LoRA as Runtime Low-Rank Updates

Deep Dive into SGLang — Chapter 20

Yin Yang

Foundation Books Project

The bug that names the chapter

Put two request rows in one batch. Rows A and B share *one* base projection, but only A selected an adapter.

- The low-rank delta A carries must not leak onto B.
- Nor may A read the slot, rank, or scale that belongs to some other row.

Runtime LoRA serving adds a request-selected low-rank update *beside* a frozen base layer — without turning each adapter into a separate model replica.

The durable rule: row-local adapter state on top of shared base-model execution.

Roadmap

The adapter-augmented operator

Selection, residency, and step alignment

Admission and the memory pool

From request ids to executed deltas

Model boundaries and residency policy

Takeaways

The adapter-augmented operator

One equation carries the whole chapter

For a hidden row x_i, a base-only request writes Wx_i. A request using adapter g writes

$$Wx_i + \frac{\alpha_g}{r_g} B_g(A_g x_i)\,.$$

- $A_g \in \mathbb{R}^{r_g \times d}$ maps the row *down* to adapter rank r_g;
- $B_g \in \mathbb{R}^{d_{\text{out}} \times r_g}$ maps it *back up*;
- α_g / r_g is the fixed inference-time scale stored with the adapter.

Wx_i is **shared**. The low-rank term is **request-local**: it must use the A_g, B_g, rank, scale, and pool slot selected for *that* row.

Adapter rank is a matrix dimension — not a distributed worker rank.

What the runtime actually observes

This is *inference* state, not training. SGLang applies already-trained tensors on the forward path; it computes no gradients.

The serving contract is narrow. For each row it must know:

which adapter uid the request named,

whether that adapter is *resident* in a backend-visible pool,

where — which slot holds its A, B, rank, and scale,

how — which backend applies the low-rank product.

The boundary is the moment an already-loaded base model receives per-request adapter state. Everything in this chapter defends that boundary.

Selection, residency, and step alignment

Two names, then one invariant

Adapter selection the per-request choice $a_t(i, \ell) \in \mathcal{A} \cup \{\varnothing\}$ of a low-rank delta at row i, module ℓ. $\varnothing$ means base-only.

Adapter residency $\text{resident}_t(a, \ell) = 1$: the tensors for a — A, B, rank, scale, target-module metadata — are loaded in a backend-visible pool *before* the row reaches the layer.

Invariant (LoRA step-alignment)
The adapter id, rank, scale, and tensor placement used for a request must stay aligned with that request across prefill, decode, and sampling.

Selection says what a row wants; residency says whether it is there to use.

The canonical failure is shape-valid

Let adapters X and Z both have rank 8 and target the same projection.

	row B names slot 2	slot 2 owns ...		
before eviction	selected X	X	$\Rightarrow$	base $+ X$ (correct)
after reuse	stale index	Z	$\Rightarrow$	base $+ Z$ (wrong user)

The tensor dimensions still work; the forward pass still finishes. The error is *semantic* — the response is specialized with another adapter's learned delta.

This is why ownership and load-completion sit beside tensor shape in the formal rule.
Measurement (hit rate, load latency, per-token overhead) is valid only after the alignment holds.

Admission and the memory pool

Admitting an adapter: validate, normalize, seat

The manager turns a request-visible uid into validated state $M_t = (U_t, R_t, S_t, B_t)$: known uids, checked references, resident slot map, backend metadata.

- **Validate:** reject added-vocabulary adapters, DoRA, duplicate names, over-rank / missing-target configs, and all-slots-pinned.
- **Normalize:** rewrite checkpoint Q/K/V $\rightarrow$ fused `qkv_proj`, gate/up $\rightarrow$ `gate_up_proj`; a target absent from $\nu(\mathcal{T}_{\mathrm{ckpt}})$ is *not* resident.
- **Seat:** the pool holds `max_loras_per_batch` slots (default 8, incl. the base pseudo-id $\varnothing$); evict, zero, load.

A pinned adapter is non-evictable but still costs one finite slot — so pinning may never consume them all.

From concept to code: the admission gate

Validation happens *before* the adapter can enter the residency map.

from `python/sglang/srt/lora/lora_manager.py`

```
def validate_new_adapter(self, lora_config, lora_ref):
    if lora_config.lora_added_tokens_size > 0:
        raise ValueError("...doesn't support adapters that add tokens...")
    if lora_config.use_dora:
        raise ValueError("...doesn't support DoRA adapters")
    # ... reject duplicate name, then check pool compatibility:
    incompatible = memory_pool and not memory_pool.can_support(lora_config)
    if incompatible:
        raise ValueError("...rank ... within `--max-lora-rank` ... target modules ...")
    if lora_ref.pinned and self.num_pinned_loras >= self.max_loras_per_batch - 1:
        raise ValueError(...)  # pinning may not starve unpinned traffic
```

DoRA looks like LoRA at the file boundary but changes the operator — so it is refused, not silently applied as ordinary LoRA.

From request ids to executed deltas

Two arrays, two different objects

Forward-batch routing turns request-ordered ids into backend arrays. For ids $[\varnothing, X, Y, X]$ with resident slots $\varnothing \mapsto 0$, $Y \mapsto 1$, $X \mapsto 2$:

$$\underbrace{w = [0, 2, 1, 2]}_{\text{row-indexed: which slot each row reads}} \qquad \underbrace{(r_k, \alpha_k) \text{ per slot}}_{\text{slot-indexed: how to read it}}$$

- Rows 1 and 3 *share* slot 2's rank/scale — both chose X.
- Row 2 reads slot 1's entries — it chose Y.
- The base row maps to slot 0, rank 0: it appears in w but adds no delta.

A row-indexed weight array is not a slot-indexed metadata array — the chapter keeps them apart on purpose.

Execution: segment, apply, scatter back

Request order $[X, X, Y, \varnothing, Y]$ is convenient for the scheduler; a *segmented* kernel wants adjacent tokens sharing one slot.

- **SGMV** = segmented matrix-vector LoRA: stable-sort tokens by slot, build `seg_lens`/`seg_indptr`, record a permutation.
- In row-major form the update $B_i(A_i x)$ is applied as $(HA_i^\top)B_i^\top$ with scale $s_i = \alpha_i / r_i$.
- Chunked SGMV = same contract, bounded launch sizes for long streams.

Reordering changes execution order, not ownership. A row that selected slot 1 must still read slot 1 after routing — the inverse scatter restores request order.

From concept to code: sort tokens, keep ownership

from python/sglang/srt/lora/backend/triton_backend.py

```
def compute_sgemm_routing(self, use_cuda_graph):
    """Sort tokens by adapter and build merged segments for sgemm LoRA."""
    wi = bi.weight_indices[:bs]
    perm = torch.argsort(wi, stable=True).to(torch.int32)   # permutation
    sorted_wi = wi[perm]
    adapter_ids = torch.arange(mlpb, ...)
    seg_starts = torch.searchsorted(sorted_wi, adapter_ids)
    seg_ends   = torch.searchsorted(sorted_wi, adapter_ids, right=True)
    seg_lens   = seg_ends - seg_starts
    seg_indptr[1:] = torch.cumsum(seg_lens, dim=0)          # segment boundaries
```

perm is the whole safety story: tokens move into slot-contiguous segments, and its inverse scatters the deltas back to request rows. The slot ids in weight_indices still come from each row's own adapter choice.

Model boundaries and residency policy

Two boundaries add a coordinate, not a new rule

Most LoRA layers adapt a hidden projection. Two cases add one more axis:

Vocabulary boundary at `embed_tokens` / `lm_head`. Token ids pick LoRA-A rows; a chunked, pruned logprob pass must rebuild *pass-local* segment metadata so each scored token keeps its slot and the LoRA-B vocabulary shard.

MoE-LoRA group routed work by the pair key $(\mathrm{slot}(a), e)$, not by expert e alone — two requests on the same expert can need different A, B.

Same step-alignment invariant, extra coordinates: token id / vocab shard for heads, expert id (and `moe_tp_rank`) for MoE.

Residency policy: two costs, one measurement rule

A cost account $J(c) = (M_c, T_c, F_c, L_c)$: resident memory, transfer, batch fragmentation, low-rank compute.

- **Hot adapters** stay resident $\Rightarrow$ mostly hits, steady slots.
- **Long-tail adapters** $\Rightarrow$ eviction, CPU$\rightarrow$device copies, fragmented batches. Default eviction is LRU over the *safe* set (never a current-batch id, never a pinned one; prefer non-base victims).

This chapter reports no completed residency run — only the contract a run must honor: fix base / adapter set / trace / hardware, then join slot events, loader events, and serving metrics by request row and uid.

Takeaways

What to carry forward

1. LoRA serving = **one shared base** Wx plus **row-local** $\frac{\alpha}{r} B(Ax)$; never a model replica per adapter.
2. **Step alignment** forbids a whole class of *shape-valid* wrong-adapter bugs; measurement is meaningless until it holds.
3. Admission **validates, normalizes to fused targets, and seats** the adapter in a finite pool before any layer runs.
4. Routing builds a **row-indexed slot array** and **slot-indexed rank/scale**; segmentation reorders tokens but the *permutation preserves ownership*.
5. Vocabulary heads and MoE add coordinates (**pass-local metadata**, the **pair key**) — the same invariant, more indices.

A throughput patch that lets one request consume another's slot has changed the model and the runtime state together.