PART VI — PRODUCTION OPERATIONS

Resilience, Fault Tolerance & SRE at AI Scale

CHAPTER 19

Production Resilience, Fault Tolerance & SRE for AI Infrastructure

GPU failure rates · Checkpoint strategies · NCCL recovery · Job preemption · Cluster-level MFU protection · SRE playbooks for distributed training and inference at scale

---

> *"A 10,000-GPU training job that fails at hour 47 of a 72-hour run and cannot recover loses 47 GPU-hours multiplied by 10,000. That is 470,000 GPU-hours of compute — more than some organizations use in a year. Fault tolerance is not a reliability concern. It is an economics concern."*
>
> — Production AI Infrastructure Principle

19.1 Why Production Resilience Is the Hidden Bottleneck

The infographic framing has shifted: we moved from "GPU shortage problem" to "system bottleneck problem." The next hidden bottleneck is **operational resilience** — the ability of AI infrastructure to maintain high cluster utilization despite the inevitable reality that hardware fails, networks partition, and software deadlocks.

At the scale of 10,000–100,000 GPUs, failure is not an exception. It is a *constant background condition*. A single GPU failing every 200,000 GPU-hours sounds rare — until you realize a 10,000-GPU cluster experiences that failure statistically every 20 hours. Without resilience engineering, each failure terminates the job. With it, recovery happens in minutes.

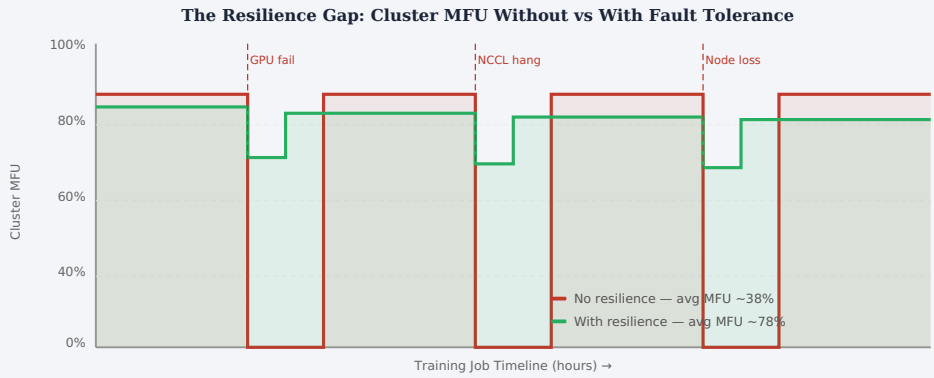


Figure 19.1 — Cluster MFU with vs without fault tolerance. GPU failures without recovery collapse utilization to 0% until manual restart. With checkpointing + auto-recovery, the dip is 5–15 minutes instead of hours.

> **THE ECONOMICS FORMULA**
>
> At $2–3/GPU-hour, a 10,000-GPU cluster costs $20,000–30,000 per hour. A 3-hour unrecoverable failure = $60,000–90,000 lost. A 15-minute checkpoint recovery = $5,000–7,500 lost. Resilience engineering ROI pays back in the first incident.

19.1.1 The New Bottleneck Hierarchy

The bottleneck hierarchy has shifted. In 2020–2024, the constraint was GPU compute availability. Today, for organizations that have secured GPU capacity, the constraint is *effective utilization* — keeping those GPUs doing productive work rather than sitting idle due to failures, hangs, and recovery overhead.

Bottleneck Era	Primary Constraint	Principal Engineer's Job
2020–2024	GPU availability (shortage)	Get access to compute
2024–2026	Memory BW, networking, power	Maximize per-GPU utilization
2026+	**Operational resilience & sustained MFU**	**Keep large jobs running, minimize MTTR**

19.2 Failure Taxonomy: What Breaks and How Often

Understanding failure modes is prerequisite to designing recovery strategies. Failures in large AI clusters fall into four categories with very different detection and recovery characteristics.

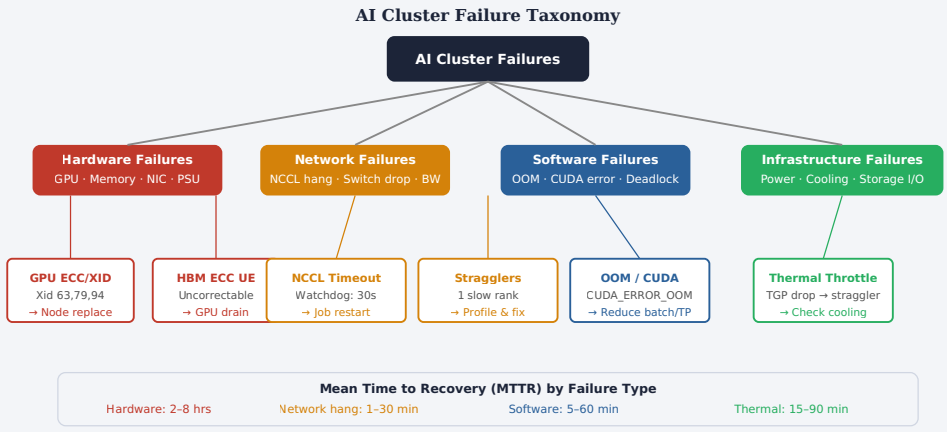


Figure 19.2 — Failure taxonomy for large AI clusters with recovery strategies and MTTR ranges.

19.2.1 GPU Failure Rates at Scale

```
GPU FAILURE RATE MODEL (Production Observations):

Single GPU MTBF (Mean Time Between Failures):
  Consumer GPU (RTX 4090): ~50,000 hours
  Data center GPU (H100 SXM): ~100,000–200,000 hours
  [ENV-SPECIFIC] Varies significantly with: power draw, cooling, workload

EXPECTED FAILURES IN A CLUSTER:
  Cluster size:     1,000 GPUs    10,000 GPUs    100,000 GPUs
  MTBF per GPU:     150,000 hrs   150,000 hrs    150,000 hrs
  Expected failures:
    Per day:        0.16          1.6            16
    Per week:       1.1           11             110
    Per month:      4.4           44             440

  → A 10,000-GPU cluster expects a GPU failure roughly every 15 hours.
  → Without recovery: every training job > 15 hours will likely fail once.

TOP NVIDIA GPU FAILURE SIGNALS (DCGM + XID Errors):
  Xid 63  → Row remapping failure (HBM ECC uncorrectable) → replace GPU
  Xid 79  → GPU has fallen off the bus (PCIe/NVLink failure) → reboot
  Xid 94  → Contained ECC error (correctable, monitor closely)
  Xid 119 → NVLink error → check NVLink topology, may need isolation

  AMD equivalent: RAS (Reliability/Availability/Serviceability) errors
  ROCm-SMI: rocm-smi --showrasinfo all → ue_count (uncorrectable errors)
```

19.2.2 NCCL Hangs — The Silent Killer

GPU hardware failures are visible and detectable. NCCL hangs are far more dangerous because they are *silent* — all ranks appear to be running, but the collective operation never completes. Without a watchdog, a NCCL hang can hold a 10,000-GPU job for hours while billing continues and no progress is made.

```
NCCL HANG ROOT CAUSES (in order of frequency):

1. NETWORK PACKET LOSS (most common)
   One corrupted or dropped packet in an AllReduce stalls all ranks.
   Detection: NCCL_TIMEOUT (default: 30s in PyTorch)
   Root cause check: ibstat | grep -i error; ethtool -S eth0 | grep error

2. STRAGGLER RANK (second most common)
   One GPU is 10× slower (thermal throttle, PCIe bandwidth degradation).
   All other ranks wait at the collective barrier indefinitely.
   Detection: per-rank timing logs; step time variance > 20% → investigate

3. DEADLOCK IN PIPELINE PARALLELISM
   PP rank 1 waiting for rank 2 while rank 2 waits for rank 1.
   Requires careful ordering of send/recv operations.
   Detection: py-spy dump → shows all threads blocked in ncclAllReduce

4. NVLink FABRIC DEGRADATION
   Partial NVLink failure reduces bandwidth but does not kill connection.
   Appears as: NCCL timeout on large tensors, success on small.
   Detection: nvidia-smi nvlink --status → check error counters

PREVENTION:
  NCCL_DEBUG=WARN              → enable NCCL warning logging
  NCCL_TIMEOUT=1800            → set 30-minute hard timeout
  TORCH_NCCL_HEARTBEAT_TIMEOUT → PyTorch watchdog (default: 120s)
  NCCL_SOCKET_IFNAME=eth0      → force specific NIC (avoid ambiguity)
```

19.3 Checkpointing Strategies

Checkpointing is the primary mechanism for limiting data loss when failures occur. The engineering challenge is that naively saving all model state is expensive — at scale, a checkpoint of a 70B parameter model in BF16 takes 140 GB, and writing that to distributed storage can itself take 5–10 minutes, during which training pauses.

Checkpoint Strategy Comparison

Strategy	Checkpoint Size	Save Time	Recovery Loss	Best For
Synchronous Full Pause → Save → Resume	140 GB (70B BF16)	**5–10 min** Training paused	**Minutes**	Small jobs <1,000 GPU
Async / Background Copy → Save in parallel	140 GB (2× memory peak	**~0 (async)** Training continues	**Minutes**	Large training runs
Incremental / Delta Changed tensors only	**5–20 GB (typical)**	**<30 sec** Minimal pause	**Minutes**	RLHF/fine-tuning runs
Distributed (PyTorch DCP) Per-rank sharded save	140 GB sharded	**1–2 min** Parallel I/O	**Minutes**	>1,000 GPU training

Principal Architect Recommendation

For jobs >1,000 GPUs: Use PyTorch DCP (Distributed Checkpoint) with async saves every 15–30 minutes.
Target checkpoint overhead: <2% of total training time. Keep 3 rolling checkpoints to protect against checkpoint corruption.

Figure 19.3 — Checkpoint strategy comparison. PyTorch DCP is the recommended approach for large-scale distributed training.

19.3.1 PyTorch Distributed Checkpoint (DCP) — Production Configuration

```
PYTORCH DCP — PRODUCTION CHECKPOINT SETUP:

import torch.distributed.checkpoint as dcp
from torch.distributed.checkpoint.state_dict import get_state_dict, set_state_dict

# Save checkpoint — each rank saves its own shard
def save_checkpoint(model, optimizer, step, ckpt_dir):
    state_dict = {"model": model, "optimizer": optimizer}
    model_sd, optim_sd = get_state_dict(model, optimizer)

    dcp.save(
        {"model": model_sd, "optimizer": optim_sd, "step": step},
        checkpoint_id=f"{ckpt_dir}/step_{step}",
        storage_writer=dcp.FileSystemWriter(f"{ckpt_dir}/step_{step}")
    )

# Key configuration parameters:
#   planner=dcp.DefaultSavePlanner()  → shard by rank automatically
#   storage_writer=FileSystemWriter   → local NFS/Lustre/S3

CHECKPOINT FREQUENCY FORMULA:
  Given: job runtime T_total hours
         checkpoint time T_ckpt minutes
         GPU-hr cost C $/hr
         failure probability per hour P_fail

  Expected wasted compute without checkpointing:
    E[wasted] = P_fail × T_total × T_total/2 × N_GPUs × C

  Optimal checkpoint interval (minimizes expected wasted compute):
    T_interval = sqrt(2 × T_ckpt / P_fail)

  Example: T_ckpt=2min, P_fail=0.07/hr (10K GPU cluster)
    T_interval = sqrt(2 × 2/60 / 0.07) = sqrt(0.952) ≈ 0.976 hr ≈ 58 min

  → Checkpoint every ~60 minutes for a 10K GPU cluster. [ENV-SPECIFIC]
```

19.3.2 Checkpointing Optimizer State — The Hidden Cost

Model weights are only part of the checkpoint. For full training recovery, you also need optimizer state (Adam momentum and variance tensors), which is typically *3× the model size* for a 32-bit Adam optimizer.

```
CHECKPOINT SIZE BREAKDOWN (LLaMA-3 70B, BF16 training):

Component              Size        Notes
──────────────────────────────────────────────────────
Model weights (BF16)   140 GB      2 bytes × 70B params
Adam m (FP32)          280 GB      4 bytes × 70B params
Adam v (FP32)          280 GB      4 bytes × 70B params
Gradient buffer (BF16) 140 GB      same shape as weights
RNG state              ~1 MB       per-rank RNG for dropout
Scheduler state        ~1 KB       negligible

TOTAL FULL CHECKPOINT:  ~840 GB    (6× model size in BF16)
TOTAL WEIGHTS ONLY:     ~140 GB    (sufficient for inference)

PRACTICAL STRATEGIES:
  1. BF16 Adam states → reduces 560 GB → 280 GB (half precision)
     Implemented via: torch.optim.AdamW(..., foreach=True)
     Loss: minor numerical precision (acceptable for pre-training)

  2. CPU offload checkpointing → optimizer state lives on CPU
     Zero-3 approach: each rank saves its shard only (~840/N_ranks GB each)
     At 1,000 ranks: each rank saves 0.84 GB → fast parallel write

  3. Storage target: NFS with >10 GB/s aggregate throughput minimum
     At 1,000 ranks × 0.84 GB × parallel: need ~840 GB write bandwidth
```

19.4 NCCL Failure Detection and Recovery

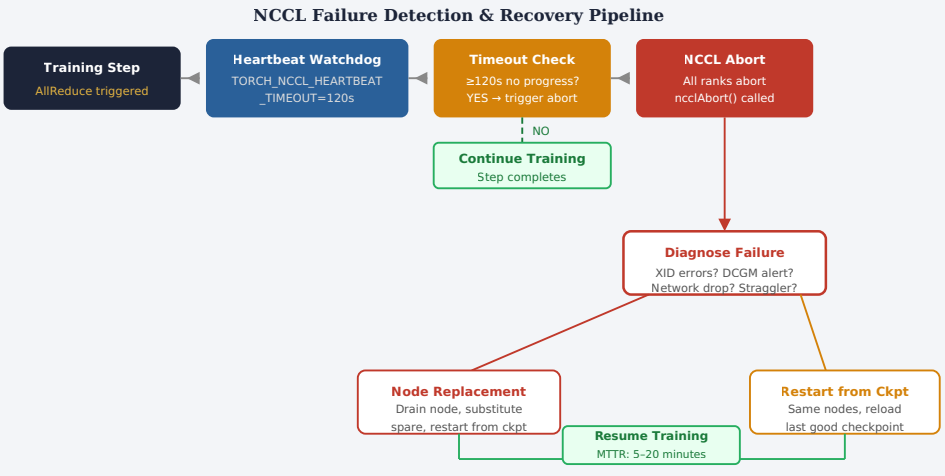


Figure 19.4 — NCCL failure detection and recovery pipeline. The watchdog timeout triggers abort, diagnosis routes to node replacement or checkpoint restart.

```
PYTORCH NCCL WATCHDOG CONFIGURATION (production):

# Environment variables for NCCL resilience
export NCCL_DEBUG=WARN
export NCCL_TIMEOUT=1800                    # 30 min hard timeout
export TORCH_NCCL_HEARTBEAT_TIMEOUT_SEC=120 # 2 min heartbeat
export TORCH_NCCL_ASYNC_ERROR_HANDLING=1    # async abort on error
export NCCL_SOCKET_NTHREADS=4              # network threading
export NCCL_NSOCKS_PERTHREAD=4

# Elastic training with torchrun (auto-restart):
torchrun \
  --nnodes=1:10 \          # min:max nodes (elastic!)
  --nproc-per-node=8 \
  --max-restarts=3 \       # auto-restart up to 3 times
  --rdzv-backend=etcd \    # distributed rendezvous
  --rdzv-endpoint=etcd:2379 \
  train.py

# Elastic training code pattern:
from torch.distributed.elastic.multiprocessing.errors import record

@record  # captures stack trace on crash
def main():
    dist.init_process_group("nccl")
    # ... training loop
    if step % 1000 == 0:
        save_checkpoint(model, optimizer, step)  # checkpoint every 1000 steps
```

19.5 GPU Health Monitoring with DCGM

NVIDIA's Data Center GPU Manager (DCGM) is the primary tool for fleet-level GPU health monitoring. It runs as a daemon on every node and exposes GPU metrics at 1-second resolution for integration with Prometheus/Grafana.

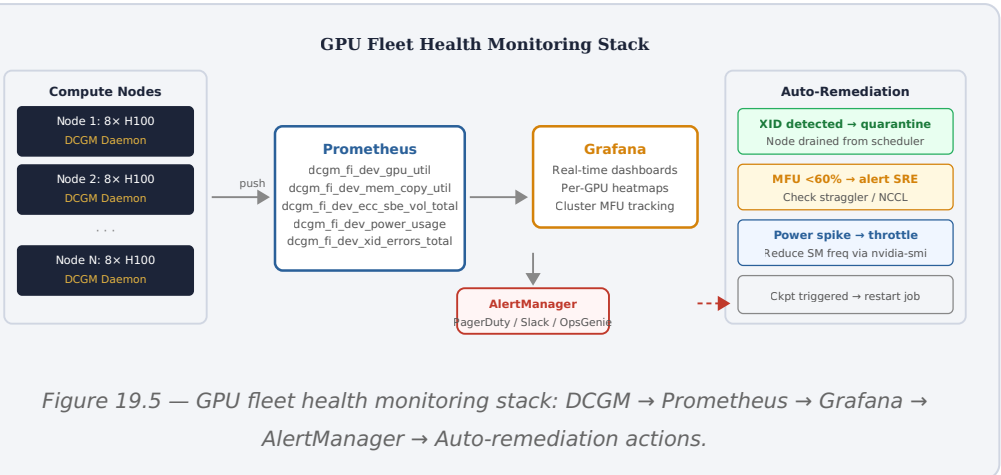


Figure 19.5 — GPU fleet health monitoring stack: DCGM → Prometheus → Grafana → AlertManager → Auto-remediation actions.

```
KEY DCGM METRICS FOR AI TRAINING CLUSTERS:

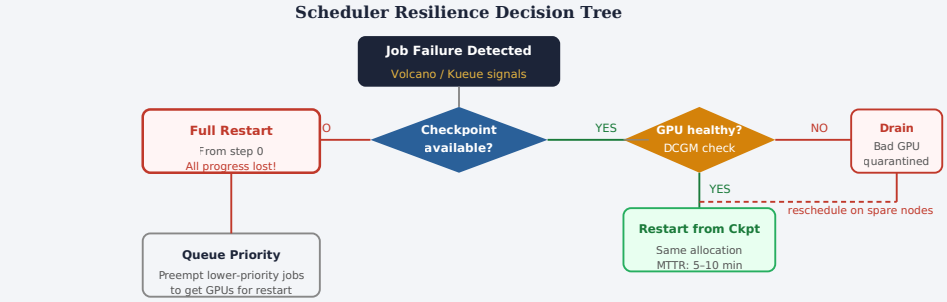
Metric                              Normal Range    Alert Threshold
──────────────────────────────────────────────────────────────────
dcgm_fi_dev_gpu_util                85-100%         <70% sustained
dcgm_fi_dev_mem_copy_util           10-60%          >95% sustained
dcgm_fi_dev_power_usage             600-700W(H100)  >700W or <200W
dcgm_fi_dev_gpu_temp                60-80°C         >85°C → throttle
dcgm_fi_dev_ecc_sbe_vol_total       0 (ideal)       >1000/hr → drain
dcgm_fi_dev_ecc_dbe_vol_total       0               >0 → IMMEDIATE drain
dcgm_fi_dev_xid_errors_total        0               >0 → investigate
dcgm_fi_dev_nvlink_bandwidth_total  ~300 GB/s/link  <250 GB/s → check

PROMETHEUS ALERTING RULES (production):
  # GPU temperature alert
  - alert: GPUTemperatureCritical
    expr: dcgm_fi_dev_gpu_temp > 85
    for: 2m
    labels: {severity: critical}
    annotations:
      summary: "GPU {{ $labels.gpu }} on {{ $labels.instance }} overheating"

  # ECC double-bit error → immediate action
  - alert: GPUUncorrectableECC
    expr: increase(dcgm_fi_dev_ecc_dbe_vol_total[5m]) > 0
    labels: {severity: critical}
    annotations:
      summary: "Uncorrectable ECC error - GPU {{ $labels.gpu }} must be drained NOW"
```

19.6 Job Scheduler Resilience: Kubernetes + Volcano

At cluster scale, job scheduling must be resilience-aware. A job scheduler that simply queues work without understanding GPU health, checkpoint availability, or partial failure modes will amplify outages rather than contain them.



Principal Rule: Never run a multi-day job without checkpoints. A restart from step 0 on a 1,000 GPU job costs tens of thousands of dollars.

Figure 19.6 — Scheduler resilience decision tree. Checkpoint availability is the primary branch point. No checkpoint = catastrophic restart.

19.6.1 Elastic Training — Surviving Node Loss Without Full Restart

```
ELASTIC TRAINING WITH PYTORCH TORCHRUN:

Concept: training job can shrink/grow from min_nodes to max_nodes
         If a node fails, job continues at smaller scale
         New node joins → job scales back up

Configuration:
  --nnodes=8:16     # run with 8 to 16 nodes, adapt on failure
  --nproc-per-node=8
  --max-restarts=5  # per agent

Code requirements for elasticity:
  1. Model must handle variable world_size between restarts
  2. DataLoader must handle new number of replicas gracefully
  3. Checkpoint must save/restore with arbitrary topology

Example elastic training loop:
  state = {"epoch": 0, "global_step": 0}

  while state["global_step"] < MAX_STEPS:
      try:
          train_one_step(model, state)
          if state["global_step"] % CKPT_FREQ == 0:
              checkpoint(state)   # always checkpoint before barriers
      except elastic.RendezvousClosedException:
          break   # membership changed, agent will restart
      except Exception as e:
          record_failure(e)
          break   # non-recoverable, escalate

TRADEOFFS:
  Pro: job survives single-node failures without full restart
  Con: requires careful gradient accumulation to maintain effective batch size
  Con: learning rate schedule must account for variable world_size
  Con: pipeline parallelism is NOT elastic (requires fixed topology)
```

19.7 SRE Playbooks for AI Infrastructure

Site Reliability Engineering (SRE) practices developed for web services translate directly to AI infrastructure, with AI-specific adaptations for GPU failure modes and distributed training semantics.

19.7.1 The Three SLOs That Matter for Training

SLO	Definition	Target	Alert Threshold
Cluster MFU	Actual FLOPS / Peak FLOPS across all GPUs	**>60%**	**<50%**
Job Completion Rate	% of scheduled jobs that complete without manual intervention	**>95%**	**<85%**
Recovery MTTR	Mean time from failure detection to training resumption	**<15 min**	**>60 min**

19.7.2 Runbooks — Common Failure Scenarios

```
RUNBOOK 1: NCCL TIMEOUT (most frequent scenario)

Symptoms: Training step time >5× normal; NCCL timeout in logs
          TORCH_NCCL_HEARTBEAT timeout fired; all ranks stuck

Step 1 — Identify straggler rank:
  # Check per-rank step times via metrics
  kubectl logs  | grep "step_time" | awk '{print $3}' | sort -n
  # Rank with 2x+ step time is the straggler

Step 2 — Check straggler node:
  ssh straggler-node
  nvidia-smi  # GPU utilization low? Temperature high?
  ibstat      # InfiniBand port state = ACTIVE?

Step 3 — If hardware healthy: restart job from last checkpoint
  kubectl delete pod
  # Job controller auto-restarts from checkpoint

Step 4 — If hardware sick: quarantine and reschedule
  kubectl taint node  nvidia.com/gpu=quarantine:NoSchedule
  kubectl delete pod
  # Job restarts on healthy nodes

Expected MTTR: 5–15 minutes

_____________________________________________________________


RUNBOOK 2: GPU XID 79 (GPU FELL OFF BUS)

Symptoms: DCGM alert XID 79; nvidia-smi shows N/A for affected GPU
          Training hangs immediately; NCCL error on the rank using that GPU

Step 1 — Identify affected node/GPU:
  dcgmi diag -g  -r 1  # run diagnostics
  nvidia-smi --query-gpu=index,pci.bus_id,ecc.errors.uncorrected.volatile.total \
             --format=csv

Step 2 — Immediate actions:
  # Quarantine node
  kubectl taint node  node.kubernetes.io/unreachable:NoSchedule

  # Check if GPU-reset recovers the device (sometimes works):
  nvidia-smi --gpu-reset -i

  # If not recovered → node replacement required
  # File hardware ticket with datacenter team

Step 3 — Resume training:
  # Job scheduler detects node drain, finds replacement, restarts from ckpt
  # Monitor: dcgm_fi_dev_xid_errors_total{instance="new_node"} == 0
```

```
Expected MTTR: 15–60 minutes (depends on spare node availability)
```

19.8 Quantifying the MFU Impact of Resilience Engineering

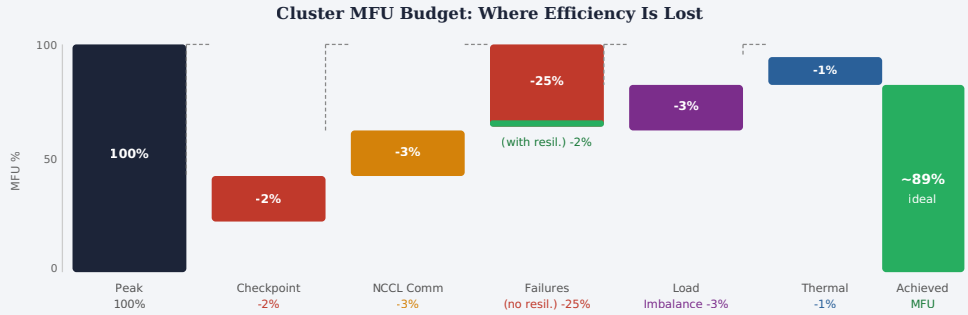


Figure 19.7 — MFU budget waterfall. Without resilience, failures alone can cost 25% MFU. With proper resilience engineering, total overhead is reduced to ~11%, yielding ~89% achievable MFU on ideal hardware.

19.9 Resilience Design Patterns for the Heterogeneous Era

As the compute stack shifts from homogeneous GPU clusters to heterogeneous architectures (GPU + CPU + ASIC + memory-pooled infrastructure), new resilience design patterns become necessary.

19.9.1 Five Resilience Invariants That Survive Every Hardware Generation

> **INVARIANT 1: CHECKPOINT OR DIE**
>
> No training job longer than 2 hours should run without checkpointing. This is not optional. The cost of a single unrecovered failure exceeds the engineering cost of adding checkpointing.

INVARIANT 2: FAILURE DETECTION MUST BE FASTER THAN FAILURE IMPACT

NCCL watchdog timeout should be shorter than your checkpoint interval. If your checkpoint is every 30 min, your watchdog should fire in <5 min, so the worst-case loss is 30 min of compute, not hours.

INVARIANT 3: SEPARATE DATA PATH FROM COMPUTE PATH

Storage for checkpoints must not share the same network fabric as training communication. A networking failure that kills NCCL should not also prevent checkpoint writes to the recovery store.

INVARIANT 4: DESIGN FOR N-1 AVAILABILITY

Any critical infrastructure component — scheduler, rendezvous server, checkpoint storage — must have redundancy. A single-point-of-failure in the control plane can take down thousands of GPUs simultaneously.

INVARIANT 5: MTTR IS A METRIC, NOT AN ANECDOTE

Track P50, P95, P99 MTTR by failure type. If your P95 MTTR for GPU failures is 3 hours, that is a SLO violation regardless of how smoothly the median case goes. Instrument recovery, not just failures.

19.9.2 Production Checklist — Before Launching a Large Training Run

#	Requirement	Verification Command	Status
1	Checkpoint tested (write + restore verified)	`python test_checkpoint.py --verify-restore`	☐
2	DCGM running on all nodes	`dcgmi discovery -l`	☐
3	NCCL watchdog configured	`echo $TORCH_NCCL_HEARTBEAT_TIMEOUT_SEC`	☐
4	GPU health check passed (DCGM diag level 2)	`dcgmi diag -g all -r 2`	☐
5	NVLink / InfiniBand all links active	`nvidia-smi nvlink --status; ibstat`	☐
6	Storage I/O benchmarked for checkpoint writes	`fio --rw=write --bs=4m --size=140G ...`	☐
7	Alert routing configured (PagerDuty/Slack)	Test alert → confirm receipt	☐
8	Elastic training enabled (if job >1 week)	`--nnodes=min:max` in torchrun args	☐
9	Thermal limits verified (ambient temp, cooling)	`dcgmi dmon -e 150,155` (temp, power)	☐
10	Runbooks accessible to on-call engineer	Link in incident management system	☐

Chapter 19 Review Questions

0. A 4,096-GPU training run has GPU MTBF of 150,000 hours. Calculate the expected number of GPU failures per 72-hour run. If each failure causes a

45-minute recovery (with checkpointing every 30 minutes), what is the expected total recovery overhead as a percentage of wall-clock time?

0. Your 2,048-GPU job has been running for 18 hours when a NCCL timeout fires. DCGM shows XID 94 (correctable ECC error) on one GPU. What is your response sequence? Should you replace the node immediately or attempt restart on the same allocation? What factors determine this decision?

0. Compare the checkpoint overhead for a 70B parameter model using: (a) synchronous full checkpoint every 30 min to NFS at 5 GB/s, (b) PyTorch DCP with 256 ranks each writing 3.3 GB shards in parallel to the same NFS at 5 GB/s aggregate. Calculate the training pause time for each. Which strategy does less harm to MFU?

0. Design a resilience architecture for a 10,000-GPU training cluster with a target MTTR of <10 minutes for NCCL failures and <45 minutes for GPU hardware failures. What components are required, and what are the key failure modes of the resilience system itself (i.e., what are the control plane single points of failure)?

0. Your cluster MFU drops from 72% to 55% over a week. DCGM shows no GPU errors, no NCCL timeouts, and CPU utilization is normal. What failure mode do you suspect, and what is your diagnostic sequence? (Hint: consider thermal, load imbalance, and network straggler hypotheses.)
