Continuous Batching | 5.

Chapter Abstract. Continuous batching improves online serving by letting requests enter and leave between model steps instead of waiting for a fixed group to finish. That mutable batch has one rule: membership can change, but request and cache invariants must not change. A waiting prefill may want admission at the same moment resident decode rows need low latency; the scheduler must decide without detaching either request from its cache or output state. This chapter follows one scheduler round: waiting work, resident decode rows, admission charges, cache-aware ordering, mixed work, and retraction. The source objects matter because each round rebuilds the executable frontier while the same logical request traces, KV reservations, sampling state, and output obligations remain attached.

5.1. The Continuous Batching Problem

One request finishes just as two resident requests need their next decode token and a long prompt reaches admission. If the runtime waits for a fixed batch to drain, useful work is already sitting idle. This chapter is about changing batch membership without changing request meaning.

Figure 5.1 shows the chapter's central constraint: request rows can enter and leave between steps while each surviving request keeps its own token trace.

Static batching is attractive because it has a simple shape: collect a fixed set of requests, run them together, and release the whole batch when the slowest request finishes. Online generation has the opposite shape. Requests arrive at different times, prompt lengths vary by orders of magnitude, output lengths are not known in advance, and the fast-memory footprint grows as decoding continues. A scheduler that waits for a whole fixed group to complete either blocks short requests behind long prompts or keeps finished requests inside a batch that no longer needs them.

Continuous batching makes the serving step dynamic while preserving each request's token stream.

Continuous batching is the runtime answer first made explicit for transformer serving by iteration-level scheduling systems such as Orca [48]. Chapter 2 introduced dynamic batching as an online serving necessity (Definition 2.7.1), and Chapter 3 gave the cost model for mixed serving steps (Definition 3.2.5). This chapter moves from the cost account to the scheduling mechanism. The runtime must choose the next feasible forward step (Definition 1.4.1) while requests are in different phases. Some requests need prefill work (Definition 2.3.1), some need one-token decode (Definition 2.3.2), and some are finished or must be retracted.

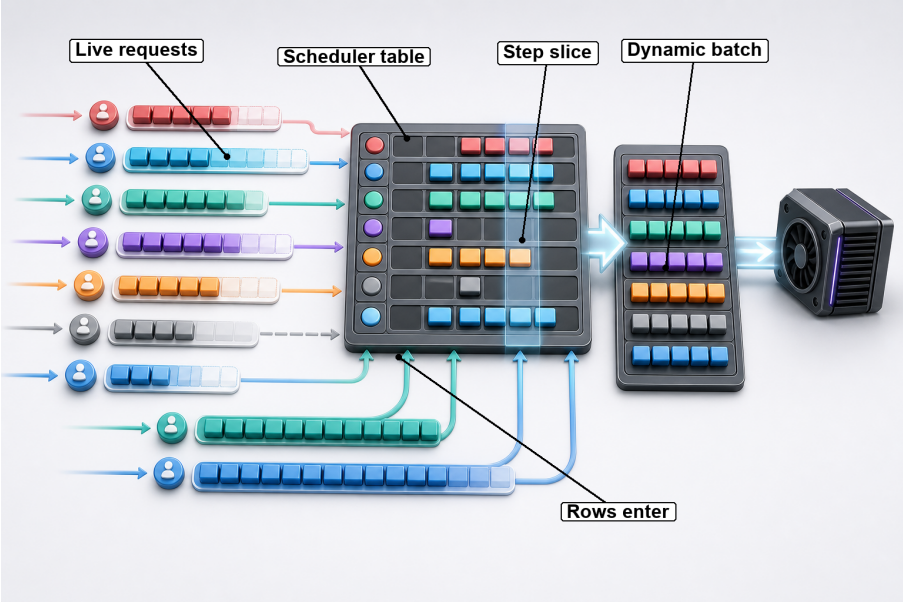

Figure 5.1.: Mutable requests forming a continuous batch. Live requests enter the scheduler table, a model-step slice selects row positions, and the resulting dynamic batch advances one step.

Definition 5.1.1 (Continuous batching) *Continuous batching is a scheduling discipline in which the active batch may change between successive model forward steps. New requests may enter after their prompt work is admitted, finished requests may leave, and memory-pressure victims may be requeued. The stable objects are the per-request token sequence and the metadata that preserves that sequence through execution. If B_t is the active executable batch at step t, continuous batching allows B_{t+1} to differ from B_t while preserving each surviving request's trace.*

Example 5.1.1 (Batch membership changes without changing request meaning) At step t, suppose the executable batch contains resident decode requests r_{done} and r_{carry}. Between steps, r_{done} finishes, r_{new}'s prompt is admitted, and r_{carry} still needs one more decode token. Continuous batching lets the next step use batch membership $[r_{\text{carry}}, r_{\text{new}}]$ rather than waiting for a fixed batch containing all three requests to complete together. r_{carry}'s next token must be the same token it would have received if r_{new} had not arrived; r_{new}'s admission changes the execution batch, not r_{carry}'s request trace.

The scheduler may change the batch between steps, but correctness is still judged request by request.

This definition separates request semantics from execution mechanics. A request's answer should not change merely because another request arrived or finished between two decode steps. The batch is an execution contract for one step. It tells the model which token positions are new and which context is already resident. It also records where new key/value state should be written and which sampling state belongs to which request.

We use a small running example throughout the discussion. The serving instance runs `google/gemma-4-E4B-it`; the numbers below are tokenized request accounts chosen for arithmetic before any performance claim. The pancake request

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

Table 5.1.: Running continuous-batching request set. The rows fix prompt length, prefix reuse, output state, and local scheduling pressure.

r_{pan} is already decode-resident, while three companion requests expose fresh-prompt and prefix-reuse admission choices. The page size is $P = 4$ KV-token slots, and the simplified free plus evictable KV budget is 20 slots before the scheduler chooses the next prefill batch.

A cached prefix means that the runtime can reuse resident KV state for the first p_i positions of a request. Visible text sharing is insufficient by itself: the cache key must agree on the model-facing facts that make those KV values valid, and later chapters give the full prefix-cache data structure. For this chapter, the capsule is enough: a valid cached prefix of length p_i reduces the fresh prefill work from the prompt length to the effective suffix e_i.

Definition 5.1.2 (Decode residency) *A request is decode-resident at step t if it remains in the running batch after its current prefill work is complete and it is eligible for next-token generation. Decode residency requires a valid request slot, sequence length, next input token, sampling state, and enough KV capacity for the next write.*

Example 5.1.2 (Decode residency as a field checklist) In the running example, r_{pan} has already completed its prompt and has two generated output tokens. It is decode-resident only if the scheduler can still name r_{pan}'s request slot, its current sequence length, the token row that should be fed next, r_{pan}'s sampling state, and a KV slot for the next write. If r_{pan} has the right visible text but no next KV write capacity, it is not decode-resident for this step. Requests r_{fresh}, r_{reuse1}, and r_{reuse2} are not decode-resident yet because their prompt work has not completed. Decode residency is therefore a state boundary: it requires prompt completion plus executable next-decode-row state.

Requests r_{fresh}, r_{reuse1}, and r_{reuse2} are waiting to be admitted for prompt computation. r_{pan} has already completed its prompt and has generated two output tokens, so it is decode-resident under Definition 5.1.2. If the scheduler admits r_{reuse1}, it does not need to recompute the first four prompt tokens; it needs to prefill only $6 - 4 = 2$ prompt tokens. If it admits r_{reuse2}, it needs $10 - 4 = 6$ prompt tokens. The page-aligned allocations are $\lceil 8/4 \rceil 4 = 8$ for r_{fresh}, $\lceil 2/4 \rceil 4 = 4$ for r_{reuse1}, and $\lceil 6/4 \rceil 4 = 8$ for r_{reuse2} before any future-token reserve. Here $\lceil n/P \rceil P$ means page-rounded token slots, with $P = 4$ in the running example. We write this page-rounded charge as $\text{alloc}_P(n) = P\lceil n/P \rceil$ when the later account needs a function name.

The example exposes the main tension. Admitting r_{fresh} and r_{reuse1} uses $8 + 4 = 12$ page-aligned prompt slots. Admitting r_{reuse1} and r_{reuse2} uses the same $4 + 8 = 12$

slots but may create better prefix locality. Admitting all three uses 20 prompt slots exactly and leaves no simplified reserve for r_{pan}'s next decode write or for future generation. A production scheduler therefore cannot ask only whether the prompt tokens fit. It must reserve for the resident decode set, future output tokens, page overhead, request slots, and policy constraints [19, 41].

The scheduling account reuses a compact set of symbols for request state, step selection, token budgets, queue ordering, prefill construction, decode maintenance, mixed work, and retraction. Table 5.2 records those symbols at the resident-decode and admission boundary.

Table 5.2.: Local notation for continuous batching. The symbols index the scheduler tuple, admission charge, page geometry, resident decode rows, and capacity decisions used throughout the chapter.

Symbol	Meaning	Role in the serving account
S_t	Scheduler state tuple $(W_t, B_t, \Lambda_t, U_t, M_t)$	Waiting work, resident batch, last prefill result, unfinished continuation, and memory/cache view.
W_t	Waiting queue	Requests available for admission and policy ordering.
B_t	Resident running batch	Decode-resident rows carried across scheduler steps.
Λ_t	Last prefill batch	Completed prefill work that may need to merge into residency.
U_t	Unfinished prefill continuation	Partially materialized prompt suffix that is neither fresh waiting work nor decode-resident work.
e_r	Effective prefill length	Fresh suffix work after valid prefix reuse.
ρ_r	Future decode reserve	Capacity protected for future generation by request r.
$A_t, E_t, R_t^{\text{cap}}$	Capacity accounts	Immediately available, evictable, and already-reserved scheduler capacity.
P	KV page size	The allocation granularity that makes token counts and memory demand differ.
$\text{alloc}_P(n) = P\lceil n/P \rceil$	Page-aligned allocation	Function that converts token demand into KV page slots.
$\mathcal{L}_{\text{res}}$	Resident decode row list	Algorithm-facing list of the running decode rows.

The SGLang implementation evidence used in this discussion follows the same four conceptual surfaces:

Concept-to-code map

- **Step selection.** The chapter's scheduler step maps to `Scheduler.get_next_batch_to_run` and the scheduler queues in `python/sglang/srt/managers/scheduler.py`.
- **Mutable request and batch rows.** Request-local state and selected batch rows map to `Req`, `ScheduleBatch`, and their fields in `python/sglang/srt/managers/schedule_batch.py`.
- **Admission budgets.** Prefix reuse, page-aligned token budgets, and future-output reservations map to `PrefillAdder` and `SchedulePolicy` in `python/sglang/srt/managers/schedule_policy.py`.
- **Model-facing step type.** The executable prefill, decode, and mixed step labels map to `ForwardMode` and `ForwardBatch` metadata in `python/sglang/srt/model_executor/forward_batch_info.py`.

Implementation claims cite the SGLang repository entry.

5.2. Requests as Mutable Scheduling Objects

The first trap in continuous batching is to treat a request as an immutable input string. During serving, the request becomes a live work record. It keeps the original prompt and generated suffix. It also carries cache hits, allocated memory, sampling state, optional grammar state for structured-output requests, and a terminal condition. Here, grammar state means a request-local constraint matcher that will later build masks for invalid next tokens; ordinary free-form requests do not carry one. The scheduler's job is to update this work item without changing the logical sequence it represents.

A request is a live state record.

Definition 5.2.1 (Continuous-batching request state) *At scheduler step t, a request is modeled as*

$$r_i(t) = \big(id_i, x_i, y_i(t), p_i(t), e_i(t), a_i(t), \ell_i(t), g_i(t), f_i(t)\big).$$

Here id_i is the request identity, x_i is the original prompt token sequence, $y_i(t)$ is the generated output token sequence, $p_i(t)$ is the available cached-prefix length, $e_i(t)$ is the number of tokens requiring prefill computation in the current round. The term $a_i(t)$ is allocated KV capacity, $\ell_i(t)$ is committed KV length, and $g_i(t)$ is absent unless constrained decoding is active for the request. The term $f_i(t)$ is the finish state.

The tuple is intentionally smaller than the SGLang request object. Some fields anchor the logical request, some record evidence discovered by cache matching, and others protect memory ownership or terminal state. The scheduler can change the latter groups between steps, but it must preserve the logical sequence that the client asked the model to extend.

Concept-to-code map

In SGLang, the continuous-batching request state (Definition 5.2.1) is represented by fields on the request object in `python/sglang/srt/managers/schedule_batch.py`. Table 5.3 groups the SGLang fields by the scheduler contract they support. The later refresh point is `Req.init_next_round_input`; admission and batch construction then consume the refreshed fields through `PrefillAdder` and `ScheduleBatch`.

Table 5.3.: SGLang field groups for the mutable request object. The table is not an exhaustive schema; it records the fields that matter for the scheduling contracts used in this chapter.

Field group	Representative SGLang fields	Scheduler role
Stable input contract	`rid`, `origin_input_ids`, `sampling_params`, `return_logprob`, `priority`, `lora_id`, `routing_key`	Fix request identity, prompt semantics, sampling or scoring requirements, and policy inputs after validation.
Logical sequence	`output_ids`, `fill_ids`	Append generated tokens and rebuild `fill_ids` as `origin_input_ids` + `output_ids` for the next round.
Prefix evidence	`prefix_indices`, `last_node`, `last_host_node`, `host_hit_length`, `cache_protected_len`	Record what prefix-cache or host-cache matching proved before admission scoring.
Current-round work	`extend_input_len`, `extend_logprob_start_len`, `is_chunked`	Name the prompt suffix and logprob rows admitted in this scheduler round after budget or chunking truncation.
Memory ownership	`req_pool_idx`, `kv_committed_len`, `kv_allocated_len`, `kv_committed_freed`, `kv_overallocated_freed`	Track which request row and KV capacity are owned, committed, or released as batches are built, filtered, and retired.
Terminal and side state	`finished_reason`, `finished_len`, `finished_output`, `to_finish`, optional `grammar`, `multimodal_inputs`, `positional_embed_overrides`	Carry stop/abort outcomes and optional model-facing constraints that can change which tokens, masks, or embeddings the batch may use.

Streaming detokenization fields, including `decoded_text`, are still mutable request state, but they support text emission rather than the scheduling decisions analyzed here.

The logical sequence is simple:

$$z_i(t) = x_i \| y_i(t).$$

In source terms, `fill_ids` is SGLang's name for this scheduler-visible logical sequence in the normal autoregressive path: the original prompt followed by generated output tokens. Chunked prefill may limit the suffix admitted in one round, but prefix lengths, extend lengths, and logprob boundaries are still derived from this request-local sequence.

The runtime complication is that not every token in $z_i(t)$ must be computed again. A prefix cache may prove that the first $p_i(t)$ token positions already have valid

resident state, so the effective prefill length used by the scheduler for that request is

$$e_i(t) = |z_i(t)| - p_i(t),$$

in the normal case. This subtraction is not a token-ID-only rule. Before the scheduler commits $p_i(t)$ and $e_i(t)$, it must prove that the reused prefix is valid for the outputs the request asks SGLang to return. It must also match any side metadata the model consumes and the amount of suffix work admitted in this round. In the running example, r_{reuse1} has $z_{\text{reuse1}} = x_{\text{reuse1}}$, $p_{\text{reuse1}} = 4$, and $e_{\text{reuse1}} = 2$. r_{reuse2} has the same cached-prefix length but a longer prompt, so $e_{\text{reuse2}} = 6$.

Replacement embeddings are supplied hidden-state vectors that override the normal embedding-table lookup at reserved token positions. They make prefix identity depend on the token id, the override vector, and the override position: two requests can share the same token ids but describe different model inputs when their replacement embeddings differ.

Table 5.4 summarizes the concrete rows used by the following discussion.

Constraint	Effect on $p_i(t)$ or $e_i(t)$
Logprobs	A request that asks for prompt-token scores may need logits for positions that a pure cache hit would skip, so prefix reuse is capped at the requested scoring boundary.
Multimodal metadata	Token IDs alone do not prove cache compatibility when replacement embeddings, media feature spans, or position overrides participate in the prefix.
Streaming session	Session-held KV can serve as context for new tokens, but it does not by itself recreate logits for arbitrary old session positions.
Chunking	Admission may send only a prefix of the remaining suffix in this round; the rest stays attached to the same request as later prefill work.

Table 5.4.: Constraint checkpoints for effective prefill length.

Example 5.2.1 (Prefix hit changes the scheduling object) Suppose two waiting requests both have raw prompt length 10. Request R has no cached prefix, so $e_R = 10$. Request S has a valid cached prefix of length 7, so $e_S = 3$. These requests have the same client-visible prompt length but different scheduler-visible work. A policy that sorts only by raw prompt length misses the fact that S can enter a smaller prefill batch. Figure 5.2 shows how the main scheduling fields are updated as scheduler-visible state rather than reread as a static prompt.

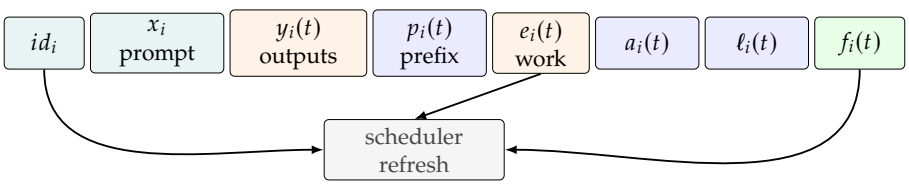

Figure 5.2.: Continuous-batching request state is a mutable record. The original prompt and identity are stable anchors, while prefix evidence, remaining work, KV accounting, generated output, and finish state are refreshed between scheduler steps.

Concept-to-code map

The method `Req.init_next_round_input` is the implementation point where this conceptual state is refreshed for a scheduling round. It forms the logical sequence, queries the prefix cache when available, stores prefix indices and nodes, and sets `extend_input_len`. The order matters: the scheduler does not first choose a batch and then discover work. It refreshes request state so that batch formation can reason about the remaining work.

Cache reuse stops before the logprob boundary.

Prefix matching is capped before this cache query. The cap is on the prefix-cache candidate, not on the request itself. If the logical sequence `fill_ids` has length n, SGLang normally asks the cache to match at most the first $n-1$ tokens. That leaves at least one token in the current extend batch, giving the logits and logprob path a newly computed position to score.

The reason is that cached KV state is attention state, not a saved probability distribution. A cache hit can tell the next row what key/value context to read; it cannot recreate the full vocabulary distribution that an earlier forward row emitted. If the caller asks for input logprobs, every prompt position whose score should be returned needs a fresh logits row in the current accounting window. The cap $\min(n-1, k)$ below is just that rule written against the requested scoring boundary.

Definition 5.2.2 (Input-logprob boundary) *For a request with `return_logprob` enabled, `logprob_start_len` set to k is the absolute prompt boundary from which input-token logprob results are requested. The first k prompt tokens may still provide context, but they are not part of the returned input-logprob span. The sentinel value -1 disables prompt input-logprob reporting, while output-token logprobs may still be returned. Within one scheduler round, `extend_logprob_start_len` is the same boundary rewritten as an offset inside the current extend suffix after prefix reuse and any current-round truncation.*

This boundary is a row-selection rule, not the target-token vector itself. After the selected rows are computed, the logprob path gathers the probability of the next original-input token predicted by each row. Algorithm 17 later materializes that shifted target-token list as `extend_input_logprob_token_ids`, in the same flat row order used by the prefill suffix.

If the request asks for input logprobs from an absolute boundary k, with `return_logprob` enabled and `logprob_start_len` set to k, the cacheable prefix is capped at $\min(n-1, k)$. This may intentionally shorten a cache hit: with ten logical tokens and `logprob_start_len` set to 4, even a longer cache match is treated as at most a four-token reusable prefix, so the requested logprob span is processed in the current round rather than hidden entirely inside cached KV state.

A separate correctness rule handles positional embedding overrides. When such overrides are present, prefix caching is disabled for the request because identical token IDs with different override vectors do not prove compatible KV state. The mutable scheduling object therefore includes correctness constraints on what may count as a reusable prefix and makes cache lookup part of correctness.

Streaming sessions add another source-level boundary to this logprob account. When a request belongs to a streaming session, asks for logprobs, and supplies a custom nonnegative `logprob_start_len`, SGLang overrides that start length

to −1 before prefix matching. The later source-level extend-batch bookkeeping interprets that sentinel relative to the current filled sequence, so the round returns logprobs for newly produced tokens. Session-held KV is sufficient context for future attention, but it is not a stored transcript of the logits produced at every earlier session position, so the scheduler does not try to reopen an arbitrary earlier boundary inside session-held KV state. This is a mutation of request state for correctness because later accounting depends on that boundary.

Finish state is part of the same request model. A request may finish because it reaches a length bound, emits a stop token, matches a stop string or regex, is aborted, or reaches a grammar terminal state. Chapter 4 treated those outcomes as distributed state-machine transitions. Here, the important point is narrower: a finished request should be filtered from active batch metadata before it can consume another decode slot.

5.3. Scheduler State and Step Selection

Continuous batching is easiest to understand as a transition system. At each iteration, the scheduler has waiting work, resident decode work, and memory state. It must choose a next forward step or prove that no useful step is currently feasible.

The scheduler chooses the next legal transition.

Local notation.

The scheduler-visible state is $S_t = (W_t, B_t, \Lambda_t, U_t, M_t)$. B_t is the resident running batch: the scheduler's decode-resident projection of the broader executing-batch state introduced in Chapter 4. U_t names at most one unfinished prefill continuation. Chapter 6 turns that continuation into the source-level `chunked_req` path. In SGLang source vocabulary, `EXTEND` is the model-facing `ForwardMode` for prefill or prompt-suffix work; the conceptual phase name remains *prefill*. Figure 5.4 later summarizes the same state tuple as the repeated selection loop from visible scheduler state to the next model step.

Definition 5.3.1 (Unfinished prefill continuation) *An unfinished prefill continuation is a request whose prompt work has been partially admitted but whose current prompt suffix is not fully materialized yet. The scheduler must remember this request separately from ordinary waiting work and from decode-resident work until the remaining suffix is processed. We write such a continuation as $U_t = (r, a, b)$, where r is the request, $[a, b)$ is the remaining prompt-suffix interval, and t is the scheduler step that owns the continuation.*

Example 5.3.1 (Half-admitted prompt) Separate from the small running-example table, suppose a scaled request r_{chunk} has a 64-token uncached prompt suffix and the current prompt budget admits only the first 32 positions. After that `EXTEND` step, r_{chunk} has left ordinary waiting state: its first 32 suffix positions have request-row entries and KV locations. It is also still outside decode residency, because positions 32–63 have not been materialized. The scheduler must keep r_{chunk} as the unfinished continuation U_t and resume the remaining suffix before it can enter ordinary decode residency. Figure 5.3 draws why the partly admitted request is neither ordinary waiting work nor decode-resident work.

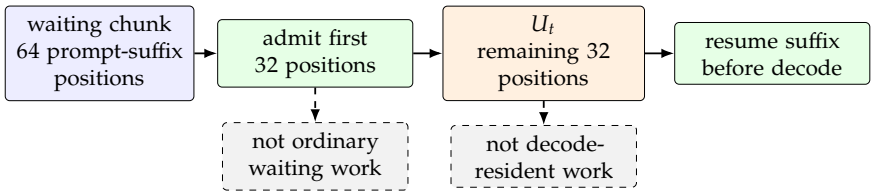

Figure 5.3.: An unfinished prefill continuation is its own scheduler state. Once a prompt suffix has been partially materialized, the request cannot be treated as fresh waiting work, but it also cannot join decode residency until the remaining suffix has been processed.

Chapter 6 specializes this continuation state for chunked prefill. Chapter 5 only needs the scheduler obligation: a partially materialized prompt cannot be treated as a fresh waiting request or as ready decode work.

Definition 5.3.2 (Continuous-batching scheduler state) *The scheduler state at step t is*

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

Here W_t is the waiting queue, B_t is the resident running batch, Λ_t is the most recent prefill batch whose results may need to be merged, U_t is an unfinished prefill continuation if one exists (Definition 5.3.1), and M_t is the memory and cache state visible to the scheduler.

This S_t is a scheduler-decision tuple: waiting work, resident batch, unfinished work, and memory view.

At a step with waiting requests $r_{\text{fresh}}, r_{\text{reuse1}}$, resident decode request r_{pan}, no unfinished prefill, and a memory state that can reserve the next writes, the tuple can be read as $W_t = [\text{fresh}, \text{reuse1}]$, $B_t = [\text{pan}]$, $\Lambda_t = U_t = \varnothing$, and M_t equal to the current token and KV budget account.

Concept-to-code map

- W_t**: waiting work.** The waiting queue maps to `Scheduler.waiting_queue`, whose entries are request objects not yet admitted to the current model step.
- B_t**: resident work.** The running decode set maps to `Scheduler.running_batch`, a `ScheduleBatch` whose rows already own request-pool and KV-cache state.
- Λ_t **and** U_t**: prior or unfinished work.** `last_batch` carries the most recent model result boundary, while `chunked_req` records an unfinished prompt suffix that must keep its request identity.
- M_t**: memory and cache view.** The request-row table, token-to-KV allocator, and prefix cache are the concrete structures consulted by `Scheduler.get_next_batch_to_run` before it admits a step.

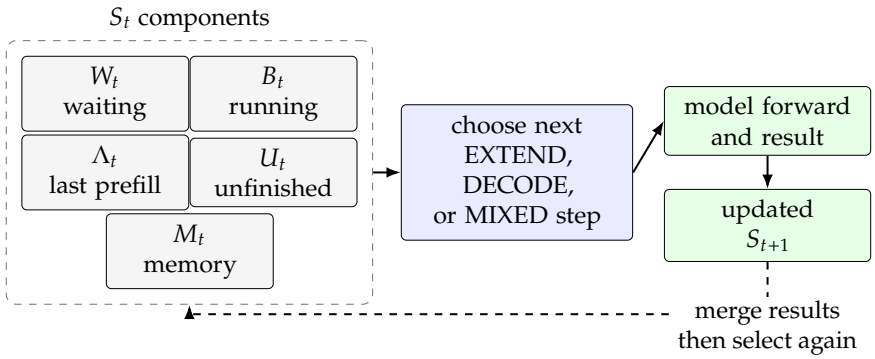

Figure 5.4.: Step selection as a transition from scheduler state to model work. Waiting requests, resident decode rows, unfinished prefill continuations, recent prefill results, and memory state jointly determine whether the next legal step is `EXTEND`, `DECODE`, or `MIXED`.

Definition 5.3.3 (Continuous-batching forward step) *A continuous-batching forward step is one model execution produced by turning scheduler state into a model-facing batch. The selected work item, forward mode, and metadata must agree: the mode is part of the scheduler contract, not just a kernel label. Here the relevant modes are:*

$$\text{EXTEND}, \quad \text{DECODE}, \quad \text{MIXED}.$$

The extend mode carries prompt-suffix spans, sequence lengths, and KV-write metadata for not-yet-cached positions; the decode mode carries one next-token input for each decode-resident request (Definition 5.1.2); and the mixed mode carries both contracts in one execution step without confusing their metadata obligations.

Example 5.3.2 (State selects the step) Let $W_t = [\text{fresh}, \text{reuse1}]$, $B_t = [\text{pan}]$, $\Lambda_t = \varnothing$, and $U_t = \varnothing$. If memory M_t can reserve prompt slots for r_{fresh} and r_{reuse1} as well as r_{pan}'s next decode slot, the scheduler can choose MIXED([fresh, reuse1], [pan]). If prompt admission fails but r_{pan}'s decode slot fits, the same state shape chooses DECODE([pan]). If r_{pan}'s next KV write also fails, the forward step is not legal until retraction, eviction, or a later state update changes M_t. Figure 5.5 draws those state-dependent step choices from the same scheduler state components.

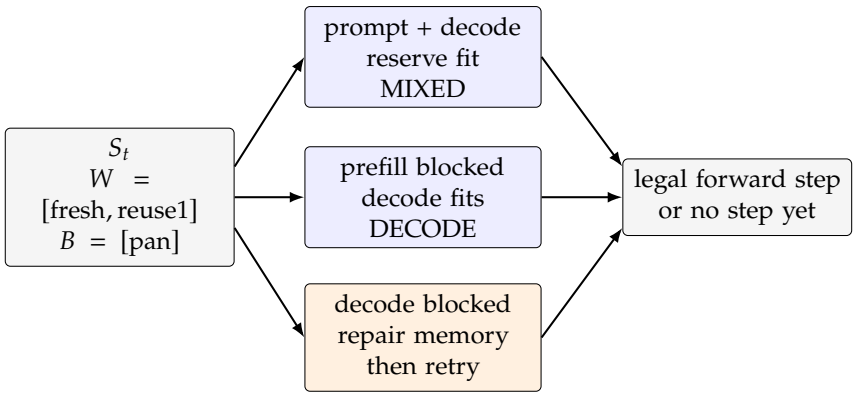

Figure 5.5.: The same scheduler state components can lead to different forward steps depending on memory feasibility. Continuous batching is therefore a state-transition problem over changing request lists.

With the continuous-batching forward step (Definition 5.3.3) named, the simplified transition is only three ordinary phases. First, the scheduler chooses u_t, the next model-facing action. Second, the model worker returns Γ_t, the result bundle for that launched action. Third, result processing merges that bundle back

Algorithm 12: One continuous-batching scheduler iteration. The pseudocode shows the state-ordering contract: completed prefill work is merged before the next decision, waiting requests are admitted only after budgets are refreshed, and decode rows reserve their next KV writes before model execution.

Input: Scheduler state $S_t = (W_t, B_t, \Lambda_t, U_t, M_t)$
Output: Next model work u_t and scheduler bookkeeping for the next state

```
merge completed prefill results in Λ_t into the resident batch B_t;
refresh memory, prefix-cache, and token-budget state in M_t;
initialize candidate prefill work X_t ← ∅;
if U_t contains an unfinished chunk and its page-aligned reserve fits then
    choose the next chunk suffix as X_t;
else
    scan W_t in policy order and append every feasible waiting request to X_t;
if B_t ≠ ∅ then
    reserve the next decode KV writes for B_t, retracting rows if required;
if X_t ≠ ∅ and policy permits mixed work with B_t then
    u_t ← MIXED(X_t, B_t);
else if X_t ≠ ∅ then
    u_t ← EXTEND(X_t);
else if B_t ≠ ∅ then
    u_t ← DECODE(B_t);
else
    u_t ← IDLE;
return u_t and the updated scheduler-side state;
```

into scheduler state by appending sampled tokens, updating cache ownership, filtering finished rows, and refreshing waiting or resident work.

$$S_t \xrightarrow{\text{choose } u_t} (u_t, S_t) \xrightarrow{\text{forward}} \Gamma_t \xrightarrow{\text{process result}} S_{t+1}.$$

Here u_t is the selected model work for this scheduler step, and Γ_t is the model result object that output processing, cache updates, and finish checks consume. The implementation has more cases, including disaggregation, overlap, speculation, data-parallel synchronization, and health handling. Those cases do not change the core idea: step selection is a repeated conversion from scheduler state into a model-worker batch.

For the running example, one possible state is

$$W_t = (\text{fresh}, \text{reuse1}, \text{reuse2}), \quad B_t = (\text{pan}), \quad \Lambda_t = \varnothing, \quad U_t = \varnothing.$$

The scheduler may choose a prefill step for r_{fresh} and r_{reuse1} if the admission budget allows it. If no waiting request is feasible, it should update and run r_{pan} as a decode batch, provided r_{pan} can reserve the next KV write. If a previous prefill batch has just completed, the scheduler first filters and merges it into the running batch before deciding the next step.

This merge-before-next-step discipline is what makes the batch continuous. After r_{reuse1} finishes its prefill step, it is no longer a waiting prompt. It becomes a decode-resident request alongside r_{pan}. The next decode step may therefore contain r_{pan} and r_{reuse1} even though r_{pan} began decoding earlier. Batch identity changed; request identity did not.

Algorithm 12 is a teaching abstraction of `Scheduler.get_next_batch_to_run` with a narrow scope. The listing keeps the invariant that matters across ordinary, chunked, mixed, and retracted execution: the scheduler first makes state ownership current, then chooses an executable batch whose KV and token budgets are already legal for the forward path.

Example 5.3.3 (One full scheduler iteration) Run Algorithm 12 on the chapter's running state:

$$W_t = (r_{\text{fresh}}, r_{\text{reuse1}}, r_{\text{reuse2}}), \quad B_t = (r_{\text{pan}}), \quad \Lambda_t = U_t = \varnothing,$$

with page size $P = 4$ and 20 free or evictable KV slots. The policy pass uses cache-aware order, so it scans r_{reuse1}, r_{reuse2}, then r_{fresh}. The resident decode row gets a one-page reserve for r_{pan}'s next-token path, leaving 16 slots for waiting prefill work.

The budget trace is page-rounded. The resident r_{pan} decode row uses a reserve of 4 slots, reducing the available budget from 20 to 16, and stays in B_t. Request r_{reuse1} has $6 - 4 = 2$ suffix tokens, rounds to 4 slots, reduces the budget to $16 - 4 = 12$, and enters X_t. Request r_{reuse2} has $10 - 4 = 6$ suffix tokens, rounds to 8 slots, reduces the budget to $12 - 8 = 4$, and also enters X_t. Request r_{fresh} has $8 - 0 = 8$ suffix tokens and needs 8 slots, so the remaining 4-slot budget is insufficient and it stays waiting.

The chosen prefill set is therefore $X_t = (r_{\text{reuse1}}, r_{\text{reuse2}})$, and the decode set is still $B_t = (r_{\text{pan}})$. If the policy permits mixed work, the iteration returns $\text{MIXED}((r_{\text{reuse1}}, r_{\text{reuse2}}), (r_{\text{pan}}))$. If mixed work is disabled, the same budget scan returns $\text{EXTEND}(X_t)$ and leaves r_{pan}'s protected decode row for the next legal decode step; if no waiting row had fit, the reserved decode row would produce $\text{DECODE}(r_{\text{pan}})$.

Concept-to-code map

SGLang's `ForwardMode` in `python/sglang/srt/model_executor/forward_batch_info.py` records this step type for the model-facing path. The mode determines which metadata fields must be populated, which attention metadata is initialized, whether CUDA graph paths are eligible, and how sampling or logits processing interprets the result.

5.4. Admission Control and Token Budgets

Figure 5.6 shows how token capacity, KV slots, request slots, and delayed work meet at the admission boundary.

Admission control answers a concrete question: which waiting requests may enter the next prefill batch without invalidating the decode set or overcommitting KV memory? The answer is stricter than prompt-token capacity. The scheduler must account for page alignment, future decode reserve, request slots, chunk limits, and cache eviction.

Admission is a budget proof over tokens, pages, and future decode slots.

This is also where serving-level partial admission becomes a batch construction decision. A request may contribute only the feasible prefix of its remaining prefill

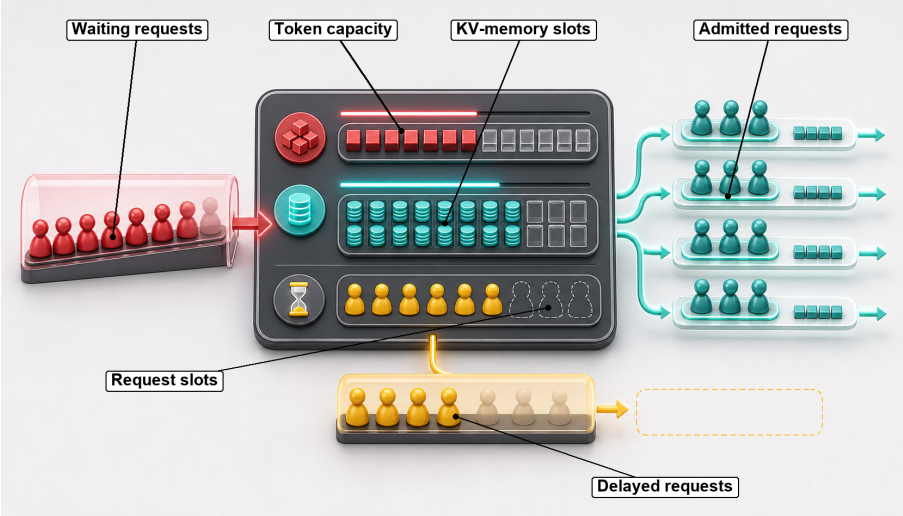

Figure 5.6.: Admission control and token budgets. A scheduler budget panel tracks token capacity, KV-memory slots, request slots, waiting requests, admitted requests, and delayed requests.

suffix to the next prefill batch, while the suffix that did not fit stays waiting under the same request identity. The chunked-prefill chapter calls that admitted prefix a prefill chunk; here, it is enough to recognize that chunked prefill is partial admission expressed as a model-facing batch boundary.

Definition 5.4.1 (Admission feasibility and page-aligned charge) *Let P be the KV page size and let*

$$\mathrm{alloc}_P(n) = P\left\lceil \frac{n}{P} \right\rceil$$

be the page-aligned allocation for n token positions. For a candidate effective prefill length e_r, future decode reserve ρ_r, and one page of conservative allocator overhead, the simplified admission charge is

$$\mathrm{charge}_P(r) = \mathrm{alloc}_P(e_r) + \rho_r + P.$$

A request r is admission-feasible for a prefill batch when this charge, request-slot demand, and queue-policy constraints fit the currently available or evictable scheduler budget. A simplified sufficient capacity inequality is

$$\mathrm{charge}_P(r) \le A_t + E_t - R_t^{\mathrm{cap}}.$$

e_r is raw suffix work; $\mathrm{charge}_P(r)$ is the admission charge.

Here e_r is the effective prefill length, ρ_r is a future decode reserve, A_t is immediately available KV capacity, E_t is evictable cache capacity, and R_t^{cap} is capacity already reserved for resident or newly chosen requests. The extra P term is a physical allocator guard: the admission path keeps one page of headroom around the extend allocation so a request that barely fits by raw suffix length does not strand the next page-boundary write. The right-hand side is also physical. A_t is free now, E_t is not free yet but can be made usable by evicting unprotected cache entries, and R_t^{cap} is subtracted because resident rows and requests already chosen earlier in the same admission pass have first claim on that capacity. This formula is smaller than SGLang's full implementation, but it captures the reason page-based serving cannot use raw token counts alone [19].

Account	What it stores	Hand-checkable failure
Request-pool row	Per-request mapping capacity, such as the row M_r that will hold logical-position entries.	KV pages may be free while no request row is available for a new admission.
Token-to-KV slots	Physical cache slots or pages where attention state will be written.	Request rows may be available while page-aligned KV capacity is short.
`out_cache_loc`	The current step's flat list of KV write locations.	The request can be admitted only if every scheduled token row has a write location in the same flat order.
Future decode reserve	Capacity held for near-future one-token decode writes.	The suffix fits now, but the request would immediately starve its first decode step.

Table 5.5.: Admission-control resource accounts. Each account stores a different slot type and exposes a different hand-checkable failure.

Admission also has more than one kind of slot. A request can fit one account and fail another, so the scheduler keeps the accounts separate: For example, a scheduler with two free request rows, twelve free KV slots, and a four-token current-step cap may admit two short requests but reject a third for request-row capacity, or reject one long request because its eight raw suffix tokens round to twelve KV slots plus future reserve. These are different failure modes, even though all of them surface as "do not add this request to the next batch."

The reserve ρ_r is an estimate, not an oracle. A useful mental model is

$$\rho_r \approx \eta_t \min\{m_r - g_r, \mathrm{CLIP}_{\mathrm{new}}\},$$

where m_r is the request's `max_new_tokens` limit, g_r is the number of already generated tokens, $\mathrm{CLIP}_{\mathrm{new}}$ is the scheduler's clipped estimate for very large output limits, and $\eta_t \in (0, 1]$ is the scheduler's current new-token ratio. When $\eta_t = 1$, the reserve treats unfinished requests as if they may reach their clipped output limits; smaller values reserve that fraction so admission can proceed when observed stops make the full limit too conservative. For example, $m_r = 128$, $g_r = 8$, $\mathrm{CLIP}_{\mathrm{new}} = 4096$, and $\eta_t = 0.5$ give $\rho_r \approx 60$ future decode slots. SGLang's admission helper implements this idea by clipping very large output limits, scaling running-request reservations by a new-token ratio, and adding page overhead; a truncated prefill chunk may reserve no future decode tokens until the final prompt chunk is admitted.

If the estimate is too large, admission becomes conservative and useful prefill work waits. If it is too small, the next decode-memory check can fail, forcing decode retraction and a recalibrated new-token ratio. In the extreme, when even the last decode row cannot reserve its next write, SGLang aborts that row rather than letting the allocator overrun.

In the running example, r_{reuse2} has $e_{\mathrm{reuse2}} = 6$. With page size 4, $\mathrm{alloc}_4(6) = 8$. If r_{reuse2} also reserves $\rho_{\mathrm{reuse2}} = 4$ future decode slots and one page of overhead, its simplified demand is $8 + 4 + 4 = 16$, not six. If the remaining budget after protecting r_{pan} is 12, r_{reuse2} cannot be admitted alone under this model even though its raw effective prefill length is only six.

Example 5.4.1 (Raw tokens versus allocated slots) With $P = 4$, three effective prefill lengths $2, 4, 5$ allocate $4, 4, 8$ slots. The raw work is 11 tokens, but the allocation is 16 slots. If the scheduler has 15 slots, the raw count says yes and the page-aligned count says no. The page-aligned count is the one that protects the allocator.

Concept-to-code map

SGLang's admission helper `PrefillAdder` in `python/sglang/srt/managers/schedule_policy.py` is the admission-budget object. For this chapter, read it as tracking remaining prefill-token budget, remaining chunk-token budget when chunked prefill is enabled, total-token reserve for running requests, request-count limits, and priority preemption. Later chapters add variant-specific cache budgets such as sliding-window or hybrid-state constraints. Its `budget_state` returns whether admission can continue, must stop due to token exhaustion, or must stop for another policy reason.

Prefill delay enters before the per-request loop as an admission gate. When enabled, SGLang samples current pool usage, creates a single-pass delayer, and lets `PrefillAdder.add_one_req` negotiate whether local prefill should be allowed at all for this scheduling pass. If the cross-rank decision says to delay, the helper returns a policy stop even when an individual request might fit the token inequality. This is why prefill delay should be modeled as a scheduling transition: it can prevent admission before the request-level budget proof is attempted.

The most important insight is that the admission decision is predictive. The scheduler does not know how many tokens a request will actually generate before an end-of-sequence (EOS) or stop condition. It uses bounded reserves, such as a clipped `max_new_tokens` contribution, to avoid filling memory with prompt state while leaving no room for future decode. That predictive reserve is why continuous batching is both a scheduling problem and a memory-risk problem.

5.5. Queue Policy: Arrival, Priority, and Reuse

When multiple waiting requests are feasible, the scheduler still has to choose an order. The simplest rule is first come, first served. That rule is easy to reason about, but it ignores reuse and urgency. In an LLM runtime, two requests that share a long prefix are not independent from a cost perspective; scheduling one can make the other cheaper.

Queue order is a policy choice once reuse enters the cost model.

Definition 5.5.1 (Cache-aware priority) *A cache-aware queue policy assigns scheduling priority using expected reuse of prefix state, while leaving admission feasibility and decode protection as separate gates. The policy must distinguish reuse that is already resident from reuse that would become available only after another waiting request runs. It may still consider sequence length, memory demand, external priority, and queue time, but it is cache-aware only when prefix-cache structure can change the ordering. Write its scoring rule as*

$$\kappa(i) = \phi(h_i^{\text{res}}, h_i^{\text{sim}}, m_i, q_i, \alpha_i),$$

where h_i^{res} is resident reusable-prefix length, h_i^{sim} is simulated in-batch reuse, m_i is estimated memory demand, q_i is queue age, and α_i is any external priority.

Example 5.5.1 (Cached prefix changes queue order) Suppose the waiting queue contains a fresh request followed by two reuse requests. The fresh request arrived first but has no cached prefix. The two reuse requests arrived later and each match four resident prefix tokens. A cache-agnostic FCFS policy chooses the fresh request. A cache-aware policy may choose a reuse request first because its effective prompt work is shorter. If r_{pan} is decode-resident (Definition 5.1.2) and high priority, the same policy may still protect r_{pan}'s next-token step before admitting either cached prompt. Cache awareness changes the queue score; it does not remove feasibility or priority constraints. Figure 5.7 draws the same FCFS-versus-cache aware choice.

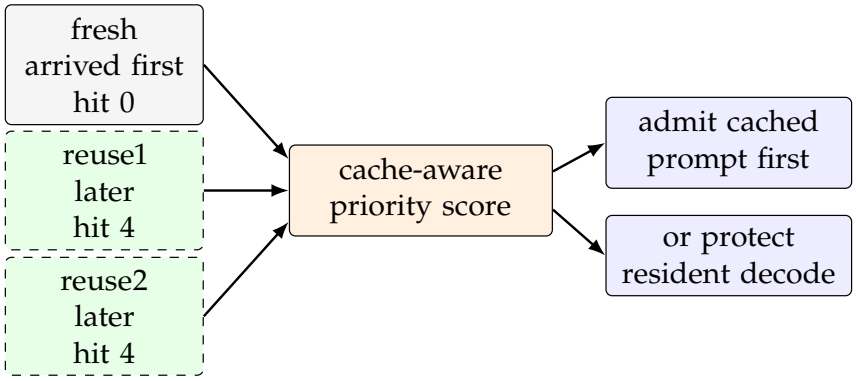

Figure 5.7.: Cache-aware priority scores feasible waiting requests using reuse state. The dashed boxes (reuse1, reuse2) carry a cached prefix; the solid box (fresh) does not. A cached prefix can move a later request ahead of FCFS order, but the final choice still has to respect decode protection and memory feasibility.

Concept-to-code map

SGLang's `SchedulePolicy` supports cache-agnostic policies such as FCFS, longest-output-first, random ordering, and routing-key affinity. It also supports cache-aware policies such as longest-prefix match and DFS-weight ordering over prefix-cache structure. The policy path is implemented in `python/sglang/srt/managers/schedule_policy.py`; prefix matching is also one of the central ideas in SGLang's RadixAttention design [49].

A *radix tree* here is just a prefix tree whose edges carry token-run labels: a path from the root spells out a token sequence, and two requests that share a leading run share the path up to where they diverge. Chapter 5 does not need the full resident-cache data structure yet. It only needs the scheduling capsule: a tree can

Algorithm 13: Waiting-queue ordering policy dispatch. The policy pass orders requests only; memory admission and decode protection are checked later by the scheduler.

Input: Waiting queue W; running batch $\mathcal{B}$; prefix tree $\mathcal{T}_{\text{prefix}}$; configured policy Π; priority settings
Output: W ordered for later admission checks

choose the active policy, falling back from LPM to FCFS when the queue is too large;
compute the ordering key for the active policy: FCFS and priority time, cache-aware prefix records, output length, random shuffle key, or routing-key affinity against $\mathcal{B}$;
if *the active policy is cache-aware* **then**
    compute prefix records P and delayed set D using in-batch reuse simulation;
    place requests whose best hit is only simulated after ordinary resident-cache hits;
sort or shuffle W using the selected ordering key, without changing request state or allocator ownership;
return W;

Algorithm 14: Prefix-hit scoring with in-batch reuse simulation. The helper records existing prefix-cache benefit and delays requests whose best reuse depends on waiting work that has not run yet.

Input: Waiting queue W; prefix tree $\mathcal{T}_{\text{prefix}}$; in-batch check threshold; deprioritization threshold
Output: Prefix records P for sorting and delayed set D

clear the simulated waiting-queue radix tree $\mathcal{T}_W$;
$P \leftarrow \emptyset$, $D \leftarrow \emptyset$;
foreach *request* $r \in W$ **do**
    let k_r be r's prompt plus output suffix;
    match k_r against $\mathcal{T}_{\text{prefix}}$ and record hit, node, and policy weight in $P[r]$;
    if *r's existing hit is below the in-batch check threshold* **then**
        match k_r against $\mathcal{T}_W$;
        if *the simulated in-batch hit reaches the deprioritization threshold* **then**
            add r to D;
        else
            insert k_r into $\mathcal{T}_W$ with a simulated value;
return *P and D*;

estimate whether one waiting request already has a useful prefix hit, or whether its best reuse depends on another waiting request that has not run yet.

The two trees below play different roles. The resident prefix tree $\mathcal{T}_{\text{prefix}}$ represents reusable KV state that already exists. The simulated waiting tree $\mathcal{T}_W$ is a dry run over the waiting queue; it allocates no KV pages and proves no real cache hit. Its only job is to keep the policy from treating "request B could reuse request A's prefix" as an immediate benefit when A has not executed yet. In plain terms, the simulation asks who would have to run first before a claimed prefix reuse becomes real.

Algorithm 13 is the scheduler-facing policy dispatch. It returns FCFS immediately when configured; otherwise it selects the active policy, applies cache-aware or priority-specific ordering, and leaves memory admission and decode protection for the later admission check. Prefix matches affect order, but they do not override memory feasibility or decode protection.

Algorithm 14 isolates the SGLang-specific cache-aware scoring step. The simulated waiting-queue tree prevents a group of weak existing hits from all moving ahead merely because they could reuse state that none of them has created yet. Here *weak* is local policy language, not a new cache state: it means the real resident-cache hit is below the in-batch check threshold. For example, if a deployment checks in-batch reuse only below a 64-token resident hit and deprioritizes simulated hits

Request	Resident hit	Simulated hit	Priority or decode protection	Final order
R_1	Strong	Not needed	No live decode row displaced	First
R_2	Weak	None yet; inserts key into $\mathcal{T}_W$	Eligible after strong hits	Second
R_3	Weak	High, but only through R_2's pending key	Delayed because reuse is not resident	Delayed
Decode row D	Existing KV	Not scored as waiting prefill	Protected when memory reserve is tight	Kept live

Table 5.6.: Queue ordering after reuse and priority scoring. The score changes ordering but not decode-residency feasibility.

of 128 tokens or more, then a request with a 40-token real hit may be delayed if its apparent 160-token benefit depends on another waiting request running first. Those numbers are illustrative; the important rule is that simulated future reuse cannot outrank already resident reuse without passing the configured thresholds.

Queue policy chooses among feasible requests; it cannot spend memory or token budget twice.

Read the pass in order. First, each request is matched against the real resident prefix cache and records whatever reusable KV state already exists. Second, requests with weak real hits are tested against the simulated waiting tree. Third, a request whose best hit comes from that simulated tree is delayed because its reuse depends on earlier waiting work. Finally, non-delayed weak-hit requests insert their keys into the simulated tree so later siblings can be classified correctly.

Example 5.5.2 (Scoring the three waiting requests) The formula in Definition 5.5.1 is intentionally policy-shaped, so use one concrete scoring rule only as a tunable default for debugging and explanation, not as a universal SGLang constant:

$$\kappa(i) = 2h_i^{\text{res}} + h_i^{\text{sim}} + 0.5q_i + 4\alpha_i - 0.25m_i.$$

Here m_i is the effective prefill suffix, q_i is queue age in scheduler ticks, and α_i is an external-priority bit. The coefficient directions are the policy contract: Increasing the coefficient on h_i^{res} moves already-resident prefix hits earlier, improving reuse and prompt efficiency while potentially pushing fresh prompts later. Increasing the coefficient on h_i^{sim} rewards reuse predicted by the waiting-queue simulation, which can cluster related prompts but must not treat simulated reuse as resident. Increasing the coefficient on q_i moves older waiting requests earlier, reducing starvation at the cost of weakening pure cache affinity. Increasing the coefficient on α_i moves externally favored work earlier, improving selected service classes while increasing fairness pressure. Increasing the magnitude of the $-m_i$ term penalizes larger remaining prefill suffixes, protecting capacity and short requests while potentially delaying long prompts.

For the running waiting requests, take r_{fresh} with $(h^{\text{res}}, h^{\text{sim}}, m, q, \alpha) = (0, 0, 8, 6, 0)$, so $\kappa = 1.0$; r_{reuse1} with $(4, 0, 2, 4, 0)$, so $\kappa = 9.5$; and r_{reuse2} with $(4, 0, 6, 2, 0)$, so $\kappa = 7.5$. This pass orders r_{reuse1} before r_{reuse2} and both before r_{fresh}. The resident high-priority r_{pan} row is not a waiting-prefill score; it is protected later by the decode-residency and admission gates. The numeric score changes queue order, not feasibility.

Chapter 8 defines the structure SGLang uses for the resident-cache tree $\mathcal{T}_{\text{prefix}}$; the scoring pass borrows the same shape for a throwaway tree $\mathcal{T}_W$ that models only the requests still waiting in this pass. Figure 5.8 shows why the simulation is needed.

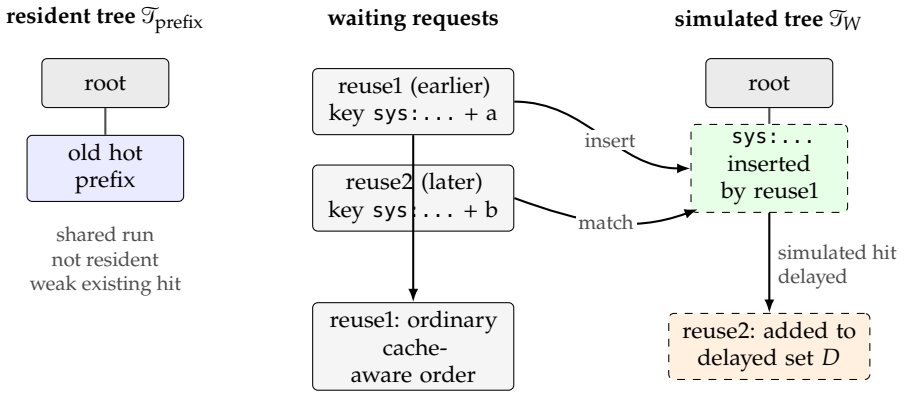

Figure 5.8.: The simulated waiting-queue tree $\mathcal{T}_W$ separates real reuse from circular reuse. Both reuse requests share a leading run that is *not* resident in $\mathcal{T}_{\text{prefix}}$, so both have a weak existing hit. The scoring pass inserts the earlier request's key into the throwaway tree $\mathcal{T}_W$; the later request then matches that simulated entry, revealing that its best reuse depends on work the earlier request has not run yet. It is moved to the delayed set D (dashed) so it is ordered after its provider instead of jumping the queue on reuse that does not exist yet.

In the running example, FCFS may admit r_{fresh} before r_{reuse1} and r_{reuse2}. Longest-prefix match in Algorithm 13 would prefer r_{reuse1} and r_{reuse2} over r_{fresh} because both have a cached prefix of length four. If only one of r_{reuse1} and r_{reuse2} can fit, r_{reuse2} has more remaining prefill work; choosing r_{reuse1} may reduce first-token latency for r_{reuse1}, while choosing r_{reuse2} may make better use of a larger prefix hit. The in-batch simulation in Algorithm 14 would handle a variant where the two reuse rows mostly depend on reuse from each other rather than reuse already present in $\mathcal{T}_{\text{prefix}}$. No ordering rule eliminates tradeoffs.

Priority scheduling adds another dimension. A high-priority request can be placed ahead of ordinary work, and under configured thresholds may preempt running lower-priority requests. In the running example, r_{pan} is already decode-resident and high priority. A policy may protect r_{pan}'s decode latency by refusing to admit r_{fresh}, r_{reuse1}, and r_{reuse2} together even when their prompt allocations seem to fit. Priority improves service for selected classes but creates a fairness obligation for the rest of the queue.

Concept-to-code map

- **Priority gate.** The policy decision to let a higher-priority waiting request challenge resident work lives in the scheduler admission path in `python/sglang/srt/managers/scheduler.py`.
- **Victim selection.** The lower-priority rows that may be released map to `PrefillAdder.preempt_to_schedule` in `python/sglang/srt/managers/schedule_policy.py`.
- **Release boundary.** The actual row release maps to `ScheduleBatch.release_req` in `python/sglang/srt/managers/schedule_batch.py`; the preemption guard skips finished-but-not-yet-filtered rows so release and filtering remain distinct scheduler events.

Algorithm 15: Priority preemption victim selection with a finished-request guard. Finished rows are skipped because their KV cache may already have been released by result processing even though batch filtering has not yet removed them.

Input: New request r; running batch $\mathcal{B}$; preempted set $\mathcal{E}$; remaining-token budget M; threshold δ
Output: A sufficient lower-priority victim set, or $\emptyset$

if *priority preemption is disabled* **then**
  return $\emptyset$;
$\mathcal{C}_{\text{cand}} \leftarrow$ unfinished rows in $\mathcal{B} \setminus \mathcal{E}$, sorted from least to most preferred;
$N \leftarrow r.\text{extendLen} + \min(r.\text{maxNewTokens}, \text{CLIP}) - M$;
$\mathcal{Y}_{\text{victim}} \leftarrow \emptyset$;
foreach *candidate* $x \in \mathcal{C}_{\text{cand}}$ **do**
  if *the priority gap from* r *to* x *is at most* δ **then**
    break;
  add x to $\mathcal{Y}_{\text{victim}}$;
  subtract x's token offset from N;
  if $N \leq 0$ **then**
    break;
if $\mathcal{Y}_{\text{victim}} = \emptyset$ *or* $N > 0$ **then**
  return $\emptyset$;
return $\mathcal{Y}_{\text{victim}}$;

Algorithm 15 makes the finished-request guard operational. First, the candidate set excludes rows that have already been selected or already finished. Then the scan stops as soon as priority or capacity says the proposed release is no longer justified. Finally, the routine returns victims only when they are sufficient; the running batch is still untouched.

Algorithm 16: Committing priority preemption after victim selection. Scheduler resources are released only for rows whose ownership was proven valid before the running batch is filtered.

Input: Running batch $\mathcal{B}$; selected victims $\mathcal{Y}_{\text{victim}}$; already preempted set $\mathcal{E}$
Output: Whether the batch was mutated for priority preemption

```
if Y_victim = ∅ then
    return false;
foreach row x ∈ B do
    if x ∈ Y_victim then
        release x's scheduler resources immediately;
filter B to rows not in Y_victim;
append Y_victim to E;
return true;
```

Algorithm 16 is the mutation boundary that follows victim selection. First, an empty victim set leaves the batch unchanged. Then selected rows release scheduler resources before filtering removes them from the resident batch. Finally, the preempted set records the rows so later policy passes do not release them again. This ordering keeps priority preemption from turning a latency policy into an allocator bug: a row whose finish path has already released KV cache is no longer a valid preemption victim.

Routing-key ordering is different from prefix reuse. A routing key groups requests by an application-level affinity visible to the scheduler. In SGLang, the routing-key policy can prefer waiting requests whose key is common in the running batch. This is useful when co-scheduling by affinity improves locality or downstream behavior, but it should not be confused with a proof of token prefix reuse.

5.6. Constructing a Prefill Batch

A prefill batch is the executable form of an admission decision.

Once admission chooses requests, the scheduler must give ModelRunner an executable batch. The model does not consume a Python list of abstract requests. It consumes token tensors, positions, sequence lengths, request-pool indices, cache-write locations, sampling metadata, and optional side metadata. The batch is the alignment of those objects.

The row-order obligation is the local idea. A *request-pool index* names the runtime row for one request. *Cache-write locations* name the KV slots that new token state will fill. *Sampling metadata* carries row-ordered token-choice policy, including temperature, filters, penalties, logit bias, custom-processor masks, seeds, and grammar objects. *Side metadata* carries optional logprob, multimodal, token-type, embedding-override, or model-family fields in that same request order. The later Definition 14.1.5 expands the sampling side; this section only needs the fact that all of those fields share one request axis.

That alignment carries more than a shape convention. Batching deliberately breaks each request object into model-efficient tensors and side tables, so request identity has to be reconstructed by a shared order and by token-span coordinates.

Definition 5.6.1 (Scheduler-to-model identity agreement) *For a batch with ordered request list $(r_1, \ldots, r_b)$, scheduler-to-model identity agreement means that every model-facing field produced from those requests preserves the same request identity after batching decomposes the requests into tensors and side tables. If slot j in a per-request metadata object refers to r_j, then the sequence lengths, request-pool index, sampling state, logprob state, grammar state, multimodal state, and the flat-token span and cache-write locations derived from that slot must also describe r_j. Compatible tensor shapes are necessary but not sufficient: a batch can have the right first dimensions while assigning one request's side state or KV writes to another request.*

This input-side contract is the counterpart of Definition 4.5.4. Both say that one request-order axis governs all per-request arrays; here the check applies before model execution, while Chapter 4 applies it after batched outputs leave the model path. Once request objects have been flattened into tensors, the model path cannot recover a lost identity association from tensor shape alone. The sampling-state swap in the next example mirrors Chapter 4's swapped finish-reason example.

Example 5.6.1 (Two-request prefill metadata) Let the ordered request list be (fresh, reuse1). r_{fresh} has an eight-token prefill suffix, meaning eight prompt tokens still have to run in this pass, and final sequence length 8; r_{reuse1} has a two-token prefill suffix and final sequence length 6. The flat input tensor therefore contains ten token rows, but `seq_lens` $= [8, 6]$, `extend_lens` $= [8, 2]$, request-pool indices, sampling state, and output-cache spans must all use the same request order (fresh, reuse1). If the sampling state is ordered (reuse1, fresh), the tensor shapes can still look legal while the next-token policy, grammar mask, temperature/filter settings, or logprob records are assigned to the wrong request. Figure 5.9 draws the same request-order contract across flat tokens, lengths, cache spans, and sampling state.

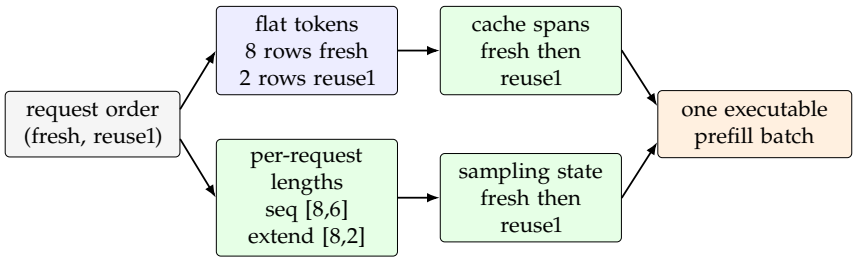

Figure 5.9.: Scheduler-to-model identity agreement means that flat token spans and per-request metadata share one request order. The model-facing batch is correct only when lengths, row indices, cache locations, and sampling state all name the same requests.

For a prefill batch, the essential construction is:

$$\text{input suffix for } r_i = z_i[p_i : |z_i|],$$

where $z_i = x_i \| y_i$ and p_i is the cached-prefix length. The construction is not a matrix transpose; it is a split between two indexing axes. Token-indexed objects, such as suffix token IDs and KV-write locations, have one row per admitted suffix token and are flattened in suffix order. Request-indexed objects, such as sequence lengths, prefix lengths, request-pool rows, and sampling state, have one entry per admitted request and remain in scheduler order. Attention backends then use the request-indexed metadata to interpret the flat rows as separate causal sequences.

Flat token rows follow suffix order; request-indexed metadata still follows the ordered request list.

Algorithm 17: Constructing an extend batch after admission. The routine turns an ordered request list into flat token tensors, per-request lengths, KV-write locations, side metadata, and sampling metadata that all name the same requests.

Input: Ordered admitted requests $R = (r_1, \ldots, r_b)$; cached-prefix state; request-pool allocator; token-to-KV allocator; optional logprob, multimodal, token-type, embedding-override, and Mamba tracking metadata [14]
Output: An extend-mode scheduler batch whose tensor and metadata rows agree

set the batch forward mode to extend, or to the diffusion-LM extend mode when that model family is active;
derive cached-prefix lengths, flat suffixes, final lengths, and model-family length fields in scheduler order;
flatten all input suffixes, preserving the order of R;
build device tensors and batch fields for token ids, prefix lengths, extend lengths, final sequence lengths, and total extend-token count;
allocate request-pool rows and output KV slots for the same ordered requests;
assign request rows, prove each final length minus prefix length equals the extend length, and mark request-level KV lengths;
translate optional logprob, embedding, multimodal, token-type, and Mamba side metadata into the same flat-suffix and request-row coordinates;
build row-ordered sampling metadata from the completed scheduler batch;
return *the extend-mode batch contract consumed by the model-worker path*;

In the running example, a prefill batch containing r_{fresh} and r_{reuse1} has suffix lengths 8 and 2. Its flattened token volume is 10, but its per-request lengths are different: r_{fresh}'s final sequence length after prefill is 8, while r_{reuse1}'s is 6. The batch therefore needs both a token-indexed flat input tensor and request-indexed arrays such as `seq_lens`. Losing the request-indexed axis would make the attention backend unable to separate r_{fresh}'s eight positions from r_{reuse1}'s two positions.

Concept-to-code map

The chapter's prefill-batch construction step maps to `ScheduleBatch.prepare_for_extend` in `python/sglang/srt/managers/schedule_batch.py`. The book's flat suffix tensor maps to `input_ids` and `extend_num_tokens`; its request-indexed lengths map to `seq_lens`, `prefix_lens`, and `extend_lens`; and its future KV writes map to output cache locations allocated through the memory layer. Input logprob requests add `extend_input_logprob_token_ids`. Embedding overrides map absolute request positions through the cached prefix and flat-suffix offset into `replace_embeds` and `replace_positions`. The result is later transformed through the model-worker path into `ForwardBatch`, whose source comment summarizes the chain:

ScheduleBatch → ModelWorkerBatch → ForwardBatch.

Algorithm 17 has a construction job and a correctness job. First, it turns an admission result into the tensor surfaces the model-worker path consumes: flat suffix tokens, length ledgers, request-pool rows, and KV-write locations. Then it translates side channels such as logprob targets, embedding overrides, multimodal inputs, token-type IDs, and Mamba tracking fields into the same batch contract. Finally, it builds row-ordered sampling metadata, so every product remains indexed by the same request order even though suffix tokens are flattened.

SGLang's extend batch preparation follows this shape, including the request-level KV committed and allocated lengths and the conversion of side-channel metadata into flat-token coordinates.

For Mamba-like layers, the side metadata includes the tracking mask, tracked sequence length, and destination state slot described by Definition 11.4.2. When the extra Mamba tracking buffer is enabled, that destination may come from an alternating per-request tracking pair; those fields obey the same request order as lengths, cache locations, and sampling state. This is the recurrent-state analogue of keeping `out_cache_loc` aligned: the saved object is a convolution-history-plus-SSM snapshot at a boundary, not a per-token KV span.

The important abstraction is that `prefix_lens` and `out_cache_loc` are part of the same claim. A prefix length says how much context should already be available. An output cache location says where newly computed KV state will be written. If these disagree, the model can read the wrong context, overwrite the wrong cache slots, or return logits for a position that does not correspond to the request's logical sequence.

Logprob and multimodal metadata make the consistency requirement stricter. For input logprobs, the boundary from Definition 5.2.2 selects which suffix rows need prompt-token scores. For a nonnegative boundary k_i, cached-prefix length p_i, and current suffix length e_i, the local start offset s_i stored as `extend_logprob_start_len` is

$$s_i = \min(\max(k_i, p_i) - p_i, e_i).$$

The aligned object `extend_input_logprob_token_ids` is a shifted target-token vector, not another copy of the input suffix.

Example 5.6.2 (Shifted target IDs for input logprobs) If an original input is $[1, 2, 3, 4]$ and the current admitted suffix slice is $[1, 2]$, the logits rows for that slice gather target IDs $[2, 3]$. If the current admitted suffix slice is $[3, 4]$, the target vector is $[4, 0]$, where 0 is padding after the last original input token. The offset formula gives the same boundary in flat suffix coordinates. Suppose the cache has already covered $p_i = 2$ prompt tokens, the logprob boundary asks for scores starting at $k_i = 3$, and the current suffix length is $e_i = 2$. Then

$$s_i = \min(\max(3, 2) - 2, 2) = 1,$$

so the first suffix row is skipped and the second suffix row gathers target 4. If $k_i \le p_i$, the maximum clamps the start to the cached boundary; if k_i lies past the admitted suffix, the final minimum clamps the start to e_i.

This vector is for input-token logprobs; output-token logprobs are accumulated on the generation path, so the shifted IDs here are taken from `origin_input_ids` rather than from generated `output_ids`. The vector is easy to materialize while the scheduler still has `origin_input_ids`, prefix lengths, logprob-start boundaries, and chunk boundaries; the logits processor later sees flat rows and can gather one vocabulary column per row from the materialized GPU tensor.

A multimodal request may carry token-embedding overrides or position metadata. Extend-batch construction must convert each override from request coordinates

to flat suffix coordinates before the vectors reach the model runner. The later multimodal transformer serving chapter gives the full multimodal version of this contract. The scheduler still has to preserve a single request ordering across all of these side channels. That is why Definition 5.6.1 states a correctness requirement for the model-facing identity agreement.

5.7. Maintaining the Decode Batch

Decode residency turns a request into one-row recurring work.

After a request has enough cached context to generate, it becomes a resident decode request. Decode has a compact per-step shape: each resident request contributes one next-token input and, in the non-speculative case, needs one new KV write location for the result.

Definition 5.1.2 named the status. This section studies how the runtime preserves it across repeated decode steps, filtering, and memory reservation.

Example 5.7.1 (Resident set across two decode steps) At step t, r_{reuse1} and r_{pan} are decode-resident. The scheduler forms a two-row decode batch, uses each request's previous output token as the next input, and allocates one new KV location per request. If r_{reuse1}'s sampled token reaches its stop condition, result processing removes r_{reuse1} before step $t + 1$. r_{pan} stays resident with an advanced sequence length, so the next decode batch has one row rather than two. Figure 5.10 draws that recurrence from a two-row decode batch to a one-row survivor batch.

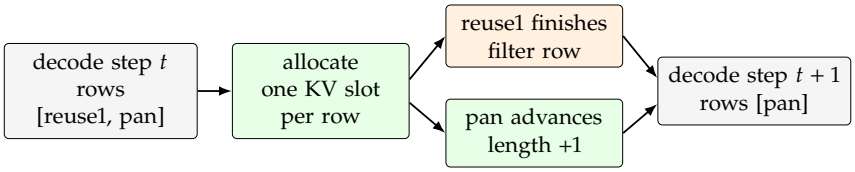

Figure 5.10.: Decode residency is recurring one-token work for live requests. A resident request stays in the running batch only while it remains unfinished, keeps valid metadata, and can reserve the next KV write.

The important point is that membership changes while the per-row decode contract does not. Adding r_{reuse1} changes the resident set from one row to two, and filtering it later changes the set back to one; each surviving row still contributes exactly one next-token input and one possible KV write.

Decode advances one logical token per live row; page allocation happens only at page boundaries.

Before a decode step can launch, the runtime checks whether enough memory is available. With page size P, a non-speculative request needs a new page only when its committed length is exactly at a page boundary. A mid-page request already owns the physical page that will receive the next token, so its immediate decode allocation is zero new slots even though its logical length still advances by one. If four resident requests use $P = 8$, and two of them are exactly divisible by 8 while two are mid-page, only the two boundary-crossers claim fresh pages; the next decode step needs $2P = 16$ slots.

Finished requests must be filtered before they can consume future decode capacity.

Algorithm 18: Maintaining a decode batch. Each live resident request contributes one new input row, one logical position advance, and one next-token KV write before the result-commit routine updates visible request state.

Input: Resident decode batch $\mathcal{B}$; previous output tokens; KV allocator; prefix cache; sampling state
Output: An executed generation batch ready for result commit

if *the next decode write cannot be reserved after prefix-cache eviction* **then**
  retract rows until next writes fit, or abort the last row if one request cannot fit;
set $\mathcal{B}$'s forward mode to decode;
clear prefill-only embedding and context-parallel metadata;
if *speculative decoding owns decode preparation* **then**
  delegate the rewrite and return;
advance sampling-penalty, input-token, stale-output, and encoder-decoder side state from the previous output tokens;
allocate one next-token KV write location for each resident request;
advance decode-batch positions, request KV lengths, and logical sequence lengths by one row-aligned position;
run the model worker and sampler on the prepared decode batch;
return *the executed batch and sampled-token payload for Algorithm 19*;

Speculative-decoding capsule.

Speculative decoding is a separate decode path in which candidate tokens may be drafted, verified, and accepted or rejected before the ordinary one-token resident batch is finalized. Chapter 5 only needs the ownership boundary: if a speculative path owns decode preparation, the ordinary decode-batch rewrite must hand off to that path instead of allocating and advancing rows as if each request accepted exactly one token. The detailed draft/verify policy appears later; the continuous-batching invariant here is that whichever path owns preparation must return a batch whose request order, KV writes, sampling state, and finish state still agree.

Concept-to-code map

- **One-token decode advance.** `ScheduleBatch.prepare_for_decode` in `python/sglang/srt/managers/schedule_batch.py` sets decode mode, clears stale prefill-time embedding fields, uses the previous output token as the next input, allocates one new cache location per request in the ordinary non-speculative path, increments request-level committed and allocated KV lengths, and advances sequence lengths. This is the source-level counterpart of

$$|z_i(t+1)| = |z_i(t)| + 1$$

  for each nonfinished decode-resident request that accepts one token.
- **Decode memory test.** The next-write feasibility check maps to `ScheduleBatch.check_decode_mem`, which computes required KV slots and asks the prefix cache to evict enough space if possible.
- **Resident-batch repair.** Dropping finished or retracted rows maps to `ScheduleBatch.filter_batch`; it preserves selected request indices and slices or rebuilds sequence lengths, request-pool indices, output-token entries, sampling metadata, and active speculative or encoder-decoder side state.
- **Scheduler handoff.** Scheduler result processing in `python/sglang/srt/managers/scheduler.py` calls the batch-filtering path before the next decode step can treat the resident batch as live work.

Algorithm 19: Committing a post-sampling result. The sampled token is not visible until request state, finish state, optional side records, streaming output, and batch membership have been advanced consistently.

Input: Executed generation batch $\mathcal{B}$; sampled token IDs; logits/logprob payloads; finish state
Output: Updated request objects, streamable outputs, and next resident batch

```
move sampled IDs and requested side tensors to host-visible records;
foreach request row i and request r_i in batch order do
    classify the row as skipped, chunk-logprob-only, or token-committing;
    if the row is skipped then
        continue;
    if the row is chunk-logprob-only then
        update chunk-local accounting, mark it nonstreamable, and continue;
    append the sampled token or accepted-token list to r_i.output_ids;
    update reasoning-token, customized-info, logprob, hidden-state, and grammar side records;
    check stop strings, stop tokens, length limits, aborts, and model-family finish rules;
    if r_i is now terminal then
        collect routed-expert metrics and release request-owned KV state;
    else if r_i just completed prefill outside decode then
        publish its prefix state for reuse;
emit streamable token or text deltas in request order;
filter terminal or retracted rows before the next decode admission decision;
return the surviving resident batch;
```

Algorithm 18 separates the recurring decode contract from the model execution itself. The scheduler first proves that the next writes fit, then rewrites the batch into decode mode, advances request and batch lengths together, and returns the sampled payload to Algorithm 19. In SGLang this sequence is spread across the decode-memory check, decode-batch preparation, result processing, and batch filtering.

The result-processing side is worth spelling out separately because it is where a sampled token becomes part of the request trace.

Algorithm 19 captures the other half of decode residency: after the model writes logits and the sampler chooses a token, result processing must append that token to the owning request trace. First, the sampled token becomes request state; then the same row order governs side records, finish decisions, streaming output, and filtering. In SGLang the prefill-result path, decode-result path, streaming-output path, and batch filtering step all participate in that same shape-preservation contract.

Decode maintenance is therefore a loop invariant: every live resident request advances by one logical position, every finished request leaves, and every tensor that remains in the batch still uses the same request order.

5.8. Merging, Mixed Work, and the Boundary with Chunked Prefill

Continuous batching becomes visible when new prompt work joins old decode work. After a prefill batch finishes, its requests are no longer merely waiting prompts. If they still need generation, they must join the resident decode set.

The merge operation is the moment where a one-step prefill batch becomes part of the continuing decode batch.

Mixed work shares one step ledger even when prefill chunks and decode rows have different costs.

For the running example, suppose r_{reuse1} is admitted first and finishes its prefill step. Before the merge, $B_t = (\text{pan})$ and $\Lambda_t = (\text{reuse1})$. After the merge,

$$B_{t+1} = (\text{pan}, \text{reuse1}), \quad \Lambda_{t+1} = \varnothing.$$

r_{pan} stays resident across the boundary. r_{reuse1} is the newly merged member because its prefill result becomes decode-resident work. The next decode step uses one token for r_{pan} and one token for r_{reuse1}. Nothing in this transition says that the two requests arrived together or have the same total length. They are co-resident because the scheduler found a feasible step at this point in time.

In SGLang, this ownership transfer is a scheduler-and-batch handoff. The scheduler filters the previous extend batch and merges it into `running_batch` when appropriate; the batch helper then concatenates request lists and per-request metadata after synchronizing any pending verification result that could affect sequence lengths. SGLang places the scheduler handoff in the scheduler (`scheduler.py`), at `Scheduler.get_next_batch_to_run`. The batch merge helper is in `schedule_batch.py`, at `ScheduleBatch.merge_batch`. This is a source-level example of Definition 5.1.1: the batch changes identity while each request keeps its logical trace.

`MIXED` mode is the nearby but distinct case. In the broad continuous batching sense, a mixed step packs prefill work and decode work into one forward step. It is still a model forward step in the Chapter 1 sense (Definition 1.4.1) and the Chapter 3 mixed-serving-step account (Definition 3.2.5) specialized to a continuous-batching scheduler decision. Chunked prefill later specializes that idea by deciding how much prefill work can share the step with decode rows. The implementation method `ScheduleBatch.mix_with_running` appends the running decode requests to a new source-level extend batch, adjusts prefix and extend lengths for the decode portion, and records which requests were decoding. The result is a single forward batch whose token tensor contains both prefill tokens and one-token decode inputs.

Only the handoff matters here. Chunked prefill decides how a long prompt is split and how decode can piggyback on the chunks; the SARATHI line of work is one source for that algorithmic tradeoff [1]. The continuous-batching rule that makes mixing legal is local to each request: new positions, prefix length, output cache locations, and sampling metadata must still describe the same logical sequence.

Table 5.7 separates the continuous-batching decision from the later chunked-prefill policy that chooses how much suffix work joins a mixed step.

The boundary is worth naming because it prevents a common misunderstanding. Continuous batching does not require every step to be mixed. A scheduler can run pure prefill steps, pure decode steps, or mixed steps. What makes the system continuous is that the resident request set is updated between steps instead of being fixed for the lifetime of a static batch.

Table 5.7.: Continuous batching to chunked-prefill boundary. Continuous batching owns the legality of changing the resident request set between steps; chunked prefill owns the later choice of how long-prompt suffix work is split.

Boundary item	Continuous-batching owner	Neighboring owner
Resident request set	Admits, merges, retracts, and filters requests between steps.	Chunked prefill later decides how much suffix work a long prompt contributes to one mixed step.
Mixed forward metadata	Preserves row order, positions, prefix lengths, output cache locations, and sampling metadata for each logical request.	The forward-batch and attention paths consume the mixed metadata; they do not decide which requests became co-resident.
Invalid confusion	Treating every mixed step as a chunked-prefill policy decision.	Treating chunk size or piggybacking policy as proof that the resident decode set is static.

5.9. Retraction Under Memory Pressure

Admission uses estimates. A request may generate more tokens than expected before it stops, cache eviction may fail to recover enough contiguous capacity, or a conservative reserve may still be too optimistic near a page boundary. Continuous batching therefore needs a recovery transition for decode memory pressure.

Retraction repairs scheduler residency while preserving generated-text semantics.

Definition 5.9.1 (Continuous-batching retraction) *The opening serving-problem chapter introduced decode retraction as memory recovery. In the continuous-batching loop, the retraction transition is the particular batch rewrite that filters selected rows out of the resident decode set, hands their reusable KV ranges back to memory management, rebuilds the surviving batch metadata, and requeues the affected requests with enough state to reconstruct their next legal input. If B_t is the resident decode batch and $\mathcal{D}_t^{\text{ret}} \subseteq B_t$ is the retracted subset, the rewritten batch is $B'_t = B_t \setminus \mathcal{D}_t^{\text{ret}}$.*

Retraction is the last stage of a reserve-miss lifecycle:

The same decode-maintenance concept-to-code map continues here, but the ownership story is short: the scheduler checks decode memory before preparing the next decode step. If `check_decode_mem` fails, the batch retraction helper chooses requests to retract, releases their resources through `release_req`, filters the batch to the remaining indices, updates the new-token-ratio estimate, and requeues retracted requests through the scheduler path. SGLang places the decode-memory check in the scheduler (`scheduler.py`), at `Scheduler.update_running_batch`. The batch retraction helper is in `schedule_batch.py`, at `ScheduleBatch.retract_decode`.

Algorithm 20 makes the recovery ordering in continuous-batching retraction (Definition 5.9.1) explicit: memory is released before the active batch is filtered, and abort is reserved for the case where even the final survivor cannot continue. The listing is therefore about scheduler state repair before any change to already streamed tokens.

Stage	What changes	Reader checkpoint
Admission estimate	ρ_r and page overhead let a request enter with a predicted continuation reserve.	The estimate is a policy guess, not a guarantee that future decode pages will exist.
Decode-memory check	Page-boundary requests ask for their next KV page.	`check_decode_mem` tests the physical page account before launch.
Eviction attempt	Unprotected prefix-cache pages are reclaimed if possible.	No live decode row is removed yet.
Retraction	One or more resident rows are filtered, released, reset, and requeued.	Already visible output remains, but decode residency is lost.
Recalibration or abort	The new-token ratio is adjusted; the final survivor is aborted only if even one row cannot fit.	Recovery stays nonterminal until there is no legal resident batch left.

Table 5.8.: Reserve-miss lifecycle for retraction. Each stage changes runtime state and gives the reader a checkpoint.

Example 5.9.1 (Retracting one page-boundary row) Consider a simplified decode batch with r_{pan}, r_{reuse1}, and r_{reuse2}, page size $P = 4$, and only four free KV slots. If r_{pan} and r_{reuse1} are both at page boundaries and r_{reuse2} is mid-page, the next decode step needs $2P = 8$ slots. Eviction cannot supply the missing four slots. The scheduler can retract one request at a page boundary, reducing the immediate next-step need to four slots. The retracted request loses its current decode residency (Definition 5.1.2), but its client-visible request identity and already emitted output are not erased. Figure 5.11 draws the retracted row, released page, and survivor decode batch.

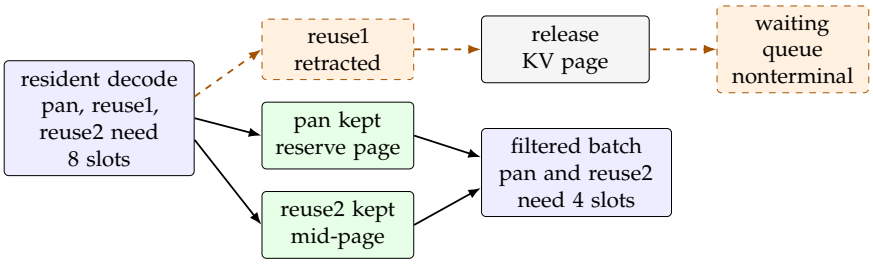

Figure 5.11.: Retraction repairs decode memory pressure by filtering r_{reuse1} out of the resident batch, releasing its KV page, and returning the nonterminal request to the waiting queue. The solid survivor rows form the next decode batch; the dashed path preserves request identity without keeping decode residency.

The reset matters. `Req.reset_for_retract` clears prefix indices, prefix-cache node references, extend length, batch indices, allocated and committed KV lengths, and several side-state fields. A prefix-cache node reference is a handle to reusable-prefix metadata, not a KV slot; Chapter 8 defines the radix-node structure. The request is marked as retracted so later scheduling can reconstruct its next-round input and find whatever prefix state remains valid. If the request used input embeddings in a way that cannot be safely recombined with generated output IDs, SGLang discards generated output IDs for shape-consistency reasons. That special case is an example of source fidelity: Definition 5.9.1 says "may continue," while the implementation must respect representation constraints.

The distinction from that serving-level account is the batch surgery. A serving-level account can say that memory pressure moves a request out of decode residency; the continuous-batching account must also say which per-row tensors,

Algorithm 20: Decode-memory retraction transition. Retraction removes rows from the resident decode batch while preserving the distinction between waiting requests and terminal aborts.

Input: Resident decode batch $\mathcal{L}_{\text{res}}$; KV allocator state; retraction policy
Output: Surviving decode batch, retracted requests, optional aborts, and a new continuation estimate

order candidate rows according to the current retraction policy;
initialize $R \leftarrow \emptyset$ and $A \leftarrow \emptyset$;
while *decode memory is insufficient and more than one candidate remains* **do**
    choose one candidate row j to remove from $\mathcal{L}_{\text{res}}$;
    release j's KV resources without inserting them into the prefix tree;
    reset request-local scheduling fields for future reconstruction;
    append request j to R;
if *the last surviving request still cannot fit* **then**
    mark that request aborted by out-of-memory recovery;
    release its resources and append it to A;
filter every batch tensor to the surviving row order;
recompute the continuation estimate from the surviving requests;
return $\mathcal{L}_{\text{res}}$, R, A, *and the estimate*;

prefix references, ratio estimates, and queue entries change before the next forward step can be formed.

Retraction differs from preemption by priority, although both remove work from the running set. Priority preemption is a policy decision to serve a more important request. Decode retraction is a memory recovery decision. Both transitions rely on the same deeper invariant: a removed request must not leave stale metadata inside the active batch.

Retraction also differs from abort. Abort creates a terminal state and sends an abort result. Retraction creates a nonterminal scheduling state. A reader tracing SGLang should ask a simple question at every removal point: did this request become terminal, or did it become waiting work again?

5.10. Throughput, Latency, and Fairness in Step Selection

In continuous batching, throughput, latency, and fairness are consequences of the same repeated decision: at scheduler step t, should the system admit waiting prefill work, protect resident decode work, exploit prefix reuse, or pause because memory and policy constraints bind? A policy that always fills the largest possible prefill batch can delay first tokens for short requests. A policy that always protects decode latency can leave prompt work waiting. A policy that always maximizes prefix reuse can reorder the queue enough to create fairness problems.

Fairness claims need the same unit of waiting, service, and admitted token work.

Table 5.9 explains why queue wait is not a metric attached after the scheduler decision. When continuous batching keeps resident decode work moving while fresh prompts wait, T_{sys} grows for the waiting class even if the accelerator remains busy. Little's law (Definition 1.2.1) gives useful vocabulary: for a stable system, average in-flight concurrency is related to arrival rate and average time in system by

$$\bar{N}_{\text{flight}} = \lambda T_{\text{sys}}$$

Scheduler choice	Throughput effect	Latency effect	Fairness risk
Admit a large prefill batch	More dense prompt work per launch	Resident decode tokens may wait	Short prompts can wait behind long prompt work.
Protect resident decode	Stable inter-token cadence	Waiting prompts see higher TTFT	New arrivals can starve under sustained decode load.
Cluster by prefix reuse	Less recomputed prompt work	Provider/follower ordering can raise TTFT	Hot prefix families can dominate admission.
Delay for memory reserve	Avoid invalid KV writes or retraction churn	Queue wait grows	Requests near page boundaries may be favored or penalized.

Table 5.9.: Continuous-batching policy pressures at one scheduler step.

[24]. If the arrival rate λ is fixed and average waiting plus service time T_{sys} grows, more requests are present in the system. This queueing vocabulary is a coarse model of SGLang: GPU steps are not identical service quanta and KV memory creates admission constraints, but it explains why queue wait cannot be ignored when comparing continuous batching policies.

Definition 5.10.1 (Constrained work conservation) *A scheduler is work-conserving relative to constraints when every idle scheduler state is justified by the absence of runnable forward work after policy, memory, and synchronization constraints are applied. This is an idleness verdict, not a command to keep the accelerator busy at any cost: priority policy, prefill delay, memory reserve, or distributed synchronization may make a locally idle device a valid decision. In notation, if*

$$\mathcal{U}_{\text{prefill}}(S_t) \cup \mathcal{U}_{\text{decode}}(S_t) \neq \emptyset,$$

then a constrained-work-conserving scheduler must choose some runnable u_t from that union unless an explicit constraint removes it; if the union is empty after those constraints are applied, idleness is not by itself a scheduler bug.

Example 5.10.1 (Constrained idleness) Suppose the waiting queue has one prompt r_{fresh}, and the decode batch has one resident request r_{pan}. If r_{fresh}'s prefill allocation fits and r_{pan}'s next decode allocation fits, an idle accelerator violates constrained work conservation. If r_{fresh} is delayed by a configured prefill policy and r_{pan}'s next KV page cannot be reserved until eviction or retraction finishes, the same local idleness is a valid constrained state. The definition separates avoidable idleness from idleness forced by policy or resource constraints. Figure 5.12 draws that verdict as a decision over feasible prefill and decode work.

Example 5.10.2 (Binding memory constraint) At a later scheduler step, suppose only 4 KV slots remain free after protected decode state is accounted for. Two waiting candidates are otherwise eligible: r_{reuse1} needs 3 new suffix slots, while r_{reuse2} needs 6. With page size $P = 4$, the feasibility charges are

$$\text{alloc}_P(3) = 4, \qquad \text{alloc}_P(6) = 8.$$

Memory therefore removes r_{reuse2} from the runnable prefill set but leaves r_{reuse1} runnable. If no policy or synchronization rule forbids the smaller request, idling violates constrained work conservation: the legal step is EXTEND(r_{reuse1}), or a decode step if a resident decode row is also runnable. If an external priority rule temporarily permits only r_{reuse2} and the resident decode row is waiting on a synchronization barrier, then the same 4-slot state can be validly idle. The binding constraint is visible because the failed 6-slot candidate is not the same as an empty runnable set.

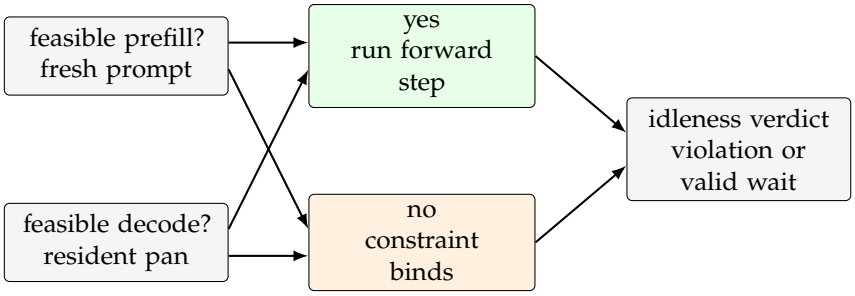

Figure 5.12.: Constrained work conservation classifies idle time by the legal work available at that scheduler state. If feasible prefill or decode work exists, idleness is suspicious; if policy, memory, or synchronization blocks all work, idleness is part of the constrained policy.

Constrained work conservation (Definition 5.10.1) defines when idleness is a violation rather than a policy choice. The throughput side prefers large, efficient steps. A larger prefill batch can increase dense-token work per launch and improve hardware utilization. A larger decode batch can amortize scheduling overhead across more requests. Cache-aware policies can reduce recomputation, converting raw prompt tokens into shorter prefill suffixes, the prompt tails that still need execution after prefix reuse [49]. Each improvement is still a scheduler choice about which runnable work moves now and which runnable work waits.

The latency side splits into several quantities. Time to first token includes queue wait, prompt admission, prompt execution, any prefix-cache load, and sampling before the first visible output. Inter-token latency is dominated by the cadence of later decode steps. A policy can improve one while worsening the other. For example, admitting r_{reuse2}'s six-token prefill suffix before r_{pan}'s next decode might improve r_{reuse2}'s first-token time but increase r_{pan}'s inter-token gap.

Fairness is the third side, but it is also a step-selection property. Suppose W_t contains two fresh requests, two reuse candidates from the same prefix family, and one low-priority long prompt, while B_t contains r_{pan}. FCFS protects arrival order, longest-prefix matching may cluster the reuse candidates, priority scheduling may protect r_{pan}'s decode cadence, and routing-key affinity may keep one application family local. None of these policies is simply fair; each defines which request class is allowed to wait at this step and which class gets the next legal forward pass.

The definitions make the tradeoffs explicit. Admission feasibility (Definition 5.4.1) states which steps are legal. Cache-aware priority (Definition 5.5.1) states which

legal step is preferred when reuse is part of the policy. Identity agreement (Definition 5.6.1) and decode residency (Definition 5.1.2) state what must remain true after that choice. The measurement section later in this chapter should therefore attach throughput and latency scores to scheduler choices: a row is continuous-batching evidence only when it says which waiting, resident, cached, or memory-constrained work the scheduler favored.

5.11. Correctness Invariants

The scheduler is allowed to reorder, merge, filter, and retract work only because it preserves request-level invariants. These invariants are stronger than "the code did not crash." They say what must be true for a continuously changing batch to represent the same logical requests.

Batch membership may change only while request identity stays fixed.

Invariant 5.11.1 (One scheduler location)**.** *Every nonterminal request is owned by exactly one active scheduler location*: *waiting, resident decode, current prefill work, chunked continuation, or a well-defined transfer/staging state.*

Invariant 5.11.2 (Logical sequence)**.** *At next-round input construction, a normal text request's logical token sequence remains* $z_i = x_i \| y_i$*, the original prompt followed by generated output, unless a source-level exception is handled explicitly.*

Invariant 5.11.3 (Cache validity)**.** *Prefix cache indices used by a request must correspond to a valid prefix of the same logical token sequence and compatible cache key. A cache hit for another prompt, Low-Rank Adaptation (LoRA) key, multimodal embedding override, or incompatible position state is not valid for this request.*

Invariant 5.11.4 (Batch metadata order)**.** *All per-request metadata in a batch uses the same request order, so filtering, merging, and retraction must update or rebuild every dependent metadata object before the next forward step.*

Example 5.11.1 (What the bug looks like: batch metadata order) Assume a decode batch is ordered (A, B). The scheduler has `seq_lens` $= [65, 4097]$. Its request-pool rows are `req_pool_indices` $= [p_A, p_B]$. Its KV-write vector is `out_cache_loc` $= [s_A, s_B]$, where s_A and s_B are the next KV write slots for A and B. A filtering or merge path then rebuilds only `out_cache_loc` as $[s_B, s_A]$. The length vector, request-pool rows, and sampling metadata remain in (A, B) order.
Nothing has to crash. The tensor shapes still agree, and the model can return plausible-looking tokens. The bug is semantic: B's new KV row is written into A's slot, the next decode step reads a mixed request history, and later output or logprob metadata can remain attached to the wrong request. This is why Invariant 5.11.4 is an order invariant as well as a shape invariant.

Invariant 5.11.5 (Decode memory reservation)**.** *A decode step may launch only after the scheduler has reserved enough KV capacity for the next write locations required by that step, including page effects and any active speculative or hybrid-cache reserve.*

Table 5.10.: Continuous-batching invariants, violations, and source evidence.

Invariant	Example violation	Source evidence
One scheduler location	A request remains in W_t after it has also been merged into B_t, so the scheduler can admit or decode the same logical request twice.	Scheduler and scheduler batch state.
Logical sequence	Retraction rebuilds the next input from the prompt alone and drops generated output, changing the resumed request's meaning even though its identity is unchanged.	`Req.init_next_round_input`.
Cache validity	A prefix-cache node from another cache key is accepted as this request's hit, so KV state represents the wrong logical prefix.	Schedule policy and memory-cache state.
Batch metadata order	`filter_batch` drops one token row but leaves sampling metadata in the old order, allowing one request to receive another request's sampler or length state.	`ScheduleBatch.filter_batch`; `merge_batch`.
Decode memory reservation	A page-boundary decode step launches without reserving the next KV page, so the write can overrun allocator capacity or force late recovery.	`check_decode_mem`; `prepare_for_decode`.
Terminal filtering	A finished request remains in the running batch after its terminal state is reported and keeps consuming decode slots or emitting late output.	`Req.check_finished`; `filter_batch`.

Invariant 5.11.6 (Terminal filtering)**.** *A request that has reached a terminal finish state may report terminal output and diagnostics, but the active batch must eventually exclude it from future decode steps.*

Table 5.10 turns Invariant 5.11.1, Invariant 5.11.2, and Invariant 5.11.3 into review failures and source-reading anchors. It also applies Invariant 5.11.4, Invariant 5.11.5, and Invariant 5.11.6. These are the ordinary ways a mutable batch can look plausible while representing the wrong request state.

The table is a checklist for code reading. `filter_batch` is where batch metadata order is most visibly at risk. `merge_batch` is where newly extended requests become resident alongside older decode requests. `check_decode_mem` and `retract_decode` protect decode memory reservation. `init_next_round_input` and prefix matching protect logical sequence and cache validity. Runtime request identity is the distributed version of the same idea: every local transition must preserve the request trace.

5.12. Continuous-Batching Measurements

The measurements test three continuous-batching claims. First, batch membership changes over time rather than remaining a static request set. Second, scheduling-policy effects depend on workload shape. Third, public latency and throughput rows do not by themselves prove cache reuse, page slack, priority fairness, or retraction behavior.

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

The rows reuse the Chapter 2 Thor/Qwen measurement slice. Source-level SGLang claims in this edition are anchored to the recorded source metadata; the Thor benchmark rows identify the measured runtime by NVIDIA's container package. Unless stated otherwise, result numbers are arithmetic means across completed public benchmark rows. The direct scheduler traces are single-host instrumented runs that use the recorded SGLang source metadata plus optional scheduler-event counters, and they are reported as event evidence rather than folded into the completed public-metric means.

Table 5.11 anchors the measured rows before the chapter turns them into claims.

The RTX 5090 transfer run repeats the Chapter 5 prefix-policy pair on a single desktop Blackwell GPU while keeping the same SGLang source revision and Qwen2.5-0.5B model. Table 5.12 is therefore a hardware-transfer check, not a larger-model scale law. It preserves the direction of the Thor row: `fcfs` has lower median TTFT and higher throughput than `lpm`, but the gap is much smaller on the 5090 row. The same boundary still applies: the row has cache-report counters from the server metrics surface, but not a per-request cache-hit trace.

Figure 5.13 turns the completed arrival-rate trace into the missing picture: the running request count rises and falls as the scheduler admits, advances, and finishes requests, while the queued count shows short admission stalls. The plotted trace includes warmup and readiness requests in addition to the 96 benchmark prompts, which is why the trace parser observes 102 distinct request hashes. Detailed event counts and incomplete pressure diagnostics remain in the experiment coverage record.

Figure 5.14 puts the completed non-smoke benchmark rows from the canonical all-suite run on a common visual surface. Range bars show completed measurement

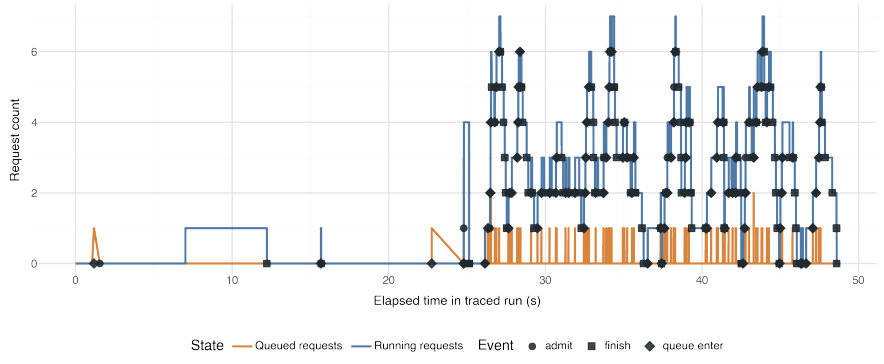

Figure 5.13.: Single-host scheduler event trace for `ch5_arrival_rate_4`. The blue line is running batch size, the orange line is queued request count, and markers show queue entry, admission, and finish events. The figure directly observes batch membership changing between scheduler steps; it does not show retraction behavior because no retraction event occurred in this completed trace.

Table 5.11.: Results at a glance for the Chapter 5 Thor measurements. Values reuse the shared Thor/Qwen slice and summarize completed public benchmark rows; instrumented traces are interpreted separately.

Experiment	Canonical configuration	Metrics	Result and boundary
Harness smoke validation	Short random prompts, completed-device collection	TTFT, ITL, output tok/s, E2E latency	Default smoke completed on all devices: 133 output tok/s, 162 ms TTFT, 8.03 ms ITL, 222 ms E2E.
Prefix reuse policy comparison	Model, page size, schedule policy, shared-prefix prompt family, arrival pattern	TTFT, output tok/s, total tok/s	`fcfs`: 392 ms TTFT, 3,763 output tok/s (39,635 total tok/s). `lpm`: 6,356 ms TTFT, 577 output tok/s (6,081 total tok/s). This row has no cache-hit counter.
Arrival-rate sweep	Model, hardware, request length distribution, offered load	throughput, TTFT, E2E latency	1 req/s: 561 total tok/s, 664 ms median E2E. 4 req/s: 2,160 total tok/s, 596 ms median E2E. Queue wait for the public row is not directly reported; the single-host trace below reports it separately.
KV-pressure contrast	KV budget, page size, max new tokens, prompt lengths	throughput, TTFT, E2E latency	Tight KV budget: 36,880 total tok/s, 4,818 ms TTFT. Roomier budget: 40,432 total tok/s, 3,946 ms TTFT. The completed public rows do not report retraction counts.
Page-size sensitivity	Page size or allocator setting, repeated workload	throughput, TTFT, E2E latency	Page size 1: 30,802 total tok/s, 2,168 ms E2E. Page size 16: 20,825 total tok/s, 3,186 ms E2E. Page slack is not directly reported.
Priority scheduling sanity point	Uniform request priority at 4 req/s with priority scheduling enabled	throughput, TTFT, E2E latency	Uniform-priority run completed on all devices: 2,179 total tok/s, 33 ms TTFT, 494 ms E2E. This row is a compatibility check for the priority path; the class-mix evidence comes from the scheduler trace below.

Table 5.12.: RTX 5090 transfer check for the Chapter 5 prefix-policy pair. The single-device row uses SGLang `612785ffdcaf`, the Qwen2.5-0.5B-Instruct model, 64 generated-shared-prefix prompts, and infinite request rate.

Policy	Total tok/s	Output tok/s	TTFT ms	E2E ms	Boundary
`lpm`	101,766	9,647	106.7	414.1	One RTX 5090 row; prefix reuse policy selected, no per-request hit trace.
`fcfs`	114,964	10,898	83.1	367.1	Same model, source revision, and generated-shared-prefix scenario.

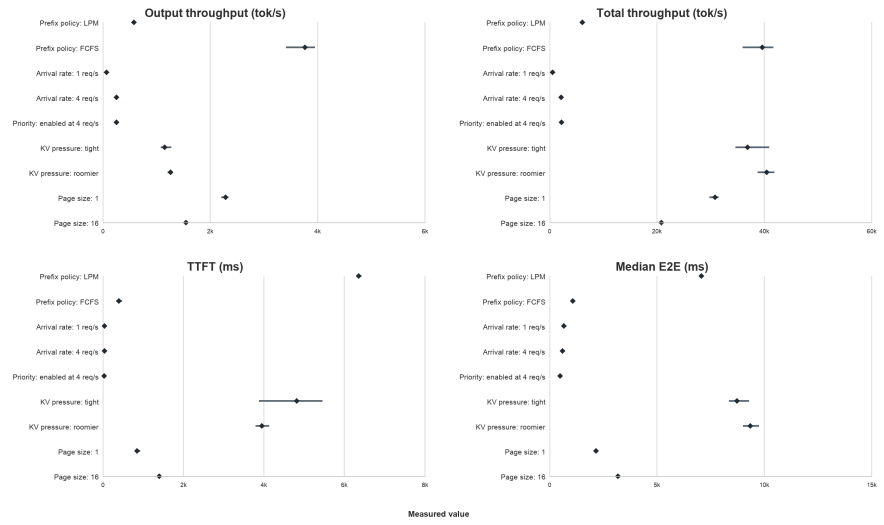

Figure 5.14.: Chapter 5 Thor metric contrasts. Horizontal bars show completed-row ranges, and black diamonds are means across completed rows. The figure uses only public benchmark metrics; direct scheduler event evidence is reported separately in Figure 5.13.

spread, and black diamonds mark arithmetic means. The plot separates output-token and total-token throughput so the prefix-policy rows are not folded into one throughput definition.

The split between the two figures is deliberate. The canonical public benchmark rows still report latency and throughput through the public benchmark surface. The traced canaries add scheduler-stage evidence for batch membership, queue entry, admission, finish, and priority labels on one host. Cache-hit tokens, page-aligned slot slack, starvation windows, and completed retraction frequency remain outside the successful trace evidence. The uniform-priority public row should therefore not be read as a priority-ordering contrast; the mixed-priority trace is the row that supports the priority-class queue-wait claim.

The prefix-policy row is the important cross-chapter contrast. Chapter 2's scheduler-policy A/B used a larger generated shared-prefix workload: 128 prompts, 16 prefix groups, 8 prompts per group, and 1024-token shared system prompts. This chapter's prefix-policy row uses 64 prompts, 8 groups, 8 prompts per group, and the shorter Chapter 5 generated-prefix shape from the Volume I scenario catalog. The reversed ordering is therefore a workload-shape result: the same policy pair can change sign when the sharing geometry, prompt volume, and queue shape change. The row proves a policy-conditioned latency and throughput contrast; it does not by itself prove reduced prefill work without cache-hit or reused-token counters.

Figure 5.15 sketches the ordering mechanism that is consistent with the measured TTFT gap. Longest-prefix-match clusters the eight prompts of a prefix group so they are admitted back to back. Because the group's first prompt must run before any sibling can reuse its prefix, every later sibling waits behind that provider, and on this small bursty queue the reorder pushes group-tail first tokens far out in time. First-come ordering interleaves prompts from different groups, so a first token is produced sooner even though it does less reuse. The figure visualizes only the queue-ordering effect on TTFT; it does not assert a cache-hit count, which the public row does not carry. This is Algorithm 14's two-tree lesson in measurement form: prefix sorting clusters siblings, but sibling reuse is unavailable until the provider prompt has actually run.

This is the point of the surprising 6,356 ms versus 392 ms row. LPM is not inherently a lower-latency policy; it is a reuse-seeking ordering policy. On a small burst where each prefix family has one provider and seven potential followers, clustering can delay the first visible token for the tail of a family before reuse has paid back the queueing cost. The measurement therefore teaches a scheduler lesson rather than a cache-speed claim: reuse-aware ordering needs fairness and arrival-shape checks, or it can improve locality while making reader-visible TTFT worse.

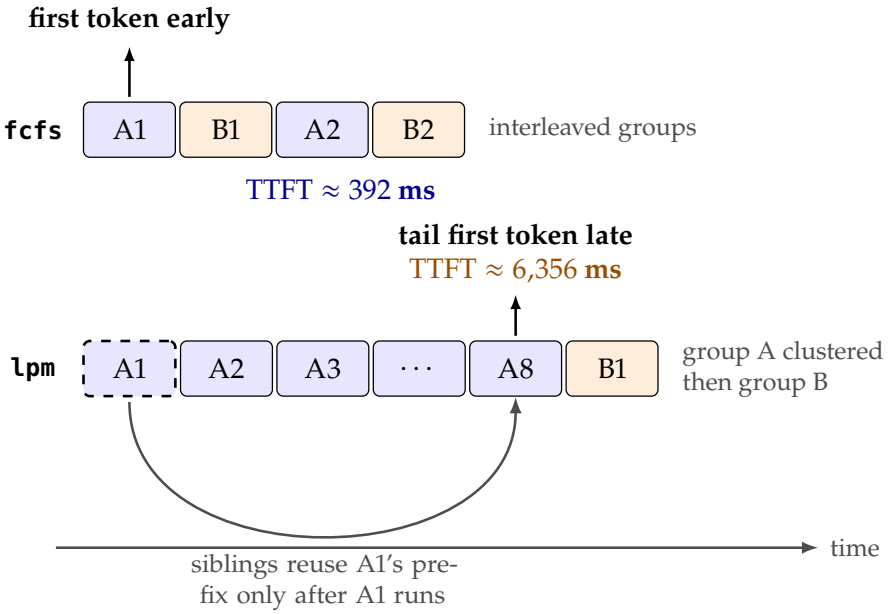

Figure 5.15.: Why longest-prefix-match raised TTFT on this workload. First-come ordering (top) interleaves prompts from different prefix groups, so an early first token appears (measured TTFT ≈ 392 ms). Longest-prefix-match (bottom) clusters a prefix group's eight prompts so siblings can reuse the group's prefix, but each sibling can only reuse after the provider A1 runs (dashed); on this small bursty queue the clustering pushes group-tail first tokens far out, consistent with the measured TTFT ≈ 6,356 ms. The figure shows the ordering effect only and makes no cache-hit-count claim.

The completed traces support two direct claims: batch membership changes over time, and priority labels can be joined to queue waits for the traced load mix. They do not yet support claims about cache-hit tokens, page-aligned slot slack, starvation windows, or completed nonzero retraction evidence. The public prefix-policy row should therefore be read as a policy-conditioned latency and throughput contrast; a reduced-prefill-work claim would require cache-hit or reused-token counters.

5.13. Summary

Continuous batching converts an online request stream into legal model forward steps while preserving each request's token sequence, cache state, memory reservation, and output semantics. Read the chapter as one scheduler round that has to rule out three failures.

- Membership changes, but request identity must not. Request state (Definition 5.2.1) and scheduler state (Definition 5.3.2) keep waiting work, resident decode work, previous prefill work, chunked continuation, memory/cache state, and finish state in separate locations.

- Admission can fit token counts while still failing physical capacity. Admission feasibility (Definition 5.4.1) is page-aligned and predictive because prompt-token demand, future decode reserve, and KV page geometry all matter before a batch reaches the model.
- Retraction can recover memory while still breaking a live trace. Decode residency (Definition 5.1.2), identity agreement (Definition 5.6.1), and retraction (Definition 5.9.1) keep row order, sampling state, and KV ownership attached when work is filtered or removed.

Those failures explain the scheduler history. Continuous batching could not remain a pure throughput loop once requests could share prefixes, consume page-sized KV capacity, overlap with output repair, or arrive with priority and preemption policy. Queue order, cache benefit, future decode slots, active latency, and resident-state ownership all became part of the next executable batch. Constrained work conservation (Definition 5.10.1) is the throughput-side verdict: idleness is a scheduling loss only after feasible prefill, feasible decode, and explicit policy constraints have been accounted for.

Table 5.10 is the failure-mode checklist for those failures. The most important question is not whether every tensor has the right shape. It is whether a request sits in exactly one scheduler location and keeps its logical sequence, cache evidence, and row-ordered metadata aligned. It must also reserve decode memory before that memory is needed and leave the batch when terminal filtering says it is done. Batch metadata order and decode memory reservation are the easiest failures to miss, because a batch can still look numerically plausible while carrying another request's sampler state or writing beyond reserved KV capacity.

The next chapter treats chunked prefill as mixed work: long prompt suffixes can overlap decode while the scheduler protects first-token latency and resident generation.

5.14. Exercises

1. Given three requests with prompt lengths 7, 12, and 5, cached prefix lengths 3, 8, and 0, and page size $P = 4$, compute each request's effective prefill length and page-aligned prefill allocation.
2. In the running example, suppose each admitted request reserves four future decode slots plus one page of overhead. With total simplified budget 20, which subsets of r_{fresh}, r_{reuse1}, and r_{reuse2} are feasible if r_{pan}'s next decode page has already been protected?
3. Construct a scheduler-state tuple

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

   for the running example after r_{reuse1} and r_{reuse2} have finished a prefill step but before they are merged into the decode batch.
4. Explain why continuous batching does not require batch membership to be stable. Which invariant replaces stable batch membership?
5. Given waiting requests with cached prefix lengths

$$[0, 12, 4, 12],$$

show the order produced by a longest-prefix policy. Then describe one fairness problem this order can create.

6. A decode batch has four resident requests and page size $P = 8$. Two requests have `kv_committed_len` exactly divisible by 8, while the other two are mid-page. In the ordinary non-speculative case, how many new KV slots must be available before the next decode step?
7. Describe a concrete metadata-shape-consistency bug involving `seq_lens`, `req_pool_indices`, and sampling metadata. What observable error could it cause?
8. Retraction removes a request from the decode batch but does not necessarily abort it. Give a state-transition explanation of how the request can later continue correctly.
9. Use Little's law informally to explain why increasing average scheduler queue wait increases the number of requests present in the system at a fixed arrival rate. Why is this not by itself a complete performance model for SGLang?
10. Pick one SGLang source function from the implementation anchors and write a five-step behavioral trace that explains what state changes, what invariant is protected, and which later chapter covers the details deferred here.

Chunked Prefill and Overlap Scheduling

6.

Chapter Abstract. A ten-thousand-token prompt should not freeze three active chats. Chunked prefill splits the uncached suffix so the scheduler can admit a bounded piece, run resident decode work between pieces, and still finish the long prompt. Overlap scheduling then moves scheduler and result-processing work off the critical path without letting any request observe unfinished state. This chapter follows one long prefill request alongside three resident chats. It traces suffix admission, mixed prefill/decode execution, and future-token resolution. Guardrails and policy tuning explain when the overlap is allowed; measurements show what the policy changes.

6.1. The Long-Prefill Scheduling Problem

A ten-thousand-token prompt enters while three resident chats are waiting for their next token. The prompt is legitimate work, but an indivisible prefill can make every small decode step wait behind it. This chapter turns one blocking prefill into schedulable pieces. The durable idea is that a long prompt must compete in the same online decision as resident decode tokens; it should not become a private lane that stops the rest of the batch from moving.

Continuous batching solves the problem of mutable batch membership, but it does not by itself bound the size of a single prompt computation. A request with a ten-thousand-token uncached suffix can still occupy one prefill step for much longer than the one-token decode steps of resident requests. This creates a serving problem rather than merely a kernel problem: already-admitted requests care about time per output token, while a waiting long prompt cares about time to first token, and both objectives compete for the same model worker.

Chunking turns one long prompt into schedulable units of work.

The asymmetry follows from the cost model in earlier chapters. Prefill work (Definition 2.3.1) can schedule thousands of new token rows, and each row attends to its preceding context. Decode work (Definition 2.3.2) usually schedules one new row per resident request. If the scheduler treats a long prefill suffix as indivisible, the resident decode batch waits until the whole suffix finishes. Section 6.2 formalizes that suffix as a step-indexed object (Definition 6.2.1). Iteration-level scheduling systems such as Orca made the batch boundary dynamic [48]; chunked prefill makes one large prefill item itself interruptible. SARATHI frames the same pressure as an opportunity to piggyback decode work with bounded prefill chunks [1].

Definition 6.1.1 (Long-prefill interference) *Long-prefill interference occurs when a prefill step for a waiting request is large enough to delay decode progress for already resident requests beyond the latency target implied by the serving policy. The interference is measured at the scheduler level: it is the time during which feasible decode work exists but cannot run because an indivisible prefill step is occupying the model worker. If T_{prefill} is the indivisible prefill duration and $T^{\max}_{\text{decode}}$ is the policy's decode-delay target, interference is present when feasible decode work waits through $T_{\text{prefill}} > T^{\max}_{\text{decode}}$.*