21. Mixture-of-Experts

Chapter Abstract. MoE serving turns one dense feed-forward step into a sparse route, execute, communicate, and combine path. Each token row may move to a different expert group, but the final output still belongs to the row that requested it. SGLang has to preserve two accounts at once: the logical experts chosen by the model and the physical expert instances that execute the work. Sparsity may move work; it cannot move request ownership.

21.1. The MoE Serving Contract

MoE serving adds sparse routing before work can become balanced GPU execution.

Mixture-of-experts (MoE) serving replaces one dense feed-forward layer with a sparse path [45]. Each token row chooses a small set of experts. The runtime moves or groups token slices by those choices. Expert kernels run on the grouped rows, and combine returns weighted outputs to the original batch order. The systems problem is that sparsity breaks the simple dense invariant that tensor row order alone tells the runtime which request owns each output row. Later definitions name the routed token, expert routing, and combine operation that make this sparse path executable.

The running trace uses a four-token slice through a `Qwen/Qwen1.5-MoE-A2.7B` routed layer. Four token rows enter the layer, the router selects two logical experts per row, and the selected weights are

$$t_0 : \{0, 1\}, \quad t_1 : \{0, 2\}, \quad t_2 : \{0, 3\}, \quad t_3 : \{1, 2\},$$

with weights $(0.7, 0.3)$, $(0.6, 0.4)$, $(0.8, 0.2)$, and $(0.55, 0.45)$. One routed assignment is one token row paired with one selected expert, so top-2 routing turns four token rows into $4 \cdot 2 = 8$ assignments. The trace has $R = 8$ valid routed assignments. Counting by logical expert gives three assignments to E_0, two to E_1, two to E_2, and one to E_3. In request order the trace is four rows wide; in execution order it is four expert groups wide. That conversion is the main source of MoE serving complexity.

Two ledgers must remain visible. The *semantic ledger* records the logical experts and routing weights selected by the model. The *execution ledger* records the physical expert instance, rank, dispatcher record, runner layout, padding state, and combine metadata used to execute that choice. Correct MoE serving is the agreement between those ledgers at combine time. SGLang exposes the same agreement through routing, dispatch, runner, and combine surfaces [42]. Later concept-to-code maps name the concrete helper functions, fused layers, dispatchers, expert-parallel placement state, kernels, and tests.

MoE changes placement and row order inside one layer. It does not change which logical token row owns the final result. The contract follows the row through four names: the scheduler row, the router's logical expert choices, the

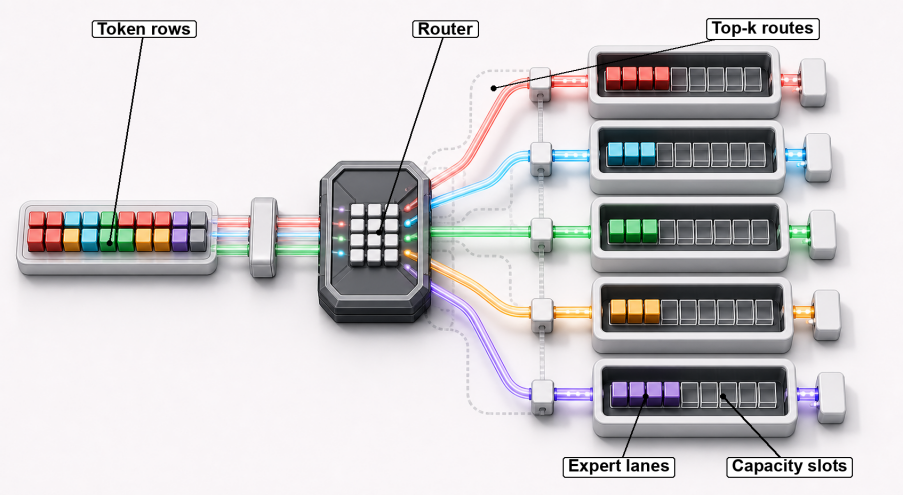


Figure 21.1.: Token routing in an MoE layer. The router turns request-order token rows into top-k expert assignments, while capacity and placement metadata determine which physical expert slots receive those routed copies.

execution layout that may permute or pad routed copies, and the combine record that restores request order.

Figure 21.1 shows why the execution ledger needs more than the router's top-k ids: physical slots and capacity decide where the routed copies actually run.

This trace gives the local vocabulary before backend names appear. A *logical expert* is the learned expert function named by the model. A *physical expert instance* is the copy, shard, or rank-local slot that can execute a logical expert for the current deployment (Definition 22.2.1). *Routing scores* are the gate values computed for one token row over eligible logical experts. The token's *top-k experts* are the selected logical experts, together with the routing weights used when outputs are reduced (Definition 21.1.2). A *token slice* is the materialized hidden row, or routed copy of that row, sent to an expert. EP means expert parallelism and TP means tensor parallelism in this local MoE account. The broader parallelism taxonomy develops the full placement vocabulary (Definition 22.1.3 and Definition 22.1.2).

Definition 21.1.3 formalizes expert identity. Section 21.2 develops router records. Section 21.3 binds the launch and placement state. Section 21.4 follows dispatch and combine. Section 21.5 develops the runner layouts that consume token slices.

Definition 21.1.1 (MoE serving contract) *For one routed layer, the MoE serving contract is the requirement that token row ids, logical top-k expert ids, routing weights, placement metadata, dispatch layout, runner input layout, and combine metadata produce the same row-indexed sparse result as*

$$y_t = \sum_{j=1}^{k_t} \alpha_{t,j} E_{e_{t,j}}(x_t),$$

even if the runtime permutes, pads, replicates, or communicates intermediate token slices.

For token row t_0, suppose $\mathrm{TopK}(g_{t_0}, 2) = \{E_0, E_1\}$ with routing weights $\alpha_{t_0,1} = 0.7$ and $\alpha_{t_0,2} = 0.3$. The serving contract says the final row must equal $0.7E_0(x_{t_0}) + 0.3E_1(x_{t_0})$, even if dispatch renames the physical expert slots or moves the two routed copies across ranks.

The serving contract states the end-to-end sparse-layer obligation. The top-k routing record below is the per-token evidence that the router hands to dispatch and combine to satisfy that obligation.

Definition 21.1.2 (Top-k routing record) *A top-k routing record for token row t is*

$$R_t^{\text{route}} = \left((e_{t,j}, \alpha_{t,j})\right)_{j=1}^{k_t},$$

together with the score-transform policy π_t that produced it and any mask or padding convention that says which selected slots are real routed assignments. Each $e_{t,j}$ is a logical expert id and $\alpha_{t,j}$ is the routing weight consumed by combine.

$e_{t,j}$ is the logical expert; physical placement is resolved later.

For the same row t_0, the routing record is $R_{t_0} = ((E_0, 0.7), (E_1, 0.3))$, $k_{t_0} = 2$, and π_{t_0} is the score rule, such as softmax over the selected experts. If padding adds a dummy slot for a kernel, the mask marks that slot as non-real so combine ignores it.

The routing record names logical experts. The next definition separates those logical functions from the physical replicas, rank-local slots, or backend locations that may execute them at runtime.

Definition 21.1.3 (Logical and physical expert identity) *A logical expert identity e names the learned expert function E_e used in the sparse formula. A physical expert identity p names the runtime instance, replica, shard, rank-local slot, or backend-specific slot that executes a logical expert for a particular placement snapshot. The launch and placement section later names the current logical-to-physical relation Φ_s; this definition only separates the model's learned expert function from the runtime slot that executes it.*

The distinction between logical and physical identity is the first audit rule. EPLB, elastic EP, and backend-specific dispatch can change the physical id that receives a token slice, but they must leave an auditable path back to the logical expert id and routing weight selected for that token. In the running trace, for example, t_0's first routed copy may travel to a physical replica rather than a slot literally named E_0, but combine still owes row t_0 the weighted result $0.7E_0(x_{t_0}) + 0.3E_1(x_{t_0})$.

The row-level audit is the organizing spine. It has four larger blocks. The first block defines the sparse formula, launch record, and placement snapshot. The second follows the execution path that permutes rows into expert-major work and reconstructs them. The third explains load feedback, overlap, costs, and measurement. The fourth closes the boundary with neighboring distributed and kernel topics.

Within those blocks, the trace order stays concrete: route, configure, dispatch, run, combine, rebalance, account, and measure. Every later section returns to the same four-token trace by asking which part of that ledger has changed and which part must remain invariant.

The sparse path uses a compact set of symbols repeatedly: router records, expert identities, placement snapshots, dispatch layouts, and load feedback. Table 21.2 keeps that notation near the first sparse serving contract.

Block	Sections	Question for one token row
Sparse contract	Sections 21.2 and 21.3.	Which logical expert ids, routing weights, score policy, padding convention, launch knobs, and placement snapshot define the executable sparse formula?
Execution path	Sections 21.4, 21.5, and 21.6.	Which physical ids, dispatch record, runner layout, dtype, scale metadata, and inverse state carry the row out and back?
Feedback and evidence	Sections 21.7, 21.8, and 21.8.	Which expert counts, overlap boundaries, remote-assignment counts, and correctness gates make performance claims interpretable?
Contract and audit	Sections 21.9 through 21.10.	Which handoff surface carries the remaining detail, and which source path proves that the same sparse formula survived execution?

Table 21.1.: MoE serving-contract chapter map. The rows connect each chapter block to the token-row question it answers.

Symbol	Meaning	Role in the serving account
t	Token row index	The row whose sparse result must return to request order.
x_t, y_t	Hidden input and sparse output	The row-local values before routing and after combine.
R_t^{route}	Top-k routing record	The selected expert ids, weights, score policy, and mask for row t.
R	Valid routed-assignment count	The number of real top-k slots after masking invalid or padded entries.
$e_{t,j}, \alpha_{t,j}$	Expert id and routing weight	The jth logical expert choice and combine coefficient for row t.
E_e	Logical expert function	The learned expert that defines sparse-layer semantics.
Φ_s	Logical-to-physical placement relation	The physical replicas, shards, or rank-local slots allowed to execute E_e.
Π_{disp}	Dispatch layout	The expert-major permutation, padding, and inverse metadata used by kernels.
N_p	Physical expert load	The real routed assignments assigned to physical expert instance p; padding rows do not count.

Table 21.2.: Local notation for MoE serving. The symbols keep logical routing, physical placement, dispatch order, and load evidence separate.

21.2. Router Records and Sparse Semantics

The selected expert ids are not meaningful without the policy that produced them.

The router is the first place where the dense feed-forward contract changes. For each real token row it computes scores over logical experts, applies the configured selection policy, emits selected ids and weights, and records any format convention needed by dispatch. A bare value of k is therefore insufficient. The executable router policy also includes score transform, grouping, bias, routed scaling, fused shared expert columns, padding behavior, and top-k output format.

Definition 21.2.1 (Routed token) *A routed token is the row-level object* $(t, x_t, R_t^{\text{route}})$ *produced when a token representation* x_t *is assigned to one or more experts by a gating function during one MoE layer. The routing record* R_t^{route} *is part of the token's runtime meaning until combine has reconstructed the output row.*

A routed token is the data item that carries a row through sparse execution. Expert routing is the decision procedure that creates its routing record.

Definition 21.2.2 (Expert routing) *Expert routing is the per-token decision*

$$\pi_{\text{route}}(x_t) = R_t^{\text{route}}$$

that maps a hidden-state row to one or more logical experts, records the corresponding routing weights, and preserves enough row identity for weighted expert outputs to be combined back into the original batch order.

This extends the Volume I row-alignment contract: expert routing changes where a token row executes, not which request row the combined result belongs to.

Example 21.2.1 (Why top-k ids alone are not the route) Consider two token rows with top-2 logical expert ids $[3, 7]$. In a plain softmax policy the stored weights may be normalized over the selected pair. In a sigmoid-with-bias policy the bias can affect which experts are selected while the stored weights use the un-biased sigmoid values. If fused shared experts are active, additional shared columns may be appended under a separate scaling convention. Dispatch cannot recover those semantics from the ids $[3, 7]$ alone; it needs the routing record that says which policy produced the ids, which weights are stored, and which post-processing rule already ran.

Example 21.2.2 (One token row through route, dispatch, and combine) For token row t, the router starts with logits g_t. The configured policy turns g_t into ids and weights, for example $(\mathrm{id}, w) = ((3, 7), (0.62, 0.38))$, then applies any logical-to-physical expert remap, shared-expert extension, or padding mask before dispatch. Dispatch uses that completed record to send copies of x_t to the selected expert locations. Combine later consumes the expert outputs together with the same weights and row identity to reconstruct token row t's feed-forward result in the original batch order.
The load-bearing fact is that ids, weights, remapping, and row identity form one runtime record. If a source path changes only the id tensor while another path uses the old weights or old physical-expert map, tensor shapes can still match while the sparse feed-forward result belongs to the wrong route.

Running trace checkpoint.

In the four-token trace from Section 21.1, row t_0 leaves the router as

$$R_{t_0}^{\text{route}} = ((E_0, 0.7), (E_1, 0.3)).$$

That visible pair of ids and weights is only the compact form. A complete trace also has to answer four questions: which score transform produced the weights, which grouped or shared-expert convention was active, whether the returned tensor is already a completed route record, and which post-processing rule changed ids before dispatch. The source-reading distinction is therefore not which kernel name appears in a profile first; it is whether the boundary emitted logits for a later selector or an executable routing record for t_0.

Selection boundary map.

The practical source-reading entry point is `select_experts`. It first extracts the fields in `TopKConfig`, applies any expert-location transform to logits and correction bias, and then chooses the selection path. The branch order matters because the same tensor shapes can mean different policies.

After the branch returns, the route record R_t^{route} is not complete until id post-processing has applied every active placement convention: logical-to-physical remapping, padded-region masking, fused shared-expert handling, and DeepEP shared-slot remapping. The useful debugging question is therefore which policy fields in C forced the branch and which post-processing rule changed the ids before dispatch. In source, C is `TopKConfig` and the shared post-processing helper is `_post_process_topk_ids`.

Algorithm 7 dispatches router logits to one policy path before applying the common id post-processing that dispatch will consume. It normalizes the inputs into the router's local view by reading `TopKConfig` and applying any expert-location transform. It then selects one implementation family: grouped, biased grouped, native Torch, generic fused top-k, raw-logit FlashInfer, or a custom router. Across those families, ids may be remapped, padded rows may be masked, shared experts may be appended or interleaved for DeepEP, and the expert-distribution recorder sees the final ids.

Table 21.3.: Router boundary records. Each boundary consumes and emits sparse routing state with a reader-facing rule.

Boundary	Consumes	Emits	Reader-facing rule
Softmax or sigmoid top-k	Gating logits plus `TopKConfig`'s score rule.	`topk_ids` and `topk_weights`.	The stored weights mean what the score transform says they mean.
Fused grouped gate	Logits, grouping fields, correction bias, shared-expert settings, and scaling fields.	A completed routing record with ids, weights, group choice, and shared-column convention.	Dispatch consumes the record; it must not rerank the returned weights.
Router-projection accelerator	Hidden rows and router weights before top-k selection.	Dense router scores.	This is still before expert choice; a later policy owns ids and weights.
Custom router shim	Hidden rows plus model-family router weights or fused router configuration.	A completed routing record, or a fallback logit matrix on unsupported hardware.	The trace must record which side of the boundary the shim returned.
Native or fallback selector	Logits under assertion-checked feature limits.	Standard top-k tensors.	Useful as a checked baseline, but not proof that all fused conventions apply.

The CUDA softmax top-k path is the concrete kernel form of the same router contract, anchored by `sgl-kernel/csrc/moe/moe_topk_softmax_kernels.cu` and its `topkGatingSoftmax` implementation. The source-boundary concordance later keeps the implementation anchors in one place, so the prose can focus on the row contract rather than repeating path lists.

Softmax and Sigmoid Top-k Kernels.

SGLang carries both softmax and sigmoid top-k kernels because different MoE families define router scores differently. The motivation is semantic first: softmax makes selected weights compete inside a probability simplex, while a sigmoid-style gate treats expert scores independently and may apply a correction bias before selection. A single generic top-k call would hide which score transform produced the expert ids that combine later trusts.

The intuition is that the router kernel produces a row contract together with a ranking. For each token row, the softmax or sigmoid top-k kernel must write selected ids and selected weights. That record must match the policy that `select_experts` advertised. The sigmoid path checks two-dimensional gating logits and output tensors, supports float32/float16/bfloat16 inputs, allocates a float workspace when the expert count is not a supported power-of-two case or has more than 256 experts, and can renormalize selected weights after using optional correction bias for selection.

The API contract is deliberately asymmetric. Plain top-k entries mutate caller-owned output buffers, while fused grouped gates return a completed record. That difference is a source-reading clue: a caller that already owns selected

Algorithm 7: `select_experts` dispatches router logits to exactly one top-k policy path, then applies the common id post-processing that dispatch will consume.

Input: Hidden rows X; router logits G; `TopKConfig` C; optional location record Λ_{loc}
Output: `StandardTopKOutput` carrying weights, ids, and transformed logits

1 read the top-k size, score function, grouped-routing fields, bias, native-Torch flag, fused shared experts, custom router, and scaling fields from C;
2 transform G and correction bias through Λ_{loc} when expert-location dispatch is active;
3 set k_{routed} to k minus the fused shared-expert count;
4 select exactly one top-k policy path: grouped or biased grouped, native Torch, generic fused or raw-logit fused, or a custom router;
5 reject feature combinations unsupported by the selected path before executing it;
6 execute the selected path to produce routed expert ids and weights;
7 post-process ids and weights for logical-to-physical remapping, padded rows, fused shared experts, and DeepEP shared slots;
8 record the selected expert distribution and return `StandardTopKOutput`;

weight and id buffers is asking for a score transform and selection kernel. In source, those buffers are `topk_weights` and `topk_ids`. A caller that invokes `moe_fused_gate` is asking the kernel to allocate and return the selected row record itself. In the running trace, both forms can produce a pair that looks like $((E_0, 0.7), (E_1, 0.3))$, but the former says "these buffers were filled under this score rule," while the latter says "this kernel already fused the score rule, grouping rule, and output convention into the record."

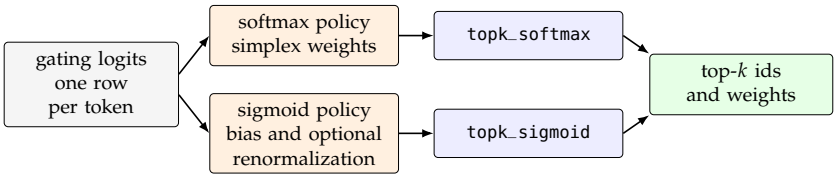


Figure 21.2.: MoE top-k kernels encode the router score policy. Softmax and sigmoid paths can produce the same tensor shapes, but their ids and weights are meaningful only together with the transform, bias, and renormalization rule that created them.

Figure 21.2 separates the score transform from the kernel interface: the output shapes can match while the weight semantics differ.

Router records name logical expert choices before placement turns them into physical work.

Example 21.2.3 (Same logits, different router record) For one token with expert logits $[4, 3, 0]$ and $k = 2$, a softmax router and a sigmoid router may choose the same expert ids $(0, 1)$. The weights still do not mean the same thing: the softmax weights are normalized over the competitive distribution, while the sigmoid path can score each expert independently, apply a correction bias, and optionally renormalize only the selected pair. Combine must consume the ids and weights under the policy that produced them. The independent sigmoid form is useful when the model wants several experts to look simultaneously plausible instead of forcing all expert scores to compete inside one simplex. Correction bias then steers which experts are selected, but the stored combine weight remains the model score gathered for the selected expert rather than the biased choice score.

For every branch, the trace fields to preserve are the same: score function, grouping mask, routed-versus-shared columns, scaling placement, output format, and padding convention. Combine cares about those fields because ids and weights are only meaningful under the policy that produced them; a padded slot or shared tail column can occupy tensor space without becoming ordinary routed model work.

Fused grouped-gate kernels.

The fused-gate kernels sit at the same semantic boundary as `topk_softmax` and `topk_sigmoid`, but they fold more router policy into the device entry point. The `moe_fused_gate` path handles grouped expert selection: it scores expert groups, selects the top groups, selects experts inside those groups, and can account for fused shared experts and routed scaling. The `kimi_k2_moe_fused_gate` path is the same kind of boundary for the Kimi K2 case. Because all routed experts belong to one group there, the kernel can skip grouped selection while still returning selected ids and weights.

The speed benefit matters, but the contract is broader: the returned top-k tensors already encode grouping, bias, renormalization, shared-expert convention, and scaling placement for later dispatch.

`moe_fused_gate` is therefore an eligibility decision as well as a semantic operator. CUDA reaches it only when the grouped problem matches the kernel's row geometry: the expert count is a power of two, the expert count is divisible by the number of expert groups, and each group has at most 32 experts. Other valid router policies can still be executed by a different top-k path, so a profile that lacks `moe_fused_gate` is not automatically evidence that the model used a different routing rule. It may mean that a Kimi-specific one-group gate, a JIT grouped top-k, AITER, MATE, Torch-style, CPU, or fallback branch preserved the same routing contract under a different guard.

`moe_fused_gate` is the representative source-level anatomy because it contains the grouped-routing decisions that ordinary top-k hides. For one token row, the kernel transforms logits with sigmoid, adds correction bias for choice, removes unselected groups, selects routed experts, appends any fused shared experts, and normalizes or scales the stored weights. The stored weight is the original sigmoid score for the selected expert, not the bias-adjusted choice score. That distinction is the reason the tests compare against a reference that uses bias to choose experts but gathers weights from the un-biased sigmoid scores.

Example 21.2.4 (moe fused gate on one row) Take four routed experts in two groups, $G_0 = \{E_0, E_1\}$ and $G_1 = \{E_2, E_3\}$, with $k = 2$, `topk_group`= 1, no fused shared expert, and renormalization enabled. Suppose the sigmoid scores after loading the row are

$$s = [0.88, 0.73, 0.62, 0.60]$$

and the correction bias is $b = [0, 0, 0.35, 0.35]$. The choice scores are

$$s + b = [0.88, 0.73, 0.97, 0.95].$$

The group score is the sum of the top two choice scores in each group, so G_0 scores 1.61 and G_1 scores 1.92. The selected group is therefore G_1, and the routed expert ids are E_2, E_3. The important implementation detail is that the stored weights come from the original sigmoid scores, not the biased choice scores: before renormalization they are $[0.62, 0.60]$, and after renormalization they become approximately $[0.508, 0.492]$. The bias moved the row into a different group, but combine still receives weights under the kernel's un-biased score convention.

Example 21.2.5 (Fused shared expert columns and scaling) Use the same selected routed weights from the previous example, but set `num_fused_shared_experts` to 1, `topk` to 3, and routed scaling $\lambda = 2.5$. The kernel appends a shared column after the two routed columns: its id is 4, the first id after the four routed experts, and its pre-normalized weight is $(0.62 + 0.60)/\lambda$. The final normalization divides by the routed sum 1.22, so the stored record is approximately

$$\texttt{topk_ids} = [2, 3, 4], \qquad \texttt{topk_weights} = [0.508, 0.492, 0.400].$$

If `apply_routed_scaling_factor_on_output` is true, the same kernel then multiplies all stored weights by λ, producing approximately $[1.270, 1.230, 1.000]$. The shared column is not a third routed expert; its id range and scaling convention must travel with the top-k record so combine and later runner code do not apply routed scaling to the shared contribution twice.

Source-boundary concordance.

The concordance is shortest when it follows the route-record concept instead of enumerating every fast path. It separates the policy fields, the boundary chooser, and the completed record that downstream dispatch consumes. The specialized names matter only insofar as they answer one question: did this source boundary return logits, or did it return a complete route record?

Router boundary concordance

- **Policy fields.** `TopKConfig` carries score function, top-k, grouping, correction bias, fused shared experts, routed scaling, padding, and output-format choices.

- **Boundary chooser.** `select_experts` reads that policy, applies expert-location transforms when active, chooses one selection family, and returns `StandardTopKOutput` after common id post-processing.
- **Logits-only accelerator.** `dsv3_router_gemm` moves router projection work into a specialized path, but its output is still a dense score matrix for a later selector.
- **Completed-record boundaries.** `moe_fused_gate`, `kimi_k2_moe_fused_gate`, `fused_moe_router_shim`, JIT `grouped_topk`, and FlashInfer `fused_topk_deepseek` have already chosen columns and weights.
- **Model-family routing.** A path such as `Grok1MoE` may use `fused_moe_router_shim` to fuse projection, softcapping, and top-k on CUDA, while a fallback path materializes logits and then calls `TopK`.

The concordance keeps source names available without making the reader memorize them as a separate taxonomy.

Example 21.2.6 (Logits are not a routing record) Suppose a decode step takes the `dsv3_router_gemm` fast path for two token rows. The returned matrix still has one score per expert, so dispatch cannot consume it as `topk_ids`. A later top-k policy must still choose columns and assign weights. By contrast, a `fused_moe_router_shim` or `moe_fused_gate` path has already made the column choices. Passing its weights back through a second generic `torch.topk` would silently change the router policy that combine is supposed to preserve. This failure changes semantics before shape checks can catch it. If logits are misread as ids, the row may be dispatched to arbitrary expert columns before the model's selection policy has run. If a completed record is reranked, the chosen ids can change while the stored weights, grouping mask, or shared-column convention still describe the old route. In both cases tensor ranks can look plausible while the combined row is no longer the sparse result selected by the router.

The local rule is the boundary distinction: some paths return logits for a later selector, while others have already returned the executable routing record.

SGLang marks unsupported routing combinations explicitly. Native Torch top-k does not accept padded-token counts or expert-location dispatch; custom routing likewise rejects padded-token counts and expert-location dispatch; and several non-grouped paths assert that output-side routed scaling is not implemented. Padded regions are represented by setting routed expert ids to -1 only in paths that know how to mask them. These assertions are part of the algorithm: a router choice is not complete until its score transform, padding behavior, remapping behavior, and output format are compatible with the dispatcher and runner.

Fused shared experts add a small but important wrinkle. They occupy tail columns in the top-k result even though they are not ordinary routed experts selected by the gate. SGLang therefore removes those shared columns from grouped top-k selection, protects them from ordinary expert-location remapping, and, for DeepEP-class backends, remaps the routed columns into an interleaved layout where each rank also has a unique shared expert slot.

Example 21.2.7 (Fused shared expert columns) Assume $K = 3$, two routed choices, one fused shared expert, two EP ranks, and two routed experts per rank. If a token's routed choices are logical (E_1, E_3), the top-k record has two routed columns plus one shared tail column. For a DeepEP-class backend, the routed ids are shifted by the gaps reserved for shared slots: $E_1 \mapsto 1$ and $E_3 \mapsto 4$. The shared column is assigned the rank's shared slot, such as id 2 on rank 0 or id 5 on rank 1. The trace must record which columns are routed logical experts and which columns are fused shared experts; otherwise a dispatcher or EPLB remap can accidentally treat shared work as a normal routed expert.

In the running trace from Section 21.1, the visible result of routing is the four row-major top-k records and their weights. The hidden result is the policy evidence behind them: score transform, grouping mask, scaling choice, shared-expert convention, output format, and padding behavior. Those fields explain why E_0 receives three assignments and E_3 receives one; without them the loads remain counts without becoming executable MoE state.

The router output is therefore the first contract boundary. For the running trace, this is where the eight valid routed assignments become real semantic work rather than just a count. Later stages inherit that policy record; they do not re-rank logits or treat a masked slot as expert work. The next section decides where the selected logical experts live, and the dispatch section moves the routed copies while preserving this record for combine.

21.3. Launch Configuration and Placement State

MoE knobs select execution paths along with faster implementations.

A routed row can choose logical expert e_7 and still run in the wrong place if launch configuration and placement state disagree about the physical expert that owns e_7. Server configuration therefore has to make the executable path visible before dispatch begins. A useful MoE launch record identifies the expert-parallel deployment and the two active backends. It also records any low-latency dispatch mode and the placement-balancer state that can change between serving steps.

Expert-parallel deployment.

An expert-parallel deployment places the physical instances of a routed MoE layer's experts across an expert-parallel rank group of size P_{ep}. SGLang selects this size through the expert-parallel deployment flag family. The deployment is valid only when the router, expert-location map, dispatcher, runner, and combine path all agree which physical instances are local to each EP rank.

With $P_{\text{ep}} = 4$, the deployment might use four EP ranks and assign each rank a subset of physical expert instances. A token routed to logical expert e_7 is executable only after the placement map and dispatcher agree which physical instance of e_7 one of those four ranks owns.

The expert-parallel deployment names the rank group; the placement map below names which physical instances in that group may execute each logical expert.

Definition 21.3.1 (Logical-to-physical expert map) *A logical-to-physical expert map is the placement relation*

$$\Phi_s : \mathcal{E}_{\text{logical}} \to 2^{\mathcal{P}_{\text{phys}}}$$

that assigns the model's expert ids to the physical expert instances or replicas that execute them for the current deployment and load-balancing state s.

If logical expert e_3 is replicated on physical instances p_1 and p_5, then $\Phi_s(e_3) = \{p_1, p_5\}$. A hot-expert mitigation can change Φ_s for future routed tokens, but already-dispatched rows still have to combine through the physical instance recorded for their slot.

Example 21.3.1 (Logical expert, physical instance, and backend pair) Suppose token row t_0 routes to logical expert e_7 with weight 0.8, and the current snapshot has $\Phi_s(e_7) = \{p_2, p_6\}$. If the dispatcher chooses physical instance p_6, the runner must execute the weights for e_7 materialized at p_6, and combine must return that result to t_0's top-k slot. Changing the runner backend from a local grouped GEMM to a DeepEP-backed path does not change the logical expert id or weight; it changes the record format that carries t_0 to p_6 and back.

Definition 21.3.2 (EPLB placement snapshot) *An expert-parallelism load-balancing (EPLB) placement snapshot is the tuple*

$$s = (\Phi_s, \Psi_s, c_s, \rho_s, \omega_s)$$

containing the logical-to-all-physical map, physical-to-logical map, valid replica counts, optional rank-dispatch map, and weight materialization state that a single MoE step must use consistently from dispatch through runner and combine.

Expert-parallel deployment fixes where expert instances may live. The backend pair below fixes how routed rows move to those instances and how the expert computation consumes them.

MoE backend pair.

The MoE backend pair $(B_{\text{a2a}}, B_{\text{run}})$ records the implementation pairing for one step: B_{a2a} is the all-to-all dispatch/combine backend and B_{run} is the expert computation backend. The pair fixes both the communication record format and the grouped-GEMM runner layout for one MoE execution path.

For an expert-parallel run, B_{a2a} could be an all-to-all dispatcher that exchanges routed rows across EP ranks, while B_{run} is a grouped GEMM runner expecting expert-major padded rows. Pairing that dispatcher with a runner that expects token-major rows would make the backend pair invalid even if both backends are individually supported.

MoE runtime selector.

Launch record field	Example value	Question it answers
Tensor and expert parallel sizes	`--tp8--ep8`	Which rank group can own each routed expert instance?
All-to-all backend	`deepep`	Which dispatch/combine record carries token rows across EP ranks?
Runner backend	`deep_gemm`	Which expert-major layout and scale metadata will the local runner consume?
Placement snapshot	$\Phi_s(e_7) = \{p_2, p_6\}$	Which physical instance may execute logical expert e_7 for this step?

Table 21.4.: MoE launch record fields. The record ties placement, parallelism, and dispatch choices to one sparse step.

The MoE runtime selector is the compatibility decision that turns server backend strings and model-layer metadata into one active router-output convention, dispatch/combine record format, runner core, fused-function path, and pre-/post-permute adapter for an MoE step. It is not a separate algorithm that changes expert choice; it selects the implementation contract that must preserve the routing record, including the combine evidence needed to restore rows after dispatch.

The expert-parallel deployment and backend-pair account explain why a common DeepSeek-style launch pattern [3, 4] binds several knobs at once. The model path, dispatch backend, runner backend, tensor-parallel size, and expert-parallel size form one compatibility statement rather than five independent preferences.

Concept-to-code map

- **Expert-parallel deployment flags.** `--expert-parallel-size`, `--ep-size`, and `--ep`.
- **Dispatch backend category.** `--moe-a2a-backend` chooses the dispatch and combine record family, such as a local path, a DeepEP path, or a transfer-backend path.
- **Runner backend category.** `--moe-runner-backend` chooses the expert-runner layout and numeric evidence, such as an automatic selector, a Triton-style runner, a CUTLASS-style grouped runner, or a DeepGEMM-style runner.
- **Source selector objects.** `initialize_moe_config`, `get_moe_a2a_backend`, `get_moe_runner_backend`, `MoeA2ABackend`, `MoeRunnerBackend`, `FusedMoE`, `MoeRunner`, `FusedOpPool`, `PermuteMethodPool`, `DeepEPMode.resolve`, `ExpertLocationMetadata`, and `EPLBManager`.
- **Placement and load-balancing knobs.** `--init-expert-location`, `--ep-dispatch-algorithm`, `--ep-num-redundant-experts`, `--enable-eplb`, and the EPLB recorder-window and rebalance settings.
- **Backend alias registry.** The full accepted alias set is source-owned. For a concrete launch, trace the chosen alias through the enum or registry to the dispatch record, runner layout, and placement evidence it requires.

The runtime selector paragraph is useful because several source files read the same backend state at different boundaries. At startup, `initialize_moe_config` stores the all-to-all and runner backend enum values from server arguments. The stored types are `MoeA2ABackend` and `MoeRunnerBackend`. Later calls to `get_moe_runner_backend` and `get_moe_a2a_backend` act as contract selectors: they

Table 21.5.: Source boundaries for MoE launch placement. Each boundary consults selector state and changes the runtime contract in a specific way.

Source boundary	Selector state consulted	Contract effect
`initialize_moe_config`	Server values for `--moe-a2a-backend`, `--moe-runner-backend`, speculative variants, and DeepEP mode.	Creates the enum state later read by router, dispatcher, and runner code.
`TopK.forward_cuda` and `select_experts`	`get_moe_runner_backend` plus `TopKConfig`.	Chooses standard, Triton-kernel, bypassed, raw-logit FlashInfer, grouped, native, or custom router-output conventions.
`FusedMoE.__init__`	Runner backend, A2A backend, EP/TP world sizes, quantization method, and fused shared-expert count.	Builds `MoeRunnerConfig`, creates weights, creates the MoE runner, and installs the dispatcher for the layer.
`MoeRunner.__init__` and `MoeRunner.run`	Runner backend, dispatch-output format, LoRA state, and registered fused functions.	Either runs a fused A2A-plus-runner function or selects a runner core plus `PermuteMethodPool` adapters for combine.

determine which router-output format, dispatcher record, runner core, fused runner path, and pre-/post-permute adapters are legal for the current sparse step.

The practical consequence is that a trace should record both the configured backend pair and the selector branch that actually executed. For example, `flashinfer_trtllm_routed` affects the top-k output convention before the expert runner sees any rows, while a missing fused function in `MoeRunner` can make a nominal backend pair invalid even after routing has succeeded. Section 21.2 explains the router-output side of this selector; Section 21.5 explains the runner-layout side.

For example, a specialized all-to-all dispatcher fixes the dispatch/combine format, a specialized grouped-GEMM runner fixes the expert-major runner layout, and equal tensor-parallel and expert-parallel sizes keep the rank spaces aligned. In the SGLang source, every nonlocal MoE all-to-all backend handled in server arguments forces $P_{\text{ep}} = P_{\text{tp}}$; hybrid $P_{\text{ep}} < P_{\text{tp}}$ belongs to the local backend path.

At the SGLang server surface, the all-to-all choices include local, DeepEP, transfer-backend, and FlashInfer-style paths. The runner choices include automatic, Triton, CUTLASS, FlashInfer, and DeepGEMM-style paths. These names are not independent performance hints: each one selects a dispatch, runner, or quantization contract that must still preserve the sparse formula.

This section stops at the launch contract. The dispatch section explains how DeepEP-class backends, local paths, and FlashInfer-class paths turn that contract into concrete count windows, handles, packed buffers, and combine metadata. Keeping that split prevents backend names from obscuring the trace ledger introduced in Section 21.1: the launch surface selects an executable path, while dispatch proves that the path carries row identity, logical expert ids, and routing weights intact.

Configuration surface	Runtime object it selects	Contract question
Dispatch backend	Dispatcher class and all-to-all record format.	Which dispatch and combine record format carries row identity across ranks?
Runner backend	Expert runner class and quantization method.	Which expert-major layout, dtype, scale metadata, and block alignment can the runner consume?
Dispatcher mode	Normal or low-latency dispatch mode.	Which count, buffer, and handle fields must combine interpret?
Initial placement and redundant-expert policy	Expert-location metadata and placement dispatch.	Does a logical top-k id map to the intended physical replica set?
Placement-balancer settings	Recorder windows, rebalance intervals, layer chunks, and utilization thresholds.	Is a placement update based on a named count window and applied between MoE steps?

Table 21.6.: Configuration surfaces for MoE runtime selection. Each surface selects an object that answers a different contract question.

21.4. Dispatch, All-to-All, and Combine

Expert layers reorder token rows, then must return them to request order.

After routing, the MoE layer has a sparse formula but not yet an executable layout. Dispatch converts token-major top-k records into expert-major work: it maps logical expert ids through the current placement metadata, moves or permutes hidden rows to the physical experts that own them, and records enough inverse state for combine. Write that dispatch layout as Π_{disp} when the chapter needs the layout as a single object. Expert parallelism adds a communication edge because some selected experts may live on another rank.

For the four-token trace, the same row assignments have two useful views:

$$\begin{aligned}
&\text{token-major, first half:} && (t_0, E_0), (t_0, E_1), (t_1, E_0), (t_1, E_2),\\
&\text{token-major, second half:} && (t_2, E_0), (t_2, E_3), (t_3, E_1), (t_3, E_2),\\
&\text{expert-major, first groups:} && E_0 : \{t_0, t_1, t_2\};\ E_1 : \{t_0, t_3\};\\
&\text{expert-major, final groups:} && E_2 : \{t_1, t_3\};\ E_3 : \{t_2\}.
\end{aligned}$$

The rows are unchanged; only the execution order changes.

Definition 21.4.1 (Expert-parallel all-to-all) *Expert-parallel all-to-all is the collective exchange in which each rank q sends the token slices*

$$\{(t, j, x_t, e_{t,j}, \alpha_{t,j}) : \text{owner}(e_{t,j}) \neq q\}$$

selected for experts owned by other ranks and receives the token slices selected for its local experts. The payload includes enough row and routing metadata for combine to invert the exchange.

Expert-parallel all-to-all (Definition 21.4.1) differs from all-reduce: ranks do not all receive the same summed tensor. Each rank receives the subset of routed token work that belongs to its resident physical experts, and the combine path

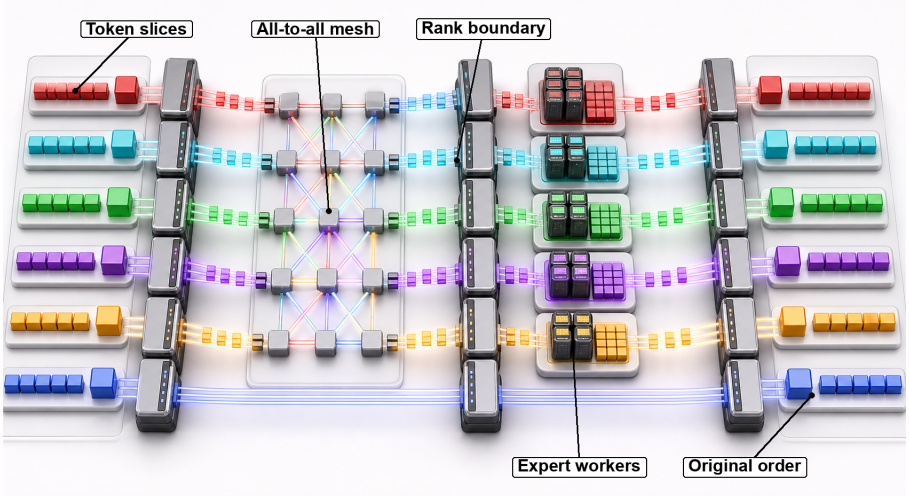


Figure 21.3.: Expert-parallel all-to-all movement. Token slices cross rank boundaries to reach the physical experts that own their selected work, while the right-hand return lanes remind the reader that combine still needs row-order metadata.

Table 21.7.: Routed token-row placement around MoE all-to-all. The rows show how logical routes become owner-local expert work.

Token row	Logical routes	Owner after placement	Movement
t_0	E_0, E_1	q_0, q_0	Both copies stay on rank 0.
t_1	E_0, E_2	q_0, q_1	The E_2 copy crosses to rank 1.
t_2	E_0, E_3	q_0, q_1	The E_3 copy crosses to rank 1.
t_3	E_1, E_2	q_0, q_1	The E_2 copy crosses to rank 1.

later uses dispatch metadata to return weighted outputs to the original token rows. Figure 21.3 draws the same asymmetry as movement between owner ranks followed by a row-order return path.

Example 21.4.1 (Two-rank expert exchange) Suppose rank 0 owns E_0, E_1, rank 1 owns E_2, E_3, and token t_0 selects E_1 and E_3. Rank 0 can evaluate the E_1 slice locally, but the E_3 slice must move to rank 1. At the same time, rank 1 may send a token slice for E_0 back to rank 0. After expert execution, combine uses the original row ids and routing weights so t_0's two expert outputs reduce back into row 0.

For the chapter's four-token running trace, assume the scheduler batch is resident on rank 0 before expert dispatch and the two-rank placement is $E_0, E_1 \mapsto q_0$, $E_2, E_3 \mapsto q_1$. The eight routed assignments then sit as follows before and after the all-to-all:

After dispatch, rank 0's expert-major buffer contains the five copies for E_0 and E_1, while rank 1's buffer contains the three copies for E_2 and E_3. Combine must still return exactly four rows. It uses the stored original row ids and routing weights, not the expert-major buffer order, so the three remote copies above are reduced back into t_1, t_2, t_3 rather than appended as new request rows.

Algorithm 8: MoE dispatch, expert execution, and combine for one layer. Routing ids, weights, placement, dispatch metadata, and inverse layout are one contract so expert parallelism does not lose token-row identity.

Input: Token hidden rows X; router logits G; top-k policy π_{top}; placement map M; dispatcher D
Output: Output rows Y in the same token order as X

compute top-k expert ids and weights from G under π_{top};
if *the selected routing mode is incompatible with padding, remapping, or scaling* **then**
    return *configuration error*;
map logical expert ids through M to physical dispatch ids while retaining the logical ids and weights;
use dispatcher D to permute or exchange hidden rows by physical dispatch id;
run each local expert on the rows delivered to it;
use the matching combine metadata to invert the dispatch layout;
foreach *original token row* i **do**
    reduce exactly the logical expert outputs selected for i with their routing weights;
trim padding and apply the configured TP or EP reduction if required;
return Y *in original token-row order*;

Definition 21.4.2 (MoE combine) *MoE combine is the inverse of dispatch for one routed layer: it gathers the expert outputs produced for each original token row and reduces them with the stored top-k routing weights into one output row. For row t, the logical reconstruction is*

$$y_t = \sum_{(e,\alpha)\in R_t^{\text{route}}} \alpha\, u_{t,e},$$

where $u_{t,e}$ is the expert output returned by the dispatch path for the logical expert e.

Combine uses original row ids and routing weights, not the permuted expert-major order.

If row t_0 routed to experts E_1 and E_3 with weights 0.7 and 0.3, then $R_{t_0} = \{(E_1, 0.7), (E_3, 0.3)\}$ and combine reconstructs $y_{t_0} = 0.7u_{t_0,E_1} + 0.3u_{t_0,E_3}$ in the original row position.

Dispatch-combine contract trail

`FusedMoE.forward` preserves the original hidden dimension, calls the selected dispatcher, runs the quantization-specific expert backend, invokes the matching combine path, trims any padded hidden dimension, and all-reduces only when the layer is configured to reduce MoE TP or EP results. This sequence is why dispatch format is a correctness boundary rather than a performance detail: once a runner consumes a permuted layout, MoE combine (Definition 21.4.2) must receive matching inverse metadata.

Read the algorithm as the four-token trace above made executable. Token-major top-k records first become expert-major work buffers; expert-parallel all-to-all moves only the slices whose physical experts live on another rank. The dispatcher records original row ids and routing weights before that shuffle, because combine must reduce expert outputs back into exactly the original token rows rather than returning the expert-major order.

Algorithm 8 dispatches tokens to selected experts, executes expert-local work, and combines the routed outputs back into token order. It first computes top-k routing and rejects incompatible routing modes. It then maps logical experts to physical placement, dispatches rows, and runs the local experts. The combine phase inverts the dispatch layout, reduces selected expert outputs back into

each original token row, trims padding, reduces across parallel groups when required, and returns row-ordered outputs.

DeepEP as a Dispatcher Backend.

DeepEP supplies one dispatch format inside the execution block while the MoE layer formula stays unchanged. At the server surface, it is selected by the `deepep` value of `--moe-a2a-backend`. Inside the runtime, `DeepEPDispatcher` wraps the DeepEP buffer, creates SGLang dispatch-output records, and hands those records to the selected MoE runner and combine path. DeepEP owns communication layout and handle state. SGLang still owns router policy, placement metadata, runner eligibility, and final row reconstruction.

Definition 21.4.3 (DeepEP dispatch) *DeepEP dispatch is the expert-parallel communication step in which SGLang sends routed hidden-state rows and their routing metadata to the ranks that own the selected physical experts, then later uses matching DeepEP combine state to return expert outputs to the token rows that requested them. Write the resolved dispatch record as*

$$D_{\text{ep}} = (X, \text{topk}, P, C, h),$$

where X are routed hidden rows, P is physical-expert placement, C are count windows or masks, and h is the DeepEP handle state consumed by combine.

The reader-level distinction for DeepEP dispatch (Definition 21.4.3) is the resolved DeepEP record. Normal mode is the throughput-oriented path and carries per-rank, per-RDMA-rank, and per-expert count windows with normal-mode handle state. Low-latency mode is the decode-oriented path and carries packed receive buffers, masked or expected counts, ids, weights, and low-latency handle state. Automatic mode chooses one of those formats before dispatch materializes buffers, so combine must consume the resolved format rather than a generic DeepEP-shaped tensor [5, 24].

For the running trace, DeepEP only changes how the eight routed assignments move between ranks. It may pack the rows, split them by rank, and attach normal or low-latency count windows. The row-restoration obligation is the one stated by Algorithm 8.

Backend-specific dispatchers refine this general contract rather than changing the MoE formula. DeepEP, FlashInfer, and local paths can disagree about count windows, handles, packed buffers, and scale metadata, but they still owe the same row-restoration evidence to combine. The configuration surface in Section 21.3 names the executable path; dispatch proves that the named path carries the row-major sparse formula into expert-major work and back.

21.5. Expert Runner Layouts and Kernel Eligibility

First-read note. Read this section as the point where routed rows become runner input. Kernel-family details are reference material for backend reviews; the main question is simpler. Which slots are real routed assignments, which slots are padding, and which physical expert owns the rows the runner will consume?

Keep the performance story in that order. Router policy decides the real assignments, placement decides the physical expert owner, the runner layout adds padding or permutation for a legal kernel shape, and combine proves that the weighted outputs return to token-major order. Eligibility tables below are therefore hardware feature lists and evidence checklists: each row says which extra evidence the runner needs before a faster path is allowed to preserve the same sparse formula.

Dispatch produces expert work, but the runner decides how that work becomes matrix multiplication. A dense MLP sees one row-major tensor and one pair of weight matrices. An MoE runner sees many smaller per-expert row groups, often after all-to-all, sorting, block padding, quantized activation packing, and backend-specific permutation. The runner has to keep dispatcher output aligned with the local expert rows the kernel will consume. It also keeps dtype, scale metadata, and the combine layout attached to those rows.

Definition 21.5.1 (Grouped expert GEMM) *Grouped expert GEMM is the runner contract that evaluates per-expert MLP matrix products for a set of expert-major token blocks in one grouped kernel schedule, rather than launching an unrelated GEMM for every expert. For physical experts p with real counts N_p and block size q, the runner consumes the padded row groups $q\lceil N_p/q \rceil$, the local expert range, and the dtype or scale layout required by the selected backend.*

Grouped expert GEMM (Definition 21.5.1) is motivated by launch and occupancy efficiency. After routing, the runtime no longer has one dense matrix multiply over T rows; it has many smaller expert MLPs over N_p rows. Launching each expert as a separate tiny matrix multiply can waste time in launches, poor tile occupancy, and repeated metadata setup. Grouped GEMM sorts rows into expert-major blocks, describes all valid groups to one runner, and lets the runner schedule the groups as one MoE kernel path. Grouping's correctness obligation is the

Runner fact	Evidence carried to the kernel	Failure if mismatched
Real routed count	Per-expert row counts before padding.	A padding slot is treated as model work, or a real row is skipped.
Physical expert ownership	Local expert interval and placement snapshot.	The runner applies weights for a different physical expert instance.
Payload and scale layout	Activation dtype, weight dtype, and scale metadata.	The kernel accepts the shape but decodes the values under the wrong numeric layout.
Combine layout	Inverse permutation and top-k slot metadata.	The weighted result returns to the wrong token row.

Table 21.8.: Expert-runner facts carried to MoE kernels. Each fact has kernel evidence and a failure mode if the record is inconsistent.

serving contract from Section 21.1: the runner may change layout and padding only when combine can still reconstruct the selected sparse row.

Example 21.5.1 (Grouped GEMM from the four-token trace) With one physical instance per logical expert, the running trace has physical counts $(3, 2, 2, 1)$. A grouped runner with $q = 4$ receives four expert groups with padded shapes $4, 4, 4, 4$, even though only eight of the sixteen slots are real routed assignments. Triton, DeepGEMM, CUTLASS, and FlashInfer-class runners can differ in dtype, scale layout, and pre-/post-permute format, but the grouped schedule still represents the same sparse formula.

Runner layout adapter.

For dispatcher output D_{out} and runner G, call $(P^G_{\text{pre}}, P^G_{\text{post}})$ a runner layout adapter when $P^G_{\text{pre}}(D_{\text{out}})$ is a valid input for G, and $P^G_{\text{post}}(G(P^G_{\text{pre}}(D_{\text{out}})))$ is a valid combine input for the active dispatcher.

For a grouped Triton runner, P^G_{pre} might sort routed rows by expert and pad each expert block to the kernel tile size; G then runs one grouped GEMM table; and P^G_{post} restores the token-major order and top-k slot metadata that combine expects.

`MoeRunner` either uses a fused dispatcher/runner function or retrieves registered pre-permute and post-permute methods from `PermuteMethodPool`. Missing registrations are assertion failures because the runner would otherwise consume rows in a layout it does not understand. A Triton path can build sorted and padded token ids before expert GEMMs, while a no-permutation path can pass through layouts already prepared by the communication backend. Backend selection depends on the layout compatibility proof between dispatch, runner, and combine.

Definition 21.5.2 (Quantized-runner eligibility) *For backend B, activation dtype a, weight dtype d, scale metadata σ, local expert interval $\mathcal{L}_q^{\text{expert}}$, and device capability h, execution is legal only when an eligibility predicate*

$$Q(B, a, d, \sigma, \mathcal{L}_q^{\text{expert}}, h) = \text{true}$$

is satisfied by the runner and kernel implementation.

Q is a legality predicate; smaller dtype still fails without matching scale and device metadata.

For example, an FP8 blockwise runner may make Q true only when $a = \text{bf16}$, $d = \text{fp8}$, σ contains the expected per-block scales, $\mathcal{L}_q^{\text{expert}}$ names the local expert interval owned by this rank, and h supports the kernel's tile instructions. A missing scale tensor makes Q false even when the expert weights have the right shape.

Quantized MoE makes runner eligibility visible. FP8 blockwise experts need per-block scale tensors and kernels that support the target architecture. FP4, MXFP4, W4A8, CUTLASS, FlashInfer, and AITER paths each bind a different set of dtype, scale-layout, local-expert-range, and device-capability assumptions. Reducing hidden payload bytes is not enough if the runner needs activation scales, block scales, or all-gathered expert weights in a layout the dispatcher did not produce.

Pipeline stage	Representative kernel or record	Meaning in the sparse formula
Route	`topk_softmax`, `topk_sigmoid`, or fused router output.	Defines the logical expert ids and weights for each real token row.
Prepare	`prepare_moe_input`, `problem_sizes1`, `problem_sizes2`, `expert_offsets`.	Counts real assignments and describes expert MLP problems.
Align and permute	`moe_align_block_size`, `sorted_token_ids`, `experts_ids`, `num_tokens_post_pad`.	Adds runner padding without creating new routed model work.
Quantize and execute	Activation quantization plus `fp8_blockwise_scaled_grouped_mm`, `cutlass_w4a8_moe_mm`, or FP4/NVFP4 runner paths.	Runs expert-local matrix products under the selected dtype and scale layout.
Shuffle, weight, and reduce	`apply_shuffle_mul_sum`, `permutation`, `factors`.	Uses inverse permutation to fetch expert-major outputs, applies top-k weights, and writes token-major rows.
Already-token-major reduction	`moe_sum_reduce`, `routed_scaling_factor`.	Sums an already token-by-top-k output layout across the routed axis.

Table 21.9.: Sparse grouped-GEMM pipeline stages. The stages connect representative kernels or records to the sparse cost formula.

Preparing and Aligning Expert Problems.

Before a grouped GEMM can run, the runtime has to turn token-major router output into expert-major problem descriptions. The motivation for `prepare_moe_input` and `moe_align_block_size` is that grouped kernels want regular problem tables, offsets, and padded blocks, while the router emits irregular top-k assignments in token order. These helpers are therefore part of the kernel family, not incidental packing code.

The intuition is a two-stage ledger. `prepare_moe_input` counts how many routed assignments belong to each expert, writes two (M, N, K) problem-size tables for the up/gate and down projections, computes expert offsets and optional block-scale offsets, and emits input/output permutations. Then `moe_align_block_size` rounds each expert's row count to the block size that the selected runner consumes, then fills the sorted-token and block-owner records. The outputs `sorted_token_ids` and `experts_ids` separate sorted slots from block owners. `num_tokens_post_pad` records the padded total, making real routed rows and padding rows distinguishable evidence for the grouped GEMM.

Reading the chain left to right prevents two common mistakes. First, `topk_softmax` and `topk_sigmoid` define the sparse row contract consumed by every later stage. Their speedup over a generic top-k call is secondary to that semantic role. Second, `prepare_moe_input` and `moe_align_block_size` manufacture the problem table that tells a grouped GEMM which expert rows are real, which rows are padding, and where each result must return. Third, combine is not a cosmetic inverse permutation: `apply_shuffle_mul_sum` applies the selected routing weights while returning rows to token-major order, while `moe_sum_reduce` reduces the top-k expert contributions for runner paths that already hold

token-by-expert outputs in a reducible layout. Both kernels must preserve the same logical formula, $\sum_{j=1}^{k_t} \alpha_{t,j} E_{e_{t,j}}(x_t)$, even when their buffer layouts differ.

The layout distinction is the contract. `apply_shuffle_mul_sum` expects expert outputs shaped as $[m \cdot k, d]$, an output buffer shaped as $[m, d]$, a $[m \cdot k]$ permutation from token-major routed slots to expert-major result rows, and optional $[m \cdot k]$ factors such as top-k weights. For token row i, it implements

$$\text{out}_i = \sum_{j=0}^{k-1} f_{i,j} \, \text{input}_{\text{perm}_{i,j}},$$

where the multiplication is applied elementwise across the hidden dimension. `moe_sum_reduce` has a different precondition: its input is already laid out as [token_num, topk_num, hidden_dim], so it sums the middle axis and applies the scalar `routed_scaling_factor`. A runner that still has expert-major rows needs the shuffle form; a runner that has already packed each token's contributions beside that token needs the reduction form. This is a first-read boundary: the exact kernel entrypoints are advanced, but the layout choice is not optional. Flat routed rows use $[m \cdot k, d]$; restored token/top-k tensors use [token, topk, hidden].

Implementation Sketch: From Router Slots to Kernel Tables.

The implementation bridge is a set of counts, scans, and row maps. SGLang does not ask the grouped GEMM to rediscover the MoE semantics. It gives the runner a table whose M values count real routed assignments, a separate alignment record whose padded slots satisfy the kernel's block shape, and a combine map that proves where each expert result returns.

Kernel eligibility depends on both expert layout and the token grouping produced upstream.

Algorithm 9: Preparing real MoE grouped-GEMM problem tables. Expert counts and token permutation maps describe only routed tokens, before runner block padding is introduced.

Input: Routing array A = topk_ids shaped $[m, k]$; expert count E; runner block size q; hidden widths n and h; optional route factors F
Output: Grouped-GEMM problem tables and token permutation maps

foreach *expert* $e \in \{0, \dots, E-1\}$ **do**
  count $c_e = \#\{r : A_r = e\}$;
  set `problem_sizes1[e]` to $(c_e, 2n, h)$ and `problem_sizes2[e]` to (c_e, h, n);
prefix-scan real counts into `expert_offsets` and optional `blockscale_offsets`;
foreach *token-major routed slot* $r = i \cdot k + j$ **do**
  append row i to expert A_r's segment and write both permutation maps;
return *real problem sizes, expert offsets, and token permutation maps*;

Example 21.5.2 (Four-token $k = 2$ prepare, align, and combine trace) Let `topk_ids` be

$$\begin{bmatrix} 0 & 1 \\ 0 & 2 \\ 0 & 3 \\ 1 & 2 \end{bmatrix}, \qquad k = 2, \quad E = 4, \quad q = 4,$$

where q is the grouped-runner block size. Token-major routed slots are numbered $r = i \cdot 2 + j$, so slots $0, \dots, 7$ are the eight real assignments. Counting by expert gives

$$c = (3, 2, 2, 1), \qquad \text{expert_offsets} = [0, 3, 5, 7, 8].$$

These offsets describe real work. The grouped runner then pads each expert to a multiple of q, giving padded counts $(4, 4, 4, 4)$, padded offsets $[0, 4, 8, 12, 16]$, and `num_tokens_post_pad`=16.
One legal `sorted_token_ids` record is

$$[0, 2, 4, 8,\ 1, 6, 8, 8,\ 3, 7, 8, 8,\ 5, 8, 8, 8],$$

where the four padded expert blocks are $E_0 = (0, 2, 4, 8)$, $E_1 = (1, 6, 8, 8)$, $E_2 = (3, 7, 8, 8)$, and $E_3 = (5, 8, 8, 8)$. Sentinel 8, equal to `numel(topk_ids)`, marks padding, not a ninth routed slot. The block-owner record is `experts_ids` $[0, 1, 2, 3]$ for the four padded blocks. Grouped GEMM consumes real problem sizes $M = (3, 2, 2, 1)$ while using the padded slots for launch shape. If token t_0's routing weights are $(0.6, 0.4)$, its two real routed slots are 0 and 1, which appear at expert-major positions 0 and 4. The shuffle-combine row is therefore

$$y_{t_0} = 0.6\, u_{\text{pos}\,0} + 0.4\, u_{\text{pos}\,4}.$$

Padding positions containing sentinel 8 must not enter this gather-weight sum.

Algorithm 9 is the real-count half. First, it counts routed slots per expert and builds GEMM problem sizes from those counts. Then it prefix-scans offsets. Finally, token-major routed slots write the permutation maps that later combine paths will use.

Algorithm 10: Aligning MoE tokens to runner block size. Padded block records use the real counts but are not themselves the real expert problem sizes.

Input: Real expert counts c_e; runner block size q; routing array A
Output: Aligned token ids, block expert ids, and padded token count

round c_e to $p_e = q\lceil c_e/q \rceil$, prefix-scan p_e, and store the total in `num_tokens_post_pad`;
fill `experts_ids`, real `sorted_token_ids`, and sentinel numel(A) padding entries;
return *the runner-facing padded block record*;

Algorithm 11: Combining MoE expert outputs back to token rows. Expert-major outputs are gathered through the permutation map, while token-major buffers reduce their routed axis directly.

Input: Grouped expert outputs; permutation maps; optional route factors F
Output: Token-major output rows

run the selected grouped expert GEMM on real problem sizes and aligned runner blocks;
if *expert outputs are still expert-major* **then**
    gather by `output_permutation`, multiply by F, and sum by token row;
else
    reduce the already [token, top-k, hidden] buffer across its routed axis and apply `routed_scaling_factor`;

Algorithm 10 is the alignment half. First, real counts are rounded to the runner block size. Then padded offsets define launch shape. Finally, sentinel rows fill padding slots so launch geometry changes without adding real routed tokens.

Algorithms 9, 10, and 11 keep the eligibility boundary visible. First, `prepare_moe_input` builds real problem tables and permutations. Then `moe_align_block_size` builds the runner-facing padded block record. Finally, the selected combine helper returns token-major output rows.

The `apply_shuffle_mul_sum` helper is the gather-weight-reduce form for expert-major outputs. The `moe_sum_reduce` helper is the direct reduce form for a token-major [token, top-k, hidden] buffer.

Reference source table. The helper table below is for source readers who want to match the contract to implementation mechanisms. The main argument has already been stated: real-count tables, padded-block tables, and combine maps are related but distinct records.

Two details are easy to misread in source. First, `numel(topk_ids)` is a sentinel for padded `sorted_token_ids` entries; it is not another token row. Second, `experts_ids` names the owner of each padded runner block. It can therefore have entries for alignment capacity that never correspond to real routed assignments. A review should check that real counts drive the grouped GEMM problem sizes, padded counts drive launch shape, and combine consumes the layout actually produced by the runner.

Figure 21.4 shows the handoff from token-major top-k records to grouped-runner metadata: counts and offsets preserve real work, while padded slots serve kernel shape.

Helper	Implementation mechanism to recognize	Source-reading consequence
`prepare_moe_input`	One block per expert counts occurrences in `topk_ids`; a one-thread prefix pass writes `expert_offsets` and optional `blockscale_offsets`; a second per-expert pass writes `input_permutation` and `output_permutation`.	`problem_sizes1` and `problem_sizes2` describe real rows, not padded runner slots.
`moe_align_block_size`	The general path counts into shared memory, rounds counts to `block_size`, prefix-scans padded counts, writes `experts_ids`, then sorts token slots with atomic positions; a small-batch path fuses these steps in one block.	`num_tokens_post_pad` and `experts_ids` describe aligned blocks, while sentinel `sorted_token_ids` entries are padding.
`moe_align_block_size` JIT	The generated path keeps the same contract, adds a > 1024-expert v2 scan, and caps the shifted expert-id domain at 4096 entries.	JIT changes launch generation and expert-count coverage, not the meaning of the alignment buffers.
`apply_shuffle_mul_sum`	One CUDA block owns one output token row; threads walk vectorized hidden chunks, gather expert-major rows through `permutation`, optionally multiply by `factors`, and store the sum.	The permutation is part of correctness, not a post-processing convenience.
`moe_sum_reduce`	The wrapper checks contiguous $[token, topk, hidden]$ input, then selects among a BF16 vector fast path, small-token kernels, warp-per-token kernels, and specialized top-k cases.	This helper is only correct after the runner has already placed each token's expert contributions beside that token.

Table 21.10.: Expert-runner helper recognition guide. The helpers point source readers to implementation mechanisms and their reading consequences.

Example 21.5.3 (Eight assignments, sixteen runner slots) In the four-token trace, the logical expert loads are $(3, 2, 2, 1)$. With block size $q = 4$, `moe_align_block_size` gives each expert a four-row runner block, so the grouped path has sixteen slots even though only eight routed assignments are real. `prepare_moe_input` must still report problem sizes whose M values are $3, 2, 2, 1$ for the real rows. Counting the padded slots as semantic work would double the sparse formula for this toy trace.

Example 21.5.4 (Two combine layouts for one token) Suppose token t_0 routed to E_1 and E_3 with weights 0.7 and 0.3. If the expert runner writes those outputs in expert-major rows, then `apply_shuffle_mul_sum` uses the two permutation entries for t_0 to find u_{0,E_1} and u_{0,E_3}, multiplies them by the two factors, and writes $0.7u_{0,E_1} + 0.3u_{0,E_3}$ into token-major row 0. If another runner has already produced a contiguous [token, top-k, hidden] buffer, then `moe_sum_reduce` can sum the second axis directly. The two kernels compute the same sparse row only when their input-layout assumptions are true.

The examples above leave one source-reading question: which implementation records carry this runner contract? The helper table above names the broad

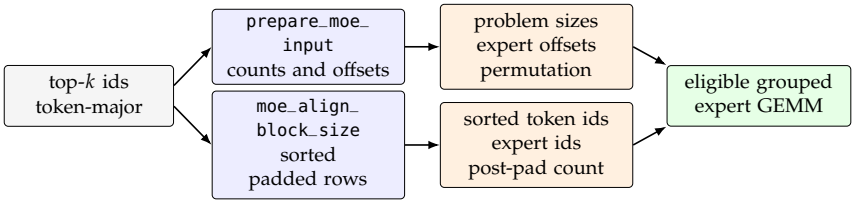


Figure 21.4.: MoE prepare and alignment kernels turn irregular top-k assignments into grouped-GEMM metadata. Counts and offsets describe real expert work; padded sorted-token slots describe alignment that must not be counted as extra routed assignments.

anchors: Python extension wrappers, real-count builders, alignment kernels, JIT alignment, and reducer layouts. Those records are the bridge from the four-token trace to grouped-runner eligibility. They preserve real counts, expert offsets, permutation maps, sorted token ids, block owners, padded totals, and reducer preconditions; JIT changes how the alignment kernel is generated, not what those records mean.

Quantized runner variants then add backend-specific evidence to the same record. The FP8 grouped-GEMM case is implemented in `sgl-kernel/csrc/moe/fp8_blockwise_moe_kernel.cu`. It is exposed as `fp8_blockwise_scaled_grouped_mm`; the namespaced Python extension is `sgl_kernel::fp8_blockwise_scaled_grouped_mm`. Its wrapper forwards activation, weight, output, and scale pointer tables; FP8 payloads; float32 scale tensors; stride tensors; scale-layout descriptors; `problem_sizes`; `expert_offsets`; and workspace. For each expert e, `problem_sizes` supplies (M_e, N_e, K_e) for real routed rows, `expert_offsets` selects the expert-major slice, and the pointer tables select the matching activation, weight, output, and scale bases. The launch is therefore one grouped schedule over many expert-local GEMMs, not permission to count padded alignment slots as routed tokens.

The FP8 case is representative. W4A8 CUTLASS, FP4, NVFP4, FlashInfer, and TRT-LLM-style paths change payload packing, scale layout, architecture support, or local expert ownership checks. They do not introduce a new row-identity rule: the same `problem_sizes`, `expert_offsets`, payload pointers, scale metadata, and combine target must still describe the current routed step. The FP8 unit test makes that intended semantics explicit by comparing each expert slice with a dense matrix-multiply baseline: one grouped launch, many expert-local GEMMs, and real routed rows separated from padding.

Grouped-runner evidence checklist.

The checklist below is the part to keep from the reference snapshot. It asks what any selected runner must prove, regardless of the backend family or quantized payload format.

The runner stage is therefore where performance and correctness meet. A kernel can be fast in isolation and still be ineligible for the current MoE step if it does not consume the dispatcher layout, scale metadata, block padding, and expert ownership that the step actually produced.

Runner evidence	Four-token trace anchor	Failure if missing
Real row counts	Problem sizes use $(3, 2, 2, 1)$, not the sixteen padded slots.	Padding is counted as routed expert work, or a real row is skipped.
Padding convention	Sentinel entries fill the $q = 4$ runner blocks.	Alignment slots enter combine as if they were token assignments.
Physical expert ownership	The local interval and placement snapshot name the physical instances that own each logical expert.	A token slice runs through a slot materialized for another logical expert.
Numeric layout	Payload dtype, scale tensors, strides, and device capability satisfy Q.	The kernel accepts the shape but interprets values under the wrong scale or payload format.
Combine layout	Pre-/post-permute methods and reducer layout match the dispatcher output.	Expert-major outputs reduce into the wrong token row.

Table 21.11.: Runner evidence checklist for selected MoE backends. The rows name the trace anchor and failure guarded by each runner fact.

21.6. Routed-Row Correctness Invariants

Shape-correct tensors are not enough; row and expert identity must also survive.

The MoE layer is correct only when dispatch and runner optimizations preserve the sparse formula selected by the router. Most bugs in this area do not look like obvious shape failures. A stale placement map, mixed DeepEP mode, or wrong scale layout can still produce tensors of the expected rank. The audit must therefore check row identity, logical expert identity, routing weights, padding state, and dispatch format together.

Invariant 21.6.1 (MoE step-alignment)**.** *Each token output must be reduced from exactly the experts selected for that token under the same routing probabilities used to dispatch it.*

For one sampled token row, the trace should preserve both the value that changes at each boundary and the identity that must not change:

Padding and capacity state.

Padding and capacity state is the runtime record that separates real routed assignments from buffer slots used for padded token rows, grouped-GEMM block alignment, capacity checks, or unsupported overflow states. Only valid non--1 assignments with a logical expert id and routing weight count as model work.

Let T be the number of real token rows and k the router top-k. The valid routed-assignment count is

$$R = \left| \{(t, j) : 0 \le t < T,\ 0 \le j < k,\ \text{topk_ids}_{t,j} \neq -1\} \right|.$$

For a physical expert p with N_p real rows and grouped-kernel block size q, the runner may allocate $q\lceil N_p/q \rceil$ slots. The difference between allocated slots and real rows is alignment slack without extra model work.

Table 21.12.: Routed-row identity across MoE boundaries. Each boundary changes a row record while preserving the identity needed for combine.

Boundary	Sample row record	Identity that must survive
Router scores	Row t_0 has logits $G_{0,*}$ and uses active `TopKConfig`.	The row is still request row 0.
Top-k choice	Logical ids (E_1, E_3) with weights $(0.70, 0.30)$.	The selected logical experts and weights are the sparse formula for t_0.
Placement map	Physical dispatch ids (P_1, P_3), perhaps on different ranks.	Each physical id must map back to E_1 or E_3.
Dispatcher format	The row enters a standard, DeepEP, or FlashInfer dispatch record.	Packing, all-to-all, and padding may change layout, not ownership.
Runner input	Local expert buffers receive hidden slices for P_1 and P_3.	The runner must not reinterpret a slice as another expert or row.
Combine output	Combine writes $0.70E_1(x_{t_0}) + 0.30E_3(x_{t_0})$ back to row 0.	The final row must match the original token row and logical choices.

Table 21.13.: Dispatcher formats and combine obligations. The dispatch output must carry enough information for combine to restore request order.

Dispatcher format	Dispatch output carries	Combine must know
Standard	Expert-grouped hidden rows, top-k ids and weights, and the permutation or inverse layout used by the local runner.	Original token row for each real assignment, selected logical expert, routing weight, and padded slot mask.
DeepEP normal	Received hidden states, top-k ids and weights, per-expert receive counts, and normal-mode handle state.	Count windows and returned rows for each rank and expert.
DeepEP low-latency	Packed received states, masked or expected token counts, and low-latency ids and weights.	Packed-token masks, expected counts, and row identity before reducing packed outputs.
FlashInfer or custom	Backend-owned dispatch records, physical expert layout, and possibly activation-scale metadata.	Matching combine record, scale layout, physical-to-logical interpretation, and output row ids.

Definition 21.6.2 (Dispatcher format contract) *A dispatcher format contract is the typed agreement between dispatcher, runner, and combine about hidden-row order, physical expert ids, logical expert interpretation, routing weights, padded-slot masks, count windows, activation-scale metadata, and inverse-layout state for one MoE execution. For backend b, write the contract as*

$$C_b = (O_b, P_b, W_b, M_b, I_b),$$

for row order, physical experts, routing weights, padding masks, and inverse layout.

Invariant 21.6.3 (Padding and capacity invariant)**.** *Every non-padding routed assignment must either reach exactly one compatible physical expert instance or fail before expert execution; padding and alignment slots may consume buffer capacity but must not add, remove, or reweight token outputs.*

The dispatcher hook surface preserves the same contract while allowing auxiliary behavior to attach to MoE execution. `BaseDispatcher` can wrap dispatch and

Failure mode	Why shape checks can miss it	Review evidence
Stale EPLB map	The row still reaches an expert-shaped weight tensor.	Current location metadata and physical-to-logical map.
DeepEP mode mismatch	Normal and low-latency records both contain rows, ids, and weights.	Resolved `DeepEPMode`, dispatch output, and matching combine input format.
Padding counted as work	Alignment slots have valid buffer shapes.	Real routed count R, padded slot count, and -1 or mask convention.
Fused shared expert remap error	Shared expert columns look like top-k columns.	Number of fused shared experts and shared-slot remap.
Runner scale-layout mismatch	A grouped GEMM can launch with right-rank tensors.	Runner backend, quantization method, scale layout, block size, and local expert range.

Table 21.14.: Routed-row correctness failure modes. Shape checks can miss these cases, so review evidence must preserve route identity.

combine with pre- and post-hooks. A hook may adjust hidden states, top-k output, dispatch output, combine input, or final hidden states only at named format boundaries. `MoeRunner.run` uses the same discipline for MoE-LoRA: when LoRA is enabled it avoids the fused runner path, derives LoRA hooks from hidden states and top-k ids, and passes those hooks into either a direct-dispatch runner or the permuted runner core.

Example 21.6.1 (Plausible tensor, wrong expert identity) Suppose t_0 selects logical E_0 with weight 0.8 and logical E_3 with weight 0.2. Under an EPLB map, logical E_0 may execute on physical replica P_1. If a stale dispatcher map sends the first slice to a physical slot currently holding logical E_2, combine writes $0.8\hat{y}_1 + 0.2\hat{y}_3$ to row 0 instead of $0.8E_0(x_{t_0}) + 0.2E_3(x_{t_0})$. The failure is an identity failure that matrix-shape checks can miss.

21.7. EPLB, Load Balance, and Overlap Control

Less dense parameter work can still create a hot expert.

Sparse activation reduces average expert work only when the routed assignments are distributed well enough for the step's critical path. If one physical expert receives most assignments, the layer waits for that expert even though the total number of active experts is small. Runtime expert placement is therefore part of MoE serving itself.

Definition 21.7.1 (Expert load) *Expert load is the number of real routed token assignments that a physical expert instance must process in a serving step. For physical expert p,*

$$N_p = \left|\{(t, j) : p \in \Phi_s(e_{t,j}) \text{ and slot } (t, j) \text{ is real}\}\right|.$$

If a step sends 12 real routed slots to physical expert p_2, then $N_{p_2} = 12$ for that step. Padding rows used only for grouped-kernel shape regularity do not increase the load count.

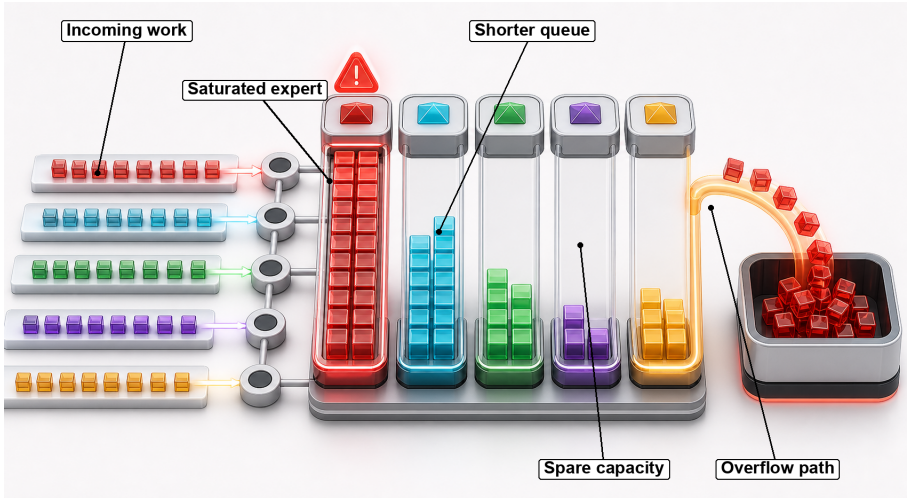


Figure 21.5.: Load imbalance in MoE serving. The full leftmost expert queue is the saturated hot expert, while shorter neighboring queues retain spare slots. Overflowed work on the fallback path shows why the serving bottleneck can be placement and dispatch rather than the total number of active experts.

Expert load is measured on physical instances named by the logical-to-physical map from Definition 21.3.1. A placement policy can change that map for future token rows, but already-dispatched rows still have to combine through the physical instance recorded for their slot.

Figure 21.5 makes the imbalance term visible: one hot physical expert can dominate the step even when neighboring experts have spare capacity.

Definition 21.7.2 (Expert imbalance) *Expert imbalance is the skew of real routed assignments across physical expert instances during a serving step. A simple scalar account is*

$$\rho_{\text{imb}} = \frac{\max_p N_p}{\text{mean}_p N_p},$$

computed over the physical experts participating in the measured window. Larger values mean the step is more likely to wait for one hot expert even when total routed arithmetic is small.

N_p counts real routed slots on physical experts; padding does not count as load.

In the notation of Definition 21.7.2, loads $(8, 8, 8, 40)$ have mean load 16 and $\rho_{\text{imb}} = 40/16 = 2.5$. The layer's critical path is likely dominated by the fourth physical expert even though the total routed-token count is only 64. The reason is concurrency: expert groups are scheduled as parallel work, so the layer cannot combine and return until the busiest participating expert has produced its rows.

EPLB makes placement a mutable runtime object. The manager records logical expert counts over a configured window, checks whether the average utilization rate is low enough to justify a rebalance, computes `ExpertLocationMetadata`, and may apply updates layer by layer in chunks. That metadata contains physical-to-logical maps, logical-to-all-physical maps, valid replica counts, and optional rank-dispatch maps. Static dispatch maps a logical top-k id directly through a partial rank map; dynamic dispatch chooses among valid redundant physical experts. Weight loading must materialize the same logical expert into every local physical replica exposed by the metadata, including fused shared experts that bypass normal routed-expert remapping. The result is the placement snapshot from Definition 21.3.2.

Algorithm 12: EPLB placement update. Rebalancing changes where logical experts live, not the top-k choices or weights already attached to a token step.

Input: Recent logical counts C; current metadata M; routed weights by layer; update layer set $\mathcal{L}_{\text{upd}}$
Output: Updated physical expert placement and dispatch maps for later MoE steps

wait until the EPLB rebalance interval has elapsed;
dump the recorder window and read `logical_count` and average utilization;
if *the utilization threshold says rebalancing should be skipped* **then**
    keep M unchanged and return;
compute new `ExpertLocationMetadata` from C: physical-to-logical map, all physical replicas for each logical expert, and any rank-local dispatch map;
split $\mathcal{L}_{\text{upd}}$ into chunks when the update should yield between layer groups;
foreach *layer chunk* $G \subseteq \mathcal{L}_{\text{upd}}$ **do**
    compare old and new physical-to-logical maps for every local physical slot;
    satisfy unchanged, same-GPU, same-node, and cross-node slots by logical expert id;
    copy the received or local buffer into each destination expert weight tensor;
    update global location metadata only for layers in G;
return *metadata whose logical ids, physical replicas, and dispatch maps agree*;

The migration mental model is count, plan, move, publish. Count logical expert demand over a recorder window. Plan a new logical-to-physical replica map. Move or copy weights by logical expert id so every destination slot holds the expert named by the new map. Publish the updated metadata only after a layer chunk is consistent, so in-flight MoE steps never mix old dispatch metadata with new weight placement.

Algorithm 12 updates expert placement from load feedback by measuring imbalance, choosing migrations, and publishing a revised placement map. It first collects expert-load observations and decides whether a rebalance interval has elapsed. If a rebalance is due, it computes a new logical-to-physical placement while preserving expert ids and replica constraints. The new placement is published only at a step boundary where in-flight dispatch metadata will not change underneath a batch, and the published placement state is then used by later router and dispatcher calls.

Algorithm 12 makes the timing boundary explicit. A rebalance must be treated as a placement-map update between MoE executions while the logical router output for an in-flight token remains bound to its original row. The step that observes a placement map must also observe the matching expert weights and dispatch metadata.

Example 21.7.1 (EPLB changes physical placement, not logical routing) Suppose one layer has four logical experts and the recorder observes logical counts $(120, 20, 20, 20)$ over a window. With one physical instance per logical expert, the hottest physical expert has 120 assignments and the physical mean is 45, giving $\rho_{\text{imb}} = 2.67$. If the deployment provides two redundant physical expert slots, an EPLB placement can expose three physical replicas for logical expert E_0 and one each for E_1, E_2, and E_3. A token routed to logical E_0 may now be dispatched to P_0, P_1, or P_2, but it is still an E_0 expert evaluation with the same routing weight.

The EPLB tradeoff is memory and update traffic versus maximum physical expert load. Rebalancing too often can spend serving time updating placement;

Table 21.15.: EPLB transition evidence. Each phase has an allowed transition and the evidence needed before changing placement state.

Phase	Allowed transition	Required evidence
Before an MoE step	The active physical-to-logical map, valid replica counts, and rank-dispatch map are read.	One placement snapshot is attached to the step's router and dispatcher metadata.
During dispatch, runner, and combine	That snapshot is reused without publishing a replacement for the same in-flight layer work.	The combine record can name the placement snapshot that dispatch used.
Recorder-window boundary	Logical counts are dumped and a candidate EPLB placement is computed.	Candidate map is not consumed by rows already in dispatch or combine.
Layer-chunk update	Expert weights are moved or copied, then metadata is published for the completed chunk.	Every published physical slot contains the logical expert named by the new map.
After publication	Later MoE steps route through the new placement.	Router, dispatcher, weight loader, and combine agree on the same snapshot.

rebalancing too rarely leaves decode steps pinned behind hot experts. The phase table above is the audit rule: each MoE step consumes one published placement snapshot, not a mixture of old weights and new metadata.

Elastic Expert Backup and Missing-Weight Recovery.

The preceding placement account assumes the logical expert needed for a new physical slot can be copied from an old local or peer slot. Elastic expert parallelism adds the harder case: after rank recovery or a placement change, a rank may need a logical expert whose weights are not present in any usable local slot. The runtime then needs a slower but still source-grounded repair path. The current SGLang backup path creates a DRAM copy of routed expert weights, registers the backup and local parameter buffers with the transfer engine, and uses that copy only when the P2P placement update reports missing logical experts.

Definition 21.7.3 (Expert backup record) *An expert backup record is the host-side descriptor that lets an elastic-EP worker reload expert weights without rereading the whole checkpoint. In source,* `BackupDramReq` *carries the producing rank, a checkpoint-name-to-address map, a transfer session id, and the registered buffer size. The record stores the addresses used to materialize weights named by a later placement map.*

Algorithm 13 prepares the backup source of truth. First, backup managers materialize local expert tensors into registered CPU buffers. Then clients register destination parameter storage with the transfer engine. Finally, later repair steps can name byte ranges by logical expert id instead of rereading the whole checkpoint.

Algorithm 14 is the fast movement path. First, each rebalance chunk is classified by where its expert bytes can come from. Then local copies and P2P transfers fill reachable slots. Finally, the placement metadata reports only the logical experts that still need backup or disk repair.

Algorithm 13: Elastic expert backup setup. Backup managers publish DRAM sources and clients register destination parameter buffers before any placement update tries to repair missing experts.

Input: Candidate placement metadata M'; current model expert weights; optional expert backup records B; update layer set $\mathcal{L}_{\text{upd}}$
Output: Registered expert backup sources and destination buffers

when elastic expert backup is enabled, launch one backup manager per node;
each backup manager loads routed expert tensors for its node's logical expert range into one contiguous CPU byte buffer;
register the backup buffer with the transfer engine and publish a `BackupDramReq` to ready clients;
each client registers its resident model-parameter buffers with the transfer engine;
return *backup records and registered destination buffers*;

Algorithm 14: Elastic expert placement update before backup repair. Existing expert weights move through unchanged, same-GPU, free-rider, same-node, or cross-node paths before missing logical experts are repaired.

Input: Candidate placement metadata M'; update layer set $\mathcal{L}_{\text{upd}}$
Output: Updated placement metadata and any missing logical experts

foreach *rebalance chunk* $G \subseteq \mathcal{L}_{\text{upd}}$ **do**
    run the expert-location updater on G;
    classify every destination physical slot as unchanged, same-GPU copy, free-rider copy, same-node P2P, or cross-node P2P;
    execute the planned local copies and P2P transfers keyed by logical expert id;
    update the global placement metadata for G and return any missing logical experts;
return *updated placement metadata and missing logical experts*;

Algorithm 15 is the final repair listing. First, it narrows the checkpoint names to missing logical experts when the model supports precise filtering. Then it prefers the registered backup client, because that path writes directly into local destination parameter buffers. Finally, it falls back to disk reload only when the backup path cannot serve the missing expert bytes.

Algorithms 13, 14, and 15 separate placement planning from weight availability. First, backup managers and clients prepare registered byte ranges. Then the ordinary EPLB update tries cheap materialization: unchanged slots, same-GPU copies, reuse of a slot already filled in the same update, and P2P movement from same-node or cross-node peers. Finally, only the logical experts still missing after that movement are loaded through the backup client or disk fallback. The safety point is the same as the placement snapshot invariant: a later MoE step may consume a new physical slot only after its logical id, destination tensor, and weight bytes agree.

Table 21.16 makes the review checks explicit before the prose follows individual failure cases.

Algorithm 15: Elastic expert backup repair. Missing logical experts are filled from a registered backup client or disk fallback before visible placement can name their physical slots.

Input: Missing logical experts; placement metadata M'; optional backup records B
Output: Local physical expert slots whose weights agree with visible placement

build a checkpoint-name filter for missing logical experts when the model can provide one;
if *a registered backup client is ready* **then**
    find local destination slots from M', map checkpoint names to runner parameters, and batch-read the needed byte ranges into parameter storage;
else
    reload selected expert weights from disk, or reload the full weight set when no precise filter exists;
return *the repaired slots before later MoE steps consume the chunk*;

Example 21.7.2 (Repairing a missing expert after a placement update) Suppose rank r owns physical slots P_2 and P_3. The old map says $P_2 \mapsto E_2$ and $P_3 \mapsto E_3$, but the new EPLB map for layer ℓ requires $P_2 \mapsto E_0$. If no local slot on rank r currently holds E_0, the updater first asks whether some peer can send E_0 through same-node or cross-node P2P. If rank recovery or placement state makes that source unavailable, E_0 is reported as missing for layer ℓ. The backup client then uses the global metadata to see that P_2 is a local destination for logical E_0, reads the registered projection byte ranges for E_0, and writes them into P_2's resident parameter tensors. A later token routed to E_0 can now choose P_2 without changing the sparse formula $E_0(x)$.

Definition 21.7.4 (MoE communication-computation overlap) *MoE communication-computation overlap is a schedule that keeps the same dispatch, runner, and combine dependencies but places some all-to-all, expert, attention, or shared-expert work on different micro-batches or streams so communication latency is hidden behind independent computation. It changes the critical path while preserving the sparse MoE formula. The contract is step-local: TBO and SBO may move work across micro-batches or streams only when dispatch handles, combine inputs, placement snapshots, and row-restoration evidence remain attached to the step that produced them. Write such a schedule as a partial order*

$$\mathcal{O} = (V, \prec),$$

where vertices are dispatch, expert, shared-expert, attention, and combine tasks, and $u \prec v$ records a dependency that overlap must preserve.

SGLang exposes two MoE overlap ideas at the server surface. Two-Batch Overlap (TBO), enabled by `--enable-two-batch-overlap`, splits work into micro-batches and uses yield points so attention or expert work for one micro-batch can run while dispatch or combine for another micro-batch is in flight. Single-Batch Overlap (SBO), enabled by `--enable-single-batch-overlap`, uses dispatcher and runner hooks around dispatch, combine, and down-GEMM boundaries so independent work inside one batch, such as shared-expert computation, can overlap with communication. TBO requires a non-`none` MoE all-to-all backend because there is no expert-parallel communication term to hide when MoE A2A is disabled.

Failure	Why shape checks may miss it	Source-side guard
Metadata moves before weights are repaired	The destination tensor can have the expected shape and dtype while still containing the old logical expert.	The updater copies or receives weights before the rebalance call returns control to later steps.
Stale CPU placement copy	Dispatch and backup lookup can consult a CPU copy that no longer matches the device placement tensors.	Metadata updates refresh the CPU maps used by logical-to-physical lookup.
Backup client not ready	P2P reports a missing logical expert, but no registered DRAM source can serve it.	The manager waits for client readiness; the runner falls back to disk reload if backup is unavailable.
Wrong projection shard	`gate_proj`, `up_proj`, and `down_proj` can share fused runner storage with different slices.	The client maps checkpoint names to `w1`, `w3`, and `w2` destinations before transfer.
Transfer-size mismatch	Source and destination pointers are both legal addresses, so the error can look like corrupted expert math.	The backup client checks that parameter bytes equal the recorded backup byte size before issuing the transfer.
Rank recovery changes active peers	The placement map may name a rank whose communication group is not yet recovered.	Elastic-EP recovery tracks active ranks and broadcasts location metadata after recovered ranks rejoin.

Table 21.16.: Failure checks for elastic expert backup. The relevant invariant is not only tensor shape: physical expert slots must contain the logical expert named by the placement snapshot that dispatch will use.

Example 21.7.3 (Overlap hides latency but not evidence) Suppose a decode step spends 2.0 ms in expert dispatch and combine and 3.5 ms in independent attention or shared-expert work. A perfect overlap schedule could hide at most 2.0 ms of communication under the independent work. The accounting still has to record the same top-k ids, dispatch format, runner backend, combine input, and row-restoration trace; overlap changes when evidence is produced while preserving which evidence is required.

21.8. MoE Cost Account and Measurement Artifact Protocol

The central tradeoff is that sparse activation reduces dense parameter work but introduces routing, permutation, communication, combine, and placement-update overhead. Sparse activation helps only if those overheads stay off the critical path. A useful first account is

$$\begin{aligned} C_{\text{moe}} \approx\ & C_{\text{route}}(T, E) + C_{\text{pack}}(R, d_{\text{model}}) \\ & + \sum_{p} C_{\text{expert}}\left(q \left\lceil N_p / q \right\rceil, d_{\text{model}}, I\right) \\ & + C_{\text{comm}}(R_{\text{remote}}, d_{\text{model}}, b_{\text{elt}}) + C_{\text{combine}}(R, d_{\text{model}}). \end{aligned}$$

Table 21.17.: Local notation for the MoE serving cost account. The symbols separate routing, expert work, and transferred payloads.

Symbol	Meaning in the MoE serving account
T	Real token rows entering the routed layer in this step.
k	Router top-k choices per token row before padding or invalid slots.
R	Valid routed assignments after masking invalid -1 expert ids.
d_{model}	Hidden width of the routed token row.
b_{elt}	Bytes per hidden element for the dispatched payload dtype.
E	Number of logical experts exposed by the model layer.
I	Intermediate width of each expert MLP.
q	Grouped-kernel block size used when padding expert-major rows.
N_p	Real routed assignments assigned to physical expert instance p.
R_{remote}	Routed assignments whose hidden rows cross a rank boundary before expert execution.

Table 21.18.: MoE measurement status. The section provides the cost account and artifact protocol, while completed throughput and placement comparisons remain future measurements.

Evidence surface	Status in this chapter	Claim allowed
Cost account	Defined over routing, packing, expert work, communication, combine, and placement update terms.	Explains which counters a future measurement must preserve.
Artifact schema	Specified below as $A_{\text{moe}} = (I, \theta, R, P, F, G, M)$.	Reproducibility contract for future MoE rows.
Completed MoE measurement	Not present in this edition.	No routed-token throughput, all-to-all speedup, EPLB improvement, or runner backend performance claim.
Implementation evidence	Mapped in Section 21.9.	Source-backed boundary evidence, not a measured cost comparison.

Here E is the logical expert count, I is the expert intermediate width, q is the grouped-kernel block size, N_p is physical expert load, and R_{remote} counts routed assignments whose hidden rows must move to another rank.

Measurement Artifact Protocol.

This subsection defines the artifact schema for MoE measurement claims. The cost account above becomes measurable when one routing or placement choice varies while the model, request trace, hardware, and measurement window stay fixed. The primary reported quantities are expert imbalance, routed-token throughput, and all-to-all volume. Those quantities become evidence only when the artifact ties them to routing, placement, dispatcher, and correctness metadata from the same measured window.

MoE cost rows should separate routed token volume, padding, and communication cost.

This chapter does not report a completed MoE measurement row. Table 21.18 keeps the protocol useful without letting it read as if the benchmark already exists.

MoE measurement artifact.

A MoE measurement artifact is the reproducible record that binds fixed inputs, one changed routing or placement variable, routing and placement metadata, dispatcher and runner formats, correctness gates, and reported latency or throughput

Measured term	Artifact fields	Minimum correctness gate
Routing and packing	`TopKConfig`, top-k ids, weights, masks, valid routed count R, and padded-slot count.	Routed-assignment conservation.
Expert execution	Physical counts N_p, runner backend, local expert range, dtype, scale layout, and block size q.	Runner eligibility and physical-to-logical consistency.
Communication	A2A backend, DeepEP mode when active, per-rank counts, payload dtype, remote assignments, and scale-byte policy.	Matching dispatch and combine formats.
Placement feedback	Recorder window, logical counts, candidate placement, moved weights, layer chunks, and published snapshot.	No in-flight step mixes old weights with new metadata.

Table 21.19.: Measured MoE cost terms and artifact fields. Each term needs a minimum correctness gate before it supports a throughput claim.

metrics for a measured token window. Write the artifact as

$$A_{\text{moe}} = (I, \theta, R, P, F, G, M),$$

where I is the fixed input slice, θ the changed variable, R routing metadata, P placement metadata, F dispatcher and runner formats, G correctness gates, and M measured metrics.

A measurement record preserves the routing trace alongside final tokens per second. The executable-path record includes the MoE server knobs: the all-to-all backend, runner backend, DeepEP mode, expert-dispatch algorithm, EPLB algorithm, recorder mode, recorder buffer size, rebalance interval, rebalance layer chunk size, redundant expert count, minimum rebalance utilization threshold, and whether two-batch or single-batch overlap is enabled. It also includes `TopKConfig`, grouped or biased routing flags, logical top-k ids, weights, logical-to-physical expert maps, dispatcher output format, combine input format, and whether the top-k output was standard, Triton-kernel, or bypassed.

The recorder fields make the experiment reproducible. In statistic mode, the EPLB recorder dumps `logical_count` over the buffered window and `average_utilization_rate_over_window`. In detailed mode, it preserves per-forward-pass records and the last physical-to-logical map. The derived imbalance metric must be explicit, for example $\max_p N_p / \text{mean}_p N_p$, coefficient of variation over physical expert counts, or SGLang's utilization rate. DeepEP volume is derived from per-rank and RDMA-rank token counts multiplied by hidden width and payload dtype bytes, with activation-scale bytes reported separately [5, 24].

The artifact makes every cost term auditable rather than reporting one aggregate throughput number:

The artifact schema is stable across those runs and reuses the same gates established above. Each measured window binds three records: fixed context, row trace, and outcome. Fixed context names the SGLang source commit, model identifier, server knobs, request-trace hash, token-count window, and changed variable. The row trace names the top-k policy, output format, physical maps, physical counts, remote assignments, payload dtype, scale-byte policy, padded-slot count,

Table 21.20.: MoE run-sweep interpretation variables. Placement and dispatcher variables determine what each run can compare.

Run	Placement variable	Dispatcher variable	Interpretation target
Static baseline	EPLB disabled; one placement map for the run.	Fixed A2A backend and mode.	Baseline imbalance and remote volume.
Negative control	Same placement map as baseline.	Same dispatcher, runner, router outputs, and request trace.	Measurement noise and run-to-run drift.
EPLB enabled	Same router outputs; rebalance window and redundant experts varied.	Same dispatcher as baseline.	Whether placement reduces hot physical experts.
Dispatcher mode	Same placement metadata.	DeepEP normal versus low-latency, or FlashInfer versus standard where supported.	Whether communication format dominates.
Runner backend	Same placement and dispatch metadata.	Eligible quantized or fused runner variants.	Whether expert kernel layout dominates.

rebalance decision, and a sampled row-restoration trace. The outcome names throughput, latency distribution, imbalance, all-to-all volume, and the noise band observed by the negative control.

Interpretation fails if any of the core gates fail: R is not the count of non--1 top-k slots, a sampled row cannot be restored to its original logical ids and weights, a physical dispatch id is invalid under the recorded maps, alignment slots are counted as model work, scale bytes are omitted from quantized communication accounting, or a negative control drifts more than the claimed effect.

A sampled row-restoration record can be literal: `row_id=0`; logical ids (E_1, E_3); weights $(0.70, 0.30)$; physical dispatch ids (P_1, P_3); `dispatcher=deepep_normal`; `combine_input=deepep_normal`; no padding slot attached to the sample; and a pass/fail field reserved for the measured run. Without that schema, a faster run can be hard to attribute to better balance, less communication, or a different kernel layout.

The same account gives the measurement checklist. Each term is interpretable only if the measurement record stores the corresponding counter and the trace-ledger evidence that produced it:

The break-even comparison is against the dense MLP that the MoE layer replaces. Sparse activation helps only when the routed path's route, pack, expert, communication, combine, and placement-update terms stay below the dense projection cost for the same token rows. That is not guaranteed by $k \ll E$. If N_p is skewed, the step waits for the hottest physical expert; if q is large relative to N_p, alignment slack turns sparse rows into idle kernel slots; if many assignments are remote, all-to-all traffic can dominate the saved parameter work.

Cost term	Counter to preserve	Trace evidence that makes it valid
C_{route}	Token rows T, logical experts E, top-k policy, score transform, and grouped-routing flags.	`TopKConfig`, logical ids, weights, masks, and shared-expert convention.
C_{pack} and C_{combine}	Valid routed assignments R, padded slots, dispatch format, and inverse-layout work.	Dispatcher output, combine input, row ids, and non-semantic padding evidence.
C_{expert}	Physical counts N_p, block size q, runner backend, dtype, and scale layout.	Local expert interval, physical-to-logical map, quantization metadata, and block-aligned row counts.
C_{comm}	Remote assignments R_{remote}, per-rank counts, payload dtype bytes, and scale-byte policy.	A2A backend, DeepEP mode or equivalent dispatch format, send/receive counts, and placement snapshot.
Placement update cost	Rebalance window, moved weights, layer chunk size, and publication timing.	EPLB recorder window, candidate map, materialized expert weights, and the snapshot consumed by later steps.

Table 21.21.: MoE cost terms and preserved counters. Trace evidence makes each counter valid for the cost account.

Example 21.8.1 (A numeric break-even account) Take $T = 128$ token rows, $d_{\text{model}} = 4096$, FP16 hidden payloads, and a dense MLP intermediate width $I_{\text{dense}} = 14336$. Ignoring activation and launch constants, the dense MLP account is about

$$2Td_{\text{model}}I_{\text{dense}} \approx 15.0 \times 10^9$$

multiply-add terms. If the MoE layer uses $k = 2$, $R = 256$ valid routed assignments, and each expert has $I_{\text{expert}} = 2048$, the unpadded expert account is about 4.3×10^9. If block alignment pads 256 real assignments to 384 expert-kernel rows, the expert term rises to about 6.4×10^9. If 160 assignments are remote, the lower-bound FP16 send-and-return payload is $2 \cdot 160 \cdot 4096 \cdot 2 \approx 2.6$ MB before scale bytes, handles, and alignment buffers. The MoE path breaks even only if the measured route, pack, communication, combine, padding, and placement-update latency stays below the dense work it avoided.

Example 21.8.2 (Payload bytes versus scale bytes) If three routed assignments cross rank boundaries with $d_{\text{model}} = 4096$, an FP16 payload lower bound is $2 \cdot 3 \cdot 4096 \cdot 2 = 49{,}152$ bytes for send and return hidden-state payloads. An FP8 payload would reduce only the payload term to 24,576 bytes. The measurement still has to report any activation-scale or block-scale bytes separately, because FlashInfer, FP8, FP4, MXFP4, W4A8, and CUTLASS-style paths do not all carry the same scale layout.

Table 21.22.: MoE bottleneck symptoms and first mitigations. The evidence signal identifies the likely cost term before tuning.

Symptom	Likely cost term	Evidence signal	First mitigation
One expert dominates the step	C_{expert} through high N_p	Physical counts, max-to-mean load, valid replicas, and recorder window.	Enable or retune EPLB.
High remote assignment share	C_{comm} through R_{remote}	Per-rank counts, payload dtype, and scale-byte policy.	Revisit placement, A2A backend, or DeepEP mode.
Fast kernel but poor latency	C_{pack} and C_{combine}	Dispatcher output, inverse metadata, padded slots, and pre-/post-permute method.	Use a dispatcher/runner pair with fewer conversions.
Frequent rebalance stalls	EPLB metadata and weight movement	Rebalance interval, layer chunk size, utilization threshold, and movement records.	Increase the window or reduce update granularity.

21.9. Boundaries and Implementation Evidence

Inference routing maps token rows to physical expert work.

The MoE inference contract is runtime routing and expert placement. Router scores, selected expert ids, dispatch buffers, physical expert locations, and weighted combine outputs are serving observables. MoE pretraining losses, expert specialization quality, and the reasons a model learned a distribution may explain those observables, but they are not state that the runtime can repair during inference.

The runtime boundary starts after the scheduler has selected token rows and before a dense MLP would normally run. Its subject is the serving contract that turns router scores into executable expert work: top-k ids and weights, optional logical-to-physical expert remapping, dispatch format, expert execution, and weighted combine back into original row order. Cache allocation, continuous batching, and attention backend choice remain assumptions unless they change that routed-row contract.

Parallelism and collectives enter through expert placement and dispatch. The runtime question is whether a logical expert id denotes the right physical replica and whether every dispatched token returns to the row that selected it. Runtime EPLB metadata, DeepEP or FlashInfer dispatch formats, fused shared experts, and physical-replica weight materialization matter because they can change the executable meaning of a top-k id during inference. Backend microarchitecture matters when it changes dtype scales, block layout, local expert range, dispatch metadata, or the reduction expected by combine.

Table 21.23 separates MoE serving state from nearby concerns that often look similar in traces or profiles.

Table 21.24 then maps neighboring runtime areas to the objects they hand to, or receive from, the MoE layer.

Those handoffs make implementation evidence useful: the SGLang implementation trail has to connect router-score production, `TopKOutput`,

Easy confusion	Outside the MoE boundary	Inside the MoE boundary
Training load balancing versus EPLB	Auxiliary training losses and expert specialization quality.	Runtime count windows, replica placement, and metadata updates used by later inference steps.
Gateway load balancing versus expert routing	Choosing a worker, queue, or backend for a request before model execution.	Choosing logical experts for a token row inside one routed layer.
Collective algorithm versus dispatch contract	Ring, tree, transport, or kernel implementation details for communication.	Row ids, counts, physical expert ids, and combine metadata carried across the exchange.
Kernel speed versus runner eligibility	Microarchitectural tuning after the expert-major layout is fixed.	Whether dtype, scale layout, local expert range, padding, and fused shared experts match the selected runner.
Capacity policy versus padding	Model- or service-level choices to drop, cap, or reroute assignments.	Non-semantic alignment slots and masked -1 expert ids that must not change the sparse formula.

Table 21.23.: Common MoE boundary confusions. The inference contract is layer-local expert routing and reconstruction, not training objective design, gateway routing, collective algorithm design, or kernel microarchitecture by itself.

`ExpertLocationMetadata`, dispatcher metadata, and fused expert runner selection rather than treating the MoE layer as an isolated sparse-MLP formula.

Implementation Evidence.

The current SGLang source presents MoE as a serving stack rather than a single feed-forward-layer replacement. Router and top-k code define selected logical experts; fused and backend-specific runners define how grouped expert GEMMs consume routed rows; token dispatchers define the data-movement format; and EPLB plus elastic expert-parallel modules define how physical expert placement can change while the router's logical meaning is preserved.

The implementation lesson is that MoE support became a serving stack rather than a model-file feature. Code reorganization, fused Triton config gates, model-family configs, latent MoE support, shared runner helpers, and tuned expert kernels all matter because they can change the same execution record: logical top-k ids, physical expert placement, dispatcher format, runner eligibility, and combine metadata must describe one execution step.

Concept-to-code map

- **Router policy and top-k output.** `TopKConfig`, `TopKOutput`, and router fast paths identify the policy that produced the logical expert ids, weights, grouping rule, and padding convention for the row.
- **Layer-local MoE execution boundary.** `FusedMoE.forward` and EP forward paths show where hidden rows, top-k output, dispatcher output, runner input, combine input, and final reduction meet.
- **Dispatch and combine layout record.** The dispatch/combine record is the typed object that preserves row order, physical expert ids, and inverse layout.
- **Runner eligibility for grouped expert work.** `PermuteMethodPool`, grouped runners, and quantized kernels check whether the dispatcher output is

Table 21.24.: MoE handoff objects across neighboring runtime areas. MoE serving owns top-k row routing, dispatch, expert execution, and combine; adjacent chapters provide rank spaces, communication payloads, worker selection, and kernel launch layouts.

Runtime area	Boundary responsibility	Handoff object for MoE execution
MoE serving	Top-k ids and weights, logical-to-physical expert maps, dispatch format, expert execution, combine, and routed-row accounting.	Top-k output, expert-location metadata, dispatcher output, combine input, and physical expert counts.
Parallelism taxonomy	Process-group naming, rank spaces, and DP/TP/EP placement vocabulary.	Rank ids and expert-parallel groups consumed by the MoE layer.
Collectives and communication kernels	Collective algorithms, transport choices, and communication cost models.	Per-rank token counts, send and receive buffers, and payload-byte accounting.
Routing and load balancing	Scheduler-level request routing, backend selection, and service-level load policy.	Selected worker, batch, and serving trace before a token row enters the MoE layer.
Kernel design in SGLang	Kernel microarchitecture after runner input layout is fixed.	Expert-major rows, dtype and scale layout, local expert range, and block-aligned row counts.

eligible for the dtype, scale layout, local expert range, grouped-GEMM problem table, and combine format.

- **EPLB placement snapshot.** `EPLBManager`, placement metadata, and reference checks establish which placement snapshot proves that logical expert ids still denote the intended physical experts.
- **EPLB physical-slot update.** `ExpertLocationUpdater.update`, P2P movement cases, and canary checks separate slots that stayed unchanged, copied locally, reused a filled slot, or required same-node or cross-node transfer.
- **Elastic backup ownership.** `ExpertBackupManager`, `BackupDramReq`, and registered DRAM buffers identify which rank owns the backup bytes for a logical expert and whether those bytes are registered for transfer.
- **Elastic EP local weight materialization.** `ExpertBackupClient.update_weights` and projection-shard remapping show which missing logical experts must be loaded into local physical slots and whether their byte counts match the destination parameter slices.
- **Elastic EP recovery and placement broadcast.** `ElasticEPStateManager`, active-rank recovery, and metadata broadcast show which ranks are healthy enough to participate and whether recovered placement metadata has rejoined the communication groups.

The concordance is a checklist rather than a directory tour. It starts at the model wrapper that creates router logits and a `TopK` object. The resulting `TopKOutput` enters `FusedMoE.forward`, where hidden rows, ids, and weights first meet the selected dispatcher. The dispatcher record is a typed layout; runner eligibility, combine order, and EPLB or elastic-EP state all have to agree with that layout before later dispatch consumes the placement snapshot. Kernel-specific evidence then becomes supporting detail: FP8, W4A8, FP4, NVFP4, FlashInfer, TRT-LLM, and CUTLASS variants are relevant only insofar as their payloads, scale layouts, expert offsets, and reference checks preserve this same contract. Performance claims should then point back to the same counters named in Section 21.8.

21.10. Summary

The MoE chapter follows one row through a sparse layer. The router may duplicate that row into several expert assignments, dispatch may permute or communicate those assignments, and the runner may pad them for a grouped kernel. The final combine step still owes the original token row the weighted sum selected by the model. The main risk is that a locally reasonable load-balancing, communication, or kernel optimization changes which expert result belongs to which token.

The executable record starts with router policy: score transform, grouping, bias, routed scaling, padding, shared expert columns, and top-k output format. Dispatch then chooses a typed record format. Standard, DeepEP normal, DeepEP low-latency, FlashInfer, and custom records carry different inverse state even when their hidden rows have compatible shapes. Runner eligibility depends on dispatcher layout, dtype, scale metadata, local expert range, block padding, and fused shared expert slots. The MoE runtime selector (Section 21.3) binds these choices into one compatibility decision rather than a set of independent backend strings.

The step-alignment invariant (Invariant 21.6.1) and the padding-capacity invariant (Invariant 21.6.3) come before performance numbers. EPLB placement must be read as a snapshot, so a token step cannot mix a new physical-to-logical map with old expert weights or old combine metadata. Payload bytes, scale bytes, padding slots, remote assignments, and placement-update traffic remain separate accounts when MoE variants are compared. Elastic expert backup extends that snapshot rule: a new physical expert slot must be materialized by unchanged ownership, local copy, P2P transfer, DRAM backup transfer, or disk reload before later dispatch treats it as the logical expert named by the map. MoE communication-computation overlap (Definition 21.7.4) changes when that evidence is produced, not which dispatch, combine, placement, and row-restoration evidence the step must carry.

21.11. Exercises

1. Define routed token and expert load for the running MoE example. Then name the tensors that provide the selected logical expert $e_{t,j}$ and routing weight $\alpha_{t,j}$ in the sparse mixture formula.
2. For four tokens with top-$k = 2$ and selected experts $\{0,1\}, \{0,2\}, \{0,3\}, \{1,2\}$, compute R, the logical expert loads, and the padded slot count when $q = 4$.
3. If the same trace has hidden width $d_{\text{model}} = 4096$, FP16 payloads, and three routed assignments cross rank boundaries, estimate the send-plus-return hidden-state bytes before scale tensors and alignment buffers.
4. Given map $P_0, P_1, P_2 \mapsto E_0$, $P_3 \mapsto E_1$, $P_4 \mapsto E_2$, and $P_5 \mapsto E_3$, list the valid replicas and explain why any valid E_0 replica preserves the sparse formula.
5. A local EPLB update changes P_2, P_3, P_4, P_5 from (E_2, E_3, E_4, E_5) to (E_0, E_0, E_3, E_5). With E_0 on same-node peer P_0 and cross-node peer P_6, classify how P_2 and P_3 obtain E_0 when all peers are healthy, when the same-node peer is down, and when no E_0 peer is available.
6. In a launch pattern with all-to-all backend `deepep`, runner backend `deep_gemm`, tensor parallel size 8, and expert parallel size 8, identify the backend roles, parallel sizes, and compatibility assumption. Then state what inverse state the combine step must know for standard and DeepEP dispatch formats.

7. Explain why `prepare_moe_input` and `moe_align_block_size` are correctness boundaries before a grouped GEMM, and why padding or block-alignment slots must not be counted as routed model work.
8. For `topk_ids` rows $(0,1),(0,2),(1,2)$, $k = 2$, $E = 3$, and block size $q = 4$, compute the real expert counts, `expert_offsets`, padded counts, `experts_ids`, sentinel padding positions, and the combine expression for token t_0 if its routing weights are $(0.6, 0.4)$.
9. Compare Two-Batch Overlap and Single-Batch Overlap. What dependency or metadata evidence must remain intact for each? For each overlap mode, name the dispatch state, combine state, and stream or event dependency that proves the overlap did not change which routed token rows were returned.
10. Review a proposed MoE backend change. List the router, dispatcher, runner, placement, padding, combine, and measurement evidence that must remain aligned; include one rejecting fallback condition and one divergence counter.

Part VI.

Parallel and Distributed Serving