctable errors don't crash anything — but a rising trend is evidence worth acting on before it becomes an uncorrectable one, which is exactly the kind of signal Section 6.4's OS-vs-BMC distinction is meant to catch early.

[Illustrative] A node exhibits an intermittent, hard-to-reproduce fault. A reboot appears to resolve it, and the node returns to service — only for the same fault to reappear weeks later. The reboot didn't fix anything; it happened to clear a transient state that the underlying (unaddressed) condition eventually reproduced.

[Illustrative] Hardware inconsistency observed across a batch of otherwise-identical nodes is eventually traced to firmware drift rather than a hardware defect in any individual node — a reminder that "identical hardware" doesn't guarantee identical behavior once firmware versions have been allowed to diverge across a fleet.

PART 4

HPC Storage

Chapter 7, Chapter 8

CHAPTER 7

STORAGE

HPC Storage Architecture

7.1 Where This Fits

Chapter 2 introduced shared storage as a system, deliberately without depth, and pointed here for the rest. This chapter delivers that depth: what "a system, not a server" actually means, and how NFS, Lustre, and IBM Storage Scale each implement the idea differently. Chapter 8 is where you'll diagnose problems in what this chapter describes.

7.2 Local vs. Shared Storage

Local storage — disks physically attached to one server — is fast and simple but visible only to that one node. HPC workloads routinely span many nodes, and a job that starts on one compute node needs to see the same files whether it (or its output) is later touched from another node entirely. Shared storage exists to make that true: one storage system, visible consistently across every node that needs it.

7.3 Shared Filesystem Concepts: Client and Server, Metadata and Data

A shared filesystem has clients — the compute, GPU, and login nodes accessing it — and one or more servers actually holding the data and serving requests. Nearly every HPC-scale shared

filesystem also separates two kinds of operations: metadata operations (information about files — names, permissions, sizes, locations) and data operations (the actual file contents being read or written). The reason for the split is throughput isolation: an operation that only needs metadata — listing a directory, checking a file's size — shouldn't have to wait behind a large, unrelated data transfer competing for the same resource, and vice versa.

How this separation is actually implemented — as dedicated servers, as separate services, or some other arrangement — varies considerably between storage systems, which the next three sections make concrete.

7.4 NFS

NFS (Network File System) is the most broadly familiar shared filesystem model: a server exports a directory, and clients mount it over the network. In its traditional form, one server handles both metadata and data operations for what it exports, which keeps the architecture simple but means that server is the single point serving all client requests for that export. NFS remains common in HPC environments, particularly for home directories and for workloads that don't need the throughput a parallel filesystem is built for — though whether NFS suits a given workload depends on the workload, not on NFS's age or ubiquity.

7.5 Lustre

Lustre is a parallel filesystem purpose-built for HPC-scale throughput. It splits metadata and data explicitly across dedicated components: one or more Metadata Servers (MDS), each managing a Metadata Target (MDT) holding filesystem structure information, and one or more Object Storage Servers (OSS), each managing one or more Object Storage Targets (OST) holding actual file data. A client's metadata operations go to the MDS; its data operations go

directly to the relevant OSS/OST — in parallel, across multiple OSTs for a single large file if it's striped across them, which is where Lustre's throughput advantage for large, sequential I/O comes from.

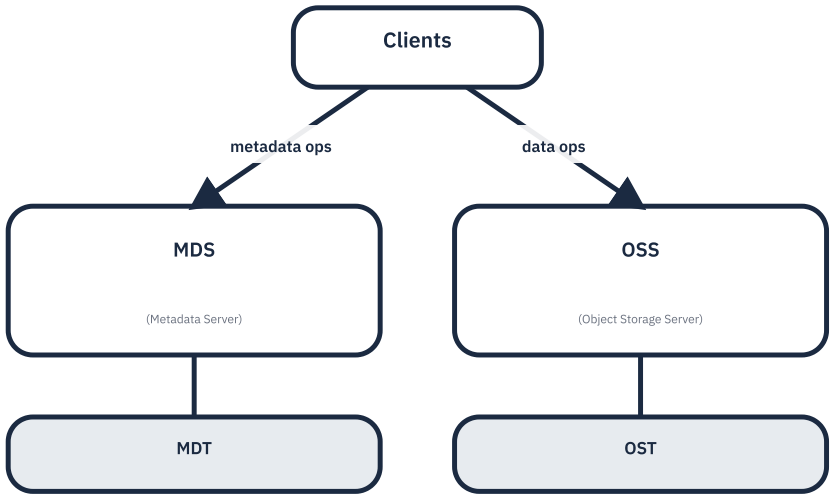


Figure 7.1 — Lustre Architecture. Lustre separates metadata operations (MDS/MDT) from data operations (OSS/OST) so one doesn't block the other. *Components: Clients, Metadata Server(s) (MDS) with Metadata Target (MDT), Object Storage Server(s) (OSS) with Object Storage Targets (OST), network fabric connecting all. Connections: Clients → MDS (metadata operations) and Clients → OSS (data operations), both over the fabric.*

7.6 IBM Storage Scale (GPFS)

IBM Storage Scale (formerly known as GPFS, General Parallel File System) is another parallel filesystem used at HPC scale, built around a somewhat different internal architecture — data is organized across Network Shared Disks (NSDs), with metadata handling distributed rather than concentrated on dedicated MDS-style servers in the way Lustre structures it. The underlying goal is the same as Lustre's — parallel, high-throughput access across many clients — achieved through a different specific mechanism.

This book doesn't take a position on which approach is preferable; the two systems solve the same problem with different architectural choices, each with real deployments at scale.

7.7 Comparing the Three, Conceptually

Table 7.1 — NFS vs. Lustre vs. IBM Storage Scale, by Use-Case Fit

	NFS	Lustre	IBM Storage Scale
Metadata/data separation	Not separated in the traditional model	Explicit (MDS/MDT vs. OSS/OST)	Distributed, not concentrated on dedicated metadata servers
Typical HPC role	Home directories, smaller-scale shared access	Large-scale parallel scratch/work storage	Large-scale parallel storage, often in mixed or multi-site deployments
Scaling characteristic	Single-server bottleneck in its traditional form	Scales by adding OSS/OST pairs	Scales by adding NSD servers/disks

This table is a conceptual orientation, not a benchmark or a recommendation — the right choice for a given site depends on workload, existing infrastructure, and operational expertise, none of which this book can generalize about responsibly.

7.8 Throughput, IOPS, Latency, and Capacity

These four terms get used loosely but describe different things. Throughput is data moved per unit time — relevant for large, sequential transfers. IOPS (I/O operations per second) matters more for many small operations, common in metadata-heavy workloads. Latency is the delay before an operation completes, which matters even when throughput and IOPS both look fine, particularly for workloads sensitive to per-operation delay. Capacity is simply how much data fits — and a filesystem nearing capacity can behave differently (sometimes worse) than the same filesystem with room to spare, independent of its raw throughput or IOPS numbers. Treating any one of these as a stand-in for "storage performance" overall is a common oversimplification.

7.9 Storage/Network Relationship

Shared storage, as Chapter 2 established, is reachable over the same data/fabric network that carries job-to-job traffic — or, depending on site design, over a separately built storage network. Either way, storage performance and network performance are connected: a job's read or write competes for the same network resources as other traffic on that path.

Figure 7.2 — Storage Network Topology. Storage traffic and job-to-job traffic share the same data/fabric network introduced in Chapter 2 — or, depending on site design, a separately built storage network. *Components: compute/GPU nodes, a generic storage system block, and the fabric connecting them, consistent with*

Chapter 2's terminology. Connections: Compute/GPU ↔ Fabric ↔ Storage, single path.

7.10 Trade-offs and an Illustrative Scenario

Every architecture choice in this chapter trades something for something else: NFS's simplicity against its single-server bottleneck; Lustre's and Storage Scale's parallel throughput against their added operational complexity. None of these trade-offs make one system universally better — they make each system a better or worse fit for a specific workload.

[Illustrative] A team selects a filesystem primarily for its large-file sequential throughput numbers, without weighing that their actual workload consists mostly of many small files with heavy metadata activity — code repositories and small configuration files, not large simulation output. The mismatch shows up as unexpectedly poor performance despite the filesystem's throughput credentials being entirely accurate for the workload it was actually designed around.

CHAPTER 8

STORAGE

Storage Troubleshooting

8.1 The Layered Method

A storage complaint almost always arrives as one word: slow, stuck, or unavailable. None of those words say where the actual problem is. This chapter organizes storage troubleshooting around five layers a symptom can originate from — Application, Client, Network, Storage Service, Storage Backend — and the discipline of gathering evidence to figure out which one is actually at fault before doing anything about it.

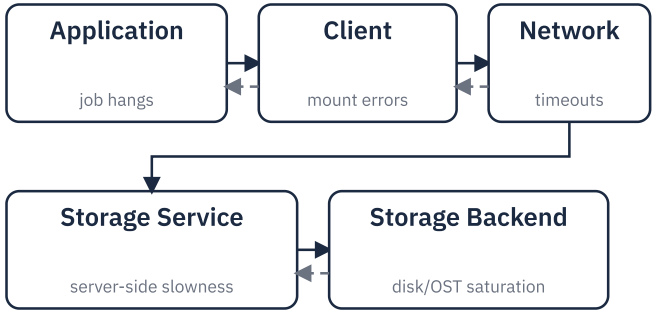


Figure 8.1 — Storage Troubleshooting Layer Model. Five layers a storage symptom can originate from. Section 8.2 shows how evidence narrows down which one. *Components: five labeled stages in sequence — Application, Client, Network, Storage Service, Storage Backend — each with one example symptom ("Application: job hangs," "Client: mount errors," "Network: timeouts," "Storage Service: server-side slowness," "Storage Backend: disk/OST saturation"). Connections: a linear chain,*

with a note that evidence collection can start at any point and work in either direction.

8.2 "Filesystem Is Slow" vs. "Application Is Slow"

These are different claims, and confusing them wastes time. "The filesystem is slow" implies every client, every application, touching that filesystem should show similar symptoms. "The application is slow" might mean only this one job, with its particular access pattern, is affected — while everything else touching the same storage is fine.

The fastest way to tell them apart is comparison: is another job, on another node, accessing the same storage right now, also slow? If yes, the evidence points toward the Network, Storage Service, or Storage Backend layers — something shared. If no — if this one application is uniquely affected — the evidence points toward the Application or Client layer: this job's own access pattern, or something specific to the node it's running on.

8.3 Mount and Access Problems

A filesystem that isn't mounted at all is the simplest case, and `mount` (Chapter 3, read-only when run without arguments) confirms it quickly. A "stale" mount — one that was working but has lost contact with its server without the client fully noticing — behaves differently: commands against it hang rather than failing immediately, because the client is still waiting for a server that isn't responding.

Permission problems are a Client/Application-layer issue more often than a storage-system fault: a user's access mismatch (ownership, group membership, permission bits — Chapter 3) frequently looks like a storage problem to the person hitting it, but resolves the same way any other Linux permission issue does.

8.4 Capacity and Metadata Exhaustion

`df` (Chapter 3, read-only) shows space usage, but "space available" and "can still write" are not always the same claim. A filesystem can show free space by capacity while being unable to accept new files because it has run out of inodes — the accounting structures a filesystem uses to track each file, which are finite and separate from raw storage capacity. Metadata pressure is a related but distinct symptom: not exhaustion, but a metadata service (an MDS, for instance) under enough load that metadata-only operations — listing a directory, checking a file's status — slow down independently of actual data throughput.

8.5 I/O Wait and Hanging Applications

Chapter 4 introduced I/O wait at the host level; here, the same evidence (`vmstat`, `iostat`) gets applied specifically to shared-storage cases. The distinguishing question at this layer, building on Section 8.2: is the I/O wait isolated to this node/job, or visible everywhere touching the same storage? An application that hangs entirely, rather than just running slowly, often indicates a client waiting on a request the server never answered — closer to a stale-mount-style symptom than ordinary throughput contention.

8.6 Client vs. Server Distinction; NFS Notes

Every symptom in this chapter can, in principle, originate on the client side (this one node's mount, cache, or network path) or the server side (the actual storage service or backend). NFS makes this distinction sharply visible: a client holding a stale file handle — a reference to a file the server no longer recognizes, often after a server restart or export change — produces client-side hangs or errors that have nothing wrong with the server's current state at

all. Confirming whether other clients mounting the same export are affected is the fastest way to separate a client-specific problem from a server-side one.

8.7 Lustre and Storage Scale Conceptual Notes

Chapter 7 covered these systems' architecture; here, the focus stays symptom-level. In a Lustre-style architecture, a symptom isolated to metadata operations (directory listings, file-status checks) while data throughput remains fine points toward the MDS specifically, distinct from an OSS/OST-level symptom affecting actual data transfer. The same client-vs-server, metadata-vs-data isolation logic from earlier sections applies directly — this chapter doesn't introduce new commands for these systems beyond what Chapter 7 already named conceptually, since exact diagnostic tooling for a specific parallel filesystem is worth verifying against that system's current documentation rather than treated as fixed here.

8.8 Misconceptions

"Storage is slow" is not, by itself, a diagnosis — Section 8.2 exists because that phrase can mean five different things. Free space reported by `df` doesn't guarantee you can still write (Section 8.4). And a hung `ls` doesn't mean the whole filesystem is unavailable — it might mean one slow metadata operation, not systemic failure.

8.9 Worked Scenarios

Table 8.1 — Command Safety, This Chapter

Category	Commands
● SAFE / READ-ONLY	`df`, `du`, `lsblk`, `mount` (no arguments), filesystem-specific read-only status queries

■ CHANGES STATE	`mount <device> <path>`, `umount`
▲ POTENTIALLY DISRUPTIVE	Forcing an unmount of a filesystem with active users or open files

[Illustrative]

Scenario 1 — Filesystem Not Mounted After Reboot

SYMPTOM

After a node reboot, a job fails immediately with a "path not found" error for a directory that should exist.

INITIAL CHECKS

Run `mount` to confirm whether the expected filesystem is actually mounted.

COMMANDS

`mount`, `df`.

OUTPUT INTERPRETATION

The expected mount point is absent from `mount`'s output entirely.

POSSIBLE CAUSES

The mount wasn't configured to persist across reboot; a dependency (network, a prerequisite service) wasn't ready when the mount was attempted at boot.

FAULT ISOLATION

Check boot-time logs (`journalctl`) for a mount failure message around boot time.

ROOT CAUSE

The mount attempt at boot ran before the network path to the storage server was ready, and wasn't retried.

RESOLUTION

Mount it manually to restore service immediately, then correct the boot-time ordering/dependency so it mounts reliably on future reboots.

VALIDATION	Reboot the node (in a maintenance window) and confirm the mount succeeds automatically.
PREVENTION	Check other nodes provisioned the same way for the same latent misconfiguration, rather than treating this as isolated.

[Illustrative]

Scenario 2 — "Slow Filesystem" Traced to an Application's Access Pattern

SYMPTOM	A user reports the shared filesystem is "slow" for their job.
INITIAL CHECKS	Check whether another job, on another node, accessing the same storage right now is also slow (Section 8.2's comparison test).
COMMANDS	`iostat`, `vmstat`, plus checking whether other clients are similarly affected.
OUTPUT INTERPRETATION	Other clients show normal I/O behavior; only this job's node shows elevated wait time, correlated with this job's own I/O calls.
POSSIBLE CAUSES	The job itself is issuing many small, uncoalesced I/O operations rather than fewer, larger ones; the storage system genuinely has a problem (ruled out by the comparison test above).
FAULT ISOLATION	The comparison test isolates this to the Application layer, not Network/Storage Service/Backend.

ROOT CAUSE	The application's I/O pattern (many small operations) doesn't suit the storage system's strengths, independent of any storage-system fault.
RESOLUTION	Adjust the application's I/O pattern (e.g., batching writes) if feasible — a Storage Backend fix isn't applicable here since none exists.
VALIDATION	Confirm improved throughput after the access-pattern change, if made.
PREVENTION	Document the access-pattern guidance for similar workloads to avoid repeat investigations reaching the same conclusion.

[Illustrative]

Scenario 3 — Inode Exhaustion Despite Free Space

SYMPTOM	A job fails to create new files with a "no space" error, despite `df` showing available capacity.
INITIAL CHECKS	Check inode usage specifically, not just space usage.
COMMANDS	`df` (capacity), a filesystem-appropriate inode-usage check.
OUTPUT INTERPRETATION	Space usage shows well under capacity; inode usage shows at or near its limit.
POSSIBLE CAUSES	A workload generating an unusually large number of small files (common with certain checkpointing or logging patterns).
FAULT ISOLATION	Confirmed by inode usage specifically, ruling out a genuine capacity issue.

ROOT CAUSE	The workload's file-creation pattern exhausted available inodes well before exhausting raw capacity.
RESOLUTION	Clean up unnecessary small files, or adjust the workload to create fewer, larger files where feasible.
VALIDATION	Confirm inode usage drops to a sustainable level and the job can create files again.
PREVENTION	Monitor inode usage alongside capacity, not instead of it, for workloads known to generate many small files.

[Illustrative]

Scenario 4 — NFS Stale File Handle Causes a Hung Application

SYMPTOM	An application accessing an NFS-mounted path hangs indefinitely rather than erroring.
INITIAL CHECKS	Check whether the NFS server was recently restarted or reconfigured (a common trigger for stale handles).
COMMANDS	Client-side NFS mount status checks; comparing behavior against other clients mounting the same export.
OUTPUT INTERPRETATION	This client's handle to a specific file/directory no longer resolves on the server side, while other clients (or a fresh mount) work normally.
POSSIBLE CAUSES	Server-side export change or restart invalidated the client's cached file handle.

FAULT ISOLATION	Other clients unaffected narrows this to a client-side stale-reference issue, not a server-wide fault.
ROOT CAUSE	**The client held a reference to a file handle the server no longer recognized after its restart/reconfiguration.**
RESOLUTION	Remount the affected path on the client to obtain a fresh handle.
VALIDATION	Confirm the application can access the path normally post-remount.
PREVENTION	Where server restarts/reconfigurations are planned, communicate them so affected clients can remount proactively rather than discovering the hang mid-job.

[Illustrative]

Scenario 5 — Metadata Server Slowdown Under a Specific Workload Pattern

SYMPTOM	Metadata-only operations (directory listings, file-status checks) slow down noticeably across many clients, while large-file data throughput remains largely unaffected.
INITIAL CHECKS	Confirm the slowdown is isolated to metadata operations specifically, using the metadata-vs-data distinction from Section 8.7, rather than a general storage-wide symptom.
COMMANDS	Metadata-service-side load/status queries (implementation-specific); comparison of metadata-op latency against data-throughput measurements over the same period.

OUTPUT INTERPRETATION	Metadata-op latency is well above its normal baseline for the affected window while data throughput stays close to normal — confirming the bottleneck is specific to metadata handling, not general storage load.
POSSIBLE CAUSES	A workload pattern generating an unusually high rate of metadata operations (e.g., many small files, frequent directory scans); metadata-service resource contention independent of any single workload.
FAULT ISOLATION	Correlating the metadata-load spike against per-job scheduling data identifies a single running job whose pattern (a large scan across many small files) accounts for the bulk of the additional metadata operations during the affected window.
ROOT CAUSE	One job's file-access pattern generated a metadata operation rate the service wasn't sized to absorb alongside its normal load, producing latency for every client sharing that metadata service — not a metadata-service fault in itself.
RESOLUTION	Work with the job's owner to reduce the metadata-operation rate where possible (e.g., batching small-file access, avoiding repeated full-directory scans), and confirm the metadata service has no independent fault contributing to the slowdown.
VALIDATION	Confirm metadata-operation latency returns to baseline for affected clients after remediation.

PREVENTION	Document this workload pattern as a known metadata-load risk, and consider metadata-operation-rate limits or scheduling guidance for jobs with similar small-file-heavy access patterns going forward.

PART 5

InfiniBand & RDMA

Chapter 9, Chapter 10, Chapter 11, Chapter 12

CHAPTER 9

INFINIBAND / RDMA

InfiniBand Fundamentals

9.1 Where This Fits

Chapter 2 named "the cluster interconnect" without going further. This chapter is where InfiniBand specifically enters the book: its components, its vocabulary, and enough of its shape that a fabric diagram stops looking like an unlabeled tangle of lines. Administration (Chapter 11) and troubleshooting (Chapter 12) both build directly on what's introduced here — neither is covered in this chapter.

9.2 What InfiniBand Is and Why It Exists

InfiniBand is a networking technology built specifically for the low-latency, high-bandwidth, RDMA-capable communication that large-scale parallel computing depends on — originally designed as an interconnect for exactly this purpose, rather than adapted from general-purpose Ethernet networking. It remains a common choice for the data/fabric network in traditional HPC environments (Chapter 2), though it is not the only option — Chapter 10 covers RoCE as an Ethernet-based alternative offering similar RDMA capability.

9.3 HCA

A Host Channel Adapter (HCA) is the network interface card that connects a node to an InfiniBand fabric — the InfiniBand equivalent, in role, of an Ethernet NIC, though built around InfiniBand's own protocols and typically supporting RDMA directly in hardware.

9.4 Switches and Ports

An InfiniBand switch connects multiple links together, forwarding traffic between the HCAs and other switches attached to it — analogous in role to an Ethernet switch. A port is a specific physical connection point, on either an HCA or a switch, that a single link attaches to.

9.5 Links and Link States

A link is the physical connection — typically a cable — between two ports. Each port has a link state describing its current condition: commonly a progression through states like Down, Initializing, Armed, and Active, reflecting the process of a link being established and brought fully into service. "Active" is the state a healthy, ready-for-traffic link should show — though, as Chapter 12 covers, a link showing Active isn't automatically free of errors; link state and error-free operation are related but distinct facts.

9.6 Fabric and Topology, Conceptually

The fabric is the complete set of HCAs, switches, and links forming the network as a whole. At scale, InfiniBand fabrics are commonly built in tiered, tree-like topologies designed to give many nodes

multiple paths to reach each other — a subject Part 5's later chapters return to when it becomes operationally relevant (fault isolation, performance) rather than purely descriptive.

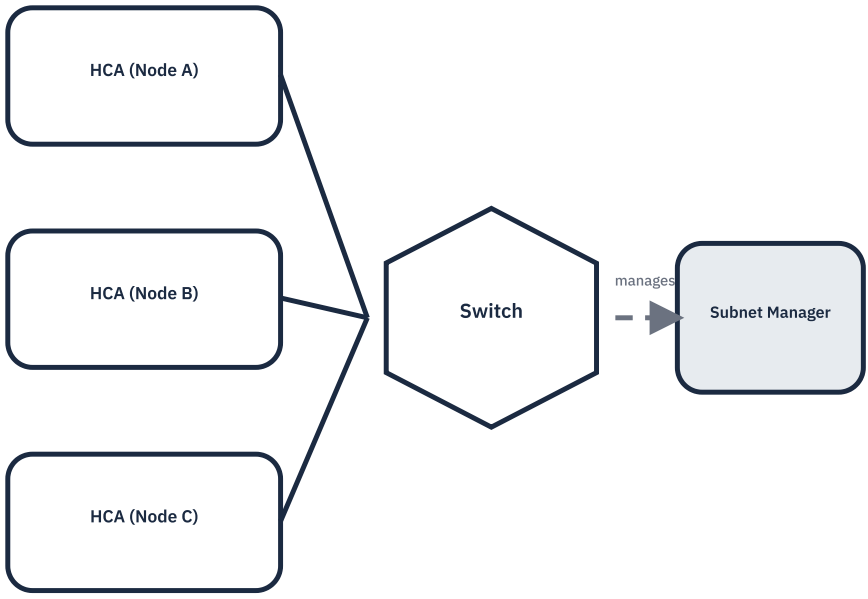


Figure 9.1 — InfiniBand Fabric Topology. A minimal InfiniBand fabric: HCAs, a switch, and the links between them, managed by a subnet manager. *Components: several HCAs (one per node), a switch (or two, showing a minimal multi-tier shape), links connecting HCAs to the switch and switches to each other, one subnet manager instance shown as a logical role.*

9.7 The Subnet Manager's Role

Unlike an Ethernet network, an InfiniBand fabric doesn't configure itself through the same mechanisms (like ARP and switch-learned MAC tables) Ethernet relies on. Instead, a subnet manager (SM) — software running somewhere on the fabric, often on a management node — discovers the fabric's topology, assigns addressing (Section 9.8), computes routing between nodes, and monitors the fabric's state, updating its configuration as things

change (a link going down, a node joining). Without an active subnet manager, an InfiniBand fabric generally can't pass traffic at all — which makes the SM's own health a fabric-wide concern, not an afterthought.

9.8 LIDs and GUIDs

Every port on the fabric has a Local Identifier (LID) — an address the subnet manager assigns, used for routing traffic within that subnet — and a Globally Unique Identifier (GUID), a fixed, hardware-assigned identifier similar in spirit to an Ethernet MAC address. Neither is an IP address, and thinking of them that way leads to confusion: LIDs are assigned and can change if the SM reassigns them (for instance, after a topology change), while an IP address, if one is needed on top of InfiniBand, is a separate concept layered on via IPoIB (Chapter 10).

9.9 Partitions and PKeys

InfiniBand supports partitioning a fabric into logical groups using partition keys (PKeys), controlling which ports can communicate with which others — conceptually similar in purpose to VLANs on Ethernet, though implemented differently. Not every deployment uses partitioning; where it's used, it's a subnet-manager-administered concept, covered operationally in Chapter 11.

9.10 Bandwidth vs. Latency

Bandwidth is how much data can move per unit time; latency is how long a single operation takes to complete, regardless of how much data is involved. InfiniBand is generally associated with both high bandwidth and low latency, but they're independent properties — a link can be high-bandwidth and still latency-

sensitive to congestion, and a workload's actual performance often depends more on which of the two it's more sensitive to than on either number in isolation.

9.11 RDMA Relationship

InfiniBand's most significant capability is native support for RDMA (Remote Direct Memory Access) — letting one node's application read or write another node's memory directly, without routing through the receiving side's CPU or full network stack. This is what gives InfiniBand its latency and CPU-overhead advantage for the workloads that need it. Chapter 10 covers RDMA in depth, including how it compares to running RDMA over Ethernet (RoCE).

9.12 IPoIB Relationship

IPoIB (IP over InfiniBand) lets ordinary IP traffic run over an InfiniBand fabric, which is convenient for compatibility with tools and applications that expect a standard IP interface — but it does not carry RDMA's performance characteristics; it's a compatibility layer, not a performance path. Chapter 10 covers this distinction in full.

9.13 UFM Context

NVIDIA UFM is a fabric management and monitoring platform that sits above manual, tool-by-tool inspection of a fabric, giving a consolidated view of fabric health and telemetry. It doesn't replace the subnet manager — it complements it, from a monitoring and management perspective. Part 8 (Chapter 22) covers UFM in depth; this chapter only places it in context.

9.14 Diagnostic Tools, Named Only

Two tools are worth recognizing by name at this stage, without yet learning their full usage: `ibstat`, which reports an HCA's port state and basic link information, and `ibv_devinfo`, which reports detailed information about the HCA device itself. Both are read-only. Chapter 11 covers using them, and several related tools, in practice.

Figure 9.2 — HCA-to-Switch Path. Every fabric connection is this chain. Chapter 12 uses this same chain to isolate faults. *Components: one node's HCA, its port, the link, the switch port it connects to. Connections: a linear chain — Node → HCA → Port → Link → Switch Port.*

9.15 Misconceptions and an Illustrative Scenario

LIDs and GUIDs are not IP addresses, and treating them as if the same rules applied would cause confusion once IPoIB enters the picture in Chapter 10. A link showing "Active" is not the same claim as "error-free" — Chapter 12 returns to this directly. And not every fabric requires manual LID assignment; the subnet manager typically automates it, though the specifics can vary by implementation.

[Illustrative] Before assuming a fabric-wide problem, an engineer runs `ibstat` on the node reporting trouble and confirms the port's link state and negotiated rate look normal — a quick, read-only check that either rules out the simplest explanation immediately or gives a concrete reason to look further, consistent with "Collect Evidence" before forming a theory.

CHAPTER 10

INFINIBAND / RDMA

RDMA, IPoIB and RoCE

10.1 Where This Fits

Chapter 9 named RDMA, IPoIB, and mentioned RoCE only in passing. This chapter is where each gets defined precisely enough that you can tell, given a description of a network, which one you're actually looking at — and why that distinction changes what's operationally required of it.

10.2 Traditional IP Networking, Briefly

In conventional IP networking, an application hands data to the operating system's network stack, which processes it through several layers, involves the CPU throughout, and eventually hands it to a NIC for transmission — and the reverse on receipt. This works well for most purposes, but every one of those steps costs CPU time and adds latency, which becomes a real limitation for the tightly-coupled, high-frequency communication common in HPC and distributed AI workloads.

10.3 What RDMA Actually Changes

RDMA (Remote Direct Memory Access) lets a network adapter move data directly between one machine's memory and another's, without routing the transfer through the CPU or the full kernel network stack on either side — commonly described as kernel bypass. Because the data doesn't need to be copied between kernel

and application buffers along the way either, this is often called zero-copy. The result is markedly lower latency and lower CPU overhead per transfer than traditional networking, which is precisely why RDMA-capable networking matters for HPC and AI workloads whose performance is sensitive to exactly those two things.

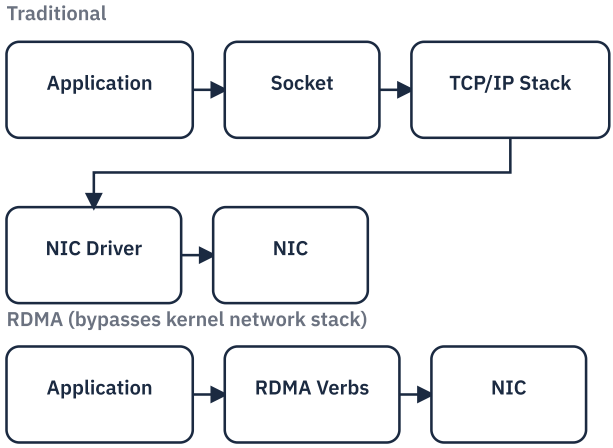


Figure 10.1 — RDMA Data Path vs. TCP/IP Stack. RDMA bypasses the kernel's network stack, which is the source of its latency and CPU-overhead advantage. *Components: two parallel stacks — Application → Socket → TCP/IP Stack → NIC Driver → NIC (traditional); Application → RDMA Verbs → NIC (RDMA, bypassing the kernel network stack).*

10.4 InfiniBand-Native RDMA

InfiniBand was designed with RDMA as a native capability rather than an add-on — an HCA (Chapter 9) implements RDMA operations directly, and applications access it through a low-level RDMA verbs interface rather than conventional sockets. This is the "default," highest-performing way RDMA is used on an InfiniBand fabric.

10.5 IPoIB

IPoIB (IP over InfiniBand) presents the InfiniBand fabric to the operating system as if it were an ordinary IP-capable network interface, letting standard IP-based tools and applications work over InfiniBand without modification. This is genuinely useful for compatibility — but IPoIB traffic still goes through the traditional IP stack path described in Section 10.2, not the RDMA path. IPoIB gives you connectivity over InfiniBand hardware; it does not give you RDMA's performance characteristics. Confusing the two — assuming that because traffic is moving over InfiniBand hardware it must be getting RDMA's benefits — is one of the more common misunderstandings in this area.

10.6 RoCE Overview

RoCE (RDMA over Converged Ethernet) carries RDMA semantics over an Ethernet fabric instead of an InfiniBand one. The application-facing RDMA verbs interface is essentially the same as InfiniBand's; what differs is the underlying transport and fabric. This lets sites use RDMA-capable communication on Ethernet infrastructure, at the cost of Ethernet needing to behave in ways it doesn't by default — covered in Section 10.9.

10.7 RoCEv1 vs. RoCEv2

RoCEv1 operates directly at the Ethernet link layer and is not routable beyond a single Ethernet broadcast domain. RoCEv2 encapsulates RDMA traffic over UDP/IP instead, making it routable across a normal IP network — a meaningful practical difference. RoCEv1 is largely legacy at this point; RoCEv2 is the version relevant to virtually any current deployment, and this book's later references to RoCE assume v2 unless stated otherwise.

10.8 Ethernet-Based RDMA in Context

RoCE and native InfiniBand RDMA solve the same fundamental problem — low-latency, low-CPU-overhead data movement — using different underlying fabrics. Neither is a strictly superior general solution; the right choice depends on existing infrastructure, operational expertise, and specific workload requirements, not on RoCE being "InfiniBand's Ethernet knockoff" or InfiniBand being unconditionally faster. This book presents both without declaring a winner.

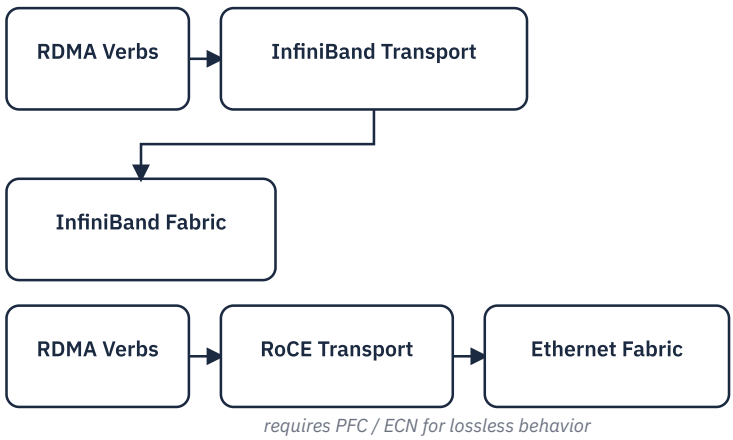


Figure 10.2 — RoCE vs. Native InfiniBand Comparison. Both carry RDMA semantics; the fabric underneath — and what that fabric requires to behave losslessly — differs. *Components: two side-by-side stacks — RDMA Verbs → InfiniBand Transport → InfiniBand Fabric; RDMA Verbs → RoCE Transport → Ethernet Fabric (with a PFC/ECN annotation).*

10.9 Lossless/Congestion Considerations: PFC and ECN

RDMA protocols generally assume a fabric that doesn't casually drop packets under load — InfiniBand's own link-level flow control provides this natively. Ethernet, by contrast, is designed to tolerate and recover from packet loss at higher layers, which works fine for most traffic but conflicts with RDMA's expectations. RoCE deployments address this by configuring Ethernet to behave losslessly for RDMA traffic specifically, primarily through two mechanisms: Priority Flow Control (PFC), which lets a switch pause a specific traffic class rather than drop packets from it under congestion, and Explicit Congestion Notification (ECN), which signals congestion before it becomes severe enough to require pausing at all.

This remains a conceptual introduction — PFC and ECN's exact configuration is switch-vendor- and platform-specific, and getting them wrong is a well-known source of RoCE deployment problems. This book does not provide a PFC/ECN configuration tutorial; the point here is understanding why RoCE has this operational requirement at all, which native InfiniBand deployments don't face in the same way.

10.10 RDMA Transport Concepts

RDMA verbs support a few transport types, most notably Reliable Connected (RC) — a connection-oriented, guaranteed-delivery mode most applications use — and Unreliable Datagram (UD), a connectionless mode without delivery guarantees, used where its lower overhead suits the application better. This book names these to support recognition, not to teach RDMA application programming.

10.11 Application Communication Implications

An application written against RDMA verbs directly gets the full latency/overhead benefit, provided the underlying fabric (native IB or properly configured RoCE) actually delivers it. An application relying on IPoIB, or on ordinary sockets over any fabric, doesn't get RDMA's benefit regardless of what hardware is underneath — the application has to actually use the RDMA path for the hardware's capability to matter. This is a recurring theme this book returns to in Part 9 (NCCL), where transport selection becomes directly relevant to distributed AI training performance.

10.12 Choosing Between Them: A Decision Framework, Not a Winner

Table 10.1 — Decision Framework

Situation	Likely Fit	Why
Need RDMA performance, InfiniBand infrastructure already in place	Native IB RDMA	Direct hardware support, no additional lossless-Ethernet configuration needed
Need IP-compatible connectivity over an existing InfiniBand fabric, performance not the priority	IPoIB	Compatibility layer, simplest to use for non-performance-critical traffic
Need RDMA performance, Ethernet infrastructure and expertise already in place	RoCE (v2)	RDMA benefit without a separate fabric, at the cost of PFC/ECN configuration discipline

Traffic doesn't need RDMA at all	Ordinary IP networking	No reason to take on RDMA's additional complexity

This table is a starting framework, not an exhaustive decision procedure — real deployments weigh additional factors (existing skills, vendor relationships, cost) this book doesn't attempt to generalize about.

10.13 Misconceptions and Illustrative Scenarios

RDMA and InfiniBand are not the same thing — InfiniBand is one way to get RDMA; RoCE is another. RoCE is not simply "InfiniBand over Ethernet" in a casual sense — it carries RDMA semantics over Ethernet, but only works well when the underlying Ethernet fabric is configured to behave losslessly. And IPoIB, as Section 10.5 already established, does not give an application RDMA's performance benefit merely by virtue of running over InfiniBand hardware.

[Illustrative] An application expected to benefit from InfiniBand's low latency shows performance closer to ordinary networking than expected. Investigation finds it's communicating over IPoIB rather than through the RDMA verbs interface — the hardware supports RDMA, but this particular application was never using that path.

[Illustrative] A newly deployed RoCE fabric shows intermittent packet loss under sustained load that a native InfiniBand deployment at the same site doesn't exhibit. Conceptually, this points toward the Ethernet fabric's lossless behavior (PFC/ECN) not being fully configured as RDMA traffic requires — not a hardware fault, and not evidence that RoCE is inherently less reliable than InfiniBand.

CHAPTER 11

INFINIBAND / RDMA

InfiniBand Administration

11.1 Where This Fits

Chapter 9 named the tools this chapter now actually teaches. The distinction to hold onto through this chapter and the next: this chapter is about using these tools correctly, to confirm things are configured and working as expected. Chapter 12 is about using the same tools when something isn't — reading their output for evidence of a fault rather than confirmation of health. Not every environment has every tool covered below installed by default; exact package availability depends on the distribution and driver stack (in-box kernel drivers vs. a vendor-supplied stack), which is worth confirming on a given site rather than assumed.

11.2 Discovering Adapters

`ibv_devinfo` (read-only) lists the InfiniBand HCAs present on a node and reports detail about each — device name, firmware version, and port information. `ibstat` (read-only) gives a more concise, port-state-focused view of the same adapters. Running either after a node is provisioned is a reasonable first check that the HCA is actually recognized before assuming anything about fabric connectivity specifically.

11.3 Checking Ports and Link State

`ibstat` and `ibstatus` (both read-only) report each port's link state (Chapter 9's Down/Initializing/Armed/Active progression) and negotiated rate. A healthy port shows Active with the rate expected for that hardware generation; anything else — a state stuck below Active, or a rate lower than expected — is worth investigating before assuming the fabric itself, rather than this one port, is the issue.

11.4 Fabric Discovery

`ibnetdiscover` (read-only) walks the fabric from the perspective of the node it's run on, reporting the topology it can see — switches, links, and the nodes reachable through them. `ibswitches` (read-only) lists the switches discovered on the fabric specifically. Both depend on the subnet manager being active and the fabric being in a state where discovery can complete; a discovery that returns an unexpectedly small or incomplete picture of the fabric is itself a symptom worth noting, not just a lack of information.

11.5 Subnet Manager Concepts and Administration

The subnet manager (Chapter 9) is commonly run as OpenSM, an open-source implementation, though vendor-supplied alternatives exist. A production fabric typically runs the SM with a standby instance ready to take over if the active one fails — SM failover. Confirming this failover actually works, rather than assuming it does because it's configured to, is a legitimate administrative task:

verifying which instance is currently master, and, in a controlled maintenance window, testing that a standby actually takes over cleanly.

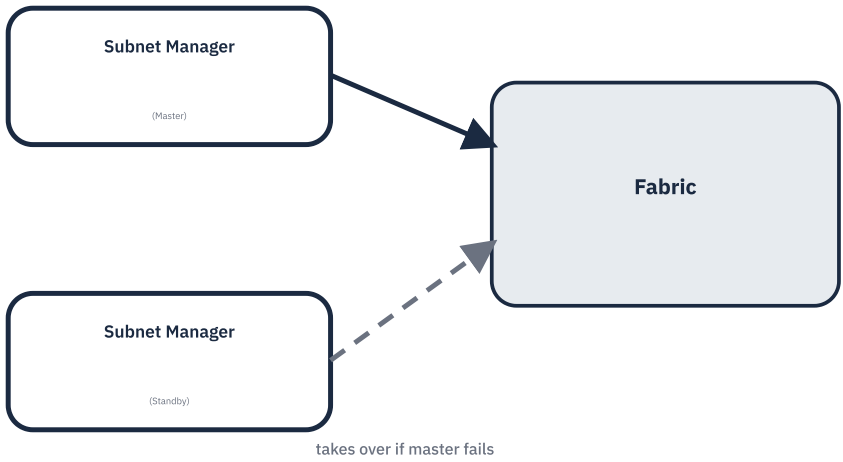


Figure 11.1 — Subnet Manager Failover Topology. A properly configured fabric has a standby subnet manager ready to take over — but this must be verified, not assumed. *Components: two SM instances (Master, Standby), the fabric they manage, a dashed "takes over if master fails" connector.*

Administratively, changes to the SM's configuration — how it assigns LIDs, whether partitioning is enabled — are configuration-file-driven and take effect on the SM's next relevant action (a fabric sweep, a restart), which makes them state-changing but not necessarily instantly disruptive, depending on what's changed.

11.6 LIDs, GUIDs, and PKeys in Practice

`ibstat` and related tools report a port's currently assigned LID directly. GUIDs, being hardware-fixed, show up the same way regardless of any SM reconfiguration — useful as a stable identifier when a LID has changed across a fabric event. Where PKeys are in use, the SM's own configuration is the source of truth for which

ports belong to which partitions; inspecting a specific port's PKey membership is part of confirming a partitioning scheme is actually applied as intended, not just configured.

11.7 Link Diagnostics

`iblinkinfo` (read-only) reports link-level detail across the fabric — state and rate for each link it can discover, in one consolidated view, which is often more convenient than checking `ibstat` port by port across many nodes. `ibdev2netdev` (read-only) maps an InfiniBand device to its corresponding network device name where relevant (useful specifically when working with IPoIB interfaces, Chapter 10) — helpful for confirming you're looking at the right interface when a system has both InfiniBand and Ethernet devices present.

11.8 Fabric Health Overview

`perfquery` (read-only in its default query mode) reports per-port performance and error counters — the evidence Chapter 12 leans on heavily for fault isolation, but also useful administratively as a baseline: capturing a healthy fabric's counter values gives you something to compare against later. `ibdiagnet` (read-only in its standard scan mode) runs a broader fabric-wide diagnostic sweep, checking topology consistency and aggregating counter data across many ports at once — useful both as a periodic health snapshot and, in Chapter 12, as a troubleshooting tool.

11.9 IPoIB Administration Notes

Where IPoIB is in use, the resulting interface (commonly named something like `ib0`) is administered with the same ordinary Linux networking tools covered in Chapter 3 — `ip a`, for instance, shows its address and state exactly as it would for any other

interface. `ibdev2netdev` (Section 11.7) is the tool specific to confirming which InfiniBand device a given IPoIB interface corresponds to.

11.10 UFM Relationship

Everything in this chapter can be done manually, tool by tool, on any given node. NVIDIA UFM (Chapter 9's forward reference, covered in full in Part 8) provides a consolidated, fabric-wide view of much of this same information — port states, error counters, topology — without needing to log into individual nodes or run each tool separately. This chapter's manual tools remain useful and necessary regardless of whether UFM is deployed at a site; UFM is a convenience and a monitoring layer, not a replacement for understanding what the underlying tools report.

11.11 Safe vs. Disruptive Operations

Table 11.1 — Command Safety Classification

Category	Commands
● SAFE / READ-ONLY	`ibstat`, `ibstatus`, `ibv_devinfo`, `ibdev2netdev`, `iblinkinfo`, `ibnetdiscover`, `ibswitches`, `perfquery` (query mode), `ibdiagnet` (standard scan mode)
■ CHANGES STATE	Subnet manager configuration changes, PKey table updates
▲ POTENTIALLY DISRUPTIVE	Forcing an SM failover outside a planned window, disabling/re-enabling a port on a live fabric, `perfquery` counter-reset operations

11.12 Operational Checklists

Table 11.2 — Operational Checklists

Task	Steps
New adapter onboarding	1. `ibv_devinfo` — confirm the HCA is detected. 2. `ibstat` — confirm port state reaches Active at the expected rate. 3. `ibnetdiscover` — confirm the node appears in the fabric topology as expected.
Fabric health snapshot	1. `perfquery` across relevant ports — capture current counter values as a baseline. 2. `ibdiagnet` — run a standard scan and review for topology or counter anomalies. 3. Record results for future comparison.

11.13 Misconceptions and a Scenario

Not every environment has every tool covered in this chapter installed — package availability depends on the distribution and driver stack, and this is worth confirming rather than assuming. `ibdiagnet` is not solely a troubleshooting tool — used proactively, it's also how you'd capture the fabric-health baseline Section 11.12 recommends. And a fabric having a standby subnet manager configured doesn't guarantee failover actually works cleanly when needed — that's worth testing, not assumed.

[Illustrative] During a planned maintenance window, the active subnet manager instance is taken down deliberately to verify failover. The standby instance takes over fabric management within the expected window, and `sminfo` on each node confirms the new master is recognized fabric-wide — but the exercise also turns up a switch whose configured SM priority would have made it a poor failover candidate had the standby been unavailable, prompting a priority correction before the next maintenance cycle.

CHAPTER 12

INFINIBAND / RDMA

InfiniBand Troubleshooting

12.1 The Layered Model

An InfiniBand symptom can originate at any of eight layers: Host, HCA, Port, Cable/Optics, Switch, Fabric (topology-wide), Subnet Manager, or Application. The layer where a symptom is first noticed is not always the layer where its root cause actually lives — an application-level communication failure can trace all the way back to a single bad cable, and a fabric-wide performance complaint can trace to one congested link rather than anything wrong with the fabric as a whole. This chapter's method is to start at the layer where the symptom appeared, then be willing to walk backward through the chain until evidence, not assumption, identifies where the fault actually is.

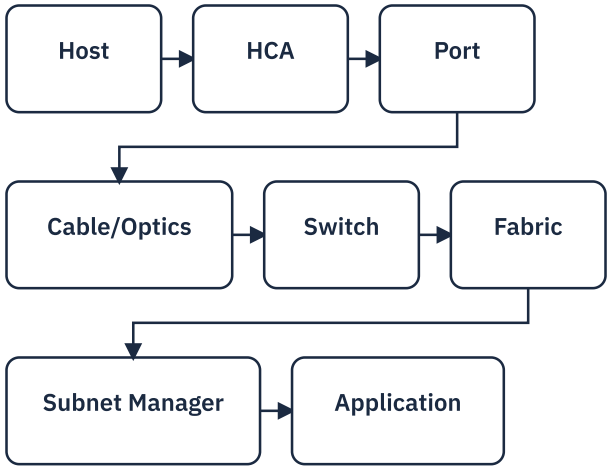


Figure 12.1 — InfiniBand Troubleshooting Flowchart. Start at the layer where the symptom appears, but be ready to walk backward — the root cause is often upstream of where the symptom was first noticed. *Components: eight labeled stages in sequence — Host, HCA, Port, Cable/Optics, Switch, Fabric, Subnet Manager, Application — each with one example symptom. Connections: a linear chain with explicit callouts that a symptom observed at "Application" can require walking backward through every prior stage.*

12.2 The Command Pattern

Every command section below follows the same structure: what the command shows, what healthy output looks like, what failure looks like, and what to check next. This is deliberate — it's the same discipline as the ten-section scenario format, applied one command at a time rather than one full incident at a time.

12.3 Host and HCA-Level Symptoms

`ibv_devinfo` / **`ibstat`** — *Shows:* whether the HCA is detected and its basic state. *Healthy:* the expected device appears, with port state reaching Active. *Failure:* the device is absent entirely, or present but with every port stuck below Active. *Check next:* if the device is entirely absent, this is a host/HCA-level problem (driver, seating, firmware) rather than anything fabric-side — move to Section 12.4 only once the HCA itself is confirmed present.

A driver mismatch after a kernel or driver-stack update is a common host-level cause for an HCA that was previously working to suddenly stop being detected — worth checking before assuming a hardware fault.

12.4 Port-Level Symptoms

`ibstat` / **`ibstatus`** — *Shows:* per-port link state and negotiated rate. *Healthy:* Active state, at the rate expected for the hardware. *Failure:* a state stuck at Down, Initializing, or Armed; or Active at a lower rate than expected. *Check next:* a port stuck below Active with no obvious host-side cause points toward Section 12.5 (cable/optics) or Section 12.6 (the switch side of the same link).

`perfquery` — *Shows:* per-port error and performance counters. *Healthy:* low or stable counters over time. *Failure:* rapidly incrementing error counters (CRC errors, symbol errors) even if the link state itself shows Active. *Check next:* rising error counters on an otherwise-Active port point toward Section 12.5 before anything higher up the stack.

12.5 Cable/Optics Problems

Physical-layer issues — a damaged cable, a dirty or failing optical transceiver, a poor seating — commonly present as either a port that won't reach Active at all, or as one that reaches Active but shows elevated CRC/symbol error counters despite appearing to work. A link that flaps (repeatedly transitions between Active and a lower state) without any configuration change is a strong physical-layer indicator, though not the only possible cause — Section 12.6's switch-side possibilities are worth ruling out too before replacing hardware.

12.6 Switch-Level Symptoms

A switch-side fault can look identical, from the host's perspective, to a cable problem — a port that won't train to Active, or elevated errors on an otherwise-normal-looking link. `iblinkinfo`, run to inspect the specific switch port a suspect link connects to, is the fastest way to check whether the problem is consistent from both ends of the link (pointing toward the cable/optics) or appears only on one side (pointing toward that side's port or the switch itself).

12.7 Fabric-Wide Symptoms

`ibnetdiscover` (Chapter 11) run during a suspected fabric-wide issue can reveal topology anomalies — nodes or switches missing from the expected topology, or unexpected asymmetries — that point toward a broader problem rather than a single link. A node reported as unreachable fabric-wide, when its own HCA and port both check out fine (Sections 12.3–12.4), points toward something between it and the rest of the fabric: a specific switch, a specific set of links, or a topology inconsistency `ibnetdiscover` can help surface.

12.8 Subnet Manager Problems

A fabric with a non-functioning or unreachable subnet manager (Chapters 9 and 11) generally can't complete address assignment or respond to topology changes — symptoms can range from new nodes never coming up on the fabric at all, to an existing fabric failing to adapt after a link or node change. Checking which SM instance is currently master, and whether it's actually responding, is the first step; a planned failover test (Chapter 11) that doesn't complete cleanly is itself evidence of an SM-layer problem, distinct from anything wrong with the fabric's physical layer.

12.9 Application-Level Symptoms

An application reporting a communication failure or timeout doesn't tell you, by itself, which lower layer is actually responsible — that's the point of walking the model backward. Confirming the HCA, port, and basic fabric reachability (Sections 12.3–12.7) before assuming the application itself is at fault avoids the common mistake of debugging application code for a problem that's actually a bad cable three layers down.

12.10 Performance Degradation

Not every InfiniBand problem is a "down" problem. A fabric or link that's fully Active, with no obvious errors, can still perform below expectation — often traced to congestion on a specific link that many flows happen to cross, rather than any single point being unhealthy in the binary up/down sense. `perfquery`'s counters, read over time rather than as a single snapshot, are the primary evidence here: a link with disproportionately high utilization relative to its neighbors is a reasonable place to start, before assuming a broader fabric-wide slowdown.

12.11 Common Mistakes

Assuming a flapping link is always a cable problem, without checking whether the same flapping is visible from the switch side too (Section 12.6). Treating `ibdiagnet`'s error output as necessarily current, without checking whether the counters it's reporting are historical (accumulated since the last reset) rather than reflecting an active, ongoing problem. And assuming a configured SM failover means zero-impact failover in practice — there can be a brief window during a real failover event, and assuming otherwise without having tested it is exactly the gap Chapter 11's operational checklist is meant to close.

12.12 Worked Scenarios

Reused from Chapter 11's Table 11.1, extended:

Table 12.1 — Command Safety, This Chapter

Category	Commands
● SAFE / READ-ONLY	All read-mode uses from Chapter 11 (`ibstat`, `ibstatus`, `ibv_devinfo`, `iblinkinfo`, `ibnetdiscover`, `ibswitches`, `perfquery` query mode, `ibdiagnet` standard scan)
■ CHANGES STATE	`ibportstate` used to query and change port administrative state (its disable/enable action applies to switch ports, not HCA ports)
▲ POTENTIALLY DISRUPTIVE	`ibportstate` disable/enable on a live, in-service switch port; any action risking interruption of fabric traffic while diagnosing

[Illustrative]

Scenario 1 — Link Flap Isolated to a Cable vs. a Port

SYMPTOM	A link repeatedly transitions between Active and Down without any configuration change.
INITIAL CHECKS	Confirm the flap is real (not a monitoring artifact) via `ibstat` observed over a short period.
COMMANDS	`ibstat`, `iblinkinfo` (checked from both ends of the link).
OUTPUT INTERPRETATION	State transitions correlate on both the HCA-side and switch-side view of the same link.
POSSIBLE CAUSES	A damaged or marginal cable/optic; a failing port on either end; a seating issue.
FAULT ISOLATION	Swapping the cable (if feasible) while keeping both endpoints fixed isolates whether the fault travels with the cable or stays with a specific port.
ROOT CAUSE	**The flapping continues after the cable swap, indicating the fault travels with a port rather than the original cable — a marginal transceiver or port-side connection rather than a bad cable.**
RESOLUTION	Replace or reseat the implicated port's transceiver/optic; escalate to hardware replacement (Chapter 6) if the fault persists after that.
VALIDATION	Confirm the link holds Active state, error-free, over a sustained observation period after remediation.

PREVENTION	Add a cable-swap step to the standard link-flap triage checklist so future occurrences are isolated to cable vs. port in one pass rather than by trial and error.

[Illustrative]

Scenario 2 — HCA Not Detected After a Reboot

SYMPTOM	A node's HCA, previously working, is no longer detected after a reboot.
INITIAL CHECKS	`ibv_devinfo` and `ibstat` to confirm the device is genuinely absent, not just showing an unusual state.
COMMANDS	`ibv_devinfo`, `ibstat`, `dmesg` (Chapter 4, for driver-load messages).
OUTPUT INTERPRETATION	No device listed at all; `dmesg` may show a driver load failure or mismatch.
POSSIBLE CAUSES	A driver/firmware mismatch introduced by a recent update; a physical seating issue; a genuine hardware fault.
FAULT ISOLATION	Checking `dmesg` for driver-load errors distinguishes a software-side cause from a physical/hardware one.
ROOT CAUSE	`dmesg` shows a driver load failure referencing a firmware version the currently-loaded driver doesn't support — the reboot exposed a driver/firmware mismatch that had been introduced by an earlier, incomplete update.

RESOLUTION	Align the driver and firmware versions to a supported combination (updating whichever side is behind), then reload the driver.
VALIDATION	Confirm the HCA is detected and reaches Active state post-remediation.
PREVENTION	Treat driver and firmware versions as a single unit to update together, and verify the pairing against the vendor's supported-combinations documentation before rolling an update out fleet-wide (Chapter 6).

[Illustrative]

Scenario 3 — Port Shows "Active" but Has High Error Counters

SYMPTOM	A link appears healthy by state (Active), but application performance across it is inconsistent.
INITIAL CHECKS	Check `perfquery` counters, not just link state.
COMMANDS	`perfquery`.
OUTPUT INTERPRETATION	CRC or symbol error counts are elevated and increasing over the observation period, despite the link showing Active throughout.
POSSIBLE CAUSES	A marginal cable/optic that trains successfully but doesn't transmit cleanly; electrical interference; a connector that's seated but not fully secure.

FAULT ISOLATION	Comparing this link's error rate against comparable links on the same fabric — an outlier points toward this specific link's physical layer.
ROOT CAUSE	A marginal physical-layer connection that establishes a link but doesn't sustain clean transmission under load.
RESOLUTION	Reseat or replace the cable/optic.
VALIDATION	Confirm error counters return to a baseline consistent with comparable healthy links.
PREVENTION	Include error-counter checks, not just link-state checks, in routine fabric-health snapshots (Chapter 11).

[Illustrative]

Scenario 4 — Node Unreachable on the Fabric

SYMPTOM	A node cannot communicate with any other node on the fabric, though it was working previously.
INITIAL CHECKS	Confirm the node's own HCA and port state first (Sections 12.3–12.4) before assuming a broader fabric issue.
COMMANDS	`ibstat` (local), `ibnetdiscover` (from another working node, to see whether this node appears in the topology at all).
OUTPUT INTERPRETATION	The node's own HCA/port show Active locally, but it's absent from `ibnetdiscover`'s view from elsewhere on the fabric.

POSSIBLE CAUSES	A fault on the specific switch port or link this node connects through, upstream of the node's own hardware; a subnet manager issue preventing this node from being properly addressed.
FAULT ISOLATION	Checking the specific switch port this node connects to narrows the fault to the switch/link side rather than the node's own HCA.
ROOT CAUSE	A fault on the upstream switch port, invisible from the node's own local diagnostics.
RESOLUTION	Address the switch-side fault (port reset, cable check at the switch end, or escalation if the switch hardware itself is implicated).
VALIDATION	Confirm the node reappears in `ibnetdiscover`'s topology from another node's perspective, and that basic communication succeeds.
PREVENTION	When a node's own diagnostics look clean but connectivity fails, check the topology from another node's perspective earlier in the process, rather than re-checking the same node repeatedly.

[Illustrative]

Scenario 5 — Subnet Manager Failover Not Completing as Expected

SYMPTOM	During a planned SM failover test, the standby does not take over cleanly within the expected time.

INITIAL CHECKS	Confirm which SM instance is currently reporting as master, and whether the standby is actually reachable.
COMMANDS	SM status queries (implementation-specific); `ibnetdiscover` to observe fabric-wide impact during the test.
OUTPUT INTERPRETATION	The standby SM is reachable and running, but its configured priority is lower than expected, so it does not assume mastership within the test's expected window.
POSSIBLE CAUSES	Standby SM misconfiguration; a network path issue between the two SM instances; a version or configuration mismatch between master and standby.
FAULT ISOLATION	Comparing the standby's configured priority and version against the (working) master's configuration isolates the gap to a configuration mismatch rather than a reachability or software fault.
ROOT CAUSE	**The standby SM's priority was left at a default value during setup rather than being explicitly configured to take over promptly, so it technically could fail over but not within the expected time.**
RESOLUTION	Correct the standby's priority configuration to match the intended failover behavior, and confirm both instances agree on the fabric's expected master/standby roles.
VALIDATION	Re-test failover in a controlled window and confirm it completes within an acceptable time.

PREVENTION	Treat SM failover as something tested periodically, not configured once and assumed — directly reinforcing Chapter 11's operational-checklist guidance.

[Illustrative]

Scenario 6 — Fabric-Wide Performance Degradation Traced to One Congested Link

SYMPTOM	Multiple applications across the fabric report inconsistent performance, with no single node or job clearly at fault.
INITIAL CHECKS	Rule out any individual node-level cause first; the breadth of the symptom suggests something shared.
COMMANDS	`perfquery` across multiple links; `ibdiagnet` for a fabric-wide sweep.
OUTPUT INTERPRETATION	One specific link shows disproportionately high utilization relative to comparable links elsewhere in the topology.
POSSIBLE CAUSES	A topology design that routes more traffic through this link than others (an imbalance); a failed or degraded redundant path forcing more traffic onto this one.
FAULT ISOLATION	Checking whether an expected redundant path is actually available and healthy narrows this to either a design imbalance or a masked prior failure.
ROOT CAUSE	A previously degraded redundant path had silently forced more traffic than intended onto the remaining healthy link, which was not itself broken but was oversubscribed.

RESOLUTION	Restore the degraded redundant path, redistributing load back across both.
VALIDATION	Confirm the previously congested link's utilization returns to an expected level, and reported application performance normalizes.
PREVENTION	Monitor for asymmetric link utilization as a leading indicator of a masked redundant-path failure, rather than only alerting on links that are fully down.

PART 6

NVIDIA GPU Infrastructure

Chapter 13, Chapter 14, Chapter 15

CHAPTER 13

GPU

NVIDIA GPU Infrastructure

13.1 Where This Fits

Chapter 6 introduced the GPU as a distinct server-hardware category; Chapter 2 placed it as a distinct node role. This chapter is where GPU infrastructure gets its own depth: the driver/CUDA stack, visibility commands, and health signals an infrastructure engineer checks before assuming anything is wrong. Chapter 14 is where you diagnose what's actually broken; this chapter is about recognizing what healthy looks like first.

13.2 The GPU as Infrastructure

A GPU node pairs one or more accelerators with general-purpose CPUs, connected over PCIe (Chapter 6) and, on multi-GPU nodes, potentially over NVLink (Chapter 15). The CPU still handles process scheduling, I/O, and orchestration; the GPU executes the parallel, accelerator-bound portion of a workload. Neither replaces the other — a GPU without a CPU to launch and feed it work isn't useful, and a CPU running GPU-bound work without the GPU is simply slow. Infrastructure engineers care about this relationship because failures can originate on either side and present similarly: a job that "hangs" might be a GPU problem, a CPU-side scheduling problem, or neither.

13.3 The Driver/CUDA/Application Stack

Three layers sit between an application and the GPU hardware, and each can fail independently of the others. The **NVIDIA driver** is the kernel-level component that lets the operating system recognize and communicate with the GPU at all — without it, the GPU is invisible to everything above it. The **CUDA runtime** is the layer applications actually link against to issue GPU work; it depends on a compatible driver being present but is a separate, versioned component. The **application** itself sits on top, calling into CUDA. A working driver doesn't guarantee a CUDA application will run (a runtime/driver version mismatch is possible), and a working CUDA application yesterday doesn't guarantee it still works after a driver update.

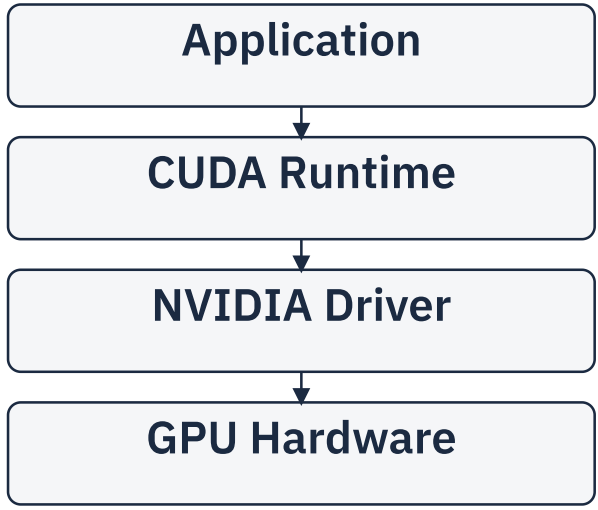


Figure 13.1 — Driver/CUDA/Application Stack. Each layer can fail independently — a working application layer doesn't confirm the driver or hardware are healthy. *Components: GPU hardware → NVIDIA driver → CUDA runtime → application, shown as a vertical stack.*

13.4 GPU Visibility

Confirming a GPU is present and recognized is the first check, before anything else. `nvidia-smi` (read-only) is the primary tool: it lists every GPU the driver recognizes, along with driver version, utilization, memory usage, temperature, and power draw at a glance. If a GPU that should be present doesn't appear in `nvidia-smi`'s output at all, the problem is at the driver or hardware level, not the application (Chapter 14 covers this symptom in depth).

Beneath `nvidia-smi`, three general Linux tools help confirm what the driver and hardware layers actually see: `lspci` (read-only) lists PCI devices, including the GPU itself, independent of whether the NVIDIA driver has claimed it — useful for confirming the hardware is detected by the system at all, even if the driver has a problem. `lsmod` (read-only) lists loaded kernel modules, confirming whether the NVIDIA driver module is actually loaded. `modinfo nvidia` (read-only) reports detail about the driver module itself, including its version.

What healthy looks like: `nvidia-smi` lists every expected GPU with a plausible driver version and no error state; `lspci` shows the expected GPU(s) as PCI devices; `lsmod` shows the NVIDIA driver module loaded. **What to check when it isn't:** if `lspci` doesn't show the GPU, that's a hardware/PCIe-level absence (Chapter 6/14); if `lspci` shows it but `lsmod` doesn't show the driver loaded, that's a driver-loading problem, not a hardware one.

13.5 PCIe and GPU Memory

GPUs connect to the rest of the node over PCIe, and the negotiated PCIe generation and link width affect how fast data moves between host memory and the GPU. `nvidia-smi -q` (read-only) reports the GPU's current PCIe link state, which is worth comparing against the hardware's rated capability — a link trained at a lower width or

generation than expected is a real, if easily overlooked, performance-limiting condition (Chapter 14 covers this as a fault category).

GPU memory (often called device memory) is physically separate from the system's main memory and is not automatically shared with it — moving data between them costs time, which is one reason GPU-resident data layout matters for performance, though this book doesn't go deeper into that than infrastructure-level awareness requires.

13.6 GPU Topology, Introductory

On a multi-GPU node, not every GPU pair connects to every other the same way — some may share a fast NVLink connection, others may only reach each other over PCIe. `nvidia-smi topo -m` (read-only) reports this as a matrix. This chapter introduces the command at recognition level only; Chapter 15 covers reading and acting on its output in full.

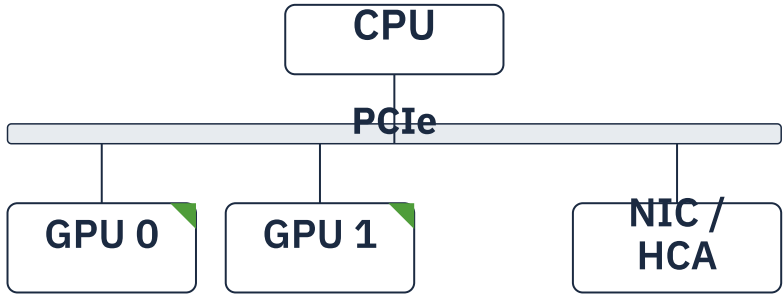


Figure 13.2 — GPU Node Component Overview. A GPU node's PCIe topology determines how CPU, GPU, and network interface reach each other — Chapter 15 covers this in depth. *Components: CPU, PCIe, one or more GPUs, NIC/HCA — a GPU-node-specific refinement of Chapter 6's server block diagram.*

13.7 GPU Health and DCGM

`nvidia-smi` gives a quick snapshot; NVIDIA's Data Center GPU Manager (DCGM) is built for deeper, more systematic health checking and monitoring across a fleet of GPUs, rather than one node at a time. `dcgmi diag` runs a diagnostic pass at a chosen level of thoroughness — lighter levels are quick sanity checks; heavier levels run more demanding tests and can themselves stress the GPU, which is why they're not something to run casually against a GPU with active work on it. DCGM is also the foundation most fleet-wide GPU monitoring (Chapter 21) is built on, rather than a replacement for `nvidia-smi`'s quick, single-node view.

13.8 GPU Resource Allocation and MIG

In a shared HPC cluster, GPUs are allocated to jobs by the scheduler (Slurm, Chapter 19 covers this in depth) — this chapter only needs the concept: a job requests some number of GPUs, and the scheduler matches that request against what's physically available. Multi-Instance GPU (MIG) is a related but distinct concept: on GPUs that support it, MIG partitions a single physical GPU into multiple, fully isolated instances, each schedulable independently. MIG is a resource-partitioning tool, not a performance feature — it doesn't make a GPU faster, it lets one physical GPU serve multiple smaller workloads with hardware-level isolation between them.

13.9 Common Infrastructure Failure Points

A few categories are worth naming here, without diagnosing them — that's Chapter 14's job: driver/CUDA version mismatches, thermal throttling under sustained load, PCIe link degradation,

and ECC memory errors (correctable and uncorrectable). Recognizing these categories exist is enough at this stage; the next chapter is where you learn to tell them apart using evidence.

13.10 Misconceptions and an Illustrative Scenario

"CUDA" isn't one thing — the driver, the CUDA runtime, and the CUDA toolkit (used for building applications) are related but distinct, and version compatibility between them is a real constraint, not a formality. A working `nvidia-smi` output doesn't guarantee a CUDA application will run — that only confirms the driver and hardware layers, not the runtime layer above them. And MIG is not "a faster GPU" — it's the same physical GPU, partitioned.

[Illustrative] A newly provisioned GPU node shows fewer GPUs in `nvidia-smi`'s output than the hardware inventory expects. Before assuming a hardware fault, checking `lspci` first is the right instinct: if all expected GPUs appear there, the hardware is present and the problem is at the driver level — a faster, cheaper thing to fix than a hardware replacement, and worth ruling in or out before escalating.

CHAPTER 14

GPU

GPU Troubleshooting

14.1 The Layered Model

A GPU-related symptom can originate at any of six layers: Application, CUDA, Driver, GPU, PCIe, or Host. As with InfiniBand (Chapter 12), the layer where a symptom is first noticed isn't always where its root cause lives — an application-level CUDA error can trace back to a driver mismatch, a PCIe fault, or genuine GPU hardware degradation. This chapter's method is the same discipline as Chapter 12's: start where the symptom appears, then be willing to walk backward through the chain.

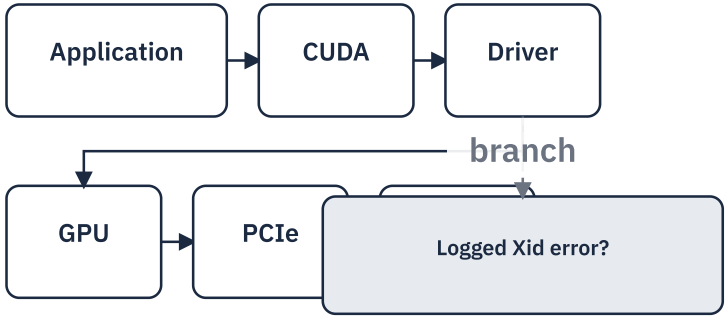


Figure 14.1 — GPU Fault Triage Flow. Start at the layer where the symptom appears; a logged Xid error points toward the GPU/driver layer specifically. *Components: the six-layer chain (Application→CUDA→Driver→GPU→PCIe→Host) with a branch point distinguishing Xid-error symptoms from non-Xid symptoms.*

Every scenario in this chapter uses the full ten-section format established in Chapter 4.

14.2 GPU Not Detected

When `nvidia-smi` (Chapter 13) shows fewer GPUs than expected, or none at all, the fault sits at the Host, PCIe, or Driver layer — never the application, since the application never gets far enough to be the cause. `lspci` (Chapter 13) is the fastest way to separate these: if the GPU appears in `lspci`'s output, the hardware is physically present and detected by the system, and the problem is at the driver layer (not loaded, or failing to load — check `lsmod` and `dmesg`/`journalctl` for load errors, Chapter 4). If the GPU doesn't appear in `lspci` at all, the problem is at the Host/PCIe layer — a seating issue, a PCIe slot problem, or a genuine hardware fault.

14.3 Driver and CUDA Compatibility Problems

CUDA applications depend on a compatible pairing between the installed driver and the CUDA runtime/toolkit version they were built or linked against. After a driver upgrade or rollback, an application that worked yesterday can fail today with a version-mismatch error — `nvidia-smi -q` reporting the current driver version, compared against the application's documented CUDA requirement, is the fastest way to confirm this category before looking anywhere else.

14.4 GPU Xid Errors

An Xid error is a message the NVIDIA driver logs (visible via `dmesg`/`journalctl`, Chapter 4) when it detects a general GPU-related error condition. Xid codes are numbered, and NVIDIA publishes an official reference explaining what each code means —

this book names a small number of well-established, stable codes as examples, and defers to that official reference for the complete, current list rather than reproducing it here. Two worth knowing by name because they're common and unambiguous: **Xid 79** ("GPU has fallen off the bus") indicates the GPU has stopped responding on the bus entirely — typically a hardware- or power-related condition, not something software can recover from without a reset or reseat. **Xid 48** (a double-bit, uncorrectable ECC error) indicates a memory error the GPU's error-correction couldn't fix — a hardware signal, not a software bug. Not every Xid code indicates hardware failure; some reflect application-level or driver-level conditions. The specific code is what determines the category, which is exactly why guessing without checking the official reference is a mistake worth avoiding.

14.5 GPU Memory and ECC-Related Symptoms

GPU memory, like server DIMMs (Chapter 6), uses ECC to detect and, for single-bit errors, correct memory errors automatically. A rising count of correctable ECC errors (visible via `nvidia-smi -q`) is a leading indicator worth acting on, similar in spirit to Chapter 6's DIMM correctable-error scenario — not an emergency, but evidence a component may be degrading. An uncorrectable ECC error is more serious and often accompanies an Xid event.

14.6 PCIe Problems

A GPU can be detected and functional while still running below its capability if its PCIe link trained at a lower generation or width than the hardware supports (Chapter 13). This doesn't show up as an outright failure — it shows up as unexplained underperformance, which is why checking `nvidia-smi -q`'s reported link state is worth doing before assuming a software-level performance problem.

14.7 Thermal and Power Issues

Sustained heavy GPU workloads generate significant heat, and a GPU approaching its thermal limit will throttle its own clock speed to stay within safe operating limits — a self-protective behavior, not a fault by itself, but one worth recognizing rather than mistaking for an application-level performance regression. `nvidia-smi`'s temperature and power-draw fields are the first place to check when a workload's throughput drops without any code or configuration change.

14.8 GPU Reset Problems and MIG Issues

Resetting a GPU (rather than rebooting the whole node) can sometimes recover it from certain fault states, but not all — a GPU stuck in a state where even a reset doesn't succeed usually indicates a deeper hardware or driver issue requiring a full reboot or hardware attention. MIG-related issues (Chapter 13) typically present as a mismatch between what a job expects (a full GPU) and what it actually received (a MIG instance with a subset of the GPU's resources) — worth checking explicitly rather than assuming a performance problem is unrelated to how the GPU was partitioned.

14.9 DCGM Diagnostics

When `nvidia-smi`'s quick view doesn't explain a symptom, `dcgmi diag` at a deeper diagnostic level can surface issues a quick snapshot misses — at the cost of stressing the GPU during the test, which is why it's not run casually against a GPU with active jobs (Chapter 13).

14.10 Distinguishing Hardware, Driver, and Application/CUDA Problems

A useful habit: hardware problems tend to show up as Xid errors, ECC events, or "not detected" symptoms — evidence the driver itself is reporting. Driver/software problems tend to show up as version-mismatch errors or a GPU that's detected but behaves unexpectedly. Application/CUDA problems tend to show up as errors specific to one application while `nvidia-smi` and other tools report the GPU itself as healthy. None of these are foolproof signals alone — they're where to look first, not where to stop looking.

14.11 Worked Scenarios

Table 14.1 — Command Safety, This Chapter

Category	Commands
● SAFE / READ-ONLY	`nvidia-smi`, `nvidia-smi -q`, `lspci`, `lsmod`, `modinfo`, `dcgmi diag` (light levels)
■ CHANGES STATE	Driver module reload
▲ POTENTIALLY DISRUPTIVE	GPU reset on a node with active jobs, `dcgmi diag` at heavier diagnostic levels (can stress the GPU)

[Illustrative]

Scenario 1 — GPU Falls Off the Bus

SYMPTOM	A GPU that was working suddenly disappears from `nvidia-smi`'s output during a running job.
INITIAL CHECKS	Check `dmesg` / `journalctl` for an Xid event around the failure time.
COMMANDS	`nvidia-smi`, `dmesg`, `lspci`.
OUTPUT INTERPRETATION	`dmesg` shows an Xid 79 event ("GPU has fallen off the bus") at the time the job's error began, and `lspci` no longer enumerates the device.
POSSIBLE CAUSES	A power delivery issue; a severe thermal event; a genuine hardware fault.
FAULT ISOLATION	A cold reboot is used to see whether the device re-enumerates at all — distinguishing a device that comes back (favoring power/thermal/seating) from one that doesn't (favoring a hardware fault).
ROOT CAUSE	**The GPU re-enumerates cleanly after a full power cycle, with no recurrence under equivalent load — consistent with a transient power or seating condition rather than a failed component.**
RESOLUTION	Reseat the card if physically accessible, confirm the power connection, and return the node to service; escalate to RMA only if the Xid 79 event recurs (Chapter 6/25).
VALIDATION	Confirm the GPU reappears in `nvidia-smi` and holds stable under a light diagnostic pass.

PREVENTION	Track recurrence across reboots — a GPU that repeatedly falls off the bus is an RMA candidate, not a reseat-and-forget case.

[Illustrative]

Scenario 2 — Xid Error Classified via the Official Reference

SYMPTOM	An application crashes intermittently with no clear application-level cause.
INITIAL CHECKS	Check `dmesg` / `journalctl` for an Xid entry correlated with the crash time.
COMMANDS	`dmesg`, `journalctl`, `nvidia-smi -q`.
OUTPUT INTERPRETATION	An Xid code appears in the log at the relevant timestamp.
POSSIBLE CAUSES	Depends entirely on the specific code — hardware, driver, or application-triggered condition.
FAULT ISOLATION	Look up the specific Xid code against NVIDIA's current official reference rather than assuming its category from memory.
ROOT CAUSE	In this illustrative case, the code corresponds to a driver-level condition rather than a hardware fault.
RESOLUTION	Address the driver-level condition (e.g., a known issue fixed in a later driver version) rather than replacing hardware.
VALIDATION	Confirm the application runs a full cycle without the Xid recurring.

PREVENTION	Build a habit of checking the official Xid reference for any new code seen, rather than guessing based on a previously seen, different code.

[Illustrative]

Scenario 3 — CUDA Version Mismatch After a Driver Rollback

SYMPTOM	An application that worked yesterday fails today with a CUDA-related error.
INITIAL CHECKS	Check whether the driver changed recently.
COMMANDS	`nvidia-smi -q` (driver version), comparing against the application's documented CUDA requirement.
OUTPUT INTERPRETATION	The currently installed driver version no longer meets the minimum the application's CUDA runtime requires.
POSSIBLE CAUSES	A driver rollback (e.g., during incident response elsewhere) without accounting for this application's requirement.
FAULT ISOLATION	Confirmed directly by the version comparison — no further isolation needed.
ROOT CAUSE	**Driver/CUDA version incompatibility introduced by an unrelated change.**
RESOLUTION	Restore a compatible driver version, or rebuild/relink the application against a CUDA version compatible with the current driver.
VALIDATION	Confirm the application runs successfully post-fix.

PREVENTION	Track driver/CUDA compatibility requirements per application, especially before any fleet-wide driver change.

[Illustrative]

Scenario 4 — ECC Error Pattern Informing an RMA Decision

SYMPTOM	`nvidia-smi -q` shows a slowly increasing count of correctable ECC errors on one GPU over several weeks.
INITIAL CHECKS	Confirm the trend is real (increasing over time) rather than a one-time event.
COMMANDS	`nvidia-smi -q`, tracked over time.
OUTPUT INTERPRETATION	The correctable-error count rises week over week rather than staying flat, and the errors concentrate on the same memory region rather than appearing scattered and random.
POSSIBLE CAUSES	A degrading memory cell; a marginal component nearing failure.
FAULT ISOLATION	The consistent, concentrated, worsening trend (rather than isolated one-off events) distinguishes a genuinely degrading component from normal, expected background correctable-error activity.
ROOT CAUSE	A memory cell degrading toward failure — correctable today, but trending toward an eventual uncorrectable (Xid 48) event if left in service.

RESOLUTION	Proactively schedule the GPU for RMA before the trend reaches an uncorrectable error, rather than waiting for a job-terminating failure (Chapter 6/25).
VALIDATION	Confirm the replacement GPU shows a stable, low ECC error count under equivalent load.
PREVENTION	Set a threshold for correctable-error growth rate that triggers a proactive RMA conversation, rather than waiting for an uncorrectable event.

[Illustrative]

Scenario 5 — Low GPU Utilization Despite a Running Job

SYMPTOM	A job is running, but `nvidia-smi` shows near-0% GPU utilization for an extended period.
INITIAL CHECKS	Confirm the job is actually supposed to be GPU-bound at this stage, not just at certain phases.
COMMANDS	`nvidia-smi`, checking process binding/`CUDA_VISIBLE_DEVICES` (Chapter 19).
OUTPUT INTERPRETATION	The job's process is running, but not on the GPU `nvidia-smi` is showing, or not using the GPU at this stage of execution.
POSSIBLE CAUSES	A scheduling/binding misconfiguration (Chapter 19); the application is genuinely in a CPU-bound phase; a real GPU fault preventing work from being dispatched.

FAULT ISOLATION	Confirming which GPU the job's process is actually bound to rules the binding explanation in or out.
ROOT CAUSE	In this illustrative case, the job's process was bound to a different GPU than the one being observed.
RESOLUTION	Correct the binding/allocation rather than concluding the GPU itself is faulty.
VALIDATION	Confirm utilization appears on the correct GPU once binding is corrected.
PREVENTION	Check binding explicitly whenever "0% utilization" is the symptom, before assuming a hardware or application fault.

[Illustrative]

Scenario 6 — PCIe Link Downgraded, Silently Degrading Throughput

SYMPTOM	A GPU-bound workload runs noticeably slower than an equivalent job on an otherwise identical node.
INITIAL CHECKS	Compare `nvidia-smi -q`'s reported PCIe link generation/width between the two nodes.
COMMANDS	`nvidia-smi -q`.
OUTPUT INTERPRETATION	One node's GPU reports a lower PCIe link generation or width than the other, despite identical hardware specifications.
POSSIBLE CAUSES	A seating issue; a PCIe slot or riser problem; a BIOS/firmware setting difference (Chapter 6/25).

FAULT ISOLATION	The comparison against an identical node isolates this to a link-training issue specific to this node, not a workload or software difference.
ROOT CAUSE	**The GPU's PCIe link trained at a reduced width due to a physical seating issue.**
RESOLUTION	Reseat the GPU (with the node powered down) and confirm link training improves.
VALIDATION	Confirm `nvidia-smi -q` reports the expected link generation/width post-reseat, and throughput matches the comparison node.
PREVENTION	Include PCIe link state in new-node and post-maintenance validation checklists (Chapter 25), not just GPU presence.

CHAPTER 15

GPU

GPU-to-GPU Communication / NVLink / NVSwitch

15.1 Where This Fits

Chapter 13 introduced GPU topology at recognition level; Chapter 5 previewed CPU-GPU-NIC affinity before either GPUs or fabric had been covered in depth. This chapter closes both loops: how GPUs on the same node actually reach each other, and why that topology matters for the multi-node communication Chapter 23 (NCCL) builds on. This is not an NCCL troubleshooting chapter — that stays Chapter 24's job.

15.2 PCIe Topology Recap

Every GPU connects to the rest of a node over PCIe (Chapters 6 and 13). On a multi-GPU node, GPUs may sit behind different PCIe switches or root complexes, which means the path between any two given GPUs — and between a GPU and a specific NIC — isn't uniform. Two GPUs "close" to each other in PCIe terms reach each other faster than two GPUs that have to communicate through a longer path, potentially crossing the CPU.

15.3 NVLink

NVLink is a high-bandwidth, point-to-point interconnect connecting GPU pairs directly, bypassing PCIe for that connection. Where present, an NVLink connection between two GPUs offers substantially more bandwidth than a PCIe path between the same two GPUs would — though this book avoids citing a specific bandwidth figure, since exact numbers are generation-specific and change with each new GPU architecture. Not every GPU pair on a node is necessarily NVLink-connected; some multi-GPU nodes connect only a subset of pairs directly, with the rest reachable only via PCIe.

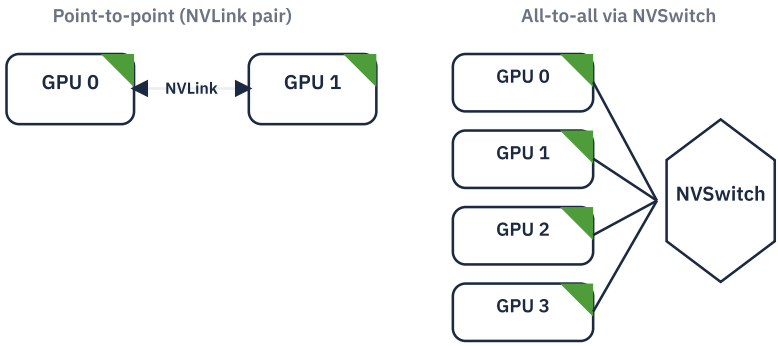


Figure 15.1 — NVLink/NVSwitch Topology. NVLink connects GPU pairs directly; NVSwitch extends this to all-to-all connectivity at higher GPU counts. *Components: multiple GPUs connected point-to-point via NVLink versus connected through an NVSwitch.*

15.4 NVSwitch

As GPU counts per node grow, connecting every GPU to every other directly via point-to-point NVLink becomes impractical. NVSwitch solves this by acting as a switch specifically for NVLink traffic, giving every GPU behind it an equally fast path to every other GPU behind it — an all-to-all topology, rather than a point-to-

point one. A node built around NVSwitch behaves differently, topologically, than one with only direct GPU-to-GPU NVLink pairs: in the NVSwitch case, there's no "close" or "far" GPU pair within the switch's domain, which simplifies the topology-awareness question Chapter 23's NCCL transport selection has to answer.

15.5 GPU Peer-to-Peer Communication

Peer-to-peer (P2P) communication means one GPU can access another's memory directly, without routing through host (CPU) memory as an intermediate step. P2P is available over NVLink and, in some configurations, over PCIe as well — but availability depends on the specific topology and isn't automatic just because two GPUs are technically capable of it. An application or library has to actually use the P2P path for it to matter; simply having NVLink present doesn't guarantee every piece of software takes advantage of it.

15.6 Reading `nvidia-smi topo -m`

This command (introduced at recognition level in Chapter 13) reports a matrix showing, for every pair of GPUs on the node, what kind of connection exists between them — commonly distinguishing an NVLink connection from a PCIe-only path, and further distinguishing PCIe paths by whether they stay within a single PCIe switch, cross to a different one, or cross the CPU/NUMA boundary entirely (tying directly back to Chapter 5's NUMA-locality principle). The matrix also typically reports CPU affinity — which NUMA node each GPU is closest to — which is exactly the information needed to bind a process sensibly (Chapter 5, Chapter 19).

What healthy/expected looks like: the matrix is internally consistent with the node's documented topology — GPUs expected to be NVLink-connected show as such, and CPU affinity matches

the node's known NUMA layout. **What to check when it's surprising:** a GPU pair expected to be NVLink-connected showing only a PCIe path is worth investigating (Chapter 14) before assuming a performance problem elsewhere is unrelated to topology.

15.7 GPU-NIC-CPU Affinity, Closing the Loop

Chapter 5 previewed this with a deliberately coarse diagram, before GPUs or fabric had been introduced. Now, with both covered, the full picture: a network interface (HCA, Chapter 9) is also physically attached to the node's PCIe topology, closer to some NUMA nodes and GPUs than others. A multi-node job that needs to move data from a GPU, out through a NIC, to another node performs better when that GPU and NIC are "close" — sharing a NUMA node and a short PCIe path — than when the data has to cross the full width of the node's topology first.

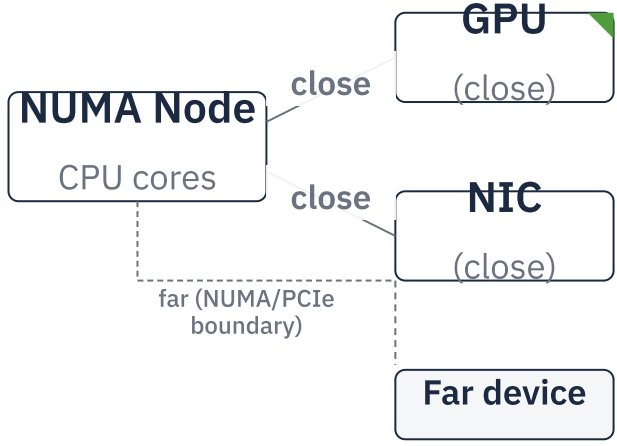


Figure 15.2 — GPU-NIC-CPU Affinity Map. Closing the loop from Chapter 5 — GPU and NIC placement relative to NUMA nodes affects multi-node job performance. *Components: a NUMA node's CPU cores, a GPU, and a NIC, showing "close" (same NUMA node, short PCIe path) versus "far" (crosses NUMA/PCIe boundary) placement — the full version of Chapter 5's Figure 5.2 preview.*

15.8 NCCL Relationship, Briefly

Everything in this chapter — NVLink, NVSwitch, PCIe topology, GPU-NIC affinity — is exactly what NCCL's transport selection (Chapter 23) decides between when moving data across GPUs, whether on one node or many. This chapter provides the vocabulary; Chapter 23 explains how NCCL actually chooses, and Chapter 24 covers diagnosing it when the choice or the resulting performance looks wrong.

15.9 Misconceptions and an Illustrative Scenario

More GPUs on a node doesn't automatically mean faster multi-GPU jobs — it depends entirely on the topology connecting them. NVLink being physically present doesn't mean every application or library automatically uses it — the software has to take the P2P path deliberately. And PCIe-only GPU pairs aren't incapable of communicating — they communicate at lower bandwidth and higher latency, often through the CPU, rather than not at all.

[Illustrative] A multi-GPU job shows a meaningful performance regression compared to expectations, eventually traced not to any hardware fault but to the scheduler placing the job's processes across GPUs and CPUs in a way that ignored the node's actual NVLink/PCIe/NUMA topology — a placement problem (Chapter 19), not a hardware one.

PART 7

Slurm

Chapter 16, Chapter 17, Chapter 18, Chapter 19, Chapter 20

CHAPTER 16

SLURM

Slurm Architecture

16.1 What Slurm Does

Chapter 2 introduced the scheduler as a control-plane concept, logically above the compute/GPU nodes and outside the data path. This chapter makes that concrete for Slurm specifically, in enough depth that Chapters 17 through 20 can build on it without re-explaining it each time.

Slurm is a workload manager — a distinct job from what Chapter 3/4's Linux process-management tools do. Linux manages processes on one node: what's running, who owns it, how much of that one node's CPU and memory it's using. Slurm manages workload across an entire cluster: which job runs on which node or set of nodes, when, and with what resources, across potentially thousands of nodes it never directly executes anything on itself.

Slurm's job, broken down, is four related but distinct functions:

- **Scheduling** — deciding which pending job should run next, and where, given everything else competing for the same capacity (Chapter 18 covers this decision process in depth).
- **Allocation** — actually reserving the specific nodes, CPUs, memory, and other resources (GPUs among them, Chapter 19) for a job once scheduling decides it can run.
- **Launch** — starting the job's processes on the allocated nodes and managing them for the job's duration.

- **Accounting** — recording what happened, for billing, fairshare calculation (Chapter 18), and historical analysis (Chapter 17's `sacct`).

These four functions map onto a chain of relationships: a **user** submits a **job**; a job requests **resources**; **partitions** group **nodes** into policy boundaries a job's request is evaluated against; once scheduled, a job receives an **allocation** — the specific resources it was granted. None of scheduling, allocation, launch, or accounting is optional or interchangeable with another — a job can be correctly scheduled and allocated but fail to launch, and each of those is a different failure category (Chapter 20 diagnoses this distinction directly).

16.2 Core Slurm Components

`slurmctld` (the controller) is Slurm's central decision-maker. It holds the cluster's live state — which nodes exist, what they're doing, what's queued — and performs the scheduling and allocation functions from Section 16.1. It typically runs on a management node (Chapter 2).

`slurmd` (the compute node daemon) runs on every compute and GPU node capable of accepting jobs. It receives launch instructions from the controller, starts and supervises the job's processes on that node, and reports the node's state back. `slurmd` performs the launch function — it doesn't decide anything, it executes what `slurmctld` tells it to.

`slurmdbd` (the accounting daemon) is backed by a database and performs the accounting function — recording who ran what, using how much of which resources, and for how long. This is architecturally separate from `slurmctld`'s live state: `slurmdbd` answers "what happened," not "what's happening right now." A site can, in principle, have accounting data lag or gaps without that

affecting live scheduling — the two systems fail independently, which matters directly for Chapter 20's accounting-problem scenarios.

Client commands (`sinfo`, `squeue`, `sbatch`, and the rest of Chapter 17's toolkit) are how users and administrators actually interact with these three components — they aren't components themselves, just interfaces to them.

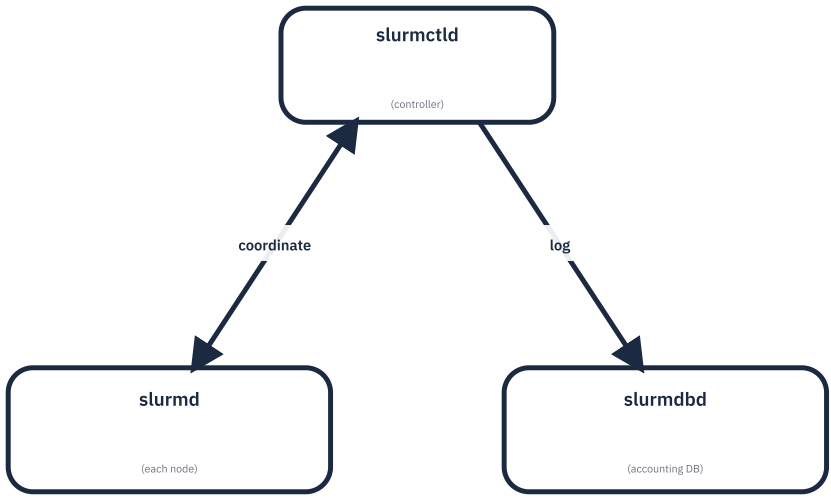


Figure 16.1 — Slurm Component Diagram. Slurm's three components and how they relate — the controller decides, compute daemons execute, the accounting database records. *Components: slurmctld (controller), slurmd (on each compute/GPU node), slurmdbd (accounting database). Connections: controller↔compute-daemon coordination (bidirectional), controller↔accounting-database logging (one direction, historical).*

Not every Slurm deployment looks identical beyond these three core pieces. **Table 16.1** distinguishes what's essentially always present from what varies by site:

Table 16.1 — Slurm Components: Core vs. Site-Dependent

Component	Status	Notes

`slurmctld`	Always present	The cluster cannot schedule work without it
`slurmd`	Always present on compute/GPU nodes	Runs on every node capable of accepting jobs
`slurmdbd`	Commonly present, but not universal	Some smaller or simpler deployments run without persistent accounting; larger sites almost always run it
Backup/standby `slurmctld`	Site-dependent	Configured for controller failover on production clusters where continuity matters; not present on every deployment
`slurmrestd` (REST API interface)	Optional, site-dependent	Provides programmatic/API access to Slurm; deployed where a site needs integration beyond the CLI tools

This book covers the core three components and the client commands built on them; optional components are named here for completeness and recognition, not taught in depth.

16.3 Control Plane vs. Compute/Data Plane

Chapter 2 established a general principle: the scheduler coordinates without being part of the data path. Here's what that means concretely, following a job from submission to execution.

A user submits a job (`sbatch`, Chapter 17). This request goes to `slurmctld` — the control plane. The controller evaluates the request, and once scheduling and allocation (Section 16.1) decide it can run, `slurmctld` sends a launch instruction to `slurmd` on the

allocated node(s) — still control-plane traffic, still not the job's actual work. `slurmd` then starts the job's processes on its node. From that point on, the job's actual work — reading input, computing, writing output, and for multi-node jobs, communicating between nodes — happens over the data/fabric network (Chapter 2, Part 5), never through `slurmctld` or `slurmd` as an intermediary.

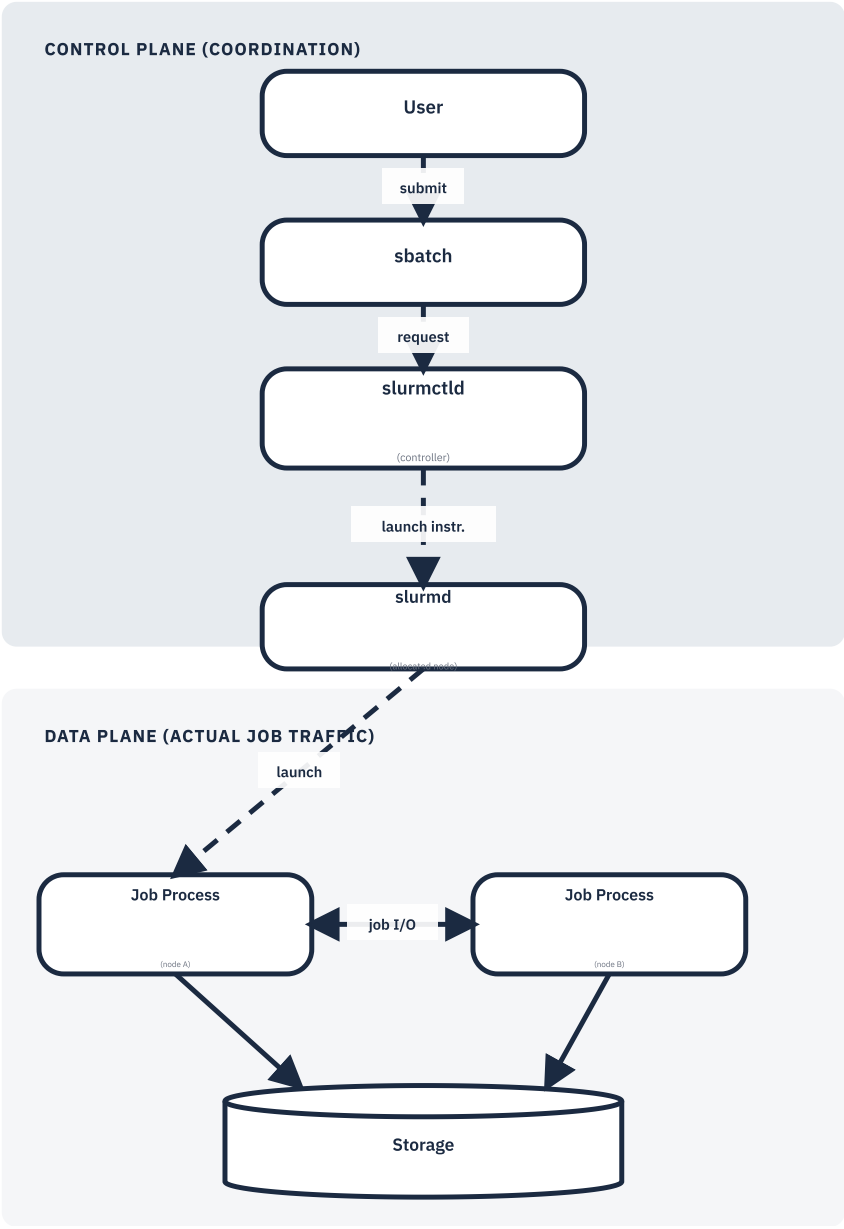


Figure 16.2 — Control Plane vs. Compute/Data Plane: The Job Launch Path. Every arrow up to 'launch' is coordination. Everything after it is the job's own traffic, and Slurm is no longer in that path. *Components: User → sbatch → slurmctld (control plane), slurmctld → slurmd (control-plane instruction, dashed), slurmd → job*

processes (launch), job processes ↔ job processes / storage (data plane, solid, over the fabric). Labels: "Control Plane (coordination)" and "Data Plane (actual job traffic)," mirroring the styling of Chapter 2's Figure 2.2.

This distinction is what makes Chapter 20's troubleshooting layers (User Request → Slurm Job → Scheduler → Node → Resource → Application) meaningful rather than arbitrary: a problem in the control plane (the controller can't reach a node) is a fundamentally different kind of problem than one in the data plane (the job launched fine but its multi-node communication is failing) — the second isn't Slurm's problem at all once launch has succeeded.

16.4 Slurm Job Lifecycle

A job moves through a conceptual sequence of states from submission to completion:

Table 16.2 — Job Lifecycle

State	What's Happening
SUBMITTED	The job has been received by `slurmctld` and is being evaluated.
PENDING	The job is eligible but waiting — on resource availability, priority, or a dependency (Chapter 18).
ALLOCATED / RUNNING	Resources are reserved and `slurmd` has launched the job's processes on the allocated node(s).
COMPLETING	The job's processes have finished or been signaled to finish, and Slurm is cleaning up the allocation (running any configured epilog steps) before fully releasing it.
COMPLETED / FAILED /	

CANCELLED	The job has reached a terminal state, recorded by `slurmdbd` for later query via `sacct` (Chapter 17).

The exact set of state names, sub-states, and how long a job can linger in COMPLETING depend on Slurm version and site configuration (for instance, what a site's epilog scripts do before releasing an allocation) — this table gives the conceptual shape, not a guaranteed-universal state machine.

This lifecycle is the backbone Chapter 20 diagnoses against directly: a job stuck in PENDING, a job that reached RUNNING but never actually produced output, and a job stuck in COMPLETING are three different symptom categories, each pointing toward a different layer of Section 16.3's control/data-plane split.

16.5 Partitions, Nodes, and Resources

A **partition** is a named, policy-defined subset of the cluster's nodes — its own limits, its own access rules, its own scheduling behavior. Partitions can overlap; a given node can belong to more than one. A **node**, in this context, is one of the physical or virtual machines Chapter 2 described, made available to Slurm as somewhere work can run.

Resources are what a job actually requests and what Slurm actually allocates: CPUs, memory, and Generic Resources (GRES, Chapter 19) — most notably GPUs. A job can also request a **constraint** — a feature tag a site can assign to nodes (for instance, a specific CPU generation or a specific network configuration), letting a job's request be matched against node properties beyond the basic CPU/memory/GRES accounting. This chapter names constraints as a concept; using them in a job submission is Chapter 17's territory, and Chapter 17 owns command-level syntax generally — this chapter stops at vocabulary.

16.6 Scheduler Decision Flow

When `slurmctld` decides which pending job runs next, several factors are weighed together, not in isolation:

- **Resource availability** — does the requesting job's need fit within what's currently free, in a partition it's eligible for?
- **Priority** — a value influenced by fairshare (historical usage) and QOS (policy overlays), both covered in depth in Chapter 18.
- **Constraints** — does an eligible node actually match what the job asked for (Section 16.5)?
- **Backfill** — can a lower-priority job use capacity that would otherwise sit idle while a higher-priority job accumulates enough resources, without delaying that higher-priority job (Chapter 18)?
- **Reservations** — capacity a site has explicitly set aside for a specific purpose or time window, outside normal competitive scheduling (named here, not taught in depth).

No single one of these determines the outcome by itself, and exactly how they're weighted against each other is a site configuration choice — this book does not claim one universal scheduling configuration, since real clusters tune this differently based on their own policy goals. Chapter 18 covers each of these factors in the depth this chapter only has room to name.

16.7 Slurm Configuration and Site Dependencies

Slurm's behavior — which partitions exist, which nodes belong to them, how scheduling factors are weighted, how accounting is structured — is governed by configuration, commonly centered on `slurm.conf` along with related files for GRES definitions,

accounting policy, and node-specific settings. This book does not teach editing this configuration directly; exact syntax is version-sensitive and carries real consequences for a live cluster, which puts it outside this book's scope of "what an infrastructure engineer needs to operate and troubleshoot," and squarely into site-administration territory.

What matters here is recognizing that partitions, limits, scheduling weights, and accounting structure all trace back to this configuration — they are site choices, not fixed properties of Slurm itself. Two clusters running the same Slurm version can behave quite differently in scheduling behavior purely because of configuration differences, which is exactly why Chapter 18 avoids asserting a single default policy, and why unexpected-looking behavior (Chapter 20) is often a configuration question before it's a bug.

16.8 Architecture Troubleshooting Map

This table previews where Chapter 20 starts looking for each symptom category, tied back to the components and lifecycle this chapter just established. It is a map, not a diagnosis — Chapter 20 works each of these in full.

Table 16.3 — Symptom to Architecture Layer

Symptom	Likely Layer	First Evidence
`slurmctld` unavailable / cluster-wide command failures	Controller	Controller process/service status; standby failover state if configured
Node shows DOWN	Controller ↔ compute-daemon communication	`scontrol show node` reason field

Job stuck PENDING	Scheduler decision (Section 16.6)	`squeue` reason code, `sprio`
Job allocated but never actually starts running	Launch path (control plane → `slurmd`)	`slurmd` status on the allocated node
GPU allocation mismatch	Resource/GRES layer	`scontrol show job` GRES vs. `scontrol show node` GRES
Accounting data missing or inconsistent	`slurmdbd`	`slurmdbd` service/connectivity status

CHAPTER 17

SLURM

Slurm Commands

17.1 Where This Fits

Chapter 16 named Slurm's components; this chapter teaches the commands you actually use, organized around a real workflow — submit, monitor, inspect, cancel, audit — rather than as an isolated list. Scheduling policy (why a job lands where it does) is Chapter 18's job; this chapter shows the commands working correctly, not diagnosing a problem (Chapter 20).

17.2 Checking Cluster and Node State

`sinfo` (read-only) is usually the first command worth running on an unfamiliar cluster — it reports partitions, node counts, and node states at a glance. A quick `sinfo` before submitting anything tells you whether the cluster (or the partition you intend to use) is in a state that could accept your job at all.

17.3 Submitting Work

Slurm offers three related but distinct ways to submit work, and picking the right one matters. **`sbatch`** submits a batch script for later, asynchronous execution — you provide a script, Slurm queues it, and it runs whenever resources are available; this is the normal way to submit unattended work. **`srun`** launches work directly and can be used either standalone (requesting resources and running a command in one step) or inside an existing

allocation (launching parallel tasks within resources already granted). **`salloc`** requests an allocation without immediately running anything against it — useful for interactive work, where you want a shell inside the allocation to run several commands against it yourself.

A simple `sbatch` submission looks like: `sbatch myjob.sh` — where `myjob.sh` is a script containing `#SBATCH` directives (specifying resources) followed by the actual commands to run. All three of these commands change state — each one either queues or immediately consumes cluster resources.

17.4 Monitoring Jobs

`squeue` (read-only) shows the current state of jobs Slurm is tracking — pending, running, or in various other states — including, for a pending job, a brief reason code. Run without arguments, it typically shows every job on the cluster; most sites let you filter to your own jobs. `squeue` is the fastest way to answer "is my job running yet, and if not, why does Slurm say it's waiting."

17.5 Inspecting Job and Node Detail

`scontrol show job <jobid>` and `scontrol show node <nodename>` (both read-only in this query form) give far more detail than `squeue`'s summary view — every resource request, every state flag, every reason Slurm is tracking for that specific job or node. When `squeue`'s brief reason code isn't enough to understand what's happening, `scontrol show job` is usually the next command to reach for. `scontrol` also has administrative subcommands that change state (e.g., updating a node's state) — covered in Chapter 20, where they're actually needed.

17.6 Canceling Jobs

`scancel <jobid>` (state-changing) terminates a job, whether pending or running. Canceling another user's job is typically restricted by permissions; canceling your own is generally unrestricted but still irreversible for whatever work that job had done — worth being certain about before running it, the same discipline Chapter 4 established for `kill -9`.

17.7 Auditing Completed Work

`sacct` (read-only) queries Slurm's accounting database (`slurmdbd`, Chapter 16) for historical job information — resource usage, exit codes, run time — for jobs that have already completed or failed. This is historical data, not live state; for a job still running, `squeue` is the right tool, not `sacct`. `sstat` (read-only) reports live resource-usage statistics for a currently running job — a middle ground between `squeue`'s state summary and `sacct`'s post-completion accounting.

17.8 Account/Association Administration, Briefly

`sacctmgr` (read-only in its `show` query forms; state-changing for `add`/`modify`/`delete` operations) manages Slurm's accounting hierarchy — users, accounts, and associations that fairshare (Chapter 18) is computed against. This book covers only the read-only query forms in depth; modifying the accounting hierarchy is a site-administration task with cluster-wide policy consequences, similar in spirit to Chapter 3's caution around ad hoc production changes.

17.9 A Full Workflow, Start to Finish

Put together: check cluster state with `sinfo`, submit work with `sbatch`, watch it with `squeue`, inspect it in detail with `scontrol show job` if something looks unclear, cancel it with `scancel` if needed, and once it's done, audit what actually happened with `sacct`.

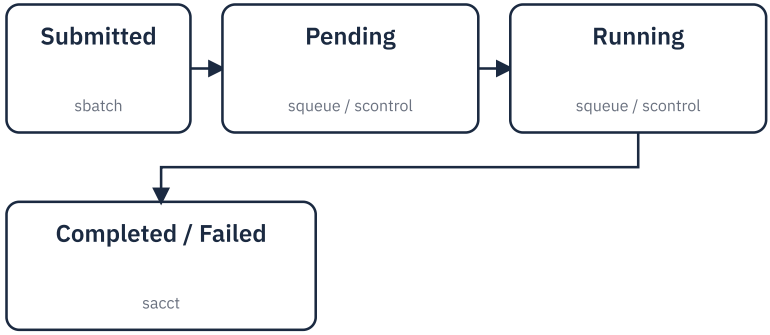


Figure 17.1 — Slurm Job Lifecycle. Each stage of a job's life has a command for acting on it and a command for inspecting it. *Components: Submitted → Pending → Running → Completed/Failed, with the command used at each transition or inspection point annotated (sbatch at submission, squeue/scontrol for pending/running inspection, scancel for cancellation, sacct for post-completion audit).*

Table 17.1 — Command Safety Classification

Category	Commands
● SAFE / READ-ONLY	`sinfo`, `squeue`, `scontrol show ...`, `sacct`, `sstat`, `sacctmgr show ...`
■ CHANGES STATE	`sbatch`, `srun`, `salloc`, `scancel` (your own job)

▲ **POTENTIALLY DISRUPTIVE**	`scancel` against another user's job (where permitted), `scontrol` administrative subcommands (Chapter 20), `sacctmgr add` / `modify` / `delete`

[Illustrative] A user submits a job with `sbatch` and it fails within seconds, while a colleague's job with similar resource requests sits pending for an hour. `squeue`'s brief reason field distinguishes these immediately — one shows a completed/failed state with an exit code, the other shows a specific pending reason — turning what looked like the same kind of "not running" into two clearly different situations before any deeper investigation (Chapter 20) is needed.

[Illustrative] A user wants to know whether their completed job actually used the memory it requested, for future reference. `sacct` with the appropriate fields reports requested versus actual memory usage for the completed job — turning a vague "was that job efficient" question into a specific, evidence-backed answer.

CHAPTER 18

SLURM

Slurm Job Scheduling

18.1 Where This Fits

Chapter 17 taught the commands; this chapter explains the decision-making behind what those commands report — why a job runs when it does, not just how to ask. Chapter 20 covers what to do when this logic appears to produce a wrong or stuck result; this chapter explains the logic itself.

18.2 The Job Lifecycle, Scheduling View

From the scheduler's perspective, a submitted job becomes eligible once its requested resources could, in principle, be satisfied by some node or set of nodes in its partition. Eligible jobs are then prioritized against each other, and placed as resources actually become available. "Pending" covers the entire span between submission and actually running — which is why a pending reason code (Chapter 17's `squeue`) matters so much: it tells you which part of this process a job is waiting on.

18.3 Partitions as Capacity/Policy Boundaries

Chapter 16 introduced partitions as named, possibly-overlapping groupings of nodes with their own limits. From a scheduling perspective, a partition is the first boundary a job's request is

checked against — a job requesting resources a partition can't provide (more nodes than it has, a resource type it doesn't offer) will never become eligible in that partition, regardless of priority.

18.4 Priority and Fairshare

Every eligible job has a priority value Slurm uses to decide scheduling order. Fairshare is one major input into that value: it adjusts a job's priority based on the submitting user's or account's recent historical usage relative to their allocated share — heavier recent usage lowers future priority, lighter usage raises it. This is not "everyone gets equal priority" — it's a mechanism that lets priority self-correct over time based on who's actually been using the cluster.

18.5 QOS

Quality of Service (QOS) definitions are policy overlays a job (or an account) can be assigned — they can adjust priority, impose additional limits (maximum run time, maximum resource count), or grant exceptions to normal limits. A job failing immediately with a QOS-related limitation is a policy rejection, not a bug, and not the same thing as a job that's merely pending for resource availability.

18.6 Backfill

Backfill is the scheduler's mechanism for filling gaps: when a high-priority job needs to wait for enough resources to accumulate, backfill can schedule smaller, lower-priority jobs into the resources that would otherwise sit idle in the meantime — as long as doing so doesn't delay the high-priority job's eventual start. This

is why a smaller job sometimes starts running before a larger, higher-priority one that was submitted earlier — not favoritism, an opportunistic use of otherwise-wasted capacity.

18.7 Preemption

Where configured, preemption lets a higher-priority job displace a already-running lower-priority one, freeing resources immediately rather than waiting for that job to finish naturally. Preemption and backfill solve different problems — backfill uses idle capacity opportunistically; preemption reclaims already-allocated capacity forcibly. Not every site enables preemption; its presence and exact behavior are configuration-dependent.

18.8 Resource Requests

A job's CPU and memory requests (among others) are what the scheduler matches against available capacity. Requesting too little can cause a job to be killed or throttled once it actually needs more than it asked for (Chapter 4's OOM scenario is one concrete version of this); requesting too much can leave a job pending far longer than necessary, waiting for a larger allocation than the work actually needs. Neither mistake is visible from the job script alone — `sacct` (Chapter 17) after the fact is how you'd confirm whether a completed job's request matched its actual usage.

18.9 Node States and Pending Reasons

Nodes carry a state (Chapter 20 covers diagnosing unexpected ones in depth) that affects whether the scheduler considers them for new work. A pending job's reason code, visible in `squeue`/`scontrol show job` (Chapter 17), reflects the specific thing Slurm is waiting on — resource availability, a priority

ordering, a dependency on another job, among others. Reading the reason code is the evidence-gathering step Chapter 1's philosophy calls for, before assuming a pending job is simply "stuck."

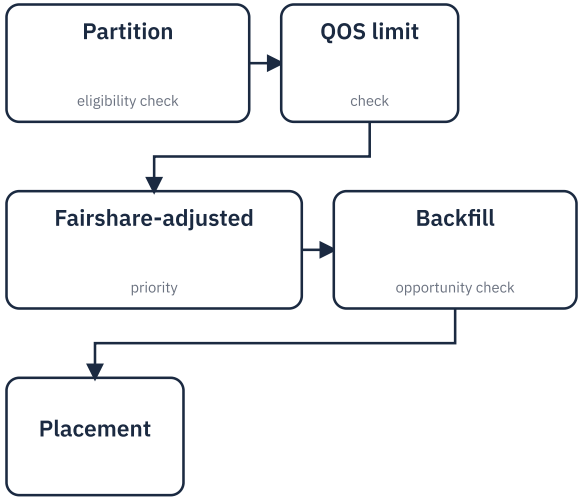


Figure 18.1 — Slurm Scheduling Decision Flow. A job's path to running passes through several independent checks — a pending job may be waiting on any one of them. *Components: partition eligibility check → QOS limit check → fairshare-adjusted priority → backfill opportunity check → placement.*

18.10 Scheduling Efficiency (Performance)

Backfill and QOS interact directly with overall scheduling throughput: a partition with very restrictive QOS limits can leave capacity idle even when backfill-eligible jobs exist, if those jobs don't fit within the QOS constraints. This is a site-tuning consideration, not a universal default — how a given cluster's fairshare/backfill/QOS configuration trades off responsiveness against throughput depends entirely on that site's policy choices, which this book doesn't attempt to generalize about.

Table 18.1 — Scheduling Concepts Compared

Concept	What It Controls
Partition	Which nodes/capacity a job can even be considered against
QOS	Policy limits and priority adjustments, assignable per job/account
Reservation	Capacity set aside for a specific purpose/time window, outside normal scheduling

18.11 Misconceptions and Illustrative Scenarios

Fairshare doesn't mean equal priority — it means priority that adjusts based on usage history. Backfill doesn't mean smaller jobs always jump ahead — only when doing so doesn't delay higher-priority work. A pending job isn't automatically stuck — pending is often normal, evidence-backed waiting.

[Illustrative] Two jobs, submitted around the same time with nearly identical resource requests, start at very different times. Checking each job's priority (`sprio`, forward-referenced here, covered in Chapter 20) reveals one account had used significantly more cluster time recently than the other — fairshare, not a scheduler fault, explains the difference.

[Illustrative] A multi-job dependency chain silently stops progressing. The dependent job's pending reason references a job ID that, on inspection, failed rather than completed — Slurm is correctly waiting on a dependency that will never be satisfied, not malfunctioning.

CHAPTER 19

SLURM

Slurm GPU Scheduling

19.1 Where This Fits

Chapter 18 covered general scheduling policy; Chapter 13 covered GPU infrastructure. This chapter is where they meet: how a job actually requests and receives GPUs, and what the application sees as a result. GPU hardware faults are Chapter 14's territory; this chapter is about the request-to-allocation-to-visibility chain working correctly — or not.

19.2 GRES: Generic Resources

Slurm generalizes "a resource beyond CPU and memory that a job can request" as a Generic Resource (GRES). GPUs are the primary HPC example, but GRES as a mechanism isn't GPU-specific — it's a slot Slurm uses for any countable, schedulable resource a node can offer. A node's configuration declares what GRES it has (e.g., how many GPUs); the scheduler matches job requests against that declared inventory.

19.3 Requesting GPUs

A job requests GPUs through GRES-related flags on `sbatch`/`srun` (Chapter 17) — commonly a form like `--gres=gpu:2` (request two GPUs) or, on more recent Slurm versions, `--gpus=2`, `--gpus-per-node`, `--gpus-per-task`, and related more GPU-specific flags. The `--gpus*` family of flags depends on Slurm's `select/cons_tres`

plugin being configured on the cluster — where it isn't, jobs requesting them are rejected, and the more general `--gres=gpu:N` form is the one that applies instead. Exact flag names and behavior have evolved across Slurm versions and site configurations, which is why this book names the concept precisely while flagging the exact syntax for verification against the specific Slurm version and configuration in use before relying on it.

19.4 What the Application Actually Sees

Once a job is allocated GPUs, Slurm sets `CUDA_VISIBLE_DEVICES` (an environment variable, Chapter 3) in the job's environment, restricting which GPUs the application can see and use. This is a job-local, remapped view — a job allocated one specific physical GPU might see it as device `0` from its own perspective, regardless of that GPU's actual physical device number. This remapping is normal and expected; it's not evidence a job received the wrong GPU.

19.5 Validating Requested vs. Allocated

Three related but distinct facts are worth checking independently when something looks off: what the job requested (visible in the job script or `scontrol show job`), what Slurm actually allocated (also `scontrol show job`), and what the application can see (`CUDA_VISIBLE_DEVICES` inside the job's environment, or the job's own `nvidia-smi` output). A mismatch between any two of these three is diagnostic evidence, not a single fixed symptom — checking all three separately is the point.

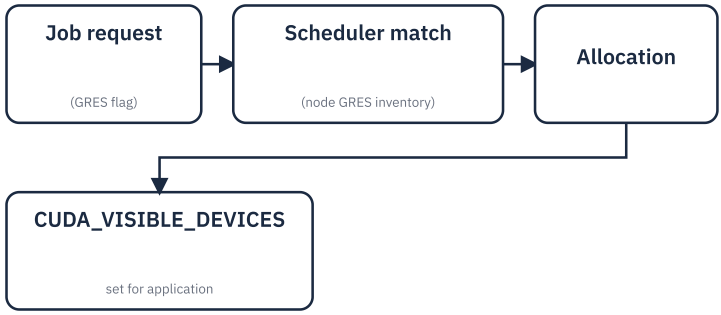


Figure 19.1 — GRES/GPU Allocation Flow. A GPU request flows from the job script to the application's own view of available devices — each step can be independently verified. *Components: job request (GRES flag) → scheduler match against node GRES inventory → allocation → CUDA_VISIBLE_DEVICES set for the application.*

19.6 Topology-Aware Scheduling, Conceptually

A job requesting multiple GPUs doesn't automatically get GPUs that are well-connected to each other (Chapter 15's NVLink/PCIe topology) unless the scheduler is configured to consider topology when placing it. This is a real, if easily overlooked, gap: a job can receive the requested *number* of GPUs while still landing on a topologically poor combination, which shows up as a performance problem (Chapter 15's scenario), not an allocation error.

19.7 MIG and Scheduling

Where MIG (Chapter 13) is in use, a job requests a MIG instance rather than a full GPU, and Slurm's GRES configuration has to reflect the specific MIG instances available on a node for this to work correctly. A job expecting a full GPU but receiving a MIG instance (or vice versa, due to a configuration mismatch) will

typically fail or behave unexpectedly at the application level, not at the scheduling level — the allocation "succeeds" from Slurm's perspective even though the outcome isn't what was intended.

19.8 Common Allocation Mistakes

Table 19.1 — GRES Misconfiguration Checklist

Symptom	Likely Cause
Job requests GPUs but lands on a GPU-less node	GRES not declared in that node's configuration despite the node having GPUs
Job gets fewer GPUs than requested silently	Partial GRES availability combined with a flag that doesn't strictly enforce the exact count
Application reports the wrong number/type of GPU	`CUDA_VISIBLE_DEVICES` mismatch — check the allocation, not just the request
Multi-GPU job's GPUs aren't well-connected	Topology-unaware placement (Section 19.6)

Distinguishing a Slurm allocation problem from a GPU hardware/driver problem: if `scontrol show job` shows the expected GRES allocation and `CUDA_VISIBLE_DEVICES` matches it, but the application still fails, the problem has moved out of Slurm's territory and into Chapter 14's (driver, CUDA, or hardware). If the allocation itself doesn't match the request, the problem is here, in scheduling/configuration — not a hardware fault at all.

19.9 Misconceptions and Scenarios

Requesting N GPUs doesn't guarantee they're on the same NVLink domain unless topology-aware constraints are actually configured and used. `CUDA_VISIBLE_DEVICES` numbering is a job-local view, not necessarily matching physical GPU IDs. MIG instances are scheduled similarly to full GPUs but aren't interchangeable with them — a configuration mismatch between what's requested and what's declared produces confusing, not obviously-GRES-related, failures.

[Illustrative] A job requests two GPUs and Slurm reports the allocation succeeded, but the application fails immediately with a device-not-found-style error. Checking `scontrol show job`'s GRES allocation against the node's actual GRES configuration reveals the node was never correctly configured with GPU GRES in the first place — the job was scheduled onto a node Slurm believed had GPUs available, but didn't, in practice, correctly declare them.

[Illustrative] A multi-GPU job shows poor scaling despite receiving the requested number of GPUs, eventually traced to the scheduler placing the job's GPUs without regard to NVLink topology — ties directly to Chapter 15's scenario, viewed here from the scheduling side rather than the topology side.

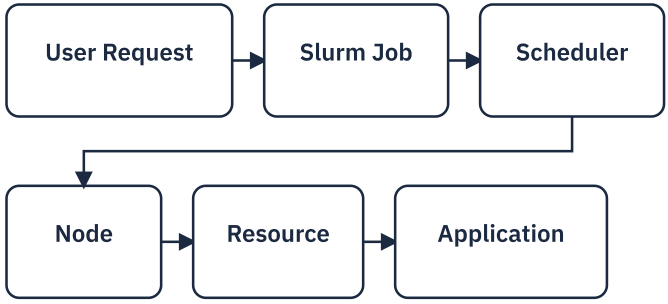
CHAPTER 20

SLURM

Slurm Troubleshooting

20.1 The Layered Model

A Slurm-related symptom can originate at any of six layers: User Request, Slurm Job, Scheduler, Node, Resource, or Application. As with every other troubleshooting chapter in this book, the layer where a symptom shows up isn't always where the fix belongs — a job that "fails" might be a user-error resource request, a scheduler-policy effect, a node problem, a resource mismatch, or a genuine application issue.



Each layer has its own common pending/failure causes — walk the chain with evidence, not guesswork

Figure 20.1 — Slurm Pending-Job Diagnostic Flow. A pending or failed job's cause is usually findable by walking this chain with evidence, not guesswork. *Components: the six-layer chain with branch points for the most common pending/failure causes at each layer.*

Every scenario in this chapter uses the full ten-section format.

20.2 Job Pending, Diagnosed

The full diagnostic path: `squeue` for the reason code, `scontrol show job` for detail, `sprio` (Chapter 18, introduced fully here) for priority visibility, and `sinfo` for whether the relevant partition even has capacity right now. Most "why is my job pending" questions resolve at one of these four steps.

20.3 Node Drain vs. Node Down

A node marked **DRAINED** is typically an operator-initiated state — the node will finish any current work but won't accept new jobs, usually ahead of planned maintenance. A node marked **DOWN** usually reflects the scheduler losing contact with it or a health check (Chapter 21) failing. Treating these the same — assuming either always means "broken hardware" — leads to wasted effort; `scontrol show node` reports the specific reason associated with the state, which is the evidence that distinguishes them.

20.4 Allocation Problems and Incorrect Resource Requests

A job that fails immediately, rather than pending, is often a request Slurm can reject outright — a resource request no partition can satisfy, or a QOS limit violation (Chapter 18). This category is usually user error or a policy mismatch, not a cluster fault, and `scontrol show job`'s recorded reason (or the immediate error at submission time) typically says so directly.

20.5 GPU Allocation Failures

Building on Chapter 19: a GPU allocation failure can mean the request itself was invalid, the scheduler's GRES matching failed to find a suitable node, or the node's GRES configuration doesn't actually match its physical GPUs (a stale-state problem, common right after a node reboot if GRES detection didn't run correctly). Checking `scontrol show node` for that node's currently detected GRES, compared against what's expected, isolates this quickly.

20.6 slurmctld/slurmd Communication Problems

When `slurmd` on a compute node can't communicate with `slurmctld` (Chapter 16), that node typically shows as unresponsive or down from the scheduler's perspective, even if the node itself and any jobs already running on it are otherwise healthy. This is a network/service problem between two Slurm components, not necessarily a hardware fault on the compute node — the same host-level checks from Chapter 3/4 (is `slurmd` running, is the management network reachable) apply directly.

20.7 Accounting Problems

A discrepancy between what a user expects `sacct` (Chapter 17) to show and what it actually shows is often a timing or query-window issue — accounting data is written after certain job-state transitions, and querying too soon or with the wrong time window can look like missing data when nothing is actually lost. Genuine accounting database problems (Chapter 16's `slurmdbd`) are rarer and typically show up as a broader pattern (many jobs affected, not just one).

20.8 Scheduling Anomalies

When priority or backfill behavior looks wrong (a job runs later than expected, or earlier than expected, relative to others), `sprio`'s breakdown of a job's priority components is the evidence to check before assuming a bug — fairshare, QOS, and age-based factors (Chapter 18) each contribute, and an "anomaly" is often one of them working exactly as configured, just not as the observer expected.

20.9 Common Mistakes

Canceling and resubmitting a pending job without checking why it was pending treats a symptom, not a cause — if the underlying reason (a QOS limit, a resource shortage) hasn't changed, the resubmitted job will likely pend for the same reason. This is the same "fix the symptom, not the root cause" mistake Chapter 1's philosophy exists to prevent.

20.10 Worked Scenarios

Table 20.1 — Command Safety, This Chapter

Category	Commands
● SAFE / READ-ONLY	`scontrol show ...`, `sdiag`, `squeue`, `sprio`, `sacct`
■ CHANGES STATE	`scontrol update` (e.g., changing a node's state)
▲ POTENTIALLY DISRUPTIVE	Returning a node to service (resume) without confirming the underlying drain/down reason is actually resolved

[Illustrative]

Scenario 1 — Job PENDING for Hours

SYMPTOM	A submitted job has remained in PENDING state for several hours with no obvious progress.
INITIAL CHECKS	Run `squeue` and read the reason code for this job specifically.
COMMANDS	`squeue`, `scontrol show job`, `sprio`, `sinfo`.
OUTPUT INTERPRETATION	The reason code indicates resource unavailability in the requested partition; `sinfo` confirms that partition is near full capacity.
POSSIBLE CAUSES	Genuine resource contention; an overly large resource request; a QOS/partition limit the job doesn't fit within.
FAULT ISOLATION	Comparing the job's request against currently available capacity (`sinfo`) and its priority (`sprio`) relative to other pending jobs narrows this to genuine contention, not a configuration error.
ROOT CAUSE	**The cluster's requested partition was near full capacity, and the job's priority placed it behind other legitimately higher-priority work.**
RESOLUTION	No intervention required — the job will run once capacity/priority allow; alternatively, resubmitting with a smaller request or a different partition if the wait is unacceptable.

VALIDATION	Confirm the job transitions to RUNNING once capacity becomes available, consistent with the priority-based expectation.
PREVENTION	For workloads sensitive to wait time, plan resource requests and partition choice with current cluster load in mind, rather than assuming default availability.

[Illustrative]

Scenario 2 — Node Shows DRAINED

SYMPTOM	A node stops accepting new jobs, though jobs already running on it continue.
INITIAL CHECKS	`scontrol show node` for the recorded reason.
COMMANDS	`scontrol show node`.
OUTPUT INTERPRETATION	The node's state includes a reason string indicating an operator-initiated drain, with a timestamp and (often) a brief note.
POSSIBLE CAUSES	Planned maintenance; a manual drain ahead of a firmware update (Chapter 25); a precautionary drain after a suspicious health signal (Chapter 6/21).
FAULT ISOLATION	The reason string itself typically resolves this — distinguishing operator-initiated from automatic requires reading it, not guessing.
ROOT CAUSE	**In this scenario, the drain was operator-initiated ahead of scheduled maintenance.**
RESOLUTION	No fix needed — this is working as intended; the node returns to service once maintenance completes and it's explicitly resumed.

VALIDATION	Confirm the node accepts new jobs again after being resumed post-maintenance.
PREVENTION	Ensure drain reasons are always recorded with enough detail that a different engineer, days later, can understand why a node was drained without needing to ask.

[Illustrative]

Scenario 3 — Node Shows DOWN

SYMPTOM	A node is marked DOWN, and the scheduler is not sending it new work.
INITIAL CHECKS	Distinguish communication failure from health-check failure via `scontrol show node`'s reason.
COMMANDS	`scontrol show node`, host-level checks (Chapter 3/4) if reachable, `dmesg` / `journalctl` on the node itself if accessible.
OUTPUT INTERPRETATION	The reason indicates the scheduler stopped receiving responses from `slurmd` on this node.
POSSIBLE CAUSES	The node itself is down/unreachable (a genuine outage); `slurmd` crashed or isn't running while the node is otherwise up; a management-network issue (Chapter 2) preventing communication specifically.
FAULT ISOLATION	Attempting to reach the node directly (SSH, Chapter 3) separates "node is genuinely unreachable" from "node is reachable but `slurmd` isn't responding."

ROOT CAUSE	In this scenario, the node was reachable via SSH, but `slurmd` had stopped running.
RESOLUTION	Restart `slurmd` on the affected node (Chapter 3's systemd commands apply directly) and confirm it reconnects to the controller.
VALIDATION	Confirm `scontrol show node` reflects the node returning to a normal state after `slurmd` restarts.
PREVENTION	Monitor `slurmd`'s own service health (Chapter 21) as a distinct signal from general node reachability, since the two can differ.

[Illustrative]

Scenario 4 — Job Fails Immediately with a QOS Limitation

SYMPTOM	A job fails within seconds of submission, and the user assumes it's a bug.
INITIAL CHECKS	Check the immediate submission response or `scontrol show job` for a QOS-related rejection reason.
COMMANDS	`scontrol show job`, `sacctmgr show qos` (read-only).
OUTPUT INTERPRETATION	The job's resource request exceeds a limit imposed by the QOS assigned to the user's account.
POSSIBLE CAUSES	The user is unaware of their account's QOS limits; the request was a typo (e.g., an extra zero on a resource count).

FAULT ISOLATION	Comparing the request against the specific QOS limit confirms this directly — no further isolation needed.
ROOT CAUSE	**The job's request exceeded a configured QOS limit, and Slurm correctly rejected it.**
RESOLUTION	Resubmit within the QOS limit, or request a QOS/limit change through the appropriate site process if the limit itself is the actual problem.
VALIDATION	Confirm the corrected submission is accepted and proceeds normally.
PREVENTION	Make QOS limits visible to users before submission (documentation, a pre-submission check) so this resolves before submission rather than after.

[Illustrative]

Scenario 5 — Dependency Deadlock Between Two Jobs

SYMPTOM	A job with a dependency on another job never starts, even though the cluster has available capacity.
INITIAL CHECKS	Check the state of the job it depends on.
COMMANDS	`scontrol show job` (for both the dependent job and the one it depends on), `sacct` (if the dependency target has already finished).
OUTPUT INTERPRETATION	The job being depended upon shows a FAILED state, not COMPLETED.

POSSIBLE CAUSES	The dependency was configured for successful completion specifically, and the upstream job failed; a misconfigured dependency chain referencing the wrong job ID.
FAULT ISOLATION	Confirming the upstream job's actual final state (via `sacct`) resolves this directly.
ROOT CAUSE	**The dependency chain was correctly waiting on a job that failed rather than completed, and Slurm's dependency logic was working exactly as configured.**
RESOLUTION	Address why the upstream job failed, then resubmit the dependent chain, or adjust the dependency condition if failure-tolerant continuation was actually intended.
VALIDATION	Confirm the dependent job starts once its dependency condition is genuinely satisfied.
PREVENTION	Consider whether dependency chains should distinguish "must complete successfully" from "must simply finish" based on the actual workflow need, rather than defaulting to one without thinking it through.

[Illustrative]

Scenario 6 — GPU Allocation Failure Traced to Stale GRES State After a Node Reboot

SYMPTOM	A GPU job is scheduled onto a specific node, but the application reports no GPUs available.
INITIAL CHECKS	Compare the node's currently configured GRES against its actual, currently detected GPUs.

COMMANDS	`scontrol show node`, `nvidia-smi` (Chapter 13) on the node directly.
OUTPUT INTERPRETATION	`nvidia-smi` on the node shows all GPUs present and healthy, but `scontrol show node` reports zero GRES configured for that node — the controller's view of the node's resources doesn't match reality.
POSSIBLE CAUSES	GRES auto-detection didn't run correctly after a node reboot; a driver-loading delay meant GPUs weren't visible when `slurmd` started and reported its inventory.
FAULT ISOLATION	The timing gap between the driver becoming ready and `slurmd` starting is confirmed by comparing boot-sequence timestamps for the driver module load against `slurmd`'s own start time in its log.
ROOT CAUSE	**`slurmd` started and reported its GRES inventory to the controller before the GPU driver had finished initializing, so it correctly reported zero GPUs available at that moment — a boot-sequencing issue, not a hardware or configuration fault.**
RESOLUTION	Restart `slurmd` on the affected node now that the driver is ready, so it re-reports the correct GRES inventory to the controller.
VALIDATION	Confirm the node's reported GRES matches its actual GPU inventory, and a subsequent GPU job allocates correctly.

PREVENTION	Sequence node-boot health checks (Chapter 6/21) so GPU/driver readiness is confirmed before `slurmd` starts and reports its resource inventory to the controller.

PART 8

Monitoring & UFM

Chapter 21, Chapter 22

CHAPTER 21

MONITORING

HPC Monitoring

21.1 Where This Fits

Chapter 6 introduced node health checks as a concept, deferring their operational use here. This chapter is about monitoring philosophy generally — what to watch, and why — across every layer this book has covered so far. It is deliberately tool-agnostic; no specific monitoring product is taught or recommended.

21.2 Monitoring Philosophy

Good monitoring distinguishes leading indicators (a signal that predicts a future problem, like a rising correctable-error count, Chapter 6) from lagging indicators (a signal that confirms a problem has already happened, like a job failure). A cluster that only monitors lagging indicators finds out about problems the same way its users do — after the fact. Leading indicators are what let an engineer act before an outage, not just document one afterward.

21.3 What to Monitor, By Layer

CPU and memory (Chapter 4/5): utilization trends, memory pressure, and NUMA-related metrics where relevant — mostly useful as trend data; acute diagnosis is Chapter 4's job, not monitoring's. **Storage** (Chapter 7/8): capacity trends, latency, and error rates from the storage system itself, distinct from any one

job's I/O pattern. **Network/fabric** (Chapter 9–12): link state and error counters fleet-wide — the same evidence Chapter 12 uses for a single incident, aggregated across the whole fabric so a pattern (Chapter 26's asymmetric-utilization scenario) becomes visible before it's reported as a complaint. **GPU** (Chapter 13/14): temperature, power, ECC error trends, and utilization — DCGM (Chapter 13) is the typical foundation for fleet-wide GPU monitoring specifically. **Scheduler** (Chapter 16–20): queue depth, pending-job counts by reason, and node-state distribution (how many nodes are drained/down at any time) as a fleet-health signal in its own right.

Each of these is monitored for trend and pattern; none of this chapter re-teaches how to diagnose a single incident in any of these layers — that's each layer's own chapter.

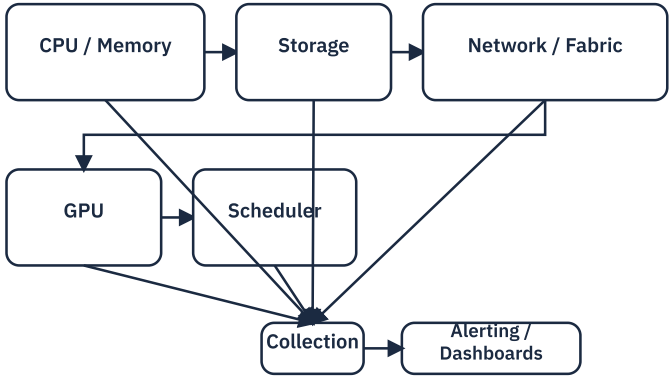


Figure 21.1 — Monitoring Data Flow. Monitoring data flows from every layer into a shared view — correlation across layers is what turns raw data into a useful signal. *Components: metrics/logs/events sources from each layer (CPU/memory, storage, network/fabric, GPU, scheduler) flowing into collection, then into alerting/dashboards.*

21.4 Node Health Checks in Practice

Chapter 6 introduced node health checks (NHC) as a concept — automated tests confirming a node is actually fit for work, not just powered on. In practice, these checks run at points like before a job starts or on a recurring schedule, and a failing check typically drains the node (Chapter 20) automatically rather than waiting for a human to notice a pattern of failed jobs. The quality of an NHC setup is measured by what it catches before a job runs, not by how quickly it's noticed after several jobs have already failed on a bad node.

21.5 Logs and Events as Monitoring Inputs

Metrics (numeric trends) and logs/events (discrete occurrences, Chapter 4's `dmesg` / `journalctl`) answer different questions and are most useful together — a metric might show a GPU's error count rising, while the corresponding log entries show exactly when and what kind of error, letting an engineer correlate the trend with specific events rather than treating the number in isolation.

21.6 Alerting Philosophy

An alert should mean something actionable is happening — not simply that a metric crossed a threshold someone picked once and never revisited. Too many low-value alerts cause real ones to be missed or ignored (alert fatigue), which is a genuine cost, not a minor inconvenience. Alerting design is as much about what *not* to alert on as what to.

21.7 Capacity Monitoring

Capacity monitoring asks a different question than fault monitoring: not "is something broken right now" but "how much runway is left before something will need attention" — storage filling up over months, a GPU fleet's utilization trending toward saturation, and similar trend-based questions. This is planning data, not incident data, and conflating the two dashboards tends to bury the planning signal under noisy operational alerts.

21.8 Correlation Across Layers

A single real-world symptom often has evidence at more than one layer simultaneously — a storage slowdown (Chapter 8) might show up as elevated I/O wait (Chapter 4) on many nodes at once, which is itself a correlation signal pointing away from any one node and toward the shared resource. Monitoring that only looks at one layer at a time misses exactly this kind of cross-layer pattern, which is often the fastest way to distinguish "many things are independently wrong" from "one shared thing is causing many symptoms."

21.9 Proactive vs. Reactive Monitoring

Put together: proactive monitoring uses leading indicators, trend data, and cross-layer correlation to catch problems before they're user-visible; reactive monitoring (alerting on already-occurred failures) is still necessary but is the floor, not the goal. A mature monitoring setup does both, and knows which is which.

"What to Monitor" Checklist by Layer

Layer	Primary Signal Type	Chapter for Deep Diagnosis

Hardware/OS	Trend + event	Ch.4, 6
Storage	Trend + latency	Ch.7, 8
Fabric	Error-counter trend	Ch.9–12
GPU	Trend + Xid events	Ch.13, 14
Scheduler	Queue/state distribution	Ch.16–20

21.10 Misconceptions and an Illustrative Scenario

More alerts do not mean better monitoring — often the opposite. Monitoring does not tell you the root cause — it tells you something is different, which is where troubleshooting (not monitoring) takes over. Uptime is not the same as health — a node can be up, reachable, and still failing every job it's given.

[Illustrative] A node passes a basic liveness check (it responds on the network, `slurmd` is running) but has been failing every job assigned to it for hours. The liveness check was never designed to catch this — it confirms the node is *up*, not that it's *fit for work*. A workload-aware health check (actually running a small representative task, not just pinging a service) would have caught this before the fourth or fifth failed job, not after.

CHAPTER 22

MONITORING

NVIDIA UFM

22.1 Where This Fits

Chapters 9 and 11 both mentioned UFM as a forward reference, without depth. This chapter fulfills that: what UFM is for, how it relates to the subnet manager, and when it's the right starting point versus when Chapter 12's manual tools are still needed. This is not a low-level troubleshooting chapter — Chapter 12 owns that.

22.2 UFM vs. the Subnet Manager

NVIDIA UFM (Unified Fabric Manager) is a fabric management and monitoring platform. It is not a subnet manager, and it does not replace one — the subnet manager (Chapters 9 and 11) is what makes the fabric operate at all (address assignment, routing computation); UFM sits alongside it as a monitoring and management layer, consolidating information a fabric administrator would otherwise gather tool-by-tool. A fabric can run without UFM (using only manual tools and a subnet manager); UFM adds visibility and convenience, not a missing operational requirement.

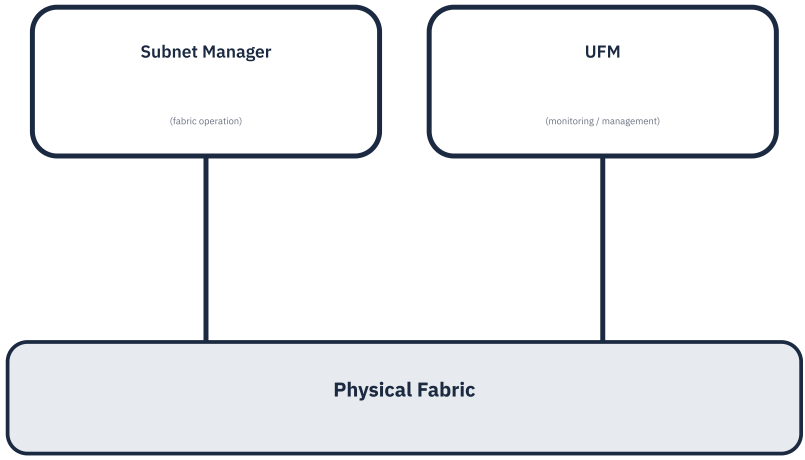


Figure 22.1 — UFM's Position in the Fabric Management Stack. UFM sits alongside the subnet manager as a monitoring and management layer — it doesn't replace what keeps the fabric running. *Components: subnet manager (fabric operation) and UFM (monitoring/management layer), both shown relative to the physical fabric, drawing from the same underlying fabric data manual tools also access.*

22.3 Topology Visibility

UFM maintains a consolidated view of the fabric's topology, comparable in substance to what `ibnetdiscover` (Chapter 11) reports from any single node's perspective, but aggregated centrally and kept current without needing to re-run a manual sweep. For a large fabric, this is a genuine time saver over manually correlating multiple `ibnetdiscover` outputs.

22.4 Port and Health Tracking

UFM tracks per-port state and error counters fleet-wide — the same underlying data `ibstat`/`perfquery` (Chapter 11) report per node, consolidated into one view rather than requiring a login to each

node. This is what makes UFM useful for spotting a pattern (Chapter 26's asymmetric-link-utilization scenario) that would be tedious to notice by checking nodes one at a time.

22.5 Events and Alarms

Beyond passive visibility, UFM can raise events and alarms proactively when it detects a condition worth attention — a port's error counters crossing a threshold, a link going down — rather than requiring someone to notice during a manual check. This shifts fabric monitoring from purely reactive (someone runs a sweep and finds a problem) toward the proactive philosophy Chapter 21 describes generally, applied specifically to the fabric layer.

22.6 UFM Variants, Briefly

NVIDIA offers UFM as more than one product tier — including a telemetry-focused tier and a more full-featured, "Enterprise"-oriented tier with expanded fabric management capabilities, among other offerings in the UFM family. This book names the fact that multiple tiers exist without comparing their exact feature sets in product-marketing depth, since those differences change across releases and choosing between them is a licensing/deployment decision for a given site, not a technical concept this book needs to teach.

22.7 UFM's Relationship to Troubleshooting

UFM is a reasonable starting point for a fabric-wide question — "where, broadly, should I look" — because it aggregates the same evidence Chapter 12's manual tools use, without requiring a sweep across many nodes first. It is not a replacement for Chapter 12's fault-isolation methodology: once UFM points at a specific link,

port, or switch, the detailed diagnosis — what healthy looks like, what failure looks like, what to check next — still follows Chapter 12's approach.

22.8 Misconceptions and a Scenario

UFM does not replace the subnet manager — they perform entirely different roles. UFM does not show fundamentally different data than manual tools show — it shows largely the same underlying fabric data, consolidated and made proactive. And having UFM deployed doesn't mean Chapter 11/12's tools become unnecessary knowledge — UFM accelerates finding *where* to look; detailed isolation still often needs those tools.

[Illustrative] A fabric-wide performance complaint, difficult to localize by checking nodes one at a time, is resolved noticeably faster by using UFM's consolidated view to identify the specific switch or link responsible, compared to a manual `ibdiagnet`-based sweep across many nodes.

PART 9

NCCL & AI Workloads

Chapter 23, Chapter 24

CHAPTER 23

NCCL

NCCL Fundamentals

23.1 Where This Fits

Chapter 1 already drew the essential line between MPI and NCCL, and Chapter 15 covered the GPU-to-GPU topology (NVLink, NVSwitch, PCIe) that NCCL operates over. This chapter builds on both: what NCCL actually is, its core vocabulary, and how it selects a transport — without re-deriving Chapter 1's framing or Chapter 15's topology content from scratch.

23.2 What NCCL Is

NCCL (NVIDIA Collective Communications Library) is a library providing GPU-to-GPU communication primitives that are topology-aware — meaning it's built specifically to move data efficiently between GPUs given whatever physical connections (NVLink, PCIe, InfiniBand, Ethernet) actually exist between them. It is not a general-purpose parallel-programming framework; its scope is specifically collective and point-to-point communication between GPUs.

23.3 Ranks and Communicators

A **rank** is a participant in an NCCL operation — typically one process, often corresponding to one GPU. A **communicator** is the group of ranks that will communicate together for a given operation; every rank in a communicator needs a consistent view

of who else is participating for a collective operation to complete correctly. This vocabulary deliberately mirrors MPI's own rank/communicator concepts, which is intentional on NCCL's part — its documentation describes NCCL's API as closely following the collectives API MPI popularized, precisely so that engineers already familiar with MPI find NCCL's interface natural.

23.4 The Collectives: AllReduce, Broadcast, Reduce, AllGather

These four operations, conceptually:

- **Broadcast** — one rank's data is sent to every other rank in the communicator.
- **Reduce** — every rank's data is combined (e.g., summed) into a result held by one rank.
- **AllReduce** — every rank's data is combined, and the combined result ends up on *every* rank, not just one. This is the operation most central to distributed training, since it's how gradients computed independently on each GPU get combined into a single, consistent update.
- **AllGather** — every rank's data is collected onto every rank, without combining it — each rank ends up with the full concatenated set.

This book explains what each operation moves and produces, not the underlying ring or tree algorithm NCCL uses internally to implement it efficiently — that implementation detail is beyond this book's infrastructure-operations scope.

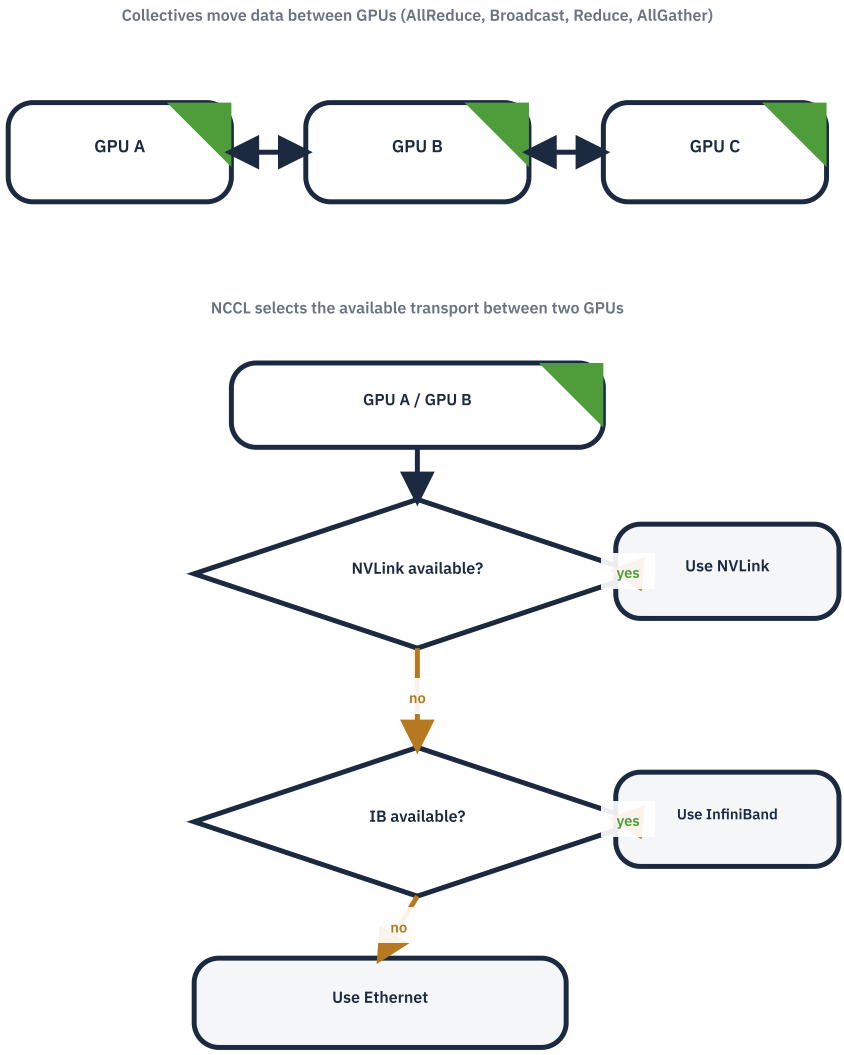


Figure 23.1 — NCCL Collective Operations and Transport Selection. What each collective moves, and how NCCL decides which physical path to use. *Two-part figure: (a) a visual of AllReduce/Broadcast/Reduce/AllGather showing data movement between a small set of GPUs; (b) a decision-flow showing NCCL selecting NVLink vs. IB vs. Ethernet based on what's available between two GPUs (ties to Chapter 15's topology matrix).*

23.5 NCCL and CUDA

NCCL integrates directly with CUDA — its operations are issued onto CUDA streams, meaning NCCL communication can be interleaved with GPU computation using the same stream-based model CUDA applications already use elsewhere. This is one of the ways NCCL differs structurally from MPI: NCCL collectives take a stream argument, tying communication directly into the GPU's own execution model, rather than treating communication as a separate, CPU-orchestrated step.

23.6 NCCL and MPI — Getting This Right

This is worth stating precisely, since it's a common point of confusion. **MPI** (Message Passing Interface) is a long-standing, general-purpose standard for message passing in parallel and distributed computing, implemented by libraries such as Open MPI and MPICH, and used across HPC applications broadly — not just AI workloads. **NCCL** is a GPU-specific library, scoped to collective and point-to-point communication between GPUs, built to be topology-aware in exactly the way Chapter 15 described.

NCCL is **not** "MPI for GPUs," and it does not replace MPI. Distributed GPU applications aren't built one single way: some stacks use MPI for process launch and general-purpose message passing, some use NCCL specifically for GPU-to-GPU collective communication, and some use both together alongside other process-management components, depending on the framework and deployment. Where both appear in the same stack, a common pattern is MPI (or an MPI-like launcher) handling process coordination across nodes, while NCCL handles the GPU-to-GPU collective operations inside the training loop — but this is a

common pattern, not a universal architecture, and this book does not assume it's the only way distributed GPU communication is built.

23.7 Transport Selection

NCCL selects a transport for each pair of communicating GPUs based on what's actually available between them: NVLink or NVSwitch if present (Chapter 15), otherwise PCIe within the node, and InfiniBand or RoCE (Chapter 10) for communication between nodes. This selection is topology-driven and mostly automatic — NCCL inspects what's physically available and chooses accordingly, rather than requiring an application to specify the transport directly.

23.8 Distributed AI Workload Context

In a distributed training job, NCCL typically operates underneath a higher-level training framework, handling the GPU-to-GPU (and node-to-node) data movement the framework's distributed-training logic requires — most commonly, AllReduce operations combining gradients computed independently on each GPU. This book covers the communication layer NCCL provides; it does not cover training-loop design or ML framework internals, which are outside its infrastructure-operations scope.

23.9 Common Environment Variables, Introduced

A small number of environment variables control NCCL's behavior and diagnostics — most notably `NCCL_DEBUG`, which controls how much diagnostic logging NCCL produces (Chapter 24 covers using this operationally). This chapter names it for recognition; exact

variable names, defaults, and behavior should be verified against NVIDIA's current NCCL documentation, since these can change across NCCL versions.

23.10 Misconceptions and an Illustrative Scenario

NCCL is not "MPI for GPUs" — restated once more because it's the single most important distinction in this chapter. A collective operation is not a single network call — it's a coordinated, multi-step operation across every rank in the communicator. And NCCL's transport selection being automatic doesn't mean it's always optimal for a given job's actual topology — Chapter 15's placement-matters lesson applies here directly.

[Illustrative] The same NCCL-based job runs fast when confined to GPUs on a single node (using NVLink) but noticeably slower once spread across multiple nodes. This isn't necessarily a fault — it reflects NCCL selecting a different, inherently higher-latency transport (InfiniBand/RoCE between nodes, per Chapter 10) than it used within a single node (NVLink). Chapter 24 covers distinguishing this expected transport-related difference from an actual fault.

CHAPTER 24

NCCL

NCCL Troubleshooting

24.1 The Layered Model

An NCCL-related symptom can originate at any of six layers: Application, NCCL, CUDA/Driver, GPU, RDMA/Network, or Fabric. Most "NCCL errors" are actually NCCL reporting a problem from a lower layer — a GPU fault (Chapter 14), a driver mismatch (Chapter 13/14), or a fabric problem (Chapter 12) — rather than a defect in NCCL itself. This chapter's job is helping you tell which layer is actually responsible, and explicitly hands off to Chapter 12 for anything that turns out to be fabric-layer, and to Chapter 14 for anything that turns out to be GPU-layer, rather than re-diagnosing either from scratch.

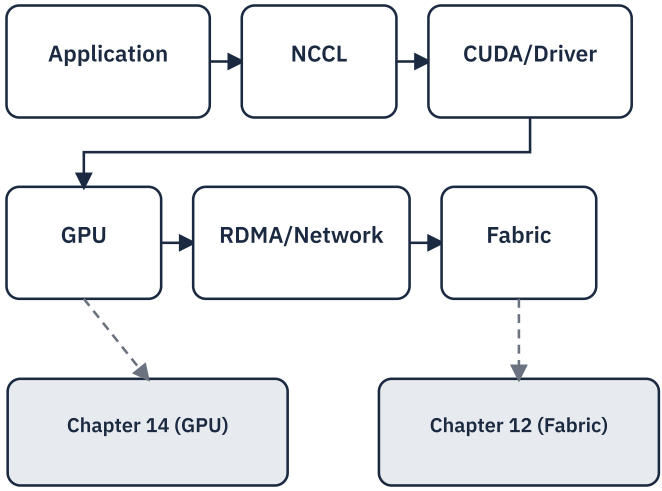


Figure 24.1 — NCCL Diagnostic Flow. NCCL sits between the application and the hardware — most NCCL 'errors' are actually reports from a lower layer. Chapters 12 and 14 own the fabric- and GPU-layer diagnosis this chapter hands off to. *Components: the six-layer chain (Application→NCCL→CUDA/Driver→GPU→RDMA/Network→Fabric) with explicit branch arrows toward Chapter 12 (fabric) and Chapter 14 (GPU) for those layers' full treatment.*

Every scenario in this chapter uses the full ten-section format.

24.2 NCCL Debug Logging

`NCCL_DEBUG` (set to a level such as `WARN` or `INFO`, per current NCCL documentation — exact levels and names should be verified before relying on them) is the primary evidence source for any NCCL problem: it controls how much diagnostic output NCCL produces about its own initialization, transport selection, and communication. Before assuming a specific cause, enabling this logging and re-running is almost always the first "Collect Evidence" step (Chapter 1), the same way `dmesg` / `journalctl` are the first step for a Linux-level symptom (Chapter 4).

24.3 Initialization Failures

NCCL's setup phase involves ranks discovering and confirming each other (a rendezvous) before any actual communication begins. A failure here typically means some rank couldn't be reached by the others — often a network-reachability problem between nodes, a mismatch in how many ranks each process expects, or an environment-variable misconfiguration affecting how ranks discover each other. `NCCL_DEBUG` output at this stage usually names which rank failed to respond, which is the fastest way to narrow from "initialization failed" to "this specific node/process is the problem."

24.4 Hangs and Timeouts

A hang differs from an outright error: every rank is still running, but the collective operation never completes. The key diagnostic question is whether *one specific rank* is the straggler (still computing, or stuck, while every other rank waits on it) or whether the network genuinely isn't delivering data between ranks. Checking each rank's own state (is its GPU actually working, Chapter 14; is its process still making progress) before assuming a network fault avoids chasing a network problem that doesn't exist when the real cause is one slow or stuck participant.

24.5 Rank and GPU Visibility Problems

An NCCL error referencing a rank's GPU can trace back to a `CUDA_VISIBLE_DEVICES` mismatch (Chapter 19) — a rank not seeing the GPU it expects because of a Slurm allocation issue, not an NCCL or hardware fault at all. Checking the job's actual GPU

allocation (Chapter 19's three-way check: requested/allocated/visible) before assuming NCCL itself is broken is the right order of operations here.

24.6 CUDA/Driver Mismatch Surfacing as NCCL Errors

Because NCCL operates through CUDA (Chapter 23), a driver/CUDA version problem (Chapter 14) can surface as an NCCL initialization or runtime error rather than a clearly-labeled CUDA error. If `NCCL_DEBUG` output or the application's error message references CUDA directly, Chapter 14's driver/CUDA compatibility checks are the right next step, not continued NCCL-specific investigation.

24.7 Network/RDMA/InfiniBand Problems — Explicit Hand-Off to Chapter 12

When NCCL's transport selection (Chapter 23) chooses InfiniBand between nodes, a fabric-layer problem — a degraded link, a subnet manager issue, anything Chapter 12 covers — will surface as an NCCL-level symptom (a hang, a timeout, an outright communication error), even though the root cause has nothing to do with NCCL itself. **This chapter does not re-teach InfiniBand fault isolation.** Once evidence points to the network/fabric layer (persistent problems isolated to specific nodes or links, rather than following the application around), the correct next step is Chapter 12's methodology — its layered model, its command pattern, its scenarios — applied directly, not a parallel NCCL-specific version of the same investigation.

24.8 RoCE-Specific Considerations

Where NCCL is running over RoCE (Chapter 10) rather than native InfiniBand, the same lossless-Ethernet requirements (PFC/ECN, Chapter 10) apply, and a poorly configured RoCE fabric can produce NCCL-visible symptoms (timeouts, degraded performance) that trace back to that configuration rather than to NCCL or the GPUs. Consistent with Chapter 10's scope decision, this book keeps PFC/ECN conceptual here as well — recognizing this as a possible cause is the goal, not a PFC/ECN tuning guide.

24.9 Topology Problems

A GPU topology mismatch across nodes (Chapter 15) — one node's GPUs connected differently than another's — can cause inconsistent performance across ranks without any outright error. `nvidia-smi topo -m` (Chapter 15) on each node involved, compared against each other, is the right evidence to gather when performance is inconsistent across otherwise-identical-looking nodes.

24.10 Performance Degradation (Not a Failure)

Not every NCCL problem is a hang or an error — a job that completes but runs slower than expected is a different fault category, needing different evidence: comparing per-node or per-link timing (where available) to find whether the slowdown is uniform (suggesting a systemic transport or topology choice) or isolated to specific ranks/nodes (suggesting a specific degraded component, often traceable to Chapter 12's fabric-layer evidence).

24.11 Common Mistakes

Blaming NCCL for what is actually an InfiniBand fabric problem (Chapter 12) or a GPU topology problem (Chapter 15) is the single most common mistake this chapter is written to prevent — NCCL is frequently the messenger, not the source. Treating a hang as automatically a network problem, without first checking whether one rank is simply stuck or slow, is the second most common one.

24.12 Worked Scenarios

[Illustrative] **Scenario 1 — NCCL Init Hangs on a Multi-Node Job**

SYMPTOM	A multi-node NCCL job hangs during initialization, never reaching the training/communication phase.
INITIAL CHECKS	Enable `NCCL_DEBUG` logging and re-run to see which rank(s) fail to be reached.
COMMANDS	NCCL debug logging output; `ibstat` / `ping`-equivalent reachability checks (Chapter 9/11) between the nodes involved.
OUTPUT INTERPRETATION	The debug log shows most ranks successfully discovering each other, but one specific rank never responds.
POSSIBLE CAUSES	That rank's node is unreachable on the network path NCCL uses; a firewall or configuration difference specific to that node; that rank's process failed to start at all.

FAULT ISOLATION	Checking basic reachability to the specific node named in the log separates a network-layer cause from a process-launch cause.
ROOT CAUSE	In this illustrative case, the node was reachable for management traffic but not on the specific interface NCCL was attempting to use.
RESOLUTION	Correct the network interface configuration (`NCCL_SOCKET_IFNAME` or equivalent, per current NCCL documentation) so NCCL selects the correct, reachable interface.
VALIDATION	Confirm the job initializes successfully across all nodes after the correction.
PREVENTION	Validate NCCL's interface selection as part of new-node onboarding (Chapter 25), not just at first job-failure.

[Illustrative]

Scenario 2 — NCCL Timeout Under Load, Straggler vs. Real Network Fault

SYMPTOM	A previously working multi-node job begins timing out during collective operations under sustained load.
INITIAL CHECKS	Determine whether one specific rank is behind, or whether the slowdown is uniform across all ranks.
COMMANDS	`NCCL_DEBUG` logging, per-rank progress/timing if the application exposes it, `nvidia-smi` on each node (Chapter 13/14).

OUTPUT INTERPRETATION	One rank's GPU shows abnormal behavior (e.g., elevated temperature approaching a throttling point, Chapter 14) while others are normal.
POSSIBLE CAUSES	Thermal throttling on one node slowing that rank specifically; a genuine network problem affecting only that node's path; an application-level imbalance in work distribution.
FAULT ISOLATION	The GPU-level evidence (throttling on one specific node) points away from the network entirely.
ROOT CAUSE	**Thermal throttling on one node was slowing that rank enough to trigger a timeout across the whole collective, which waits for every rank.**
RESOLUTION	Address the thermal issue (Chapter 6/25) on the affected node — a hardware/facility fix, not an NCCL or network configuration change.
VALIDATION	Confirm the job completes without timeouts once the affected node's thermal behavior is corrected.
PREVENTION	Monitor per-node GPU temperature trends (Chapter 21) so a degrading cooling condition is caught before it causes a job-visible timeout.

[Illustrative]

Scenario 3 — An RDMA-Layer Problem Surfacing as an NCCL Error

SYMPTOM	A multi-node NCCL job intermittently fails with a communication error, with no clear pattern by job or code version.

INITIAL CHECKS	Check whether the failures correlate with specific nodes or links rather than specific jobs.
COMMANDS	`NCCL_DEBUG` logging; Chapter 12's InfiniBand fault-isolation toolset (`ibstat`, `perfquery`, `iblinkinfo`) once a node/link pattern is suspected.
OUTPUT INTERPRETATION	`NCCL_DEBUG` output points to a transport-level failure on a specific rank pair rather than a collective-logic error, and every failing job shares one node in common regardless of which code or dataset it runs.
POSSIBLE CAUSES	A degraded link or port on the fabric (Chapter 12); a subnet manager issue affecting a subset of nodes.
FAULT ISOLATION	Running Chapter 12's `perfquery` / `iblinkinfo` checks against the common node's ports finds elevated error counters on one port, isolating the fault to that port's physical layer rather than to NCCL or the application.
ROOT CAUSE	**A degrading port on the common node's HCA — an RDMA/fabric-layer fault, not an NCCL defect, that only became visible through NCCL's own error reporting.**
RESOLUTION	Hand off to Chapter 12's methodology once the fabric layer is implicated — this chapter's role ends at correctly identifying that hand-off point.

VALIDATION	Confirm the NCCL job completes reliably once the underlying fabric issue (diagnosed and resolved via Chapter 12) is addressed.
PREVENTION	Correlate NCCL failure patterns with fabric-layer monitoring (Chapter 21/22) so a recurring node/link pattern is caught by fabric monitoring before it's reported as an application problem.

[Illustrative]

Scenario 4 — Wrong Network Interface Selected on a Multi-NIC Node

SYMPTOM	A multi-node job runs, but at far lower throughput than the fabric should support.
INITIAL CHECKS	Check which network interface NCCL actually selected, via debug logging.
COMMANDS	`NCCL_DEBUG` logging (interface selection is typically reported at this level).
OUTPUT INTERPRETATION	NCCL selected a lower-bandwidth management-network-style interface rather than the high-bandwidth fabric interface (Chapter 2's management-vs-data-network distinction, made concrete here).
POSSIBLE CAUSES	Multiple network interfaces present on the node with ambiguous or default interface-selection behavior; missing or incorrect interface-selection configuration.
FAULT ISOLATION	Confirmed directly by the debug log's reported interface choice.

ROOT CAUSE	NCCL's automatic interface selection picked the wrong interface on a node with more than one candidate.
RESOLUTION	Explicitly configure the correct interface (per current NCCL documentation's interface-selection variable) rather than relying on automatic selection on multi-NIC nodes.
VALIDATION	Confirm throughput matches expectations for the fabric once the correct interface is selected.
PREVENTION	Explicitly set interface selection as a standard part of multi-NIC node configuration, rather than relying on automatic detection.

[Illustrative]

Scenario 5 — GPU Topology Mismatch Across Nodes Causing Inconsistent Performance

SYMPTOM	A multi-node job's performance varies noticeably depending on which specific nodes it lands on.
INITIAL CHECKS	Compare `nvidia-smi topo -m` output (Chapter 15) across the nodes involved.
COMMANDS	`nvidia-smi topo -m` on each node.
OUTPUT INTERPRETATION	One node's GPU topology differs from the others — e.g., a GPU pair connected via NVLink on most nodes but only via PCIe on one.
POSSIBLE CAUSES	A hardware configuration difference between nodes believed to be identical; a seating/link-training issue (Chapter 14's PCIe scenario) affecting one node's topology.

FAULT ISOLATION	The topology comparison directly identifies the outlier node.
ROOT CAUSE	One node's NVLink connection between two GPUs had degraded to a PCIe-only path, silently changing its topology from its peers'.
RESOLUTION	Investigate and remediate the specific node's hardware (Chapter 14/25) rather than treating this as an application- or NCCL-level issue.
VALIDATION	Confirm the corrected node's topology matches its peers, and job performance becomes consistent regardless of node placement.
PREVENTION	Include topology verification (not just GPU presence) in fleet-consistency checks (Chapter 6/25), since "identical" nodes can silently diverge.

[Illustrative]

Scenario 6 — Gradual Multi-Node Performance Regression, Not a Code Change

SYMPTOM	A recurring multi-node job's runtime has been gradually increasing over several weeks, with no corresponding code change.
INITIAL CHECKS	Compare current per-node/per-link timing evidence against historical baselines, if available (Chapter 21's capacity-monitoring philosophy applied here).
COMMANDS	`NCCL_DEBUG` logging over several runs; Chapter 12's `perfquery` for fabric-wide error/utilization trends.

OUTPUT INTERPRETATION	One fabric link shows a slowly increasing error rate over the same period as the performance regression.
POSSIBLE CAUSES	A degrading cable/optic (Chapter 12); increasing contention from other traffic sharing the same link over time.
FAULT ISOLATION	The correlated timing between the error-rate trend and the performance regression points toward the degrading link specifically.
ROOT CAUSE	**A slowly degrading cable was introducing increasing retransmission/error overhead, gradually slowing every job whose NCCL traffic crossed that link.**
RESOLUTION	Replace the degrading cable (Chapter 12/25) — a fabric-hardware fix, not an application or NCCL change.
VALIDATION	Confirm the job's runtime returns to its historical baseline once the link is replaced.
PREVENTION	Trend fabric error counters over time (Chapter 21/22), not just check them at a single point, so a slow degradation is caught before it becomes a multi-week performance mystery.

PART 10

HPC Data Center Operations

Chapter 25

CHAPTER 25

DATA CENTER

HPC Data Center Operations

25.1 Role of Data Center Operations in HPC

Chapter 6 introduced server hardware, firmware baselines, and the concepts behind burn-in, validation, node recovery, and RMA — deliberately at the level of *what and why*, deferring *how* to this chapter. This chapter is that operational depth, and it's deliberately broad, because it's the bridge between everything this book has covered as infrastructure (compute, storage, fabric) and the physical facility that has to actually support it.

Compute, network, and storage don't operate independently of the facility around them — a rack's power and cooling capacity constrains how much compute density it can actually hold (Chapter 6's GPU-node consequence, made concrete here); a cabling mistake affects fabric behavior (Part 5); a cooling failure affects every layer above it at once. This is also why physical faults routinely present as software symptoms: a GPU throttling under a cooling problem (Chapter 14) looks like a performance regression until someone checks airflow; a node repeatedly rebooting under a marginal power connection looks like a firmware bug until someone checks the physical connection. Data center operations is where the physical layer either supports every chapter before this one, or quietly undermines it.

25.2 Rack Planning and Layout

A rack's usable capacity is expressed in rack units (U), and equipment placement within it isn't just a matter of fitting — serviceability matters too: a node that can't be pulled, opened, or reseated without disturbing its neighbors costs more time during every future repair than one placed with clearance in mind. Cable paths (front, rear, or overhead, depending on site design) need to be planned before equipment goes in, not improvised afterward. Weight and loading are real constraints, particularly for GPU-dense racks, which are often meaningfully heavier than an equivalent rack of general-purpose compute nodes — a rack's floor-loading limit is a real number, not a formality.

Power and network cabling are commonly kept physically separated within a rack, per many sites' internal standards — reducing the chance of one type of work (a network change) accidentally disturbing the other (a power connection). This book doesn't prescribe one universal rack layout; layouts vary by site standards, facility constraints, and equipment mix, and a competent infrastructure engineer reads and follows the site's own standard rather than assuming a generic one applies.

25.3 Power Architecture

Most HPC-grade servers ship with redundant power supply units (PSUs) — commonly two, each capable of powering the node alone if the other fails. Redundancy at the PSU level only matters if it's matched by redundancy upstream: a common and genuinely costly mistake is connecting both of a node's PSUs to the same power feed (the same PDU, ultimately the same upstream circuit or source) — which means a single upstream failure still takes the node down, even though it has two PSUs. Real redundancy means each PSU connects to a genuinely independent feed — commonly

described as A/B power feeds, where "A" and "B" trace back to separate upstream sources, not just separate power strips plugged into the same outlet.

PDUs (power distribution units) deliver power to a rack's equipment, and branch circuits — the specific circuit a PDU or group of outlets draws from — have their own capacity limits; overloading a branch circuit by not accounting for the actual draw of a fully populated, GPU-dense rack is a real and avoidable planning mistake, not just a theoretical concern. This book covers the concepts an infrastructure engineer needs to recognize — is this rack's power actually redundant, or does it only look that way — without going into circuit-level electrical design, which belongs to a qualified electrical engineer, not this book.

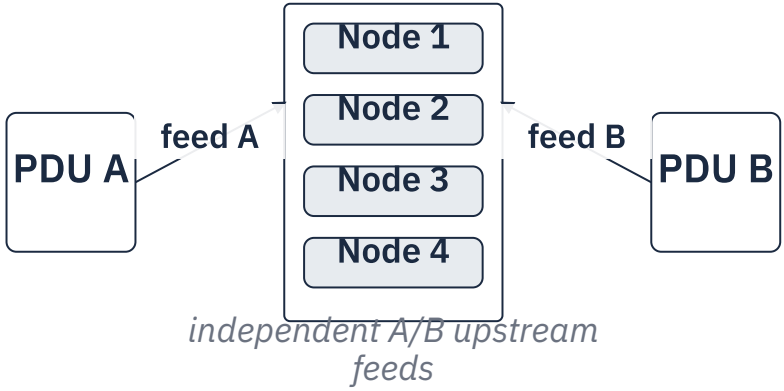


Figure 25.1 — Rack and Power Distribution Diagram. Physical layout decisions — density, redundancy, cabling discipline — have direct consequences for how fast a fault can be isolated later, and redundant PSUs only protect against a failure if their upstream feeds are genuinely independent. *Components: a rack with nodes, two independent PDUs (A/B feeds, tracing to separate upstream sources) feeding it, and a labeled cabling path.*

25.4 Cooling and Airflow

Data centers commonly use a hot-aisle/cold-aisle arrangement: racks organized so that cold air intake on one side and hot air exhaust on the other don't mix, which keeps cooling efficient. Every server has a defined inlet (cold air in) and exhaust (hot air out) side, and equipment installed facing the wrong way, or a blanking panel left out of an empty rack slot, lets hot exhaust air recirculate into a cold-air intake — a local cooling failure that doesn't show up as a facility-wide alarm, just as one unexplained hot node.

GPU nodes generate substantially more heat per rack unit than general-purpose compute nodes (Chapter 6), which makes airflow discipline matter more, not less, in GPU-dense racks — the margin for a cooling mistake is smaller. Fan and thermal alarms (BMC-visible, Chapter 6) are typically the first automated signal something's wrong, and a rising temperature trend, left unaddressed, has a direct relationship to hardware reliability and lifespan — though this book doesn't cite specific numerical thresholds, since acceptable operating ranges vary by hardware generation and should be sourced from the specific hardware's own documentation rather than assumed universal.

25.5 HPC Cabling

An HPC node commonly has several distinct cable types running to it, each serving a different network or purpose: a management/BMC connection (Chapter 6), general Ethernet, InfiniBand or other fabric connections (Part 5), and storage-specific connections where a site's storage network is physically separate from its compute fabric. Keeping these visually and physically distinguishable — through consistent labeling and,

where a site's standards call for it, color-coding — is what makes Chapter 12's fault-isolation methodology fast in practice rather than slow.

Cable length matters beyond simply "long enough to reach" — excess slack has to be managed (not left as a tangled loop that obstructs airflow or another technician's access), and for fiber connections, bend radius matters: bending fiber too sharply degrades or breaks it, a physical constraint worth knowing even without going into optical engineering depth. For fabric connections specifically, three general categories are worth recognizing by name: **DAC** (Direct Attach Copper — a fixed copper cable assembly, typically for shorter runs), **AOC** (Active Optical Cable — an optical cable with the transceiver electronics built in, for longer runs than DAC supports), and standard **fiber with separate transceivers** — each with different length/cost/power tradeoffs a site's design accounts for, at a level this book doesn't need to go deeper than recognition.

Every cable, at both ends, should be documented — not just labeled physically, but recorded somewhere queryable, so "what connects to what" is a lookup, not an investigation.

Table 25.1 — Example Cable Documentation Record

Cable ID	Type	Endpoint A	Endpoint B	Length	Notes
IB-R12-014	InfiniBand (AOC)	Node `gpu-node-014`, HCA port 1	Switch `ib-sw-03`, port 22	5m	Installed 2026-03
ETH-R12-014	Ethernet (Cat6)	Node `gpu-node-014`, management NIC	Switch `mgmt-sw-01`, port 14	3m	—

PWR-R12-014-A	Power (A feed)	Node `gpu-node-014`, PSU 1	PDU `pdu-a-rack12`	—	A-feed
PWR-R12-014-B	Power (B feed)	Node `gpu-node-014`, PSU 2	PDU `pdu-b-rack12`	—	B-feed, independent upstream (Section 25.3)

This is illustrative structure, not a mandated schema — the point is that each record answers "what connects to what, and where" without requiring anyone to trace a physical path by hand.

25.6 Installation and Commissioning Workflow

A new node's path from arrival to production follows a conceptual sequence, though the specific tooling implementing each step is site-specific:

Rack readiness (Section 25.2 confirmed) → **power validation** (confirming both feeds are live and independent, Section 25.3) → **physical installation** → **cabling** (Section 25.5, documented as it's connected, not afterward) → **BMC/network configuration** (Chapter 6) → **firmware baseline application** (Chapter 6) → **OS provisioning** (Chapter 2's provisioning concept, applied) → **hardware validation** (confirming the node's own hardware is healthy, Chapter 6) → **network/fabric validation** (confirming the node is correctly visible on the fabric, Part 5) → **storage validation** (confirming expected mounts are present and working, Part 4) → **scheduler integration** (confirming the node is visible and correctly configured in Slurm,

Part 7) → **burn-in** (Section 25.7) → **workload validation** (a representative real job, not just synthetic tests) → **handover** to production.

This sequence is conceptual — the specific tools and exact ordering some steps can be safely parallelized in vary by site. What doesn't vary is the principle: skipping a step doesn't remove the risk it was meant to catch, it just moves that risk into production, where it's more expensive to find.

25.7 Burn-In and Validation

Burn-in exists because a node passing a quick power-on check and a node holding up under sustained, real workload are different claims — a node should not enter production simply because it boots and looks correct at a glance. Operationally, burn-in means running a sustained, deliberately demanding workload — CPU- and memory-intensive at minimum, GPU-intensive as well on GPU nodes (Chapter 13's DCGM diagnostics are the typical tool here) — for a defined duration, checked against a defined acceptance threshold.

Validation is broader than burn-in alone: it also means checking storage connectivity actually works as expected (not just that a mount command succeeded), checking network/fabric visibility and health (Part 5), and confirming the firmware/BIOS baseline (Chapter 6) actually matches the fleet standard rather than assuming the provisioning process applied it correctly. Monitoring during validation — not just a pass/fail result at the end — matters too, since a node that "passes" but showed a concerning trend partway through (a temperature spike, a transient error) is different from one that was clean throughout. This book does not prescribe one universal burn-in test duration or threshold; what counts as sufficient varies by hardware generation, workload type, and site risk tolerance.

25.8 RMA and Hardware Replacement

Before replacing anything, evidence should support the failure determination — an Xid code (Chapter 14), an ECC error trend (Chapter 6/14), a burn-in failure (Section 25.7), or equivalent — not just a suspicion. Once confirmed, the component's identification (serial number, exact slot/position) needs to be recorded precisely enough that the correct physical part is the one actually removed and returned — a mismatch here (removing the wrong component) is its own avoidable failure mode.

The replacement procedure itself should include: physically swapping the component, confirming the new component is correctly recognized (the same visibility checks from its relevant chapter — Chapter 6, 13, or 14 depending on what was replaced), and re-running validation (Section 25.7) rather than assuming a replacement is automatically good. Firmware and configuration drift is a real risk here too — a replacement part can arrive with a different firmware version than the fleet baseline (Chapter 6), which needs to be caught and corrected as part of the replacement, not discovered later as a mysterious inconsistency. Finally, the failed component is returned to the vendor following that vendor's (or the site's own) RMA process, with the documentation from Section 25.9 attached.

Table 25.2 — RMA Documentation Checklist

Item	Why It Matters
Component identification (serial number, slot/position)	Ensures the correct part is returned and replaced
Failure evidence (error codes, logs, burn-in results)	Supports the RMA claim and informs the vendor's diagnosis

Timeline (when first observed, when confirmed)	Supports pattern analysis across the fleet later
Firmware/configuration state at time of failure	Rules out a configuration cause before blaming hardware
Post-replacement validation result	Confirms the replacement actually resolved the issue before closing the record

25.9 Asset and Configuration Tracking

Asset tracking is what makes firmware-baseline enforcement (Chapter 6) and RMA history (Section 25.8) usable as fleet-wide data rather than anecdotal memory. A useful asset record ties together physical location, identity, and configuration state in one place:

Table 25.3 — Example Asset Record

Field	Example Value
Rack position	Rack 12, U18–U20
Hostname	`gpu-node-014`
Serial number	(vendor-assigned, node chassis)
Asset ID	`AST-2026-00842`
MAC address(es)	Management NIC, data NIC(s)
HCA/port identifiers	HCA GUID; connected switch/port (Table 25.1)
Firmware/BIOS version	Baseline version and date applied (Chapter 6)
Replacement history	Prior RMA records, if any (Section 25.8)

Without this kind of record, a question like "how many nodes still have the outdated firmware version" has no reliable answer, and a fleet-wide pattern (Chapter 6's firmware-drift scenario) only becomes visible by accident rather than by design.

25.10 Physical Troubleshooting Decision Tree

A physical-layer problem can manifest at any higher layer — a power issue can look like a boot failure, a cabling issue can look like a fabric fault, a cooling issue can look like a performance regression. The reverse discipline matters just as much: before assuming a symptom is physical, walk the layers systematically.

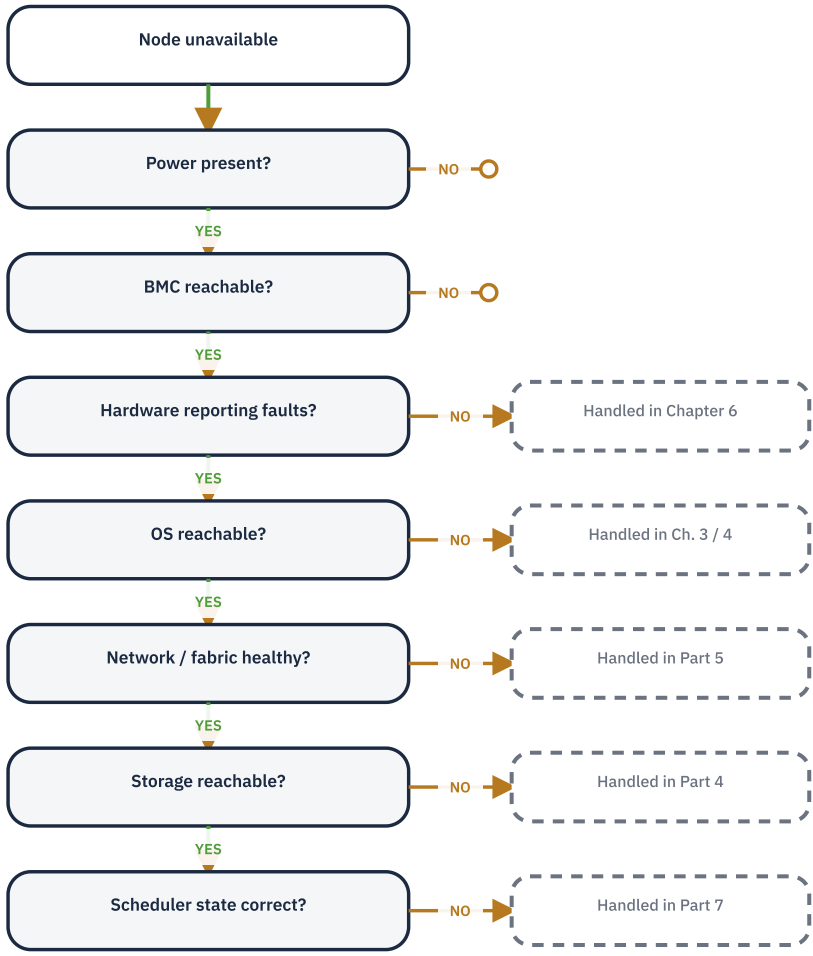


Figure 25.2 — Physical Troubleshooting Decision Tree. Work down from the physical layer before assuming a software-level fault — but don't stop at the first 'yes' either, since a symptom can still originate further down the chain. *Components: "Node unavailable" at the top, branching sequentially: Is power present? → Is BMC reachable? → Is hardware reporting faults (Chapter 6)? → Is the OS reachable (Chapter 3/4)? → Is the network/fabric healthy (Part 5)? → Is storage reachable (Part 4)? → Is scheduler state correct (Part 7)? Each "no" branch points to that layer's own chapter; each "yes" branch continues down the chain.*

The value of this sequence is that it's cheap to walk in order and expensive to skip: confirming power and BMC reachability takes a moment and rules out (or confirms) the least software-dependent layers first, before any time is spent on OS-, fabric-, storage-, or scheduler-level investigation that would be wasted if the real answer was "the node never had power."

25.11 Safety

Physical work carries real risk, and the practices here are non-negotiable baseline expectations, not suggestions: follow electrical safety practices around power work; use ESD (electrostatic discharge) precautions when handling internal components; use correct lifting/handling technique for heavy equipment (GPU servers are frequently heavier than general-purpose ones); treat recently-running hardware as hot until confirmed otherwise; manage cables so they don't create a trip hazard or an airflow obstruction; and follow change-control procedures so physical work is tracked, not improvised. This book does not provide detailed hazardous-procedure instructions — site-specific safety training and procedures take precedence over anything general written here, and they should never be bypassed for the sake of speed.

25.12 Cross-Layer Troubleshooting Scenarios

[Illustrative] A fault-isolation effort during an active incident takes noticeably longer than it should because the affected rack's cabling wasn't labeled according to standard, and the specific physical connection in question has to be traced by hand rather than looked up — exactly the cost Section 25.6's cabling-documentation discipline is meant to avoid.

[Illustrative] A node passes its full burn-in and validation process cleanly, and only begins showing a hardware-related failure pattern weeks into real production workload — a case where burn-in's demanding-but-finite test didn't reproduce a condition that only emerged under sustained, real-world usage patterns.

PART 11

Practical Troubleshooting Playbooks

Chapter 26, Chapter 27

CHAPTER 26

TROUBLESHOOTING

Practical HPC Troubleshooting Playbooks I — Infrastructure & Fabric

26.1 How to Use This Playbook

Every chapter so far has taught troubleshooting within one domain. Real incidents don't always announce their domain up front, and some of the most time-consuming problems are the ones that are intermittent, capacity-driven, maintenance-induced, or caused by configuration drift rather than an acute failure. This chapter and Chapter 27 apply the book's full methodology to scenarios chosen specifically for that reason — none of them repeat a failure mode already used in Chapters 4, 8, 12, 20, 24, or 25. This chapter covers infrastructure and fabric; Chapter 27 covers compute and workload.

Symptom Index

Symptom	Likely Domain(s)	Scenario	See Also
Intermittent node reboots	Linux/hardware	26.2 #1	Ch.4, 6
Fleet nodes behaving inconsistently	Linux/hardware	26.2 #2	Ch.3, 6

Alert with no matching symptom	Linux/hardware	26.2 #3	Ch.21
Outage after an ignored capacity warning	Linux/hardware	26.2 #4	Ch.21
Storage capacity surprise	Storage	26.3 #5	Ch.7, 21
Storage impact after a maintenance change	Storage	26.3 #6	Ch.8, 25
Intermittent slow storage path	Storage	26.3 #7	Ch.8
Multi-layer fabric failure	InfiniBand/fabric	26.4 #8	Ch.9, 11, 12
Traffic-pattern-dependent link issue	InfiniBand/fabric	26.4 #9	Ch.12
Partition/PKey drift	InfiniBand/fabric	26.4 #10	Ch.9, 11
Fabric expansion asymmetry	InfiniBand/fabric	26.4 #11	Ch.12, 22
Wrong node serviced during maintenance	Data center/hardware	26.5 #12	Ch.25
Wrong part RMA'd	Data center/hardware	26.5 #13	Ch.25
Firmware drift resurfacing	Data center/hardware	26.5 #14	Ch.6, 25
"Node unavailable" — provisioning, not hardware	Ambiguous	26.6 #15	Ch.2, 3

"Node unavailable" — fabric, not hardware	Ambiguous	26.6 #16	Ch.12

26.2 Linux and Hardware Cross-Cutting Scenarios

[Illustrative]

Scenario 1 — Intermittent Node Reboot Pattern

SYMPTOM

A specific node reboots unexpectedly every few weeks, with no obvious trigger.

INITIAL CHECKS

Check `journalctl`/BMC System Event Log (Chapter 6) around each reboot's timestamp for a common signature.

COMMANDS

`journalctl` (boot logs), BMC SEL review.

OUTPUT INTERPRETATION

The BMC SEL shows a thermal-threshold event immediately preceding each reboot, at roughly the same time of day across occurrences — a pattern `journalctl` alone (which only sees the OS-side abrupt restart, not the cause) wouldn't reveal.

POSSIBLE CAUSES

A thermal event triggering a protective shutdown; a power delivery issue; a kernel panic with an unclear root cause.

FAULT ISOLATION

The recurring time-of-day correlation points toward an environmental/cooling condition (e.g., a shared cooling load peak) rather than a random hardware fault, which the BMC SEL's thermal-threshold entries directly confirm.

ROOT CAUSE	The node's cooling margin is thin enough that a recurring, predictable thermal load (shared with nearby equipment) pushes it over its protective shutdown threshold.
RESOLUTION	Address the node's cooling margin (airflow, filter/dust check, or rack-level cooling load rebalancing per Chapter 25) rather than treating each reboot as an isolated event.
VALIDATION	Confirm the node runs through several of its previous failure intervals without recurrence.
PREVENTION	Track reboot timestamps fleet-wide (Chapter 21) to catch a recurring pattern before it's dismissed as one-off.

[Illustrative]

Scenario 2 — Configuration Drift Across a Node Fleet

SYMPTOM	Jobs behave inconsistently across nodes that are supposed to be identical.
INITIAL CHECKS	Compare configuration (package versions, environment module availability, Chapter 3) across a sample of nodes.
COMMANDS	`dnf list installed` (Chapter 3), `module avail` (Chapter 3), comparing outputs across nodes.
OUTPUT INTERPRETATION	One subset of nodes has a different package or module version than the rest.
POSSIBLE CAUSES	An out-of-band manual change on some nodes; a provisioning update (Chapter 2) that didn't reach every node.

FAULT ISOLATION	Comparing against the provisioning system's intended state identifies which nodes drifted.
ROOT CAUSE	A subset of nodes missed a provisioning update due to being offline during the update window.
RESOLUTION	Re-run provisioning against the affected nodes to bring them back to the fleet baseline.
VALIDATION	Confirm configuration matches across the full fleet after remediation.
PREVENTION	Add a periodic fleet-wide configuration-consistency check rather than relying on provisioning always reaching every node on the first attempt.

[Illustrative]

Scenario 3 — Monitoring Alert with No Matching Real Symptom

SYMPTOM	A monitoring alert fires repeatedly, but no corresponding job failure or user complaint ever follows.
INITIAL CHECKS	Confirm the alert's underlying metric and threshold, and whether the threshold was ever validated against real impact.
COMMANDS	Review the alerting rule/threshold (Chapter 21) and the historical metric data behind it.
OUTPUT INTERPRETATION	The metric does cross the configured threshold, but the threshold doesn't correspond to any actual operational problem at that level.

POSSIBLE CAUSES	A threshold set arbitrarily rather than based on observed impact; a metric that isn't actually a good proxy for the condition it's meant to catch.
FAULT ISOLATION	Comparing threshold-crossing events against actual incident history over time confirms the mismatch.
ROOT CAUSE	**The alert threshold was never tuned against real-world impact data.**
RESOLUTION	Re-tune or replace the alert with one based on actual observed correlation to real problems.
VALIDATION	Confirm the revised alert fires only when a real, actionable condition exists, over a subsequent observation period.
PREVENTION	Review alert thresholds periodically against actual incident correlation (Chapter 21's alerting philosophy), not just set once and forget.

[Illustrative]

Scenario 4 — Capacity-Exhaustion Warning Ignored Until an Outage

SYMPTOM	A shared resource (e.g., a filesystem, Chapter 7) runs out of capacity, causing widespread job failures.
INITIAL CHECKS	Check whether a capacity-trend warning existed before the outage.
COMMANDS	`df` (Chapter 3), historical capacity-trend data (Chapter 21).

OUTPUT INTERPRETATION	Capacity trend data shows a steady, predictable approach toward exhaustion over the preceding weeks.
POSSIBLE CAUSES	A capacity warning existed but wasn't acted on; no capacity-trend monitoring existed at all.
FAULT ISOLATION	Reviewing whether the warning existed and was visible to someone responsible confirms which case applies.
ROOT CAUSE	A capacity-trend warning existed but wasn't routed to anyone positioned to act on it before exhaustion.
RESOLUTION	Free capacity immediately to restore service, then address the routing gap.
VALIDATION	Confirm capacity returns to a healthy margin and stays there under continued monitoring.
PREVENTION	Ensure capacity-trend warnings (Chapter 21) are routed to someone who can act on them well before exhaustion, not just logged.

26.3 Storage Cross-Cutting Scenarios

[Illustrative] **Scenario 5 — Storage Capacity Growth Outpacing Projection**

SYMPTOM	A shared filesystem's capacity is consumed faster than the site's growth projections anticipated.
INITIAL CHECKS	Review recent usage-growth rate against the original projection basis.

COMMANDS	`df` (Chapter 3), historical capacity trend (Chapter 21).
OUTPUT INTERPRETATION	Growth rate has meaningfully accelerated relative to the historical trend the projection was based on.
POSSIBLE CAUSES	A new workload type with much higher storage demand than prior workloads; a change in how existing workloads use storage (e.g., more frequent checkpointing).
FAULT ISOLATION	Identifying which workload(s) or user(s) account for the accelerated growth narrows the cause.
ROOT CAUSE	**A new workload's checkpointing frequency was significantly higher than assumed in the original capacity projection.**
RESOLUTION	Adjust capacity planning to reflect the new workload's actual demand, and/or work with the workload owner on checkpoint retention policy.
VALIDATION	Confirm the revised growth projection tracks actual usage going forward.
PREVENTION	Revisit capacity projections when a new workload type is onboarded, rather than assuming historical growth rates still apply.

[Illustrative]

Scenario 6 — Maintenance-Window Storage Change Causing Unexpected Client Impact

SYMPTOM	A planned storage maintenance change causes broader client-side impact than expected.

INITIAL CHECKS	Compare the change's actual scope against its planned scope.
COMMANDS	Client-side mount status checks (Chapter 3/8) across affected nodes.
OUTPUT INTERPRETATION	A set of nodes not on the maintenance's planned client list shows the same stale-mount symptom as the nodes that were planned for — those nodes mount the changed storage path indirectly, through a second, undocumented mount dependency.
POSSIBLE CAUSES	An underestimated dependency between the changed component and clients not originally identified as affected.
FAULT ISOLATION	Tracing each affected node's mount table back to the changed component confirms the indirect dependency as the reason the impact extended past the planned client list.
ROOT CAUSE	**The maintenance plan's client-impact list was built from the direct/documented mount relationships only, missing an indirect dependency that wasn't recorded anywhere.**
RESOLUTION	Remount the affected path on the unplanned clients to restore access, and document the previously-missing dependency.
VALIDATION	Confirm all affected clients recover normal access post-remediation.
PREVENTION	Map dependencies more thoroughly before planned maintenance, including indirect/less obvious client relationships.

[Illustrative]

Scenario 7 — Intermittent Slow-Path Issue Reproducing Only Under a Specific Access Pattern

SYMPTOM	A storage slowdown is reported intermittently, and initial investigations find nothing wrong.
INITIAL CHECKS	Gather detail on exactly what access pattern was running when the slowdown was observed, each time it recurred.
COMMANDS	`iostat` / `vmstat` (Chapter 4) captured specifically during a reported episode, not after the fact.
OUTPUT INTERPRETATION	The slowdown correlates with a specific, infrequent access pattern (e.g., a particular batch job's directory-scan behavior) rather than being random.
POSSIBLE CAUSES	A metadata-heavy access pattern (Chapter 8) that only a specific, infrequently-run job triggers.
FAULT ISOLATION	Correlating the slowdown's timing against the job schedule identifies the specific trigger.
ROOT CAUSE	**An infrequently-run job's directory-scan pattern created metadata pressure (Chapter 8) that affected other concurrent storage access.**
RESOLUTION	Adjust the job's access pattern, or schedule it to avoid overlapping with sensitive concurrent workloads.
VALIDATION	Confirm no further slowdown reports correlate with that job's subsequent runs.

PREVENTION	When a symptom is intermittent, capture evidence *during* the next occurrence rather than only investigating after the fact once the evidence is gone.

26.4 InfiniBand and Fabric Cross-Cutting Scenarios

[Illustrative]

Scenario 8 — Multi-Layer Failure Combining a Link Issue with a Subnet Manager Response Delay

SYMPTOM	After a link failure, fabric recovery takes noticeably longer than expected.
INITIAL CHECKS	Check both the link's own state (Chapter 12) and the subnet manager's response/reconvergence time.
COMMANDS	`iblinkinfo`, `perfquery` (Chapter 11/12), subnet manager status checks (Chapter 11).
OUTPUT INTERPRETATION	The link itself recovers quickly, but the subnet manager takes much longer than expected to update routing accordingly.
POSSIBLE CAUSES	A subnet manager under unusual load at the time; a subnet manager configuration affecting reconvergence speed.
FAULT ISOLATION	Confirming the link-level recovery time separately from the routing-update time isolates the delay to the SM layer specifically.

ROOT CAUSE	The subnet manager's reconvergence behavior, not the link itself, accounted for most of the recovery delay.
RESOLUTION	Investigate and address the SM-layer factor (load, configuration) separately from the original link issue.
VALIDATION	Confirm a subsequent, similar link event recovers within an acceptable time.
PREVENTION	Measure fabric recovery time as a combined metric (link recovery + SM reconvergence), not just link state alone.

[Illustrative]

Scenario 9 — Intermittent Link Degradation Reproducing Only Under a Specific Traffic Pattern

SYMPTOM	A specific link shows error-counter increases only during certain job types, not consistently.
INITIAL CHECKS	Correlate error-counter timing (Chapter 12's `perfquery`) against the job schedule on nodes using that link.
COMMANDS	`perfquery`, correlated with job scheduling data (Chapter 17/20).
OUTPUT INTERPRETATION	Error counts rise specifically during a job type with an unusual, bursty communication pattern.
POSSIBLE CAUSES	A marginal physical connection that only manifests errors under a specific traffic characteristic (e.g., burst congestion) rather than under steady load.

FAULT ISOLATION	The correlation with a specific job type, rather than overall utilization, narrows this to a traffic-pattern-sensitive fault.
ROOT CAUSE	A marginal cable/optic (Chapter 12) that behaved acceptably under steady load but showed errors under bursty traffic.
RESOLUTION	Replace the marginal component even though it "passes" under typical steady-state testing.
VALIDATION	Confirm the error counts stay flat under the same bursty job type post-replacement.
PREVENTION	Include bursty/worst-case traffic patterns in fabric health testing, not just steady-state checks.

[Illustrative]

Scenario 10 — Configuration Drift in Partition/PKey Setup

SYMPTOM	A routine fabric audit reveals partition/PKey assignments that don't match documented policy.
INITIAL CHECKS	Compare current PKey assignments (Chapter 9/11) against the documented intended configuration.
COMMANDS	Subnet manager configuration review (Chapter 11).
OUTPUT INTERPRETATION	A specific subset of ports carries a PKey assignment that doesn't appear anywhere in the current documented policy, though the fabric itself has been functioning without any reported problem.

POSSIBLE CAUSES	A manual change made during a past incident that was never reverted or documented.
FAULT ISOLATION	Cross-referencing the undocumented PKey against past incident/change records traces it to a temporary workaround applied during an earlier fault-isolation effort that was never cleaned up afterward.
ROOT CAUSE	A temporary configuration change made during incident response became permanent by omission — the change worked, so no one revisited whether it should be reverted or formally documented.
RESOLUTION	Either formally document the change as intended policy (if it's still needed) or revert it to match documented policy (if it was genuinely meant to be temporary).
VALIDATION	Confirm the corrected configuration matches documented policy and passes a follow-up audit.
PREVENTION	Treat fabric configuration audits as routine (Chapter 11's operational checklists), not just reactive to a reported problem.

[Illustrative]

Scenario 11 — Fabric Expansion Introducing an Asymmetry

SYMPTOM	After adding new switches/nodes to an existing fabric, some existing paths show elevated utilization relative to others.
INITIAL CHECKS	Compare per-link utilization (Chapter 12) before and after the expansion.

COMMANDS	`perfquery` across the fabric, before/after comparison.
OUTPUT INTERPRETATION	A subset of links, unrelated to the newly added hardware, show disproportionately higher utilization post-expansion.
POSSIBLE CAUSES	The expansion changed routing in a way that concentrated more traffic onto existing paths than intended; the new topology wasn't fully symmetric with the old one.
FAULT ISOLATION	Comparing the routing computed before and after the expansion (via topology inspection, Chapter 11) identifies the specific change.
ROOT CAUSE	**The fabric expansion, while functionally correct, introduced a routing asymmetry that concentrated traffic onto a subset of pre-existing links.**
RESOLUTION	Adjust topology/routing configuration to restore balance, or accept the asymmetry if capacity still supports it.
VALIDATION	Confirm utilization redistributes to an acceptable balance post-adjustment.
PREVENTION	Model expected routing changes before a fabric expansion, not just after observing the resulting utilization pattern.

26.5 Data Center and Hardware Lifecycle Cross-Cutting Scenarios

[Illustrative]

Scenario 12 — Wrong Node Powered Down During a Maintenance Window

SYMPTOM	During planned maintenance, a node not scheduled for work loses power/service unexpectedly.
INITIAL CHECKS	Compare the maintenance plan's intended target against what was actually actioned.
COMMANDS	Asset tracking records (Chapter 25) cross-referenced against the maintenance plan.
OUTPUT INTERPRETATION	The physical location acted upon doesn't match the intended target once cross-referenced against asset records.
POSSIBLE CAUSES	A labeling/asset-tracking gap (Chapter 25) leading to the wrong physical node being identified; a rack-position documentation error.
FAULT ISOLATION	Confirmed directly by the asset-record cross-reference.
ROOT CAUSE	Asset tracking records didn't accurately reflect the physical rack position, leading to the wrong node being actioned.
RESOLUTION	Restore the affected node to service, and correct the asset record.
VALIDATION	Confirm the corrected asset record matches physical reality on a follow-up audit.

PREVENTION	Verify asset-tracking accuracy periodically (Chapter 25), not only when a maintenance mistake reveals a gap.

[Illustrative]

Scenario 13 — Wrong Part RMA'd Due to an Asset-Tracking Gap

SYMPTOM	An RMA is processed, but the replacement doesn't resolve the original symptom.
INITIAL CHECKS	Confirm the part actually removed/returned matches the part identified as faulty.
COMMANDS	Asset tracking and RMA documentation review (Chapter 25).
OUTPUT INTERPRETATION	The serial number recorded in the RMA paperwork doesn't match the serial number physically present in the slot the fault was diagnosed against — the asset record and physical reality had already diverged before the RMA was processed.
POSSIBLE CAUSES	An asset-tracking error identifying the wrong serial number/slot as the faulty component.
FAULT ISOLATION	Physically re-inspecting the slot's actual current serial number against both the RMA record and the original fault diagnosis confirms the mismatch originated in the asset record, not in the diagnosis itself.

ROOT CAUSE	A prior, undocumented part swap (e.g., during an earlier repair) left the asset-tracking record out of sync with the slot's true physical contents, so the RMA was correctly processed against the record but not against the actual faulty part.
RESOLUTION	Correct the asset record to match physical reality, then RMA the actual faulty part.
VALIDATION	Confirm the correctly identified part, once actually replaced, resolves the original symptom.
PREVENTION	Cross-check serial numbers/slot positions at the point of physical removal, not just from records, before shipping an RMA.

[Illustrative]

Scenario 14 — Firmware Drift Resurfacing After a New Node Batch Is Added

SYMPTOM	After adding a new batch of nodes to an existing fleet, a subset of "old" nodes begin showing inconsistent behavior relative to the new ones.
INITIAL CHECKS	Compare firmware/BIOS baseline (Chapter 6/25) between the new batch and the existing fleet.
COMMANDS	Firmware version inventory comparison (Chapter 25's asset tracking).
OUTPUT INTERPRETATION	The new batch shipped with a newer firmware baseline than the existing fleet was standardized on.

POSSIBLE CAUSES	The fleet's firmware baseline was never revisited after the original standardization, and new hardware now defaults to a different version.
FAULT ISOLATION	The version comparison confirms this directly.
ROOT CAUSE	**The existing fleet's firmware baseline had become outdated relative to what new hardware ships with, and the two were never reconciled.**
RESOLUTION	Decide on and apply a single, current firmware baseline across both old and new nodes.
VALIDATION	Confirm consistent behavior across the full fleet once firmware is reconciled.
PREVENTION	Revisit the firmware baseline decision whenever new hardware is added, rather than treating it as a one-time, permanent standard.

26.6 Ambiguous "Node Unavailable" Scenarios

[Illustrative]

Scenario 15 — "Node Unavailable" Traced to a Provisioning/Network Issue, Not Hardware

SYMPTOM	A node reports as unavailable to the scheduler (Chapter 20), and the on-call assumption is a hardware fault.
INITIAL CHECKS	Attempt to reach the node directly before assuming hardware failure.
COMMANDS	SSH (Chapter 3) to the node; `ip a` / `ss` if reachable.

OUTPUT INTERPRETATION	The node is reachable via SSH and appears otherwise healthy, but isn't correctly registered on the management network path Slurm uses.
POSSIBLE CAUSES	A provisioning step (Chapter 2) that didn't complete correctly for this node's management-network registration.
FAULT ISOLATION	Reachability via SSH but not via the scheduler's expected path isolates this to a provisioning/registration gap, not hardware.
ROOT CAUSE	**An incomplete provisioning step left the node functionally fine but invisible to the scheduler's expected communication path.**
RESOLUTION	Re-run the relevant provisioning step to correctly register the node.
VALIDATION	Confirm the scheduler reports the node as available after re-registration.
PREVENTION	Include scheduler-visibility as an explicit check in the new-node-to-production checklist (Chapter 25), not just basic reachability.

[Illustrative]

Scenario 16 — "Node Unavailable" Traced to a Fabric-Layer Issue, Diagnosed by Elimination

SYMPTOM	A node reports as unavailable, and initial host-level and hardware checks find nothing wrong.

INITIAL CHECKS	Systematically rule out each layer — host (Chapter 4), hardware (Chapter 6), scheduler communication (Chapter 20) — before considering the fabric.
COMMANDS	Host-level checks (clean), `scontrol show node` (clean at the Slurm layer), then `ibnetdiscover` from another node (Chapter 12) to check this node's fabric visibility.
OUTPUT INTERPRETATION	The node is absent from the fabric topology as seen from other nodes, despite being reachable on the management network.
POSSIBLE CAUSES	A fabric-layer fault (Chapter 12) specific to this node's data-network path, distinct from its management-network reachability (Chapter 2's separation made concrete here).
FAULT ISOLATION	The management-vs-data-network split directly explains why host-level checks looked clean while the node was still functionally unavailable for job traffic.
ROOT CAUSE	A fabric-layer fault on this node's data-network connection, invisible to management-network-based host checks.
RESOLUTION	Apply Chapter 12's InfiniBand fault-isolation methodology to the specific link/port involved.
VALIDATION	Confirm the node reappears in fabric topology and the scheduler resumes sending it work.
PREVENTION	When host-level and hardware checks are both clean but a node remains unavailable, escalate to fabric-layer checks systematically rather than repeating the same host-level checks.

CHAPTER 27

TROUBLESHOOTING

Practical HPC Troubleshooting Playbooks II — Compute & Workload

27.1 How to Use This Playbook

This chapter continues Chapter 26's approach, applied to GPU, Slurm, and NCCL domains. As with Chapter 26, none of these scenarios repeat a failure mode already used in Chapters 14, 20, or 24 — each emphasizes cross-layer evidence, intermittent behavior, capacity limits, maintenance mistakes, or configuration drift at the compute/workload layer.

Symptom Index (Continued from Chapter 26)

Symptom	Likely Domain(s)	Scenario	See Also
GPU failure only under sustained load	GPU	27.2 #17	Ch.14
Intermittent Xid tied to a power event	GPU	27.2 #18	Ch.14, 25
MIG misconfiguration reducing throughput	GPU	27.2 #19	Ch.13, 19

GPU memory capacity miss at scale	GPU	27.2 #20	Ch.13, 14
Scheduling issue plus misread node-drain	Slurm	27.3 #21	Ch.20
QOS configuration drift	Slurm	27.3 #22	Ch.18
Node returned to service with stale drain reason	Slurm	27.3 #23	Ch.20
Partition silently exhausted	Slurm	27.3 #24	Ch.18, 20
Cross-layer communication failure (GPU + fabric)	NCCL	27.4 #25	Ch.12, 14, 24
Intermittent NCCL slowdown tied to a rack	NCCL	27.4 #26	Ch.24
Performance degradation only at scale	NCCL	27.4 #27	Ch.24
Communication failure misattributed to NCCL	NCCL	27.4 #28	Ch.19, 24

Monitoring alert storm	Monitoring/capacity/maintenance	27.5 #29	Ch.21
GPU memory capacity planning miss	Monitoring/capacity/maintenance	27.5 #30	Ch.13, 21
Firmware rollout mistaken for code regression	Monitoring/capacity/maintenance	27.5 #31	Ch.6, 25
Environment/container drift across nodes	Monitoring/capacity/maintenance	27.5 #32	Ch.3

27.2 GPU Cross-Cutting Scenarios

[Illustrative]

Scenario 17 — GPU Failure Only Under Sustained Multi-GPU Load

SYMPTOM

A GPU passes a quick health check but fails during long-running, multi-GPU jobs specifically.

INITIAL CHECKS

Run a sustained diagnostic (`dcgmi diag` at a deeper level, Chapter 13/14) rather than relying on the quick check alone.

COMMANDS

`dcgmi diag` (deeper level), `nvidia-smi -q` (thermal/power trend during a sustained run).

OUTPUT INTERPRETATION

The deeper diagnostic surfaces an issue the quick check didn't stress enough to reveal.

POSSIBLE CAUSES

A marginal component that only fails under sustained thermal/power load; a cooling issue specific to sustained multi-GPU utilization.

FAULT ISOLATION	The deeper diagnostic's specific failure mode narrows this to a load-dependent condition, not an always-present fault.
ROOT CAUSE	A marginal GPU component that a quick check couldn't stress sufficiently to detect.
RESOLUTION	RMA the affected GPU (Chapter 25) based on the deeper diagnostic's evidence.
VALIDATION	Confirm the replacement GPU passes the same sustained diagnostic without failure.
PREVENTION	Include a sustained, not just quick, diagnostic pass in new-GPU-node validation (Chapter 25).

[Illustrative]

Scenario 18 — Intermittent Xid Correlated with a Power Event

SYMPTOM	An Xid error appears intermittently, with no clear software trigger.
INITIAL CHECKS	Check whether Xid occurrences correlate with any recorded power-related event (Chapter 6's BMC/SEL evidence).
COMMANDS	`dmesg` / `journalctl` (Xid timestamps), BMC SEL review (Chapter 6).
OUTPUT INTERPRETATION	Every Xid timestamp lines up, within a few seconds, with a power-supply-related entry in the BMC SEL — a correlation that isn't visible from `dmesg` / `journalctl` alone.
POSSIBLE CAUSES	A power delivery irregularity affecting the GPU specifically under certain conditions.

FAULT ISOLATION	The consistent timestamp correlation across multiple occurrences, rather than a one-off coincidence, points to the power subsystem rather than a software or application-level trigger.
ROOT CAUSE	A marginal power supply unit intermittently dipping below the GPU's required delivery tolerance under load, triggering the Xid without any software-side cause.
RESOLUTION	Replace the implicated power supply unit (Chapter 6/25) rather than pursuing a software-side fix for a hardware-side condition.
VALIDATION	Confirm no further Xid occurrences correlate with subsequent power events after remediation.
PREVENTION	Correlate Xid events against power/thermal telemetry (Chapter 21) as standard practice, not just in response to a recurring complaint.

[Illustrative]

Scenario 19 — MIG Misconfiguration Silently Reducing Throughput

SYMPTOM	A workload's throughput is lower than expected, without any outright error.
INITIAL CHECKS	Confirm whether the job is running on a full GPU or a MIG instance (Chapter 13/19).
COMMANDS	`nvidia-smi` (MIG state), `scontrol show job` (Chapter 19, GRES allocation).

OUTPUT INTERPRETATION	The job is running on a MIG instance with a fraction of the GPU's resources, while the workload was sized assuming a full GPU.
POSSIBLE CAUSES	A GRES/MIG configuration mismatch (Chapter 19) between what was intended and what was actually configured.
FAULT ISOLATION	Confirmed directly by comparing the allocation against the workload's assumptions.
ROOT CAUSE	A MIG configuration was left enabled on a node intended for full-GPU workloads.
RESOLUTION	Reconfigure the node's MIG state (or the job's request) to match the intended usage.
VALIDATION	Confirm throughput matches expectations once the mismatch is corrected.
PREVENTION	Document and enforce which nodes are MIG-partitioned versus full-GPU as part of fleet configuration tracking (Chapter 25).

[Illustrative]

Scenario 20 — GPU Memory Capacity Exhaustion Only at Scale

SYMPTOM	A workload runs fine in small-scale testing but fails with GPU memory errors once scaled up.
INITIAL CHECKS	Compare GPU memory usage at small scale against the larger scale's actual per-GPU demand.
COMMANDS	`nvidia-smi` (memory usage) during both a small-scale and large-scale run.

OUTPUT INTERPRETATION	Per-GPU memory usage grows with scale in a way the small-scale test didn't reveal (e.g., due to increased buffering or batch size assumptions).
POSSIBLE CAUSES	The workload's memory usage doesn't scale the way capacity planning assumed.
FAULT ISOLATION	The direct comparison across scales confirms the growth pattern.
ROOT CAUSE	Capacity planning assumed memory usage would remain flat with scale, when it actually grew.
RESOLUTION	Adjust the workload's configuration (batch size, buffering) or request GPUs with sufficient memory headroom for the actual scaled demand.
VALIDATION	Confirm the workload completes without memory errors at full scale post-adjustment.
PREVENTION	Test memory usage at a representative scale before committing to a capacity plan based on small-scale testing alone.

27.3 Slurm and Scheduling Cross-Cutting Scenarios

[Illustrative]

Scenario 21 — Scheduling Issue Compounded by a Misunderstood Node-Drain State

SYMPTOM	Jobs are pending longer than expected, and the on-call engineer assumes a scheduler bug.

INITIAL CHECKS	Check the state of nodes in the relevant partition (Chapter 20) before assuming a scheduling-logic problem.
COMMANDS	`sinfo`, `scontrol show node` (Chapter 17/20).
OUTPUT INTERPRETATION	A significant fraction of the partition's nodes are in a DRAINED state, reducing effective capacity well below what `sinfo`'s node count alone suggests at a glance.
POSSIBLE CAUSES	A planned maintenance drain (Chapter 20/25) that wasn't communicated clearly, making reduced capacity look like a scheduler malfunction.
FAULT ISOLATION	Confirmed directly by the node-state check.
ROOT CAUSE	**Reduced effective capacity from planned drains, not a scheduling defect, explained the longer pending times.**
RESOLUTION	No scheduler fix needed; communicate planned drains and their capacity impact more clearly going forward.
VALIDATION	Confirm pending times return to normal once drained nodes are restored to service.
PREVENTION	Include expected capacity impact in maintenance communication, so longer pending times aren't mistaken for a fault during planned drains.

[Illustrative]

Scenario 22 — Configuration Drift in QOS Settings Causing a Gradual Priority Skew

SYMPTOM	Over time, one account's jobs consistently start noticeably later than others with similar resource requests.
INITIAL CHECKS	Compare the account's QOS assignment and limits (Chapter 18) against other, similarly-situated accounts.
COMMANDS	`sacctmgr show qos`, `sprio` (Chapter 17/18/20).
OUTPUT INTERPRETATION	The account's QOS was changed at some point (for a since-resolved, specific reason) and never reverted.
POSSIBLE CAUSES	A temporary QOS adjustment made during a past incident that became permanent by omission.
FAULT ISOLATION	Confirmed by comparing the QOS assignment's history/rationale against its current, ongoing effect.
ROOT CAUSE	A temporary policy change was never reverted after its original justification no longer applied.
RESOLUTION	Revert the QOS assignment to the standard configuration, or formally justify keeping it if circumstances still warrant it.
VALIDATION	Confirm the account's job start times return to a pattern consistent with similarly-situated accounts.

PREVENTION	Track temporary policy changes with an explicit expiration or review date, rather than leaving them indefinite by default.

[Illustrative]

Scenario 23 — Node Returned to Service Without Clearing Its Drain Reason

SYMPTOM	A node accepts jobs again after maintenance, but its recorded state still references an old drain reason, causing confusion during a later investigation.
INITIAL CHECKS	Check `scontrol show node`'s current reason field against what actually happened most recently.
COMMANDS	`scontrol show node` (Chapter 17/20).
OUTPUT INTERPRETATION	The reason field still shows the original maintenance-related drain reason, even though the node has been back in service and running jobs successfully for some time.
POSSIBLE CAUSES	The node was resumed without explicitly clearing/updating its state reason.
FAULT ISOLATION	Confirmed directly — the node is functionally fine, only its recorded reason is stale.
ROOT CAUSE	Standard resume procedure didn't include clearing the stale reason field.
RESOLUTION	Update the node's state reason to reflect its current, correct status.
VALIDATION	Confirm the reason field is accurate going forward.

PREVENTION	Make clearing/updating the state reason an explicit step in the node-resume checklist (Chapter 20/25), not an assumed side effect of resuming.

[Illustrative]

Scenario 24 — Partition Silently Exhausted of a Specific Resource Type

SYMPTOM	Jobs requesting a specific resource type pend indefinitely, while jobs not requesting it run normally in the same partition.
INITIAL CHECKS	Check the partition's total and currently-allocated inventory of that specific resource type (e.g., a particular GRES type, Chapter 19).
COMMANDS	`sinfo`, `scontrol show node` (GRES inventory, Chapter 19/20).
OUTPUT INTERPRETATION	The specific resource type is fully allocated across the partition, while general CPU/memory capacity remains available.
POSSIBLE CAUSES	A resource type with limited supply relative to demand, not visible in a general capacity check that only looks at CPU/memory.
FAULT ISOLATION	Confirmed directly by checking that specific resource type's allocation state.
ROOT CAUSE	The specific resource type was fully committed, and the pending reason for jobs requesting it was accurate, just not obvious from a general capacity glance.

RESOLUTION	No fault to fix — communicate the specific resource constraint to affected users, or add capacity if the constraint is a genuine ongoing bottleneck.
VALIDATION	Confirm the pending reason is understood and matches actual resource availability going forward.
PREVENTION	Monitor specific, limited resource types (not just aggregate CPU/memory) as their own capacity signal (Chapter 21).

27.4 NCCL and Communication Cross-Cutting Scenarios

[Illustrative]

Scenario 25 — Communication Failure Requiring Evidence from Both GPU and Fabric Layers

SYMPTOM	A multi-node training job fails intermittently with a communication error that doesn't clearly implicate one specific layer.
INITIAL CHECKS	Gather evidence from both the GPU layer (Chapter 14) and the fabric layer (Chapter 12) simultaneously, rather than investigating one exhaustively before considering the other.
COMMANDS	`nvidia-smi -q` (Chapter 14) and `perfquery` / `iblinkinfo` (Chapter 12), both on the nodes involved in the failure.

OUTPUT INTERPRETATION	`nvidia-smi -q` shows a low, individually-unremarkable rate of correctable ECC errors on one node's GPU, while `perfquery` on that same node's HCA shows a similarly low, individually-unremarkable rate of link errors — neither alone would normally be flagged as a fault.
POSSIBLE CAUSES	A combination of a marginal GPU condition and a marginal fabric condition, neither severe enough alone to cause a clear, isolated failure, but sufficient together.
FAULT ISOLATION	Reviewing both layers' evidence side by side, rather than clearing each individually against its own normal threshold, shows the failures only occur on the node carrying both low-grade conditions simultaneously — neither condition's own threshold would have triggered an isolated alert.
ROOT CAUSE	**Two independently minor, sub-threshold conditions — one GPU-side, one fabric-side — combining on the same node to intermittently push a data transfer past what either marginal component could reliably sustain alone.**
RESOLUTION	Address both contributing conditions rather than just one: schedule the GPU for evaluation per Chapter 14's ECC guidance, and remediate the marginal link per Chapter 12's fault-isolation methodology.
VALIDATION	Confirm the job completes reliably once both contributing conditions are addressed.

PREVENTION	When a symptom doesn't cleanly implicate one layer, gather evidence from every plausible layer in parallel rather than sequentially exhausting one at a time.

[Illustrative]

Scenario 26 — Intermittent NCCL Slowdown Correlated with a Specific Rack

SYMPTOM	Multi-node jobs run slower specifically when scheduled onto nodes within one particular rack.
INITIAL CHECKS	Compare job performance by rack/node placement rather than assuming the issue is random.
COMMANDS	`NCCL_DEBUG` logging (Chapter 24) across jobs, correlated with node placement (Chapter 20).
OUTPUT INTERPRETATION	A clear correlation emerges between the specific rack and reduced performance, independent of job type.
POSSIBLE CAUSES	A shared fabric uplink serving that rack specifically that's degraded or oversubscribed (Chapter 12).
FAULT ISOLATION	The rack-level correlation points toward shared infrastructure specific to that rack, not individual nodes within it.
ROOT CAUSE	A degraded shared uplink serving that rack was affecting every job using nodes within it.

RESOLUTION	Address the specific uplink (Chapter 12/25) rather than treating this as a per-node or per-job issue.
VALIDATION	Confirm performance normalizes for jobs using that rack once the uplink is addressed.
PREVENTION	Include rack-level performance correlation as a standard check (Chapter 21) when investigating "some nodes seem slower" reports.

[Illustrative]

Scenario 27 — Performance Degradation Only Appearing at Larger Scale

SYMPTOM	A distributed job's per-GPU throughput drops as the job is scaled to more nodes, beyond what communication overhead alone would explain.
INITIAL CHECKS	Compare measured scaling behavior against a reasonable expectation for the collective operations involved (Chapter 23).
COMMANDS	`NCCL_DEBUG` logging across a range of scales, timing comparison.
OUTPUT INTERPRETATION	Throughput degrades faster than expected once the job spans more nodes than fit within a single top-of-rack switch's domain (Chapter 12).
POSSIBLE CAUSES	A topology-related bottleneck that only becomes relevant once communication has to cross additional switch tiers.

FAULT ISOLATION	Comparing scaling behavior within a single switch domain versus across multiple domains isolates the effect to the topology crossing point.
ROOT CAUSE	Communication crossing additional fabric tiers introduced latency/bandwidth costs the smaller-scale testing never exercised.
RESOLUTION	This may not be a "fault" to fix but a topology reality to plan around — e.g., preferring node allocations that minimize tier-crossing where the scheduler allows (Chapter 19's topology-aware scheduling).
VALIDATION	Confirm scaling behavior matches expectations once topology-aware placement is applied.
PREVENTION	Test scaling behavior across realistic topology boundaries before assuming small-scale results will extrapolate linearly.

[Illustrative]

Scenario 28 — Communication Failure Initially Blamed on NCCL, Traced to an Application Misconfiguration

SYMPTOM	A distributed job fails with an NCCL-reported communication error, and the team's first assumption is a network or NCCL bug.
INITIAL CHECKS	Check the job's actual rank count and launch configuration against what the application expected.
COMMANDS	`NCCL_DEBUG` logging (Chapter 24), job launch configuration review.

OUTPUT INTERPRETATION	The number of ranks NCCL was told to expect doesn't match the number of processes actually launched.
POSSIBLE CAUSES	A launch-script error under- or over-specifying the process count relative to the allocation (Chapter 17/19).
FAULT ISOLATION	The rank-count mismatch is directly visible in the debug output once checked.
ROOT CAUSE	A launch-configuration error, not a network or NCCL defect, caused ranks to never fully assemble into a valid communicator.
RESOLUTION	Correct the launch configuration to match the actual allocation.
VALIDATION	Confirm the job initializes and runs successfully with the corrected launch configuration.
PREVENTION	Validate rank count against allocation size as an automatic pre-flight check in job launch scripts, rather than discovering a mismatch only at NCCL initialization.

27.5 Monitoring, Capacity, and Maintenance Scenarios (Workload Layer)

[Illustrative] **Scenario 29 — A Monitoring Alert Storm Obscures the One Alert That Mattered**

SYMPTOM	During an incident, dozens of alerts fire simultaneously, and the actually-relevant one is missed initially.

INITIAL CHECKS	Review the full alert set chronologically to find which alert fired first, before the cascade.
COMMANDS	Alert history review (Chapter 21).
OUTPUT INTERPRETATION	One specific alert (e.g., a fabric link error, Chapter 12) preceded all the others by several minutes, with the rest being downstream consequences.
POSSIBLE CAUSES	A monitoring setup that alerts on symptoms at every layer independently, without suppressing downstream alerts once a root-cause alert has fired.
FAULT ISOLATION	The chronological ordering identifies the root alert directly.
ROOT CAUSE	A single root-cause condition triggered a cascade of downstream alerts, and the alerting system didn't distinguish cause from effect.
RESOLUTION	Address the root-cause condition the first alert identified.
VALIDATION	Confirm the downstream alerts clear once the root cause is resolved.
PREVENTION	Implement alert correlation/suppression logic (Chapter 21) so a root-cause alert is distinguishable from its downstream consequences during future incidents.

[Illustrative]

Scenario 30 — GPU Memory Capacity Planning Miss Causing OOM-Style Failures at Scale

SYMPTOM	A workload that ran fine in a pilot phase begins failing with GPU memory errors once rolled out more broadly.
INITIAL CHECKS	Compare the pilot's actual configuration against the broader rollout's configuration for differences.
COMMANDS	`nvidia-smi` (memory usage) comparison between pilot and rollout jobs.
OUTPUT INTERPRETATION	The broader rollout uses a larger batch size or additional concurrent processes per GPU that the pilot didn't exercise.
POSSIBLE CAUSES	Capacity planning based on the pilot's configuration without accounting for how the rollout's configuration would differ.
FAULT ISOLATION	Confirmed by the direct configuration comparison.
ROOT CAUSE	**The rollout's actual configuration demanded more GPU memory per instance than the pilot-based capacity plan assumed.**
RESOLUTION	Adjust the rollout's configuration or GPU allocation to match actual memory demand.
VALIDATION	Confirm the rollout runs without memory errors post-adjustment.
PREVENTION	Validate capacity assumptions against the actual production configuration, not just the pilot's, before a broader rollout.

[Illustrative]

Scenario 31 — A Firmware Rollout Mistaken for a Code Regression

SYMPTOM	GPU workload performance changes noticeably around the same time as an unrelated code deployment, and the code is initially blamed.
INITIAL CHECKS	Check whether any firmware/driver rollout (Chapter 6/25) occurred around the same time.
COMMANDS	Asset/firmware tracking records (Chapter 25) cross-referenced against the performance-change timeline.
OUTPUT INTERPRETATION	A firmware rollout completed within the same window as the code deployment, creating a coincidental correlation.
POSSIBLE CAUSES	The firmware change itself altered GPU behavior (e.g., a power/thermal management change) independent of the code deployment.
FAULT ISOLATION	Reverting or isolating each change independently (where feasible) separates the two candidate causes.
ROOT CAUSE	**The firmware rollout, not the code deployment, was responsible for the performance change.**
RESOLUTION	Address the firmware-related behavior change directly (adjust configuration, or coordinate with the vendor if unexpected) rather than continuing to investigate the code.
VALIDATION	Confirm performance returns to the expected pattern once the firmware factor is addressed.

PREVENTION	Track infrastructure changes (firmware, driver) on the same timeline as application deployments (Chapter 21/25), so coincidental correlations are caught quickly rather than assumed away.

[Illustrative]

Scenario 32 — Environment/Container Version Drift Causing Inconsistent Job Behavior

SYMPTOM	The same job script produces different results depending on which node it lands on.
INITIAL CHECKS	Compare the environment module (Chapter 3) or container image version actually loaded on each node.
COMMANDS	`module list` (Chapter 3), container image version/digest check (Chapter 3).
OUTPUT INTERPRETATION	Different nodes have different default module versions or cached container image versions available.
POSSIBLE CAUSES	A module or container image update that reached some nodes but not others (configuration drift, Chapter 26's Scenario 2 pattern, here at the software-environment layer specifically).
FAULT ISOLATION	Confirmed directly by the version comparison across nodes.
ROOT CAUSE	An incomplete software-environment update left some nodes on an older module/container version than others.

RESOLUTION	Synchronize the software environment across all nodes to the intended version.
VALIDATION	Confirm consistent job behavior across nodes once versions are synchronized.
PREVENTION	Pin explicit module/container versions in job scripts where reproducibility matters, rather than relying on "default" resolving consistently across a fleet that can drift.

PART 12

Reference

Chapter 28, Chapter 29

CHAPTER 28

REFERENCE

HPC Command Cheat Sheet

28.1 How to Use This Cheat Sheet

This chapter is a lookup tool, not a teaching chapter. Every entry below was introduced and explained in an earlier chapter — the "See" column names it. Safety classifications here are consolidated exactly as they appear in each command's home chapter, not independently re-derived, so this table stays consistent with the rest of the book rather than becoming a second, possibly-drifting source of truth.

Safety legend: **S** = SAFE/READ-ONLY · **C** = CHANGES STATE · **P** = POTENTIALLY DISRUPTIVE · **X** = CONTEXT-DEPENDENT (safety depends entirely on what's executed through it).

28.2 Linux — Filesystem and Processes

Command	Purpose	What to Look For	Safety	See
`ls`, `pwd`, `cd`	Navigation	—	S	Ch.3
`ps`	List processes	State code (running/sleeping/zombie), owner	S	Ch.3, 4

`whoami`, `id`	Confirm current user identity/groups	Group membership, fo permission issues
`chmod`, `chown`	Change permissions/ownership	Confirm target before running, especially re
`sudo`	Run a command with elevated privilege	Depends entirely on th command run through
`systemctl status <svc>`	Check service state	Active/inactive, recen lines
`systemctl start/stop/enable/disable`	Change service state/boot behavior	—
`systemctl restart` (on a service others depend on)	Restart a service	Confirm dependents f
`journalctl`	Read systemd journal	Recent events, correla timestamp
`dmesg`	Read kernel ring buffer	Hardware/driver even require elevated privil `kernel.dmesg_res` is set
`kill <pid>`	Request clean process termination (SIGTERM)	—
`kill -9` / `pkill`	Force termination (SIGKILL)	Can leave resources/s inconsistent
`ssh`	Connect to a remote host	Classification covers connecting only
`ssh-keygen`	Generate a key pair	Creates/can overwrite key files

`ssh-copy-id`	Install a public key on a remote host	—	C	Ch.3
`dnf list installed` / `dnf info <pkg>`	Check installed packages/versions	—	S	Ch.3
`dnf install`	Install a package	Avoid ad hoc on production nodes	C/P	Ch.3
`env`	List environment variables	—	S	Ch.3
`export VAR=value`	Set a session environment variable	Session-scoped only	C	Ch.3
`module avail` / `module list`	List available/loaded modules	—	S	Ch.3
`module load` / `unload`	Change session's loaded modules	Session-scoped only	C	Ch.3
`apptainer run`	Run a container image	Depends entirely on the image	X	Ch.3

* `dmesg`'s read access can require elevated privilege depending on the `kernel.dmesg_restrict` sysctl; the exact privileges/capabilities required depend on the operating system and its configuration — don't assume it's always readable without elevated access.

28.3 Linux — CPU and Memory

Command	Purpose	What to Look For	Safety	See
`top` / `htop`	Live per-process CPU/memory view	Sustained high-CPU processes	S	Ch.4
`free`	Memory/swap usage	"Available," not just "free"	S	Ch.4
`vmstat`	System snapshot incl. I/O wait	Wait column vs. actual CPU busy	S	Ch.4
`lscpu`	CPU topology	Socket/core/thread/NUMA layout	S	Ch.5
`numactl --hardware`	NUMA topology	Node count, memory per node	S	Ch.5
`numastat`	NUMA memory access stats	Local vs. remote access ratio	S	Ch.5
`numactl <bind-flags> <cmd>`	Launch a process with CPU/memory binding	Confirm binding matches actual topology	C	Ch.5

28.4 Linux — Networking

Command	Purpose	What to Look For	Safety	See
`ip a` / `ip link`	Host interface state	Interface up, address present	S	Ch.3, 4
`ss`	Active sockets/listening ports	Service listening where expected	S	Ch.3, 4

28.5 Storage

Command	Purpose	What to Look For	Safety	See
`lsblk`	List block devices	Local disks/partitions	S	Ch.3
`df`	Filesystem capacity	Space used vs. available	S	Ch.3, 8
`du`	Directory space usage	Identify what's consuming space	S	Ch.8
`mount` (no args)	List current mounts	Confirm expected mount is present	S	Ch.3, 8
`mount <dev> <path>`	Mount a filesystem	—	C	Ch.3, 8

`umount`	Unmount a filesystem	Confirm no active users first	C/P	Ch.8

28.6 InfiniBand

Command	Purpose	What to Look For	Safety	See
`ibstat` / `ibstatus`	Port/link state	Active state, expected rate	S	Ch.9, 11, 12
`ibv_devinfo`	HCA device detail	Device recognized, firmware version	S	Ch.9, 11
`iblinkinfo`	Link-level detail fabric-wide	State/rate for every discoverable link	S	Ch.11, 12
`ibnetdiscover`	Fabric topology discovery	Node/switch reachability	S	Ch.11, 12
`ibswitches`	List discovered switches	—	S	Ch.11
`ibdev2netdev`	Map IB device to net device (IPoIB)	Confirm correct interface	S	Ch.11
`perfquery`	Per-port error/performance counters	Rising error counts	S	Ch.11, 12

`ibdiagnet`	Fabric-wide diagnostic sweep	Topology/counter anomalies	S	Ch.11, 12
`ibportstate`	Query/change port administrative state	—	C/P	Ch.12

28.7 GPU

Command	Purpose	What to Look For	Safety	See
`nvidia-smi`	GPU state snapshot	Device present, utilization, temperature, power	S	Ch.13, 14
`nvidia-smi -q`	Detailed GPU query	PCIe link state, ECC counts, driver version	S	Ch.13, 14
`nvidia-smi topo -m`	GPU topology matrix	NVLink vs. PCIe-only connections, CPU affinity	S	Ch.13, 15
`dcgmi diag`	GPU health diagnostic	Level-dependent; heavier levels stress the GPU	S/P	Ch.13, 14
`lspci`	PCI device listing	GPU detected at hardware level	S	Ch.13, 14

`lsmod`	Loaded kernel modules	NVIDIA driver module loaded	S	Ch.13, 14
`modinfo nvidia`	Driver module detail	Driver version	S	Ch.13

28.8 Slurm

Command	Purpose	What to Look For	Safety	See
`sinfo`	Cluster/partition/node state	Partition capacity, node states	S	Ch.17
`squeue`	Job queue state	Pending reason, running state	S	Ch.17, 18, 20
`scontrol show job/node`	Detailed job/node info	Full state, reason codes, GRES allocation	S	Ch.17, 19, 20
`scontrol update`	Change node/job state (admin)	—	C	Ch.20
`sacct`	Historical job accounting	Resource usage, exit code	S	Ch.17, 20

`sstat`	Live resource usage for a running job	—	S	Ch.17
`sbatch`	Submit a batch job	—	C	Ch.17
`srun`	Launch work directly/within an allocation	—	C	Ch.17
`salloc`	Request an allocation	—	C	Ch.17
`scancel`	Cancel a job	Own job vs. another's (permissions)	C/P	Ch.17
`sacctmgr show ...`	Query accounting hierarchy	—	S	Ch.17, 20
`sacctmgr add/modify/delete`	Change accounting hierarchy	Site-policy consequence	C/P	Ch.17
`sprio`	Priority breakdown for pending jobs	Fairshare/QOS/age components	S	Ch.18, 20
`sdiag`	Scheduler diagnostic statistics	—	S	Ch.20

28.9 UFM

No CLI commands — UFM (Chapter 22) is a UI/API-driven fabric management and monitoring platform, not a command-line tool.

28.10 NCCL

No CLI commands — NCCL (Chapters 23–24) is a library, controlled at runtime through environment variables.

Variable	Purpose	What to Look For	Safety	See
`NCCL_DEBUG`	Controls NCCL's diagnostic logging verbosity	Set for a run to gather initialization/transport evidence	S	Ch.23, 24
`NCCL_SOCKET_IFNAME` (or current equivalent)	Influences network interface selection	Confirm correct interface on multi-NIC nodes	S	Ch.24

Exact NCCL environment variable names and defaults should be verified against current NVIDIA NCCL documentation before relying on them, as they can change across versions.

CHAPTER 29

REFERENCE

HPC Interview & Scenario Questions

29.1 How to Use This Chapter

These questions test reasoning, not recall. Each includes answer guidance describing a sound reasoning path — not a single memorized fact — consistent with this book's Observe → Collect Evidence → Isolate → Root Cause → Fix → Validate → Prevent philosophy (Chapter 1). Use them to prepare for an interview or to test your own troubleshooting judgment against a domain you've just studied.

29.2 Linux and Performance Troubleshooting

1. A node shows a load average higher than its core count, but `top` shows low CPU utilization. Walk through your reasoning. *Guidance:* High load with low CPU utilization points toward processes in an uninterruptible-sleep (I/O-wait) state rather than CPU contention (Chapter 4). Check `vmstat`'s wait column and `iostat` next, not more CPU-focused tools.

2. A job dies with no application-level error message. How do you determine whether it was killed by the OOM killer? *Guidance:* Check `dmesg` / `journalctl` around the failure timestamp for an OOM-related kernel message naming the process (Chapter 4) before assuming an application bug.

3. You need to kill a hung process, but `kill <pid>` has no effect after a reasonable wait. What do you do, and what are you risking? *Guidance:* Escalate to `kill -9` (SIGKILL) only after confirming SIGTERM had no effect; recognize this risks leaving shared resources or job accounting inconsistent (Chapter 4) — investigate what the process was blocked on afterward, not just move on.

4. A user reports `dmesg` returns a permission error on a node they can normally use it on. What's the likely explanation? *Guidance:* Likely `kernel.dmesg_restrict` is enabled, requiring elevated privilege (`CAP_SYSLOG`) to read the kernel log buffer — a distribution/hardening-dependent behavior, not necessarily a new fault on that specific node.

5. Two "identical" nodes show different behavior for the same job. What's your first move? *Guidance:* Compare configuration state directly — package versions, loaded modules (Chapter 3), firmware baseline (Chapter 6) — rather than assuming the job or application is the variable; configuration drift is a distinct root-cause category worth checking for explicitly.

6. Explain the difference between a zombie process and a genuinely hung process, and why the distinction matters operationally. *Guidance:* A zombie (Chapter 4) has already exited and is waiting on its parent to collect its status — it's not consuming meaningful resources and doesn't need to be "killed." A hung process is still running/blocked and consuming resources. Treating a zombie as if it needs forceful intervention, or a hung process as harmless, are both mistakes with different consequences.

29.3 HPC Architecture and CPU/NUMA

7. Explain why HPC clusters typically separate management and data/fabric networks. *Guidance:* Control/monitoring traffic and job I/O traffic have different bandwidth/latency characteristics;

separating them prevents one from starving the other (Chapter 2). A candidate should also note this is a design principle, not a specific product.

8. A multi-socket server runs the same job slower on one node than an identical one. What would you check first? *Guidance:* `numastat` to check local vs. remote memory access ratio (Chapter 5) — a NUMA-binding mismatch is a common, checkable cause before assuming a hardware difference.

9. What's the difference between CPU affinity and CPU pinning, and when might pinning make things worse? *Guidance:* Affinity constrains where a process can run; pinning applies it deliberately, usually for locality. Pinning to a NUMA node that doesn't hold the process's actual memory can perform worse than leaving the kernel free to place it (Chapter 5).

10. Why doesn't a cluster's architecture look identical at every site, even for similar workloads? *Guidance:* Architecture reflects tradeoffs (Chapter 2/7) — ownership model, workload mix, existing infrastructure — not a single "correct" design; a good answer avoids presenting any one architecture as universal.

11. A GPU node's power/cooling requirements differ meaningfully from a general-purpose compute node's. Why does this matter to an infrastructure engineer, not just a facilities engineer? *Guidance:* Node density, rack planning, and PDU redundancy (Chapter 25) all follow from what's actually installed; treating GPU nodes as a drop-in equivalent to compute nodes for facility planning risks under-provisioning power/cooling.

29.4 Storage

12. A user says "the filesystem is slow." What's your first question, and why? *Guidance:* Whether another job/node accessing the same storage right now is also affected (Chapter 8) — this single check

separates a shared-resource problem from an application-specific access-pattern issue.

13. `df` shows plenty of free space, but a job can't create new files. What's happening? *Guidance:* Likely inode exhaustion (Chapter 8) — a separate accounting structure from raw capacity. Check inode usage specifically, not just space.

14. Explain why Lustre and GPFS/Storage Scale both separate metadata operations from data operations architecturally. *Guidance:* So that operations needing only metadata (a directory listing, a size check) don't compete with large data transfers for the same resource (Chapter 7) — throughput isolation, not an arbitrary design choice.

15. A storage system passes every server-side health check, but users still report intermittent hangs. What layers haven't you checked yet? *Guidance:* Client-side (stale mounts/file handles) and network-path layers (Chapter 8) — a clean server doesn't rule out a client-specific or path-specific issue.

29.5 InfiniBand, RDMA, and RoCE

16. A port shows "Active" but an application over it performs poorly. What do you check, and why doesn't "Active" settle the question? *Guidance:* `perfquery` for error counters (Chapter 12) — link state and error-free operation are related but distinct facts; Active doesn't mean error-free.

17. Explain the difference between IPoIB and native RDMA, and why an application using IPoIB might not get the performance an engineer expects from InfiniBand hardware. *Guidance:* IPoIB is a compatibility layer running through the traditional IP stack; it doesn't carry RDMA's kernel-bypass benefit (Chapter 10). The hardware supports RDMA; the application has to actually use the RDMA path for it to matter.

18. What's the practical difference between RoCEv1 and RoCEv2, and why does it matter operationally? *Guidance:* RoCEv1 is link-layer only and not routable beyond one Ethernet broadcast domain; RoCEv2 encapsulates over UDP/IP and is routable (Chapter 10) — a real deployment-scope difference, and RoCEv1 is largely legacy at this point.

19. Why does RoCE require lossless Ethernet configuration (PFC/ECN) while native InfiniBand doesn't need the equivalent configured separately? *Guidance:* InfiniBand provides link-level flow control natively; Ethernet doesn't by default, so RoCE has to configure it explicitly for RDMA's loss-sensitive assumptions to hold (Chapter 10).

20. A link is flapping. How do you determine whether it's the cable, the port, or the switch? *Guidance:* Check `iblinkinfo` from both ends of the link (Chapter 12) — if both sides show correlated flapping, suspect the cable/optic; if only one side does, suspect that side's port or switch.

21. What's the subnet manager's role, and what happens to a fabric if it fails without a working standby? *Guidance:* The SM handles addressing, routing computation, and topology response (Chapter 9/11); without an active SM, the fabric generally can't pass traffic. A candidate should note that standby failover must be tested, not just configured (Chapter 11).

22. Explain why LIDs and GUIDs are not the same kind of thing as an IP address. *Guidance:* LIDs are SM-assigned and can change after a topology event; GUIDs are hardware-fixed (Chapter 9) — neither behaves like a persistent, routable IP address, and IPoIB exists specifically to bridge that gap for IP-based tools.

23. A fabric-wide performance complaint turns out to trace to one congested link, not a broad fabric problem. How would you have found that efficiently? *Guidance:* Compare per-link utilization/error

counters (`perfquery`, or UFM's consolidated view, Chapter 22) across the fabric rather than checking nodes one at a time — an outlier link stands out quickly in an aggregated view.

29.6 GPU, NVLink, and NVSwitch

24. `nvidia-smi` shows fewer GPUs than expected on a node. What's your triage order, and why? *Guidance:* Check `lspci` first (Chapter 13/14) — if the GPU appears there, it's a driver-layer issue (faster/cheaper to fix); if it's absent even from `lspci`, it's a hardware/PCIe-layer issue.

25. Explain what an Xid error is and why you shouldn't assume its meaning from memory. *Guidance:* An Xid is a driver-logged GPU error condition (Chapter 14); codes vary widely in severity and category (hardware vs. software), and NVIDIA's official reference is the source of truth — guessing risks misclassifying a hardware fault as benign or vice versa.

26. A multi-GPU job scales poorly even though it received the correct number of GPUs. What would you check? *Guidance:* `nvidia-smi topo -m` (Chapter 15) to check whether the allocated GPUs are actually well-connected (NVLink) or only reachable via PCIe/across NUMA boundaries — a topology-aware placement gap, not a hardware fault.

27. What's the difference between NVLink and NVSwitch, architecturally? *Guidance:* NVLink is point-to-point between GPU pairs; NVSwitch extends this to all-to-all connectivity for higher GPU counts (Chapter 15) — a node's topology differs meaningfully depending on which is present.

28. Why is MIG not "a faster GPU," and what problem does it actually solve? *Guidance:* MIG partitions one physical GPU into isolated instances (Chapter 13) — it's a resource-isolation/multi-tenancy tool, not a performance feature; a MIG instance has a fraction of the full GPU's resources.

29. A GPU shows a rising count of correctable ECC errors. Is this an emergency? What would you do? *Guidance:* Not an emergency by itself, but a leading indicator (Chapter 6/14) worth acting on — track the trend, and consider a proactive RMA before it becomes an uncorrectable error.

30. Explain the three-way check (requested/allocated/visible) for diagnosing a GPU allocation problem, and why checking only one isn't enough. *Guidance:* What a job requested, what Slurm allocated, and what `CUDA_VISIBLE_DEVICES` shows the application (Chapter 19) are three independently-checkable facts — a mismatch between any two is the actual evidence, not assumed from any single one.

31. A workload runs fine at small scale but fails with GPU memory errors at production scale. What's your reasoning process? *Guidance:* Compare per-GPU memory usage across scales directly (Chapter 27) rather than assuming the small-scale test was representative — usage patterns (batching, buffering) often don't stay flat with scale.

29.7 Slurm

32. Explain the difference between `sbatch`, `srun`, and `salloc`, and when you'd use each. *Guidance:* `sbatch` submits for asynchronous batch execution; `srun` launches work directly (standalone or within an allocation); `salloc` requests an allocation without immediately running anything against it, useful for interactive sessions (Chapter 17).

33. A job is PENDING. Walk through your full diagnostic path. *Guidance:* `squeue`'s reason code first, then `scontrol show job` for detail, `sprio` for priority breakdown, and `sinfo` for whether the partition has capacity at all (Chapter 20) — in that order, evidence before conclusion.

34. Explain fairshare in your own words, and correct the common misconception about it. *Guidance:* Fairshare adjusts priority based on recent historical usage relative to allocated share (Chapter 18) — it is not "everyone gets equal priority"; heavy recent users see reduced priority, not blocked access.

35. What's the difference between backfill and preemption? *Guidance:* Backfill opportunistically uses otherwise-idle capacity for lower-priority jobs without delaying higher-priority ones; preemption forcibly reclaims already-allocated capacity from a running lower-priority job (Chapter 18) — different mechanisms for different problems.

36. A node shows DOWN. How do you distinguish a genuine hardware outage from a `slurmd` communication problem? *Guidance:* Attempt to reach the node directly (SSH, Chapter 20) — if reachable but `scontrol show node` still shows DOWN, suspect `slurmd` itself rather than the node's hardware; check whether the service is running.

37. Explain why canceling and resubmitting a pending job is often the wrong first move. *Guidance:* It treats the symptom, not the cause (Chapter 20/1) — if the underlying pending reason (a QOS limit, resource contention) hasn't changed, the resubmitted job will likely pend for the same reason.

38. A GPU job's allocation looks correct in `scontrol show job`, but the application reports no GPUs available. What's your next step? *Guidance:* Check the node's actual GPU inventory directly (`nvidia-smi`, Chapter 20) against what Slurm believes it has (GRES) — a stale-GRES-state mismatch after a reboot is a known pattern, distinct from an allocation-request error.

39. What's the architectural difference between `slurmctld`, `slurmd`, and `slurmdbd`, and why does that separation matter for troubleshooting? *Guidance:* Controller decides, compute daemon executes, accounting database records (Chapter 16) — each fails

independently, so a symptom in one doesn't necessarily implicate the others; diagnosing correctly means knowing which component actually owns the behavior in question.

29.8 UFM

40. How does UFM relate to the subnet manager — is it a replacement? *Guidance:* No — UFM is a monitoring/management layer; the subnet manager (Chapter 9/11) is what makes the fabric operate at all. UFM consolidates and makes proactive largely the same data manual tools also access (Chapter 22).

41. When would you start a fabric investigation with UFM versus jumping straight to Chapter 12's manual tools? *Guidance:* UFM is a good starting point for "broadly, where's the problem" across a large fabric; once a specific link/port/switch is implicated, detailed isolation typically still uses the manual toolset and methodology (Chapter 22).

29.9 NCCL

42. Why is it inaccurate to call NCCL "MPI for GPUs"? *Guidance:* MPI is a general-purpose message-passing standard used across HPC broadly; NCCL is a GPU-specific, topology-aware collective communication library (Chapter 23). NCCL's API follows conventions MPI popularized, which is why they're confused, but neither replaces the other — real stacks may use MPI, NCCL, or both together depending on architecture.

43. An NCCL job hangs during a collective operation. How do you distinguish a slow/stuck rank from a genuine network fault? *Guidance:* Check whether one specific rank is behind (still computing, or stuck) while every other rank waits, versus a uniform, symmetric delay across all ranks (Chapter 24) — the pattern of who's waiting on whom is the key evidence.

44. An NCCL error references the network layer. What's your first move, and why shouldn't you treat this as an NCCL bug by default? *Guidance:* Enable `NCCL_DEBUG` logging and check for a node/link pattern (Chapter 24); if the fabric layer is implicated, hand off to Chapter 12's InfiniBand methodology directly rather than continuing to investigate NCCL itself — NCCL is frequently the messenger, not the source.

45. Explain how NCCL selects a transport between two GPUs, and why the same job might behave differently within one node versus across nodes. *Guidance:* NCCL inspects what's actually available — NVLink/NVSwitch if present, otherwise PCIe within a node, InfiniBand/RoCE between nodes (Chapter 23) — and picks accordingly. A performance difference between intra-node and inter-node communication can be an expected consequence of transport choice, not a fault.

46. A distributed job's per-GPU throughput drops as it's scaled to more nodes, beyond what communication overhead should explain. What would you investigate? *Guidance:* Whether the job now crosses additional fabric switch tiers it didn't at smaller scale (Chapter 27) — a topology-crossing cost that small-scale testing wouldn't have exercised, rather than assuming a code or NCCL regression.

29.10 Data Center Operations

47. Why does cabling/labeling discipline matter for troubleshooting speed, not just tidiness? *Guidance:* Fault isolation (Chapter 12) often requires quickly identifying a specific physical connection; untraceable cabling turns a quick lookup into a manual trace, directly slowing incident resolution (Chapter 25).

48. A node passes burn-in cleanly but fails weeks later under real production load. Does this mean burn-in failed as a process? *Guidance:* Not necessarily — burn-in reduces risk, it doesn't

eliminate it (Chapter 25); some conditions only emerge under sustained real-world usage patterns burn-in's finite test window doesn't fully replicate.

49. Why is asset tracking described as a prerequisite for firmware-baseline enforcement, rather than a separate concern? *Guidance:* Without accurate records of what's deployed where, questions like "how many nodes still have outdated firmware" have no reliable answer (Chapter 25) — enforcement depends on visibility, which asset tracking provides.

29.11 Cross-Domain Architecture and Diagnostic Reasoning

50. A multi-node training job fails intermittently with a communication error. Walk through your full diagnostic approach across layers. *Guidance:* A strong answer gathers evidence from GPU (Chapter 14), NCCL (Chapter 24), and fabric (Chapter 12) layers in parallel rather than exhausting one before considering another, since the failure pattern doesn't cleanly implicate a single layer up front (Chapter 27's cross-layer scenario is the model case).

51. How would you distinguish a Slurm allocation problem from a GPU hardware/driver problem when a GPU job fails? *Guidance:* Check whether the allocation itself (Chapter 19) matches the request — if it does, and the application still fails, the problem has moved into Chapter 14's territory (driver/CUDA/hardware), not Slurm's.

52. Describe a scenario where a symptom's apparent domain and its actual root-cause domain are different, and explain how you'd catch that. *Guidance:* A "node unavailable" report (Chapter 26) can trace to hardware, provisioning, or fabric — the right approach is

systematic elimination by layer (host reachability, hardware, scheduler communication, then fabric) rather than assuming the first plausible domain.

53. What's the difference between monitoring and troubleshooting, and why does conflating them cause problems? *Guidance:* Monitoring (Chapter 21) tells you something changed; troubleshooting (Chapters 4, 8, 12, 14, 20, 24) tells you why. Treating a monitoring alert as a diagnosis skips the evidence-gathering step this book's whole methodology is built around.

54. How would you design a new-node-to-production checklist that spans hardware, network, and scheduler visibility? *Guidance:* A strong answer sequences burn-in/validation (Chapter 25), fabric/topology confirmation (Chapter 9-15), and scheduler-visibility confirmation (Chapter 20/26's provisioning-gap scenario) — recognizing that passing one doesn't imply passing the others.

55. You're told "the cluster is slow" with no further detail. What are your first three questions? *Guidance:* (1) Slow for everyone, or one user/job? (2) Since when, and did anything change around that time? (3) Slow at which layer — compute, storage, fabric, or scheduling wait time? A strong answer treats "slow" as a symptom requiring the same Observe→Collect Evidence→Isolate discipline (Chapter 1) as any other vague report, not a specific diagnosis in itself.

Back Matter

Glossary

Definitions below are consolidated from how each term is actually defined and used in the manuscript — none are newly invented for this glossary. Where a term wasn't used in the book, that's stated explicitly rather than filled in.

BMC (Baseboard Management Controller) — A small, independent management processor included on most servers that can report on hardware health even when the main operating system is unresponsive or hasn't booted, capturing events (in a System Event Log, SEL) the OS-level log may never see. (Chapter 6)

CPU — Central Processing Unit; the general-purpose processor handling process scheduling, I/O, and orchestration on a node, distinct from a GPU's parallel, accelerator-bound role. (Chapters 5, 13)

CUDA — NVIDIA's GPU computing platform. This book distinguishes the CUDA *driver* (kernel-level, makes the GPU visible to the OS), the CUDA *runtime* (what applications link against), and the CUDA *toolkit* (used to build applications) as related but distinct, version-sensitive layers. (Chapter 13)

DCGM (Data Center GPU Manager) — NVIDIA's tool for systematic, fleet-wide GPU health checking and monitoring, distinct from `nvidia-smi`'s single-node snapshot view. Its diagnostic levels (`dcgmi diag -r 1/2/3`) range from quick readiness checks to comprehensive, GPU-stressing validation. (Chapters 13, 14)

ECN (Explicit Congestion Notification) — A mechanism that signals network congestion before it becomes severe enough to require pausing traffic, used alongside PFC to make Ethernet behave losslessly for RoCE traffic. Kept conceptual throughout this book, by design. (Chapter 10)

GID (Group ID) — A Linux identifier for a group a user belongs to, reported by the `id` command alongside a user's UID; relevant in HPC because permission errors on shared, multi-tenant nodes are often a group-membership issue rather than a missing privilege. (Chapter 3)

GPFS / IBM Storage Scale — A parallel filesystem (GPFS is the former name; IBM Storage Scale is the current name) that distributes both data and metadata handling across Network Shared Disks (NSDs), rather than concentrating metadata on dedicated servers the way Lustre does. (Chapter 7)

GPU — Graphics Processing Unit; in this book's context, an accelerator handling parallel, accelerator-bound computation, connected to the rest of a node via PCIe and, on multi-GPU nodes, potentially NVLink. (Chapters 6, 13)

GRES (Generic Resource) — Slurm's general mechanism for representing a schedulable resource beyond CPU and memory — GPUs are the primary HPC example, requested via flags such as `--gres=gpu:N` or the GPU-specific `--gpus*` family (which requires the `select/cons_tres` plugin). (Chapters 16, 19)

GUID (Globally Unique Identifier) — A fixed, hardware-assigned InfiniBand identifier, similar in spirit to an Ethernet MAC address — unlike a LID, it doesn't change after a topology event. Not an IP address. (Chapter 9)

HBA — Not used in this book. (The book uses HCA for InfiniBand network adapters; HBA typically refers to Fibre Channel storage adapters, outside this book's scope.)

HCA (Host Channel Adapter) — The network interface card connecting a node to an InfiniBand fabric — the InfiniBand equivalent, in role, of an Ethernet NIC, typically supporting RDMA directly in hardware. (Chapter 9)

HPC (High Performance Computing) — This book's working definition, from an operations perspective: a set of systems (compute, storage, networking, scheduling) that fail in specific, recognizable ways, which an infrastructure engineer learns to operate, troubleshoot, diagnose, and maintain — as opposed to the academic definition of aggregated computing resources solving problems too large for a single machine. (Chapter 1)

IPoIB (IP over InfiniBand) — A compatibility layer presenting an InfiniBand fabric to the OS as an ordinary IP-capable network interface. It provides connectivity over InfiniBand hardware but does not carry RDMA's kernel-bypass performance benefit. (Chapters 9, 10)

LID (Local Identifier) — A subnet-manager-assigned InfiniBand address used for routing traffic within a subnet. Unlike a GUID, it can change if the SM reassigns it after a topology change. Not an IP address. (Chapter 9)

Lustre — A parallel filesystem that explicitly separates metadata (via Metadata Servers/Targets, MDS/MDT) from data (via Object Storage Servers/Targets, OSS/OST), allowing large, sequential I/O to scale across many OSTs in parallel. (Chapter 7)

MPI (Message Passing Interface) — A long-standing, general-purpose standard for message passing in parallel and distributed computing, implemented by libraries such as Open MPI and MPICH, used across HPC applications broadly. Distinct from NCCL — the two are not interchangeable, and this book explicitly rejects the framing of NCCL as "MPI for GPUs." (Chapters 1, 23)

NCCL (NVIDIA Collective Communications Library) — A GPU-specific, topology-aware library for collective and point-to-point communication between GPUs (AllReduce, Broadcast, Reduce, AllGather). Its API closely follows conventions MPI popularized, which is why the two are often confused. (Chapters 1, 23, 24)

NFS (Network File System) — A shared filesystem model where a server exports a directory and clients mount it over the network; in its traditional form, one server handles both metadata and data operations for what it exports. (Chapter 7)

NUMA (Non-Uniform Memory Access) — The property that, on a multi-socket server, the cost of a memory access depends on where the memory physically lives relative to the requesting CPU — local (same socket) access is faster than remote (different socket) access. (Chapter 5)

NVLink — A high-bandwidth, point-to-point interconnect connecting GPU pairs directly, bypassing PCIe for that connection. Not every GPU pair on a node is necessarily NVLink-connected. (Chapter 15)

NVSwitch — A switch specifically for NVLink traffic, giving every GPU behind it an equally fast, all-to-all path to every other GPU behind it, rather than the point-to-point topology plain NVLink pairs provide. (Chapter 15)

OS (Operating System) — Used throughout in its general sense; this book's Linux-specific content (Part 2) assumes a RHEL-family distribution unless otherwise noted. (Chapters 3, 6, 21, 25)

PDU (Power Distribution Unit) — Delivers power to a rack's equipment, typically with redundancy — but redundancy only holds if each PSU's feed traces to a genuinely independent upstream source (an A/B feed), not just a separate outlet on the same circuit. (Chapter 25)

PFC (Priority Flow Control) — A mechanism letting an Ethernet switch pause a specific traffic class rather than drop packets from it under congestion, used to make Ethernet behave losslessly for RoCE traffic. Kept conceptual throughout this book, by design. (Chapter 10)

PKey (Partition Key) — Used to partition an InfiniBand fabric into logical groups, controlling which ports can communicate with which others — conceptually similar in purpose to VLANs on Ethernet, though implemented differently. (Chapters 9, 11)

PSU (Power Supply Unit) — Most HPC-grade servers ship with redundant PSUs; redundancy only provides real protection if each PSU connects to an independently-sourced power feed, not the same upstream circuit. (Chapter 25)

RDMA (Remote Direct Memory Access) — Lets a network adapter move data directly between one machine's memory and another's, bypassing the CPU and the kernel network stack (kernel bypass) on both sides — the basis of InfiniBand's and RoCE's performance advantage. (Chapters 9, 10)

RHEL (Red Hat Enterprise Linux) — The RHEL-family distribution lens this book's Linux content (Part 2) is written from, stated explicitly rather than assumed silently. (Chapter 3)

RMA (Return Merchandise Authorization) — The formal process of returning a failed hardware component to the vendor for replacement or repair, requiring documentation of the failure evidence, timeline, and component identification. (Chapters 6, 25)

RoCE (RDMA over Converged Ethernet) — Carries RDMA semantics over an Ethernet fabric instead of an InfiniBand one. RoCEv1 is link-layer-only and largely legacy; RoCEv2 encapsulates over UDP/IP and is routable. Requires a lossless-Ethernet configuration (PFC/ECN) that native InfiniBand doesn't need in the same way. (Chapter 10)

QOS (Quality of Service) — In Slurm, a policy overlay assignable to a job or account that can adjust priority or impose additional resource limits, distinct from fairshare (which adjusts priority based on historical usage). (Chapter 18)

UFM (Unified Fabric Manager) — NVIDIA's fabric monitoring and management platform, offered in multiple tiers (including a telemetry-focused tier and a more full-featured "Enterprise" tier, among other offerings). Complements, but does not replace, the subnet manager. (Chapters 9, 22)

References & Further Reading

The commands, tools, and technical claims in this book were checked against the official documentation and reference sources listed below. The book presents these concepts, procedures, and troubleshooting guidance in the author's own explanatory framework; readers should consult the cited authoritative sources for current product behavior and licensing terms.

NVIDIA Documentation

- NVIDIA Xid Errors documentation — docs.nvidia.com/deploy/xid-errors
- NVIDIA CUDA documentation — docs.nvidia.com/cuda
- NVIDIA Driver release documentation — docs.nvidia.com
- NVIDIA DGX/HGX system documentation — docs.nvidia.com
- NVIDIA Data Center GPU Manager (DCGM) documentation — docs.nvidia.com/datacenter/dcgm
- NVIDIA NVLink/NVSwitch architecture documentation — docs.nvidia.com
- NVIDIA GPUDirect RDMA documentation — docs.nvidia.com

Slurm / SchedMD Documentation

- Slurm Overview and Architecture documentation — slurm.schedmd.com
- `sbatch` documentation — slurm.schedmd.com/sbatch.html
- `squeue` documentation — slurm.schedmd.com/squeue.html
- GRES (Generic Resource) documentation — slurm.schedmd.com/gres.html
- Slurm documentation on partitions, QOS, fair share, backfill, preemption, reservations, and job dependencies — slurm.schedmd.com

Linux / Red Hat Documentation

- Linux man-pages project — `dmesg(1)`, `syslog(2)` — man7.org
- `sudo(8)` man page — man7.org
- `whoami(1)` man page — man7.org
- OpenSSH `ssh-keygen(1)` and `sshd(8)` documentation — man.openbsd.org
- Red Hat documentation on DNF/YUM package management — docs.redhat.com
- Red Hat Enterprise Linux System Administrator's Guide — access.redhat.com
- kernel.org documentation, including `Documentation/filesystems/proc.rst` and the `kernel.dmesg_restrict` sysctl
- Apptainer official documentation — apptainer.org
- Lmod official documentation — lmod.readthedocs.io

RDMA / InfiniBand Documentation

- `infiniband-diags` project documentation (linux-rdma) — covers `ibstat`, `ibstatus`, `iblinkinfo`, `ibportstate`, `ibdiagnet`, `perfquery`, `ibnetdiscover`, `ibswitches`
- NVIDIA MLNX_OFED documentation — docs.nvidia.com/networking
- InfiniBand Trade Association (IBTA) specifications — infinibandta.org
- `rdma-core` project documentation — github.com/linux-rdma/rdma-core
- NVIDIA Networking RoCE documentation — docs.nvidia.com/networking

NVIDIA UFM Documentation

- NVIDIA UFM Enterprise User Manual — docs.nvidia.com/networking

- NVIDIA UFM Telemetry documentation — docs.nvidia.com/networking
- NVIDIA UFM Cyber-AI product documentation — docs.nvidia.com/networking

CUDA / NCCL Documentation

- NVIDIA NCCL official documentation — docs.nvidia.com/deeplearning/nccl
- MPI Forum — mpi-forum.org

Storage Documentation

- Lustre Operations Manual — Whamcloud / OpenSFS
- IBM Storage Scale (GPFS) documentation — ibm.com
- Red Hat NFS administration documentation — access.redhat.com

This list reflects the official vendor and project documentation this book's commands, tool behavior, and technical terminology were checked against. Readers should always confirm current behavior against the linked authoritative source for their own environment and software version before relying on it in production, per the Disclaimer.

About the Author

Subhani Roshan

Subhani Roshan is an HPC and data center infrastructure professional with hands-on experience supporting Linux-based HPC and AI infrastructure. His areas of experience include InfiniBand networking, NVIDIA GPU infrastructure, Slurm, UFM, cluster operations, hardware troubleshooting, and data center operations. Through this handbook, he shares a practical, operations-focused approach to diagnosing, troubleshooting, and maintaining modern HPC infrastructure.

Continue Your HPC Learning

This book covers the operational depth an HPC infrastructure engineer needs across Linux, storage, InfiniBand, NVIDIA GPUs, Slurm, and NCCL — but no single book, and no single read-through, covers everything a fast-moving field like HPC and AI infrastructure will ask of you over a career.

If you found this handbook useful, you can continue building on it:

- Revisit Chapter 29's interview and scenario questions periodically — reasoning skills sharpen with repetition, not a single pass.
- Use Chapter 28's cheat sheet as a living reference on the job, not just while reading.
- Where this book flagged a claim as version-sensitive or configuration-dependent, make a habit of checking your own environment's current behavior against the authoritative source named — that habit outlasts any single book's shelf life.

Thank you for reading.

Subhani Roshan