Telemetry, Profiling & Tools

You Cannot Optimize What You Cannot Measure — Building the Full Observability Stack from Silicon Counters to Fleet Dashboards

Part VIII · Observability & Architecture · ~42 pages · Prerequisites: Ch4 (HBM), Ch7 (Kernels), Ch12 (Benchmarking), Ch14 (Networking)

> *"The difference between a 40% MFU cluster and a 65% MFU cluster is usually not hardware. It is the quality of the observability stack that lets engineers see exactly where compute is being wasted."*
>
> — Production AI Infrastructure Principle

A 10,000-GPU training cluster generates tens of thousands of metrics per second from every GPU, NIC, switch, and storage node. Without the right observability stack, a silent performance regression can waste millions of dollars before anyone notices. In one production incident at a large hyperscaler, a driver update degraded Tensor Core utilization by 18%—dropping MFU from 58% to 47%—for 11 days before an engineer noticed the training loss curve was converging slower than expected. By then, the cluster had burned an estimated $2.1M in wasted compute at $3/GPU-hr. With a 15-second DCGM polling cycle feeding a Prometheus alert on `DCGM_FI_PROF_PIPE_TENSOR_ACTIVE`, the regression would have paged on-call within 4 minutes of the driver rollout.

This chapter builds the complete observability architecture from silicon hardware counters to user-facing dashboards and automated regression CI/CD pipelines. Every tool is covered in depth—when to use it, what specific metrics to examine, and what the numbers mean in context of real hardware behavior.

LEARNING OBJECTIVES BY LEVEL

FOUNDATIONAL

- Understand the seven layers of AI system observability and why each layer exists
- Know what DCGM measures and why GPU hardware counters differ from framework-level metrics
- Distinguish when to use Nsight Systems (timeline) vs. Nsight Compute (kernel counters)

INTERMEDIATE

- Configure a complete DCGM → Prometheus → Grafana pipeline with correct scrape intervals and retention
- Write PromQL alert rules that fire before SLA violations occur, not after
- Interpret an Nsight Systems timeline to identify communication/compute overlap gaps

PRINCIPAL LEVEL

- Design the full observability architecture for a 10,000-GPU cluster with metric cardinality management
- Build automated performance regression CI/CD that gates driver/framework updates statistically

• Choose the right tool at each scope: silicon, kernel, job, and fleet — and explain the cost/benefit tradeoffs

17.1 The Observability Architecture

Before writing a single PromQL query, you need a mental model of *why* the observability stack has multiple layers. Each layer exists because no single tool can efficiently capture information at all granularities simultaneously. A hardware counter that fires every GPU clock cycle (1.98 GHz on H100) would generate 1.98 billion samples per second per GPU—physically impossible to collect. Each layer is a different sampling rate, storage cost, and precision/recall tradeoff.

17.1.1 The Seven-Layer Telemetry Stack

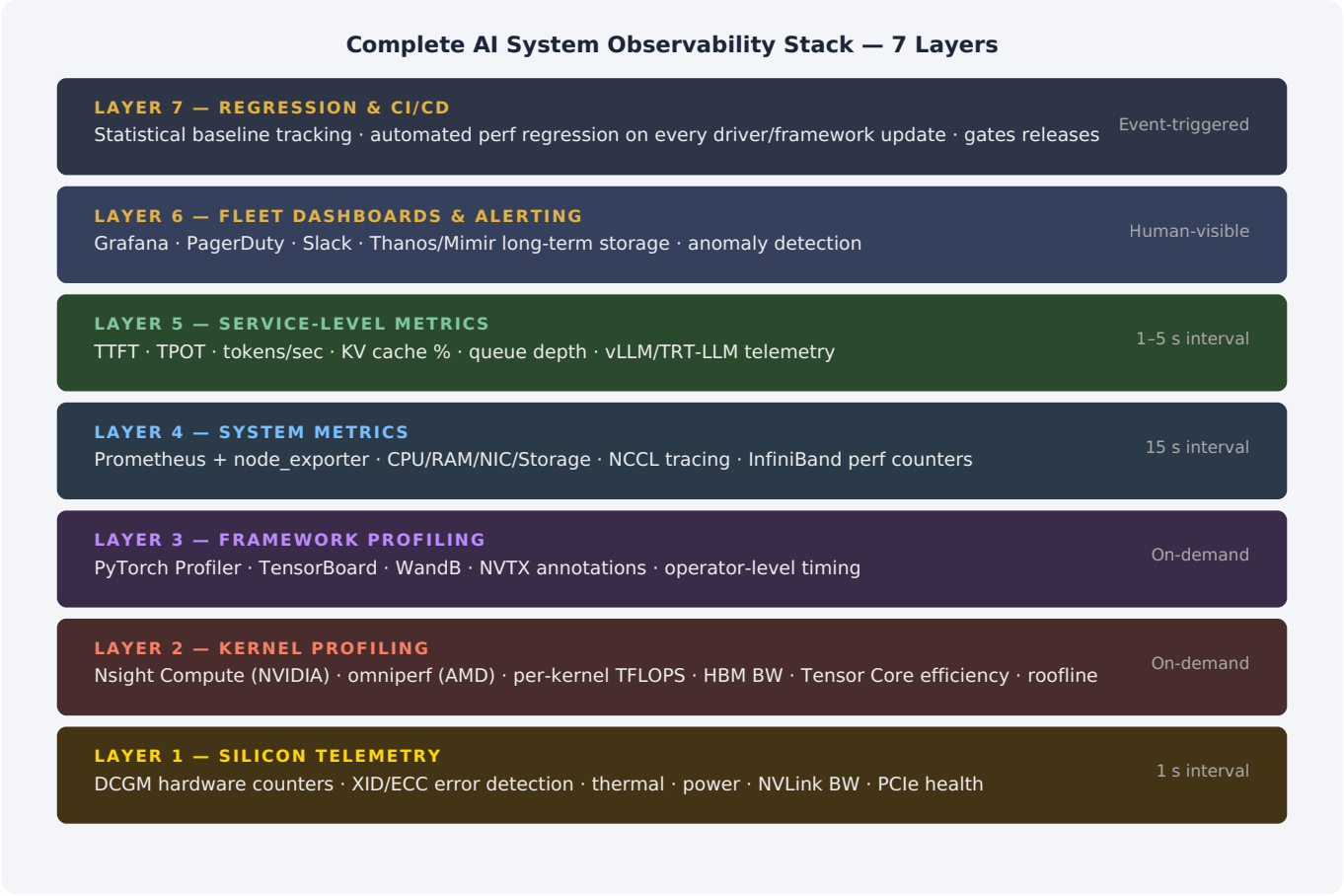


Figure 17.1 — The seven-layer AI observability stack. Lower layers sample faster and are more precise; upper layers aggregate and make data human-actionable. Layers 1–4 are always-on; Layers 2–3 are on-demand due to high overhead.

17.1.2 The Critical Insight: Tool Selection by Scope

The most common mistake engineers make is using a kernel profiler (Nsight Compute) when they should be using a timeline profiler (Nsight Systems), or the reverse. The choice depends on what question you're answering:

Question	Correct Tool	Why
"Why is my training step 40% slower than expected?"	**Nsight Systems (timeline)**	Need to see time decomposition — what fraction is compute vs. AllReduce vs. data loading
"Why is my attention kernel running at 45% of peak TFLOPS?"	**Nsight Compute (counters)**	Need kernel-level metrics: SM efficiency, HBM BW saturation, occupancy
"Is GPU 47 overheating on node 23?"	**DCGM / Prometheus**	Fleet-level health; no profiler overhead; persistent monitoring
"Did our P99 TTFT SLA degrade after the vLLM update?"	**vLLM metrics → Prometheus → Grafana**	Service-level; needs percentile histograms over time
Using Nsight Compute for every regression	**Wrong**	Adds 3–10× overhead to kernel execution; cannot run in production

> **PRINCIPAL INSIGHT: PROFILING OVERHEAD IS NEVER FREE**
>
> Nsight Compute adds 3–10× execution overhead per kernel and is not suitable for production or long-horizon profiling. It also serializes kernel launches, preventing overlap detection. Nsight Systems has much lower overhead (~2–5% CPU, negligible GPU) but still should not run continuously. DCGM runs continuously with <0.5% GPU overhead — it is the only always-on tool for silicon metrics.

17.2 Silicon Layer — DCGM Hardware Counters

NVIDIA Data Center GPU Manager (DCGM) is the foundation of GPU cluster observability. It reads hardware performance counters directly from GPU silicon — registers built into the chip that increment as hardware events occur — at 1-second intervals with less than 0.5% GPU utilization overhead. Understanding DCGM is not optional for AI infrastructure engineers: it is the first diagnostic tool to open in every performance investigation.

17.2.1 Critical DCGM Metrics — What They Measure and Why

```
# ── TIER 1: Must monitor always-on ──────────────────────────────────

DCGM_FI_DEV_GPU_UTIL              SM utilization %  (0–100)
  └─ % of time ≥1 warp is active on ≥1 SM
  └─ <60% with a running job → launch gap, comm stall, or data wait
  └─ Does NOT mean compute-bound: memory-bound kernels can show 100% SM util

DCGM_FI_PROF_PIPE_TENSOR_ACTIVE   Tensor Core active fraction  (0–1)
  └─ Fraction of SMs running MMA instructions
  └─ <0.30 = Tensor Cores underutilized → shape mismatch or scalar ops dominating
  └─ Healthy range: 0.50–0.80 for mixed training workloads

DCGM_FI_PROF_DRAM_ACTIVE          HBM bandwidth utilization  (0–1)
  └─ Fraction of DRAM cycles with active data transfer
  └─ >0.85 = memory-bound (check roofline)
  └─ Combined with TENSOR_ACTIVE: if both high → hardware is efficient

DCGM_FI_DEV_GPU_TEMP              Junction temperature °C
  └─ H100 SXM max: 83°C; throttle onset: ~80°C
  └─ Alert threshold: 82°C (before throttle, not after)

DCGM_FI_DEV_POWER_USAGE           Instantaneous power draw W
  └─ H100 SXM TDP: 700W; alert if >660W sustained for >60s

DCGM_FI_DEV_ECC_SBE_VOL_TOTAL     Single-bit ECC errors (correctable)
  └─ 1 SBE/hour = early hardware failure warning
  └─ Schedule GPU replacement within 48–72h

DCGM_FI_DEV_ECC_DBE_VOL_TOTAL     Double-bit ECC errors (UNCORRECTABLE)
  └─ ANY DBE = immediate GPU quarantine + job evacuation

# ── TIER 2: Interconnect health ─────────────────────────────────────

DCGM_FI_PROF_NVLINK_TX_BYTES      NVLink TX bytes/sec
  └─ H100 NVLink 4.0: 900 GB/s total; per-link ~112 GB/s
  └─ <50% during AllReduce → NVLink link failure or TP misconfiguration

DCGM_FI_PROF_PCIE_TX_BYTES        PCIe TX bytes/sec (GPU→CPU)
  └─ H100 PCIe gen5 x16: ~64 GB/s
  └─ High PCIe BW with low NVLink BW → data incorrectly routed via CPU
```

17.2.2 XID Error Codes — Production Decision Tree

XID errors are GPU hardware exception codes reported in the kernel log and via DCGM. Not all XIDs require the same response. Misinterpreting them is a common operations failure mode.

XID Code	Meaning	Production Action
`XID 48`	Double-bit ECC (DBE) — uncorrectable memory error	**IMMEDIATE: Evacuate jobs, quarantine GPU, replace within 24h**
`XID 63`	Single-bit ECC spike — correctable, but HW degrading	**Schedule replacement within 48–72h; monitor hourly rate**
`XID 74`	NVLink Error — data integrity or link failure	**Check NVLink link state; may cause silent data corruption in TP**
`XID 79`	GPU has fallen off the bus (catastrophic)	**IMMEDIATE: Node requires physical inspection and reboot**
`XID 45`	Preemptive cleanup — typically job timeout	**Normal; verify job scheduler timeout settings**
`XID 31`	GPU memory page fault (driver bug or OOM)	**Check for OOM in application logs; may indicate memory leak**

> **SILENT CORRUPTION RISK: XID 74**
>
> NVLink errors (XID 74) are particularly dangerous in tensor-parallel training. A partial NVLink failure can deliver wrong values silently — AllReduce succeeds but accumulates incorrect gradients. Training loss will appear to converge but the model will be corrupt. Any XID 74 on a TP training job requires immediate job restart and a 30-minute NVLink bandwidth test before resuming.

17.2.3 DCGM Deployment and Configuration

```
# ── Kubernetes DaemonSet (recommended for container clusters) ──────────────
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: dcgm-exporter
  namespace: monitoring
spec:
  selector:
    matchLabels:
      app: dcgm-exporter
  template:
    spec:
      nodeSelector:
        nvidia.com/gpu: "true"
      containers:
      - name: dcgm-exporter
        image: nvcr.io/nvidia/k8s/dcgm-exporter:3.3.5
        ports:
        - containerPort: 9400   # Prometheus scrapes this port

# ── Bare-metal (SLURM clusters) ────────────────────────────────────────────
dcgmi discovery -l                    # List all GPUs
dcgmi diag -r 1                       # Quick health check (30 sec)
dcgmi diag -r 3                       # Full validation (20 min, pre-job)
dcgm-exporter --address :9400 &       # Start Prometheus exporter
nv-hostengine &                       # Required background daemon

# ── Verify DCGM is collecting ──────────────────────────────────────────────
curl http://localhost:9400/metrics | grep DCGM_FI_DEV_GPU_UTIL
# Should return: DCGM_FI_DEV_GPU_UTIL{gpu="0",UUID="GPU-..."} 87
```

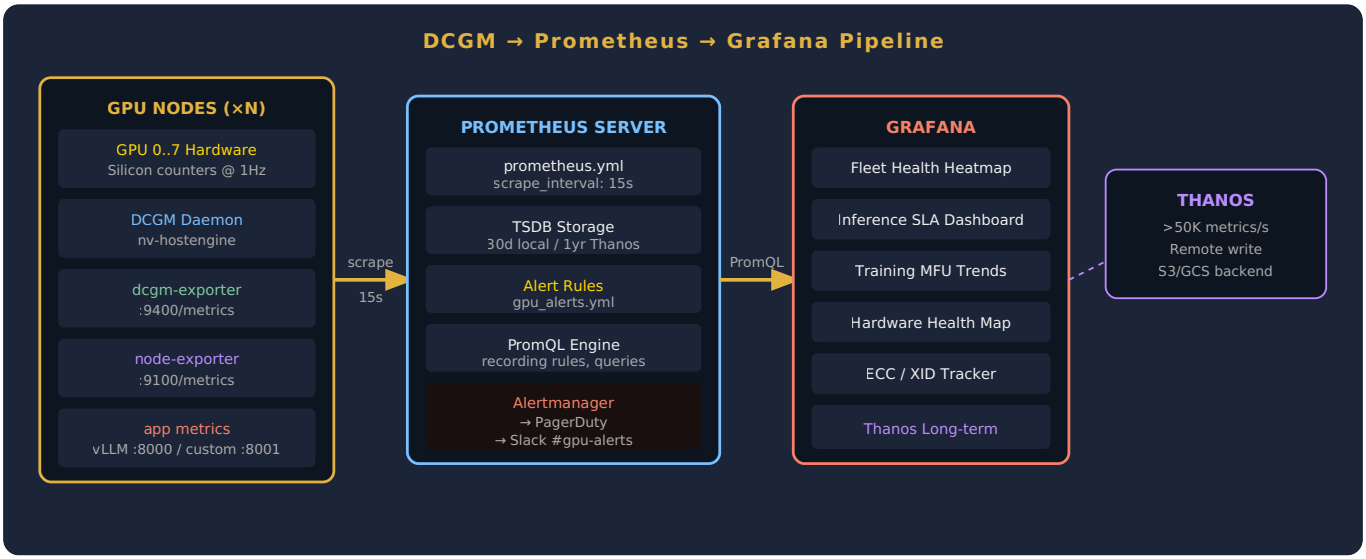


Figure 17.2 — DCGM → Prometheus → Grafana data flow. Every GPU node runs DCGM daemon + dcgm-exporter. Prometheus scrapes all exporters every 15 seconds. For clusters >10K GPUs, Thanos handles remote write and long-term storage.

17.3 Kernel Layer — Nsight Compute & Nsight Systems

When DCGM tells you something is wrong (e.g., Tensor Core utilization dropped from 68% to 41%), Nsight tools answer the follow-up question: *exactly which kernel, and why?* NVIDIA provides two complementary tools with fundamentally different purposes that engineers routinely confuse:

Tool	Question Answered	Granularity	Overhead	Use Case
Nsight Systems (nsys)	"Where is time going in my pipeline?"	Timeline: CPU+GPU+NCCL	2–5% (safe for most workloads)	Step decomposition, overlap gaps, CPU stalls
Nsight Compute (ncu)	"Why is this specific kernel slow?"	Per-kernel hardware counters	3–10× slowdown	Roofline analysis, occupancy, memory efficiency

17.3.1 Nsight Systems — Timeline Profiling

```
# ── Basic timeline profile ──────────────────────────────────────────
nsys profile     --trace=cuda,nvtx,nccl,osrt \  # CUDA + annotations + NCCL + OS runtime
    --sample=cpu \                 # Host-side CPU sampling
    --output=profile_%p \          # profile_<pid>.nsys-rep
    --stats=true     python train.py --steps 15

# ── NVTX annotations (add to training loop) ─────────────────────────
import torch
for step, batch in enumerate(dataloader):
    with torch.cuda.nvtx.range("DataLoading"): data = prepare_batch(batch)
    with torch.cuda.nvtx.range("Forward"):
        with torch.cuda.nvtx.range("Attention"): attn = attention(data)
        with torch.cuda.nvtx.range("FFN"):      ffn  = ffn(attn)
        loss = criterion(ffn, targets)
    with torch.cuda.nvtx.range("Backward"):  loss.backward()
    with torch.cuda.nvtx.range("AllReduce"):  optimizer.step()

# ── Reading the stats summary (--stats=true output) ─────────────────
# CUDA Kernel Summary:
#   volta_s884gemm_128x128_ldg8_f2f_tn  48.2%   ← dominant: GEMM kernel OK
#   ncclAllReduceRingLLKernel           31.1%   ← AllReduce: ~1/3 of step time
#   void at::native::elementwise_kernel   8.4%   ← scalar elementwise: investigate
```

17.3.2 Nsight Systems Timeline — What to Look For

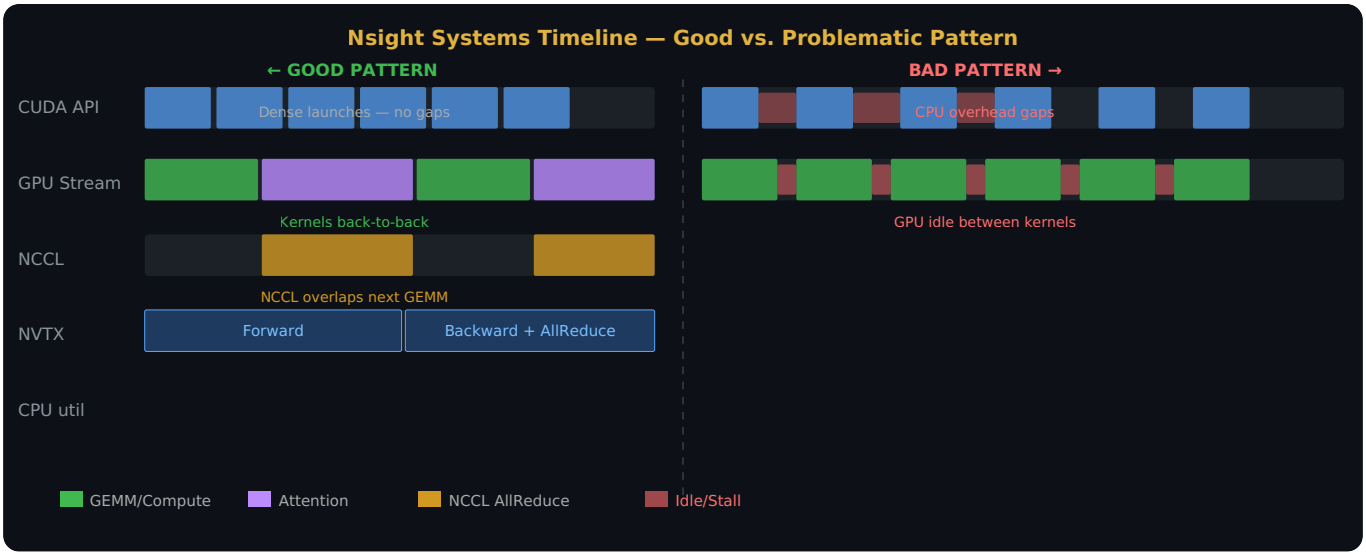


Figure 17.3 — Nsight Systems timeline patterns. Left (good): dense kernel launches, NCCL overlapping with compute, no GPU idle gaps. Right (bad): CPU launch overhead creates idle gaps between GPU kernels — the GPU is stalling waiting for the host to issue the next kernel.

17.3.3 Nsight Compute — Kernel-Level Analysis

```
# ── Profile a specific kernel ──────────────────────────────────────
ncu     --kernel-name "flash_attention_fwd"     --launch-skip 5 \              # Skip 5 warmup launches
    --launch-count 3 \             # Profile 3 launches
    --set full     --section SpeedOfLight_RooflineChart     --section MemoryWorkloadAnalysis     -o attention_kernel

# ── Key metrics to check in the output ─────────────────────────────
# sm__throughput.avg.pct_of_peak_sustained_elapsed    → SM throughput %
# l1tex__t_bytes_pipe_lsu_mem_global_op_ld.avg        → L1 load bytes
# dram__bytes_read.avg                                → HBM read bytes
# sm__pipe_tensor_cycles_active.avg                   → TC cycles active

# ── Roofline interpretation ────────────────────────────────────────
# Compute-bound kernel (AI > ridge_point ~295 FLOP/byte on H100):
#   → Optimize for throughput: increase batch size, tile sizes
#   → Check TC utilization: if low, verify BF16 dtype and tensor dimensions
#
# Memory-bound kernel (AI < ridge_point):
#   → Optimize for BW: reduce HBM traffic (fusion, quantization)
#   → Increase arithmetic intensity: fuse elementwise ops into GEMM epilogue

# ── Compare profiles (regression detection) ────────────────────────
ncu-ui --diff baseline.ncu-rep current.ncu-rep
# Red lines = regressions, Green lines = improvements
```

17.4 Framework Layer — PyTorch Profiler & WandB

The framework layer sits between kernel-level profiling and system-level monitoring. PyTorch Profiler captures operator-level timing—*which PyTorch operations took how long, on which device*—without the physical kernel counter overhead of Nsight Compute. It is the right tool when you know a training step is slow but don't yet know whether the bottleneck is a specific operator, data loading, or gradient synchronization.

17.4.1 PyTorch Profiler — Production Usage

```
import torch
from torch.profiler import profile, ProfilerActivity, schedule, tensorboard_trace_handler

# ── Recommended production pattern: skip warmup, profile specific steps ──
prof_schedule = schedule(
    wait=2,        # Skip first 2 steps (warmup)
    warmup=1,      # 1 step profiler warmup
    active=3,      # Profile 3 steps
    repeat=1
)

with profile(
    activities=[ProfilerActivity.CPU, ProfilerActivity.CUDA],
    schedule=prof_schedule,
    on_trace_ready=tensorboard_trace_handler('./prof_logs'),
    record_shapes=True,      # Shape info for each op (adds ~5% overhead)
    with_stack=True,         # Python call stack (adds ~15% overhead)
    with_flops=True,         # Estimated FLOPS per op
    profile_memory=True,     # Memory allocation tracking
) as prof:
    for step, batch in enumerate(dataloader):
        loss = model(batch)
        loss.backward()
        optimizer.step()
        prof.step()          # Required: advance profiler state

# ── Print summary ──────────────────────────────────────────────────
print(prof.key_averages().table(
    sort_by="cuda_time_total",  # Sort by GPU time
    row_limit=20
))
# Output format:
# Name                          CPU%   CUDA%    #Calls   FLOPS
# aten::mm                      1.2%   38.4%    2048     ---
# aten::scaled_dot_product...   0.8%   21.2%    512      ---
# nccl:all_reduce               0.1%   18.6%    128      ---
# aten::layer_norm              0.4%   4.1%     2048     ---
# ProfilerStep*                 99.5%  100.0%   1        ---  <-- total
```

> **WHAT TO LOOK FOR IN PROFILER OUTPUT**
>
> If `nccl:all_reduce` exceeds 20% of CUDA time, you are communication-bottlenecked (check Ch14 for AllReduce optimization). If `aten::copy_` or `aten::to` operations appear in top-10, you have unexpected host-device transfers. If a non-GEMM elementwise op exceeds 5%, consider fusing it into an adjacent kernel with torch.compile.

17.4.2 WandB and Experiment Tracking

WandB (Weights & Biases) serves a different purpose from PyTorch Profiler: it tracks *experiment-level* metrics over the course of a full training run, not step-level internals. The right configuration emits both ML metrics (loss, perplexity, gradient norm) and infrastructure metrics (step time, MFU, GPU memory) to enable correlation during post-run analysis.

```
import wandb, time, torch

# ── Initialize with infrastructure-aware tags ──────────────────────
wandb.init(
    project="llama3-70b-pretrain",
    config={"model": "llama3-70b", "tp": 4, "pp": 1, "dp": 8,
            "batch_size": 2048, "seq_len": 4096, "dtype": "bf16"},
    tags=["h100", "tp4", "bf16"]
)

# ── Per-step logging (log every 10 steps to avoid Prometheus overhead) ──
for step, batch in enumerate(train_loop):
    t0 = time.perf_counter()
    loss = train_step(batch)
    step_time = time.perf_counter() - t0

    tokens_per_sec = tokens_in_batch / step_time
    model_flops = compute_model_flops(batch)          # from model config
    peak_flops = 989e12 * num_gpus                    # H100 SXM BF16 TFLOPS
    mfu = model_flops / (peak_flops * step_time)

    if step % 10 == 0:
        wandb.log({
            "train/loss":       loss.item(),
            "train/step_time":  step_time,
            "train/tokens_sec": tokens_per_sec,
            "train/mfu":        mfu,
            "train/grad_norm":  grad_norm,
            "hw/gpu_mem_used":  torch.cuda.memory_allocated() / 1e9,
            "hw/gpu_mem_peak":  torch.cuda.max_memory_allocated() / 1e9,
        }, step=step)
```

17.5 System Layer — Prometheus Configuration

Prometheus is a pull-based time-series database. Every 15 seconds, its scrape engine sends HTTP GET requests to each registered exporter endpoint, parses the metrics text format, and stores samples in its local TSDB. Understanding its data model and configuration options is essential for running it at AI cluster scale.

17.5.1 Complete prometheus.yml for AI Clusters

```yaml
# prometheus.yml — Production configuration for 1,000-GPU AI cluster
global:
  scrape_interval:     15s
  evaluation_interval: 15s
  external_labels:
    cluster: 'gpu-cluster-prod'
    region:  'us-east-1'

scrape_configs:
  # GPU metrics via dcgm-exporter on every GPU node
  - job_name: 'dcgm'
    file_sd_configs:                  # file-based service discovery
      - files: ['/etc/prometheus/gpu_nodes.json']
        refresh_interval: 60s
    relabel_configs:
      - source_labels: [__address__]
        target_label: instance
      - target_label: __metrics_path__
        replacement: /metrics

  # OS-level (CPU, RAM, NIC, storage)
  - job_name: 'node'
    file_sd_configs:
      - files: ['/etc/prometheus/all_nodes.json']

  # vLLM inference server metrics
  - job_name: 'vllm'
    static_configs:
      - targets: ['inference-01:8000', 'inference-02:8000']
    metrics_path: '/metrics'

  # SLURM scheduler metrics
  - job_name: 'slurm'
    static_configs:
      - targets: ['slurm-head:9341']

rule_files:
  - "gpu_alerts.yml"
  - "recording_rules.yml"

storage:
  tsdb:
    retention.time: 30d
    retention.size: 500GB

remote_write:
  # For fleets >10K GPUs: write to Thanos for long-term storage
```

```
  - url: "http://thanos-receive:10908/api/v1/receive"
    queue_config:
      max_samples_per_send: 5000
      batch_send_deadline: 5s
```

17.5.2 Production Alert Rules

```yaml
# gpu_alerts.yml — Production-validated alert rules
groups:
  - name: gpu_critical
    rules:
      # DBE = uncorrectable: immediate GPU quarantine
      - alert: GPUDoubleBitECC
        expr: increase(DCGM_FI_DEV_ECC_DBE_VOL_TOTAL[1m]) > 0
        for: 0m           # Fire immediately — no tolerance
        labels:
          severity: critical
        annotations:
          summary: "UNCORRECTABLE ECC error on {{ $labels.gpu_uuid }}"
          runbook: "Quarantine immediately. Job evacuation required."

      # Thermal throttle prevention: alert before threshold, not after
      - alert: GPUHighTemperature
        expr: DCGM_FI_DEV_GPU_TEMP > 82
        for: 2m
        labels:
          severity: warning
        annotations:
          summary: "GPU {{ $labels.gpu_uuid }} at {{ $value }}°C (throttle at 83°C)"

      # MFU regression detection: >10% sustained drop
      - alert: TrainingMFURegression
        expr: |
          (avg_over_time(training_mfu[5m])
           < 0.9 * avg_over_time(training_mfu[30m] offset 1h))
           and avg_over_time(training_mfu[30m] offset 1h) > 0.4
        for: 5m
        labels:
          severity: warning
        annotations:
          summary: "Training MFU dropped >10% on job {{ $labels.job_id }}"

      # KV cache pressure: inference SLA risk
      - alert: KVCachePressure
        expr: vllm:gpu_cache_usage_perc > 0.92
        for: 1m
        labels:
          severity: warning
        annotations:
          summary: "KV cache at {{ $value | humanizePercentage }} — preemption risk"

      # SBE rate: early hardware failure warning
      - alert: GPUSingleBitECCRate
        expr: rate(DCGM_FI_DEV_ECC_SBE_VOL_TOTAL[1h]) > 1
```

```
        for: 30m
        labels:
          severity: warning
        annotations:
          summary: "GPU {{ $labels.Hostname }} showing >1 SBE/hour — schedule replacement"
```

17.5.3 Key PromQL Queries for AI Infrastructure

```
# ── Fleet-wide Tensor Core utilization ──────────────────────────
avg(DCGM_FI_PROF_PIPE_TENSOR_ACTIVE) by (job_id)

# ── GPU straggler detection: any GPU >20pp below its node average ──────
(DCGM_FI_DEV_GPU_UTIL - avg by (Hostname)(DCGM_FI_DEV_GPU_UTIL)) < -20

# ── TTFT P99 for inference SLA tracking ─────────────────────────
histogram_quantile(0.99,
  rate(vllm:time_to_first_token_seconds_bucket[5m]))

# ── Inference decode throughput (tokens/sec fleet-wide) ─────────────
sum(rate(vllm:request_generation_tokens_sum[1m]))

# ── GPUs in thermal throttle state ──────────────────────────────
count(DCGM_FI_DEV_GPU_TEMP > 80) by (cluster)

# ── NVLink utilization fraction (saturated = good in AllReduce) ────────
(DCGM_FI_PROF_NVLINK_TX_BYTES + DCGM_FI_PROF_NVLINK_RX_BYTES) / 900e9
```

17.6 Service Layer — TTFT, TPOT, and SLA Monitoring

For inference systems, hardware utilization is a means to an end. The end is serving user requests within a Service Level Objective (SLO): TTFT P99 under 500ms, TPOT P50 under 30ms, availability 99.9%. The service layer tracks these end-user-visible metrics and connects them to the infrastructure layer so you know whether a TTFT violation is caused by GPU compute saturation, KV cache eviction, or request queuing.

17.6.1 Critical vLLM Metrics

Metric	Type	Healthy Range	When Degraded
`vllm:gpu_cache_usage_perc`	Gauge	<0.85	>0.92: preemption risk; swap begins; TTFT spikes
`vllm:num_requests_running`	Gauge	<max_batch	= max_batch saturated: queuing begins
`vllm:num_requests_waiting`	Gauge	<10	>50: queue building; p99 TTFT will exceed SLO
`vllm:time_to_first_token_seconds`	Histogram	P99 < SLO	P99 spike: compute contention, KV swap, or prefill queue
`vllm:time_per_output_token_seconds`	Histogram	P50 < 30ms	P50 rise: decode batching suboptimal or HBM BW saturation
`vllm:spec_decode_draft_acceptance_rate`	Gauge	>0.60	<0.45: disable speculative decoding; overhead exceeds benefit

> **KV CACHE OCCUPANCY: THE #1 INFERENCE OPERATIONS METRIC**
>
> `vllm:gpu_cache_usage_perc` is the single most actionable inference metric. When it crosses 85%, preemption becomes likely. When it crosses 92%, GPU-to-CPU KV swap begins — adding 400–800ms of HBM→PCIe→CPU latency per evicted sequence. Alert at 85% to give operators time to respond; alert critical at 92% to trigger auto-scaling. (See Chapter 11 for KV cache management architecture.)

17.7 Fleet Layer — Grafana Dashboard Design

Grafana translates Prometheus time-series data into human-readable dashboards. The design of these dashboards is as important as the metrics themselves: a poorly designed dashboard will be ignored; a well-designed one catches problems in 30 seconds. For AI clusters, four dashboards cover 95% of daily operational needs.

17.7.1 The Four Essential AI Cluster Dashboards

Dashboard	Primary Metric	Key Panel Types	Alert Integration
Fleet Health Heatmap	SM utilization per GPU	Heatmap (hostname × time), stat panels	Green >70%, yellow 40-70%, red <40%
Inference SLA	TTFT P50/P99, KV cache %	Time series, gauge, histogram panels	Annotation when SLO fires
Training MFU Tracker	MFU per job	Time series with expected baseline band	Red fill when MFU >10% below baseline
Hardware Health	Temp, ECC, power	Temperature heatmap, ECC error table	Red row = GPU needing replacement

> **PRINCIPAL TIP: RECORDING RULES ARE MANDATORY AT SCALE**
>
> At 1,000 GPUs x 15s scrape x 10 metrics = 667 data points/second. A 24h Grafana query at 15s resolution must process 5.7M points per panel and will time out. Recording rules pre-aggregate to 60s, reducing query load 4x. At 10,000 GPUs with Thanos, recording rules are non-negotiable.

17.8 AMD Toolchain — omniperf, ROCprofiler, AMDSMI

AMD GPU clusters (MI300X, MI355X) require a different toolchain but the *same conceptual layers*. The mapping is direct: DCGM maps to AMDSMI, Nsight Systems maps to ROCprofiler-systems, Nsight Compute maps to omniperf. Framework and fleet layers (PyTorch Profiler, Prometheus, Grafana) are vendor-neutral on both platforms.

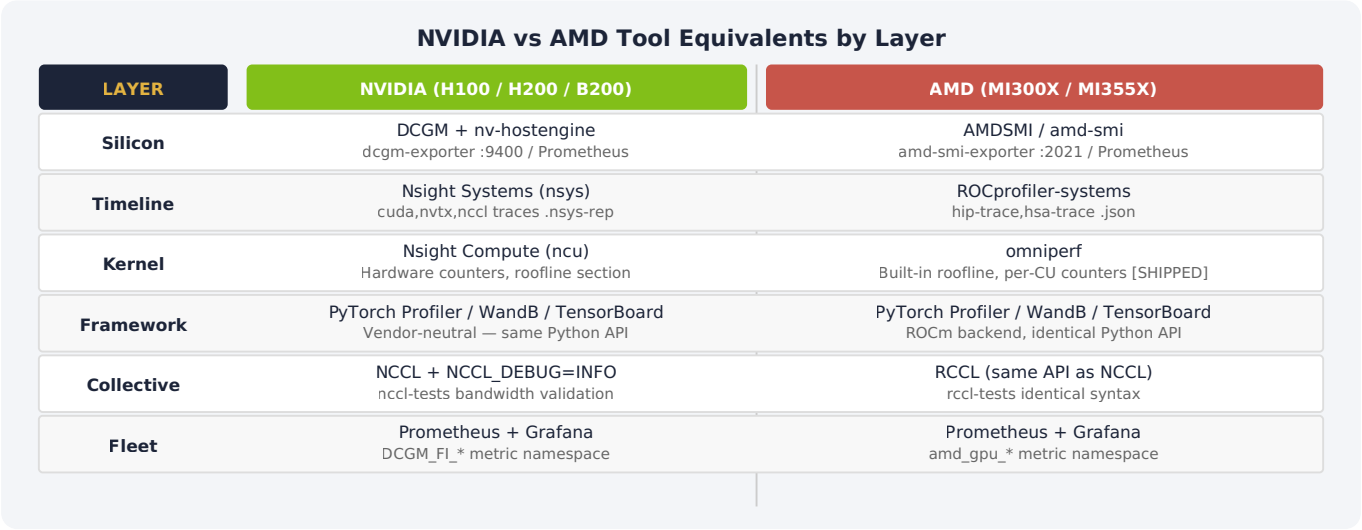


Figure 17.4 — NVIDIA vs AMD observability tool mapping. Framework and fleet layers are vendor-neutral. The silicon, timeline, and kernel layers require platform-specific tooling.

17.8.1 omniperf Roofline on MI300X

```
# Profile with roofline data only (faster)
omniperf profile --name attention_run --roof-only -- python eval_attention.py
omniperf analyze -p attention_run/ --filter-kernel "flash_attention" --roof-only

# MI300X ridge point:
# FP16 peak:  1307 TFLOPS | HBM3 BW: 5.3 TB/s
# Ridge:      1307 / 5300 = 247 FLOP/byte
# Decode at batch=1: ~4-10 FLOP/byte → deeply memory-bound on MI300X too

# AMD-SMI live monitoring
amd-smi monitor --watch 1 --watch-time 60
amd-smi static -g 0 --ras              # RAS error info (ECC equivalent)
rocm-smi --showrasinfo all | grep uncorrect  # Any uncorrectable error = quarantine
```

17.9 Performance Regression CI/CD Pipeline

Every driver update, framework version bump, or model change is a potential performance regression. Without an automated CI pipeline, regressions surface in production after users complain — often days later. With a statistical regression pipeline, a 5% GEMM throughput drop fires a failing CI check before the PR merges.

```
#!/bin/bash  # kernel_regression_check.sh
KERNEL="flash_attention_fwd"
BASELINE_DIR="/benchmarks/baseline"
CURRENT_DIR="/tmp/current_profile"
REGRESSION_THRESHOLD=0.05

# Step 1: Profile current code (10 runs for statistical validity)
ncu --kernel-name ${KERNEL} --launch-count 10 --csv \
    -o ${CURRENT_DIR}/profile python eval_attention.py > metrics.csv

# Step 2: Statistical comparison via t-test (not naive point comparison)
python3 compare_profiles.py \
    --baseline ${BASELINE_DIR}/metrics.csv \
    --current  metrics.csv \
    --threshold ${REGRESSION_THRESHOLD} \
    --confidence 0.95

# Exit 0 = OK, Exit 1 = regression detected (CI fails, PR blocked)
```

STATISTICAL BASELINE, NOT POINT COMPARISON

Kernel metrics vary ±2-4% per run due to clock jitter and thermal state. A naive comparison fires false alerts constantly. The correct approach: collect 10+ baseline runs, compute mean ± 1.5 standard deviations as the confidence interval, and flag only regressions that fall below the lower bound with $p < 0.05$ (two-sample t-test). This eliminates false positives while catching real regressions above 3-4%.

17.10 Profiling Diagnostic Cookbook

The following decision tree gives you a systematic path from symptom to root cause for the most common AI infrastructure performance problems. Each node specifies: the first metric to examine, the command to run, and the decision branch based on the result.

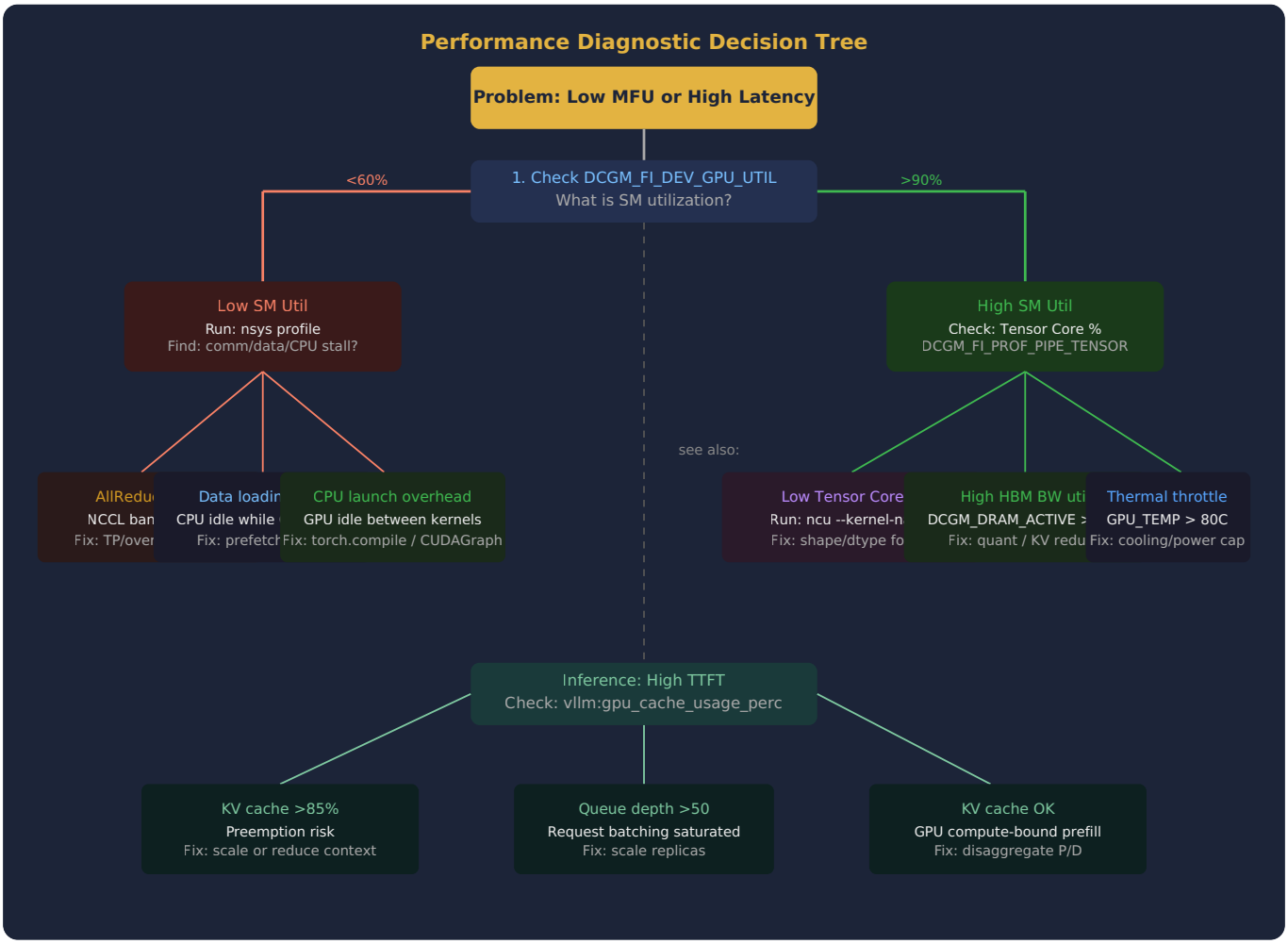


Figure 17.5 — Performance diagnostic decision tree. Start with SM utilization to branch into low-GPU-util (scheduling/ communication) vs. high-GPU-util (kernel efficiency) problems. Inference path branches on KV cache occupancy.

17.10.1 Quick-Reference Diagnostic Commands

Symptom	First Command	Key Metric	Action if Positive
Low MFU (<50%)	`nsys profile --trace=cuda,nvtx,nccl python train.py --steps 10`	% time in NCCL vs GEMM	If NCCL >25%: optimize overlap (Ch14)
Kernel regression after update	`ncu --kernel-name X --set full python eval.py`	SM throughput % change	>5% drop: compare ncu-ui --diff baseline current
TTFT P99 spike	`promql: vllm:gpu_cache_usage_perc avg`	KV cache occupancy	>85%: reduce max_context or scale up
GPU thermal throttle	`dcgmi health -g 0 --check all`	DCGM_FI_DEV_GPU_TEMP	>82°C: check cooling, power cap at 600W
ECC error rate	`dcgmi stats -e`	SBE rate per hour	>1/hr: schedule replacement in 48-72h
NVLink bandwidth low	`dcgmi stats --nvlink 0`	NVLINK_TX/RX bytes/s	<400 GB/s during AllReduce: check link state
Training loss anomaly	`wandb run page > loss curve + grad norm`	Gradient norm spike	Spike + NVLink XID 74: silent corruption, restart job

Worked Example: 10,000-GPU Observability Stack Design

PRINCIPAL-LEVEL WORKED EXAMPLE — COMPLETE OBSERVABILITY ARCHITECTURE

Problem: Design the complete observability stack for a 10,000-GPU training and inference cluster running mixed workloads (Llama3-70B pretraining + production serving). Budget: metric retention 1 year. Alert response time: <5 minutes. Scale: ~10K GPU × 25 metrics = 250,000 metric series.

Step 1 — Estimate metric cardinality and Prometheus requirements:

```
GPUs:                  10,000
Metrics per GPU:       25 (DCGM) + 5 (node-exporter per-NIC) = 30
Total active series:   10,000 x 30 = 300,000 series
Scrape interval:       15 seconds
Ingestion rate:        300,000 / 15 = 20,000 samples/second

Single Prometheus TSDB limit:        ~10M series (fine for 300K)
But 20K samples/s x 86400s x 365d = 630 billion samples/year
At ~2 bytes/sample:                  1.26 TB raw per year

Decision: Single Prometheus for hot data (30d), Thanos remote write to
          S3 for cold storage (1yr at ~50GB/month compressed with Mimir).
```

Step 2 — Alert latency calculation:

```
Problem: MFU drops from 62% to 47% at T=0 (driver update).
How quickly does our stack detect it?

T=0:    Driver update deployed
T=15s:  DCGM polls GPU; DCGM_FI_PROF_PIPE_TENSOR_ACTIVE updates
T=30s:  Prometheus scrapes dcgm-exporter (next 15s interval)
T=45s:  Alert rule evaluation (evaluation_interval: 15s)
T+for:  Alert rule has "for: 3m" to avoid flapping on transient dips
T=225s: Alert fires to Alertmanager (3 minutes 45 seconds from incident)

Verdict: Under 5-minute SLA for Tensor Core utilization drop alert.
But "for: 3m" means we accept up to 3.75 min detection lag to eliminate
false positives from brief scheduling hiccups. Tunable per criticality.
```

Step 3 — On-demand profiling trigger design:

```
Always-on cost:  DCGM polling + Prometheus = 0.5% GPU util overhead
Nsight Systems:  +2-5% CPU overhead, 0% GPU (safe for production spot-checks)
Nsight Compute:  +300-1000% kernel overhead — NEVER always-on

Automated trigger rule:
  IF training_mfu drops > 10% for > 5 minutes:
    → Automatically capture 60s Nsight Systems trace on rank 0
    → Upload trace to centralized storage bucket
    → File Jira ticket with trace link and MFU delta
    → Page on-call (Severity: P2)

This gives engineers a pre-recorded timeline trace from the moment
the regression occurred — not hours later after they log in.
```

Step 4 — Cost comparison with and without observability:

```
Without observability stack (baseline):
  Regression detection time:         3-7 days (user reports slow training)
  Wasted compute (10K GPU x $3/hr):  3 days x 24h x 10,000 x $3 = $2.16M

With proper observability stack:
  Regression detection time:         4 minutes
  Wasted compute (4 min):            10,000 x 0.067h x $3 = $2,000

Observability infrastructure cost (Prometheus + Thanos + Grafana):
  EC2 + storage ~$5K/month = $60K/year

ROI: Prevents even a single week-long regression = $5M+ saved.
Observability is not a cost center. It is profit protection.
```

Key Takeaways

CORE CONCEPTS

- The AI observability stack has seven layers: silicon telemetry, kernel profiling, framework profiling, system metrics, service-level SLAs, fleet dashboards, and regression CI/CD — each at a different sampling rate and overhead tradeoff.
- DCGM provides silicon-level GPU health counters at 1-second intervals with <0.5% overhead. It is the only tool that can run always-on at fleet scale. `DCGM_FI_PROF_PIPE_TENSOR_ACTIVE` is the single most revealing metric for LLM training efficiency.
- Nsight Systems (timeline) and Nsight Compute (kernel counters) serve fundamentally different purposes. Confusing them is the most common profiling mistake: nsys for "where is time going," ncu for "why is this kernel slow."
- The AMD toolchain (AMDSMI, ROCprofiler-systems, omniperf) maps directly to the NVIDIA equivalent layer-by-layer. Framework and fleet layers are vendor-neutral.

COST & PRODUCTIVITY IMPACT

- A single undetected week-long MFU regression on a 10,000-GPU cluster costs $5M+ at $3/GPU-hr. A $60K/year Prometheus + Grafana stack that reduces detection time from 7 days to 4 minutes delivers >80× ROI in the first prevented incident.
- Automated regression CI/CD that blocks driver/framework updates with >5% kernel performance degradation eliminates the class of regressions that historically take 3–7 days to diagnose — translating directly to recovered cluster efficiency and reduced ML team debugging overhead.
- `vllm:gpu_cache_usage_perc` alerting at 85% gives operators 5–10 minutes to respond before KV swap begins adding 400–800ms to inference latency — preventing SLO violations for zero additional infrastructure cost.

PRINCIPAL-LEVEL INSIGHT

- The diagnostic question to ask first is never "which tool should I use?" but "what question am I answering?" — SM utilization tells you *if* compute is being used; Tensor Core utilization tells you *how efficiently*; Nsight Systems tells you *where time is going*; Nsight Compute tells you *why a kernel is slow*. These are four distinct questions requiring four distinct tools.
- Metric cardinality management is the hidden scaling problem in observability. At 10K GPUs × 30 metrics = 300K series, Prometheus handles it fine. At 100K GPUs, naive scaling breaks: Thanos federated architecture, recording rules pre-aggregating 60% of queries, and aggressive label dropping (strip per-pid labels) are mandatory, not optional.
- XID 74 (NVLink error) during tensor-parallel training can produce silent gradient corruption — training converges to a subtly wrong model. Any XID 74 on a TP job requires immediate restart and a 30-minute NVLink bandwidth validation test, regardless of whether loss appears normal.

Review Questions

FOUNDATIONAL Build the mental model — no math required

1. Explain the difference between Nsight Systems and Nsight Compute. A colleague says "I'll use Nsight Compute to find out why my training step is 30% slower than expected." Why is this likely the wrong choice, and which tool should they use first?
2. A GPU shows DCGM_FI_DEV_GPU_UTIL = 95% but training MFU is only 38%. Is the GPU "working"? What does this combination of metrics tell you about where the inefficiency lies?
3. What is an XID 48 error, and what immediate operational action does it require? How does it differ from XID 63, and why does the difference in severity matter for cluster operations?

INTERMEDIATE Apply and calculate — production context

1. You are running 1,000 GPUs with DCGM scraping 30 metrics per GPU every 15 seconds. A team member asks you to store 2 years of raw metric history locally in Prometheus. Calculate the storage requirement. What architecture change would you recommend, and why?
2. Your Prometheus alert for `TrainingMFURegression` has `for: 0m` (fires immediately). After deploying, you are paged 12 times in one day by transient 2-second MFU dips during AllReduce. Redesign the alert rule to eliminate false positives while keeping detection latency under 5 minutes for real regressions.

3. A vLLM inference server shows `vllm:gpu_cache_usage_perc = 0.97` and TTFT P99 has spiked from 280ms to 1,400ms. Walk through your diagnostic sequence: what two metrics do you check next, what do they tell you, and what are the two possible remediation actions?

PRINCIPAL LEVEL Design, decide, and defend — architecture reviews

1. Design a complete observability architecture for a 10,000-GPU mixed training + inference cluster. Your design must specify: which metrics are collected at which layers, the data flow from GPU hardware to on-call engineer, the retention strategy, and the alert escalation policy. Justify every decision with the metric cardinality and latency math. Budget constraint: maximum $100K/year for observability infrastructure.

2. Your team proposes always-on Nsight Compute profiling for all production training jobs to enable continuous kernel-level regression detection. Evaluate this proposal quantitatively. If you reject it, propose an alternative architecture that achieves the same regression detection goal without the overhead. What is the tradeoff your alternative accepts?

3. After a large-scale NCCL version upgrade, you notice training throughput dropped 8% across 500 jobs, but the change was not caught by your existing CI/CD pipeline. Design a comprehensive performance regression CI/CD system that would have caught this regression before production rollout. Include: what workloads to benchmark, what statistical test to use, what threshold to set, how to handle hardware variance across CI test machines, and how to gate the rollout process.