Table 19.1.: Running-example ledger for the AWQ INT4 Llama decode-time projection. Each row preserves the same logical linear operator while changing the stored representation and the metadata that makes the representation executable.

Projection path	Stored layer state	Metadata the runtime must carry	Serving contract
BF16 baseline	Dense weight W consumed by the ordinary linear path.	Shape, dtype, layout, and module identity.	Reference interpretation of Wx_i for comparison.
INT4 group path	Packed four-bit weight codes.	Group size, scale, zero point, optional group-index map, and packing order.	The same code can mean a different value under different group metadata.
FP8 or block-scaled path	Low-precision blocks with block-local scaling.	FP8 format, block shape, scale dtype, and eligible kernel.	Dispatch must choose a compatible kernel or fall back without changing Wx_i's logical meaning.
GGUF or tagged compact path	Format-tagged packed tensor from the checkpoint.	Format tag, block metadata, and conversion or direct-consumption path.	The loader cannot treat packed bytes as self-describing weights.

FP8 blocks, GGUF packed tensors, or another compact format. The request still expects the same logical projection. Output shape alone is not enough: applying the wrong group scale, dropping a zero-point field, or dispatching to a kernel with a different packing convention changes the model rather than optimizing it [8, 20].

Table 19.1 records the running-example state before the section follows its later transitions.

Quantized serving therefore treats a layer as a representation contract. Section 19.2 defines the encoded values, scale layouts, format tags, and kernel-layout vocabulary that make packed bytes executable as a quantized operator (Definition 19.2.2). Section 19.3 separates the targets of quantization: weights, activations, MoE state, and KV-cache state have different lifetimes and consuming boundaries. Section 19.4 then states the shape-preserving failures that motivate the implementation discipline. The following sections follow the same contract through SGLang's method registry, checkpoint metadata, loading path, repacking steps, kernel dispatch, path classification, measurement protocol, handoff boundary, and source-history evolution.

The useful mental model is a lifecycle:

$$\begin{aligned}\text{checkpoint representation} &\to \text{SGLang method} \to \text{loaded layout} \\ &\to \text{dispatchable kernel} \to \text{measured serving operator.}\end{aligned}$$

Every arrow must preserve the same interpretation of the encoded state. The formulas and source anchors follow that lifecycle as a structured account of quantization formats.

The encoded weight tensor needs a ledger that follows it past loading. Table 19.2 names the dense value, encoded weight, activation row, scale group,

Symbol	Meaning	Role in the serving account
W	Original dense weight	The reference layer state whose operator semantics the quantized path must preserve.
$\widehat{W}$	Encoded or packed weight	The stored representation consumed by a quantized method, loader, or kernel.
x_i	Activation row	The request-visible input row whose quantized operator must preserve layer semantics.
G, s_G, z_G	Scale group, scale, and zero point	Metadata that gives packed codes numerical meaning inside one quantization group.
τ_q	Quantization format tag	The AWQ, GPTQ, FP8, GGUF, compressed-tensors, or kernel-family tag that drives loader and dispatch choices.
κ	Selected kernel layout	The executable layout, packing, scale placement, and workspace contract that a specialized kernel can legally consume.

Table 19.2.: Local notation for quantized serving. The symbols separate logical layer semantics, stored representation, interpretation metadata, and dispatch eligibility.

format tag, and kernel-layout contract that must agree before a lower-precision path is a serving path.

19.2. Representation Metadata: Encoded Values, Scale Layouts, and Tags

Quantized serving starts by separating dense values, encoded values, and the scale metadata that lets kernels reconstruct the logical matrix.

Quantization trades bytes and bandwidth against numerical and kernel constraints.

The representation becomes executable when SGLang can name both the stored bytes and the rule that interprets them. Figure 19.1 frames that split as dense weights, encoded blocks, scale metadata, and zero-point metadata. A packed tensor places several low-precision codes in each kernel storage unit; scale, zero-point, group-index, and format metadata say how those codes reconstruct the logical weights. Dense FP16 or BF16 tensors need no decoder, and unpacked quantized tensors give each code its own slot. Packing saves resident bytes and memory traffic, but it makes the tensor non-self-describing: the packing convention, block metadata, and format tag must travel with the bytes.

The layer operator defines the serving contract; the compressed tensor is one representation it may consume.

The concrete INT4 path is the AWQ Llama checkpoint named in the running example, which prepares the later Marlin discussion. A four-bit code q has only 16 possible values, interpreted either as signed codes $-8, \ldots, 7$ or as unsigned codes $0, \ldots, 15$ with an offset, according to the checkpoint metadata and selected SGLang method. In simple layouts, one quantization group is a consecutive block of logical tensor indices; explicit group-index metadata can assign columns to

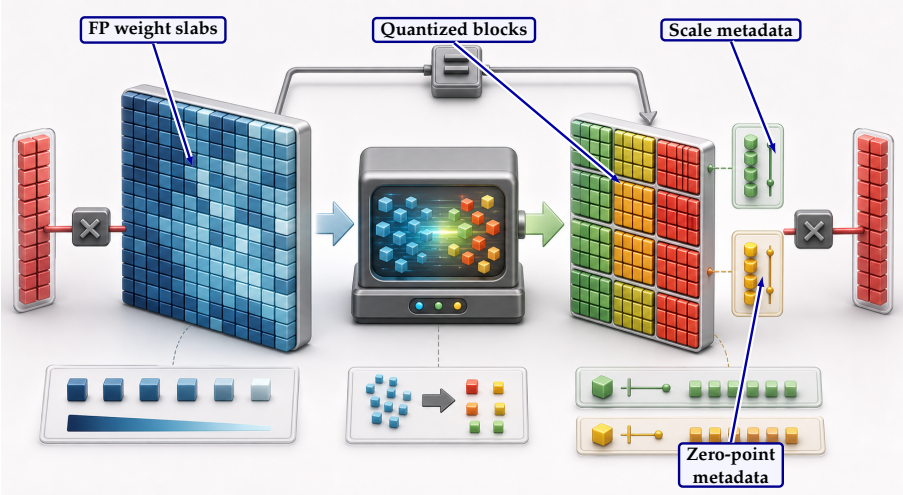

Figure 19.1.: Floating-point weights encoded into quantized blocks. Dense floating-point weight slabs being transformed into compact quantized blocks with separate scale and zero-point metadata, preserving the same logical matrix operation.

groups in a different order. For an affine unsigned INT4 group, the group-local reconstruction rule is [8, 20]:

The number-line intuition is that a small real interval is replaced by integer ticks. The scale s_G is the distance between adjacent ticks, and the zero point z_G names the integer tick that should land on real zero. Subtracting z_G recenters the code, and multiplying by s_G stretches that centered integer back onto the approximate real line.

Definition 19.2.1 (Affine reconstruction rule) *Let G be an integer quantization group: a set of logical tensor indices that share one scale s_G and one zero point z_G. Each encoded value q_i reconstructs as*

$$\widetilde{w}_i = s_G(q_i - z_G), \qquad i \in G,$$

where z_G is the encoded value that represents real zero. A symmetric signed group is the special case $z_G = 0$, giving $\widetilde{w}_i = s_G q_i$.

q_i is stored data; s_G, z_G, and G define how to read it.

Example 19.2.1 (INT4 groups and metadata) Consider eight logical weights stored as two consecutive groups of four INT4 values. Group G_0 uses $s_0 = 0.05$ and $z_0 = 8$; group G_1 uses $s_1 = 0.20$ and $z_1 = 7$. The same stored code $q = 10$ means different real values in the two groups:

$$0.05(10 - 8) = 0.10, \qquad 0.20(10 - 7) = 0.60.$$

Two INT4 codes are packed into one byte, but the byte alone does not say which scale and zero point to use. The group boundary is therefore part of the operator as executable metadata.

Example 19.2.2 (INT4 packing and FP8 field layouts) For the byte-level convention used by most low-bit weight paths, one byte stores two INT4 codes. If byte $b = \texttt{0xA3}$, the low nibble is

$$q_0 = b \;\&\; \texttt{0x0f} = 3,$$

and the high nibble is

$$q_1 = (b \gg 4) \;\&\; \texttt{0x0f} = 10.$$

The logical order is therefore not recoverable from the byte alone; the kernel layout must say whether the low nibble is column k and the high nibble is column $k + 1$, or whether a repacker has rearranged nibbles into a tensor-core-friendly tile. With $s = 0.05$ and $z = 8$, the two unsigned affine values reconstruct as $0.05(3 - 8) = -0.25$ and $0.05(10 - 8) = 0.10$.
FP8 has a different byte contract. E4M3 uses one sign bit, four exponent bits, and three mantissa bits; E5M2 uses one sign bit, five exponent bits, and two mantissa bits. E4M3 spends more of the byte on precision and is the common serving GEMM payload in this chapter; E5M2 spends more bits on exponent range. The format tag must distinguish them before any scale tensor is consulted, because the same payload byte has different exponent and mantissa fields under the two decoders.

This is the local quantization version of the Volume II preliminaries' representation-ledger rule: small byte counts are meaningful only after the reader knows which metadata interprets each stored value.

Group-index metadata is the nonconsecutive version of the same rule. With eight input columns and two groups, ordinary consecutive grouping would imply $g_{\text{idx}} = [0, 0, 0, 0, 1, 1, 1, 1]$. An activation-order GPTQ checkpoint may instead carry

$$g_{\text{idx}} = [0, 0, 1, 1, 0, 0, 1, 1].$$

Columns $0, 1, 4, 5$ then use group 0's scale and zero point, while columns $2, 3, 6, 7$ use group 1's metadata. The encoded value q_4 reconstructs with $s_0(q_4 - z_0)$, not with the group-1 metadata that a consecutive group-size rule would assign to column 4. This is why `g_idx` belongs to the operator metadata and to the loader.

INT4 is still only the local example. The same metadata question also covers W8A8 activation scales, FP8 exponent-and-mantissa formats, GGUF block metadata, and runtime KV-cache or activation quantization. The common systems problem is the same: stored bytes become an executable layer only when SGLang carries the interpretation metadata with them. The definitions below therefore separate the encoded tensor, the metadata map, the scale layout, the format tag, and the kernel layout instead of treating $\widetilde{w} = s(q - z)$ as the only quantization rule [46, 11].

Definition 19.2.2 (Quantized operator) *A quantized operator is a linear or embedding operation whose stored parameters use a compressed numeric representation together with metadata that reconstructs the value used by the arithmetic path. For a linear layer, the symbolic form is*

$$y = xD_m(\widehat{W}),$$

where $\widehat{W}$ is the encoded weight tensor, m is the interpretation metadata, and D_m is the dequantization, unpacking, or scale-application map consumed by the selected backend.

Throughout this chapter, $\widehat{W}$ denotes the encoded or packed tensor, while $\widetilde{w}_i$ denotes a reconstructed scalar value. For linear layers we use the algebraic shape convention

$$x \in \mathbb{R}^{1\times K}, \qquad D_m(\widehat{W}) \in \mathbb{R}^{K\times N}, \qquad y \in \mathbb{R}^{1\times N},$$

so K is the input or contraction dimension and N is the output dimension. Backend storage may transpose or pack this matrix, but the logical operator is always $xD_m(\widehat{W})$.

For a tiny linear layer, $x = [1, 2]$ is the activation row, $\widehat{W}$ is the stored INT4 or FP8 weight block, m contains the scale layout and numeric metadata for that block, $D_m(\widehat{W})$ is the backend-readable reconstructed weight matrix, and $y = xD_m(\widehat{W})$ is the same output shape the unquantized linear layer would produce.

The quantized operator is the semantic object: it must still implement the intended linear or embedding operation. Scale layout is the first piece of metadata that tells the runtime which encoded elements share interpretation parameters.

Definition 19.2.3 (Scale layout) *A scale layout is the partition of tensor elements into groups that share scale, zero-point, exponent, or block metadata. Equivalently, it is a map*

$$\ell_q : \mathrm{idx}(W) \to \mathcal{G}_q$$

from tensor-element indices to quantization groups; elements i and j share interpretation metadata exactly when $\ell_q(i) = \ell_q(j)$. Each group $G \in \mathcal{G}_q$ is the contiguous or layout-defined set of encoded tensor elements for which its metadata m_G are valid; applying m_G outside G, or applying another group's metadata inside G, changes the numerical operator even when the tensor shape is unchanged.

ℓ_q maps tensor indices to metadata groups; it is not a kernel layout.

For a row whose weights are grouped in blocks of 128 elements, indices $0, \ldots, 127$ map to one group and indices $128, \ldots, 255$ map to the next. Two adjacent encoded bytes can therefore have different scales when they sit on opposite sides of a group boundary.

The scale-layout definition gives metadata a scope. It is helpful to read the fields in two groups before compressing them into one tuple. The reconstruction fields, ℓ_q, s, and z, say which real value each encoded code means. The execution fields, p, η, and δ, say how many codes are packed together, which format-specific extras are present, and which backend convention applies them. In this chapter,

m denotes that combined quantization metadata:

$$m = (\ell_q, s, z, p, \eta, \delta)$$

where the components name the scale layout, scale values, zero points, packing factor, optional format-specific fields such as exponents or bias terms, and backend convention δ for applying them.

For an INT4 groupwise weight path, m might bind ℓ_q to 128-element groups, s to one scale per group, z to a zero-point table or $\varnothing$, $p = 2$ values per byte, $\eta = \varnothing$, and δ to the backend's dequantization convention. This convention must be compatible with the chosen kernel layout, but it is not the kernel layout itself. A kernel that assumes a different packing factor reads the same bytes as a different operator.

Quantization metadata states how the stored values are interpreted. Kernel layout adds the physical ordering and compatibility requirements expected by one specialized arithmetic path.

Definition 19.2.4 (Kernel layout) *A kernel layout κ is the packed tensor ordering, tile shape, workspace convention, compatibility predicate, and optional permutation metadata expected by a specialized arithmetic kernel.*

Here GPTQ and AWQ name checkpoint-side quantization schemes, not kernel layouts. GPTQ stores low-bit weights with scale, zero-point, and sometimes group-index metadata chosen to preserve the layer's output under post-training quantization. AWQ also stores low-bit grouped weights and metadata, but its calibration emphasizes activation-aware weight scaling before packing. In this chapter, the important distinction is that GPTQ or AWQ define what the packed checkpoint bytes mean, while Marlin defines a compatible executable layout and CUDA kernel path for serving those bytes [8, 20]. The loading section develops that compatibility check as a checkpoint-to-layout problem, and the dispatch section develops it as a runnable-kernel problem.

Kernel layout is backend-specific execution metadata. The format tag below is a checkpoint or runtime label that selects the decoder for the stored bytes before that execution metadata can be trusted.

Definition 19.2.5 (Format tag) *A format tag $f \in \mathcal{F}$ is runtime metadata that names the concrete encoding of a stored tensor and selects the decoder $D_{m,f}$ or direct packed-kernel family allowed to interpret those bytes.*

$D_{m,f}$ is the same interpretation map as D_m, specialized by the format tag f. When a direct packed kernel is selected instead of an explicit dequantization routine, it must still compute the same declared map $xD_{m,f}(\widehat{W})$.

GGUF is a concrete example of why the format tag has to stay with the tensor. The file format solves a loading problem: a single file carries GGML tensor names, packed tensor bytes, and per-tensor quantization types instead of relying on a separate SGLang quantization config for every tensor. The tag then becomes execution metadata: it selects the decoder or direct packed-kernel family that is allowed to interpret those bytes [11].

The same INT4 group example motivates the serving problem: SGLang must keep $\widehat{W}$, m, D_m, and the selected kernel layout aligned while the serving system changes batch shape, hardware backend, and kernel choice. If the group size is read as eight instead of four, the output tensor can keep the same shape while applying the wrong scale layout to half the row.

19.3. Targets and Ownership Boundaries

Quantization target decides which runtime boundary consumes the metadata.

A serving runtime does not quantize "the model" as one undifferentiated object. It quantizes particular state with particular lifetimes. Weight quantization is loaded once and consumed by linear, embedding, or MoE kernels. Activation quantization may be produced online for the current batch. MoE quantization has to preserve expert packing and fused-runner layout. KV quantization changes persistent decode state and therefore crosses the memory management boundary as well as the layer boundary. These targets share scale layout vocabulary, but they do not share the same ownership rules. In the notation of Definition 19.2.2, each target decides which runtime boundary owns the metadata m that makes $xD_m(\widehat{W})$ the declared serving operation rather than just a smaller tensor.

- **Weights.** Weight quantization changes how checkpoint tensors are stored and how layer methods create executable parameters such as `qweight`, `qzeros`, `scales`, block-scale tensors, or GGUF packed buffers. The representation is usually fixed before requests arrive, so the loader and layer method must validate group size, packing factor, shard shape, and backend eligibility before the first prefill or decode step consumes the layer.
- **Activations.** Activation quantization belongs to the current forward call. A W8A8 path, for example, must keep the activation scale layout, weight scale layout, output dtype, optional bias, and kernel branch aligned. Dynamic activation scales are therefore serving-time metadata: they are computed or selected for the current batch and must be consumed with the same row order that produced them [46].
- **MoE state.** MoE quantization adds expert layout to the contract. Fused expert runners may store expert weights in packed layouts, remap compressed-tensors schemes onto `FusedMoE` or `DeepEPMoE` targets, and select specialized kernels only for compatible group sizes, activation dtypes, and expert packing. A correct quantized MoE path must preserve both the numerical scale layout and the token-to-expert routing layout. Chapter 21 develops MoE routing and dispatch; here the needed quantization point is that expert weights may be owned by a runtime MoE module rather than by separate checkpoint `Linear` layers.
- **KV state.** KV-cache quantization changes resident attention state that is allocated, retained, evicted, and read across many decode steps. Registry entries that install both linear/MoE schemes and a KV-cache method therefore extend the quantization contract into page ownership, token-position validity, and attention-backend compatibility. The detailed KV allocator invariants belong to the memory chapters, but this chapter needs the boundary: a KV quantization field is part of the executable serving state when it changes how cached keys or values are stored or reconstructed.

Concept-to-code map

- **Quantization method definitions.** `python/sglang/srt/layers/quantization/` contains weight, activation, MoE, and KV-cache quantization methods.
- **KV-cache dtype and FP8 scale loading.** `python/sglang/srt/model_executor/model_runner.py` maps `--kv-cache-dtype` to the actual cache dtype and loads FP8 KV-cache scale files when `--quantization-param-path` is supplied.
- **FP8 K/V scale state.** `python/sglang/srt/layers/quantization/kv_cache.py` carries FP8 K/V scale state for attention layers.
- **FP4 KV-cache helpers.** `python/sglang/srt/layers/quantization/fp4_kv_cache_quant_method.py` and `kvfp4_tensor.py` cover FP4 KV-cache strategy, packed data, and scale-buffer helpers.
- **Quantized KV cache-pool boundaries.** `python/sglang/srt/mem_cache/memory_pool.py` owns cache-pool storage, quantize-on-write, and dequantize-on-read boundaries for quantized KV state.
- **W8A8 INT8 scale handling.** `python/sglang/srt/layers/quantization/w8a8_int8.py` handles W8A8 INT8 weight and activation scales.
- **Compressed-tensor target mapping.** `python/sglang/srt/layers/quantization/compressed_tensors/` maps target schemes for linear and fused MoE modules.
- **Quantization registry names.** `python/sglang/srt/layers/quantization/__init__.py` installs linear, MoE, and KV-cache quantization behavior.
- **Target-specific kernel tests.** `sgl-kernel/tests/test_per_token_group_quant_8bit.py`, `test_int8_gemm.py`, and `test_*fp*` tie target-specific metadata to kernel-visible behavior.

Table 19.3 splits quantization ownership by target so packed weights, activation scales, MoE state, and KV state are validated at the right boundary.

Target	State owned by the quantized path	Validation point	Shape-preserving risk
Weights	Packed checkpoint tensors, group size, zero points, scales, and packing factor.	Loader and layer-method construction before the first request.	Correct output width with wrong decoded weights.
Activations	Per-token or per-tensor input scales and output dtype for the current forward call.	Batch-time scale construction and kernel launch.	Stale or mismatched activation scales are applied to ordered rows.
MoE state	Expert packed weights, fused-runner target scheme, and token-to-expert layout.	Fused MoE or DeepEP runner construction and dispatch.	The selected expert consumes weights in the wrong layout.
KV state	Quantized cached keys or values, page ownership, and attention-backend compatibility.	Cache allocation, reuse, eviction, and attention reads.	A reused page is reconstructed with the wrong quantization rule.

Table 19.3.: Quantization target ownership. The target determines which boundary must validate metadata before a packed representation becomes executable serving state.

Example 19.3.1 (KV-cache dtype is serving state) Suppose a serving run sets `--kv-cache-dtypefp8_e4m3`. This does not rewrite checkpoint weights. It changes the representation of resident attention state that the allocator owns after each forward step. During initialization, SGLang resolves the requested cache dtype, optionally loads per-tensor K/V scales from `--quantization-param-path`, and allocates token-to-KV-pool buffers in the selected representation. During a decode append, the newly computed keys and values are converted into the cache representation and written at the physical locations assigned by the scheduler. Later attention reads the same logical history through backend metadata, but it must interpret the stored bytes under the same scale convention.
At the server-argument boundary, runtime state is `kv_cache_dtype` plus an optional scale-file path. The validation question is whether the requested cache dtype is supported on this platform and model path; otherwise the run silently uses a different byte account or rejects late. At the layer-scale boundary, runtime state is per-tensor K/V scales for FP8 or FP4 scale-side buffers. The validation question is whether scales are present, intentionally defaulted, or computed with the packed data; otherwise the cache is resident but decoded under the wrong scale.
At the cache-pool boundary, runtime state is K/V bytes, scale buffers, page locations, and owner rows. The validation question is whether quantized writes preserve the allocator's physical-location contract; otherwise another request's page or an old scale record is read. At the attention-backend boundary, runtime state is the direct FP8/FP4 read path or a dequantization workspace. The validation question is whether the selected backend can consume this cache representation efficiently; otherwise memory savings are offset by a slow or unsupported read path.
This example separates KV-cache quantization from checkpoint weight formats. The packed state is produced online, lives across decode steps, and remains valid only while cache ownership, scale metadata, and backend interpretation move together.

The important organizing rule is that the target determines where validation must happen. Weight metadata can often be rejected during loading. Activation metadata must be checked near the forward-call shape. MoE metadata must also agree with expert routing and fused-kernel layout. KV metadata must survive cache allocation and reuse across steps. Treating those cases as separate targets avoids mixing file-format parsing, kernel dispatch, and cache lifetime into one flat list of quantization schemes.

19.4. Shape-Preserving Failures and Correctness Invariants

Tensor shape is not proof that the quantized operator is correct.

The most dangerous quantization failures are shape-preserving. A packed tensor can have the expected rank. A layer can return the expected output width. A batch can keep the expected row order while the runtime applies the wrong scale group, zero point, format tag, or kernel layout. The correctness problem

is therefore semantic: each packed value must be interpreted with the same metadata account that produced it.

For a hand-computable INT4 group, suppose eight stored weights are packed into four bytes and the first group has scale $s = 0.25$ and zero point $z = 8$. The nibbles $[8, 10, 7, 12]$ decode to

$$0.25 \cdot ([8, 10, 7, 12] - 8) = [0, 0.5, -0.25, 1.0].$$

The next four nibbles may share that same scale if the group size is eight, or use a different scale if the layout uses group size four. A packed buffer with the same byte length therefore has different numerical meaning under different group metadata. This is the small example behind SGLang's insistence that `qweight`, scales, zero points, packing factor, and kernel method travel together.

Invariant 19.4.1 (Quantization step-alignment)**.** *Dequantization must interpret every packed value with the same scale, zero point, group size, and layout used when the checkpoint was produced.*

Example 19.4.1 (Same bytes, wrong scale group) Suppose a checkpoint stores one row as two four-value INT4 groups, with scales $s_0 = 0.25$ and $s_1 = 0.5$. The serving loader can still allocate the same four-byte packed buffer if it mistakenly treats the row as one eight-value group. Tensor rank, byte count, and output width all pass, but the second half of the row is decoded with s_0 instead of s_1. The bug is a step-alignment violation because the packed values are interpreted under a different group map from the one that produced the checkpoint.

In the notation of Definition 19.2.3, Invariant 19.4.1 preserves both the group map ℓ_q and the metadata attached to each group. For any encoded index i, serving-time execution must use the metadata assigned to group $\ell_q(i)$; rank, byte count, and output width are only shape guards.

The rule is more useful than a list of function names. It says that dynamic batching may concatenate activation rows, CUDA graph replay may pad a shape bucket, and a hardware backend may choose a different kernel; none of those choices may change how packed values are decoded. If `group_size`, `zero_point`, a block-shape field, or a swizzled scale tensor is separated from the weight tensor it describes, the result can have the expected output shape while carrying the wrong values. That is why SGLang's format tests and backend-specific weight-processing code belong in the serving contract, not only in an offline conversion script.

Invariant 19.4.2 (Quantized-operator metadata alignment)**.** *A quantized operator is valid only when packed weights, scale and zero-point metadata, registry method selection, backend capability, fallback behavior, and the measured accuracy/performance account all describe the same numerical operation. Symbolically, the served operator must be the declared map*

$$x \mapsto xD_{m,f}(\widehat{W})$$

for the same encoded weights $\widehat{W}$, metadata m, format tag f, and kernel layout κ used by loading, execution, and measurement.

Example 19.4.2 (Accepted bytes, missing format contract) A loader may accept `qweight` and allocate the expected packed tensor while dropping a format tag, KV-cache quantization field, or backend method option. The operator can then dispatch successfully and return a tensor with the right shape, but it no longer computes the declared $xD_{m,f}(\widehat{W})$ map. The bytes were valid only together with the metadata contract that selected their numerical interpretation.

Registry-specific formats add a second invariant: method selection must preserve the non-weight metadata that made the checkpoint valid. A ModelSlim layer-skip rule, a Quark KV-cache quantization setting, a Petit NVFP4 group-size and KV-cache algorithm, an AutoRound bit-width field, or a BlockInt8 MoE scheme is part of the same account as the packed weights. If the loader accepts the bytes and drops those fields, the runtime executes a different numerical contract; tensor allocation with the right shape is not validation. That requirement is source-level rather than cosmetic: registry entry, layer method, scale/KV-cache metadata, and backend capability must be selected as one unit.

19.5. From Checkpoint Metadata to Loaded Layout

Quantization methods bind layer construction to kernel and scale handling.

SGLang's implementation evidence for matrix multiplication under reduced numerical precision starts at a shared quantization-method interface. That interface defines the runtime questions that every scheme must answer: which activation dtypes are supported, what device capability is required, which layer method should be installed, and which weight tensors the method creates. That interface keeps every quantization scheme behind the same layer construction path. AWQ, FP8, compressed-tensors, GGUF, and unquantized execution can still carry their own scale layouts [20, 11].

Example 19.5.1 (GPTQ metadata becoming a loaded Marlin path) A dense GPTQ checkpoint does not become a fast serving path just because its weights are stored in fewer bits. The loader first resolves a GPTQ configuration: bit width, group size, symmetry, activation-order metadata, scale tensors, zero-point tensors, and any g_{idx} grouping state. The quantization method then decides whether the layer can use a direct GPTQ kernel, whether the weights must be repacked into a Marlin-compatible layout, or whether the run must fall back to a less specialized path.
In the successful GPTQ-Marlin case, the loaded layer carries packed weight fields, scale layout, group metadata, sorted or cleared index state, and workspace assumptions into execution. The kernel must consume exactly the representation the loader produced. A tensor with the right shape and a stale scale permutation is only a byte array; it is not a valid loaded layout.

Source-reader note. The lifecycle map below is contributor reference material. First-pass readers only need the runtime questions above: every quantization method answers them before a layer is executable.

Category	Examples	What SGLang must preserve
Quantization algorithm	AWQ, GPTQ, SmoothQuant-style W8A8 recipes	The numerical contract: which values are approximated, which scales and zero points decode them, and which accuracy claims belong to the checkpoint.
Format or toolchain	GGUF, compressed-tensors, ModelOpt, ModelSlim	The producer's metadata contract: tensor names, packed storage, format tags, ignored layers, target schemes, and vendor-specific precision fields.
Kernel family	Marlin, CUTLASS FP8/INT8, GGML GGUF kernels, QServe W4A8	The executable layout: tensor strides, packed dimensions, scale placement, workspace, architecture capability, and shape restrictions.
SGLang-owned boundary	Quantization registry, loader metadata parsing, repacking, `sgl_kernel` ops, fallbacks, and tests	The serving contract: one runtime path that keeps checkpoint semantics, kernel eligibility, batching, MoE routing, tensor-parallel partitions, and measurement aligned.

Table 19.4.: Quantization metadata categories. Each category has examples and a property SGLang must preserve at load time.

Concept-to-code map

The handoff has three code boundaries. The registry in `python/sglang/srt/layers/quantization/__init__.py` and `base_config.py` chooses a config object. Method implementations such as `awq/awq.py`, `gptq.py`, `compressed_tensors/`, `w8a8_int8.py`, and `gguf.py` create packed weights, scales, zero points, group indices, and format tags. The model loader fills those tensors, and Marlin, INT8, FP8, or GGUF kernels consume them only after dtype, group-size, partition, scale-layout, and workspace predicates agree.

The compact method names in this concept-to-code map are quantization families or file formats rather than tensor variables: AWQ, GPTQ, GGUF, Marlin, and compressed-tensors each imply a different packed-weight and scale contract, and W8A8 INT8 means 8-bit weights and 8-bit activations [20, 8, 9, 11, 46].

This table separates the concerns that often get mixed together. The algorithms explain why a low-precision representation should approximate a full-precision layer; the formats and toolchains explain how that representation is serialized; the kernel families explain which layouts can execute quickly; and SGLang's boundary decides whether the three actually meet in a running serving step.

Three of those names play different roles in the serving contract. AWQ is a weight-quantization family whose checkpoint contract preserves low-bit weight groups with scales, zero-point policy, bit width, and group size. GPTQ is a post-training weight-quantization method whose serving state includes packed weights, scale tensors, zero-point tensors, optional group-index metadata, activation-order metadata, and checkpoint-format rules. As Section 19.2 already set up, Marlin is the optimized executable layout and kernel family that may consume compatible AWQ or GPTQ checkpoints; it does not by itself define the checkpoint semantics [20, 8, 9].

That distinction also prevents a false ranking. For a serving deployment, AWQ is often the simpler default when its quality and kernel support are acceptable.

GPTQ is worth choosing when the measured quality of that checkpoint is better and the backend can use a compatible direct GPTQ or GPTQ-Marlin path. The book's rule is therefore not "AWQ versus GPTQ in the abstract," but whether the loaded representation, metadata, and kernel path preserve the intended layer operation efficiently.

The registry tail is part of the runtime contract and carries executable names. Additional registry entries carry vendor- or format-specific restrictions that the common quantization interface still has to preserve. Some entries map NPU-specific quantization scheme names–formats intended for neural processing unit backends such as SGLang's Ascend/ModelSlim path–to linear and MoE schemes; Chapter 29 discusses those backend paths. Other entries install both linear/MoE schemes and a KV-cache method, validate group and KV-cache quantization fields, parse checkpoint metadata, or separate linear and MoE methods. The registry is therefore not a name lookup table; it is a dispatch contract whose entries can change layer construction, KV-cache quantization, and backend eligibility even when the high-level request just says "load a quantized model."

Definition 19.5.1 (Quantization-config registry record) *A quantization-config registry record is the load-time object that turns a checkpoint or user quantization name into a layer-construction method. In SGLang, the record has two parts: a registry entry that maps a quantization name to a config subclass, and a config instance that answers capability, activation-dtype, metadata-file, override, and layer-method questions for one loaded model.*

Table 19.5 turns the lifecycle into the concrete states used below.

For a concrete AWQ checkpoint, the lifecycle is: the registry returns `AWQConfig`, the config parser reads group size, bit width, and zero-point policy, the capability and activation-dtype checks gate the backend, and `get_quant_method` installs a linear or MoE method for the layer prefix. For a GPTQ-Marlin path, the same record has to preserve GPTQ metadata while proving that the Marlin layout is legal for that layer. The registry key therefore starts the decision, but the config instance carries the state that keeps the loaded tensors and executable kernel in the same account.

The main state variable is scale layout. In the AWQ path, the method parses group size, zero-point policy, and bit width, derives the integer packing factor, and selects a linear or mixture-of-experts (MoE) method. In the GPTQ path, the method parses bit width, group size, activation-order metadata, dynamic per-module overrides, and the checkpoint format before a layer receives packed weights, optional group indices, zero-point tensors, and scale tensors. In the compressed-tensors path, the method carries the compression format, target scheme, ignored layers, and packed-module mapping used to choose linear and MoE schemes. In the FP8 path, scale tensors are reshaped, padded, or converted between per-tensor, per-block, and hardware-specific layouts before the MoE or GEMM runner receives them. In the GGUF path, the format tag selects how packed bytes become a matmul. These are not cosmetic fields: they are the data that decides whether $D_m(\widehat{W})$ means the same operation as the checkpoint producer intended.

W8A8 INT8 is a second scale-layout contract, distinct from serialized AWQ or FP8 weight loading. The INT8 method installs packed weight tensors and scale tensors, then calls either a scaled-matmul kernel or a fused quantize-and-matmul

Config surface	Source fields or methods	Why the loader needs it
Registry key	`BASE_QUANTIZATION_METHODS`, `CPU_QUANTIZATION_METHODS`, and `get_quantization_config`.	Selects the config class, rejects unsupported CPU choices, and keeps a user or checkpoint name from becoming an unvalidated backend path.
Checkpoint parser	`get_config_filenames`, `from_config`, `get_from_keys`, and `override_quantization_method`.	Finds producer metadata, normalizes aliases, and permits exceptional overrides such as a generic ModelOpt name resolving to a concrete FP8 or FP4 method.
Capability contract	`get_supported_act_dtypes` and `get_min_capability`.	Checks that the activation dtype and device capability can execute the selected method before layer weights are materialized.
Layer method factory	`get_quant_method` and `packed_modules_mapping`.	Installs the linear, MoE, or KV-cache quantization method for a layer prefix and preserves packed-module grouping created while model layers initialize.
Post-load interpretation	`create_weights`, `process_weights_after_loading`, and `apply` on the quantize method.	Creates packed tensors, performs any layout repair after checkpoint loading, and defines the execution call that consumes those tensors.

Table 19.5.: The quantization config object is a lifecycle record. The registry chooses a class; the config parses checkpoint metadata; the layer method creates and later applies the packed state.

path when activation quantization is performed at runtime. The kernel tests compare scaled INT8 GEMM against a dequantized reference, and the per-token group quantization tests check both row-major and column-major scale layouts. That means a correct INT8 path has to keep the activation scale layout, weight scale layout, bias, output dtype, and kernel capability branch aligned [46].

AWQ-Marlin is the AWQ instance of the kernel-layout contract from Section 19.2. The runtime may override an AWQ checkpoint to a Marlin-backed path only after checking layer shape and group-size compatibility; the materialized packed weights, zero-point tensors, and scales are then evidence that the optimized kernel will unpack the same integer groups the checkpoint encoded.

The compact map above shows where those contracts enter SGLang; this table focuses on the loader contract: what state is stored and how SGLang interprets it at runtime.

GPTQ-Marlin follows the same Marlin contract for GPTQ state: compatibility checks precede workspace allocation, group-index handling, weight repacking, scale permutation, and the Marlin linear call. Ordinary GPTQ keeps the GPTQ GEMM path and its own shuffle and zero-point conventions.

Marlin is the concrete loading example of a kernel layout. The problem it solves is the gap between a compact low-bit checkpoint and a fast serving GEMM. A generic path can give back much of the benefit of a GPTQ or AWQ checkpoint by

Table 19.6.: Stored quantization state and runtime interpretation. Each family requires kernels to unpack the same groups the checkpoint encoded.

Family	Stored state	Runtime interpretation
AWQ / AWQ-Marlin	INT4 packed weights, zero-point tensors, and scale tensors.	Group size, zero-point policy, and packing factor choose ordinary, CPU, NPU, MoE, or Marlin-backed methods.
GPTQ / GPTQ-Marlin	Packed weights, zero-point tensors, scale tensors, and optional group-index metadata.	Activation-order metadata, dynamic module rules, checkpoint format, and Marlin compatibility decide whether to use a direct GPTQ GEMM path or repack into a Marlin layout.
FP8 / block FP8	FP8 weights plus per-tensor, per-block, or hardware-specific scale tensors.	Serialized versus online quantization changes which scales are allocated, reshaped, padded, or converted before GEMM.
GGUF	GGML-format tensor bytes plus a `qweight_type` tag.	The tag selects unquantized matmul, MMVQ, MMQ, or dequantize-and-matmul fallback for linear, embedding, and MoE paths.

using the scales, zero points, and optional group-index metadata to expand four-bit weights into a dense FP16/BF16 matrix, or by feeding the GEMM through an unpacking convention that does not match the kernel's tile schedule. Marlin makes the compact representation executable: the loader checks the quantization type, group size, zero-point convention, activation-order metadata, device capability, and layer shape; then it repacks weights, permutes scale or zero-point metadata, and allocates the workspace expected by the CUDA kernel [9, 18].

The speed comes from keeping the hot path in the kernel layout. Instead of materializing a dense weight matrix before matmul, the Marlin path consumes packed low-bit weights in the order its GEMM kernel expects, applies the per-group scale and zero-point metadata inside the arithmetic path, and avoids a dequantized fallback when the shape is supported. In SGLang, that means Marlin is available only after compatibility checks pass. For the dense GPTQ-Marlin path, the checkpoint must be GPTQ, use a supported symmetric bit-width pair, run on a CUDA device with Marlin support, use a supported group size such as $-1, 32, 64, 128$, and satisfy the partition-shape constraints: output partitions aligned to the kernel's N tile, input partitions aligned to its K tile, and groupwise input partitions divisible by the group size. AWQ-Marlin and MoE Marlin paths use the same idea but add their own zero-point, expert-shape, and runner constraints.

This is a performance contract with conditional speed implications. If the contract fails, the correct behavior is to reject the Marlin path for that layer or route to a less-specialized method. Such a fallback can erase the reason for using the packed layout: if execution expands the packed weights to FP16/BF16 before matmul, the hot path reads the larger dense matrix and may also pay conversion, layout, or startup overhead. The realized serving path can therefore be slower than a simple expanded-weight baseline, especially for small batches or cold starts.

Example 19.5.2 (Marlin eligibility and byte traffic) Consider a GPTQ linear layer with logical weight shape $[K, N] = [4096, 4096]$, four-bit weights, and group size 128. The packed weights alone occupy, using $b_{\text{elt}} = 4/8$ bytes per encoded INT4 value,

$$4096 \cdot 4096 \cdot 4/8 = 8\,\text{MiB},$$

before scale and zero-point metadata. Expanding the same logical weights to FP16, where $b_{\text{elt}} = 2$, occupies

$$4096 \cdot 4096 \cdot 2 = 32\,\text{MiB}.$$

Without a tensor-parallel split, the validated output dimension is the full N. This layer satisfies the dense GPTQ-Marlin shape checks: $N = 4096$ is divisible by 64, $K = 4096$ is divisible by 128, and K is divisible by the group size 128. The Marlin path can therefore keep the GEMM on the packed layout and avoid materializing the 32 MiB dense matrix.
Now consider a different GPTQ layer with output width $N' = 4064$. The dense GPTQ-Marlin shape contract fails because 4064 is not divisible by 64. SGLang must choose another valid path. If that path expands the packed weights, the matmul works over a dense FP16/BF16 representation rather than the compact four-bit representation, before counting conversion and metadata overhead. The Marlin name is not what makes the operation fast; the satisfied kernel-layout contract is. The later parallelism taxonomy chapter introduces tensor parallelism; with tensor parallelism, SGLang applies the same output-width contract to the rank-local output partition N_{part}, so N_{part} must be divisible by 64, so the full N is insufficient by itself.

GGUF is the file-format case of the format-tag contract from Section 19.2. The loader maps tensor names, the iterator emits packed tensors with `qweight_type`, and the installed methods keep that tag with the bytes so Section 19.6 can select MMVQ, MMQ, unquantized matmul, or dequantized fallback. In SGLang terms, `qweight_type` is the loaded GGML quantization-type tag for one GGUF packed tensor. It is not a scale tensor; it selects block geometry, dequantization shape, and the legal packed-kernel branch for that tensor.

For MoE layers, the same tag discipline applies to stacked expert weights. SGLang treats the fused gate-and-up expert projection as one packed object: it concatenates the expert's gate projection and up projection, both of which read the token hidden state. The down projection maps the gated intermediate state back to the model hidden size. A GGUF MoE method therefore stores packed expert weights together with their quantization-type tags, while routing metadata selects and combines the experts for each token. The fast MoE kernels are the same idea as the linear GGUF paths applied after routing: they consume packed GGUF expert tensors when the tag and batch shape support the SGLang GGUF MoE ops, otherwise the fallback loops over selected experts and uses the ordinary GGUF linear dispatch.

The FP8, GPTQ, Marlin, and GGUF tests are evidence rather than structure: they pin numeric equivalence, layout permutation, fused MoE handling, and format-aware loading to reference behavior.

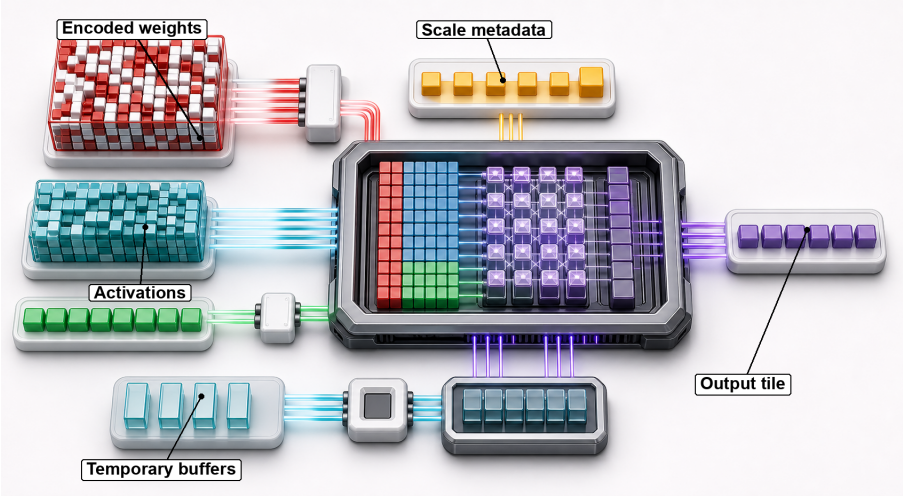

Figure 19.2.: Quantized kernel reconstruction path. A quantized GEMM kernel consuming encoded weights, scale metadata, activations, and temporary buffers to produce an output row on an accelerator.

Loading, Repacking, and Layout Normalization

Loading a quantized checkpoint is a reconstruction path: method metadata and packed tensors must become the layout expected by the serving kernel. Figure 19.2 puts that reconstruction path at the boundary between encoded weights, scale metadata, runtime activations, temporary buffers, and the output row.

A quantized layer is already committed before the serving step reaches GEMM.

A serving step sees a quantized projection only after layer construction has already committed to a method. During model loading, the layer asks its quantization configuration for a method, and that method fixes the stored-weight contract used by later decode or prefill work.

Loaded-layer contract anchors

- **Parameter creation.** `python/sglang/srt/layers/quantization/create_weights` implementations create parameters such as `qweight`, zero points, scales, block-scale tensors, or format-specific packed buffers.
- **Checkpoint fill.** `python/sglang/srt/model_loader/loader.py` and `python/sglang/srt/model_loader/weight_utils.py` fill those parameters according to the same naming and shape conventions.
- **Layout normalization.** `python/sglang/srt/layers/quantization/gguf.py`, `python/sglang/srt/layers/quantization/fp8.py`, `python/sglang/srt/layers/quantization/fp8_utils.py`, and Marlin helpers under `python/sglang/srt/layers/quantization/` reshape, pad, repack, or tag stored tensors before execution.
- **Execution handoff.** The installed linear or MoE method owns the later GEMM path, so the scheduler supplies activation rows rather than reselecting the quantization algorithm during decode.

In the running INT4 example, AWQ parses the group size and zero-point setting, computes a packing factor from the number of weight bits, and selects an AWQ linear method. The Marlin variant additionally checks whether the layer shape and group size are supported; unsupported layers fall back to the unoptimized AWQ path rather than silently reinterpreting the packed data. In an FP8 example, the scale layout can be per tensor, per block, or hardware specific, and SGLang reshapes or pads scale tensors before passing them to GEMM or MoE runners.

Algorithm 1: Quantized operator execution path. Packed weights are executable only after the method, scale layout, kernel, and activation shape agree on how to interpret the stored representation.

Input: Checkpoint tensors; quantization config; target layer shape $[K, N]$; hardware backend
Output: Executable quantized operator for one layer

choose the quantization method named by the checkpoint and runtime config;
recover $\widehat{W}$ and metadata $m = (\ell_q, s, z, p, \eta, \delta)$;
validate that ℓ_q covers the encoded tensor indices and that scale, zero-point, packing, and format-tag fields are present for every required group;
if *the method requires a specialized kernel layout* κ **then**
    check the method's group-size, dtype, device, K-partition, and N-partition predicates;
load or repack $\widehat{W}$ into the layout expected by the selected kernel;
select a compatible matmul or fused-dequantization path for the current device, dtype, and batch shape, otherwise select a correctness-preserving fallback or reject the layer;
foreach *forward call with activation* x **do**
    interpret packed weights using the matching metadata m;
    compute $y = xD_m(\widehat{W})$ through the selected kernel path;
return *the layer output and any persistent kernel metadata*;

Both examples expose the same systems rule: feasibility is a question of tensor shape and of whether the backend understands the stored layout.

GPTQ follows the same rule but carries packed weights, group-index metadata, zero-point data, scales, optional shuffle state, and bit width. Loading must produce the exact layout expected by the selected GEMM or Marlin path; ordinary GPTQ and fused-MoE GPTQ take different backend routes. That split is part of the serving contract with naming as only one surface.

For Marlin-compatible AWQ or GPTQ, loading applies the kernel-layout commitment introduced in Section 19.2. Marlin is the executable layout for the optimized kernel path. AWQ or GPTQ still define the checkpoint semantics. The Marlin utilities check supported quantization types, group sizes, partition multiples, and workspace needs before the optimized kernel is installed. GPTQ-Marlin then repacks weights, permutes scales, and handles optional activation-order metadata; AWQ-Marlin creates packed weights, zero-point data, and scales in the layout consumed by the AWQ Marlin kernel.

The local loading map above names the GGUF and FP8 anchors. The important structural point is that both pass through the same layer-construction interface before the serving step reaches GEMM, even though one begins from a file-format tag and the other from scale-layout metadata.

GGUF adds a file-format step before layer construction. The GGUF loader maps file-local tensor names back to the model's parameter names, updates tied-output-embedding state when the mapped file implies it, and feeds a quantized-weight iterator into normal model loading. The iterator emits quantization-type tags before the packed tensors and splits packed MoE expert tensors into the expert parameter names expected by SGLang. After that, the GGUF configuration installs linear, embedding, or MoE methods in the same layer-construction interface as other quantization schemes.

The algorithm below keeps the operator notation from Definition 19.2.2. The scale values s are one component of the full metadata tuple m, not a synonym for all metadata.

Algorithm 1 executes a quantized operator by validating the metadata tuple that defines D_m, checking any specialized kernel-layout predicates, and falling back or rejecting before the runtime can reinterpret packed bytes under a different scale layout. First, the loader recovers the packed tensor and every scale, zero-point, format, and packing field needed to interpret it. Then the runtime checks the selected kernel's shape, dtype, device, and partition predicates before repacking or selecting a fallback. Finally, each forward call interprets the packed bytes through the same metadata record, so the output implements $xD_m(\widehat{W})$ rather than a dense matmul with guessed scales.

19.6. Kernel Dispatch, Eligibility, and Fallbacks

First-read note. This section has a core serving argument and a source-reference layer. On a first pass, keep the serving question in view: which kernel computes the same $xD_m(\widehat{W})$ operator for this packed representation? Hardware and kernel family details are source-reference material unless they change what SGLang can run or how it interprets scales, output dtype, and slow-path behavior.

Kernel dispatch is where packed representation becomes an executable serving decision. By this point SGLang has already parsed checkpoint metadata, allocated packed weights, attached scale tensors, and chosen a quantization method. The next question is narrower and more dangerous: does the current activation shape and device match a runnable kernel? Does that kernel understand the scale layout, packed-weight convention, and slow-path policy for the same logical operator? For the quantized operator of Definition 19.2.2, dispatch is the point where SGLang chooses the backend implementation of D_m without changing the operation that the packed weights and metadata declare.

Eligibility is a safety gate before a quantized kernel can claim a speed path.

SGLang owns that boundary through the kernel extension. The exported operators fall into a few serving roles: dequantization, scaled GEMM, packed matmul, per-token activation quantization, and grouped-MoE execution. These operators are not a new quantization theory. They are the serving-time ABI that lets the loader, quantization registry, scheduler, MoE runner, and tests agree about the same packed representation.

Why SGLang Owns This Kernel Boundary.

Using an off-the-shelf kernel family is not the same as using an off-the-shelf serving path. NVIDIA libraries, TensorRT-LLM kernels, vLLM kernels, ExLlama GPTQ kernels, Marlin kernels, and llama.cpp GGUF kernels provide valuable building blocks, and SGLang explicitly adapts some of them. But SGLang still needs its own operator boundary because the kernel must fit SGLang's batching shape, tensor-parallel shard layout, CUDA graph constraints, MoE routing, checkpoint loader, device matrix, and fallback policy. Depending on another engine's private extension module would couple SGLang to that engine's build flags, release cadence, ABI assumptions, and internal tensor layouts.

The result is a practical kind of customization. SGLang vendors, adapts, wraps, and specializes kernels so they can be called from the same quantized layer methods that loaded the checkpoint. The originality is mostly at the systems

boundary: compatibility checks, repacking, scale permutation, shape validation, operator registration, runtime dispatch, backend fallback, and test coverage. That boundary is why a checkpoint produced by a separate toolchain can become a SGLang serving operator rather than merely a file that contains smaller tensors.

The GGUF execution branch keeps the format tag on the hot path. Given a GGUF format tag τ_q and an activation row count M, dispatch chooses a kernel layout κ only when that layout can decode the packed bytes under the same block convention. Unquantized GGUF types use ordinary matmul, small contiguous batches can use the MMVQ vector kernel, standard and K-quant types can use the MMQ matrix kernel, and other supported dequantization types fall back to a dense matmul after unpacking. Unsupported tags raise an error rather than guessing. SGLang enters this branch through `fused_mul_mat_gguf`; its fallback helper is `ggml_dequantize`. The GGUF tests compare dequantization, matvec, and matmul with the reference `gguf.dequantize` interpretation, so the test surface checks both τ_q and κ.

In this chapter, MMVQ, MMQ, and dequant name SGLang execution paths, not three different GGUF checkpoint formats. MMVQ is the small-row, matrix-vector-style quantized path exposed as `ggml_mul_mat_vec_a8`; it is the path SGLang tries for decode-like batches when the quantization tag supports it. MMQ is the matrix-matrix quantized path exposed as `ggml_mul_mat_a8`; it is the batched packed-weight path for supported standard and K-quant GGUF types. Dequant means that SGLang first expands the packed GGUF tensor with `ggml_dequantize` and then runs ordinary dense matmul. These operators are registered SGLang PyTorch custom ops over `torch.Tensor` inputs. Their packed formats and device functions are GGML-compatible, and the implementation is adapted from GGML-family CUDA kernels, but the serving path does not enter the full ggml graph runtime.

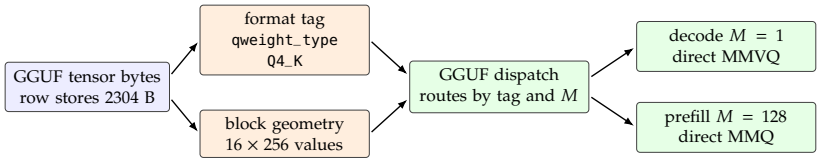

Figure 19.3.: A GGUF format tag turns packed bytes into an executable tensor. The same stored row length can have multiple plausible decoders, so SGLang keeps the `qweight_type` beside the `qweight`; the dispatch path then depends on both the tag and the current activation row count.

Figure 19.3 shows the dispatch split that the next example instantiates: the same stored row bytes and τ_q can feed decode-like MMVQ, larger MMQ, or a fallback path depending on the current activation row count.

Example: GGUF `Q4_K` tag and dispatch. Consider one GGUF linear weight tensor with logical shape $[K, N] = [4096, 4096]$ and a per-tensor format tag τ_q = `Q4_K`. That tag stores 256 logical weights per block. In SGLang's GGUF kernel headers, one such K-quant block contains a four-byte super-block scale/min pair, 12 bytes of quantized subscale metadata, and 128 bytes of four-bit codes, for 144 bytes total. A GGUF storage slice for one output channel and then the full tensor therefore store

$$(4096/256) \cdot 144 = 2304 \text{ bytes per output channel},$$
$$4096 \cdot 2304 = 9{,}437{,}184 \text{ bytes} = 9 \text{ MiB}.$$

For comparison, the same logical FP16 tensor would store the following byte count:

$$4096 \cdot 4096 \cdot 2 = 32\,\text{MiB}.$$

The 9 MiB number includes "four-bit weights" plus the block metadata that makes the bytes decodable. It is also not self-describing: the same row byte count can describe `Q4_0`, whose scale layout and block boundary apply every 32 values. The τ_q = `Q4_K` tag tells SGLang to use the 256-value K-quant block interpretation rather than a numerically different decoder with the same row byte count.

The same tag also decides the serving path. For this $N = 4096$ tensor, SGLang's non-I-matrix GGUF rule treats activation row counts $M \le 6$ as safe for the matrix-vector path. A decode-like call with $M = 1$ can therefore use the direct packed `ggml_mul_mat_vec_a8` path. A prefill call with $M = 128$ is too large for that matrix-vector branch, but τ_q = `Q4_K` belongs to the standard K-quant set that has matrix-matrix coverage, so SGLang can use `ggml_mul_mat_a8`. If the tag instead named a type with no matrix-matrix coverage, the same $M = 128$ shape would fall back to dequantizing the packed tensor and running ordinary dense matmul.

Example 19.6.1 (One `Q4_K` block inside a GEMM tile) Keep the same τ_q = `Q4_K` tensor, but zoom into one K-block for one output column. The block's 144 bytes are read as three regions: the four-byte super-block scale/min record, the twelve bytes of subscale metadata, and bytes 16:143, which hold 256 four-bit codes. A tile that needs logical weights $w_{0:15}$ reads the first eight payload bytes. For payload byte b_j, the low nibble gives $q_{2\cdot j}$ and the high nibble gives $q_{2\cdot j+1}$ under the block's layout convention.
That tag splits the 256 logical weights into eight 32-value subblocks. Logical indices 0:31 use subscale entry 0, indices 32:63 use subscale entry 1, and so on. A direct MMVQ or MMQ kernel can therefore accumulate without first materializing a dense row:

$$\text{acc}_n \leftarrow \text{acc}_n + x_k\, D_{\text{Q4_K}}\big(q_k, \text{super}, \text{subscale}_{\lfloor k/32 \rfloor}\big).$$

For a concrete first pair, suppose the first payload byte is `0xA3`, so $q_0 = 3$ and $q_1 = 10$, and suppose the decoded subblock metadata gives effective scale $s_0 = 0.0625$ and minimum offset $\mu_0 = -0.50$. The reconstructed weights consumed by the tile are

$$\widetilde{w}_0 = \mu_0 + s_0\, q_0 = -0.3125, \qquad \widetilde{w}_1 = \mu_0 + s_0\, q_1 = 0.125.$$

The kernel multiplies those values by the activation entries x_0 and x_1 while the packed bytes and scale metadata stay in the GEMM path. Reading subscale entry 1 for $k = 0$, or interpreting the high nibble before the low nibble, would keep the same output shape while changing the mathematical operator.

The FP8 path has two ledgers: stored weight format and runtime activation-scale format. The configuration records whether the checkpoint is already FP8-serialized, the activation scheme, optional block size, ignored layers, packed-module mapping, and the MXFP8 flag. MXFP8 means micro-scaled FP8. The payload values are still FP8, but the scale ledger is more constrained: SGLang splits the K dimension into 32-value groups and stores each group's scale as UE8M0,

FP8 form	Scale contract	SGLang consequence
Rowwise or per-tensor FP8	The FP8 payload is paired with coarse scale vectors or scalar scale records.	Dispatch mainly checks dtype, layout, and whether the selected scaled-GEMM backend supports the row and column scale convention.
Blockwise FP8	A rectangular weight or activation block shares scale metadata.	The block-shape validator must prove that tensor-parallel partitions preserve the configured scale groups before a block-scaled GEMM can run.
MXFP8	Each row uses 32-wide K-groups with UE8M0 `uint8` scale bytes; in SGLang the weight block size is fixed to $[1, 32]$.	Loading, activation quantization, and GEMM dispatch become backend-specific: the runtime may reshuffle weights, pack or swizzle scale tensors, and require an MXFP8-capable Triton or FlashInfer path.

Table 19.7.: FP8 scale-storage contracts. Each FP8 form carries a scale contract that changes dispatch eligibility, not layer mathematics.

an unsigned 8-bit exponent-only representation, rather than as an ordinary floating-point scale tensor. Do not read MXFP8 as a different mathematical layer; read it as a hardware-facing scale-storage contract for the same $xD_m(\widehat{W})$ operator.

Block quantization is allowed only for serialized FP8 weights with dynamic activation scales, and the block-shape validator checks tensor-parallel input and output partitions against the configured block dimensions. If weights are not serialized, the linear method creates parameter tensors in the original dtype and quantizes after loading; if they are serialized, it creates FP8 weights plus either per-tensor scales, block-scale inverses, or static input scales. MXFP8 then fixes the block size to $[1, 32]$, reshuffles weights, and can convert scales into the UE8M0 layout consumed by the selected backend before GEMM runs.

The online activation side has its own lifecycle. A BF16 or FP16 activation row cannot enter a W8A8 GEMM merely because the weight is already packed. SGLang must quantize the scheduled activation rows, produce scale metadata in the layout expected by the selected GEMM backend, and keep both tensors aligned with the same M rows. The per-token kernel `sgl_per_token_quant_fp8` emits one scale per row, while `sgl_per_token_group_quant_8bit` and its v2 path emit per-group scales for FP8 or INT8-style payloads. In MoE paths the same idea can run after expert-major permutation or after a fused activation such as `silu_and_mul`; the scale tensor is then part of the runner input, not a temporary diagnostic.

Before reading the dispatch table. Follow one loaded method: checkpoint metadata names the quantization family, loading or repacking creates the packed payload and scale layout, the eligibility predicate checks device and dtype support, and only then does dispatch pick a kernel or fallback. A packed checkpoint without a matching activation-scale record is not eligible work; it is a fallback case with a useful diagnostic.

How the FP8 and INT8 Activation Quantizer Maps to CUDA.

The online activation quantizer is a small kernel with a large contract. It does not choose the model's quantization method, and it does not decide which GEMM algorithm is numerically acceptable. Its job is narrower: for the activation rows

Table 19.8.: Quantized-kernel dispatch lifecycle. Each stage preserves a packed payload, scale layout, or fallback diagnostic.

Lifecycle stage	Kernel-facing object	Contract to preserve
Scheduled activation rows	BF16/FP16 $X \in \mathbb{R}^{M \times K}$ from the current batch.	M still names request rows or routed expert rows.
Quantized payload	`sgl_per_token_quant_fp8` or `sgl_per_token_group_quant_8bit` output X_q.	Each quantized row or group corresponds to the same source row span.
Scale record	Per-row, per-group, column-major, TMA-aligned, or UE8M0 scale layout.	The consumer GEMM must interpret scale addresses with the same grouping.
Scaled GEMM consumer	`fp8_scaled_mm`, `fp8_blockwise_scaled_mm`, DeepGEMM, FlashInfer, or Triton fallback.	The output slice returns to the original scheduled rows.

already selected by the scheduler or by an expert runner, produce a low-precision payload and a scale record whose layout the next GEMM can consume. That is why SGLang exposes both per-token FP8 and grouped INT8 quantization at the kernel extension boundary instead of treating activation quantization as an ordinary PyTorch cast.

For an activation matrix $X \in \mathbb{R}^{M \times K}$ and group size g, the grouped kernel views row i's hidden dimension as groups

$$G_{i,r} = X_{i,rg:(r+1)g}.$$

For each group it computes

$$a_{i,r} = \max(\epsilon, \max_{x \in G_{i,r}} |x|), \qquad s_{i,r} = \frac{a_{i,r}}{q_{\max}}, \qquad \widehat{x} = \operatorname{clip}\left(\frac{x}{s_{i,r}}, q_{\min}, q_{\max}\right).$$

The same source-level shape is used for FP8 and INT8 payloads: FP8 uses the E4M3 finite range, where the byte has four exponent bits and three mantissa bits, while INT8 uses the signed integer range. The destination tensor's dtype and the $(q_{\min}, q_{\max})$ arguments select the concrete specialization. The scale $s_{i,r}$ is not a debug byproduct. It is part of the matrix product: the downstream GEMM reconstructs the intended product by applying the activation scale and the weight scale with the same row and group ownership that produced $\widehat{x}$.

The serving contract follows which activation row or row-group owns the maximum and scale record.

The CUDA mapping follows directly from that equation. The first pass over a token group loads a vectorized slice of BF16 or FP16 activations and reduces the absolute maximum across the threads assigned to the group. The reduction is a warp or subwarp shuffle reduction rather than a global-memory reduction, because the scale belongs only to that row-group. One lane writes the scale record. A second pass divides by the scale, clamps to the destination range, converts to FP8 or INT8, and stores the packed output. This two-pass structure is the central implementation fact: the kernel cannot emit the quantized bytes until it has seen the group's maximum, and it cannot let another row or expert share that maximum without changing the GEMM's dequantization ledger.

`sgl_per_token_quant_fp8` is the rowwise version of the same idea. It emits one scale per activation row for W8A8 FP8 dense paths. Vectorization, warp-local reductions, and block-level staging are performance choices inside that ownership rule; they preserve the same formula and change only how the row's maximum and quantized bytes are produced efficiently.

`sgl_per_token_group_quant_8bit` generalizes the rowwise case to grouped scales and to both FP8 and INT8 payloads. Its v2 path is the important serving path for group sizes 16, 32, 64, and 128. The v2 kernel lets the scale tensor be written row-major, column-major, TMA-aligned, or in the UE8M0 encoded scale form expected by newer block-scale paths. It can also run over a masked expert-major layout, where only the routed rows for each expert are valid, and it can fuse the SwiGLU-style `silu_and_mul` operation before quantization. In that fused case the input activation has twice the output hidden width, the kernel first forms $\mathrm{SiLU}(a) \odot b$, and the scale record belongs to the post-activation tensor that the down-projection GEMM will consume.

The correctness failures mirror the invariants in Section 19.4. A wrong group size can still produce an $M \times K$ payload. A row-major scale tensor can have the same number of elements as a column-major or TMA-aligned scale tensor. A masked MoE batch can quantize padded rows unless the masked-row record gates the expert-major layout. A fused activation path can produce the right output width while scaling the pre-activation tensor instead of the post-activation tensor. The custom kernel boundary exists so these choices are explicit before the rowwise FP8, blockwise FP8, DeepGEMM, FlashInfer, or INT8 GEMM consumer sees the payload.

This is why activation quantization belongs in the kernel-dispatch account. It does not change the model's linear operator, but it creates the packed input and scale ledger without which the scaled GEMM would multiply the right shape under the wrong numeric interpretation.

Source-reader eligibility trace. The executable FP8 GEMM path adds one more runtime contract. The FP8 scaled-matrix-multiply kernel in `fp8_gemm_kernel.cu` first checks CUDA residency, two-dimensional matrix inputs, row-major A, column-major B, 16-byte alignment, FP8 E4M3 operands, float32 scale vectors, matching scale lengths, and a half or bfloat16 output dtype. Only after those checks does it branch by compute capability to SM89, SM90, SM100, or SM120 launch families. These SM labels name NVIDIA streaming-multiprocessor architecture targets, not streaming-multiprocessor counts: read SM89 as an Ada-generation target, SM90 as Hopper-generation, and SM100/SM120 as Blackwell-generation target families when matching kernels to hardware. The selected family then chooses tile shapes from the output dtype, optional bias, and M, N shape bucket. Thus the FP8 scale lifecycle does not end at loading a valid checkpoint tensor: it ends at a dispatchable tuple of layout, scales, output dtype, architecture generation, and shape bucket.

The implementation path is the same for the dense scaled-GEMM family. A layer method or FP8 utility calls the Python GEMM wrapper; the wrapper enters the registered torch-operator schema; the CUDA entrypoint checks tensor layout, dtype, scale shape, alignment, and bias; the architecture dispatch helper selects the CUTLASS template family; and the focused tests compare the result with a dequantized or scaled-matrix reference. The blockwise FP8 trace adds padded

Algorithm 2: GGUF kernel dispatch from a packed format tag. The runtime keeps $qweight_type$ in the operator path so unsupported or shape-mismatched tags fail before an incorrect dense interpretation is used.

Input: Activation matrix X; GGUF stored weight $\widehat{W}$; metadata m with tag τ_q
Output: Linear output Y, or an unsupported-tag error

if *X has zero rows* **then**
  return *an empty output with the inferred output width*;
if *τ_q is an unquantized GGUF type* **then**
  return *ordinary matmul using* $XD_{m,\tau_q}(\widehat{W})$;
compute the MMVQ row threshold from $\widehat{W}$'s shape and imatrix metadata;
if *X's row count is within the MMVQ threshold and τ_q supports MMVQ* **then**
  return *the SGLang GGUF vector op result*;
else if *τ_q supports MMQ* **then**
  return *the SGLang GGUF matrix op result*;
else if *τ_q supports explicit dequantization* **then**
  recover the dense shape, dequantize $\widehat{W}$, and **return** $XD_{m,\tau_q}(\widehat{W})$;
else
  raise an unsupported GGUF quantization-tag error;

activation rows plus a slice back to the original M. The INT8 trace preserves the row-scale and column-scale convention. This is the implementation detail that affects the quantization contract; lower-level CUTLASS template composition matters here only when it changes an eligibility check, scale layout, output dtype, or fallback decision visible to SGLang.

Reference algorithm. The GGUF branch listing below is useful when reading the dispatch source. The first-read takeaway is the same as the prose above: `qweight_type` and activation row count select a legal branch, and unsupported tags are rejected instead of guessed.

Algorithm 2 dispatches GGUF quantized kernels by reading the packed-format tag, validating tensor metadata, and selecting the matching implementation. It first handles empty inputs and unquantized GGUF tags. It then chooses the vector or matrix GGML kernel from row count and type support. If no native quantized kernel is suitable, it explicitly dequantizes before matmul; any remaining unsupported tag is rejected instead of being guessed.

Algorithm 2 separates GGUF loading from GGUF execution. The loader maps names and installs tagged packed weights; the execution path still has to choose between unquantized matmul, vector quantized matmul, matrix quantized matmul, and dequantize-then-matmul according to the tag and batch shape.

Example 19.6.2 (Dispatch fallback is still a correctness decision) A GGUF tensor with a supported `qweight_type` may miss the fast path for the current row count or quantization family. In that case, the safe outcome is not to reinterpret the packed buffer as a nearby type; it is to run the explicit dequantize-then-matmul branch or raise the unsupported-tag error. The output shape would be identical in all three cases, so the observable difference is the metadata contract: the selected branch must consume the same tag, block size, and dense shape that the loader attached to the tensor.

19.7. Path Classification and Resource Tradeoffs

Quantization trades precision, memory footprint, and kernel overhead. A smaller representation helps only when its accuracy remains acceptable and its conversion or fallback costs do not erase the memory savings.

This section follows the correctness invariants because preserving the numerical meaning of packed values is only the first requirement. The next serving issue is whether the SGLang execution path for that representation preserves the memory benefit. Figure 19.4 frames this as a trade among precision, storage bytes, metadata overhead, and transfer savings.

Smaller weights help only when conversion and kernel support do not dominate.

Quantization sounds like one simple move: store each value in fewer bits. The serving question is more specific: given a real checkpoint, what SGLang path does it actually become, and does that path preserve the intended benefit? This section answers that question through four case studies: a dense GPTQ checkpoint that becomes a GPTQ-Marlin path after SGLang's metadata and shape checks, a compressed-tensors MoE checkpoint that must land on fused expert kernels, a GGUF checkpoint whose dispatch depends on a per-tensor format tag and the current serving shape, and a quantized KV-cache run whose resident-state savings depend on cache-pool layout and attention-backend support.

These are measurement designs, not completed result tables. The completed public slice for Case 1 is reported in Section 19.8; Cases 2–4 define the path-classification and measurement records an operator would need before making the same kind of claim.

The model provider or serving operator owns the deployment decision: whether to serve a quantized checkpoint or a floating-point FP16/BF16 checkpoint at all. SGLang's decision happens after that checkpoint or explicit quantization choice is in play. It parses metadata, validates device and shape constraints, normalizes or repacks tensors when required, and dispatches a compatible kernel or fallback. The reader's job in this section is to classify that realized path first, then measure it against the relevant baseline.

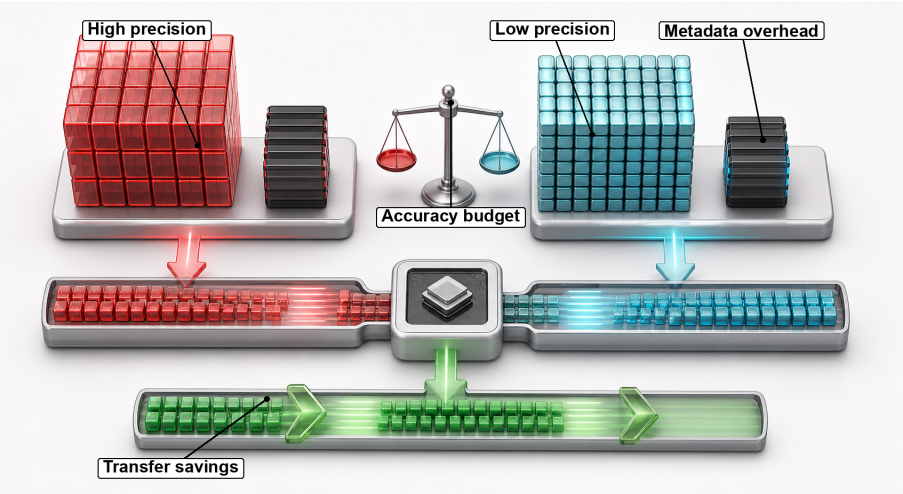

Figure 19.4.: Precision, memory, and overhead tradeoff. Comparison of high-precision storage, low-precision storage, metadata overhead, and transfer savings as balanced geometric blocks, with the accuracy budget preserved as a separate deployment constraint.

Runtime validation is narrower than experimental validation. SGLang rejects an unsupported configuration or dispatches a fallback, but the normal serving path does not prove that the quantized representation is accurate or faster for a workload. The SGLang repository provides separate engineering-facing harnesses for that evidence: contributor and CI tests for correctness, and benchmark commands that developers or operators run deliberately for a target deployment. A speedup that reads the wrong scale group is not a valid optimization. A slower fallback still counts in the benchmark, because it is part of the cost of using that quantized representation. Accuracy is part of the same budget as memory and latency: a smaller packed path that misses the accepted quality tolerance has not preserved the serving contract.

Stage 1 path classes

- *Rejected configuration.* Device capability, dtype, shape, group size, or module mapping fails before a valid quantized runtime method exists; for example, a GPTQ shape that cannot satisfy the selected Marlin predicate is rejected before timing.
- *Direct packed kernel.* The forward path calls a kernel that consumes the packed weights and their metadata directly, such as a GGUF MMQ or MMVQ path selected by the stored `qweight_type`.
- *Load-time repacked specialized kernel.* Loading first rewrites the in-memory layout, such as by repacking weights, permuting scales, sorting or clearing `g_idx`, or allocating workspace. The rewritten layout is for this model instance; SGLang does not write a new checkpoint file.
- *Dequantized fallback.* Execution explicitly expands the packed tensor to a dense weight and then runs ordinary matmul. It gives back part of the memory-traffic advantage and must still count in the benchmark.
- *Quantized resident-state path.* Runtime-generated KV state is stored in FP8 or FP4 cache buffers, with scale metadata and any dequantization workspace counted as part of the serving path.

The SGLang concordance below is deliberately narrower than the measurement protocol. It names the implementation surfaces needed to classify the realized path. The benchmark commands and correctness tests enter later, after the path class is known and the comparison has a baseline.

Path-class SGLang concordance

- **GPTQ dispatch and Marlin repacking.** `python/sglang/srt/layers/quantization/gptq.py` chooses between `sgl_kernel::gptq_gemm` and GPTQ-Marlin conversion; `marlin_utils.py` names scalar types, group sizes, shape predicates, workspace, scale permutation, zero-point conversion, and `g_idx` handling; the packaged repacking path lives under `python/sglang/jit_kernel/gptq_marlin_repack.py` and its generated-kernel C++/CUDA helper.
- **Compressed-tensors ownership.** `python/sglang/srt/layers/quantization/compressed_tensors/` maps compressed-tensors `config_groups` onto the linear or MoE method that owns each quantized layer at runtime.
- **GGUF mapping and dispatch.** `python/sglang/srt/model_loader/loader.py`,

`python/sglang/srt/model_loader/weight_utils.py`, and `python/sglang/srt/layers/quantization/gguf.py` map GGUF tensor names into SGLang parameters and dispatch by `qweight_type`.

- **Quantized KV-cache state.** `model_runner_kv_cache_mixin.py`, `memory_pool.py`, `kv_cache.py`, `fp4_kv_cache_quant_method.py`, and `kvfp4_tensor.py` connect `--kv-cache-dtype` to cache-pool allocation, FP8 scale loading, FP4 packed buffers, and quantized KV writes.

Classify the realized path before comparing speed or memory.

Stage 1 is path classification before timing. The evidence comes from the checkpoint metadata, method selection, loading-time tensor transformations, and the dispatch branch used during the forward call. Stage 2 turns that classification into a measurement record. The reader holds serving conditions fixed, runs the realized quantized path and the baseline under the same conditions, applies the same correctness or task-quality gate to both, and records latency, throughput, memory, and startup costs at matching lifecycle points. SGLang provides the benchmark tools, but the serving operator chooses the baseline, traffic trace, quality tolerance, and deployment policy.

Example 19.7.1 (Case 1: Dense GPTQ checkpoint) Use the Qwen2.5 0.5B Instruct dense model as the floating-point baseline and the GPTQ-Int4 checkpoint of the same model as the quantized variant. The checkpoint is GPTQ, four-bit symmetric, group size 128, has no activation-order metadata, and is not already serialized in Marlin format. The case asks: does SGLang realize this as ordinary GPTQ or GPTQ-Marlin, what load-time conversion is required, and which baseline answers the deployment question?

For the GPTQ case, *group size* is the number of encoded weights that share one scale and zero-point rule; group size 128 means each metadata entry applies to a 128-weight group. *Activation-order metadata* is GPTQ bookkeeping for checkpoints whose packed columns or groups were arranged using activation statistics. Its runtime consequence is `g_idx`, the mapping from input columns to quantization groups. In this case, activation order is absent, so SGLang clears `g_idx` instead of preserving or sorting it.

With Marlin already introduced as a kernel-layout example in Section 19.2, Stage 1 for this GPTQ case is a concrete loading-path audit. The deployment run leaves the quantization override unset, uses tensor parallelism 1, and targets a CUDA GPU whose compute capability reaches Marlin's CUDA path. SGLang checks the GPTQ-Marlin contract as follows [37, 9].

The table completes the path classification. SGLang records the full input size, partitioned input and output sizes, group size, weight type, and whether the layer has `g_idx`; the final row is the same shape contract it validates after the tensors load. The deployment row therefore selects `gptq_marlin`. The forced-GPTQ control run used later in Stage 2 is separate: it sets `--quantizationgptq`, so SGLang leaves the checkpoint on the ordinary GPTQ path and logs that the faster Marlin conversion was suppressed.

After the contract check passes, the operator identifies the one-time in-memory transformation that makes the path different from plain GPTQ. SGLang clears activation-order tensors to empty tensors, because this checkpoint has no

Table 19.9.: SGLang Marlin contract cases. Each case shows how settings pass through path classification and staged dispatch.

SGLang Marlin contract	Case 1 setting	Stage 1 result
The runtime-conversion branch requires a GPTQ checkpoint that is not already tagged as Marlin.	The checkpoint is GPTQ-Int4 and is not serialized in Marlin format.	Runtime-conversion branch selected.
The quantization override must permit Marlin selection.	The deployment run leaves the override unset.	The override gate passes.
The metadata must declare `quant_method=gptq` and the fields `bits`, `group_size`, `sym`, and `desc_act`.	The metadata is GPTQ with `bits=4`, `group_size=128`, `sym=true`, and `desc_act=false`.	The metadata gate passes.
The `bits/sym` pair must map to a GPTQ-Marlin scalar type.	(4, true) maps to SGLang's `uint4b8` weight type.	The weight-type gate passes.
The group size must be one of $-1, 32, 64, 128$.	`group_size=128`.	The group-size gate passes.
The device must expose Marlin's CUDA path and support the selected quantized type.	The run targets a CUDA GPU with compute capability at least 8.0.	The device gate passes.
Marlin's shape check requires output partitions divisible by 64, input partitions divisible by 128, and grouped input partitions divisible by the group size.	With tensor parallelism 1, Qwen2.5 0.5B projection dimensions use 896, 128, and 4864 as rank-local input or output widths; each is divisible by 64 when it appears as N and by 128 when it appears as K.	The layer-shape gate passes.

activation-order metadata. It also uses an empty zero-point tensor for the symmetric case. It then permutes the packed weight layout, calls the JIT Marlin repacker, and permutes scales into the Marlin layout. For this four-bit checkpoint the Marlin repacker uses a pack factor of eight, so the repacked weight is organized as Marlin tiles rather than as the original GPTQ packed matrix. SGLang also allocates a small integer CUDA workspace whose length is based on the GPU's streaming multiprocessor count. The Stage 1 answer is therefore precise: this is a non-Marlin GPTQ checkpoint that SGLang converts at load time into an in-memory GPTQ-Marlin layout, then executes through the Marlin GEMM path [9, 18].

Stage 2 for the same example measures two distinct decisions. The deployment decision compares the floating-point Qwen2.5 0.5B Instruct baseline against the GPTQ-Int4 checkpoint using SGLang's default choice, which Stage 1 says is `gptq_marlin`. The SGLang-method decision compares the same GPTQ-Int4 checkpoint on the default `gptq_marlin` path against the same checkpoint with ordinary GPTQ forced by `--quantizationgptq`. These are different tables: the first asks whether to deploy the quantized checkpoint at all, while the second asks whether SGLang's Marlin conversion preserves or improves the benefit of that checkpoint.

Stage 2 for this case instantiates the measurement record in Section 19.8. The case-specific deltas are the two comparison rows above, the one-time load/repack time for the default `gptq_marlin` path, and the branch choice for the forced `gptq` control. Fixed-shape runs may use a small `sglang.bench_one_batch` grid such as

(1, 256, 128), (8, 1024, 128), and (16, 2048, 128); online runs use the same `sglang.bench_serving` trace for baseline, default GPTQ-Marlin, and forced-GPTQ rows.

The interpretation is also concrete. A default GPTQ-Int4 row that fails the quality gate rejects the deployment decision regardless of memory savings. A passing row with lower memory but a large load-time repack cost separates cold-start cost from steady-state serving cost. A default `gptq_marlin` row with lower memory and better steady-state throughput than floating point on the target trace shows that the quantized serving path has preserved its benefit. A default `gptq_marlin` row that beats the forced `gptq` control shows that the benefit comes from SGLang's Marlin execution path rather than from the checkpoint size alone.

Example 19.7.2 (Case 2: Compressed-tensors MoE checkpoint) Use Qwen3-30B-A3B as the floating-point MoE baseline and Red Hat AI's W4A16 quantized checkpoint of the same model as the quantized variant. The checkpoint uses weight-only compressed-tensors W4A16, group size 128, `pack-quantized` storage, a `Linear` target rule, and ignored router gate modules. The case asks: does a `Linear` target rule reach SGLang's fused MoE runtime, which MoE runner is selected, and what MoE baseline should be measured?

The local MoE vocabulary is narrow. The router chooses experts, and the expert MLP usually stores gate, up, and down projections. A serving kernel often wants those routed rows in expert-major order, meaning rows for the same expert are grouped before the grouped GEMM runs; DeepEP is the expert-parallel path that adds distributed expert dispatch to that layout problem. The MoE chapter gives the full account, but those three facts are enough to read the quantization case here.

Compressed-tensors metadata acts like a routing table. Each `config_groups` entry says which target modules it applies to and which weight or activation scheme those modules should use. In this chapter, `FusedMoE` means SGLang's runtime module for executing a routed MoE layer with stacked expert weights and grouped expert kernels; it does not merge learned experts into one expert. `DeepEPMoE` is the DeepEP-class expert-parallel variant. The DeepEP dispatch and runner-layout sections of Chapter 21 give the full account.

The Stage 1 decision for the MoE case is a load-time compatibility chain. Checkpoint metadata may name ordinary linear layers, while SGLang instantiates the expert block as a fused or DeepEP-class MoE module; when there is no separate fused-MoE rule, SGLang copies the compressed-tensors rule to those fused MoE targets. The W4A16 groupwise scheme selects the WNA16 family because weights use N-bit integers while activations remain 16-bit. On the default CUDA MoE runner, the selected scheme is the CUDA WNA16 MoE implementation; on Triton or HIP backends, the analogous Triton MoE scheme is selected instead. Loading then works on stacked expert tensors for gate/up and down projections. SGLang repacks those tensors, permutes their scales, clears or sorts per-expert `g_idx`, and constructs a Marlin MoE runner for the default CUDA path. Stage 1 classifies this as a load-time, in-memory repacked specialized MoE kernel path.

Example 19.7.3 (Case 3: GGUF checkpoint) Use the Qwen2.5 0.5B Instruct dense model as the floating-point baseline and its `Q4_K_M` GGUF file as the quantized variant. The repository and file name say GGUF, but the file-level label is a checkpoint packaging convention. The dispatch decision is made from each tensor's loaded `qweight_type`, such as `Q4_K` for a linear weight. The case asks: which tags are loaded, does the measured shape hit MMVQ, MMQ, ordinary matmul, or dequantized fallback, and how should prefill and decode be reported?

GGUF enters SGLang as file-format bytes plus per-tensor format tags. As Sections 19.2 and 19.6 describe, the loader emits `qweight/qweight_type` pairs and the CUDA path classifies each forward shape by that tag. Stage 1 therefore records the branch actually taken: ordinary matmul for unquantized tensors, MMVQ for supported small-row calls, MMQ for supported larger prefill or batched calls, or explicit dequantization when the tag lacks a direct matrix path. For the `Q4_K_M` case, standard and K-quant linear tensors should be reported as shape-dependent direct packed dispatch: MMVQ for decode-like small-row calls and MMQ for larger calls [11].

Example 19.7.4 (Case 4: Quantized KV cache) Use the same model and traffic trace while changing only the cache dtype option. Compare the BF16 baseline, the FP8 E4M3 cache path, and the FP4 E2M1 cache path. This case is not a checkpoint-format comparison. It asks what happens to resident attention state after SGLang has already computed new K/V rows. For FP8, Stage 1 records whether K/V scales came from a checkpoint or a JSON file supplied through `--quantization-param-path`; if neither source provides them, the run uses default scale 1.0, which is a correctly shaped but lower-confidence numerical contract. For FP4, Stage 1 records which FP4 KV-cache method owns the packed bytes and scale side buffers, because the allocator-visible byte account includes both.

The path classification must also name the consuming attention backend. A backend with direct support for the selected cache representation can turn smaller resident state into higher token capacity or concurrency. A backend that must dequantize before attention may still be correct, but its workspace, conversion time, and extra memory traffic belong in the measured path. The baseline is therefore not "quantized model versus floating-point model"; it is BF16 KV cache versus FP8 or FP4 KV cache under the same model weights, request trace, page size, attention backend, and quality gate.

Across all four cases, Stage 2 uses the shared measurement record in Section 19.8. A fast path that misses the accepted accuracy tolerance is not a useful serving path. The case-specific additions are narrower than the protocol itself:

Case-specific Stage 2 deltas

- Dense GPTQ: record both the deployment comparison (FP16/BF16 baseline versus default GPTQ-Marlin) and the SGLang-method comparison (default GPTQ-Marlin versus forced ordinary GPTQ), including load-time repacking cost.

- Compressed-tensors MoE: record the fused or DeepEP-class MoE runner, backend family, expert packing/repacking work, and the MoE baseline selected for the same routing workload.
- GGUF: record the `qweight_type` branch taken at each measured shape, because the same file can use MMVQ for decode-like calls, MMQ for larger calls, or dequantized fallback for unsupported tags.
- Quantized KV cache: record cache dtype, scale source, page size, attention backend, direct-read versus dequantized-read behavior, scale-buffer or workspace bytes, resident-token capacity, and quality gate.

The practical design rule is the same in all four case studies: check the contract where the metadata is used. The loader can reject an impossible rank, group size, or module mapping before requests arrive. The method selector can reject a target that has no supported scheme. The kernel can reject or bypass a shape it cannot implement directly. Quantization changes the resource tradeoff only after SGLang has shown that the packed values, scales, target module, and kernel all describe the same operation, and after the operator has measured that realized path against the right baseline.

19.8. Measurement Protocol and Evidence

A useful quantization measurement changes one variable in matrix multiplication under reduced numerical precision while holding model, request trace, hardware, and measurement window fixed. The primary reported quantities are relative output error and tokens per second. Those quantities become interpretable only when they are tied to a fixed workload and correctness probe.

Quantization evidence must separate format conversion, kernel choice, and numerical tolerance.

The prediction for the Thor/Qwen slice was directional. GPTQ-Marlin should improve steady decode or output-token throughput when its kernel path is available. It may increase first-token latency because loading and dispatch have extra work. It should not be counted as a memory-saving result unless the loaded footprint actually decreases.

The public Thor/Qwen slice compares the dense `Qwen/Qwen2.5-0.5B-Instruct` baseline with its GPTQ-Int4 variant on one CUDA device class across anonymized hosts. It uses a fixed serving trace and fixed correctness probes; raw commit, patch, command-line, host, and logprob-pair details remain in experiment metadata. The quantized serving path selected GPTQ-Marlin, so the row supports a kernel-throughput comparison rather than a memory-saving claim.

Table 19.10.: Measurement record for quantized execution. A throughput number is interpretable only when provenance, workload controls, correctness probes, kernel availability, and resource metrics describe the same serving path.

Field	Record value	Why it matters
Provenance	SGLang commit, command line, model, quantization config, and backend settings.	Reconstructs the loader and kernel contract being measured.
Workload controls	Request trace, prompt and output distribution, batching policy, page size, warmup, and measurement window.	Keeps throughput changes attributable to quantization rather than workload drift.
Correctness probe	Fixed probe batch, full-logits error when instrumented or requested-token logprob-slice deltas on the serving API, and text-regression pass/fail.	Separates expected quantization error from implementation error.
Kernel availability	Selected quantized kernel, fallback path, or blocked dependency/import result.	Records missing fast executable paths for the measured representation.
Resource metrics	Output-token throughput, TTFT, inter-token latency, and peak resident memory.	Shows whether smaller storage survives conversion, scale, and launch overheads.

Definition 19.8.1 (Quantized-run measurement record) *A quantized-run measurement record is the tuple*

$$r_q = (c, h, \mu, w, b, v, k, e)$$

that binds the SGLang commit c, hardware h, model and quantization metadata μ, workload w, baseline b, quantized variant v, kernel-availability outcome k, and error/throughput metrics e. The record is complete only when the variant and baseline share the same workload controls and the correctness probe uses the same packed representation that the serving path executes.

For one run, r_q might bind c to an SGLang commit hash, h to an H100 node, μ to a W8A8 model configuration, w to a fixed prompt trace, b to the BF16 baseline, v to the quantized loader path, k to the selected kernel or fallback, and e to the measured logprob delta, TTFT, and throughput fields.

Table 19.10 anchors the measured rows before the chapter turns them into claims.

Table 19.11 reports the completed Thor measurement. The GPTQ run improved output-token throughput on most reviewed hosts and regressed it slightly on another. Median TTFT increased for GPTQ on every reviewed host, while median inter-token latency decreased. The server logs reported larger loaded-weight memory for this small GPTQ model than for the baseline, so this run supports a kernel-throughput comparison rather than a memory-saving claim.

For the running example, the baseline uses the ordinary feasible path and the variant changes only the mechanism associated with scale layout. The result supports a quantized-kernel serving comparison because the same workload controls were used and the quantized path selected GPTQ-Marlin. It should not be read as a full-logits equivalence result without additional instrumentation.

A second completed row exercises the online FP8 path on an RTX 5090 rather than GPTQ-Marlin. The external-input rerun fixed `Qwen/Qwen2.5-0.5B-Instruct`, two random-token requests with 16 input tokens and four output tokens,

Metric	Dense baseline	GPTQ-Int4 variant	Claim boundary
Output throughput	2,469 output tok/s mean; 2,420–2,504 host range	2,605 output tok/s mean; 2,431–2,797 host range	Host-level effects were mixed, so the row is not a uniform speedup claim.
TTFT	331 ms mean	449 ms mean	First-token latency increased for the quantized path.
ITL	8.17 ms mean	5.47 ms mean	Decode cadence improved after the first token.
Loaded weight memory	0.13 GB	0.72 GB mean	This small GPTQ artifact does not support a memory-saving claim.
Correctness proxy	Fixed text and requested-token logprob-slice probes	Same probes	Serving-path slice comparison only; no full-logits equivalence claim.

Table 19.11.: Thor quantization result at a glance. Values summarize the reviewed dense-versus-GPTQ slice; raw host rows and probe-pair counts remain in experiment metadata.

Variant	Resident footprint	Mean latency	Throughput	Output-slice check
Dense	0.98 GB weights; 1,218,688 KV tokens	TTFT = 11.82 ms; ITL = 3.39 ms; E2E = 22.01 ms	42.47 req/s; 169.88 output tok/s	0 errors; two of two requests emitted four tokens.
Online FP8	0.66 GB weights; 1,247,872 KV tokens	TTFT = 13.63 ms; ITL = 4.38 ms; E2E = 26.77 ms	35.43 req/s; 141.72 output tok/s	0 errors; two of two requests emitted four tokens.

Table 19.12.: RTX 5090 online-FP8 serving row from the external-input rerun. The row is a short fixed-workload comparison for one model and one online quantization policy. It is not a quantization-format sweep or a full text/logprob equivalence result.

FlashInfer attention, no warmup requests, one-way concurrency, and disabled CUDA graphs. The artifact records SGLang commit `612785ffdcaf`, PyTorch `2.11.0+cu130`, compute capability 12.0, with the raw run identifier retained in experiment metadata.

Table 19.12 should be read as a serving-path and resource result, not as an output-equivalence result. The dense and online FP8 variants both returned the expected output length with zero request errors, but the generated text matched in zero of two paired requests. That is not surprising for this tiny fixed run. The model and request window are small enough that online-quantization overheads can dominate, and FP8 changes floating-point rounding before sampling; exact text matches across two sampled requests are therefore not the expected correctness oracle. The row says the path ran with the expected length and resource footprint, not that online FP8 is faster or text-identical on this trace.

The concrete run protocol is to choose one model and one request trace, run a full-precision baseline, then run exactly one quantized variant with the same batching, page size, backend policy, prompt distribution, output length, warmup count, and measurement window. The result table reports output-token throughput, time to first token, inter-token latency, peak resident memory, maximum relative

logits error or requested-token logprob-slice delta on a fixed probe batch, mean relative error on that probe batch, and pass or fail for text-level regression checks. If the target hardware lacks the required quantized kernel, the run records that as a kernel-availability result rather than silently switching to a different algorithm.

19.9. Loader-Kernel Agreement and Handoff

Quantization serving contracts stop before calibration or tensor-core schedules.

The final handoff is narrower than the full quantization taxonomy. By the time a serving step reaches a quantized kernel, the loader has already accepted a representation, filled packed tensors, and attached the metadata that interprets them. The kernel boundary has to preserve one fact: the consumer must apply the same interpretation map D_m to the same encoded state $\widehat{W}$ and metadata m that the loader accepted.

The GGUF `Q4_K` case from Section 19.6 is the concrete check. A $[4096, 4096]$ tensor with `qweight_type` set to `Q4_K` hands the consumer 256-weight blocks with 144 stored bytes per block. For a single decode row, $M = 1$, the correct handoff keeps that block layout on the direct MMVQ-style path rather than first materializing a full FP16 matrix. The handoff is therefore the packed tensor, format tag, block metadata, activation row count, and tolerance rule together, not a generic claim that the model is "4-bit."

The contract shares vocabulary with scheduler, KV-cache, attention-backend, and benchmarking code, but reduced precision matters there only when it changes metadata, memory footprint, numerical tolerance, or the comparable workload. Kernel microarchitecture consumes the packed representation through launch arguments and reference-tolerance tests; calibration data and tensor-core instruction schedules are separate inputs to that launch contract.

The chapter's closing rule is that reduced precision is executable only when loader metadata, kernel eligibility, and evidence describe the same operator. The runtime must preserve weight identity, shard layout, scale semantics, fused-kernel compatibility, attention and MoE interactions, and benchmarkable accuracy/performance accounts in one serving path.

19.10. Summary

Quantized serving changes representation without changing the layer operator. The dangerous case is shape-preserving: a locally reasonable kernel path can interpret packed bytes with the wrong scale layout or format tag and still return a tensor of the expected size. That is why the quantized-operator metadata invariant (Invariant 19.4.2) keeps packed weights and scale metadata tied to one numerical operation. Registry methods and backend capability point to that operation too. Slow-path behavior and measurement evidence must not switch accounts.

The representation account names the stored state: the quantized operator (Definition 19.2.2), scale layout (Definition 19.2.3), kernel layout (Definition 19.2.4), and format tag (Definition 19.2.5). The dispatch account asks which runtime boundary owns that state. Weights are mostly load-time state, activations are

batch-time state, MoE quantization also carries expert layout, and KV-cache quantization crosses the memory-lifetime boundary. KV-cache quantization is online serving-state quantization, not checkpoint-only weight compression. The runtime must keep cache dtype and K/V scale state with page ownership, backend selection, and measurement evidence in one account.

SGLang does not need to be read as the inventor of every quantization algorithm it serves. Its contribution is the serving boundary that joins external checkpoint formats to method selection and layout normalization. Custom `sgl_kernel` operators make that representation executable inside SGLang; path checks, slow paths, and tests keep it tied to reference behavior. Separate quantization algorithms such as AWQ and GPTQ from storage formats such as GGUF and optimized kernel layouts such as Marlin. They are not one interchangeable list. Measurement has the final check. It keeps logical work separate from metadata overhead. It also separates launch or transfer overhead, slow-path behavior, provenance, and correctness evidence for the same serving path.

19.11. Exercises

1. Define quantized linear operator in the running example and name the state that must travel with it.
2. Using the formula $y = xD_m(\widehat{W})$, explain the roles of the stored representation, metadata tuple, interpretation function, and input activations.
3. Compare weight quantization with KV-cache quantization across owner, metadata, validation point, and a shape-preserving failure mode.
4. Distinguish AWQ, GPTQ, and Marlin. Which names describe checkpoint semantics, and which name describes an optimized executable layout or kernel path?
5. Apply the step-alignment invariant to a batch with two requests that use different scale layout values.
6. Describe one failure mode that could preserve tensor shapes while corrupting request semantics.
7. Explain how a scheduler decision can remain valid even when the backend implementation path changes.
8. Give one example of the tradeoff that smaller weight traffic competes with dequantization overhead, scale memory, and kernel availability.
9. State one handoff between quantized representation metadata and a consuming kernel.
10. Review a proposed new quantization format by listing the representation, scale-layout, format-tag, slow-path, and measurement evidence that must stay attached to one logical operator.