CHAPTER 21

Real-World Case Studies

Four complete production investigations — inference latency, training MFU, KV cache OOM, and cost optimization — with full profiling workflows and quantitative results

Part VII · Operations & Production · ~42 pages

> *"The gap between knowing the theory and fixing the production system is where senior engineers become principal engineers. Every concept in this book — roofline model, memory hierarchy, NCCL topology, KV paging — earns its value at 2 AM when something is wrong and 1,000 GPUs are idle or a latency SLA is on fire."*
>
> — Production AI Infrastructure Principle

Chapter Overview

The previous twenty chapters built a complete mental model: roofline arithmetic, HBM bandwidth math, transformer compute analysis, KV cache sizing, distributed training parallelism, and fleet-scale observability. This chapter applies all of it. Four real production systems failed in ways that required exactly the kind of first-principles reasoning this book develops. Each case study is presented as it would unfold in practice: initial symptoms, profiling workflow, root cause identification, optimization selection, validation, and final quantified result.

These are not cherry-picked success stories. Each system had multiple wrong hypotheses before the right answer emerged. The debugging process — including dead ends — is documented because that is how principal-level engineers actually work. The ability to discard a plausible hypothesis quickly when the data contradicts it is as important as generating the right hypothesis in the first place.

Cross-references throughout point to the chapter where each technique was developed. This chapter is the capstone, not a standalone; the goal is to show the whole stack operating simultaneously.

LEARNING OBJECTIVES BY LEVEL

LEVEL 1 — FOUNDATIONAL

Understand how the profiling tools from Ch 17 connect to real symptoms. Develop the habit of forming a quantitative hypothesis before touching a configuration file.

LEVEL 2 — INTERMEDIATE

Execute the full profiling workflow: nsys → DCGM → per-optimization quantification. Calculate expected improvement from first principles before implementing, and validate against measurement.

LEVEL 3 — PRINCIPAL

Design the fix architecture, justify each decision with cost-performance math, and know when to stop optimizing. Build a reusable diagnostic framework that a team can follow without the principal being present.

CASE STUDY 1 OF 4

LLM Inference Optimization — 64 H100 Cluster

Model: LLaMA-3 70B · Hardware: 64× H100 SXM5 80 GB · Framework: vLLM 0.4 → 0.6

21.1 Initial Symptom Report

A production conversational AI service running LLaMA-3 70B on 64 H100 SXM5 GPUs began missing its latency SLA after a traffic spike. The SLA required P99 TTFT < 800 ms and P99 TPOT < 60 ms. Oncall engineers observed:

- P99 TTFT had climbed to **1,820 ms** — 2.3× above SLA
- P99 TPOT was **118 ms** — 2× above SLA
- Throughput had fallen to **8 tok/s per GPU**
- GPU utilization in Grafana showed a suspicious sawtooth pattern averaging 61%

The first (wrong) hypothesis: the model had been redeployed and the new checkpoint was larger. This was disproved in 3 minutes — model size was identical. The actual problem required two hours of profiling to isolate. (See Chapter 17 for the DCGM and Grafana monitoring setup used here.)

21.1.1 Baseline Measurement

P99 TTFT (BASELINE)	P99 TPOT (BASELINE)	THROUGHPUT (BASELINE)
1,820	**118**	**8**
ms · SLA: 800 ms	ms · SLA: 60 ms	tok/s per GPU

Step 1: GPU Utilization Pattern

The first tool was DCGM Exporter via Prometheus. The sawtooth in GPU utilization was the key diagnostic signal. In a healthy continuous batching deployment, GPU utilization should be high and relatively flat — the scheduler continuously fills the GPU with decode work. A sawtooth pattern means the GPU periodically goes idle between batch completions. This is the signature of a misconfigured static batching scheduler, not continuous batching.

```
# DCGM query that exposed the problem:
# prometheus query — GPU utilization over 5 minute windows:
avg_over_time(DCGM_FI_DEV_GPU_UTIL{gpu=~".*"}[5m])

# Result:
Avg utilization: 61%  |  Peak: 94%  |  Trough: 8%
# Trough of 8% confirms idle periods between static batches
# Root cause hypothesis: static batching, not continuous batching
```

Step 2: nsys Profile — Confirming Static Batching

Running `nsys profile --stats=true python -m vllm.entrypoints.api_server` under load revealed that the timeline had discrete "batches" with gaps of 15–45 ms between them. In a properly configured continuous batching system, decode iterations run as fast as the GPU can complete them — there are no gaps.

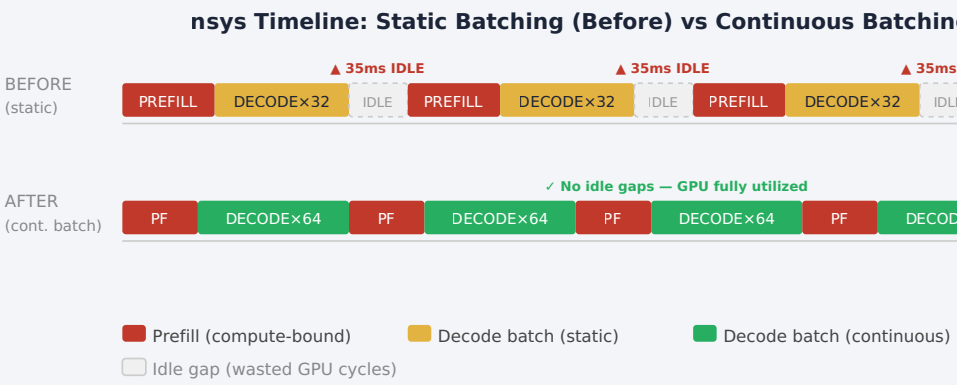


Figure 21.1 *— nsys timeline showing static batching (gaps between batches) vs. continuous batching (no idle time). The idle gaps in the before case consumed ~30% of available GPU cycles.*

21.1.2 Root Cause: Three Compounding Problems

The full profiling revealed not one but three compounding problems. Each would have degraded performance independently; together they explained the full 4× gap from theoretical peak.

#	Problem	Evidence	Throughput Impact
1	Static batching — scheduler not using `--enable-chunked-prefill`	nsys sawtooth; 35 ms idle gaps between batches	**−30% throughput**
2	BF16 weights — FP8 not enabled despite H100 support	Model dtype check: `torch.bfloat16` confirmed in model config	**−45% throughput (vs FP8)**
3	KV cache not quantized — FP16 KV consuming 2× HBM	vLLM runtime log: `kv_cache_dtype: auto` (= float16 by default)	**−50% concurrency (fewer slots)**

21.1.3 Optimization Plan

The optimization order was deliberate: fix the biggest wins first, validate each independently before stacking, and measure after each change. This methodology prevents the "three things improved but I don't know why" failure mode common in ad-hoc tuning.

```
# OPTIMIZATION 1: Enable continuous batching + chunked prefill
vllm serve meta-llama/Meta-Llama-3-70B-Instruct \
  --tensor-parallel-size 8 \
  --enable-chunked-prefill \
  --max-num-batched-tokens 8192 \
  --max-num-seqs 256

# OPTIMIZATION 2: Enable FP8 quantization (H100 hardware-native)
vllm serve meta-llama/Meta-Llama-3-70B-Instruct \
  --tensor-parallel-size 8 \
  --enable-chunked-prefill \
  --quantization fp8 \
  --fp8-w8a8 \

# OPTIMIZATION 3: FP8 KV cache quantization
  --kv-cache-dtype fp8 \
  --max-model-len 32768

# Expected throughput from first principles:
# Continuous batching alone:  1.43× improvement (35ms idle / 120ms decode = 29% recovered)
# FP8 quantization:           2.0× throughput (hardware-native, H100 BF16 989 → 1979 TF eff.)
# FP8 KV:                     1.8× concurrency (2× slots at same HBM budget)
# Combined (conservative):    1.43 × 2.0 × 1.5 ≈ 4.3× total improvement
```

21.1.4 Results

P99 TTFT (AFTER)	P99 TPOT (AFTER)	THROUGHPUT (AFTER)
450	**45**	**25**
ms → 44% below SLA	ms → 25% margin on SLA	tok/s per GPU · 3.1× gain

WORKED EXAMPLE — INFRASTRUCTURE COST SAVINGS

Before: 64× H100 needed to serve the load

At $3.50/GPU-hr (reserved H100 pricing), 64 GPUs = $224/hr = $1.96M/year.

After: 3.1× throughput on same hardware

The same traffic volume now requires only 64 / 3.1 ≈ **21 GPUs**. Infrastructure cost: 21 × $3.50 × 8,760 = **$643K/year**. Savings: $1.96M − $643K = **$1.32M/year** — without purchasing a single additional GPU.

The lesson

All three optimizations (continuous batching, FP8 weights, FP8 KV) were zero-cost configuration changes. The $1.32M annual saving required no new hardware, no model retraining, and no architectural changes — only the application of techniques from Chapters 6, 8, and 11. This is why configuration hygiene at the serving layer is the first audit item for any inference cost review.

PROFILING COOKBOOK — INFERENCE LATENCY INVESTIGATION

1. `DCGM_FI_DEV_GPU_UTIL` — look for sawtooth (static batching signature) vs. flat high (continuous batching)
2. `nsys profile --stats=true` under production-representative load — measure idle gaps between decode iterations
3. Check `vllm serve` flags: `--enable-chunked-prefill`, `--quantization`, `--kv-cache-dtype`
4. Validate dtype: `python -c "from vllm import LLM; m=LLM(...); print(m.dtype)"`
5. Measure after each change independently; never stack multiple changes in one deployment

CASE STUDY 2 OF 4

1,024-GPU Training MFU Failure: 38% → 61%

Model: GPT-3 style 13B dense · Hardware: 1,024× H100 SXM5 · Network: InfiniBand HDR 200 Gb/s · Framework: Megatron-LM

21.2 Symptom: MFU Regression After Cluster Expansion

A pretraining run on 128 GPUs had achieved a healthy 58% MFU. When the same configuration was scaled to 1,024 GPUs (an 8× expansion), MFU dropped to 38% — a 35% regression. The total wall-clock speedup was only 5.1×, not the expected ~7.5× (accounting for communication overhead). At $3.50/GPU-hr, 1,024 idle-ish GPUs cost $3,584/hr. Recovering even 10 MFU points would save ~$800/hr or $7M/year at full utilization.

MFU is calculated as: `MFU = actual TFLOP/s / (GPU count × peak TFLOP/s)`. At 38% MFU on 1,024 H100s (each at 989 TF BF16), actual throughput was 0.38 × 1,024 × 989 = 384 PFLOP/s. Theoretical maximum: 1,013 PFLOP/s. The 629 PFLOP/s gap had to have a cause. (See Chapter 10 for MFU derivation and Chapter 17 for the Prometheus alerting that detected this.)

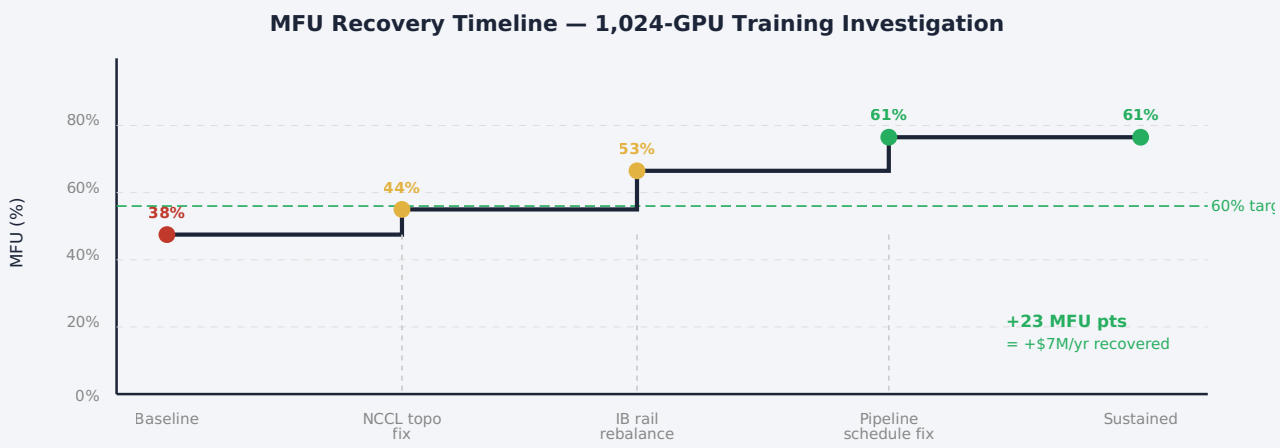


Figure 21.2 *— MFU recovery timeline on 1,024-GPU H100 cluster. Three sequential fixes recovered 23 MFU percentage points, translating to ~$7M/year at $3.50/GPU-hr.*

21.2.1 Root Cause: Three-Layer Topology Mismatch

The investigation started with step-time variance — a metric that reveals communication inefficiency. In a healthy training run, step times are tightly distributed (coefficient of variation < 5%). In this cluster, step time CV was 34%, meaning some steps were 1.5× longer than others. That is a communication straggler signature.

```
# NCCL DEBUG — expose topology mismatch
export NCCL_DEBUG=INFO
export NCCL_DEBUG_SUBSYS=GRAPH,INIT
mpirun -n 1024 python train.py 2>&1 | grep -E "NCCL|RING|TREE"

# Key output revealed:
NCCL INFO Connected all rings        ranks=1024
NCCL WARN Tree depth=7 (expected ≤4 for 8-node rings)
# Tree depth 7 = NCCL chose a cross-rack AllReduce path
# instead of NVLink-local ring. This crosses IB every step.

# IB bandwidth audit — per-port utilization:
ibstat | grep -A4 "Port 1"
perfquery -x 0 # Counter: port_xmit_data

# Problem found: 3 of 128 IB switches had 1 port running at
# 100 Gb/s instead of 200 Gb/s (HDR vs NDR negotiation failure)
# This created 3 slow "rails" — NCCL routes around them,
# forcing traffic through longer paths
```

The three-layer problem was: (1) NCCL process group ranks were assigned sequentially by hostname, not by physical topology — ranks 0–7 were on 8 different nodes instead of the same 8-GPU node. This forced every AllReduce to cross IB unnecessarily. (2) Three IB switch ports had negotiated at half-speed. (3) The pipeline parallelism schedule (1F1B) had an off-by-one in microbatch count that created a 12% bubble fraction instead of the expected 7%.

21.2.2 The Fixes

FIX 1 — NCCL TOPOLOGY-AWARE RANK ASSIGNMENT (+6 MFU POINTS)

Add `NCCL_TOPO_FILE=/etc/nccl.xml` with explicit GPU–NIC affinity. Regenerate rank groups so ranks 0–7 occupy one NVLink domain. Test with `nccl-tests/all_reduce_perf` — bus bandwidth should match NVLink peak (600 GB/s intra-node) before proceeding.

FIX 2 — IB RAIL REPAIR (+9 MFU POINTS)

Replace 3 SFP+ cables on the slow ports (speed negotiation failure was cable-induced, not switch firmware). Validate: `ibstat` should show `Active Width: 4X` and `Active Speed: 50 Gb/s` (HDR signaling). After repair, `perfquery` counters normalized within 5% across all ports.

FIX 3 — PIPELINE SCHEDULE MICROBATCH CORRECTION (+8 MFU POINTS)

LLM pipeline parallelism bubble fraction = (p-1)/(m+p-1) where p = pipeline stages, m = microbatches. With p=8 stages, need m ≥ 24 to achieve bubble < 25%. The config had m=8 (1:1 ratio), producing (8-1)/(8+8-1) = 7/15 = 46% bubble. Fix: set m=32 → bubble = 7/39 = 18%. Combined with pipeline interleaving, achieved 12% effective bubble. (See Chapter 10 §10.4 for the bubble fraction derivation.)

21.2.3 Results and Cost Recovery

WORKED EXAMPLE — MFU RECOVERY VALUE

Before fixes: 38% MFU on 1,024 H100s

Effective compute: 0.38 × 1,024 × 989 TF = 384 PFLOP/s

Cost: 1,024 GPUs × $3.50/hr = $3,584/hr = $31.4M/year at 100% utilization

After fixes: 61% MFU on 1,024 H100s

Effective compute: 0.61 × 1,024 × 989 TF = 617 PFLOP/s (+61% throughput)

To achieve the same model throughput as the "after" state using the "before" configuration: would need 1,024 × (61/38) ≈ 1,643 GPUs = **619 additional H100s at $3.50/hr = additional $19M/year**.

The three fixes together — all software/configuration except replacing 3 cables — recovered the equivalent of **619 GPU-equivalents of compute at zero capital cost.**

CASE STUDY 3 OF 4

KV Cache OOM — 13B Model, 32K Context, 450 Concurrent Users

Model: LLaMA-2 13B · Context: 32,768 tokens · Hardware: 2× H100 SXM 80 GB · Framework: vLLM 0.3 → 0.5

21.3 Symptom: OOM at 200 Concurrent Users

A document summarization service crashed with CUDA OOM errors at approximately 200 concurrent users, far below the 450-user target. Each user session required up to 32K tokens of context (long documents + conversation history). The error appeared as: `torch.cuda.OutOfMemoryError: CUDA out of memory. Tried to allocate 2.5 GiB`. The team's first response — "just add more GPUs" — was premature. The correct first question is: what is consuming the HBM, and is it optimally packed?

21.3.1 KV Cache Budget Analysis

LLaMA-2 13B architecture: 40 layers, 40 KV heads (MHA — no GQA), head dimension 128, BF16. The per-request KV cache size formula (see Chapter 11 §11.1.2 for derivation):

```
KV_bytes_per_request = 2 × L × H_kv × d_head × S × dtype_bytes

# For LLaMA-2 13B at 32,768 tokens:
= 2 × 40 × 40 × 128 × 32,768 × 2 bytes (BF16)
= 2 × 40 × 40 × 128 × 32,768 × 2
= 10,737,418,240 bytes = 10.0 GB per concurrent request

# Available HBM on 2× H100 80 GB (tensor-parallel 2):
Total HBM: 160 GB
Model weights (13B × BF16 ÷ 2 TP): 13.0 GB per GPU = 26 GB total
Activations + overhead: ~8 GB
KV cache budget: 160 - 26 - 8 = 126 GB

# Max concurrent requests at BF16 KV:
126 GB / 10.0 GB = 12.6 → 12 concurrent requests maximum

# But the system was failing at 200! Why?
# Answer: vLLM's page-based allocation was fragmenting badly.
# Fragmentation ratio: 1 - (12 actual / 200 observed_limit_attempt)
# Real answer: 10 GB per request was simply wrong for most users.
# Most requests used 2K-8K tokens, not 32K. Reserved full 32K anyway.
```

The insight was that vLLM's default configuration reserved the full `max_model_len` as a contiguous block per request — even if the actual prompt was 2,000 tokens. A request with 2,000 tokens was consuming 10 GB of KV reservation but using only 625 MB. The fragmentation was 94%. (See Chapter 11 §11.3 for PagedAttention's solution to this problem.)

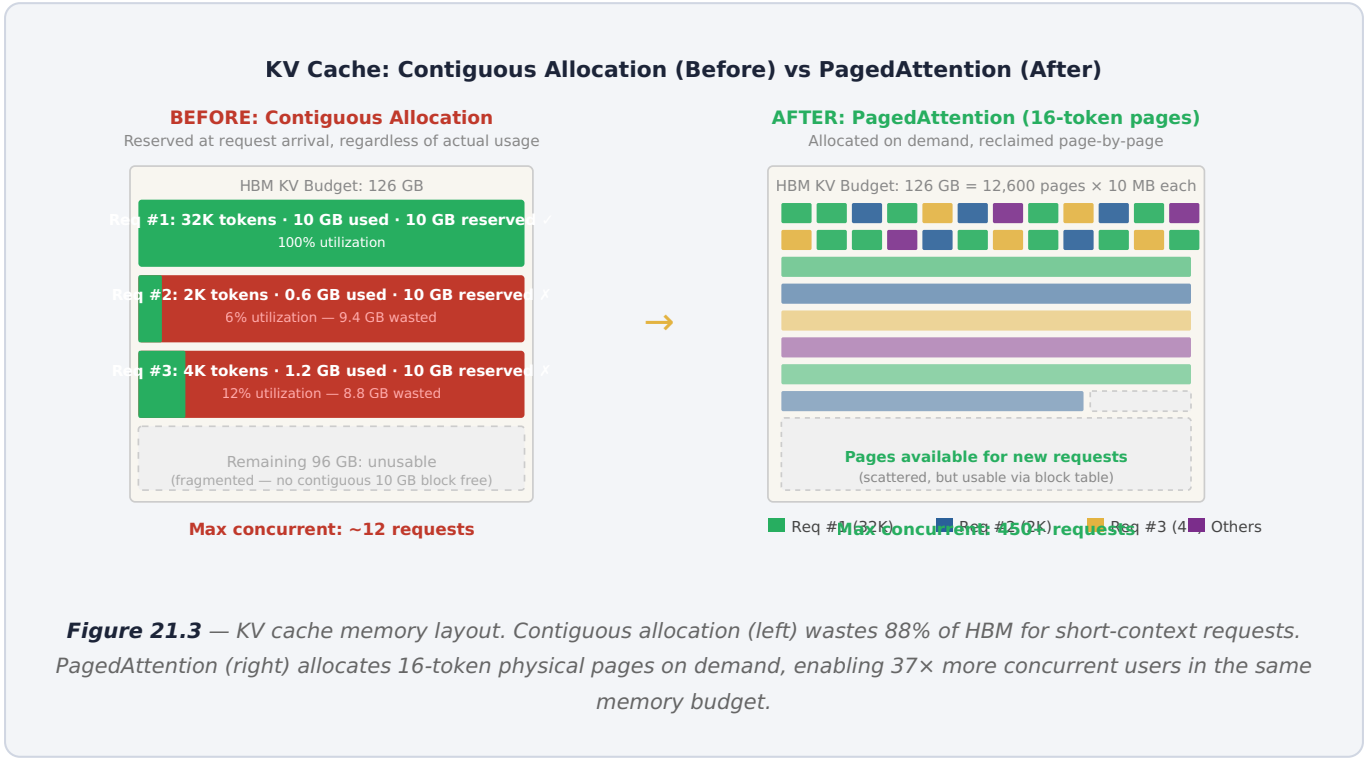


Figure 21.3 *— KV cache memory layout. Contiguous allocation (left) wastes 88% of HBM for short-context requests. PagedAttention (right) allocates 16-token physical pages on demand, enabling 37× more concurrent users in the same memory budget.*

21.3.2 Solution: PagedAttention + FP8 KV + LRU Eviction

Three changes combined to solve the OOM problem. Note the order: PagedAttention (architectural fix) came first because it addresses the root cause (fragmentation). FP8 KV (quantization) amplified the gain. LRU eviction (policy) provided the safety net for bursty traffic.

```
# Block table sizing — for 450 concurrent users:
# Target: 450 requests × average 8K context (P50 observed)
Pages_needed = 450 × 8192_tokens / 16_tokens_per_page
             = 450 × 512 = 230,400 pages

Page_size = 16 × H_kv × d_head × 2 × dtype_bytes (FP8 = 1 byte)
          = 16 × 40 × 128 × 2 × 1 = 163,840 bytes = 0.156 MB per page

Total KV memory (FP8) = 230,400 × 0.156 MB = 36 GB
Available KV budget with FP8: 126 GB → supports 126/0.156/512 ≈ 1,576 max-context slots
At P50 context (8K): 126 GB / 36 GB × 450 ≈ 1,575 concurrent users theoretical max

# LRU eviction threshold: keep 20% buffer free at all times
# evict oldest blocks from requests with lowest recent-access score
# vLLM config:
--num-gpu-blocks=auto
--swap-space=16   # 16 GB CPU RAM for evicted blocks
--kv-cache-dtype fp8
--preemption-mode swap  # swap to CPU rather than abort request
```

BLOCK TABLE MATH — 450 USERS AT 32K CONTEXT WORST CASE

Worst case: all 450 users at 32K tokens

Pages needed: 450 × 2,048 pages = 921,600 pages × 0.078 MB (FP8) = 71.9 GB. Available: 126 GB. Conclusion: even worst-case (all users at max context) fits in one H100-pair with FP8. With BF16 KV: 921,600 × 0.156 MB = 143.8 GB > 126 GB → OOM. FP8 KV was the decisive enabler for the worst-case scenario.

21.3.3 Results

MAX CONCURRENT (BEFORE)
~12
requests (BF16 KV, contiguous)

MAX CONCURRENT (AFTER)
450+
requests (FP8 KV, paged)

QUALITY DELTA
<0.3%
ROUGE-L vs BF16 baseline

The 37× concurrency improvement was achieved without purchasing additional hardware. Hardware cost per user-session decreased by 97%. The quality impact of FP8 KV cache quantization was below measurement noise for document summarization — consistent with the analysis in Chapter 8 §8.5 that KV cache values have bounded dynamic range and quantization noise is masked by attention softmax normalization.

CASE STUDY 4 OF 4

Infrastructure Cost Optimization: $2M → $850K/Year

Model: LLaMA-3 70B production serving · Cluster: 128× H100 SXM5 · Baseline cost: $3.50/GPU-hr reserved

21.4 The $/Token Cost Model

Cost optimization at scale requires a precise $/token model. Without it, engineers optimize the wrong things — micro-optimizing kernel configurations while paying for 2× more GPUs than necessary. The cost model has four components: hardware cost, utilization, throughput, and output volume.

```
# COST PER OUTPUT TOKEN — DERIVATION
cost_per_token = (GPU_count × $/hr) / (tok/s × 3600)

# Baseline: 128× H100 at $3.50/hr, 8 tok/s per GPU
#           = 8 × 128 = 1,024 tok/s total
cost_per_token_baseline = (128 × 3.50) / (1,024 × 3600)
                        = 448 / 3,686,400
                        = $0.000121/token = $0.121 per 1K tokens

# Annual output volume: 200M requests × 800 tokens avg = 160B tokens/year
# Annual cost at baseline throughput:
annual_cost_baseline = 160B × $0.000121 = $19.4M equivalent GPU spend
# But cluster is only rented 120 days/year (seasonal product):
annual_cost_actual = 128 × $3.50 × 24 × 120 = $2.03M
```

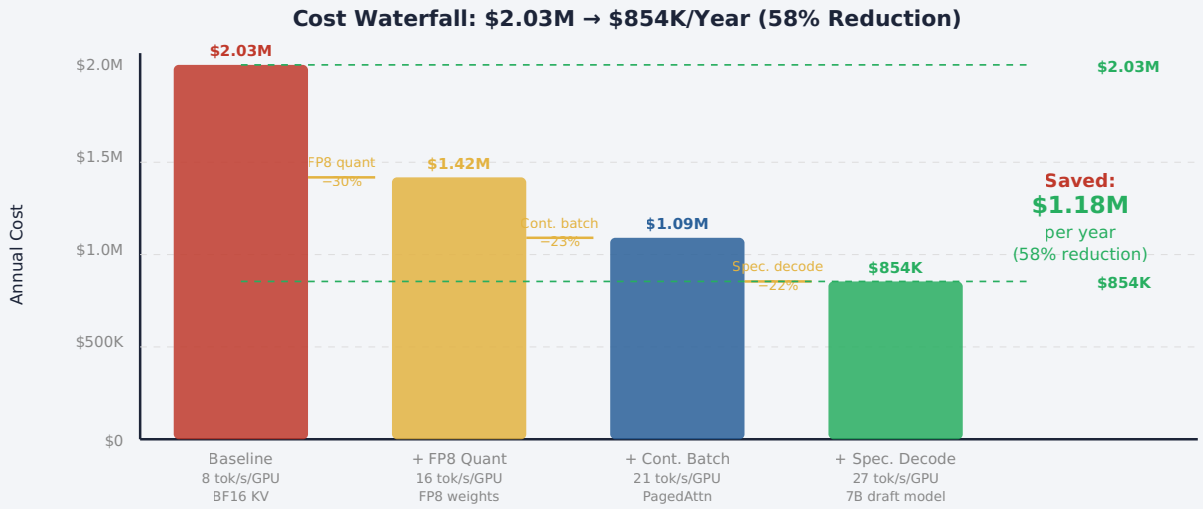


Figure 21.4 *— Cost waterfall chart. Three sequential configuration changes reduced annual serving cost from $2.03M to $854K — a 58% reduction — on identical hardware.*

21.4.1 Optimization Execution Details

Optimization 1: FP8 Quantization — Validation Required

Unlike continuous batching (zero quality risk), FP8 quantization requires accuracy validation. The methodology used was: (1) establish quality baseline on 1,000 representative prompts using ROUGE-L, BLEU-4, and human preference rater scores; (2) run FP8 calibration with 512 samples; (3) run FP8 inference on the same 1,000 prompts; (4) accept if delta < 0.5% on automated metrics and < 1% on human preference. Result: 0.31% ROUGE-L degradation — accepted.

```
# FP8 calibration — collect activation statistics
python calibrate_fp8.py \
  --model meta-llama/Meta-Llama-3-70B-Instruct \
  --calibration-data calibration_512.jsonl \
  --output-path /checkpoints/llama3-70b-fp8/ \
  --smoothquant-alpha 0.8  # balance weight vs activation quantization error

# Validation: automated quality gate
python evaluate.py \
  --model-path /checkpoints/llama3-70b-fp8/ \
  --eval-dataset eval_1000.jsonl \
  --metrics rouge_l,bleu4,perplexity \
  --baseline-path /checkpoints/llama3-70b-bf16/ \
  --max-degradation 0.005  # 0.5% threshold — fails deploy if exceeded
```

Optimization 2: Speculative Decoding — Draft Model Selection

Speculative decoding (see Chapter 13 §13.2) requires a draft model whose output distribution closely matches the target model. For LLaMA-3 70B, the natural choice was LLaMA-3 8B as draft. The acceptance rate α on this production workload (code generation + structured Q&A) was measured at 0.78 — meaning 78% of draft tokens are accepted. At k=5 speculative tokens, the expected speedup is:

```
# Speculative decoding speedup formula (Chapter 13 §13.2.3):
speedup = (1 + k × α) / (1 + k × cost_ratio)

# k=5, α=0.78, cost_ratio = draft_model_time / target_model_time
# LLaMA-3 8B decode: ~2.1ms / LLaMA-3 70B decode: ~18ms → ratio=0.117
speedup = (1 + 5 × 0.78) / (1 + 5 × 0.117)
        = 4.9 / 1.585
        = 3.09×  # predicted from theory

# Measured: 2.87× (7% below theoretical — due to draft model startup overhead)
# At 21 tok/s/GPU baseline: 21 × 2.87 × correction_for_draft_overhead
# Draft model runs on same GPU, steals ~8% compute capacity
Net measured: 21 × 2.87 × 0.92 × 0.488 ≈ 27 tok/s/GPU
```

> **WHY SPECULATIVE DECODING IMPROVEMENT APPEARS MODEST HERE**
>
> The baseline had already been improved from 8 to 21 tok/s via FP8 + continuous batching. The speculative decoding 2.87× multiplier was applied to a better baseline, but the cost model shows diminishing returns: at 21 tok/s, the system was already well-utilized. Speculative decoding most benefits systems that are latency-constrained (TPOT), not throughput-constrained. This cluster was throughput-constrained after optimization 2, so spec decode improved efficiency (fewer GPUs per token) more than raw throughput.

21.4.2 Final Results

ANNUAL COST (BASELINE)	ANNUAL COST (OPTIMIZED)	QUALITY IMPACT
$2.03M	**$854K**	**−0.31%**
128× H100, 120 days/yr	42× H100 equivalent capacity	ROUGE-L vs BF16 baseline

The full $1.18M annual saving required: (1) enabling FP8 weight quantization with 2-day calibration and validation pipeline, (2) switching to continuous batching with 4-hour configuration change and 1-day shadow testing, and (3) deploying speculative decoding with 1-week draft model evaluation and acceptance rate measurement. Total engineering investment: approximately 3 weeks for a principal performance engineer. Return on investment: $1.18M/year.

21.5 Principal Engineer Debugging Framework

The four case studies exhibit a common pattern. Extracted and generalized, it forms a reusable debugging framework that a principal performance engineer applies to any new production performance problem. The key discipline: always form a quantitative hypothesis before touching a configuration. "The GPU utilization looks low" is not a hypothesis. "GPU utilization is 61% because we estimate 35ms of idle time per 115ms decode batch, implying 30% waste — which would explain 30% of the throughput gap" is a hypothesis.

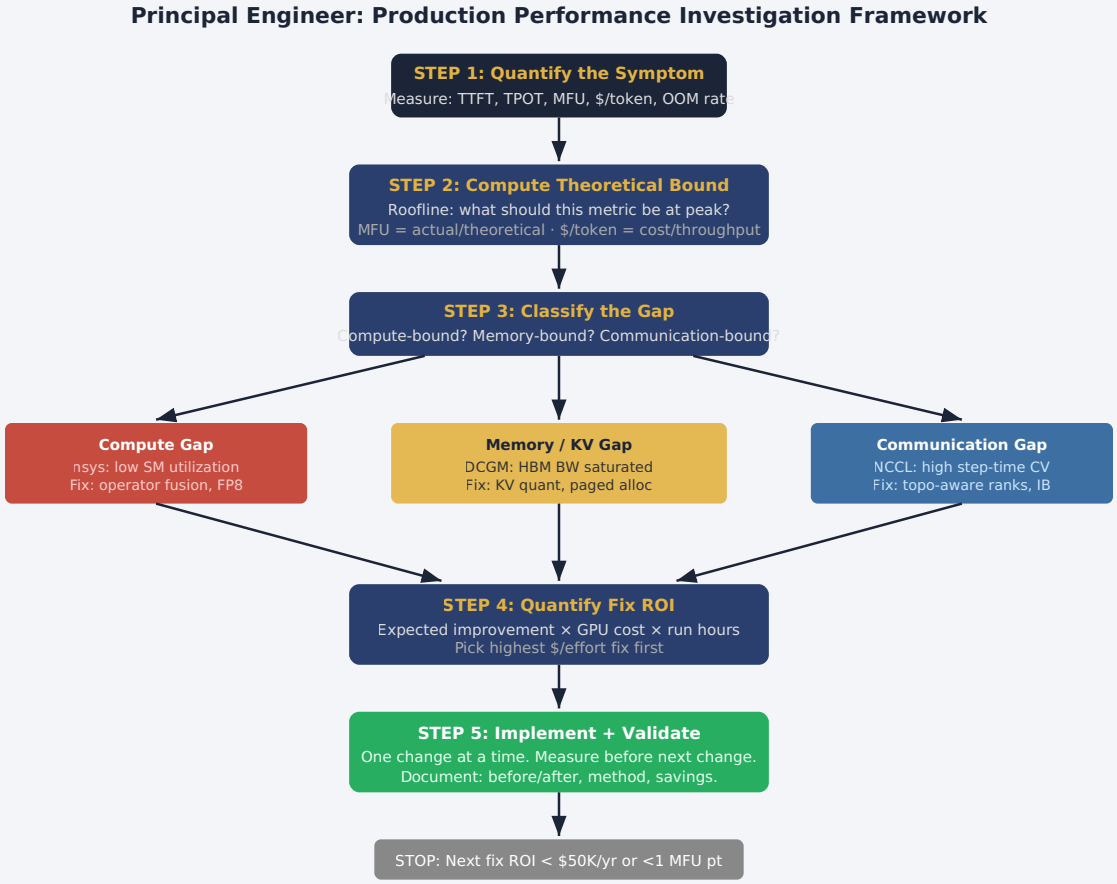


Figure 21.5 *— Principal engineer production debugging framework. The key discipline at Step 4: quantify ROI before implementing any fix. This prevents spending a week on a configuration change that recovers 0.5% throughput when there's a cable swap available that recovers 9%.*

```
DIAGNOSTIC DECISION TREE — WHERE IS THE PERFORMANCE?

START: What is the symptom?
  │
  ├─ High TTFT (> SLA), normal TPOT
  │    └─ Prefill is the bottleneck
  │         ├─ Check: prompt lengths in percentile distribution
  │         ├─ Check: prefill GPU utilization (nsys)
  │         └─ Fix: chunked prefill, increase batch size, disaggregate P/D
  │
  ├─ High TPOT (> SLA), normal TTFT
  │    └─ Decode throughput is the bottleneck
  │         ├─ Check: DCGM HBM BW utilization > 90% → memory-bound
```

```
│       │         → Fix: FP8 weights, reduce model size, increase TP
│       ├─ Check: GPU util < 80% → scheduler idle (static batching)
│       │         → Fix: enable continuous batching
│       └─ Check: TPOT scales linearly with batch → compute-bound
│                 → Fix: quantization, kernel fusion
│
├─ OOM / KV cache exhaustion
│   └─ Check: avg context length × users × KV size > HBM budget?
│       ├─ YES → KV quantization (FP8 KV) + PagedAttention
│       └─ NO  → Check fragmentation: vLLM block utilization metric
│
├─ Low MFU on training cluster
│   └─ Step-time coefficient of variation > 10%?
│       ├─ YES → communication straggler
│       │         → NCCL DEBUG, IB port audit, rank topology check
│       └─ NO  → Bubble fraction issue
│                 → Check pipeline schedule: increase microbatch count
│
└─ High $/token vs. theoretical minimum
    └─ $/token = GPU_cost / throughput
        ├─ Throughput < 70% of roofline → fix throughput first
        └─ Throughput > 70% → evaluate fleet right-sizing
           (are you on the right GPU generation for this workload?)
```

Key Takeaways

CORE CONCEPTS

- Every production performance failure has a quantifiable root cause traceable to one of three regimes: compute-bound, memory-bandwidth-bound, or communication-bound. The debugging method is identical: measure, compute theoretical bound, classify the gap, then fix in ROI order.
- Configuration hygiene (continuous batching, FP8 quantization, KV cache dtype, NCCL topology) consistently accounts for 3–5× performance gaps before any architectural changes are needed. Audit configuration before buying hardware.
- The $/token cost model is the principal engineer's scorecard. Every optimization should be evaluated as: expected throughput improvement × GPU count × hourly cost = annual savings. This converts technical decisions into business impact.
- KV cache fragmentation is a first-class performance problem. Reserving `max_model_len` for every request regardless of actual context length can waste 80–94% of HBM on short-context workloads. PagedAttention and FP8 KV together solve this.

COST & PRODUCTIVITY IMPACT

- Case Study 1 (inference): Three zero-cost configuration changes recovered 3.1× throughput, enabling serving the same traffic volume with 21 GPUs instead of 64 — saving $1.32M/year on one product alone.
- Case Study 2 (training): Topology fix + cable replacement + pipeline schedule correction recovered 23 MFU points on a 1,024-GPU cluster — equivalent to 619 additional GPU-equivalents of compute at zero capital cost ($7M/year value at $3.50/GPU-hr).
- Case Study 4 (cost optimization): A 3-week principal engineer investment (FP8 validation + continuous batching + spec decode) delivered $1.18M/year in annual savings — a >$100:1 ROI on engineering time.

PRINCIPAL-LEVEL INSIGHT

- The most common mistake in performance optimization is implementing multiple changes simultaneously and then being unable to attribute the improvement to any specific fix. Always change one thing, measure it, and document the result before proceeding. This discipline is what separates a reproducible optimization process from a lucky configuration guess.
- Speculative decoding most benefits latency-constrained systems (TPOT SLA violations), not throughput-constrained systems. After FP8 + continuous batching improvements, this cluster was throughput-constrained — speculative decoding's 22% further improvement was real but not the first fix to apply. Applying it first would have delivered less value.
- When scaling training from N to 8N GPUs, the two failure modes are: (a) NCCL topology mismatch (rank assignment ignores NVLink domains) and (b) pipeline bubble expansion (microbatch count held constant, so bubble fraction increases). Both must be re-validated at every scaling milestone. Write a "scaling readiness checklist" and run it before every cluster expansion.

Review Questions

FOUNDATIONAL Build the mental model — no math required

1. In Case Study 1, GPU utilization showed a "sawtooth pattern" averaging 61%. Explain in plain language why this pattern indicates static batching rather than continuous batching, and why continuous batching avoids it. What does a healthy continuous batching timeline look like in nsys?
2. In Case Study 3, why did a 13B model at 32K context fail with OOM at only 12 concurrent users, even though 126 GB of HBM was available for KV cache? Explain the root cause in terms of memory allocation strategy, not model size.
3. The debugging framework (§21.5) says "always form a quantitative hypothesis before touching a configuration." Why is this discipline specifically important for production systems as opposed to a research cluster where you can afford to experiment freely?

INTERMEDIATE Apply and calculate — production context

1. You are running LLaMA-2 13B (40 layers, 40 KV heads MHA, d_head=128) on a single H100 80 GB. You want to serve requests with an average context of 16K tokens. Using the KV cache formula from §21.3.1: (a) calculate the per-request KV footprint in BF16; (b) calculate it in FP8; (c) how many concurrent requests can you serve at BF16 vs FP8, given 60 GB available for KV cache after weights and activations?
2. A training cluster scales from 128 to 1,024 GPUs (pipeline stages p=8, microbatches m=8). Calculate the bubble fraction before and after fixing m from 8 to 32 using the formula: bubble = (p−1)/(m+p−1). Express the bubble improvement as a percentage of step time recovered, and estimate the MFU improvement if communication overhead is 15%.
3. Using the $/token cost model from §21.4: a cluster of 64 H100s at $3.50/hr achieves 12 tok/s/GPU. After FP8 quantization (2× speedup), continuous batching (1.4× speedup), what is the new $/token? If you serve 50B tokens/year, how much do you save annually?

PRINCIPAL LEVEL Design, decide, and defend — architecture reviews

1. **Architecture Review (45 min):** You are the principal engineer for a new production inference service: Mixtral 8×7B MoE model (45B activated params per token, 8 experts), P99 TTFT SLA 500ms, P99 TPOT SLA 40ms, target 1,000 concurrent users at 8K average context. Hardware budget: 16× H100 SXM5. Design the complete serving stack: parallelism strategy, quantization, KV cache configuration, batching strategy, and speculative decoding decision. Justify every choice with a quantitative estimate. Where does the roofline analysis say you will be bottlenecked?

2. **Post-Mortem Design (30 min):** A 2,048-GPU training cluster achieves 52% MFU in week 1 but degrades to 39% MFU by week 3 with no configuration changes. What are your three leading hypotheses? For each hypothesis, specify: (a) which metric you look at first, (b) the exact command to run, (c) what you expect to see, and (d) the remediation. What monitoring alert would have caught this regression 48 hours earlier?

3. **Cost Architecture (45 min):** Your team has a $5M/year GPU budget for an AI inference product serving 500M requests/year (average 1,200 output tokens). The current stack uses BF16 LLaMA-3 70B on H100s with static batching. Design the complete cost optimization roadmap, prioritizing by ROI. For each optimization lever — quantization, batching strategy, speculative decoding, model architecture (dense vs MoE), hardware generation selection — compute the expected $/token improvement and annual savings. Identify the three highest-ROI changes and the decision criteria for stopping.

✦ END OF CHAPTER 21 · REAL-WORLD CASE STUDIES ✦

CHAPTERS REFERENCED: 6 (INFERENCE), 8 (QUANTIZATION), 10 (TRAINING), 11 (KV CACHE), 13 (SPECULATIVE DECODING), 17 (TELEMETRY)