Chunked Prefill and Overlap Scheduling

Deep Dive into SGLang — Chapter 6

Yin Yang

Foundation Books Project

Where this chapter sits

Continuous batching (Chapter 5) made batch membership mutable. It did *not* bound the size of a single prompt computation.

- A ten-thousand-token prompt can still freeze three live chats behind one indivisible prefill step.
- This chapter turns one blocking prefill into schedulable pieces that compete in the same online decision as decode tokens.
- Then overlap scheduling moves scheduler and result work off the critical path — without letting any request observe unfinished state.

Goal: a long prompt makes progress while resident decode keeps moving, and nothing decodes against KV state that was never written.

Roadmap

The problem and the suffix object

Mixed steps and the metadata contract

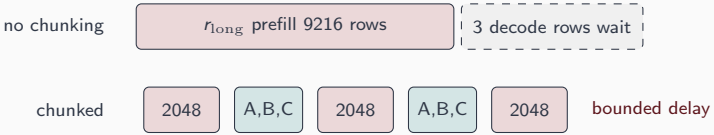
Overlap and future tokens

Tuning, guardrails, evidence

The problem and the suffix object

One indivisible prefill delays everyone behind it

The running example: r_{long} has 10,240 prompt tokens, 1,024 already cached — so 9,216 need prefill.



Long-prefill interference is a scheduler-level delay, not a kernel bug: feasible decode work exists but cannot run. Chunking keeps the same total prompt work, but gives decode rows opportunities between pieces.

The prefill suffix: the only thing we actually run

Definition (Prefill suffix)
For request r with logical sequence $z_r(t) = x_r \| y_r(t)$ and committed KV-prefix length $p_r(t)$,

$$u_r(t) = z_r(t)[\,p_r(t) :\,], \qquad e_r(t) = |u_r(t)|.$$

- Chunking starts *after* prefix reuse: $p_{r_{\text{long}}}(0) = 1024$, so $e_{r_{\text{long}}}(0) = 10{,}240 - 1{,}024 = 9{,}216$.
- A chunk is a contiguous prefix of that suffix: $c_r(t) = u_r(t)[0 : m_t]$, with $0 < m_t \leq \min(e_r(t), \kappa_t)$.

We never re-run cached context — only the uncomputed suffix, cut to a budget κ_t.

Chunk progress: advance the prefix exactly by what you wrote

Chunk progress invariant

$$p_r(t+1) = p_r(t) + |c_r(t)|, \qquad u_r(t+1) = z_r(t+1)[\,p_r(t+1) :].$$

With $\kappa_t = 2048$, r_{long}'s suffix splits as

$$2048,\ 2048,\ 2048,\ 2048,\ 1024.$$

- Advance *too far* (as if all 9216 were done): the request decodes against KV state that was never written.
- Advance *not at all*: the same suffix is admitted twice — double allocation.

Progress equals produced KV state. That single rule rules out both errors.

Mixed steps and the metadata contract

Decode rides along: a mixed step is one extend-style batch

Definition (Chunked-prefill mixed step)
$X_t^{\text{mix}} = P_t^{\text{chunk}} \cup D_t$: an admitted chunk *plus* resident next-token work, in *one* forward call. Mixed when both are nonempty.

After metadata is built, a decode row is just an extend row of length one:

$$\texttt{extend_lens} = [2048, 1, 1, 1].$$

- First entry: r_{long}'s chunk. The rest: A, B, C decode.
- SARATHI's piggybacking: pick one prefill chunk, then fill the step with decode rows so neither starves.

The semantic split must survive into one coherent metadata ledger.

The contract that keeps the mixed batch honest

Metadata-order invariant
If r_i sits at position i in the batch, *every* per-request field — pool index, prefix/extend lengths, `out_cache_loc`, sampling — must use position i; flat token order must match the per-request lengths.



Kernels read parallel flat arrays by row index. If `extend_lens` $= [8, 1, 1]$ but cache locations are ordered A', L', B', the chunk writes into a decode request's KV slot — legal shapes, silently wrong.

Overlap and future tokens

Overlap moves launch earlier, not commit out of order

Normal loop: schedule $k \to$ run $k \to$ process $k \to$ schedule $k{+}1$.

Definition (Overlap scheduler state)

$$S_t^{\mathrm{ov}} = (W_t, B_t, \Lambda_t, U_t, M_t, Q_t^{\mathrm{result}}, F_t^{\mathrm{future}}).$$

Q_t^{result}: launched batches awaiting result processing. F_t^{future}: future-token map.

Launch for $k{+}1$ may start before k's CPU processing finishes — but only if every value crossing that line is resolved or carried as a future handle. Commit, sampling, and decode residency still respect request order.

The future-token handle: a negative integer that is not a token

Definition (Future token handle)
$h = (r, j, f)$: request r, future output position j, future-map slot f. Stored in an input-ID array as the negative integer $-f$.

- Vocabulary IDs are *nonnegative*, so $-f$ rides an `int64` tensor without being mistaken for a real token.
- Resolver predicate: "input id < 0" $\Rightarrow$ replace $-f$ by the committed value in slot f.

Future-resolution invariant
No forward may consume a handle as a token ID. Every handle is resolved on the execution stream before embedding, sampling-metadata use, or attention.

From concept to code: the negative-handle resolver

The whole handle contract is one `where`.

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

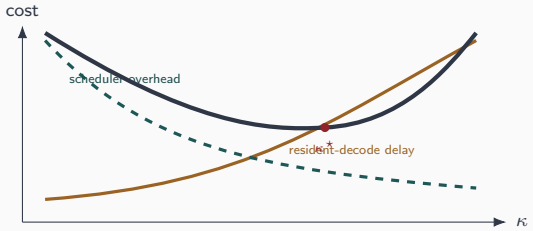
```
def _resolve_future_token_ids_native(input_ids, future_token_ids_map):
    input_ids[:] = torch.where(
        input_ids < 0,
        future_token_ids_map[torch.clamp(-input_ids, min=0)],
        input_ids,
    )
```

Read it as the invariant: `input_ids < 0` detects a handle $-f$; `future_token_ids_map[-input_ids]` is the lookup in slot f; a nonnegative real token passes through untouched. After this line, a negative input id is a bug, not a token.

Tuning, guardrails, evidence

One knob, two opposing costs

Chunk count for a suffix of length e: $N_{\text{chunks}}(e, \kappa) = \lceil e/\kappa \rceil$. For $e = 9216$: κ=2048 $\Rightarrow$ 5, κ=1024 $\Rightarrow$ 9.



Larger κ raises per-step interference but lowers per-suffix overhead. Tune from the latency budget: $T_{\text{prefill}}(\kappa) + T_{\text{launch}} \leq T^{\text{max}}_{\text{decode}}$, then check memory headroom.

Guardrails: where chunking or overlap must back off

Guard	Protects	Failure kind
Multimodal / mixed chunking	row & embedding meaning, metadata order	wrong output
Dependency-sensitive overlap	future / grammar / spec state	wrong / stale state
Consecutive-prefill overlap	TTFT contract	latency regression
Chunk-size alignment	legal KV alloc, determinism	invalid step
Distributed consistency	rank-consistent branch	deadlock

Each guard marks a place where the scheduler cannot describe one valid forward step.

Measured Thor anchors back the policy: overlap on $\rightarrow$ 14.1 ms ITL vs. 17.9 ms off; the 256-token chunk is *not* the safest — overhead hurts TTFT and throughput.

What to carry forward

1. A long prompt is a **schedulable suffix**: chunk it, advance the prefix exactly by produced KV state.
2. A **mixed step** folds chunk + decode into one extend-style batch — the **metadata-order invariant** keeps it honest.
3. **Overlap** moves launch earlier; **future handles** ($-f$) carry not-yet-produced tokens and must resolve before any forward.
4. κ is a **scheduler control parameter** balancing decode delay against per-suffix overhead.
5. Every mechanism has a **home in the SGLang scheduler, policy, and overlap utilities**.

Next: the section decks walk the suffix object, the mixed-step contract, the overlap loop, and the guardrails one mechanism at a time.