Continuous Batching

Deep Dive into SGLang — Chapter 5

Yin Yang

Foundation Books Project

The one rule of this chapter

The scheduler may change *which* requests share a step.
It may not change what each request *means*.

- One request finishes just as two resident rows need their next token and a long prompt reaches admission.
- Wait for a fixed batch to drain, and useful work sits idle.
- Continuous batching lets membership change between forward steps — while every surviving request keeps its own token trace.

Goal: follow one scheduler round — waiting work, resident decode, admission, ordering, mixed work, retraction — and see where it lives in the SGLang source.

Roadmap

The mutable batch

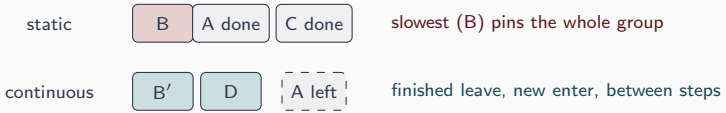
Admission and ordering

Building and maintaining the batch

Recovery, fairness, and correctness

The mutable batch

Static batching has the wrong shape



- Arrivals are staggered, prompt lengths vary by orders of magnitude, output lengths are unknown, and KV footprint grows during decode.
- Waiting for a fixed group blocks short work behind long prompts, or keeps finished requests pinning a batch.

Continuous batching makes the serving step dynamic while preserving each request's token stream (Orca-style iteration-level scheduling).

The running example: one decode-resident row, three waiting

`google/gemma-4-E4B-it`; page size $P = 4$ slots; budget 20 slots.

Request	Prompt	Cached	Out	Max new	Note
r_{fresh}	8	0	0	5	ordinary new prompt
r_{reuse1}	6	4	0	3	shares a prefix with reuse2
r_{reuse2}	10	4	0	4	shares a prefix with reuse1
r_{pan}	3	3	2	6	already decode-resident, high priority

Effective prefill $e_i = |z_i| - p_i$: reuse1 needs $6 - 4 = 2$ tokens, reuse2 needs $10 - 4 = 6$, fresh needs 8. Page-rounded: $\text{alloc}_4(2)$=4, $\text{alloc}_4(6)$=8, $\text{alloc}_4(8)$=8. *Raw prompt length hides the real work.*

Scheduler state is a five-part tuple

Definition (Continuous-batching scheduler state)

$$S_t = (W_t,\ B_t,\ \Lambda_t,\ U_t,\ M_t).$$

- W_t waiting queue — requests not yet admitted.
- B_t resident running batch — decode rows carried across steps.
- Λ_t last prefill batch — results that may need merging.
- U_t one unfinished prefill continuation (a half-admitted prompt).
- M_t memory and prefix-cache view.

One scheduler round: merge Λ_t, refresh budgets, pick the next step (`EXTEND` / `DECODE` / `MIXED`), run it, process the result into S_{t+1}.

From concept to code: try prefill first, else decode

The whole step-selection decision has a concrete home in the scheduler loop.

from `python/sglang/srt/managers/scheduler.py`

```
new_batch = self.get_new_batch_prefill()
...
if new_batch is not None:
    # Run prefill first if possible
    ret = new_batch
else:
    # Run decode (skip for prefill-only batches)
    if (not self.running_batch.is_empty()
            and not self.running_batch.is_prefill_only):
        self.running_batch = self.update_running_batch(self.running_batch)
        ret = self.running_batch if not ... else None
```

One step = assemble a feasible prefill batch if you can; otherwise advance the resident decode batch. `get_next_batch_to_run` wraps this with the merge-before-next-step discipline.

Admission and ordering

Admission is a budget proof, not a token count

Definition (Page-aligned admission charge)

$$\text{charge}_P(r) = \text{alloc}_P(e_r) + \rho_r + P, \qquad \text{alloc}_P(n) = P\lceil n/P \rceil.$$

- $\text{alloc}_P(e_r)$: page-rounded *uncached* suffix work.
- ρ_r: future-decode reserve — room for output not yet generated.
- P: one page of allocator headroom.

Feasible when $\text{charge}_P(r) \le A_t + E_t - R_t^{\text{cap}}$ (available + evictable − already reserved).

reuse2: $\text{alloc}_4(6)+4+4 = 8+4+4 = 16$ slots — not six. Raw token counts cannot protect a paged allocator.

Queue order is a policy once reuse enters the cost model



- FCFS, longest-output, random, routing-key affinity — cache-agnostic.
- Longest-prefix-match, DFS-weight — cache-aware, scored over prefix structure.
- A throwaway *simulated* waiting tree $\mathcal{T}_W$ stops a request from jumping the queue on reuse that does not exist *yet*.

Order changes; feasibility and decode protection do not. Reuse cannot be spent twice.

Building and maintaining the batch

Two indexing axes in one executable batch

The model consumes tensors, not a list of requests. Construction splits along two axes:

Token-indexed (flat, suffix order)

- suffix token ids
- KV-write locations (`out_cache_loc`)

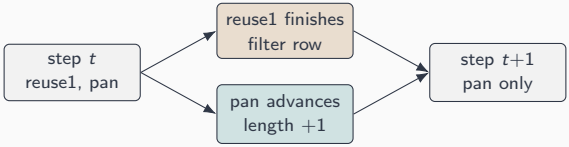
Request-indexed (scheduler order)

- `seq_lens`, `prefix_lens`, `extend_lens`
- request-pool rows, sampling state

Scheduler-to-model identity agreement
Slot j of *every* per-request array must name r_j. Compatible tensor shapes are necessary but not sufficient.

Lose the request-indexed axis and the attention backend cannot separate fresh's 8 positions from reuse1's 2.

Decode maintenance: one row, recurring



- Each live row: one next-token input, one logical position advance, $|z_i(t{+}1)| = |z_i(t)| + 1$.
- A new KV page is claimed *only* at a page boundary — mid-page decode costs zero new slots.
- Finished rows are *filtered* before they consume another decode slot.

Membership changes; the per-row decode contract does not.

Mixed work and the chunked-prefill boundary

- **Merge:** after reuse1's prefill, $B_{t+1} = (\mathrm{pan}, \mathrm{reuse1})$, $\Lambda_{t+1} = \varnothing$ — new prompt becomes decode-resident.
- `MIXED` **step:** one forward pass carries prefill tokens *and* one-token decode inputs together.

The boundary worth naming
Continuous batching owns *whether* the resident set may change between steps. Chunked prefill (next chapter) owns *how much* long-prompt suffix joins a mixed step.

Not every step is mixed. Pure prefill, pure decode, and mixed steps are all legal; what makes the system *continuous* is the updatable resident set.

Recovery, fairness, and correctness

Retraction: the release valve when the reserve was too small

Definition (Continuous-batching retraction)
Filter rows out of B_t, hand their reusable KV back, rebuild surviving metadata, and requeue the affected requests: $B'_t = B_t \setminus \mathcal{D}_t^{\mathrm{ret}}$.



Retracted is not aborted. Already-emitted output survives; only decode residency is lost. Abort is reserved for when even the last survivor cannot fit.

Six invariants keep a mutable batch honest

1. **One scheduler location** — each nonterminal request lives in exactly one place (waiting / decode / prefill / chunk / staging).
2. **Logical sequence** — $z_i = x_i \| y_i$ survives next-round reconstruction.
3. **Cache validity** — a prefix hit must match *this* request's key (token ids, LoRA, multimodal, position).
4. **Batch metadata order** — every per-request array shares one request order after filter / merge / retract.
5. **Decode memory reservation** — reserve the next KV write *before* launching the step.
6. **Terminal filtering** — a finished request must leave future decode steps.

The dangerous bugs (metadata order, missing reservation) leave shapes numerically plausible while writing one request's state into another's slot.

What the measurements do — and do not — prove

Policy (Thor, Qwen2.5-0.5B)	TTFT	out tok/s	note
`fcfs`	392 ms	3,763	interleaves groups
`lpm`	6,356 ms	577	clusters siblings behind one provider

- `lpm` is not a low-latency policy — it is a *reuse-seeking* ordering. On a small burst, clustering pushes group-tail first tokens far out.
- The reversed sign vs. Chapter 2 is a workload-shape result, not a contradiction.
- The trace *directly observes* batch membership changing; it does *not* carry cache-hit, page-slack, or retraction counters.

A batching row must name load, policy, cache state, and token budget together.

What to carry forward

1. Continuous batching changes **batch membership**, never **request meaning**.
2. Admission is **page-aligned and predictive**: charge uncached work + output reserve + headroom.
3. Queue order can be **cache-aware**, but reuse is real only after its provider runs.
4. One **request-order axis** governs all batch metadata — the easiest invariant to break silently.
5. **Retraction** recovers memory without aborting; the **future-state ledger** stays honest.

Next: Chapter 6 treats chunked prefill as mixed work — long prompt suffixes overlap decode while the scheduler protects first-token latency.