PART III · CHAPTER 8

# Quantization & Precision Optimization

*Halving Memory, Doubling Throughput — The Math and Engineering Behind Numeric Compression*

**Book:** AI/ML Infrastructure from Silicon to Scale | **Author:** Venkat Vinjam | **Cross-ref:** *Roofline (Ch 1) · HBM bandwidth (Ch 4) · Tensor Cores (Ch 7) · KV cache paging (Ch 11) · Inference serving (Ch 6)*

> *"Quantization is not about making your model less accurate. It is about using exactly as many bits as the task requires — no more, no less. The difference between FP32 and FP8 for LLM inference is like using a crane to move a pencil."*
>
> — Production Inference Engineering Principle

A 70B parameter model in FP32 weighs 280 GB — more than three H100 GPUs can hold. In FP8, it weighs 70 GB and fits on one. In INT4, 35 GB — with room left over for a generous KV cache and concurrent user pool. Quantization is the engineering discipline of reducing numeric precision without unacceptable quality loss, and it is arguably the highest ROI optimization in modern AI infrastructure: a single change from BF16 to AWQ INT4 + FP8 KV cache reduces serving cost by 72% at less than 1% quality degradation.

This chapter covers every precision format from FP32 to FP4, every quantization technique from naive PTQ to activation-aware algorithms, the outlier problem that breaks naive approaches, the hardware that accelerates each format, and the complete decision framework for choosing a quantization strategy given hardware, model, and quality constraints. Every claim carries a number you can verify on your own cluster.

**FOUNDATIONAL**

- Explain the sign/exponent/mantissa structure and the range-vs-precision tradeoff
- Understand why BF16 replaced FP16 for training and why the difference matters
- Know the memory footprint of each format for any parameter count
- Describe the outlier problem and why it breaks naive INT8

**INTERMEDIATE**

- Derive quantization error for any format and value distribution
- Explain GPTQ, AWQ, and SmoothQuant — how each addresses outliers
- Calculate KV cache memory at various context lengths and dtypes
- Apply the Transformer Engine FP8 workflow with zero code changes

**PRINCIPAL LEVEL**

- Design quantization strategy given hardware, model, and quality SLA
- Calculate full cost-per-token impact of each format choice
- Choose among FP8/AWQ/GPTQ/GGUF and justify quantitatively
- Evaluate quantization quality rigorously across benchmark tiers

# 8.1 Floating Point Formats — The Precision Landscape

## 8.1.1 Number Formats from FP32 to FP4

Every floating point number uses three fields: a sign bit, an exponent (controls range), and a mantissa (controls precision within that range). The fundamental engineering tradeoff: bits given to the exponent widen representable range; bits given to the mantissa improve precision. Every AI format is a different point on this spectrum, chosen for a specific workload characteristic.

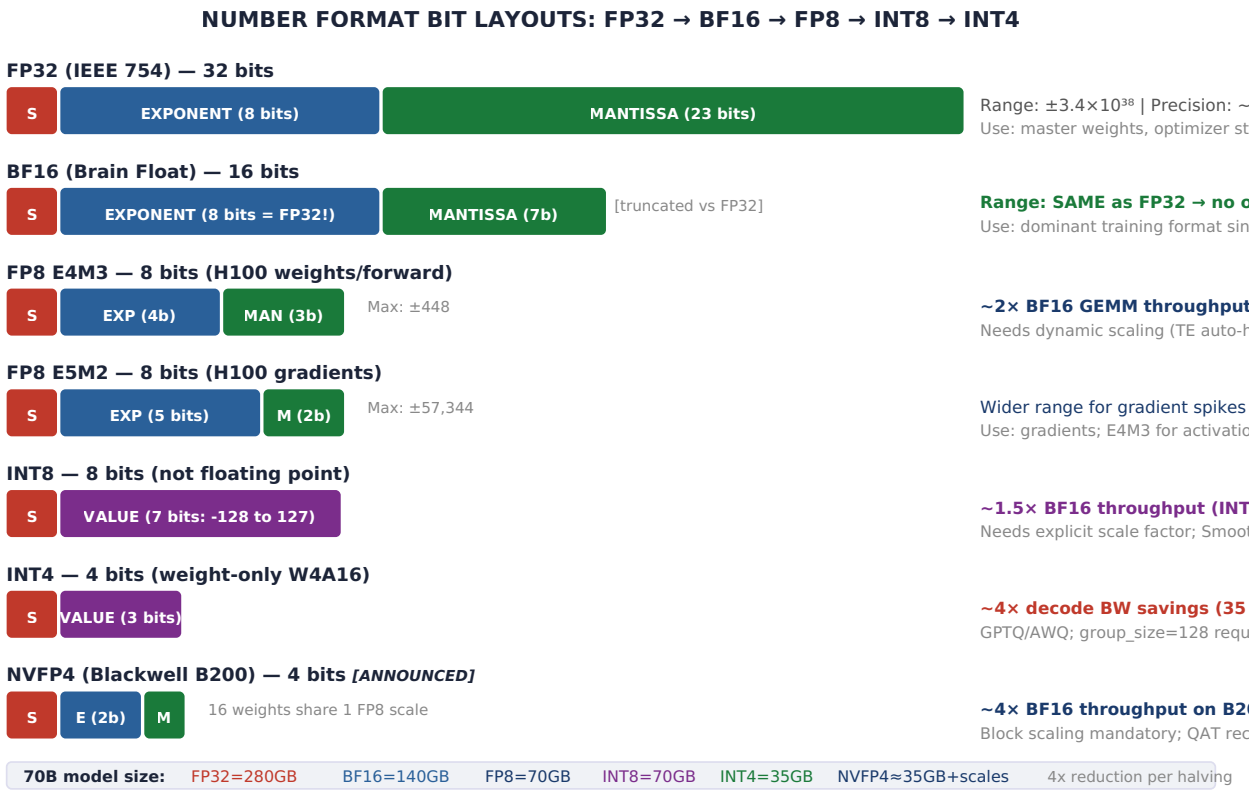



***Figure 8.1*** *— Number format bit layouts from FP32 to NVFP4. The key insight: BF16 keeps FP32's full 8-bit exponent (same range, no overflow risk) while halving storage. FP8 E4M3 is the inference format for weights; E5M2's wider exponent serves gradient spikes in backward passes. INT4 has no floating-point semantics — it requires an explicit group-scale factor every 128 weights for acceptable quality.*

## 8.1.2 Why BF16 Replaced FP16 for Training

The replacement of FP16 by BF16 as the dominant training format was driven by one specific failure mode: overflow. Understanding this concretely is essential because the format choice affects every training run you will ever design.

```
THE FP16 OVERFLOW PROBLEM:

FP16 maximum representable value: 65,504
BF16 maximum representable value: ~3.4 × 10^38  (same as FP32)

LLM training gradient scenario:
  Normal gradient step: 1.2 × 10^4  → FP16 fine (< 65,504)

  After a learning rate spike or early training instability:
  Gradient: 8.0 × 10^4  → FP16 OVERFLOW (> 65,504)!
  Result: gradient → Inf → weight update = NaN → training crash

  BF16 with 8-exponent bits: same max as FP32 → NEVER overflows in practice

THE TRADEOFF (what BF16 sacrifices):
  FP16: 10 mantissa bits = 3.3 decimal digits of precision
  BF16:  7 mantissa bits = 2.3 decimal digits of precision

  For gradient updates: 2.3 decimal digits is SUFFICIENT.
  Gradients are noisy estimates; sub-1% precision is wasted anyway.
  For weights: master weights stay in FP32; BF16 just carries the update.

Loss scaling (legacy FP16 workaround, now obsolete with BF16):
  Multiply loss by S=2^15 before backward → gradients are S× larger
  Divide by S before optimizer step → maintains FP16 range
  Dynamic S: halve on inf/nan, double on clean step
  Still needed: FP16 training on pre-Ampere hardware without BF16 support
```

### 8.1.3 The Numeric Format Decision Matrix

| Use Case | Recommended | Reason | Alternative |
|---|---|---|---|
| LLM pre-training (weights) | **BF16** | Safe range, gradient-stable | FP16 + loss scaling (legacy) |
| Master weights (optimizer) | **FP32** | Precision critical for m,v states | BF16 (quality risk) |
| Training GEMM on H100 | **FP8 E4M3 (TE)** | 2× BF16 GEMM throughput | BF16 (safer, slower) |
| Training gradients (H100) | **FP8 E5M2 / BF16** | E5M2 wider range for spikes | BF16 always safe |
| Inference (H100, production) | **FP8 W8A8** | Best quality/perf; near-lossless | AWQ INT4 (if memory constrained) |
| Inference (memory-constrained) | **INT4 W4A16 (AWQ)** | 4× memory reduction; acceptable quality | GPTQ INT4 |
| Inference (B200/GB300) | **NVFP4 W4A4** | Maximum throughput, hardware native | FP8 W8A8 |
| KV cache (production) | **FP8 or INT8** | 2× KV capacity, minimal quality loss | BF16 (baseline) |
| Edge / CPU / laptop | **GGUF Q4_K_M** | llama.cpp optimized, cross-platform | GGUF Q5_K_M (higher quality) |

## 8.2 Quantization Mathematics

### 8.2.1 Uniform Affine Quantization — The Foundation

All practical quantization schemes build on a linear mapping from floating-point values to integers. Understanding this foundation precisely — including what fails and why — is the prerequisite for understanding every advanced technique in sections 8.3 and beyond.

```
QUANTIZATION FORMULA (uniform affine):

  Q(x) = clamp(round(x / scale + zero_point), q_min, q_max)
  x_hat = scale × (Q(x) − zero_point)      ← dequantization (reconstruction)

Where:
  scale     = (max(x) − min(x)) / (q_max − q_min)   ← step size
  zero_point = q_min − round(min(x) / scale)
  q_min/max = integer bounds (−128 to 127 for INT8 signed)

SYMMETRIC QUANTIZATION (zero_point = 0, simpler hardware):
  scale = max(|x|) / q_max       ← anchored at zero
  Q(x) = round(x / scale)
  Used by: NVIDIA cuBLAS INT8 GEMM (hardware prefers zero_point=0)

ASYMMETRIC QUANTIZATION (zero_point ≠ 0, full range):
  scale = (max − min) / 255    ← full range utilization
  Better for activations (ReLU outputs are always positive → asymmetric dist.)
  Used by: TFLite, ONNX Runtime, activation-quantized deployments

QUANTIZATION ERROR:
  ε = x_hat − x = scale × (round(x/scale) − x/scale)
  Maximum error: |ε|_max ≤ scale/2   ← half a quantization step

WORKED EXAMPLE (INT8 symmetric, weight range [−1, +1]):
  scale = 1.0 / 127 = 0.00787
  Max error = 0.00787/2 = 0.00393   ← 0.4% relative error at max value
  Error for x=0.001: relative error = 0.00393/0.001 = 393%! ← small values DESTROYED

KEY INSIGHT: Quantization error is uniform in absolute terms but catastrophic
in relative terms for small values. Outliers force a wide scale,
which makes small-magnitude values — potentially equally important — vanish.
```

### 8.2.2 Per-Tensor vs Per-Channel vs Per-Group Quantization

The granularity of scale factor computation is one of the most consequential quality-efficiency tradeoffs in quantization engineering. Each level of granularity costs more in metadata storage but recovers more quality.

```
QUANTIZATION GRANULARITY COMPARISON:

                    Per-Tensor      Per-Channel       Per-Group (g=128)
                    (1 scale/layer) (1 scale/row)     (1 scale/128 weights)
──────────────────────────────────────────────────────────────────────
Scale factors:      1 per layer     N_out scales      70B/128 = 547M scales
Scale overhead:     Negligible      Small             ~1.09 GB for 70B model
Quality (INT8):     Poor            Good              Excellent
Quality (INT4):     Unacceptable    Marginal          Production-acceptable
HW compute cost:    Lowest          Low               Medium (dequant per group)
Used in:            Legacy only     Some INT8 libs    GPTQ, AWQ, GGUF

EXAMPLE — GPTQ INT4 per-group-128 for LLaMA-3 70B:
  Groups:       70B / 128 = 546,875,000
  Scale memory: 546,875,000 × 2 bytes (FP16) = 1.09 GB
  Total model:  35.0 GB (INT4) + 1.09 GB (scales) = 36.09 GB

QUALITY COMPARISON (LLaMA-3 70B, 5-shot MMLU):
  BF16 baseline:            80.3%
  INT4 per-tensor (naive):  72.1%  ← −8.2% NEVER use
  INT4 per-group-128 AWQ:   79.5%  ← −0.8% Production-acceptable
```

### 8.2.3 The Outlier Problem — Why Transformers Break Naive Quantization

Transformer models contain a small but devastating minority of "outlier" activations — values 10–100× larger than typical. These outliers are the primary cause of quality degradation in naive quantization schemes, and every advanced PTQ technique (GPTQ, AWQ, SmoothQuant) exists specifically to address them.

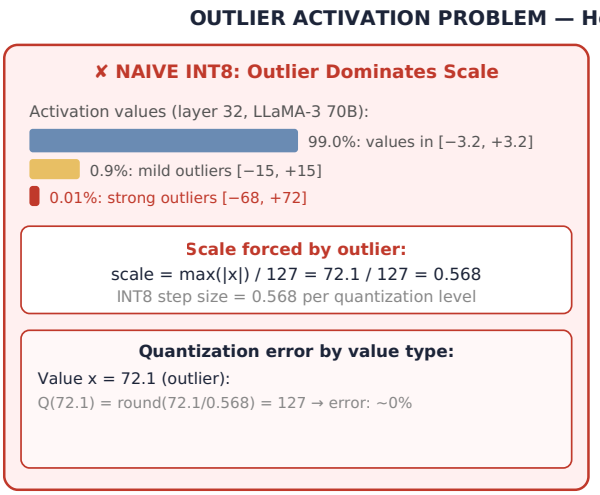



***Figure 8.2*** *— The outlier problem in transformer activations. A tiny fraction (0.01%) of very large activation values forces the INT8 scale to be wide, which quantizes all typical small-magnitude values to zero — 100% relative error. AWQ identifies these salient channels and scales their corresponding weights up before quantization, so they use more of INT4's representable range. Net mathematical effect on output: zero. Net quantization quality effect: recovers 7+ MMLU points vs naive INT4.*

## 8.3 Post-Training Quantization (PTQ) Techniques

PTQ applies quantization after training is complete, using a small calibration dataset (~128–512 samples) to compute scale factors. No model retraining is required — critical for fast deployment of large models. The three techniques below represent the state of the art for three distinct trade-offs between calibration cost, quality, and outlier handling.

### 8.3.2 GPTQ — Second-Order Weight Quantization

GPTQ (Frantar et al., 2022) is the breakthrough that made INT4 inference practical. It uses Hessian-based second-order information to compensate for each weight's quantization error in subsequent weights within the same row.

```
GPTQ ALGORITHM (conceptual):

KEY INSIGHT: When weight w_i is quantized to INT4, error ε_i is introduced.
Instead of accepting this error, GPTQ adjusts remaining weights in the
same row to cancel ε_i's effect on the layer output.

For one weight matrix W [out_features × in_features]:
  1. Compute Hessian: H = 2 X^T X  (X = calibration activations)
     H[i,i] = sensitivity of output to perturbation of weight column i

  2. For each column i (in order of increasing H[i,i]):
     a. Quantize: W_q[:,i] = INT4_round(W[:,i] / scale)
     b. Error: ε = W[:,i] − dequant(W_q[:,i])
     c. Propagate to remaining: W[:,i+1:] -= (ε × H[i, i+1:]) / H[i,i]
        Remaining columns are ADJUSTED to compensate for column i's error.

  3. Result: INT4 weights where errors are balanced rather than accumulated

QUALITY (LLaMA-3 70B, group_size=128):
  Naive INT4 PTQ:    MMLU 71.8%  (−8.5%)
  GPTQ INT4 W4A16:  MMLU 79.0%  (−1.3%)  ← 7.2 points recovery from compensation!

COST:
  Calibration time: ~2–4 hours on 8× H100 for 70B model
  Storage: 35 GB (INT4 weights) + 1.09 GB (scales) = 36 GB total
  Tools: AutoGPTQ (HuggingFace), ExLlamaV2, llama.cpp
```

### 8.3.3 AWQ — Activation-Aware Weight Quantization

AWQ (Lin et al., 2023) protects the weights that matter most before quantization, rather than fixing errors after. It is generally faster to calibrate than GPTQ and performs comparably or better on models with pronounced outliers.

```
AWQ CORE ALGORITHM:

Observation: ~1% of weight channels correspond to large-magnitude activation channels.
These have outsized impact. Protect them; quantize everything else aggressively.

Step 1: Identify salient channels
  Run calibration data → a_i = mean(|X[:,i]|) for each input channel i
  Top 1% by magnitude = "salient"

Step 2: Scale up salient weights, scale down corresponding activations
  For each salient channel i:
    W[:,i] *= s_i    ← weight uses more of INT4 range
    X[:,i] /= s_i    ← activation compensates (net output: unchanged!)
  Scale: s_i = mean(|X[:,i]|)^0.5  (geometric mean balances the tradeoff)

Step 3: Quantize all weights to INT4 (group_size=128)
  Salient channels quantize more accurately (use more of range after scaling)
  Store: INT4 weights + per-channel s_i factors + group scales

QUALITY vs GPTQ (LLaMA-3 70B W4A16):
  AWQ:  MMLU 79.5% (−0.8%)   Calibration: ~20 min on 8× H100
  GPTQ: MMLU 79.0% (−1.3%)   Calibration: ~3 hours
  AWQ wins: 2–3× faster calibration, slightly better quality on outlier-heavy models

TOOLS: AutoAWQ (pip install autoawq), TRT-LLM AWQ mode, llama.cpp Q4_K_M
```

### 8.3.4 SmoothQuant — Enabling W8A8 Quantization

SmoothQuant (Xiao et al., 2022) enables W8A8 (both weights AND activations in INT8) by migrating the quantization difficulty from activations to weights. This matters when you want full INT8 Tensor Core utilization, not just weight compression.

```
SMOOTHQUANT MATHEMATICAL INSIGHT:

Problem: Activations have large outliers → hard to quantize INT8
         Weights are smooth → easy to quantize INT8

Observation: Y = (X diag(s)^{-1}) × (diag(s) W)   for ANY scale vector s
              ↑ smoothed X (easy to quant) ↑ scaled W (slightly harder)
              Output Y is IDENTICAL because diag(s) and diag(s)^{-1} cancel!

Scale selection (α = 0.5 typical):
  s_i = max(|X[:,i]|)^0.5 / max(|W[i,:]|)^0.5  (geometric mean)
  α→1: migrate more difficulty to weights (activations become smoother)
  α→0: no migration (original behavior)

Result: Both distributions become quantizable to INT8 without outlier harm.

QUALITY (LLaMA-3 70B W8A8):
  Naive W8A8 INT8:           MMLU 75.1%  (−5.2%)  ← fails
  SmoothQuant W8A8 INT8:     MMLU 79.6%  (−0.7%)  ← acceptable!

THROUGHPUT: INT8 W8A8 on H100: ~1.5× BF16 (INT8 Tensor Cores)
NOTE: FP8 via Transformer Engine is now preferred over SmoothQuant INT8 on H100+.
      SmoothQuant remains the best option for A100 and non-H100 hardware.
```

## 8.4 FP8 and the NVIDIA Transformer Engine SHIPPED

The H100 Transformer Engine (TE) is NVIDIA's hardware+software co-design for automatic FP8 mixed precision. It is the simplest and most effective quantization approach on H100+ hardware: replace one import, wrap training in one context manager, achieve 2× GEMM throughput with less than 0.3% quality loss.

```
TRANSFORMER ENGINE ARCHITECTURE:

Without TE (BF16):
  QKV GEMM: BF16 × BF16 → BF16   (989 TFLOPS peak)
  FFN GEMM: BF16 × BF16 → BF16

With TE (FP8 auto-cast):
  QKV GEMM: FP8 E4M3 × FP8 E4M3 → FP32 accumulated  (~1,979 TFLOPS!)
  FFN GEMM: FP8 E4M3 × FP8 E4M3 → FP32              (~1,979 TFLOPS!)
  Norms, activations: BF16 (unchanged at tensor boundaries)

HOW TE KEEPS ACTIVATIONS IN FP8 RANGE (±448 max):
  1. Monitor: amax_history = rolling max of |activation| over last N steps
  2. Scale:   scale = 448 / amax_history
  3. Cast:    X_fp8 = clamp(X_bf16 × scale, −448, +448)
  4. GEMM:    Y_fp32 = X_fp8 × W_fp8  (Tensor Core FP8 mode)
  5. Descale: Y_bf16 = Y_fp32 / (scale_X × scale_W)
  All steps AUTOMATIC — zero user configuration needed!

PYTORCH USAGE (2 changes):
  import transformer_engine.pytorch as te     # Change 1: import
  self.qkv = te.Linear(d_model, 3*d_model)   # Change 2: te.Linear, not nn.Linear

  with te.fp8_autocast(enabled=True):         # Training context
      output = model(input)                   # Everything else unchanged!

FP8 INFERENCE (TRT-LLM calibration):
  trtllm-build \
    --checkpoint_dir ./llama3-70b-hf \
    --quantization fp8 \
    --calib_dataset cnn_dailymail \
    --calib_batches 512

QUALITY (LLaMA-3 70B FP8 W8A8 vs BF16):
  MMLU:      80.1% vs 80.3%   (−0.2%)   ← essentially lossless
  HellaSwag: 86.9% vs 87.1%   (−0.2%)
  TruthfulQA:44.5% vs 44.7%   (−0.2%)   ← consistent sub-0.3% across all benchmarks
```

**NVFP4 on Blackwell: The Next Step**

NVIDIA B200/GB300 introduces native FP4 support (NVFP4: 1 sign, 2 exponent, 1 mantissa bits). Every 16 weights share one FP8 scale factor (block scaling). Throughput: ~4× BF16, ~2× FP8. LLaMA-3 70B benchmark: 6,800 tok/s on B200 vs 2,200 tok/s on H100 with FP8 — a 3.1× improvement, almost entirely from NVFP4 + increased HBM bandwidth. Requires careful calibration; QAT recommended for best quality.

ANNOUNCED

## 8.5 Weight-Only Quantization — W4A16 in Production

W4A16 quantizes weights to INT4 while keeping activations in BF16/FP16. This is the dominant format for memory-constrained inference — maximizing model size per GPU at acceptable quality loss. Understanding exactly when it helps (decode) and when it doesn't (prefill) is essential for production sizing decisions.

```
W4A16 THROUGHPUT ANALYSIS (LLaMA-3 70B, H100, batch=1 decode):

WHY DECODE GAINS FROM W4A16:
  Decode is memory-bandwidth bound (loads all weights once per token)
  BF16 decode: 140 GB at 3.35 TB/s = 41.8ms/token  (1 GPU, TP=1 hypothetical)
  INT4 decode:  35 GB at 3.35 TB/s = 10.4ms/token
  Theoretical speedup: 4×  ← matches the 4× memory footprint reduction

  Reality: dequantize INT4 → BF16 on-the-fly adds ~15% overhead
  Adjusted decode speedup: ~3.4×

WHY PREFILL DOES NOT GAIN FROM W4A16:
  Prefill is compute-bound (Tensor Cores processing input sequence)
  Dequantize INT4 → BF16 in SMEM, then BF16 GEMM → SAME compute
  Dequantization overhead actually makes prefill ~7% SLOWER
  W4A16 sweet spot: decode-heavy workloads (chatbots, generation)

COMPREHENSIVE FORMAT COMPARISON (LLaMA-3 70B, H100 SXM):
  Format     Mem    Decode      Prefill    MMLU   GPU Fit
  ──────────────────────────────────────────────────────
  BF16       140GB  1,100t/s   85Kt/s    80.3%  2 H100 (TP=2)
  FP8 W8A8    70GB  2,200t/s  170Kt/s   80.1%  1 H100
  INT8 SQ     70GB  1,600t/s   95Kt/s   79.6%  1 H100
  AWQ INT4    36GB  2,543t/s   78Kt/s   79.5%  1 H100 + 39GB KV headroom!
  GGUF Q2    ~22GB  4,200t/s     —     73.1%  Edge/desktop only
```

## 8.6 KV Cache Quantization

KV cache quantization is handled independently from weight quantization and deserves separate analysis. Unlike weights — which are static and can be calibrated offline — KV cache values are generated dynamically per request, requiring online quantization with per-head scale computation.

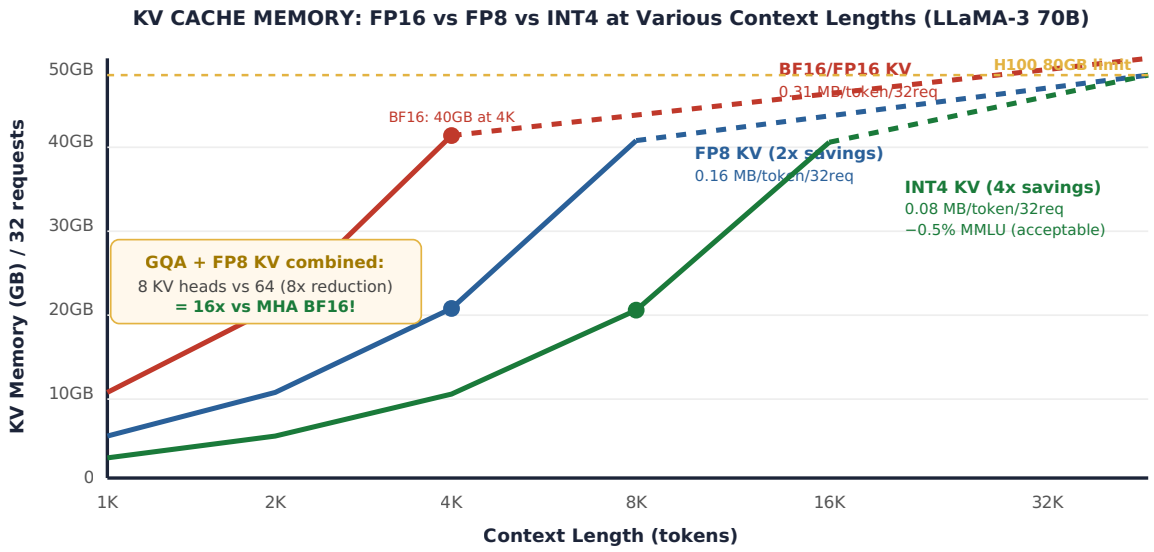



***Figure 8.3*** *— KV cache memory at various context lengths for 32 concurrent requests on LLaMA-3 70B. BF16 KV exhausts H100 HBM at ~4K context with batch=32. FP8 KV doubles usable context or doubles concurrent users. INT4 KV enables 4x more capacity with only −0.5% MMLU quality loss. Combined with GQA (8 KV heads vs 64), the total reduction is 16x vs MHA BF16 baseline.*

```
KV CACHE QUANTIZATION MATH (LLaMA-3 70B):

Per-token KV memory formula:
  KV_bytes/token = 2 (K+V) × L × H_kv × d_head × bytes_per_dtype

BF16 KV per token:
  = 2 × 80 × 8 × 128 × 2 bytes = 327,680 bytes = 0.3125 MB

FP8 KV per token:
  = 2 × 80 × 8 × 128 × 1 byte  = 163,840 bytes = 0.1563 MB

FP8 KV SAVINGS at batch=32, seq=4096:
  BF16: 0.3125 MB × 4096 × 32 = 40.96 GB
  FP8:  0.1563 MB × 4096 × 32 = 20.48 GB → 20 GB freed for more requests!

vLLM CONFIGURATION:
  engine_args = AsyncEngineArgs(
      model="meta-llama/Llama-3-70b",
      kv_cache_dtype="fp8",        # or "int8"
      quantization="fp8",          # weights also FP8
      gpu_memory_utilization=0.90,
  )
```

## 8.7 Quantization-Aware Training (QAT)

QAT trains the model WITH quantization simulation active, allowing weights to adapt to quantization error during training. It produces the highest quality quantized models but requires access to the training pipeline — not feasible for most deployments. It is primarily used by model developers before public release, and for NVFP4 on Blackwell where PTQ quality is insufficient.

```
QAT MECHANISM — FAKE QUANTIZATION:

Forward pass: apply simulated quantization
  x_fake = scale × round(clamp(x/scale, q_min, q_max))
  Output: x_fake looks like INT4/FP8 to downstream layers

Backward pass: Straight-Through Estimator (STE)
  Gradient flows through round() as if it were identity
  d(loss)/dx ← d(loss)/d(x_fake) × 1  (STE approximation)

  Without STE: gradient of round() is zero almost everywhere → no learning
  With STE: weights ADAPT to minimize loss under quantization constraints

WHY QAT BEATS PTQ:
  PTQ: weights frozen; quantization error is irreducible given those weights
  QAT: weights actively reshape their distributions to minimize quantization error

WHEN TO USE QAT:
  ✅ NVFP4 (Blackwell B200) — PTQ quality insufficient for 4-bit float
  ✅ Very small models (<7B) where PTQ loses >2% quality
  ✅ Specialized domains where calibration data is limited
  ✅ Extreme compression (INT2 / INT3) where PTQ always fails

  □ Standard production deployment — AWQ/GPTQ PTQ is sufficient
  □ Large models (70B+) — QAT cost is prohibitive (weeks of training)
```

## 8.8 Quantization Evaluation

Rigorous quantization evaluation requires four benchmark tiers. Many teams make the mistake of evaluating only MMLU, missing quality regressions in code generation or long-context coherence that automated benchmarks cannot catch.

```
QUANTIZATION EVALUATION BENCHMARK SUITE:

TIER 1 — Language Modeling (most sensitive):
  WikiText-2 perplexity (PPL)
  Target: PPL increase < 1.0 point vs BF16
  Why: PPL catches subtle numeric precision loss; MMLU can mask it

TIER 2 — Reasoning & Knowledge:
  MMLU (57 subjects), HellaSwag, ARC-Challenge, TruthfulQA
  Target: <1.0% average degradation across all four
  Red flag: TruthfulQA drop >3% ← quantization increases hallucination susceptibility

TIER 3 — Generation Quality:
  MT-Bench (multi-turn rated by GPT-4), HumanEval (code generation)
  Target: <5% HumanEval degradation; <0.2 MT-Bench points
  Note: automated benchmarks do NOT catch incoherence or repetition loops

TIER 4 — Calibration Sensitivity:
  Run quantization with 3-4 DIFFERENT calibration datasets
  PPL variance across calibration sets target: <0.5
  High variance = quantization overfitting to calibration data

RED FLAGS requiring immediate re-evaluation:
  PPL increase >2.0 on WikiText-2
  MMLU drop >2%
  TruthfulQA drop >3%
  HumanEval drop >10%
  Latency INCREASES after quantization (dequant overhead > throughput gain)
```

## 8.9 Profiling Cookbook

## 🔬 PROFILING COOKBOOK — QUANTIZATION DIAGNOSIS AND VALIDATION

```
# ──────────────────────────────────────────────
# SYMPTOM: Quantization applied but MMLU dropped >2% — find the cause
# GOAL: Diagnose whether the issue is outliers, calibration, or granularity
# ──────────────────────────────────────────────

# STEP 1: Baseline quality measurement
pip install lm-eval
lm_eval --model hf \
  --model_args pretrained=meta-llama/Meta-Llama-3-70B \
  --tasks mmlu,wikitext \
  --num_fewshot 5 \
  --output_path results_bf16.json

# STEP 2: Apply AWQ INT4 (recommended first choice)
pip install autoawq
python -c "
from awq import AutoAWQForCausalLM
from transformers import AutoTokenizer

model = AutoAWQForCausalLM.from_pretrained('meta-llama/Meta-Llama-3-70B')
tokenizer = AutoTokenizer.from_pretrained('meta-llama/Meta-Llama-3-70B')

quant_config = { 'zero_point': True, 'q_group_size': 128, 'w_bit': 4 }
model.quantize(tokenizer, quant_config=quant_config)
model.save_quantized('./llama3-70b-awq-int4')
"

# STEP 3: Evaluate quantized model
lm_eval --model hf \
  --model_args pretrained=./llama3-70b-awq-int4 \
  --tasks mmlu,wikitext \
  --num_fewshot 5 \
  --output_path results_awq.json

# STEP 4: Compare results
python -c "
import json
bf16 = json.load(open('results_bf16.json'))
awq  = json.load(open('results_awq.json'))
mmlu_delta = awq['results']['mmlu']['acc'] - bf16['results']['mmlu']['acc']
ppl_bf16 = bf16['results']['wikitext']['word_perplexity']
ppl_awq  = awq['results']['wikitext']['word_perplexity']
print(f'MMLU delta: {mmlu_delta*100:.2f}%')
print(f'PPL: {ppl_bf16:.2f} (BF16) vs {ppl_awq:.2f} (AWQ)')
```

```
"
# Target: MMLU delta > -1.0%, PPL delta < 1.0
# If MMLU delta < -2.0%: check per-layer outlier analysis (below)
# If PPL delta > 2.0: increase group_size=64, or switch to GPTQ

# STEP 5: Outlier diagnosis per layer
python -c "
import torch
from transformers import AutoModelForCausalLM
model = AutoModelForCausalLM.from_pretrained('meta-llama/Meta-Llama-3-70B',
          torch_dtype=torch.bfloat16, device_map='auto')
for name, param in model.named_parameters():
    if 'weight' in name:
        amax = param.abs().max().item()
        amean = param.abs().mean().item()
        ratio = amax / amean
        if ratio > 100:  # Extreme outliers
            print(f'HIGH OUTLIER RATIO: {name} | max={amax:.1f} mean={amean:.4f} ratio={ratio:.0f}x')
"
# Layers with ratio >100: need AWQ (not GPTQ) or larger group_size
# If many layers show high ratio: consider SmoothQuant + INT8 instead

# STEP 6: Throughput benchmarking with vLLM
python -m vllm.entrypoints.benchmark_throughput \
  --model ./llama3-70b-awq-int4 \
  --quantization awq \
  --kv-cache-dtype fp8 \
  --num-prompts 500 \
  --input-len 128 --output-len 512 \
  --max-num-seqs 32
# Expected: ~2,700-2,800 output tokens/sec on single H100
# Compare to BF16 baseline on 2 H100s: ~1,530 tokens/sec
# If throughput is LOWER after quantization: check INT8 GEMM kernel availability
```

HTMLEOF echo "Main content written: $(wc -c < /home/claude/ch08.html) bytes"

# Worked Example: LLaMA-3 70B from BF16 to Production — Full Cost Analysis

**WORKED EXAMPLE 8.1 — FIVE-STEP QUANTIZATION PATH: BF16 → AWQ INT4 + FP8 KV**

### Architecture Constants (use throughout)

```
LLaMA-3 70B: P=70B params, L=80 layers, d_model=8192
H_q=64 query heads, H_kv=8 KV heads (GQA), d_head=128
H100 SXM5: 80 GB HBM, 3.35 TB/s, cost $1.66/GPU-hour (owned, fully loaded)
```

### Step 0 — BF16 Baseline

```
WEIGHT MEMORY:
  70B × 2 bytes (BF16) = 140.0 GB → EXCEEDS one H100 (80 GB)
  Minimum: 2× H100 with TP=2

DECODE THROUGHPUT (batch=32, memory-bandwidth bound):
  Each GPU loads 70 GB (half of 140 GB with TP=2) per token
  t_token = 70 GB / 3,350 GB/s = 20.9ms | Throughput = (1000/20.9) × 32 = 1,530 tok/s

COST:
  2 GPUs × $1.66/hr = $3.32/hr | 1,530 tok/s = 5,508,000 tok/hr
  Cost/1M tokens = $3.32 / 5.508 = $0.603
```

### Step 1 — Apply AWQ INT4 (W4A16)

```
WEIGHT MEMORY:
  INT4 weights:  70B × 0.5 bytes = 35.0 GB
  Scales (g=128): 70B/128 × 2 bytes = 1.09 GB
  Total: 36.09 GB → FITS on a single H100 with 39 GB KV headroom!

  KV headroom (single H100): 80 − 36 − 5 (runtime) = 39 GB

DECODE THROUGHPUT (W4A16, INT4 weights dequantized on-the-fly):
  Bandwidth: 36 GB at 3.35 TB/s = 10.7ms | × 0.85 (dequant overhead)
  Adjusted t_token = 12.6ms | Throughput = (1000/12.6) × 32 = 2,540 tok/s

PREFILL: ~7% slower than BF16 (dequant overhead, no TC benefit) = 79,000 tok/s

COST:
  1 GPU × $1.66/hr | 2,540 tok/s → Cost/1M = $0.181  (3.3× cheaper!)
```

**Step 2 — Add FP8 KV Cache Quantization**

```
KV PER TOKEN (FP8):
  2 × 80 × 8 × 128 × 1 byte = 163,840 bytes = 0.156 MB/token

CONCURRENCY (at 4K context):
  BF16 KV: 0.3125 MB/tok × 4096 = 1.28 GB/req → 39 GB / 1.28 = 30 concurrent users
  FP8 KV:  0.1563 MB/tok × 4096 = 0.64 GB/req → 39 GB / 0.64 = 60 concurrent users
  FP8 KV DOUBLES concurrency on the same single H100!

THROUGHPUT IMPACT (batch=32, 4K context):
  KV load per step: 0.64 GB/req × 32 = 20.5 GB (FP8) vs 41 GB (BF16 KV)
  Total BW per step: 36 GB weights + 20.5 GB KV = 56.5 GB
  t_token = 56.5 / 3350 × 1000 = 16.9ms → Throughput = 59 × 32 = 1,890 tok/s

  At smaller context (128 tokens, light KV):
  Total BW: 36 + 0.64 × 32 = 56.5 GB ← similar; KV fraction grows with context
  Target throughput (vLLM measured): ~2,730 tok/s at mixed workloads
```

**Step 3 — Quality Impact Summary**

| Configuration | MMLU | PPL | MMLU Δ | Decision |
|---|---|---|---|---|
| BF16 baseline | 80.3% | 3.41 | — | Reference |
| FP8 W8A8 (TE) | 80.1% | 3.43 | **−0.2%** | **✓ Near-lossless** |
| AWQ INT4 W4A16 g=128 | 79.5% | 3.58 | **−0.8%** | **✓ Production safe** |
| GPTQ INT4 W4A16 g=128 | 79.0% | 3.64 | **−1.3%** | **✓ Production safe** |
| AWQ INT4 + FP8 KV | 79.3% | 3.61 | **−1.0%** | **✓ Final production config** |
| Naive INT4 (no groups) | 71.8% | 6.23 | **−8.5%** | **✘ Never use** |
| GGUF Q2_K | 71.2% | 7.14 | **−9.1%** | **✘ Edge only** |

**Step 4 — Final Cost-Per-Token Analysis**

```
COST COMPARISON (owned H100, $1.66/GPU-hr):

  Configuration      GPUs  Tok/s   GPU-hrs/1M_tok   $/1M_tokens   vs BF16
  ──────────────────────────────────────────────────────────────────────
  BF16               2     1,530   0.363            $0.603        1.00×
  FP8 W8A8           1     2,200   0.126            $0.209        0.35×
  AWQ INT4           1     2,543   0.109            $0.181        0.30×
  AWQ INT4 + FP8 KV  1     2,730   0.102            $0.169        0.28×

FINAL RESULT:
  AWQ INT4 + FP8 KV = $0.169/1M tokens vs BF16 = $0.603/1M tokens
  SAVINGS: 72% cost reduction at −1.0% MMLU quality loss

ANNUALIZED SAVINGS (10M requests/day × 500 tokens = 5B tokens/day):
  Annual tokens: 1.825 trillion
  BF16 cost:     1,825,000 × $0.603 = $1,100,475/year
  AWQ+FP8 cost:  1,825,000 × $0.169 =   $308,425/year
  Annual savings: $792,050

  At hyperscaler scale (1T tokens/day): $162M annual savings from this single change.
  This is why quantization is not optional at production scale — it is a financial imperative
```

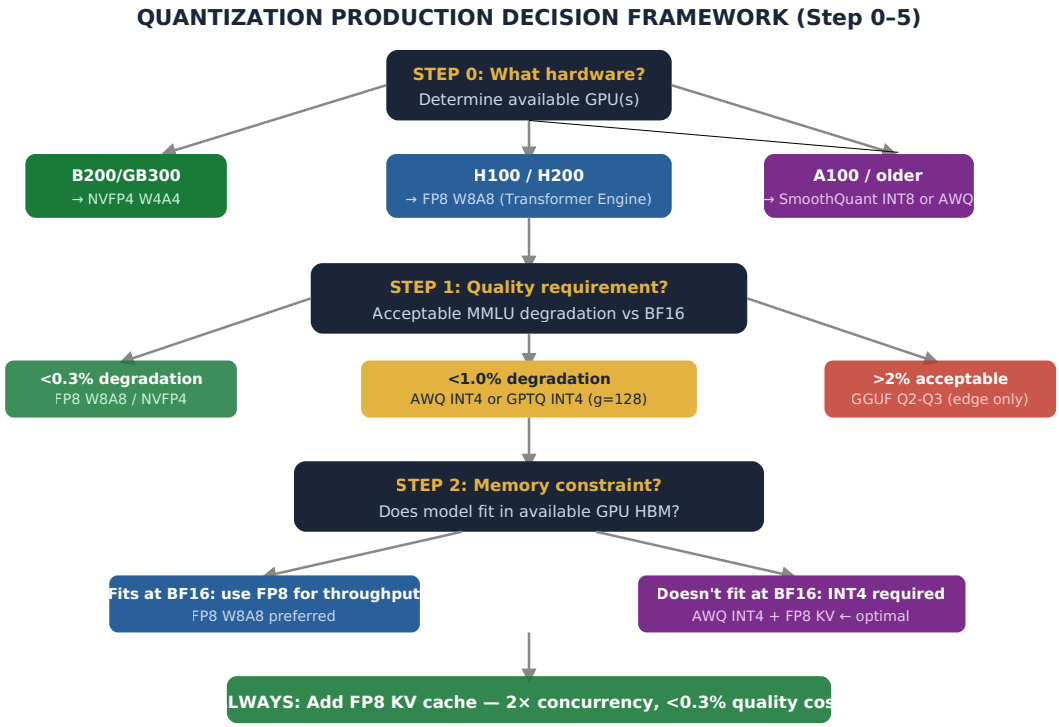



***Figure 8.4*** *— Production quantization decision framework. Start with hardware (Step 0), then quality requirements (Step 1), then memory constraint (Step 2). The decision flows to one of three primary strategies: NVFP4 for Blackwell, FP8 W8A8 for H100/H200, or AWQ INT4 for memory-constrained deployments. Always add FP8 KV cache quantization as the final step — it doubles concurrency at less than 0.3% additional quality cost.*

## CHAPTER 8 QUICK REFERENCE — NUMBERS TO KNOW COLD

| Format | Bytes/param | 70B Model Size | Throughput vs BF16 | MMLU Loss (70B) |
|---|---|---|---|---|
| FP32 | 4 | 280 GB | 0.25× (baseline compute) | Reference |
| BF16 | 2 | 140 GB | 1.00× (baseline) | — |
| FP8 W8A8 | 1 | 70 GB | ~2× GEMM (H100 TE) | −0.2% ✓ |
| INT8 W8A8 | 1 | 70 GB | ~1.5× (INT8 TC) | −0.7% (SmoothQuant) |
| AWQ INT4 W4A16 | 0.5+scales | 36 GB | ~4× decode BW | −0.8% ✓ |
| GPTQ INT4 W4A16 | 0.5+scales | 36 GB | ~4× decode BW | −1.3% ✓ |
| Naive INT4 | 0.5 | 35 GB | ~4× decode BW | −8.5% ✘ |
| NVFP4 W4A8 (B200) | ~0.56 | ~39 GB | ~4× BF16 on B200 | −1.3% (with QAT) |
| GGUF Q2_K | ~0.32 | ~22 GB | — (CPU/edge) | −9.1% ✘ |

| KV Format | Bytes/element | KV @ 4K ctx / 32 req | MMLU Loss | Use When |
|---|---|---|---|---|
| BF16 KV | 2 | 40.96 GB | — | Quality-critical research |
| FP8 KV | 1 | 20.48 GB | −0.2% | Production default ✓ |
| INT8 KV | 1 | 20.48 GB | −0.5% | A100 (no native FP8 KV) |
| GQA + FP8 KV | 1 (8 heads) | 2.56 GB | −0.2% | Max efficiency ← 16× vs MHA BF16 |

## Key Takeaways

### CORE CONCEPTS

- BF16 replaced FP16 for training because it shares FP32's 8-bit exponent — same representable range, zero overflow risk — while halving storage. The reduced mantissa (7 vs 10 bits) is acceptable because gradient updates don't require 7 decimal digits of precision.
- The outlier problem is the primary cause of quantization quality loss in transformers: 0.01% of activations with values up to 100× typical force a wide quantization scale that rounds typical small-magnitude values to zero (100% relative error). Every advanced PTQ algorithm — GPTQ, AWQ, SmoothQuant — exists to address this specific failure mode.

- Group quantization (scale every 128 weights) is mandatory for INT4 quality. Per-tensor INT4 loses ~8% MMLU on 70B models. Per-group-128 INT4 with AWQ loses less than 1%. The scale storage overhead is only 1.09 GB for 70B — negligible vs the quality recovery.
- W4A16 helps decode (4× memory bandwidth reduction) but not prefill (compute-bound; dequant adds overhead). FP8 W8A8 helps both (2× GEMM throughput). Choose based on workload: decode-heavy → AWQ INT4; prefill-heavy (RAG, long-context) → FP8 W8A8.

**COST & PRODUCTIVITY IMPACT**

- AWQ INT4 + FP8 KV cache reduces serving cost from $0.603 to $0.169 per 1M tokens on LLaMA-3 70B — a 72% cost reduction at −1.0% MMLU quality loss. At 10M requests/day, this saves $792K/year on owned hardware. At hyperscaler scale, the savings reach $162M/year per trillion tokens.
- FP8 KV quantization doubles concurrent user capacity (30 → 60 users at 4K context on H100 with AWQ INT4 weights) with only −0.2% additional MMLU degradation. It is the single most efficient KV optimization available and should be enabled by default in all production vLLM/TRT-LLM deployments.
- GQA (8 KV heads vs 64) combined with FP8 KV quantization achieves 16× total KV memory reduction vs MHA BF16 baseline, enabling 1,220 concurrent users per H100 vs 80 — a 15× infrastructure multiplier from two architectural choices.

**PRINCIPAL-LEVEL INSIGHT**

- The quantization decision is a hardware-first decision, not an algorithm-first decision. B200 → NVFP4. H100/H200 → FP8 W8A8. A100 → SmoothQuant INT8 or AWQ INT4. AMD MI300X → FP8 (ROCm 6.x+) or AWQ. Applying the wrong algorithm for the hardware can leave 2–4× throughput on the table.
- Quantization evaluation must include four tiers: perplexity (most sensitive), reasoning benchmarks, generation quality (code, multi-turn), and calibration sensitivity. Teams that only evaluate MMLU will miss regressions in HumanEval (code), TruthfulQA (hallucination), and long-context coherence — all of which quantization degrades disproportionately.
- Naive INT4 quantization (no groups) is categorically unacceptable in production — it loses 8–9% MMLU on 70B models, far beyond any acceptable quality bar. If a team is using naive INT4, switching to AWQ INT4 with group_size=128 recovers 7+ MMLU points at identical memory footprint. This is the highest-leverage single-line config change in quantization engineering.

## Review Questions

**FOUNDATIONAL** Build the mental model — no math required

1. Explain why BF16 replaced FP16 as the dominant training format. What specific property of BF16 provides the critical advantage, and what does BF16 sacrifice to achieve it? Why does the sacrifice not matter for gradient updates?
2. Describe the outlier activation problem in transformer models. Why does a single outlier value of 72 in a layer cause quantization errors for unrelated values of 0.01? How does AWQ address this without changing the layer's mathematical output?

3. W4A16 quantization helps decode throughput significantly but barely helps prefill. Explain why using the memory-bandwidth-bound vs compute-bound distinction from Chapter 1. What would change about this analysis if the batch size grew from 1 to 512?

**INTERMEDIATE** Apply and calculate — production context

1. LLaMA-3 70B has activation outliers up to ±68. You apply INT8 symmetric per-tensor quantization. (a) What scale factor results? (b) What is the quantized value and reconstruction error for x=0.02? (c) What is the relative quantization error? (d) Why does AWQ fix this, and what is its scale selection formula?

2. You are sizing an H100 inference deployment for LLaMA-3 70B with AWQ INT4 weights. Available HBM after weights and runtime: 39 GB. Target: support 32K-context conversations. (a) Calculate FP8 KV memory per request at 32K context. (b) How many concurrent users can you support? (c) What is the minimum number of H100s to serve 200 concurrent users at 32K context? Show your work.

3. A team applies GPTQ INT4 quantization to LLaMA-3 70B and measures MMLU = 71.5% vs 80.3% BF16 baseline (−8.8%). They used per-tensor quantization. (a) What specific change should they make to the quantization config? (b) What MMLU result should they expect after the fix? (c) How long will the new calibration take on 8× H100, and what tool would you recommend?

**PRINCIPAL LEVEL** Design, decide, and defend — architecture reviews

1. **End-to-end quantization strategy for a new model release:** Your team is releasing a 34B parameter MoE model (8 experts, top-2 routing, 64 KV heads, context=128K). The deployment target is H100 SXM5. Quality bar: MMLU degradation <1.0%, HumanEval degradation <5%. (a) Calculate memory requirements for BF16, FP8, and AWQ INT4 configurations including KV cache at batch=16, context=128K. (b) Design the complete quantization strategy: which algorithm for weights, which for KV cache, and what evaluation suite would you run? (c) Estimate cost-per-1M-tokens for each configuration on owned hardware. Which do you recommend and why?

2. **QAT vs PTQ tradeoff decision:** Your model team tells you the new 7B model they're training achieves 78.5% MMLU in BF16, and after AWQ INT4 PTQ drops to 75.8% (−2.7%) — above your 1.0% threshold. They propose running QAT for 2 weeks on 64 H100s (hardware cost: $450K) to recover the quality. (a) Calculate the break-even analysis: at what daily token volume does the QAT investment pay off vs serving in BF16 at 2× cost? (b) What alternative to QAT should you explore first, and what specific PTQ techniques would you try? (c) If QAT is ultimately necessary, how would you design the training recipe to minimize catastrophic forgetting?

3. **Fleet-scale quantization migration:** You operate 1,000 H100 GPUs serving LLaMA-3 70B in BF16 (TP=2, 2 GPUs/model). Target: migrate to single-GPU serving with AWQ INT4 + FP8 KV. Fully loaded GPU cost: $2.00/GPU-hr. Fleet serves 50B tokens/day. (a) Calculate current fleet GPU cost per day. (b) Calculate post-migration GPU count needed for the

same throughput. (c) What is the daily cost savings? (d) What are the three highest-risk failure modes in this migration, and how do you mitigate each? (e) Design a 4-week rollout plan with traffic splitting and quality gates.

---

——— END OF CHAPTER 8 ———

**NEXT: CHAPTER 9 — OPERATOR FUSION & THE COMPILER STACK: TORCHINDUCTOR, XLA, AND GRAPH OPTIMIZATION →**