# Post-Training Quantization Correction

From memory constraints to hardware-aware PTQ and lightweight correction

AnhND · Edge AI Summer School

# Where we are going

1. **Why quantize?**
   deployment memory wall
2. **What is quantization?**
   grid, scale, two errors
3. **Hardware decides what helps**
   memory- vs compute-bound
4. **QAT vs PTQ**
   how we reach the format
5. **The PTQ toolbox**
   isolate / flatten / correct
6. **Post-correction: CLC**
   repair residual on the lattice
7. **Synthesis**
   a deployment decision tree

Why quantize? → What is it? → **Hardware** decides what is useful → **QAT/PTQ** decides how we get there → **correction** repairs what is left → deploy.

Part 1

# Why are we quantizing?

The deployment problem, before any definitions

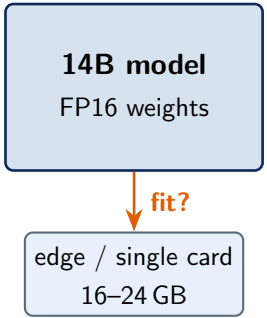
# Suppose we want to deploy a 14B model...

**The back-of-the-envelope question**

**14 billion** parameters

stored in **FP16** (or BF16) (16 bits each)

**How much memory?**



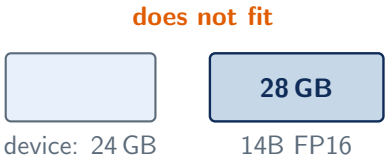

Take ten seconds. Multiply in your head before the next slide.

**Why this number matters**

Parameters × bytes-per-parameter is already a useful deployment estimate. Before any algorithm, this single product decides whether the model *fits at all*.

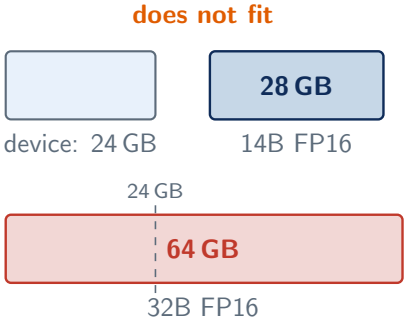
# $14\text{B} \times 16$ **bits** $\approx 28$ **GB**

$$\underbrace{14 \times 10^9}_{\text{params}} \times \underbrace{2 \text{ bytes}}_{\text{FP16}} = 28 \text{ GB}$$



# 14B $\times$ 16 bits $\approx$ 28 GB

$$\underbrace{14 \times 10^9}_{\text{params}} \times \underbrace{2 \text{ bytes}}_{\text{FP16}} = 28 \text{ GB}$$




**Scale it up**

A **32B** model in FP16:

$$32 \times 10^9 \times 2 \text{ bytes} \approx \textbf{64 GB}$$

And that is *weights only* — activations and the KV cache still need room on top.

# Keep all 14B parameters — but store each differently?

**FP16**
2 B/param $\rightarrow$ 28 GB

The key idea: **change the representation**, not the model structure. Every parameter stays — we simply spend fewer bits to store each one.

**. . . but this definition is incomplete**

"Fewer bits" explains the *mechanism*, not *why* or *when* it helps. That is the rest of the lecture.

Transition: that is quantization — but the useful question is what the **hardware** can actually exploit.

# Keep all 14B parameters — but store each differently?

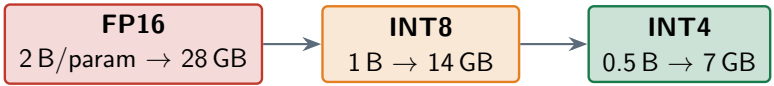

The key idea: **change the representation**, not the model structure. Every parameter stays — we simply spend fewer bits to store each one.

**. . . but this definition is incomplete**

"Fewer bits" explains the *mechanism*, not *why* or *when* it helps. That is the rest of the lecture.

Transition: that is quantization — but the useful question is what the **hardware** can actually exploit.

# Keep all 14B parameters — but store each differently?




The key idea: **change the representation**, not the model structure. Every parameter stays — we simply spend fewer bits to store each one.

**. . . but this definition is incomplete**

"Fewer bits" explains the *mechanism*, not *why* or *when* it helps. That is the rest of the lecture.

Transition: that is quantization — but the useful question is what the **hardware** can actually exploit.

# Keep all 14B parameters — but store each differently?


The key idea: **change the representation**, not the model structure. Every parameter stays — we simply spend fewer bits to store each one.

**. . . but this definition is incomplete**

"Fewer bits" explains the *mechanism*, not *why* or *when* it helps. That is the rest of the lecture.

Transition: that is quantization — but the useful question is what the **hardware** can actually exploit.

Part 2

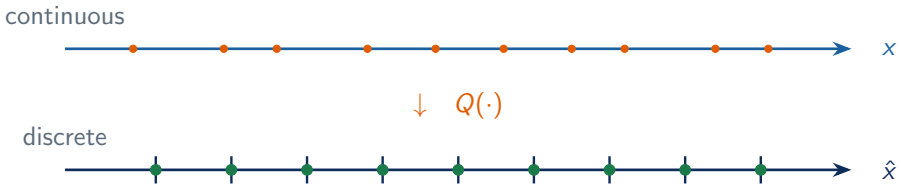
# Quantization fundamentals

Grid, scale, zero-point, and the two errors

# What is quantization?

**Working definition**

Represent a high-precision value using one value drawn from a **smaller discrete set**.

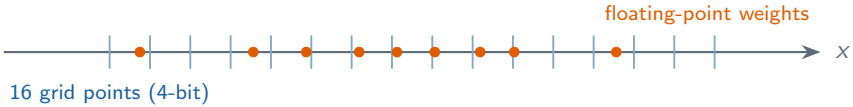



**Keep the definition general**

The discrete levels do **not** have to be equally spaced, and one code does **not** have to represent only one scalar. Every value is snapped to its nearest allowed level; the art is choosing *where* those levels sit — the next three slides map out that space of choices.

# From FP values to a discrete grid

A **4-bit** code has $2^4 = 16$ available levels. Overlay those 16 grid points on a spread of floating-point weights:

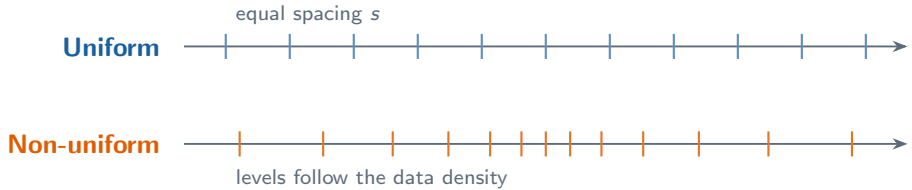



More bits $\Rightarrow$ more grid points $\Rightarrow$ finer approximation. Fewer bits $\Rightarrow$ coarser grid.

Notice the weights cluster near zero while the grid is spread evenly — the hint that an *evenly* spaced grid may not be the best fit. We take up that question next.

# Uniform vs. non-uniform quantization

The grid on the last slide was evenly spaced — but it need not be.




**Uniform**
$\hat{x} = s\,(q - z)$

**Non-uniform**
$\hat{x} = C[q]$ (level read from a table)

**Uniform grids are easier to execute; non-uniform grids can match the data distribution better.** Granularity and level geometry are *different* choices — per-group quantization can still use a uniform grid inside each group, $\hat{x} = s_g(q - z_g)$.

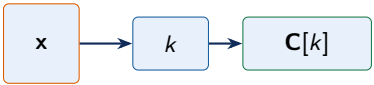
# Scalar vs. vector quantization

*Does one code have to represent one number?*

**Scalar**


one value → one code

**Vector (VQ)**



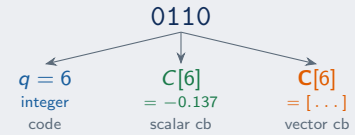

a block of values → one index → a codebook vector

Scalar: $x \to q$ Vector: $\mathbf{x} \to k, \quad \hat{\mathbf{x}} = \mathbf{C}[k]$

**Scalar quantization compresses individual values; VQ can exploit structure across several values at once.** We stay conceptual here — the point is only that "one code, one scalar" is a choice, not a law.

# Code, index, codebook, LUT — same bits, different jobs

**Same 4 bits: 0110**




| Term | Meaning |
|---|---|
| Code | generic low-bit stored representation |
| Integer code | numeric code used in affine dequant |
| Index | selects a codebook entry |
| Codebook | the reconstruction values / vectors |
| LUT | mechanism that performs the lookup |
| Packing | layout of codes in bytes/words |

**Two ways to use a LUT**

| | | |
|---|---|---|
| **Codebook lookup** | $k \to C[k] \to \times$ | decode, *then* compute |
| **Compute LUT** | $T[k]{=}\mathbf{C}[k]^\top \mathbf{x}$ | the lookup *is* the compute |

**An index is the representation; a LUT decodes it or is the compute.**

# Quantize → store an integer code

**Scope for the rest of the lecture**

We now narrow to **uniform scalar affine** quantization: it maps naturally to deployed INT formats and lets us study PTQ and hardware trade-offs cleanly. VQ / non-uniform / LUT approaches exist — we set them aside deliberately.

The mental model we carry throughout:

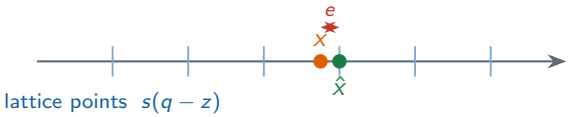

We do not store the float — we store the **integer code** $q$:

$$q \;=\; \mathcal{Q}(x) \;=\; \mathrm{clip}\Big(\Big\lfloor \tfrac{x}{s} \Big\rceil + z,\;\; 0,\;\; 2^b - 1\Big)$$

# Dequantize $\rightarrow$ approximate the original

To use the weight we map the code back to a real value:

$$\hat{x} \;=\; s\,(q - z), \qquad\qquad e \;=\; \hat{x} - x \quad \text{(the quantization error)}$$




Dequantization does *not* recover $x$ — it returns the nearest lattice point $\hat{x}$. The gap $e = \hat{x} - x$ is the price of using fewer bits, and reducing it is the goal of every method later.

# Three knobs: $(b, s, z)$

| Knob | Symbol | What it controls |
|---|---|---|
| Bit-width | $b$ | number of levels $= 2^b$ |
| Scale (step size) | $s > 0$ | grid spacing — resolution vs. range |
| Zero-point | $z \in \mathbb{Z}$ | offset; $z = 0$ gives a symmetric grid |

**$b$: levels**

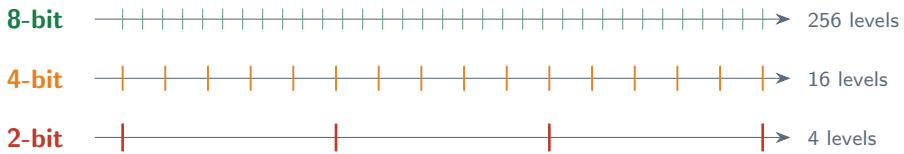
**$s$: spacing**


**$z$: offset**


Symmetric ($z = 0$) suits weights; asymmetric ($z \neq 0$) suits one-sided activations. We keep $z = 0$ for the pictures that follow.

# What happens when we reduce the number of bits?

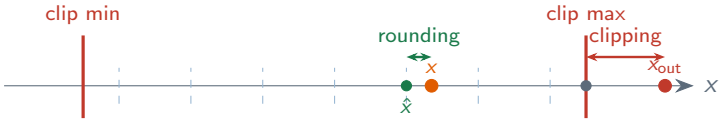
Same value range, three bit-widths — fewer bits = coarser grid.




**The compression–accuracy pull**

Each bit removed *halves* the levels: more compression, coarser approximation. So where should the levels go, and what breaks first?

# Two errors: rounding and clipping




**Rounding** — values *inside* the range snap to the nearest level. Bounded by $s/2$. Smaller $s$ shrinks it.

**Clipping**

Values *outside* the range saturate to the boundary. Error can be arbitrarily large. Wider range shrinks it.

# The fundamental range–resolution trade-off

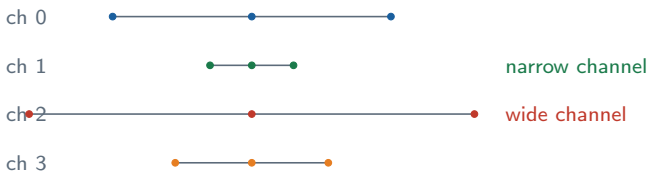
At fixed bit-width $b$, the two errors move in opposite directions.


$$\mathbb{E}[(x-\hat{x})^2] = \underbrace{\mathbb{E}\big[(x-\hat{x})^2\,\mathbf{1}_{x\in[\alpha,\beta]}\big]}_{\text{rounding}} + \underbrace{\mathbb{E}\big[(x-\hat{x})^2\,\mathbf{1}_{x\notin[\alpha,\beta]}\big]}_{\text{clipping}}$$

**Key insight.** The best range is *not* $[\min, \max]$: clipping a few outliers sharply cuts rounding error for the bulk — the root of every difficulty in Part IV.

# One scale for everything?

A single $(s, z)$ must cover the whole tensor — but channels differ.




| Granularity | Scales per layer | Accuracy | Overhead |
|---|---|---|---|
| Per-tensor | 1 | worst | none |
| Per-channel | $C$ (one per row) | good | minimal |
| Per-group | $C \times \lceil d/g \rceil$ | best | modest ($g$=128 typical) |

Finer granularity ⇒ smaller range per group ⇒ smaller step size ⇒ lower rounding error. Granularity is an axis *separate* from level geometry (frame 7): each group here still uses a uniform

# Finer granularity helps accuracy — but is it free?

**Per-group: fine local grids**


**The question that decides everything**

Can the target accelerator *execute* this efficiently?

**GPU:** per-group scales fold into the register-level unpack — essentially free.

**NPU / systolic:** per-group scaling disrupts the pipeline — often costly or unsupported.

Algorithmically finer always looks better; whether it *helps* is a hardware question — Part II.

Part 3

# Hardware-aware quantization

When do fewer bits actually help?

# The definition was incomplete

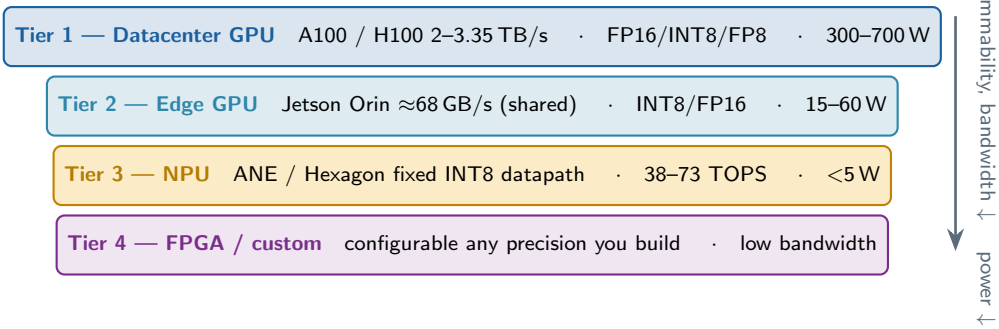
**What we said in Part I**

"Quantization = fewer bits per weight."


**The reframe that carries the whole lecture**

Quantization is a **hardware co-design problem**. Precision, scale granularity, memory layout, and the matmul kernel are all coupled to the chip you target. What flies on an A100 can be useless on a mobile NPU.

# The Edge AI hardware landscape




The key number: a **30–50× bandwidth gap** separates a datacenter GPU from an edge NPU.

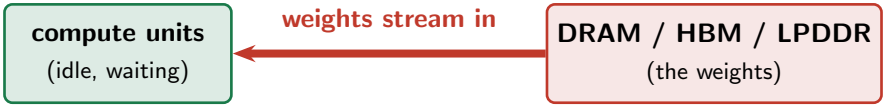
# What hardware constraints matter?

| **Memory capacity** | **Memory bandwidth** | **Native datapath** | **On-chip memory** | **Kernel / runtime** |
|---|---|---|---|---|
| does it fit at all? | how fast can weights move? | which precisions multiply? | cache / scratchpad size | is the format supported? |

**Read these as a checklist**

Every quantization decision in the rest of the lecture answers one of these five questions. **Bandwidth** (box 2) is the one that usually decides whether quantization helps at all — next.

# Memory is not only capacity — it is movement




limited bandwidth = the bottleneck

**The memory wall**

In LLM decoding, every generated token requires a full sweep of the weights through memory. The arithmetic units sit *idle* most of the pass, waiting for data. The workload is **memory-bound**: time is set by *bytes moved*, not FLOPs executed.

# A simple token-rate sanity check

$$\text{max tokens/sec} \approx \frac{\text{memory bandwidth}}{N \times \text{bytes/weight}}$$

**Llama-3 70B on an A100** (2 TB/s HBM):

**BF16 — 2 bytes**

$$\frac{140\,\text{GB}}{2000\,\text{GB/s}} = 70\,\text{ms}$$

$$\Rightarrow \approx 14 \text{ tokens/sec}$$

**INT4 — 0.5 bytes**

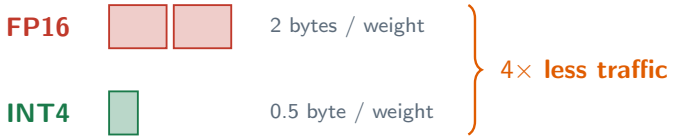
$$\frac{35\,\text{GB}}{2000\,\text{GB/s}} = 17.5\,\text{ms}$$

$$\Rightarrow \approx 57 \text{ tokens/sec}$$

4× fewer bytes ⇒ 4× the token rate — exactly.

# Why W4 helps even without INT4 multiplication

The gain is not faster arithmetic — it is **fewer bytes moved**.

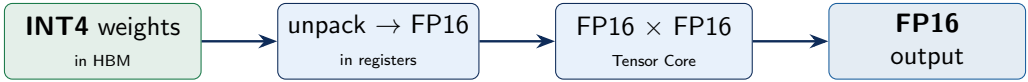



**The mechanism**

Even if the multiply still happens in FP16, moving the weights costs 4× less. On a memory-bound workload, *that* is the entire speedup — no low-precision arithmetic required.

# What does W4A16 mean?

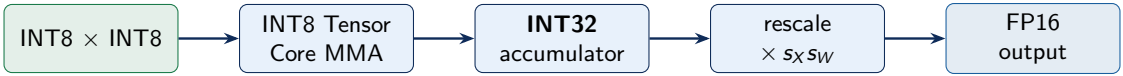
**Weights INT4, activations FP16** — weight-only quantization.




Activations stay FP16 throughout — *no* activation quantization error. The only error is the weight approximation. At 4-bit per-group, quality is largely preserved. **This is the dominant production scheme for LLMs.**

# What does W8A8 mean?

**Weights and activations INT8** $\Rightarrow$ a true integer matmul.



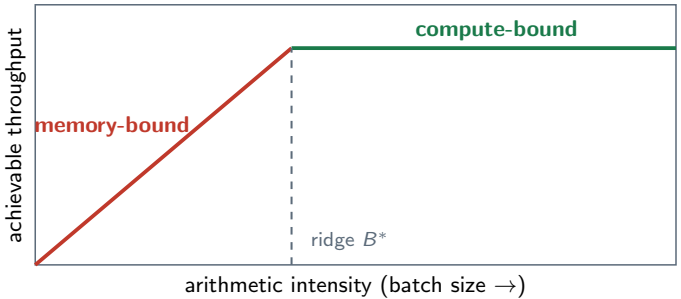

**Contrast with W4A16**

Here the *arithmetic* itself runs in INT8 — up to 2× the FP16 Tensor-Core throughput.

**But...**

Activations are now quantized too — and LLM activations have brutal outliers (Part IV).

# Memory-bound vs compute-bound




**Small batch (decode)**

Weights are moved once and barely reused $\Rightarrow$ **moving bytes dominates**.

**Large batch (prefill)**

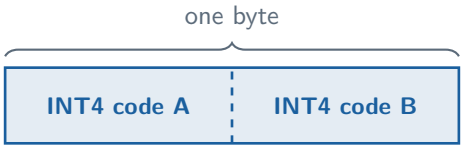
Weights reused across the batch $\Rightarrow$ arithmetic starts to dominate.

# So W4A16 vs W8A8 is not an accuracy-only choice

| Regime | What matters | Better choice |
|---|---|---|
| **Memory-bound** (small batch, decode) | fewer *weight bytes* moved | W4A16 ✓ |
| **Compute-bound** (large batch, prefill) | native low-precision *arithmetic* | W8A8 / FP8 ✓ |

The right bit-width is a function of the **execution regime**, not just tolerable accuracy loss. Ask "memory- or compute-bound?" *before* picking a scheme.

# Packed INT4 is not automatically executable INT4

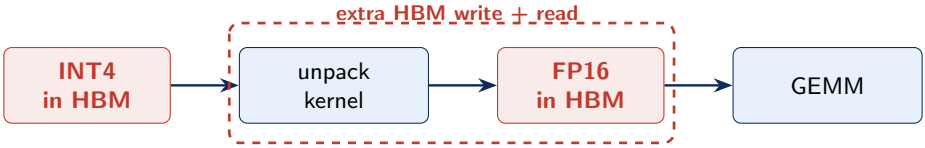



**The question**

Does the compute unit *directly* multiply this packed byte? Almost never. Two 4-bit codes share a byte; the datapath expects FP16 (or INT8) operands.

So a "4-bit checkpoint" is a *storage* format. Whether it runs fast depends entirely on the kernel that unpacks it — the next two slides. In the Part I framework, we now move from **representation** to **decode** and **compute**.

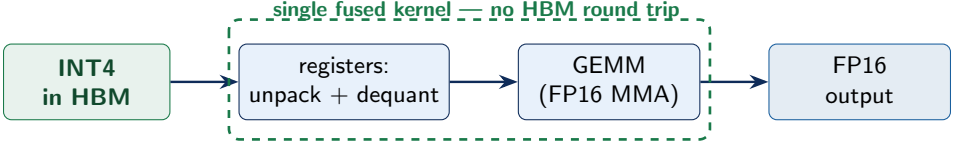
# The bad implementation




**Why this is slower than FP16**

Unpacking INT4 to FP16 and writing it back to HBM *adds* memory traffic: a pass to load INT4 and write FP16, then another to read FP16 for the matmul. Bandwidth cost goes **up**, not down.

# The good implementation: fuse it

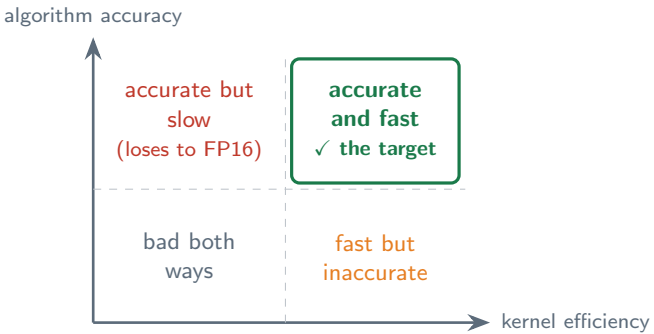



Load INT4 (4× cheaper), unpack to FP16 *in registers*, feed the Tensor Core, discard. The FP16 weights never touch HBM. HBM traffic drops 4×; the unpack costs ∼2 instructions per 8 weights — negligible.

**Where this sits in the four-stage model**

The register-level unpack+dequant *is* the **decode** stage; the Tensor Core is **compute**. Fusing them is exactly what makes a packed-INT4 *representation* executable on this **hardware**.

# Hardware takeaway: two orthogonal axes




A great algorithm with a bad kernel loses to FP16. Only the **top-right** quadrant is worth deploying. Part IV supplies the accuracy axis; this part supplied the efficiency axis.

## Part III

# QAT vs PTQ

How do we obtain the low-bit model?

# Quantization is one optimization problem

Part II fixed the *format* the hardware wants. Producing a good model in that format is **a single optimization problem** — everything else is a choice inside it.

$$\min_{\hat{\theta} \in \mathcal{G}(T,A,C)} \mathcal{L}(\hat{\theta}) \qquad \hat{\theta} \text{ must lie on the grid } \mathcal{G} \text{ fixed by } (T, A, C).$$

$T$ = transform (map before rounding) · $A$ = rate-allocation (bits, where) · $C$ = codebook (level shape)

detailed in 3 slides

**Two knobs decide everything downstream**

**Which loss** $\mathcal{L}$? — the true *task loss*, or a cheap *proxy* approximating a pretrained model.

**How much data**? — full training set, or a tiny *calibration* set (100–1000 samples).

Different answers ⇒ different paradigms — same problem.

# Two instances: QAT and PTQ

Both minimize $\mathcal{L}(\hat{\theta})$ on the same grid — they differ only in **objective** and **data budget**.

**QAT — optimize the task loss**

$$\min_{\hat{\theta} \in \mathcal{G}} \ \mathcal{L}_{\text{task}}(\hat{\theta}) \text{ on } D_{full}$$

Any init (even **from scratch**).
**Full dataset** $D$; gradients via STE.
Learn weights and grids.

**PTQ — approximate a good model**

$$\min_{\hat{\theta} \in \mathcal{G}} \ \left\| f_\theta - f_{\hat{\theta}} \right\| \text{ on } X$$

**Fixed pretrained** $\theta$.
**100–1000** calib $X$. samples; no backprop.
Only *rearranges* discrete levels.

**PTQ's proxy has scopes** (coarse → fine): **weight-only** $\|W - \hat{W}\|$ → **layerwise** $\|WX - \hat{W}X\|$ (main focus) → **block-wise**. Each is a surrogate for the task loss.

# The setup: what $(T, A, C)$ we are allowed to choose

Before solving, we fix the grid $\mathcal{G}$. A useful way to organize the design space is **three choices** $(T, A, C)$ — each one revisits a concept from Part I (uniform/non-uniform, scalar/vector, per-tensor/channel):

**Transform $T$**

Map before quantizing.

- identity
- **scaling** (SmoothQuant, AWQ)
- **rotation** (QuaRot, Hadamard)

**Rate-allocation $A$**

How many bits, where.

- per-tensor / per-channel
- group size
- **mixed precision**

**Codebook $C$**

Shape of the levels.

- uniform (affine INT)
- non-uniform (K-means)
- **vector / lattice**

Fix $(T, A, C) \Rightarrow$ grid $\mathcal{G}$; then **QAT or PTQ** solves $\min_{\hat{\theta} \in \mathcal{G}} \mathcal{L}$. Part IV searches over $T$ and $C$.

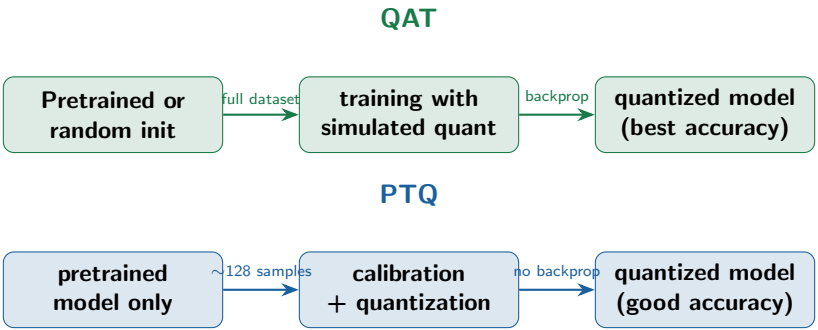
# Summary: one problem, two settings

| | QAT | PTQ |
|---|---|---|
| **Objective** $\mathcal{L}$ | task loss $\mathcal{L}_{\text{task}}$ | approximate $f_\theta$ (weight/layer/block) |
| **Data** | full training set | 100–1000 calibration samples |
| **Init** | any (incl. from scratch) | fixed pretrained $\theta$ |
| **Free variable** | weights *move* (STE grads) | discrete levels *rearrange* |
| **Cost** | training-scale (GPU-weeks) | minutes–hours, no backprop |

**Shared setup**

Both operate on a grid fixed by $(T, A, C)$. Next we drill down — **QAT** first (the ideal), then **PTQ** (the practical route for LLMs).

# Two paradigms: QAT and PTQ




**Same problem, key asymmetry.** Both minimize $\mathcal{L}$ on the $(T, A, C)$ grid, but QAT's objective is the **task loss** with **gradients** — weights adapt under quantization. PTQ's is an **approximation proxy** with **no gradients** — it can only rearrange the discrete levels of an already-trained model.

# QAT: experience quantization during training

Apply quantization in the *forward* pass so the loss "sees" the error:

$$\hat{W} = s\left(\mathcal{Q}(W/s) - z\right) \quad \text{(fake-quantize)} \qquad \Longrightarrow \qquad \mathcal{L}_{\text{task}}(\hat{W})$$

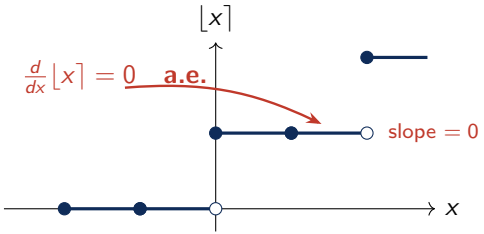
$$\xrightarrow{\text{backprop}} \quad \frac{\partial \mathcal{L}}{\partial W} \quad \xrightarrow{\text{update}} \quad W \leftarrow W - \eta \nabla_W \mathcal{L}$$

**The trick**

Weights $W$ stay **full precision** throughout training. Only the forward pass uses the fake-quantized $\hat{W}$; the backward pass updates the full-precision $W$ through it.

# But `round()` has no useful gradient

The rounding operator $\lfloor \cdot \rceil$ is piecewise constant — its derivative is zero almost everywhere, undefined at integers.



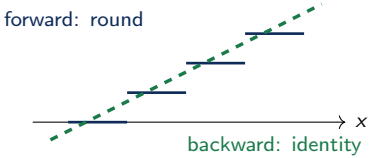

**The obstacle**

Standard backprop cannot flow a gradient through a staircase. Without a fix, the fake-quantize node blocks all learning.

# Straight-Through Estimator: pretend it's identity

**Forward:** $\hat{w} = s \left\lfloor w/s \right\rceil$ (actual rounding)

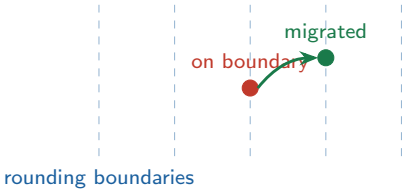
**Backward (STE):** $\dfrac{\partial \hat{w}}{\partial w} \overset{\text{STE}}{=} \mathbf{1}_{|\hat{w}| \le \beta}$ (pass gradient if not clipped)




Biased but consistent; validated across countless QAT runs; trivial to implement.

Can oscillate near boundaries; struggles at $b$=1; basic STE fixes $s$.

# Why QAT can work so well




With the full task objective, the network learns weights that are both effective *and* quantization-robust. Weights **migrate away from rounding boundaries**; the distribution reshapes to be quantization-friendly. PTQ, with no gradient, cannot do this.

# Extreme example: ternary / 1.58-bit weights

At 1–2 bits PTQ fails outright; training *from scratch* under the constraint is the only route.


three states $\Rightarrow$ $\log_2 3 \approx 1.58$ bits/weight

$$W \in \{-1, 0, +1\}^{d \times C} \qquad Y = X^\top W \quad \text{(additions only)}$$

**BitNet 1.58**

With a large token budget, ternary models approach full-precision quality at the same parameter count.

# But ultra-low-bit QAT comes with a training cost

**The training bill**

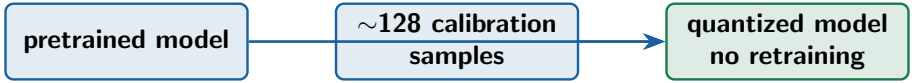
BitNet-style models train **from scratch** under the ternary constraint, on token budgets comparable to full-precision pretraining — weeks on hundreds of GPUs.

**The hardware catch**

A ternary matmul is adds/subtracts only — ideal for custom silicon with adder trees. But on today's GPUs the weights still unpack to FP16 for the Tensor Core, so the arithmetic win is not yet realised.

So ultra-low-bit QAT buys extreme compression — but only pays off with (a) a very large training budget and (b) hardware built to exploit the format. Both are out of reach for most teams.

# Why PTQ is the practical route for pretrained LLMs




**Small vision / edge model:** QAT is entirely practical — a 350M model fine-tunes with QAT on one GPU in hours.

**Released multi-billion LLM**

Full QAT means retraining on hundreds of billions of tokens — datacenter-only. **PTQ is the realistic starting point.**

Transition: PTQ fixes the objective — **minimize the layerwise reconstruction** $\|WX - \hat{W}X\|$ on the calibration set. Part IV is the search over $(T, C)$ that makes this proxy small.

Part 4

# The PTQ toolbox

Isolate, flatten, or correct the difficulty

# PTQ baseline: Round-To-Nearest (RTN)

The simplest PTQ: quantize each weight to its closest lattice point.

$$W_q = \text{clip}\Big(\big\lfloor \tfrac{W}{s} \big\rceil + z,\ 0,\ 2^b - 1\Big) \qquad \text{no optimization, no calibration loss}$$

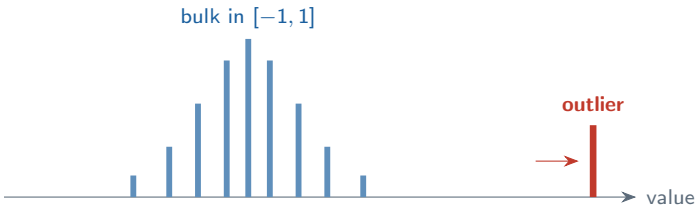

**The obvious question**

Why isn't this enough? Because one property of real weight and activation tensors quietly wrecks it — **outliers**.

# The enemy: outliers

**Definition**

An **outlier** is a value whose magnitude is 10–100$\times$ the typical value in its tensor. In LLMs these are *structural*: persistent, reproducible, concentrated in specific dimensions.




**Why they matter**

The clip range must reach the outlier — and the range sets the step size for *everything*.

# One outlier can waste almost the whole grid

**INT8**, values in $[-1, 1]$ but one outlier at 100:

$$s = \frac{100 - (-100)}{255} \approx 0.784 \qquad \Longrightarrow \qquad \left\lfloor \frac{2}{0.784} \right\rfloor + 1 = 3 \text{ levels}$$

**99%+ of values share $\approx 1\%$ of the 256 levels.**


**The range–resolution trap**

Keep the outlier $\Rightarrow$ catastrophic rounding for the bulk. Clip it $\Rightarrow$ large error that propagates through softmax / LayerNorm. **No good choice under one global scale.**

# First response: better clipping + calibration

Don't clip at $[\min, \max]$. Choose the range *deliberately*.

**Min–max**
reaches every value
(outlier dominates)

**Percentile**
clip the rare tail
(e.g. 99.9%)

**MSE-optimal**
minimize $\mathbb{E}[(w - \hat{w})^2]$

The message, not the formulas: **deliberately clipping a few rare values** improves the representation of the overwhelming majority. This is calibration — and it is prior to every algorithm that follows.

# Three PTQ strategies

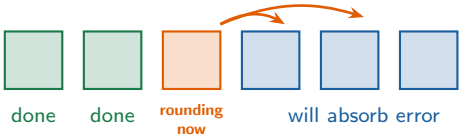

**per-channel / group** **AWQ · SmoothQuant · rotation** **GPTQ · post-correction**

**The map for the rest of Part IV**

The PTQ "zoo" is not arbitrary. Every method **isolates**, **flattens**, or **corrects** — and they compose.

# GPTQ: don't treat rounding errors independently

RTN rounds each weight in isolation. **GPTQ** asks: once we round *this* weight wrong, can the *remaining* weights absorb the error?




**GPTQ (Frantar et al. 2022)**

Quantize columns in order. After each rounding, use layer input statistics (a **second-order / Hessian** term) to nudge the unquantized weights so the layer output stays close to the original.

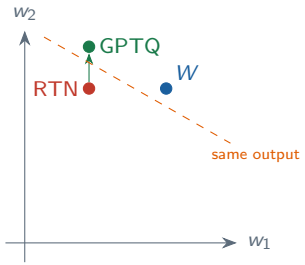
# GPTQ intuition: error redistribution

$$\Delta W_{\text{rest}} \;=\; -\frac{e_q}{[H^{-1}]_{qq}}\,\big[H^{-1}\big]_{:,q}$$

$H = 2XX^\top$ (layer Hessian)

**In words**

The rounding error $e_q$ of the current weight is pushed onto the remaining weights **in proportion to their Hessian coupling** — the directions the layer output is most sensitive to.

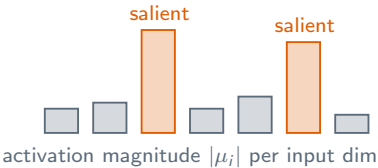



No Hessian algebra needed — the picture is: move along the "same-output" direction.

# AWQ: not all weights are equally important

**The starting observation**

Weight rows driven by **large-magnitude activation dimensions** matter far more to the output. Protect those, and quantization error where it counts drops sharply.




A tiny fraction of dimensions (often 1%) carry most of the output influence. **AWQ** finds them and makes them *easier to quantize* rather than storing them in FP16.

# AWQ: scale before quantization

$$X^\top W \;=\; \underbrace{\left(X\ \mathrm{diag}(\boldsymbol{s})^{-1}\right)}_{\tilde{X}}\ \underbrace{\left(\mathrm{diag}(\boldsymbol{s})\,W\right)}_{\tilde{W}} \qquad \textbf{product exactly preserved}$$

Pick $s_i > 1$ for salient dims: $\tilde{W}_{i,:} = s_i W_{i,:}$ scales the salient row *up*, so it occupies more of the grid — finer effective resolution where it matters.

**Free at inference**

$\boldsymbol{s}^{-1}$ folds into the preceding LayerNorm's $\gamma$ offline. **No new ops, no runtime cost** — same FP16 math, friendlier weight geometry.

# GPTQ versus AWQ

| | GPTQ | AWQ |
|---|---|---|
| **When it acts** | *after* rounding | *before* rounding |
| **Mechanism** | redistribute error (2nd-order) | rescale salient dims (activation-guided) |
| **Guided by** | layer Hessian $XX^\top$ | activation magnitudes $\lvert\mu_i\rvert$ |
| **Runtime cost** | none (offline) | none (folds into LayerNorm) |

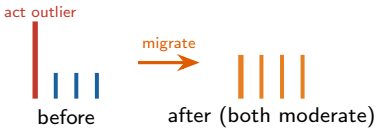
Not competitors — opposite directions on the same problem, and **composable** (AWQ then GPTQ). Crucially, both emit the *same* hardware-friendly INT4 per-group format.

# If activations must be quantized: SmoothQuant

When *activations* are quantized too (W8A8), **SmoothQuant migrates** the difficulty into the weights.

$$Y = X\,W = \left(X\ \mathrm{diag}(\boldsymbol{s})^{-1}\right)\left(\mathrm{diag}(\boldsymbol{s})\,W\right)$$

divide act. dim $i$ by $s_i$; multiply weight row $i$ by $s_i$



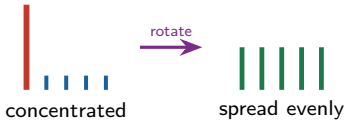

**Product preserved**

Output identical; **difficulty moves** from activations to weights ($\alpha$ tunes how much).

# Another answer: rotate the representation

Some outliers are spread across dims with no single one to migrate. An **orthogonal / Hadamard** rotation $R$ spreads the concentrated energy.

$$\tilde{x} = Rx, \quad R^\top R = I \qquad \underbrace{\|\tilde{x}\|_2 = \|x\|_2}_{\text{energy preserved}} \qquad \underbrace{\|\tilde{x}\|_\infty \leq \|x\|_\infty}_{\text{peak reduced}}$$




Since the clip range is set by $\|\cdot\|_\infty$, lowering the peak **tightens the grid** while preserving the product. Basis of **QuaRot** and **FlatQuant**.

# From algorithm to deployment: vendor toolchains

## NVIDIA GPU (datacenter / Jetson)

**Quantize** — **TensorRT Model Optimizer** runs the PTQ algorithms: **AWQ**, **GPTQ**, SmoothQuant, FP8 / NVFP4.

**Compile & serve** — **TensorRT-LLM** (W4A16, W4A8, FP8 kernels); **vLLM** / **SGLang** run the same AWQ/GPTQ checkpoints directly.

Pre-quantized **AWQ / FP8** checkpoints ship on Hugging Face — often no calibration on your side.

```
LLM("nvidia/Llama-3.1-8B-FP8")
```

## Qualcomm Snapdragon (Hexagon NPU)

**Quantize** — **AIMET** (AI Model Efficiency Toolkit): **W8A8 / W8A16 / A16W4**, Spin-Quant, SeqMSE.

**Compile & run** — **QNN / Qualcomm AI Engine Direct (QAIRT)** → Hexagon; LLMs via **Genie**. Two front-ends: **AI Hub Workbench** (cloud compile / profile on real devices) or the **CLI**.

```
qai_hub_models...export --quantize w8a8
```

Same idea, two ecosystems: **the quantizer produces a checkpoint; a hardware-specific compiler turns it into an executable kernel** — GPU paths lean on **FP8 / INT4 GEMM**, Snapdragon on fixed **INT8 / INT4 Hexagon** binaries.

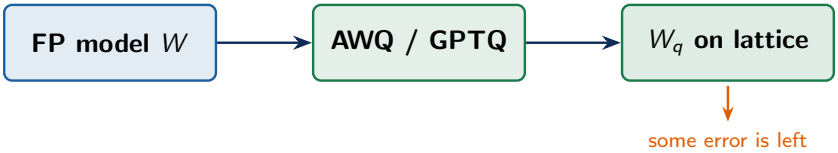
Part 5

# Post-quantization correction

Repairing the residual on the lattice — CLC

# After PTQ, are we finished?

We have the right bit-width, the right hardware format, and a good algorithm (RTN, GPTQ, AWQ, . . . ). Are we done?




**The residual is unavoidable**

The final step is always a discrete **projection onto the lattice**. No matter how good the algorithm, the projection leaves a residual. The question of this part: can we *repair* it — cheaply, in-place?

# Quantization residual at a linear layer

**Residual:** $E := W_q - W, \quad \text{column } e_j := E_{:,j}, \quad |e_{i,j}| \le s_j/2$

**Output error** at a linear layer $y = x^\top W$:

$$y_q - y \;=\; x^\top (W_q - W) \;=\; x^\top E$$

**Very simple, very general**

The layer output error is just the input times the weight residual. Everything CLC does follows from looking closely at the statistics of this one quantity, $x^\top E$.

# What if activations have non-zero mean?

Let $\mu := \mathbb{E}[x]$. Decompose the expected squared output error per channel $j$:

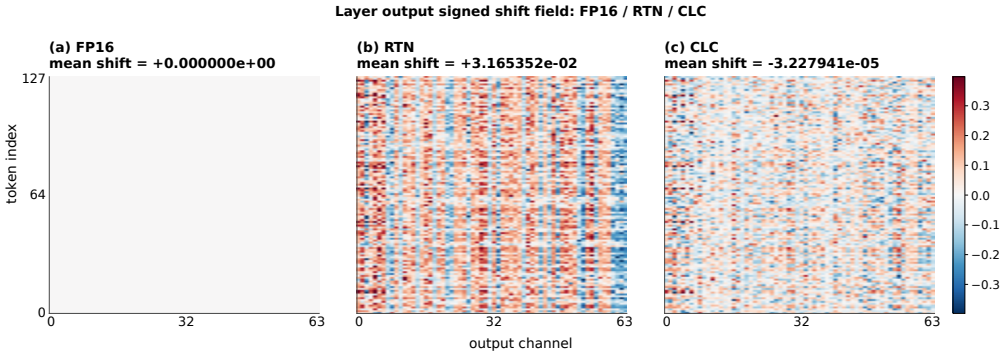
$$\mathbb{E}\left[(x^\top e_j)^2\right] = \underbrace{(\mu^\top e_j)^2}_{\text{first moment } b_j^2} + \underbrace{e_j^\top \Sigma\, e_j}_{\text{variance}} \qquad b_j := \mu^\top e_j$$

**The key observation**

The residual is *not* harmless zero-mean noise. When $\mu \neq 0$, the term $b_j = \mu^\top e_j$ is a **systematic expected output shift** — a structured first-moment drift, not just variance.

In LLMs $\mu \neq 0$: the residual stream and a few large-$|\mu_i|$ dimensions guarantee it.

# See the structured error




Signed output-shift field (tokens × channels). **FP16** flat; **RTN** drifts (mean $\approx +3\times10^{-2}$); **CLC** pulls it back to $\approx 0$ — **structured drift removed**, same lattice.

# Why not just add a bias correction?

The classical fix: add an output-side bias $\beta_j = \mu^\top \Delta w_j$ to cancel the shift.

**It does align the first moment** — exactly cancels $b_j$ in expectation.

**But two problems**

Modern LLMs (LLaMA, Mistral, Phi, Falcon) are **bias-less** — there is no bias term to write into.

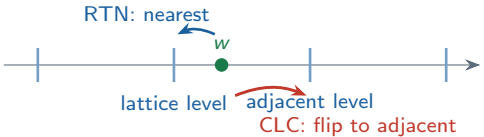
Adding one *modifies the deployment graph*.

**AdaRound / BRECQ, briefly**

Another family *optimizes the rounding decisions* after quantization (floor vs. ceiling per weight) against a calibration loss. Effective — but next we examine one lightweight correction in detail: **CLC**.

# CLC: stay on the same integer lattice

Instead of adding a bias, **CLC** revises selected *rounding decisions* — moving a weight to its *adjacent* lattice level.




Every corrected weight is still a **valid low-bit code** on the original lattice $s_j\mathbb{Z}$ — no new parameters, no bias, same format.

# Complementary lattice update

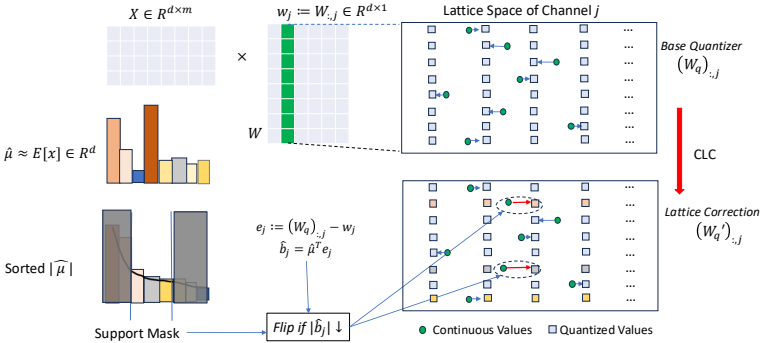
**For selected entries:**

$$\Delta_{i,j} = -\operatorname{sign}(e_{i,j})\, s_j$$

one adjacent-level flip, *against* the residual.

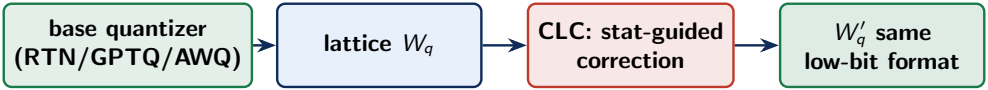
**Budget-constrained**

A greedy descent flips the entries that most reduce $\hat{b}_j = \hat{\mu}^\top e_j$, under a flip budget — variance cannot grow uncontrolled.




Green: continuous weights □: lattice levels red arrows: adjacent-level flips

# CLC as an on-top correction layer




**What CLC is, in one line**

A **drop-in post-processing step**: no new architecture, correction operates entirely in discrete weight space, output deploys in the exact same low-bit format — with a provable reduction of the first-moment error under a verifiable budget.

hardware → format → algorithm → *correction* — now, what do we deploy?

# Experiments: setup

**Models**

Four bias-less LLMs at **4-bit** and **3-bit** weight-only: Mistral-7B-v0.3, LLaMA-3.1-8B, LLaMA-3-8B, Qwen2.5-7B.

**Base quantizers**

CLC runs *on top of* each:

- **RTN** — nearest-level projection
- **GPTQ** — Hessian-aware compensation
- **AWQ** — activation-aware scaling
- **FlatQuant** — transform-based SOTA

**Calibration.** 128 C4 sequences of length 2048 — used only to estimate the activation mean $\hat{\mu}$ and build the flip mask.

Group size 128, per-channel asymmetric scaling. CLC runs *after* the base quantizer emits integer weights.

**Comparison point**

Classical **bias correction (BC)** — but these models are bias-less, so BC must *add* parameters they never had. CLC does not.

# What we measure

## Perplexity (PPL) ↓

Language-modelling loss on held-out **WikiText-2** and **C4**; lower is better.

$$\mathrm{PPL} = \exp\Big(\tfrac{1}{N}\textstyle\sum_t - \log p(x_t \mid x_{<t})\Big)$$

The primary signal — most sensitive to the residual CLC targets.

## Downstream accuracy ↑

Zero-shot accuracy on five tasks — `ARC-c`, `ARC-e`, `BoolQ`, `PIQA`, `RTE` — and their unweighted **average**. Confirms gains reach real tasks.

**Reading the tables.** Within each (model, base) block:

green — best non-FP16 result

red — *degraded* vs. its base quantizer

**FP16** is the un-quantized ceiling; every quantized row aims to close the gap back to it.

± values are standard deviations over runs; bold marks the best non-FP16 row in each block.

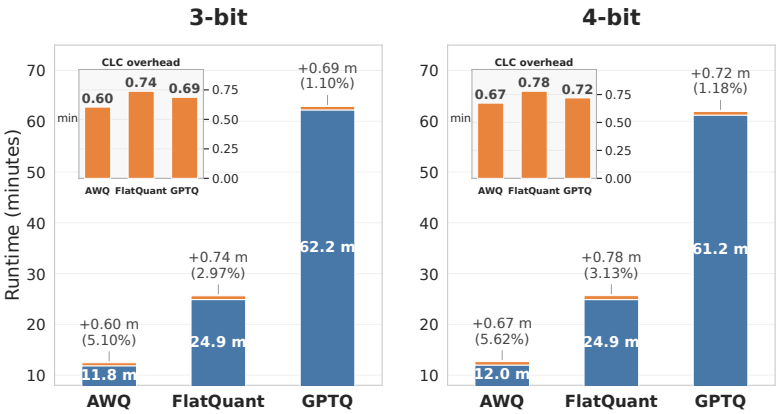
# A typical result: 3-bit weight-only (Mistral-7B)

| Method | PPL (↓) | | Zero-shot accuracy (↑) | | | | | |
|---|---|---|---|---|---|---|---|---|
| | Wiki | C4 | arc-c | arc-e | boolq | piqa | rte | Avg |
| FP16 | 4.845 | 7.535 | 0.509 | 0.802 | 0.837 | 0.800 | 0.686 | 0.727 |
| RTN | 5.680 | 8.883 | 0.417 | 0.663 | 0.774 | 0.773 | 0.621 | 0.650 |
| RTN + BC | 5.682 | 8.887 | 0.420 | 0.664 | 0.772 | 0.788 | 0.634 | 0.656 |
| **RTN + CLC** | **5.639** | **8.740** | **0.422** | **0.665** | **0.788** | **0.798** | **0.676** | **0.670** |
| AWQ | 5.572 | 8.648 | 0.439 | 0.763 | 0.780 | 0.787 | 0.603 | 0.675 |
| AWQ + BC | 5.566 | 8.633 | 0.437 | 0.760 | 0.785 | 0.787 | 0.606 | 0.675 |
| **AWQ + CLC** | **5.524** | **8.552** | **0.440** | **0.766** | **0.794** | 0.787 | **0.617** | **0.681** |
| GPTQ | 5.424 | 8.308 | 0.449 | 0.764 | 0.785 | 0.781 | 0.667 | 0.689 |
| GPTQ + BC | 5.418 | 8.302 | 0.449 | 0.764 | 0.781 | 0.783 | 0.664 | 0.688 |
| **GPTQ + CLC** | **5.382** | **8.277** | **0.453** | **0.774** | **0.808** | **0.790** | **0.671** | **0.699** |
| FlatQuant | 5.622 | 8.566 | 0.421 | 0.759 | 0.790 | 0.789 | 0.581 | 0.668 |
| FlatQuant + BC | 5.901 | 8.991 | 0.417 | 0.750 | **0.791** | 0.785 | 0.598 | 0.668 |
| **FlatQuant + CLC** | **5.587** | **8.524** | **0.433** | **0.759** | 0.788 | **0.791** | **0.607** | **0.676** |

Across *all four* base quantizers, **CLC lowers both PPLs and lifts average accuracy**, while plain BC often lands in the red (e.g. FlatQuant + BC worsens Wiki PPL 5.62 → 5.90) — gains are largest here at 3-bit, where the lattice is coarsest.

# What does the correction cost? (Mistral-7B runtime)

CLC adds only 0.6–0.8 **min** on any base quantizer — the **AWQ** ∼5% is large only *because AWQ itself is cheap*; on **GPTQ** it is ∼1%. **The correction is essentially free.**

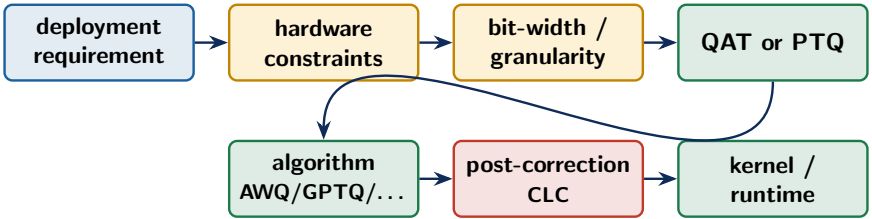


Part 6

# Synthesis

What should I actually deploy?

# The complete pipeline

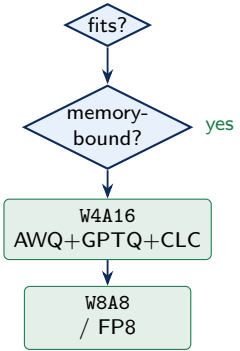



**Every part reconnected**

**Hardware** decides the format (Part II); **QAT/PTQ** decides how we reach it (Part III); the **algorithm** reduces the damage (Part IV); **correction** repairs the residual (Part V); the **kernel** makes it fast (Part II again).

# A practical Edge AI decision tree

1. **Does it fit?** $N\times$bytes vs. device memory $\rightarrow$ pick enough compression.
2. **What precision is native?** INT8-only NPU vs. flexible GPU.
3. **Memory- or compute-bound?** small batch vs. large.
4. **Weight-only or weight+activation?** `W4A16` vs. `W8A8`.
5. **QAT available?** small model $\rightarrow$ yes; big LLM $\rightarrow$ PTQ.
6. **Kernel / runtime support?** is the packed format actually executable?
7. **Correct residual?** apply CLC if it helps.




Answer them **in order**. Each answer narrows the next — the format is decided before the algorithm, and the algorithm before the correction.

# Take it home

**1 — Quantization is an optimization problem**

Given the hardware's constraints, we solve $\min_{\hat{\theta} \in \mathcal{G}(T,A,C)} \mathcal{L}(\hat{\theta})$: the hardware fixes the grid $\mathcal{G}$ (the format $T, A, C$), and the algorithm searches within it. Fewer bits help only when the memory system, datapath, packing, and kernel can exploit them — so it is **hardware co-design**, not mere compression.

**2 — A lightweight correction on top of any quantizer**

We introduced **CLC**, a lightweight correction that sits on top of commonly used quantizers (RTN, AWQ, GPTQ, FlatQuant) to recover performance *without* changing the model architecture or the deployment format — it reduces structured first-moment error while staying on the same integer lattice.

# Thank you for listening.
# Questions?