# The Inference Serving Problem | 2.

**Chapter Abstract.** Online LLM serving is an online decision problem. The runtime must coordinate stateful requests and finite accelerator work. It must also preserve resident token state, exploit prefix reuse, and meet user-visible latency and throughput goals. This chapter builds that problem from a concrete serving trace: one request is prompt-heavy, one is decode-heavy, and one becomes cheap only if the runtime recognizes reusable prefix state. The question is which work should run next, and what breaks if the runtime chooses by request count alone. The chapter first defines request state, prefill and decode work, KV capacity, and effective prompt work after prefix reuse. It then defines the metrics, the online serving decision, and the construction of feasible serving steps. Empirical anchors and source-history pressure then test why the same variables recur through the rest of the book.

## 2.1. The Serving Trace and Control Problem

At 12 ms, request D arrives and looks like another long prompt. A moment earlier, request B may have made most of D cheap, but the scheduler has to decide before hindsight can explain the relationship. That is the serving problem in miniature: the runtime must choose with only the current state visible.

Figure 2.1 shows the external service shape that creates the scheduler-facing problem. Many users send request streams, the runtime selects work for accelerator workers, resident model state must remain aligned with the live requests, and partial outputs stream back in request order.

Serving is a stateful workload.

Large language models become useful to applications when they can serve many independent users with low latency, high throughput, and predictable memory use. That service setting is different from running one isolated forward pass. New requests arrive while older requests are still decoding, users expect partial tokens to stream back, and the server must keep expensive accelerators busy without letting token state exhaust fast memory. Section 2.3 formalizes token state (Definition 2.3.3); here, read it as resident per-token model state that future positions must read. Fast memory means accelerator-resident memory, such as GPU HBM, that can hold live token state for immediate model execution. This is why SGLang's organization follows the stateful service loop through scheduling, memory, prefix reuse, and measurement.

### The Serving Control Problem

Before any SGLang-specific mechanism appears, the serving system faces a control problem. It must decide what work to run now, what work to delay, and what state must remain resident so future tokens can be generated correctly. The useful mental model is a continuously updated dispatch problem over work, resources, constraints, and objectives. The table previews prefill work and decode work before Section 2.3 defines them (Definition 2.3.1 and Definition 2.3.2).

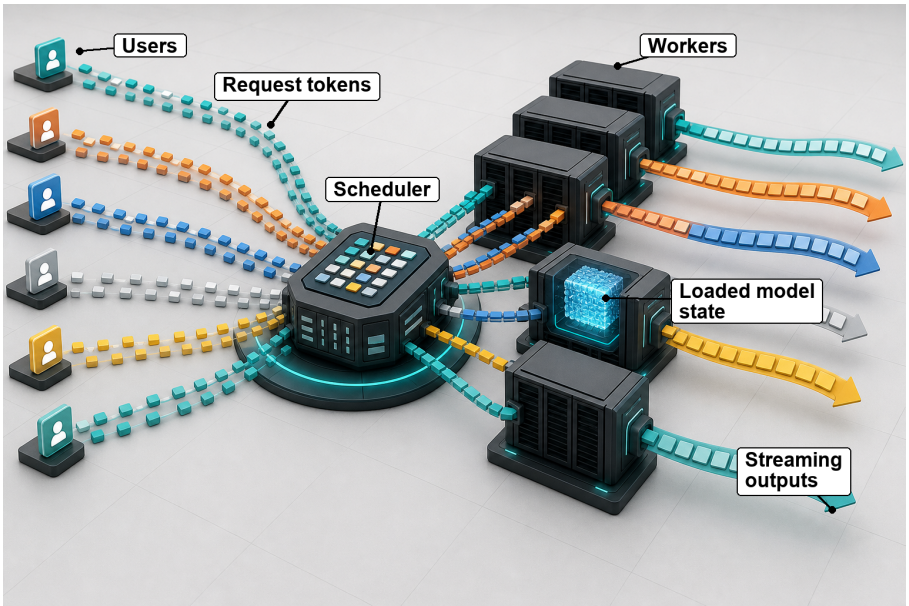



**Figure 2.1.:** Online LLM serving as a many-user streaming service. User requests enter the service, control logic selects work, accelerator workers hold the loaded model state, and output token streams leave in request order.

The running trace will keep that control problem concrete. Request B wants a large prompt admitted, request C will occupy decode for many later steps, and request D should be treated differently only after its relationship to B becomes visible. The chapter's definitions answer that one repeated question: at this decision boundary, what can the runtime see, what can it safely run, and which request pays if it waits?

At each step, serving has to choose among request promises competing for the next row.

Section 2.5 turns the latency and throughput names in the objective row into measured quantities: time to first token (Definition 2.5.2), inter-token latency (Definition 2.5.3), end-to-end latency (Definition 2.5.1), and input or output throughput (Definition 2.5.5).

The resulting problem is online transformer inference serving: repeatedly choosing which visible request work should run next, subject to finite accelerator execution capacity, finite fast memory for token state, variable arrivals, variable prompt lengths, variable output lengths, prefix reuse opportunities, and streaming output constraints. Chapter 3 will apply the Definition 1.5.2 and Definition 1.7.2 cost primitives to this online decision problem.

We will use one running example throughout the discussion. Four generation requests for `google/gemma-4-E4B-it` arrive at a single serving instance. Request A is $r_{\text{pan}}$, the pancake request from Chapter 1, now viewed as an online arrival. Requests B–D are companion pressures chosen for this chapter's control problem:

The token counts are intentionally scaled down as a compact service trace so the chapter can do arithmetic while keeping the service concrete: A/$r_{\text{pan}}$ is short in both prompt and output; B is prompt-heavy; C is decode-heavy; and D shows why raw prompt length is not always the right measure of work for the same `google/gemma-4-E4B-it` serving instance. Figure 2.2 shows the same four requests as an arrival timeline.

| Question | Serving interpretation |
|---|---|
| What work is waiting? | New prompt tokens may need prefill, live requests may need their next decode token, and completed requests may need bounded output handoff. |
| What resources exist? | The server has finite accelerator execution capacity, finite request slots, finite fast memory for token state, and finite output-stream bookkeeping. |
| What constraints bind? | Prompt and decode work have different shapes; KV state must fit in fast memory; arrivals, prompt lengths, output lengths, and prefix reuse vary over time; and per-request state must stay attached to the right client. |
| What is the objective? | Keep useful model work flowing while controlling time to first token, inter-token latency, end-to-end latency, throughput, fairness, and memory use. |

**Table 2.1.:** Serving-control questions at each decision boundary. Each question determines what the runtime can see, run, or delay.

| Request | Arrival | Prompt | Output | Note |
|---|---|---|---|---|
| A / $r_{\text{pan}}$ | 0 ms | 64 | 8 | short chat turn |
| B | 5 ms | 4096 | 2 | long prompt, short answer |
| C | 8 ms | 256 | 120 | short prompt, long generation |
| D | 12 ms | 1024 | 16 | shares a long prefix with B |

**Table 2.2.:** Online serving arrivals for the control problem. The rows fix arrival times, prompts, outputs, and companion pressures.

The same workload symbols recur when the chapter discusses request state, prefill, decode, KV memory, admission, dynamic batching, prefix reuse, and metrics. Rather than introduce those names as a detached notation block, Table 2.3 keeps each symbol attached to the trace fact that creates it.

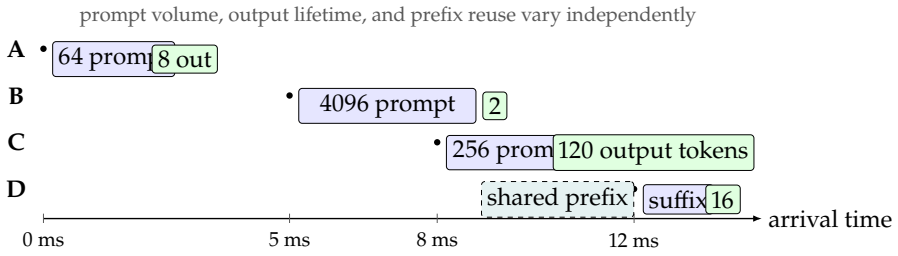



**Figure 2.2.:** The running workload is an online stream with changing pressure. The arrival order hides three different pressures: B is prompt-heavy, C is decode-heavy, and D may be cheap if its prefix state is already reusable.

**Table 2.3.:** Running workload ledger for the online inference-serving problem. The top rows keep requests A–D attached to their trace pressures; the remaining rows name the visible state, resident capacity, policy action, and latency terms that later sections formalize.

| Trace fact or symbol | Local handle | Role in the serving account |
|---|---|---|
| $A = r_{\text{pan}}$ is a short chat turn | Request identity $r$ | The user-visible work item whose prompt, output, latency, and state must stay attached across later chapters. |
| $B$ is prompt-heavy | $x_i, \lvert x_i \rvert, t_i^{\text{arr}}$ | Prompt length and arrival time explain why admitted work can dominate a step even when the request count is small. |
| $C$ is decode-heavy | $m, \tau_i^{\text{tpot}}$ | Realized output length and post-first-token timing expose resident decode pressure after the prompt has already run. |
| $D$ shares a prefix with $B$ | Prefix relation | Prefix reuse can make the raw prompt length a poor proxy for model work. |
| $\ell_i^{\text{unc}}$ | Uncached suffix length | Prompt tokens still requiring model execution after prefix reuse. |
| $\ell_i^{\text{rem}}(t)$ | Remaining prompt length | Uncached prompt work still eligible for admission at step $t$. |
| The scheduler chooses from what is visible | $\mathfrak{X}_t, \pi, u_t$ | The visible state, policy rule, and chosen action define the online decision without hindsight about future arrivals. |
| Token state consumes fast memory | $L_t, C_{\text{fast}}$ | The resident token-state ledger and fast-memory budget bound which serving steps are feasible. |
| Users observe latency, not internal steps | $\tau_i^{\text{first}}, \tau_i^{\text{e2e}}$ | The request-level first-token and end-to-end summaries tie scheduling decisions to user-visible behavior. |

## 2.2. Request State and Live Work

A request is more than a string. After tokenization, it becomes a stateful work item that persists across many serving steps. Chapter 1 used request identity as the anchor for logical positions and dynamic batches. This section specializes the request-state side of that anchor for the scheduler-facing serving problem: which state must remain visible while the runtime admits prompt work, advances decode work, releases completed work, and streams output.

A request carries phase, cache state, queue position, and output obligations.

Section 2.6 later formalizes a feasible serving step in the formal definition as selected work that satisfies the active serving constraints. It owns or references token state, accumulates generated tokens, carries finish conditions, and may have already sent part of its output to the client.

A serving step may process only a prompt slice or one decode token and leave the request live for later steps. In this sense, a request can have a history: the current step changes one suffix or one generated position, while the request state still includes earlier tokens and the resident token state they created.

**Definition 2.2.1** (Request state) *Request state is the evolving per-request record that contains the request's input tokens, generated tokens, finish status, token-state ownership or references, and output-stream progress. We write the state of request $i$ at serving time $t$ as $R_i(t)$. Its fields include the fixed input token sequence $x_i$, the generated suffix $y_i(t)$, finish status, token-state ownership or references, and output-stream progress.*

Here $i$ is the per-request index used by the serving account; later runtime chapters often write the same stable request identity as $r$ when emphasizing traces and message-carrying events rather than accounting indices.

**Example 2.2.1** (Reading request state) For request C, the same record reads differently at different serving moments:
Immediately after admission, C has a 256-token prompt, no generated tokens, an unfinished status, token state created or referenced for C, and no streamed output delivered. After three output tokens, the input prompt is unchanged, the generated-token list contains three tokens, the request may still be unfinished, the token state covers both prompt and generated positions, and output progress records how much text has reached the client.
The point is that a request carries evolving serving state along with its input string. Figure 2.3 later shows the same kind of state evolution for C while contrasting it with a request that reaches completion.

This record deliberately abstracts away many fields that SGLang must still carry. A production request can also carry priority, routing, LoRA adapter, structured-output, multimodal, or prefill/decode-disaggregation state. These fields do not change the Chapter 2 request-state definition (Definition 2.2.1), but they explain why the scheduler cannot treat all pending rows as identical strings with lengths. Later chapters develop those fields as separate request-state extensions.

In the running example, A and B may have short lifetimes after admission, but for different reasons. A has little prompt work and little decode work. B has a large prompt and a small expected output. C is the opposite: its prompt is modest, but its generated sequence may keep it live for many serving steps.

**Definition 2.2.2** (Live and completed requests) *A request is live at time $t$ if it has been admitted to model execution and has not reached its stopping condition. A request is completed once its stopping condition has been reached and any required output bookkeeping can be released.*

Figure 2.3 shows this state transition visually. The request identity stays fixed while the scheduler-visible fields change.

The distinction between live and completed requests (Definition 2.2.2) matters because live requests compete for decode work and token state, while completed requests should stop consuming those resources. This is a basic correctness rule for a high-throughput serving runtime:

**Invariant 2.2.3** (Active work excludes completed generation)**.** *Completed generation requests must not remain part of the active decode set after their finish state is visible to the scheduler, except for bounded output handoff bookkeeping.*

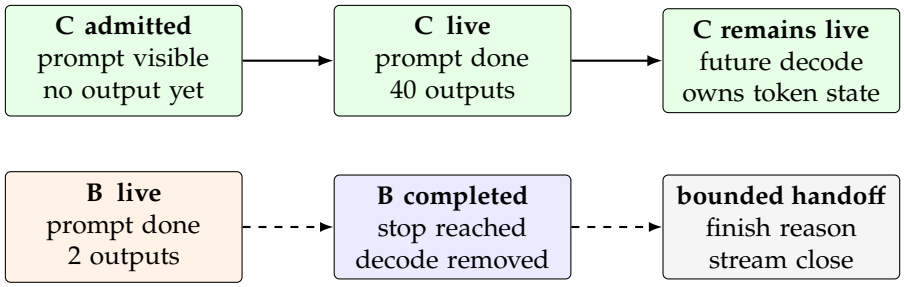



**Figure 2.3.:** Request state changes over time while request identity remains the anchor. C remains live and keeps competing for model work and resident token state; B's state record reaches a finish condition and leaves the active decode set after bounded output handoff.

The important word is active. A completed request may still need final detokenization, a finish reason in the response, or cache-release bookkeeping, but it should no longer be scheduled as future model work. If it remained in the decode population, the scheduler would count it against decode capacity, token-state memory, and running-load reporting even though no next token is needed. SGLang's runtime therefore separates finish detection from bounded output handoff: once a request's finish state is recorded, later batch filtering removes it from active decode while the output path emits the final response once.

**Example 2.2.2** (Completed request left active) Suppose requests A and B are decoding, and A reaches its finish condition at step $t$. A may still need bounded output handoff, but if the scheduler keeps A in the active decode set at step $t+1$, A consumes a decode row, token-state accounting, and scheduling capacity even though no next token is needed. The violation is not that A still has response bookkeeping; the violation is that A still competes for future model work. Figure 2.3 above shows the intended outcome for this distinction: completed work leaves active decode while bounded output handoff remains.

The rule is conceptually simple, but it drives much of SGLang's design: serving is a loop that repeatedly filters finished work, admits new feasible work, and advances unfinished work.

**Trace state.** After this section, A, B, C, and D are no longer just prompt rows in an arrival table. Each is a stateful record that can wait, run, stream, finish, or keep resident token state alive after the current serving step.

## 2.3. Prefill, Decode, and Resident Token State

The next trap in the running trace is that visible work and lasting pressure are not the same thing. Request B creates a prompt burst; request C creates a long decode tail; both leave resident token state that future steps must protect. Figure 2.4 makes the asymmetry visible: B creates a wide prompt burst, C creates a long decode tail, and both leave resident token state future steps must protect.

Transformer serving separates naturally into two forms of work. The first turns input context into reusable token state. The second extends a sequence using that state.

Prefill is prompt-wide work; decode is stepwise work. They stress different resources.

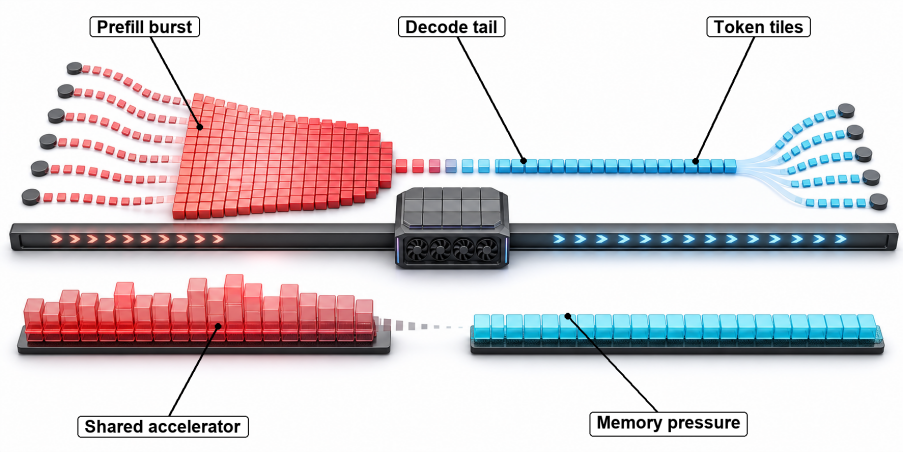



**Figure 2.4.:** Prefill burst versus decode tail. A wide front-loaded prefill burst contrasts with a long narrow decode tail, with token tiles, a shared accelerator lane, and distinct memory pressure bands.

For the serving discussion, the only transformer background we need is this: a request becomes a sequence of token positions, and an autoregressive model extends that sequence by predicting one next token at a time. Each new position uses attention over earlier positions. During serving, the runtime keeps reusable per-token attention state, usually called KV state, so later generation can read the prior context without recomputing the whole prefix from scratch. The prefill/decode split follows from this autoregressive attention pattern [44].

Chapter 1 already named prefill, decode, and mixed forward steps (Definition 1.3.1). Chapter 2 uses the same split at the scheduler boundary: prefill work spends prompt budget and creates resident token state, while decode work repeatedly advances live requests using state that already exists.

**Definition 2.3.1** (Prefill work) *Prefill work is model execution over prompt or newly admitted context tokens whose result creates token state that can be used by later generation steps. For request $i$, if $q_i$ uncached input tokens are admitted in a serving step, then $q_i$ is the request's prompt work volume for that step.*

Prefill work is the creation side of the serving step: it consumes prompt tokens and produces resident token state. The paired decode definition names the continuation side, where the runtime spends repeated steps on requests whose prior state is already available.

**Definition 2.3.2** (Decode work) *Decode work is model execution that advances live requests using already available prior token state. If a serving step advances $k$ live requests by one generated position each, its decode work volume is $k$ request-positions.*

In the running example, B contributes 4096 raw prompt tokens but only 2 target output tokens. C contributes 256 raw prompt tokens but 120 target output tokens. A static measure such as "number of requests" hides the important distinction: B pressures prompt processing and cache allocation early; C pressures repeated decode steps over time. Figure 2.5 shows this as a difference in where the work appears on a request's lifetime.

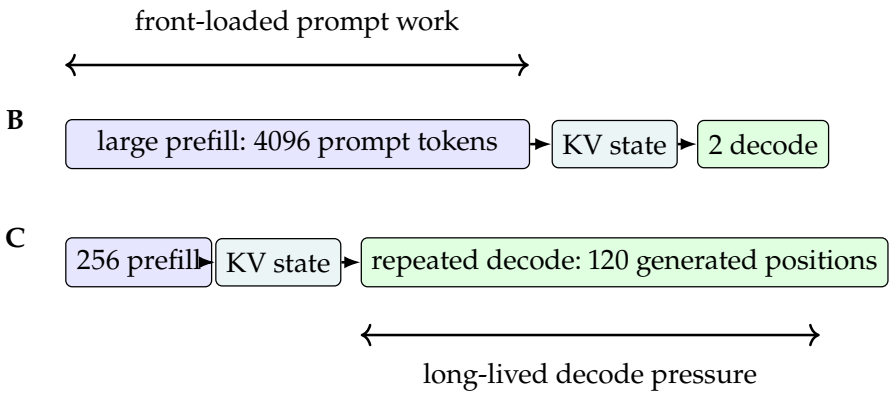



**Figure 2.5.:** Prefill and decode have different serving shapes. Request B is costly early because it creates a large amount of token state; request C is cheaper to admit but remains in the live decode population for many steps.

**Example 2.3.1** (Two kinds of volume) Suppose the system admits B and C together with no prefix reuse. The two serving moments have different units: At prompt admission, the selected work is B's 4096-token prompt plus C's 256-token prompt, for $4096 + 256 = 4352$ prompt tokens. At a later decode step, A, B, and C each advance by one generated position, so the work volume is three request-positions.
These are different units. Chapter 3 will apply the Chapter 1 model-shape and hardware cost primitives to them. Figure 2.5 above shows the same contrast as front-loaded prompt work versus long-lived decode pressure.

The distinction between prefill work (Definition 2.3.1) and decode work (Definition 2.3.2) explains why SGLang's serving logic treats newly admitted prompt work and continuing decode work differently [41].

## Resident Token State and KV Capacity

KV cache turns future attention work into resident memory pressure.

The accelerator cost of a forward pass is visible, but the persistent memory cost of generation is often the tighter serving constraint. For attention-based decoding, each admitted token can create token state that must remain available for later tokens in the same request. The exact layout depends on the model and backend, but the serving problem always has a memory budget. Prior LLM-serving systems identify KV-cache memory growth, fragmentation, and sharing as central throughput constraints [19]. This is the section's turn: after prefill and decode name the visible work, KV capacity names the hidden obligation that can block otherwise ready work.

**Definition 2.3.3** (Token state) *Token state is the per-token model state retained so that future positions in the same sequence can attend to, or otherwise depend on, the already processed tokens. For request $i$ and logical position $j$, write this resident payload as $Z_{i,j}$.*

**Example 2.3.2** (A token-state payload is not one number) For a standard attention KV layout, one $Z_{i,j}$ expands conceptually as

$$[\text{layers}] \times [\text{K,V}] \times [\text{KV heads}] \times [\text{head dimension}],$$

with a dtype attached to every stored element. The simple full-history Gemma-shaped baseline from Chapter 1 gives

$$42 \cdot 2 \cdot (256 + 256) \cdot 2 = 86{,}016 \text{ bytes}$$

per logical token position. Reading left to right, the factors are 42 effective attention layers, 2 KV heads, 256 key elements plus 256 value elements per head, and 2 bytes per stored fp16/bf16 element. The key and value dimensions are added because the cache stores both vectors for the same logical position; they would be multiplied only if they formed independent axes. The result is about 84 KiB before page slack, reservations, and Gemma-specific sharing or sliding-window refinements. A 4096-token prompt would therefore represent roughly 336 MiB of standard resident KV payload in that baseline. This is the concrete size behind the claim that memory is a scheduler constraint.

Token state is the object that persists after the forward computation has finished. The capacity definition below turns that object into a bounded resource: it asks how many such per-token states can remain resident under the current model and cache layout (Definition 1.6.6).

**Definition 2.3.4** (Fast-memory token capacity) *Fast-memory token capacity is the maximum amount of token state that can be kept resident in the serving instance's fast memory under the current model, precision, cache layout, and reservation policy. We denote the corresponding resident token-state budget by $C_{\text{fast}}$, measured in the token slots or page-aligned units used by the active cache layout.*

Capacity alone is a static budget. The scheduler needs the next definition because a proposed batch is legal only if its new token state fits while the already-live token state remains protected.

**Definition 2.3.5** (Memory-feasible batch) *A serving dynamic batch is memory-feasible if the token state newly required by its selected work fits within currently available fast-memory capacity while preserving the token state that must remain live for unfinished requests. If $L_t$ is the live resident state before step $t$ and $N(B_t)$ is the new state demanded by selected batch $B_t$, where $N(B_t)$ counts the new $Z_{i,j}$ payloads created by the selected prompt or decode positions, the basic feasibility check is*

$$L_t + N(B_t) \le C_{\text{fast}},$$

*after page rounding, reservations, and reusable-state credits are applied.*

$L_t$ is already-owned resident state; $N(B_t)$ is the new state this batch would write.

**Example 2.3.3** (Fits new writes, fails live-state preservation) Use a scaled-down ledger for the `google/gemma-4-E4B-it` running workload, where one slot stands for one logical token-state unit after cache-layout accounting. Suppose fast memory can hold ten such slots. Requests A and B already require eight slots to remain live, and the next candidate batch would write three new slots. Looking only at the three new writes is misleading: the free space is only two slots. Unless the scheduler can release, reuse, retract, or delay some live state, the batch is not memory-feasible because $8 + 3 > 10$. Figure 2.6 draws this capacity ledger directly.

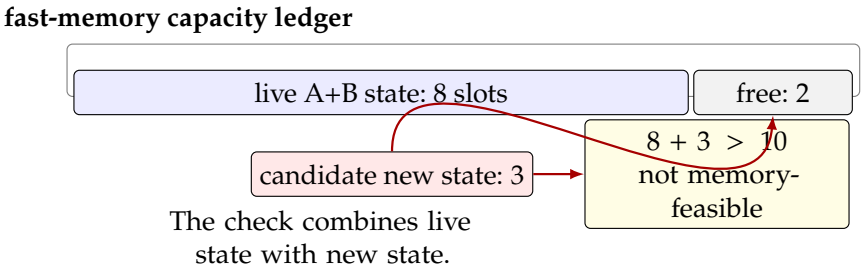



**Figure 2.6.:** A memory-feasible batch is checked against resident live state plus the new token state selected for the next step. Here three new slots cannot be admitted while eight live slots must remain resident in a ten-slot budget.

Chapter 1 already gives the transformer-local footprint formulas for standard KV storage and page-aligned resident capacity (Definition 1.6.2 and Definition 1.6.4). Chapter 2 only needs the serving consequence: resident memory scales with the number of live token positions times the model-specific per-token state payload, after cache-layout and reservation policies are applied.

This is why the discussion uses the broader phrase token state (Definition 2.3.3). In SGLang, the resident budget may include a standard full-attention KV pool, a separate sliding-window pool, Mamba or other state-space-model state [14], hierarchical cache state, or a chunk cache when radix reuse is disabled. Some model/back-end combinations deliberately disable a cache path to avoid reusing state under an invalid interpretation.

**Invariant 2.3.6** (Future token state remains accounted for)**.** *For every unfinished request, token state needed by future prompt or decode positions remains accounted for in the serving state. The state may be resident, reused through a valid cache entry, recoverable from an allowed cache tier, or explicitly released or retracted by policy, but it must not disappear from the memory ledger while future work still depends on it.*

**Example 2.3.4** (Unaccounted cache disappearance) Suppose request C is unfinished and its next decode step will need the token state for its resident prefix. A cache tier evicts C's pages to make room for other work but does not record that the pages remain recoverable from a lower tier, reusable through a valid cache entry, or released or retracted by policy. The memory ledger now shows those pages as free, admission proceeds, and another prompt slice consumes the apparent capacity. On C's next decode step, the needed state is neither resident, nor validly reusable, nor marked released. The violation is not the eviction; it is the unaccounted disappearance—the ledger claimed capacity that future work still owned. Figure 2.7 gives the resident-state picture behind this failure: live requests occupy token slots even when pressure makes eviction tempting.

Write $W_t^{\text{ready}}$ for visible prompt or decode work whose request-local dependencies are satisfied at the start of step $t$, before resource feasibility is applied. Ready work is not necessarily feasible work: it can still be blocked by memory, prompt-token budget, decode budget, queue caps, or compatibility limits.

**Definition 2.3.7** (Memory pressure) *Memory pressure is a serving state in which upcoming prompt or decode work cannot be admitted without freeing, reusing, moving, delaying, or retracting some token state. In capacity notation, pressure means that some ready work set $W_t^{\text{ready}}$ has demand $N(W_t^{\text{ready}})$ with $L_t + N(W_t^{\text{ready}}) > C_{\text{fast}}$ under the current reservation policy.*

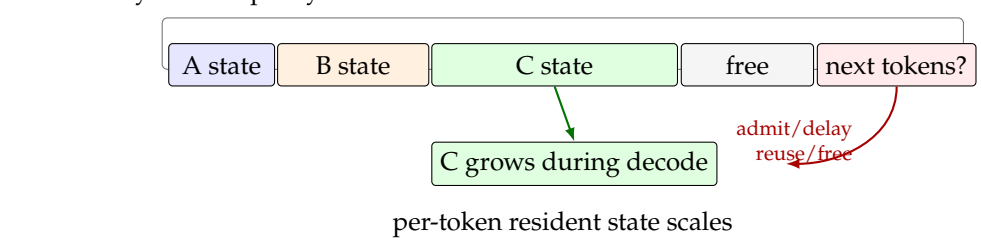



**Figure 2.7.:** KV memory pressure is a resident-state constraint. A request with a modest prompt can become the long-term memory consumer if it remains live and keeps appending generated token state. Each logical token count expands by the model's layers, KV heads, head dimension, element size, and cache-layout factor.

**Definition 2.3.8** (Decode retraction) *Decode retraction is the act of removing one or more live requests from the current decode batch, releasing their immediately needed token-state allocation, and placing them back into a state from which they can be scheduled again later. If $\mathcal{D}_t$ is the selected decode-request set and $\mathcal{D}_t^{\text{ret}} \subseteq \mathcal{D}_t$ is the retracted subset, the surviving decode-request set is $\mathcal{D}_t \setminus \mathcal{D}_t^{\text{ret}}$.*

**Example 2.3.5** (Memory-feasible decode after retraction) In the same compact ledger, suppose fast memory has room for two new token-state writes, but the current decode set contains A, B, and C. A one-token decode step over all three live requests would need three new writes, so the batch is not memory-feasible. If the scheduler retracts C, it can run a two-request decode step for A and B within the available capacity while keeping C unfinished and eligible for a future step. C is not completed; it is live work temporarily removed to make the current step fit. Figure 2.8 draws that state transition: C leaves the current decode step so A and B can fit, but C remains unfinished work.

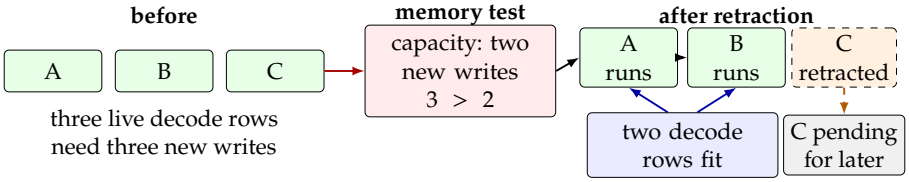



**Figure 2.8.:** Decode retraction makes a memory-infeasible decode step feasible by removing one unfinished live request from the current decode set. A and B can receive the two available token-state writes, while C returns to pending live work for a later step rather than completing.

Memory pressure (Definition 2.3.7) is the condition that makes decode retraction (Definition 2.3.8) different from normal completion. A completed request has reached its stopping condition and should leave active generation. A retracted request is still unfinished; the scheduler has temporarily given up its live decode

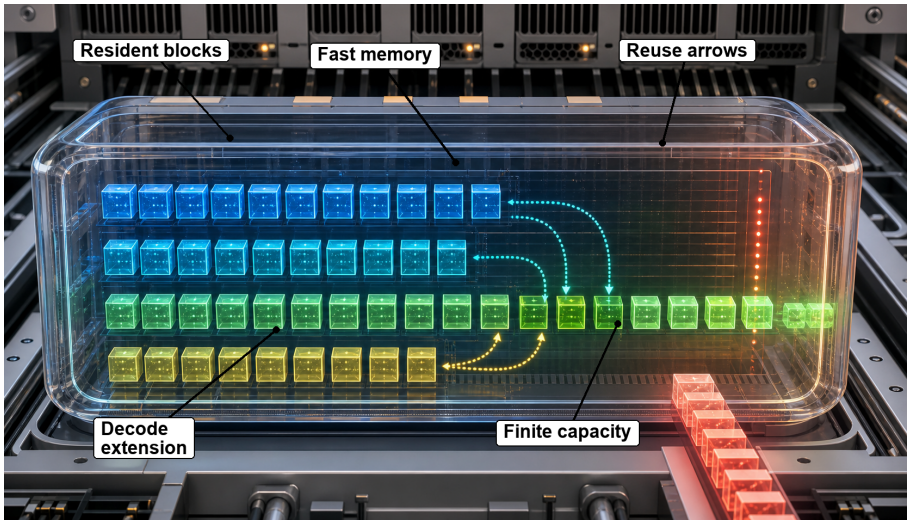



**Figure 2.9.:** Fast memory as a finite home for resident token state. Blocks are resident token state; arrows indicate conceptual reuse by later positions, decode extension of live sequences, and incoming admission pressure rather than literal memory-copy paths. New work may need admission, delay, reuse, or release decisions before it can fit.

slot because the next decode allocation does not fit. SGLang treats this as part of memory recovery: it first tries to make enough room for decode, then requeues retracted requests, and only aborts if even the remaining minimal decode work cannot be made feasible.

**Concept-to-code map**

- **Memory-feasible decode test.** The feasibility check maps to `ScheduleBatch.check_decode_mem` in `python/sglang/srt/managers/schedule_batch.py`: it computes the next decode allocation, asks the prefix cache to free reusable state when possible, and compares the demand with currently available token-state capacity. The concrete eviction routine is a later prefix-cache detail; here the key point is that eviction is attempted before retraction.
- **Decode retraction.** Retraction maps to `ScheduleBatch.retract_decode` in the same file. It orders unfinished requests, releases their token-state slots, and returns them to scheduler-owned live work. Chapter 5 traces the request-object fields used by that ordering.
- **Minimal-feasible fallback.** The fallback path maps to the same retraction routine returning `reqs_to_abort`; `Scheduler.update_running_batch` in `python/sglang/srt/managers/scheduler.py` sends those aborts when retraction still cannot make the remaining decode subset fit.

In the running example, C can become the long-term memory consumer even though B has the largest prompt. Each generated token for C extends the sequence state that future decode steps need. B's prompt is expensive early, but if B stops after two output tokens, its live lifetime can be short.

**Trace state.** B explains prompt pressure, C explains resident-state pressure, and the scheduler must keep both facts visible before it can decide whether D's future prefix relationship is useful or merely another waiting request. Figure 2.9 shows why fast-memory token capacity (Definition 2.3.4) is a resident-state constraint rather than a one-step compute constraint.

## 2.4. Prefix Reuse and Effective Prompt Work

Prompt work depends on reusable resident state as well as raw prompt length. If a request shares a prefix whose token state is already available, the serving instance can reuse that state subject to its cache rules. SGLang includes explicit logic for prefix-aware serving and cache-backed admission decisions [49].

Prefix reuse changes the marginal cost of a request.

**Definition 2.4.1** (Prefix hit) *A prefix hit for request $i$ is a nonempty prefix of $x_i$ whose token state is already available for reuse and valid under the serving system's cache rules.*

At this level, cache-rule validity has four parts. Request-local logical positions agree. The cache key records model and layer identity. The RoPE configuration satisfies the Chapter 1 position-agreement invariant. The KV-cache layout also matches what the attention backend expects. For this chapter, a cache key is only the identity test that says two pieces of resident state were produced for the same reusable prefix under compatible model, layer, position, and layout rules. A prefix-cache node is only the metadata record that protects such reusable state. The concrete radix-node layout belongs to the prefix-cache chapter; the serving decision here only needs to know whether the hit is valid and how long it is.

The hit definition names the reusable part of the prompt. The complementary quantity for scheduling is the prompt tail that still needs model execution.

**Definition 2.4.2** (Uncached suffix) *If request $i$ has prompt length $|x_i|$ and reusable prefix length $p_i$, then its uncached suffix length is*

$$\ell_i^{\mathrm{unc}} = |x_i| - p_i.$$

$\ell_i^{\mathrm{unc}}$ subtracts reusable prompt prefix only; output length is a separate account.

This Chapter 2 definition is a prompt-level length account. Later chunked-prefill scheduling turns the same idea into a step-indexed suffix, measured after generated tokens and earlier committed chunks have advanced the request.

**Definition 2.4.3** (Effective prompt work) *Effective prompt work is the amount of prompt execution still needed after accounting for reusable prefix state and cache-layout constraints (Definition 1.6.6). In the simplest token-count model, it is $\ell_i^{\mathrm{unc}}$, the uncached suffix length.*

**Example 2.4.1** (Prefix hit arithmetic) Request D in the running example has 1024 raw prompt tokens. If 900 of them are already reusable because they match earlier admitted context from B, then D has a prefix hit (Definition 2.4.1). Its uncached suffix (Definition 2.4.2) and simple effective prompt work (Definition 2.4.3) are $1024 - 900 = 124$ tokens. Under this view, D is closer to A than to B, despite its raw prompt length. Figure 2.10 draws the same arithmetic as a shared-prefix hit followed by a small uncached suffix.

This is the point where raw prompt length stops being a reliable intuition: the same request can be expensive, cheap, or still invisible depending on which prefix state the runtime has already made resident.

SGLang distinguishes more cases than this simple arithmetic shows. A prefix may already be resident on the device, may be recoverable from host memory or a storage-backed hierarchical cache, or may only be a potential in-batch reuse opportunity because another waiting request shares the same prefix. Device hits reduce both prompt execution and transfer work. Host or storage hits can reduce model execution while adding load-back or prefetch work. In-batch prefix relationships are not hits yet, but they can still change ordering: running one representative request first may create useful cache state for the others. The queue order can therefore depend on cache state that exists now and on cache state that a related admission would create. At this point, that is the scheduling lesson; the concrete radix-tree machinery belongs with the prefix-cache data structure rather than with this serving-problem model.

Figure 2.10 also marks the scheduling intuition: raw prompt length stops being reliable once prefix state is reusable.

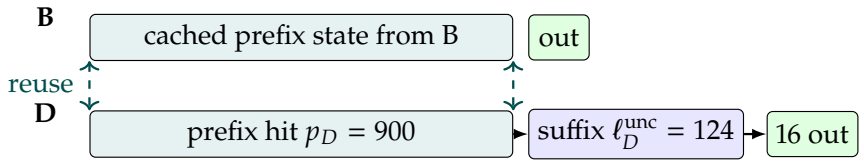



**Figure 2.10.:** Prefix reuse changes the amount of prompt execution still needed. Request D has a long raw prompt, but a large prefix hit makes its uncached suffix much smaller than its apparent length.

Prefix reuse therefore changes both scheduling and measurement. A benchmark that reports only raw input tokens may hide how much model execution was avoided by reuse. A scheduler that ignores reuse may delay work that is actually cheap. The feasible-step construction in Section 2.7 uses this effective-work account when it admits prompt slices.

**Trace state.** D is the chapter's reminder that arrival order and raw prompt length are not enough. D looks long at the API boundary, but after B makes the shared prefix resident, the scheduler should see a much smaller marginal prompt suffix.

## 2.5. Metrics and Operating Points

First-token delay and inter-token gaps measure different user-visible failures.

The prefill/decode split from Section 2.3 also shapes how serving runs are measured: prompt admission strongly affects first token delay, while repeated decode work appears as gaps between later tokens. SGLang's benchmark code reports serving metrics that are more specific than a single requests-per-second number. This section defines time to first token (TTFT), inter-token latency (ITL), and related metrics so later chapters can use them precisely.

**Definition 2.5.1** (End-to-end latency) *For request $i$, end-to-end latency is*

$$\tau_i^{e2e} = t_i^{\text{done}} - t_i^{\text{arr}},$$

*where $t_i^{\text{arr}}$ is arrival time and $t_i^{\text{done}}$ is completion time.*

End-to-end latency measures the whole request lifetime. It does not identify where the wait occurred, so the next metric isolates the user-visible delay before the first generated token appears.

**Definition 2.5.2** (Time to first token) *For a generation request that emits at least one output token, time to first token is*

$$\tau_i^{first} = t_{i,1}^{\text{out}} - t_i^{\text{arr}},$$

*where $t_{i,1}^{\text{out}}$ is the first output-token delivery time.*

If request $i$ arrives at 10 ms, delivers its first token at 85 ms, and completes at 210 ms, then $\tau_i^{\text{first}} = 75$ ms while $\tau_i^{\text{e2e}} = 200$ ms.

Time to first token ends when generation becomes visible. After that point, the serving question shifts from first response delay to the gaps between successive delivered tokens.

**Definition 2.5.3** (Inter-token latency) *If request i emits delivery times $t_{i,1}^{\text{out}}, \ldots, t_{i,m}^{\text{out}}$, then its inter-token latencies are*

$$t_{i,j}^{\text{out}} - t_{i,j-1}^{\text{out}}, \qquad 2 \le j \le m.$$

If a request delivers tokens at 40, 55, and 72 ms, its two inter-token latencies are 15 ms and 17 ms. The first-token delay belongs to TTFT.

Inter-token latency is a sequence of gaps. The following definition collapses the post-first-token part of a completed request into one average when a benchmark needs a compact summary.

**Definition 2.5.4** (Time per output token) *For a completed generation request with $m > 1$ output tokens, time per output token is*

$$\tau_i^{tpot} = \frac{\tau_i^{e2e} - \tau_i^{first}}{m - 1}.$$

If a request finishes in 180 ms, emits its first token at 60 ms, and returns $m = 7$ output tokens, then $\tau_i^{\text{tpot}} = (180 - 60)/(7 - 1) = 20$ ms per post-first-token output.

Benchmark reports may summarize these gaps by median or percentile. End-to-end latency (Definition 2.5.1) covers the whole request lifetime. Time to first token (Definition 2.5.2) is strongly affected by prompt admission, prompt processing, prefix reuse, and queueing delay. Inter-token latency (Definition 2.5.3) is more directly tied to the repeated decode path and the number of live requests that share it. They can therefore move in different directions. Time per output token (Definition 2.5.4) is a coarser post-first-token average; it can hide bursty inter-token gaps, but it is useful when comparing many completed requests with different output lengths.

Percentiles matter because batching couples requests. A long prompt admission, a large mixed step, or a memory-pressure recovery can delay tokens for other live requests even when the median request looks healthy. Means and medians

answer what a typical request saw; P95 or P99 TTFT, ITL, and end-to-end latency answer whether the service creates visible stalls for the slowest slice of requests. This is why tail metrics are serving-quality metrics rather than statistical decoration: they expose distributional failures caused by queueing, batching, and shared-memory pressure.

The TTFT definition assumes that the serving path exposes a first-token delivery event. A streaming response exposes that event, so TTFT, ITL, and TPOT occupy different spans of one request trace. A non-streaming response exposes only the completed response to the client. When a benchmark still populates a TTFT field for that path, SGLang's benchmark convention is to set TTFT equal to total latency because no first-token timestamp was observed. That convention is useful for reporting a non-streaming run, but it is not the same measurement as streaming TTFT. Figure 2.11 separates the observable events.

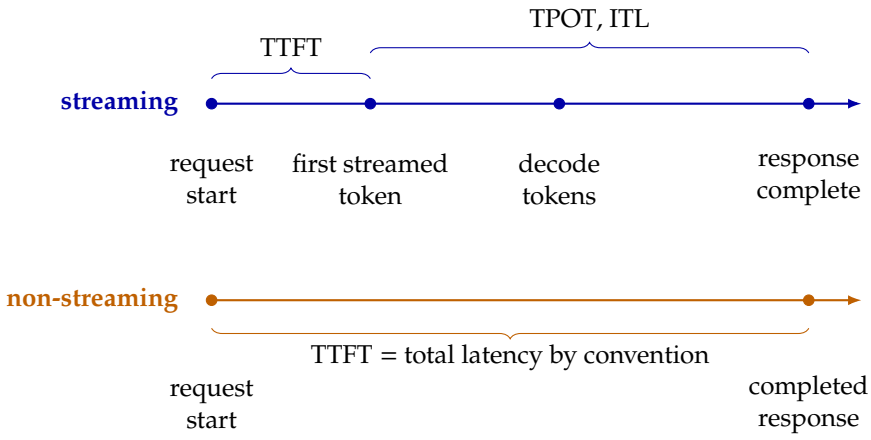



**Figure 2.11.:** Streaming and non-streaming latency semantics. Streaming TTFT names a first-token boundary, while non-streaming TTFT is equal to total latency by benchmark convention because no first-token event is observed.

**Concept-to-code map**

- **Streaming latency samples.** TTFT and inter-token latency map to `RequestFuncOutput.ttft` and `RequestFuncOutput.itl` in `python/sglang/bench_serving.py`. The native streaming path records the first observed token as TTFT, then divides each streamed chunk gap across the newly reported completion tokens before appending ITL samples.
- **Fixed-output contract.** The requested-output length maps to `RequestFuncInput.output_len` and request payload fields such as `max_tokens`, `max_new_tokens`, and `ignore_eos`; the benchmark enables `ignore_eos` by default unless the run disables or overrides it.

Figure 2.12 places the three latency families on one request timeline.

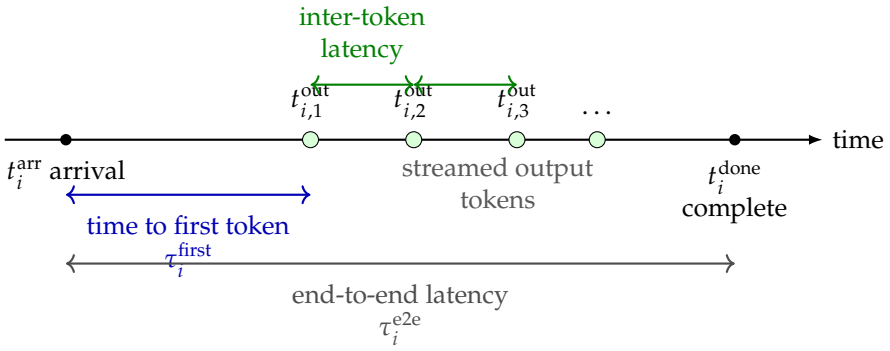



**Figure 2.12.:** Serving metrics measure different parts of the same request trace. Time to first token includes queueing and prompt work before the first visible token, inter-token latency measures gaps during streaming decode, and end-to-end latency spans arrival to completion.

**Example 2.5.1** (One request timeline) Suppose request C arrives at $a_C = 8$ ms, delivers its first three output tokens at $b_{C,1} = 46$ ms, $b_{C,2} = 56$ ms, and $b_{C,3} = 67$ ms, and completes at $c_C = 1246$ ms after $m = 120$ output tokens. The metric calculations are:
The time to first token is $46 - 8 = 38$ ms. The first two inter-token gaps are $56 - 46 = 10$ ms and $67 - 56 = 11$ ms. End-to-end latency is $1246 - 8 = 1238$ ms, and time per output token over the post-first-token span is $(1238-38)/(120-1) = 10.1$ ms.
A scheduling change that admits C earlier may improve the first number without changing the later decode gaps; a decode-heavy live set may worsen the gaps even after the first token has already arrived. TPOT is a post-first-token average over the whole completed request, so it should not be read as the same quantity as any individual inter-token gap. Figure 2.12 above shows where those timestamps and latency spans sit on one request trace.

**Definition 2.5.5** (Input and output throughput) *For a measurement interval of duration $\Delta t$, input throughput is*

$$\Theta_{in} = \frac{\textit{input tokens processed}}{\Delta t},$$

*and output throughput is*

$$\Theta_{out} = \frac{\textit{generated tokens produced}}{\Delta t}.$$

**Concept-to-code map**

- **Interval throughput.** The throughput formulas map to `BenchmarkMetrics` fields: `request_throughput` in `python/sglang/bench_serving.py` is completed requests over `dur_s`, `input_throughput` is total prompt tokens over `dur_s`, and `output_throughput` is generated tokens over `dur_s`.
- **Retokenized throughput.** Server-independent output counts map to `total_output_retokenized` and `output_throughput_retokenized`, which are computed from `tokenizer.encode(generated_text)` rather than the server-reported completion length.
- **Burstiness and observed concurrency.** Short-interval pressure maps to `max_output_tokens_per_s`, `concurrency`, and `max_concurrent_requests`:

aggregation buckets token emission and active-request intervals by wall-clock second, while `concurrency` is the sum of end-to-end latencies divided by the measurement duration.

**Definition 2.5.6** (Offered request rate and observed concurrency) *The offered request rate is the rate at which benchmark traffic submits requests. Observed concurrency is the time-average number of in-flight requests during a measurement interval. For interval length $\Delta t$ and $N_{\text{arr}}$ attempted arrivals, the offered rate is*

$$\lambda_{\text{offer}} = \frac{N_{\text{arr}}}{\Delta t}.$$

*If $N_{\text{flight}}(t)$ is the in-flight request count during $[t_0, t_0 + \Delta t]$, observed concurrency is*

$$\bar{N}_{\text{flight}} = \frac{1}{\Delta t} \int_{t_0}^{t_0+\Delta t} N_{\text{flight}}(t)\, dt.$$

The symbol $\lambda$ is not automatically the traffic-generator setting. First choose the population and interval being summarized. In a stable completed-run summary, $\lambda$ is the completed or admitted rate over that interval; in an offered-load study, $\lambda_{\text{offer}}$ must be reported separately and any excess work must appear as queue growth, admission delay, or rejected work.

Compare metrics only after fixing the population, arrival process, and aggregation window.

**Example 2.5.2** (Offered load versus in-flight work) Suppose a benchmark offers 20 requests/s for 10 seconds, so 200 arrivals are attempted. If the server completes 12 requests/s over the measured completed-run interval and each completed request spends 0.5 seconds in the system on average, then the completed-work concurrency is

$$\bar{N}_{\text{flight}} = \lambda_{\text{done}} T_{\text{sys}} = 12 \times 0.5 = 6.$$

The other 8 offered requests/s have not disappeared. If the service accepted all of them while completion stayed at 12 requests/s, the queue is growing and the 20 requests/s offered rate is not a stable Little's-law throughput. If admission control throttled or rejected them, that policy outcome belongs in the run report.

The offered request rate (Definition 2.5.6) is the traffic-generator setting; actual server admission is measured separately. The benchmark driver can submit all requests at time zero, sample interarrival gaps from a Poisson process, replay trace timestamps, or cap client-side concurrency. These modes create different queueing conditions even if the prompt and output token distributions are the same.

When the system is stable and measurements are taken over a representative interval, Little's law (Definition 1.2.1) relates average in-flight concurrency $\bar{N}_{\text{flight}}$, arrival rate $\lambda$, and average time in system $T_{\text{sys}}$:

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

This identity gives a measurement relation for a stable system [24].

**Example 2.5.3** (Throughput and concurrency) If a run processes 300 requests with 300,000 generated tokens in 100 seconds, then output throughput is 3000 generated tokens per second. If the sum of request end-to-end latencies over the same interval is 900 seconds, the observed concurrency is 900/100 = 9.

**Trace state.** The same four-request workload now has user-visible outcome coordinates. B can hurt first-token delay while its prompt is admitted, C can stretch inter-token gaps during long decode, and D can change both measurements if prefix reuse turns a long raw prompt into a short effective suffix.

## 2.6. The Online Serving Decision

Requests, prefill and decode work, resident token state, prefix reuse, and serving metrics now give us the pieces of the scheduler-facing problem. This section turns those pieces into an online decision problem: what state is visible, what action the scheduler may choose, and what makes that action feasible.

The scheduler decides before it knows the future arrival stream.

A single model forward pass has fixed inputs. A serving system does not. It sees a stream of requests over time. After each unit of execution, the visible state changes. Some requests arrived. Some finished. Some generated more tokens. Some consumed additional cache memory. The scheduler must decide again. The definitions below make the four opening questions precise. Waiting work is a request stream. Current resources and bookkeeping form visible state. The selected action must be feasible. The policy chooses among feasible next steps.

**Definition 2.6.1** (Request stream) *A request stream is a finite or countably infinite sequence of request records*

$$\mathcal{J}_{\text{arr}} = (r_i^{\text{arr}})_{i \geq 1}.$$

*Each record $r_i^{\text{arr}}$ becomes visible at arrival time $t_i^{\text{arr}}$, carries an input token sequence $x_i$, and carries request-local rules such as sampling choices and stopping conditions.*

**Example 2.6.1** (Reading the stream notation) The notation says that the serving system receives an ordered list of request records. In the running workload, request A has $t_1^{\text{arr}} = 0$ ms and a 64-token prompt. The second record represents B with $t_2^{\text{arr}} = 5$ ms and a 4096-token prompt; the third represents C with $t_3^{\text{arr}} = 8$ ms and a long expected output; the fourth represents D with $t_4^{\text{arr}} = 12$ ms and a prefix relationship to B. The request-local rules decide how each request samples and when it stops, but at this level they are state that must stay attached to the right request.
The subscript $i$ records the request's position in the arrival stream, so $\mathcal{J}_{\text{arr}}$ is closer to a log of "what arrived, when, and with which local rules" than to a matrix or tensor. A finite stream is enough for a benchmark or running example; a countably infinite stream simply leaves the service open-ended.

The important point is that the serving system must preserve request-specific state while sharing accelerator work across many requests. The next definitions name the policy rule and feasibility test that make this loop explicit.

**Table 2.4.:** Policy signals for online serving decisions. Each signal gives a different answer to scarce prompt-budget allocation.

| Policy signal | Serving interpretation |
|---|---|
| Arrival order | Prefer older visible work when no stronger signal is active. |
| Reusable prefix length | Prefer work that is likely to reuse resident token state. |
| Shared cache subtree weight | Prefer groups of requests that keep cache traversal local. |
| Expected output length | Prefer requests likely to occupy the decode set for longer. |
| Routing affinity | Prefer requests whose routing key matches the active population. |
| Request priority | Allow explicit user or control-plane priority to override the ordinary waiting order, within admission and memory limits. |

**Definition 2.6.2** (Online serving policy) *An online serving policy is a rule*

$$\pi : \mathcal{X}_t \mapsto u_t,$$

*where $\mathcal{X}_t$ is the state visible at time $t$, and $u_t$ is the next serving action. The action may admit prompt work, advance live decode work, delay work, release completed work, or report that no feasible work is ready.*

For a concrete state, $\mathcal{X}_{8\text{ms}}$ might contain pending requests $A, B, C$, resident KV pages for $A$, two available prefill slots, and one decode row already owned by $B$. The policy output $u_t$ is then one operational choice, such as admitting $C$'s prompt, decoding $B$, or waiting because the memory test fails.

**Example 2.6.2** (Reading the policy rule) At $t = 8$ ms, the visible state $\mathcal{X}_t$ includes the fact that A, B, and C have arrived, D has not, some token state may already be resident, and the current prompt and decode budgets have particular remaining capacity.
The policy $\pi$ is the scheduler rule that looks at that visible state and chooses $u_t$, the next action. One policy might choose "admit part of B's prompt," while another might choose "advance the currently live decode requests."
The notation $\mathcal{X}_t \mapsto u_t$ only says that a visible serving state is turned into one serving action; it does not say that the action is globally optimal or that future arrivals are known.

SGLang makes this abstraction concrete with several ordering rules rather than one universal queue discipline. At the Chapter 2 level, these rules can be read as different answers to a resource-allocation question: when not all visible work can run now, which pending request should receive scarce prompt budget first?

Chapter 2 only needs the common shape: each policy supplies an order over visible work before feasibility checks decide what can actually fit.

This is the online part of the formulation. An online serving policy (Definition 2.6.2) may use the current memory state and visible prompt lengths, but it cannot use D's future prefix relationship with B until D has arrived. Online serving therefore optimizes over visible state rather than over a fixed offline batch. That visibility

limit is the operational consequence of the request stream (Definition 2.6.1): future records are not available until their arrival times.

**Definition 2.6.3** (Feasible serving step) *A serving step is feasible if the selected work satisfies all active resource constraints: request slots, prompt-token budget, decode-token budget, fast-memory capacity, model compatibility constraints, and output bookkeeping constraints. Equivalently, for visible state $\mathcal{X}_t$, an action $u_t$ is feasible when $u_t \in \mathcal{U}_{\text{feas}}(\mathcal{X}_t)$; the set $\mathcal{U}_{\text{feas}}$ is defined by those resource and compatibility predicates.*

Feasibility is the legal-action boundary. It decides which serving actions the runtime may take; latency, throughput, and fairness decide which legal action is worth taking.

**Example 2.6.3** (Feasibility as a resource check) Suppose a serving step has room for 2048 uncached prompt tokens and two decode request-positions. Admitting all 4096 uncached prompt tokens of B in that step is not feasible under the prompt budget. Admitting 1024 of B's prompt and advancing A by one generated token may be feasible, but only if the resulting token state also fits in fast memory and the output bookkeeping can be maintained. Feasible means "allowed by the current constraints," not necessarily "the best possible choice."

At the 12 ms decision in the running A/B/C/D trace, make every predicate visible at once. The four independent gates are: one free request row, the uncached prompt-token budget $Q$, the decode-row budget, and the fast-memory ledger $C_{\text{fast}}$.

Suppose the request-row table can hold four live rows; A and B are already live, so either C or D can be admitted without exceeding that row cap. The prompt budget is $Q = 256$ uncached tokens, the decode budget is two request-positions, and the fast-memory budget is $C_{\text{fast}} = 5000$ logical token-state slots. A and B already account for 4162 slots. C has a 256-token prompt and a 120-token future output reserve, so its admission charge is 376 slots and $4162 + 376 = 4538 \leq 5000$.

D has 1024 raw prompt tokens, but its 900-token prefix hit leaves a 124-token suffix and a 16-token output reserve. The 144 below is not a new prompt length: the 124 uncached tokens round up to 128 page slots, then the 16-token decode reserve is added. With the page-aware charge from Section 2.7, the 124-token suffix rounds to 128 slots, so D's charge is 144 slots and $4162+144 = 4306 \leq 5000$. Admitting both prompt slices would need $256 + 124 = 380$ uncached prompt tokens, which violates $Q = 256$ even though the memory ledger would still fit. This is why feasible means all predicates pass at the same timestamp, not that a request looks small on one axis.

**Local notation ledger.**

Chapter 2 introduces several state, work, memory, and metric symbols because the serving problem crosses all four surfaces. Table 2.5 keeps the common symbols in one place. Later chapters may reuse the same letter for a narrower local account, so this table should be read as Chapter 2's serving-problem scope rather than as a global symbol declaration.

**Table 2.5.:** Local notation for the Chapter 2 inference-serving problem. The table groups request state, token-state memory, prefix reuse, latency metrics, and online-policy notation before later chapters specialize those accounts.

| Symbol | Meaning in Chapter 2 | Role in the serving problem |
|---|---|---|
| $R_i(t)$ | Request state for request $i$ at serving time $t$ | The evolving record that keeps input, output, finish state, token-state ownership, and output progress attached to one request. |
| $x_i, y_i(t)$ | Input token sequence and generated suffix | Visible request content separated into fixed prompt and growing output. |
| $Z_{i,j}$ | Resident token state for request $i$, position $j$ | The persistent model state that future decode steps may read. |
| $C_{\text{fast}}$ | Fast-memory token-state budget | The bounded resident-memory capacity measured in the active cache-layout units. |
| $L_t, N(B_t)$ | Live resident state and new state written by selected batch | The two sides of the memory-feasibility check. |
| $\ell_i^{\text{unc}}$ | Uncached suffix length after prefix reuse | The prompt work still owed after reusable prefix state is credited. |
| TTFT, ITL, $\tau_i^{\text{e2e}}$ | First-token, inter-token, and end-to-end latency quantities | User-visible timing surfaces that policy and measurements report separately. |
| $\mathcal{I}_{\text{arr}}$ | Request stream | The arrival log visible to an online serving system over time. |
| $\mathcal{X}_t$ | Serving state visible at time $t$ | The information the online policy may use when choosing a step. |
| $u_t, \mathcal{U}_{\text{feas}}(\mathcal{X}_t)$ | Chosen action and legal action set | The selected serving action and the feasibility boundary imposed by resources and compatibility constraints. |

**Definition 2.6.4** (Inference serving problem) *Given a request stream and finite serving resources, the inference serving problem is to choose an online serving policy that repeatedly maps visible serving state to feasible serving steps, so*

$$u_t = \pi(\mathcal{X}_t) \in \mathcal{U}_{\text{feas}}(\mathcal{X}_t)$$

*at each decision point. Each step must preserve per-request state, admit or delay prompt work, advance live decode work, release completed work, and maintain output bookkeeping while respecting request-slot, token-budget, compatibility, and fast-memory constraints.*

This definition separates the legal-action set from the serving objective. Feasibility says which actions are legal in the current state; it does not rank legal actions by a single scalar reward. A policy that always idles illustrates the gap: it may avoid violating a resource constraint, but it is not a useful operating point once the objective includes completed work and latency. Production policies choose among the metrics in Section 2.5: TTFT, ITL, end-to-end latency, throughput, fairness, and memory pressure can trade against one another. Later measurements therefore compare policies and workloads by the operating points they induce, rather than pretending that every serving deployment optimizes the same objective.

**Example 2.6.4** (One iteration of the serving problem) At time $t$, the visible state contains A as a live decode request, B as a pending 4096-token prompt, and finite prompt, decode, request-slot, and KV-memory budgets. A feasible action may advance A by one decode row and admit a prefix of B's prompt. After model execution and result processing, the visible state changes: A has a longer sequence, B has less pending prompt work, and the memory ledger has different committed and reserved KV state. The policy must make the next decision from that new state, without using future requests that have not arrived yet. Figure 2.13 draws that repeated online loop.

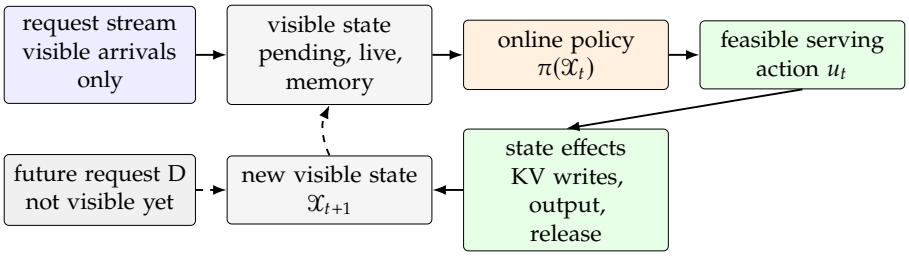



**Figure 2.13.:** The inference serving problem as a repeated online decision. The policy can use only currently visible state to choose a feasible action; model execution and result processing then mutate that state before the next decision, while future arrivals remain unavailable until they enter the stream.

This is the scheduler-facing problem: SGLang must choose queued prompt work, live decode work, memory actions, and prefix-reuse opportunities using only visible state, then judge those choices by benchmark-visible latency and throughput. The next section applies that frame to dynamic batching, admission, and backpressure; the empirical and history sections then calibrate and contextualize the model.

**Trace state.** At 8 ms the runtime can reason about A, B, and C, but D's prefix relationship to B is still future information. At 12 ms D becomes visible, and the same policy must decide whether D's cheap suffix is worth admitting before C's long decode lifetime receives more service.

## 2.7. Constructing Feasible Serving Steps and Invariants

The online decision model becomes operational only when the runtime can decide which work is eligible for the next step. This section handles the admission side of that construction: dynamic batching names the per-step work set, admission control decides which waiting prompt work enters that set, and backpressure records why ready work may still have to wait. The next section uses this admitted/pending/rejected partition to assemble the executable model step.

Table 2.6 maps those roles to the concrete source-facing distinctions used below.

### Dynamic Batching

Static batching asks the system to collect a fixed group of requests and process them as a unit. That model is poorly matched to online serving. Online serving

Dynamic batching trades uniformity for utilization.

**Table 2.6.:** Navigation map for feasible-step construction. This section keeps admission feasibility separate from the final executable step so that prompt pressure, decode progress, and policy refusal remain distinguishable before Section 2.8 chooses the model-work return.

| Part of the construction | Main decision | Local output | What can still block |
|---|---|---|---|
| Dynamic batching | Which pending and live work items should be considered together at one step. | A serving dynamic batch $B_t$ of prompt slices, decode rows, or both. | Head-of-line rules and compatibility predicates. |
| Admission and backpressure | Which waiting prompt slices can enter without violating budgets or reserves. | Admitted, pending, and rejected prompt-work sets $(A, H, R)$. | Prompt-token budget, page-aligned KV reserve, queue limits, or policy rejection. |
| Partial admission | Whether a long prompt can make prefix progress without admitting the whole remaining prompt. | A strict prompt prefix admitted now and a suffix left pending. | Latency tradeoffs between the long request and already-live decode work. |
| Step return | Whether the executable step is mixed, prefill-only, decode-only, or idle. | A concrete model-work action for the next serving boundary. | Decode memory pressure, mode flags, or an empty feasible set; handled in Section 2.8. |

has several independent sources of change. Requests arrive while other requests are already generating. Some finish early. Some carry long prompts, decode for many steps, or share prefix state. Scheduling predicates decide whether work items can share a step. Orca framed the same generative-serving mismatch as a need for iteration-level scheduling and selective batching [48].

**Definition 2.7.1** (Serving dynamic batch) *A serving dynamic batch is the scheduler-selected collection of compatible work items for one serving step from currently pending and live requests. When lowered to model execution, that collection becomes the Chapter 1 dynamic batch of scheduled positions (Definition 1.3.4). It may contain prompt work, decode work, or a controlled mixture, depending on the serving policy and resource state. We write the selected serving batch at step $t$ as $B_t$.*

The dynamic batch is a per-step selection whose membership can change at every boundary as requests arrive, finish, become blocked, or become newly cheap through reuse. In the running example, the scheduler may admit A and begin decoding it before B's prompt is fully processed. C may stay in the active population after A and B complete. D may become cheap to admit if its shared prefix is reusable. A fixed batch would make these cases interfere with each other more than necessary.

**Definition 2.7.2** (Head-of-line blocking) *Head-of-line blocking occurs when one request or one class of work prevents other feasible work from running, even though the blocked work could make useful progress under the current resource constraints. In queue notation, a head item $h$ blocks a ready item $i$ when $h \notin \mathcal{U}_{\text{feas}}$, $i \in \mathcal{U}_{\text{feas}}$, and the policy forbids selecting $i$ before $h$.*

**Example 2.7.1** (Long prompt at the head) Suppose the prompt-token budget for the next serving action is 2048, request B is first in a strict queue with 4096 uncached prompt tokens, and requests A and C each need one decode row. If the policy insists that B must be admitted as one indivisible operation before any later work can run, B does not fit and A and C still wait. The blocked decode rows are feasible under the decode budget, but the head request prevents them from making progress. Figure 2.14 contrasts that strict head-only queue with a dynamic-batching choice that admits feasible work.

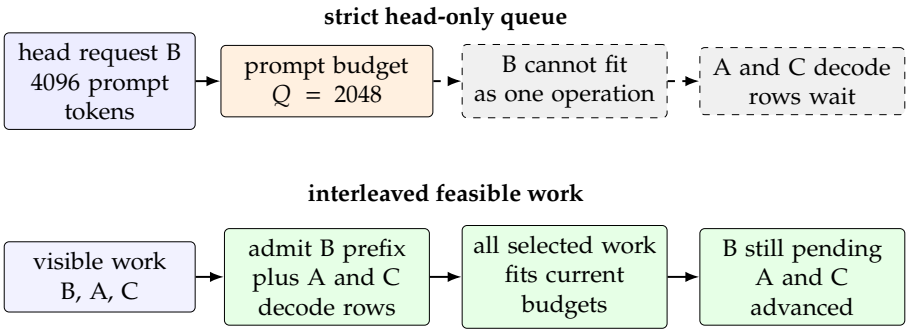



**Figure 2.14.:** Head-of-line blocking under an indivisible head request. A strict head-only rule can leave feasible decode work idle behind a prompt that does not fit, while dynamic batching can select a feasible prefix and independent decode rows without pretending the whole prompt was admitted.

Dynamic batching does not remove the cost of B's prompt, but it gives the serving policy room to interleave feasible work.

SGLang adds one more wrinkle: scheduling and execution need not be perfectly lockstep. In the overlap path, the scheduler can prepare the next batch while the previous batch's GPU work or sampling result is still being resolved. The Chapter 2 model of discrete serving steps remains useful, but a real step may carry placeholders for future token ids and delayed result processing. This is why later chapters distinguish the logical batch chosen by the policy from the exact CPU/GPU timing of when its tensors and outputs become visible.

The dynamic-batch account names the shape of a serving step. The next gate is admission: deciding which pending prompt slices may join that step without violating token budgets, memory reservations, compatibility constraints, or latency-protection policy.

## Admission Control and Backpressure

Admission control decides which waiting prompt work becomes active model work. Backpressure is what happens when ready work cannot enter because some resource is exhausted or reserved for other work.

Backpressure protects latency by refusing work the system cannot carry honestly.

**Definition 2.7.3** (Pending request) *A pending request is visible to the serving instance but has not yet had its next required prompt work admitted to model execution. Let $W_t$ denote the pending set visible at serving step $t$.*

A pending request is therefore observable but not yet executable. The next definition names the scheduler choice that moves some, all, or none of that waiting prompt work into a concrete serving step.

**Definition 2.7.4** (Admission decision) *An admission decision maps pending prompt work to a serving step. The decision may admit a whole request prefix, admit only part of the remaining prompt, or admit nothing from that request. With remaining uncached prompt length $\ell_i^{\mathrm{rem}}(t)$ for request $i \in W_t$, write the decision as*

$$a_t : W_t \to \mathbb{N}, \qquad 0 \le a_t(i) \le \ell_i^{\mathrm{rem}}(t).$$

An admission decision can also be a refusal to run ready work. The backpressure definition below gives that refusal a resource meaning rather than treating it as arbitrary delay.

**Definition 2.7.5** (Backpressure) *Backpressure is the state in which work is ready but infeasible under current resource constraints. The limiting resource may be request slots, prompt-token budget, decode-token budget, fast-memory capacity, output bookkeeping, or a compatibility constraint. As in the memory-pressure definition, ready means the request-local prerequisites are satisfied before resource feasibility is applied. In set notation, backpressure exists when the ready set $W_t^{\mathrm{ready}}$ is nonempty but the feasible subset $W_t^{\mathrm{ready}} \cap \mathcal{U}_{\mathrm{feas}}(\mathfrak{X}_t)$ is empty under the current constraints.*

B makes backpressure concrete. Nothing is merely waiting: the scheduler is choosing which latency it is willing to hurt. B is a pending request (Definition 2.7.3) until its next prompt work is admitted. If a serving step can admit at most 2048 uncached prompt tokens, B cannot enter as one full prompt operation. The choice for B is an admission decision (Definition 2.7.4): admit no prompt tokens from B in this step or admit a feasible prefix. If B is not admitted, the batch-level choice is whether the step should be spent on other feasible work, such as short prompts or decode tokens from already-live requests. These choices change latency. Holding B back may improve short-request responsiveness. Admitting a portion of B may reduce B's time to first token. Spending the step on existing decode work may keep already-live requests progressing smoothly.

**Example 2.7.2** (Prompt-budget admission) Suppose the prompt-token budget for a step is $Q = 2048$, and request B has $r_B = 4096$ uncached prompt tokens remaining. A feasible admitted amount $x_B$ must satisfy

$$0 \le x_B \le \min(r_B, Q) = 2048.$$

Choosing $x_B = 0$ delays B. Choosing $x_B = 2048$ partially admits B and leaves 2048 uncached prompt tokens for a later step. The same feasibility inequality applies to any pending request, but the latency consequences depend on the current live set and memory state. Figure 2.15 draws the same admission choice as a resource gate: B may enter partially, remain pending, or leave room for other feasible work.

SGLang uses this prompt-budget view as only one part of the admission test. Admission is also a reservation decision: a request may need page-aligned prompt allocation, an estimate of future generated tokens, and small allocator overhead before it is safe to admit. Chapter 5 derives the reusable page-aligned admission charge; here the important serving-level point is that feasible prompt work is selected under predictive memory reservation rather than raw prompt-token

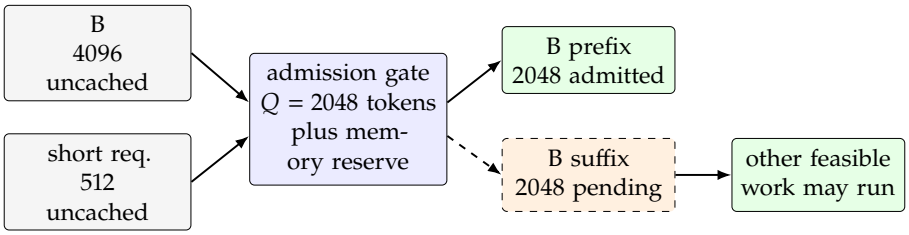



**Figure 2.15.:** Admission control as a resource gate. A long prompt may be partially admitted, while its remaining suffix stays pending under backpressure so the step can still run feasible work without pretending the whole prompt fit.

Admission charges pages, future reserve, and headroom.

capacity alone. The reserve protects the next decode step: admitting prompt work that fits this step but leaves no room for later decode would only move the failure to the next step.

For a pending request $i$, let $q_i$ be the candidate prompt slice still visible to admission after resident prefix reuse, $h_i$ any host-cache portion inside that slice that will be loaded rather than newly allocated, $P$ the allocator page size, $\rho_i$ the future-output reserve still needed after already emitted output tokens are subtracted, and $H_P$ the per-request page headroom. Read the charge as a three-line ledger before reading it as one formula:

| | | |
|---|---|---|
| new uncached suffix slots | $\mathrm{alloc}_P(\max(q_i - h_i, 0))$ | round the current slice to pages, |
| future decode reserve | $\rho_i$ | protect likely generated tokens, |
| allocator headroom | $H_P$ | cover page slack and bookkeeping. |

The predictive admission charge used by the serving-level account is

$$d_i = \mathrm{alloc}_P\big(\max(q_i - h_i, 0)\big) + \rho_i + H_P, \qquad \mathrm{alloc}_P(n) = P\left\lceil \frac{n}{P} \right\rceil.$$

The first term is the page-aligned prompt allocation for real uncached work. The second term reserves future decode state so the request can keep generating after prefill. The final term accounts for allocator headroom, such as the extra page SGLang budgets around extend allocation. A request passes the memory side of admission only when this charge fits the currently visible capacity after resident decode reserves have already been debited.

The reserve exists because output length is not known at admission. A request may have a maximum-output budget and an expected length, but its actual decode lifetime is revealed only as generation proceeds and stopping conditions fire. The scheduler therefore admits prompt work using a conservative estimate of future token-state writes. If the estimate proves too small under later live-state pressure, the system needs the memory-pressure mechanisms from Section 2.3: delay more work, retract decode rows, or abort when even the minimal feasible decode step cannot reserve memory. In this sense, reservation and retraction are two sides of the same uncertainty: one guesses enough future capacity, the other repairs the schedule when the guess no longer fits.

Algorithm 2 uses a few compact account names rather than new subroutines. Refreshing queue limits means recomputing the current prompt budget, active and waiting caps, future decode reserve, and free KV capacity from $\mathfrak{X}_t$. Prefix-cache hits mean querying the active prefix or host-cache state for each valid reusable prefix length $p_i$, then charging only the remaining suffix. The algorithm keeps those accounts visible because stale limits or stale hit lengths are exactly how admission admits work that no longer fits.

**Algorithm 2:** Admission control with backpressure for one scheduling step. The routine admits only prompt slices whose token budget, KV reserve, and compatibility checks are already feasible; otherwise work remains pending or is rejected by policy.

**Input:** Pending requests $W$; prompt-token budget $Q$; available capacity $C_{\text{avail}}$; page size $P$; future decode reserve estimates $\rho_i$; compatibility predicate $\chi$; policy order $\prec_\pi$
**Output:** Admitted prompt slices, still-pending work, and rejected or aborted work

```
read current queue limits, live decode reserve, reusable-prefix hits, and free KV capacity from the visible serving state;
initialize admitted set A, pending set H, and rejected set R;
foreach request i in order ≺_π over W do
    compute uncached remaining prompt work ℓ_i^rem after reusable prefix or host hits;
    choose a candidate slice q_i ≤ min(ℓ_i^rem, Q);
    estimate predictive admission charge d_i from page-aligned prompt allocation, future reserve, and allocator headroom;
    if q_i > 0, d_i ≤ C_avail, and χ(i) passes then
        admit slice q_i, reserve d_i, and reduce Q and C_avail;
        append i to A;
    else if queue or policy limits say i can wait then
        keep i's remaining work in H;
    else
        reject or abort i and append it to R;
return A, H, R;
```

**Concept-to-code map**

- **Future-output reserve.** The reservation term maps to `PrefillAdder._get_running_request_total_token_offset` and `PrefillAdder.add_one_req` in `python/sglang/srt/managers/schedule_policy.py`: both cap remaining `Req.sampling_params.max_new_tokens` by `CLIP_MAX_NEW_TOKENS` after subtracting already emitted `Req.output_ids`.
- **Page-rounded admission charge.** The charged prompt and output budget maps to `PrefillAdder.ceil_paged_tokens` and `PrefillAdder._update_prefill_budget` in the same file, where `extend_input_len`, reserved output tokens, and one page of allocation overhead update the remaining token and prompt budgets.
- **Effective prompt work after cache reuse.** Reused and charged prefix lengths map to `Req.host_hit_length`, `Req.prefix_indices`, and `Req.extend_input_len`: `Scheduler._prefetch_kvcache` in `python/sglang/srt/managers/scheduler.py` computes the host-loadable prefix, and `PrefillAdder.add_one_req` subtracts that hit before charging real input-token budget.

Algorithm 2 turns the resource gate into an online decision procedure. It scans waiting requests in the $\prec_\pi$ order supplied by the active policy, measures reuse, chooses a candidate slice, and charges page-aligned demand before mutation. Finally, it admits feasible work, keeps waitable work pending, or rejects or aborts requests that cannot wait before returning the three-way partition. The ordering matters: once a slice is admitted, the scheduler has promised both current prompt execution and enough state to keep the request live afterward.

| Time | Visible state | Admission arithmetic | Consequence |
|---|---|---|---|
| 0 ms | A arrives. | A's short prompt is the only prompt work. | A becomes live. |
| 5 ms | B arrives while A can decode. | B is larger than one prompt budget, so only a prefix can be admitted. | A's decode row can share the step with B's admitted prefix. |
| 8 ms | C arrives while B still owns a pending prompt suffix. | C's charge is not evaluated alone; the scheduler first decides whether B's owned suffix should continue. | C's TTFT clock starts before C is admitted. |
| 12 ms | A and B are live; C and D are both visible. | C costs $d_C = 376$ with no prefix hit; D costs $d_D = 144$ after B's reusable prefix is credited. | Ordering C first protects C's TTFT but consumes the prompt budget; ordering D first exploits reuse and delays C. |

**Table 2.7.:** Wall-clock trace of admission arithmetic. Each row shows visible state, feasibility math, and the resulting consequence.

**Example 2.7.3** (Admission trace at 12 ms) Use the 12 ms state from the running workload. A and B are live, the prompt budget is $Q = 256$, available fast-memory capacity after A and B is $C_{\text{avail}} = 838$ slots, and the policy compares C with D. C has no prefix hit, so $\ell_C^{\text{rem}} = 256$; its candidate slice is $q_C = 256$. With $P = 16$, no host hit, zero extra page headroom in this scaled logical ledger, and $\rho_C = 120$,

$$d_C = \text{alloc}_{16}(256) + 120 + 0 = 376.$$

D has a 900-token prefix hit against B, so $\ell_D^{\text{rem}} = 1024 - 900 = 124$; its candidate slice is $q_D = 124$, and with $\rho_D = 16$,

$$d_D = \text{alloc}_{16}(124) + 16 + 0 = 128 + 16 = 144.$$

If the logical ledger ignores page rounding, this is the earlier $124 + 16 = 140$ slot mental shortcut; the page-aware admission test must use 144.
If $\prec_\pi$ orders C before D, the algorithm admits C, reduces $Q$ to zero, reduces $C_{\text{avail}}$ to 462, and leaves D in $H$ even though D's memory charge would fit. The binding resource is the prompt-token budget. If a prefix-aware policy orders D first, the algorithm admits D, leaves $Q = 132$, and can at most partially admit C's remaining prompt. The same visible state therefore yields different admitted and pending sets depending on the policy order, but both outcomes are constrained by the same arithmetic.

Reading the same trace forward in wall-clock time makes the cost of that arithmetic visible:

**Definition 2.7.6** (Partial admission) *Partial admission is admission of a strict prefix of the remaining prompt work for a request, leaving the rest of that request's prompt work pending for a later serving step. In the admission notation above, request $i$ is partially admitted exactly when*

$$0 < a_t(i) < \ell_i^{\text{rem}}(t).$$

**Example 2.7.4** (Strict-prefix admission) Request B has 4096 uncached prompt tokens remaining, but the current prompt-token budget is 2048. Admitting tokens 0:2047 is partial admission: the admitted slice becomes current model work, while tokens 2048:4095 stay pending under the same request identity. Admitting all 4096 tokens would be full admission, and admitting zero tokens would be backpressure without partial admission. Figure 2.16 shows the split between the admitted prefix and the pending suffix under one request identity.

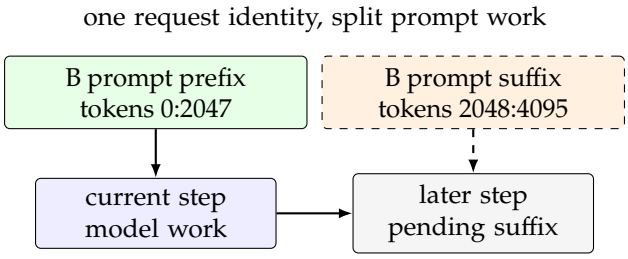



**Figure 2.16.:** Partial admission admits a strict prefix of remaining prompt work. The admitted prefix becomes current model work, while the suffix remains pending under the same request identity for a later serving step.

Partial admission (Definition 2.7.6) is the first sign that serving is a token-work management problem. The scheduler is really managing units of token work under memory and execution constraints. This prompt-slicing view is closely related to chunked-prefill serving designs that mix prompt chunks with decode work [1].

After admission control, the scheduler has not yet chosen a model step. It has only classified prompt work into admitted, pending, and rejected sets. The next question is whether admitted prompt slices can share the execution boundary with decode rows, or whether the runtime should return a prefill-only, decode-only, or idle action.

**Trace state.** B is the partial-admission case, A and C are the decode rows that should not sit idle behind an indivisible prompt, and D is the request whose admission answer changes once prefix reuse has made its suffix small.

## 2.8. Assembling the Next Serving Step

Admission control produces a partition of prompt work. The runtime still has to turn that partition, the live decode set, and the current memory state into one executable serving action. This section is the return boundary for the Chapter 2 serving problem: it decides whether the next action is mixed, prefill-only, decode-only, or idle.

The assembled step is valid only when scheduler choice, memory reservation, and output state agree.

SGLang implements these step-construction decisions with many specialized branches for cache layout, chunking, overlap, and distributed modes. Algorithms 3 and 4 split the path into two teachable contracts. First, prompt and decode worksets must be brought current and made feasible. Then the final action selector chooses mixed, prefill-only, decode-only, or idle without changing the feasibility ledger.

The helper-like phrases in the algorithm are local accounts already defined in the chapter, not extra APIs. Reusable-prefix state is the prefix-hit account from Section 2.4; chunk limits are partial-admission bounds; memory feasibility is the live-plus-new token-state check from Section 2.3.

**Algorithm 3:** Preparing feasible prompt and decode worksets for one online serving step. Prefix reuse, prompt admission, rejection handling, and decode fallback all finish before the action selector sees the candidate work.

**Input:** Waiting requests $W$; live decode set $\mathscr{L}_{\text{dec}}$; prefix cache $\mathscr{C}_{\text{pref}}$; budgets and mode flags $\mathscr{U}$
**Output:** Prompt worksets and feasible decode rows for the next action selector

refresh arrivals, promote completed prompt rows to decode, remove finished rows, and order $W$ by policy and reusable-prefix state;
compute prefix-hit lengths from $\mathscr{C}_{\text{pref}}$, cache tiers, and in-batch prefix opportunities;
$(A, H, R) \leftarrow$ Algorithm 2 using the prefix-hit lengths, chunk limits, prompt budget, and memory budget;
apply the rejection or abort side effects recorded in $R$; keep $H$ waiting;
$\mathscr{D} \leftarrow \emptyset$;
**if** $A = \emptyset$ *or mixed work is permitted* **then**
    $\mathscr{D} \leftarrow$ the feasible subset of $\mathscr{L}_{\text{dec}}$; retract or delay live work until a smaller decode batch is feasible;
**return** $(A, H, R, \mathscr{D})$;

**Algorithm 4:** Selecting the serving action after prompt and decode worksets are feasible. The selector is mixed only when both work forms are present and the mode permits sharing one forward batch.

**Input:** Admitted prompt work $A$; feasible decode rows $\mathscr{D}$; mixed-work mode flag
**Output:** Next serving action

**if** $A \neq \emptyset$, $\mathscr{D} \neq \emptyset$, *and mixed work is permitted* **then**
    **return** *mixed batch with prompt slices* $A$ *and decode rows* $\mathscr{D}$;
**if** $A \neq \emptyset$ **then**
    **return** *prefill batch* $A$;
**if** $\mathscr{D} \neq \emptyset$ **then**
    **return** *decode batch over* $\mathscr{D}$;
**return** *idle*;

Algorithm 3 is the state-refresh half of the serving step. First, completed and finished rows update the live sets. Then prefix-aware admission partitions waiting prompt work while rejection side effects are applied immediately. Finally, live decode rows are included only when they fit beside the admitted prompt slices or when prompt work is absent.

Algorithm 4 is intentionally small because all capacity mutation has already happened. First, mixed execution wins only when prompt and decode work are both feasible. Then prompt-only and decode-only actions preserve useful work without inventing new rows. Finally, the idle return means neither prepared workset can run. Figure 2.17 draws the same mixed, prefill-only, decode-only, or idle return as a decision tree.

**Example 2.8.1** (Retraction as a capacity ledger) Use a scaled-down capacity ledger with ten token-state slots. A and B already own eight slots that must remain protected for their next decode reads. The candidate mixed step tries to add C's admitted prompt slice, which would reserve five more slots. The decode-memory check sees $8 + 5 = 13 > 10$, so the branch "retract or delay live work" is not a vague fallback: C's newly admitted slice is removed from the executable step and returned to the pending prompt set.
After that retraction the protected live set is $\{A, B\}$, its eight slots fit under the ten-slot capacity, and the runtime can still return a feasible decode batch for A and B. If chunking policy allows C to be reduced to a two-slot prefix instead of removed entirely, the same branch returns the smaller mixed batch $\{A, B, C_{0:1}\}$, whose ledger is $8 + 2 = 10$. In both cases the observable operation is concrete: rows disappear from the executable batch before the model runner receives it, while the request identity remains pending rather than losing its state.

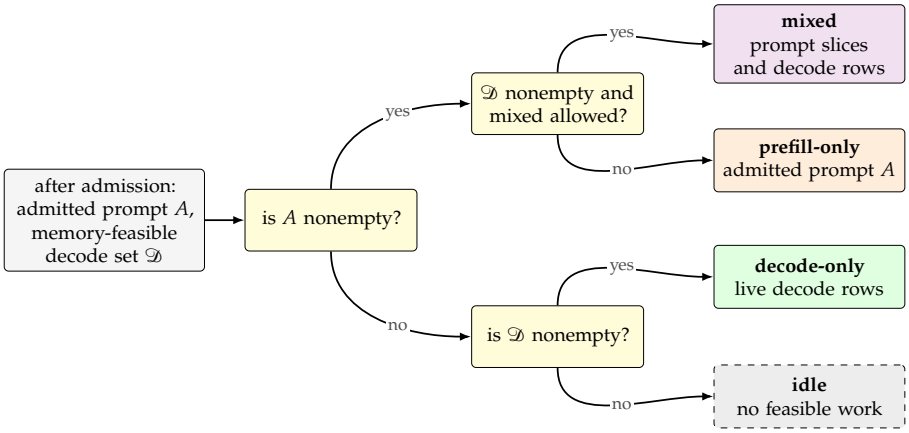



**Figure 2.17.:** Assembling the next serving step as a decision tree. After admission classifies prompt work, the runtime branches on whether admitted prompt work $A$ and a memory-feasible decode set $\mathcal{D}$ exist, and whether mixed work is allowed, to return one of four executable actions: mixed, prefill-only, decode-only, or idle. This is the branching structure of Algorithm 4.

Table 2.8 summarizes the concrete rows used by the following discussion.

A short trace shows how the definitions above compose into a serving policy. Table 2.9 uses the running workload from the opening section and aligns decision instants with the four arrivals for compactness. The scaled-down instance has a 2048-token uncached prompt budget, two decode positions per step, and enough fast-memory capacity for the listed logical token-state ledger. Page rounding and future-output reserves from the admission inequality would add implementation charges on top of these logical token counts.

This table is a compact ledger for one feasible policy. Its value is the contrast it makes possible. A prefix-aware policy could swap the 12 ms admission choice, admitting D's uncached suffix after B's prefix state is resident and delaying C instead. That choice would lower D's prompt work and possibly D's TTFT, while increasing C's waiting time. If the fast-memory budget were below 4420 logical token states, the same 12 ms row would also be the first point where the policy must trade C admission against already-live decode progress or apply memory-pressure controls.

Backpressure (Definition 2.7.5) can also become visible before a batch is formed. SGLang has paths for rejecting work when the waiting queue is full, aborting work that has waited too long, and letting priority scheduling displace lower-priority queued or running work. A complete serving model therefore has three layers of

| Algorithm phase | State it reads or mutates | Reason it is separate |
|---|---|---|
| Boundary reconciliation | Moves completed prompt work into the live decode set, removes finished decode work, and refreshes visible arrivals. | Feasibility must start from the current live state, not from the previous step's selected work. |
| Prompt admission | Delegates reusable-prefix accounting, chunk limits, page-rounded reserve, and policy rejection to Algorithm 2. | Prompt feasibility can delay, partially admit, or reject work before decode selection is considered. |
| Decode selection | Adds live decode rows only when mixed work is allowed or no prompt slice was admitted, and retracts or delays decode work when memory does not fit. | Decode progress is a latency objective, but it still competes for resident token state. |
| Executable return | Chooses mixed, prefill-only, decode-only, or idle. | This is the operational return of Definition 2.6.4: one visible state becomes one feasible model-work action for later chapters to lower into runtime and batch objects. |

**Table 2.8.:** Recap of Algorithm 4. The phases keep request-state updates, prompt admission, decode progress, and the final model execution boundary distinct.

| Time | Visible pressure | Arrival action | Step work | Resident ledger after step | Metric effect |
|---|---|---|---|---|---|
| 0 ms | A arrives; no live decode work. | Prefill A. | Admit A's 64 prompt tokens; advance no decode rows. | $A : 64$, total 64. | A becomes live, but has not emitted a token. |
| 5 ms | A is live; B arrives with a 4096-token prompt. | Run a mixed step for A and B. | Admit B tokens 0:2047; advance A by one decode token. | $A : 65$, $B : 2048$, total 2113. | A gets first output; B remains partially admitted. |
| 8 ms | A is live; B has a pending suffix; C arrives. | Continue B before admitting C. | Admit B tokens 2048:4095; advance A by one decode token. | $A : 66$, $B : 4096$, total 4162. | C's TTFT clock keeps growing while B's prompt becomes resident. |
| 12 ms | A and B are live; C and D are visible. | Admit C before D. | Admit C's 256 prompt tokens; advance A and B by one decode token each. | $A : 67$, $B : 4097$, $C : 256$, total 4420. | B gets first output; D waits despite its reusable prefix opportunity. |

**Table 2.9.:** Four arrival-order serving decisions for the Chapter 2 running workload. The table uses logical resident token-state counts to show how prompt admission, decode work, and latency consequences accumulate across steps.

**Table 2.10.:** Metric consequences of the 12 ms serving choice. Feasibility facts determine which latency or throughput metric changes.

| 12 ms choice | Feasibility fact | Metric consequence |
|---|---|---|
| Admit C first | C's 256 prompt tokens fit with A and B's decode rows. | C's TTFT starts sooner; D's reusable-prefix opportunity waits. |
| Admit D's uncached suffix first | B's resident prefix can make D cheaper than its raw prompt length suggests. | D's TTFT may fall, while C waits behind prefix-aware reuse. |
| Delay admission or retract | The resident-token budget cannot hold the new prompt plus protected decode progress. | Throughput is sacrificed to keep live decode rows feasible and avoid allocator failure. |

refusal: a request may be delayed because it is feasible later, partially admitted because only a prefix fits now, or aborted because the control-plane policy decides the queueing contract has already failed.

## 2.9. Empirical Anchors for the Problem Model

Scenario benchmarks anchor the serving model by asking whether the variables from the running trace move the measured surface in the expected direction. The model makes three concrete predictions. First, prompt-heavy and decode-heavy workloads should separate input-token throughput from output-token throughput. Second, active-request caps and memory pressure should turn resident state into queueing and first-token delay. Third, prefix reuse and arrival mode should change effective prompt work even when raw prompt tokens look similar.

A serving benchmark is incomplete without the arrival process and length mix.

The useful result is simple: workload shape, resident pressure, arrival rate, and prefix reuse move different coordinates of the serving surface. The measurements below connect those predictions to input and output throughput (Definition 2.5.5), time to first token (Definition 2.5.2), inter-token latency (Definition 2.5.3), and end-to-end latency (Definition 2.5.1).

The measurement set uses Jetson Thor devices, each with one NVIDIA Thor GPU. The devices served `Qwen/Qwen2.5-0.5B-Instruct` [39, 38] with NVIDIA's SGLang container package `0.5.5.post2`. Gemma remains the running model for shape arithmetic and source-grounded examples; the Thor/Qwen rows are a separately archived empirical slice used only to calibrate the serving axes on one measured package-and-hardware setup. Source-level SGLang claims use the edition-level SGLang release named in the preface; the Thor benchmark rows identify the measured runtime by NVIDIA's SGLang container package `0.5.5.post2`, and some Chapter 2 and Chapter 3 logs did not capture an additional container-side Git commit.

The supplementary coverage artifact for this slice is `RESULT_COVERAGE.md` in the Volume I Thor experiments directory. It records the scenario status, the Chapter 1–3 and Volume I run families behind the summarized rows, and the blocked or supplemental rows omitted from the printed tables. In the figures and experiment harness, a workload label such as 8192 → 1 means an 8192-token random input target and a one-token output target; $N \times I/O$ means $N$ requests with input-token target $I$ and output-token target $O$. When a table reports a

device mean, the underlying rows are completed runs over the included Thor devices unless an evidence-scope column says otherwise.

**Table 2.11.:** Results at a glance for the Chapter 2 Thor serving scenarios. Values are device means across the shared Thor/Qwen setup defined in this section; the rows are small-model calibration points, not a cross-hardware serving law.

| Question | Mean measured result | Interpretation for this discussion |
|---|---|---|
| Throughput-seeking versus paced serving | Offline default: 3.29k output tok/s and 7.32 s median E2E; online default: 139 output tok/s and 1.15 s median E2E. | The $\approx 23.7\times$ output-throughput gap compares a throughput-seeking offline point with a paced online latency point, not per-token model speed. |
| Active-request budget | Streaming A/B: 4.10k versus 1.05k output tok/s; 0.77 s versus 6.60 s median TTFT. | An active-request cap turns resident-work pressure into first-token delay under a matched workload shape. |
| Prompt-heavy versus decode-heavy work | Prompt-heavy serving reaches 46.10k input tok/s and 4.84 output tok/s; decode-heavy serving reaches 6.18k output tok/s. | Input-token and output-token throughput answer different questions, so a single "throughput" number hides the serving regime. |
| Shared-prefix reuse | Radix cache on/off: 5.78k versus 3.11k output tok/s; 2.80 s versus 5.24 s median E2E. | Prefix reuse changes the workload presented to the scheduler and belongs in the work account. |
| Shared-prefix scheduling | `lpm` (longest-prefix-match) versus `fcfs` (first-come, first-served): 5.87k versus 5.74k output tok/s; 2.75 s versus 2.82 s median E2E. | Policy effects are measurable, but this row is a workload-conditioned contrast rather than a global ordering of scheduling policies. |

Figure 2.18 separates the measurement set by input-token and output-token throughput. This matters because prefill-heavy and decode-heavy workloads stress different axes of the serving problem: a large prompt can make input-token throughput high while output-token throughput is nearly absent, and long generations can do the opposite.

Figure 2.19 breaks the same serving runs into the latency quantities from Definitions 2.5.1 through Definition 2.5.4. The original small-active-budget row is retained as a legacy calibration point but marked as non-streaming; the streaming active-budget A/B in Table 2.11 is the row to use for first-token or inter-token claims.

Supplemental streaming-tail rows in the supplementary coverage artifact keep the non-streaming active-budget case out of first-token and inter-token claims. They support the percentile warning in Section 2.5: a healthy median can hide a much larger tail.

Figure 2.20 shows the operating surface that contains the offline-throughput and online-latency scenarios as two points. The supplemental offered-load sweep in the supplementary coverage artifact supports the same shape: low offered load can keep first-token latency low, increasing load improves utilization, and the knee appears as completed throughput approaches the offline-throughput row while end-to-end tails grow. The 3.29k-versus-139 output-token contrast

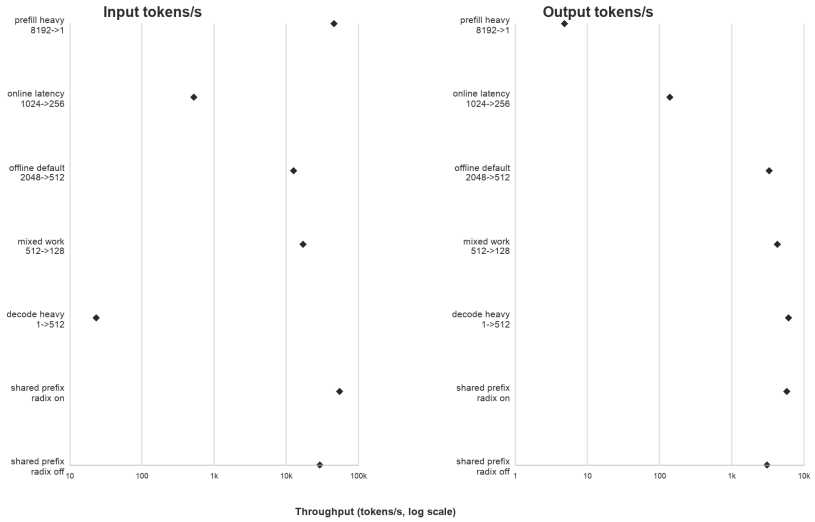



**Figure 2.18.:** Input-token and output-token throughput for selected Thor serving scenarios. Points show device means. The log scale keeps prompt-heavy and decode-heavy regimes visible in the same plot.

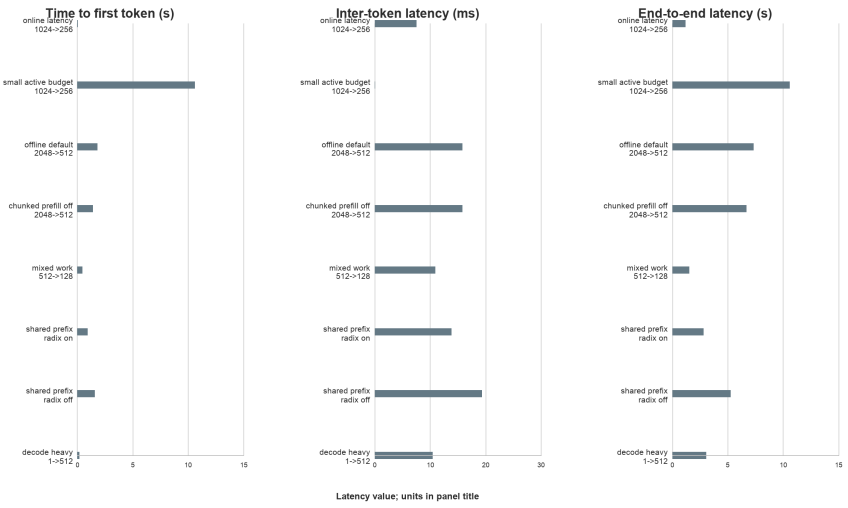
belongs on this surface: the offline row keeps a backlog near saturation, while the online row is paced by arrivals and has the shorter median end-to-end latency, 1.15 s versus 7.32 s.

The scheduler-policy row should be read as workload-conditioned evidence, not a global ordering of `lpm` over `fcfs`. The continuous-batching measurements in Chapter 5 repeat the same policy pair with a smaller shared-prefix shape and observe the opposite ordering. The two tables together are the useful result: prefix-aware scheduling is a policy knob whose effect depends on the sharing geometry and arrival shape.

Figure 2.21 isolates the shared-prefix workload with radix caching enabled and disabled. It is a small comparison, but it is a useful calibration point for effective prompt work (Definition 2.4.3): changing cache reuse changes both the generated-token rate and the completion latency seen by the benchmark.

The supplementary coverage artifact preserves the details left out of the chapter body: supporting run families, diagnostic failures, streaming-tail rows, and offered-load sweep rows. The conclusion here is narrower: these completed Thor/Qwen rows make the serving axes concrete without turning one small-model, one-hardware-class slice into a general performance law.

Use these scenario-conditioned numbers as calibration points. Their interpretation depends on the model, NVIDIA container, Thor software stack, common server arguments, benchmark traffic, and device state. Their value in this discussion is that each abstract scheduler variable has a measured consequence: arrival rate changes the latency/throughput point, active-request limits change how much work can be resident, prompt shape changes whether input or output tokens dominate, and prefix reuse changes the amount of fresh work the runtime must admit.




*Note:* the small-active-budget row is non-streaming; its TTFT is total latency by convention and its ITL is unmeasured.

**Figure 2.19.:** Latency components for selected Thor serving scenarios. Bars show device means. Panel titles carry the units because time to first token and end-to-end latency are shown in seconds, while inter-token latency is shown in milliseconds. The small-active-budget row is non-streaming: its TTFT bar equals total latency by benchmark convention, and its ITL value means no streamed-token gaps were observed.

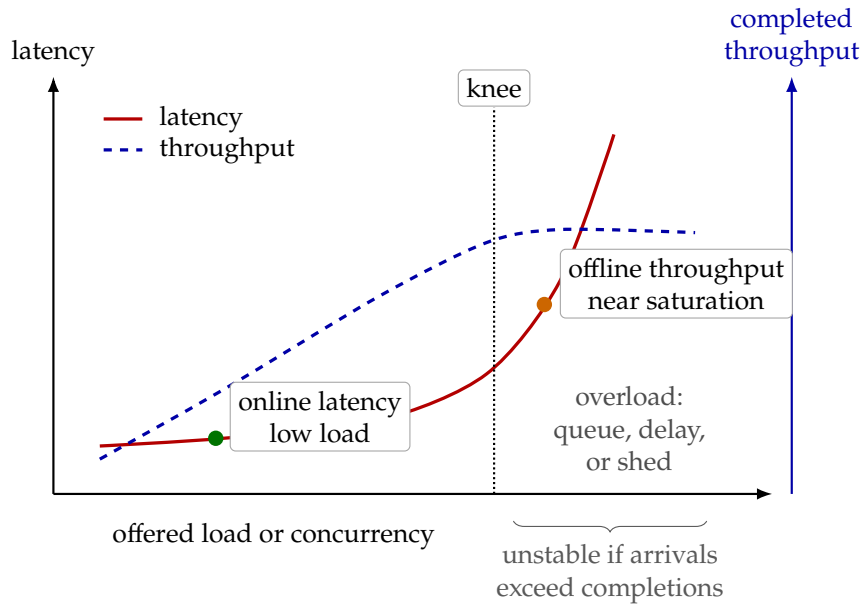



**Figure 2.20.:** Latency/throughput operating surface. The schematic shows why the offline-throughput and online-latency measurements should be read as operating points, not as a single global optimum. Stable measurements lie before the overloaded region; once offered work exceeds completed work, a run must account for queue growth, admission control, or rejected work.

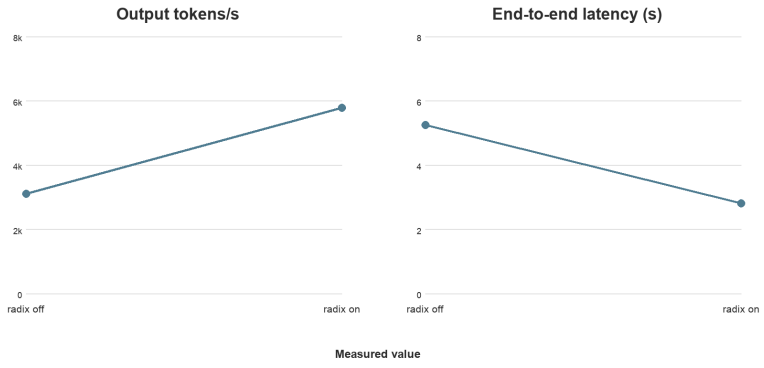



**Figure 2.21.:** Shared-prefix serving with radix caching disabled and enabled. Large points show device means. Enabling radix-cache reuse increases output-token throughput and reduces median end-to-end latency for this generated shared-prefix workload.

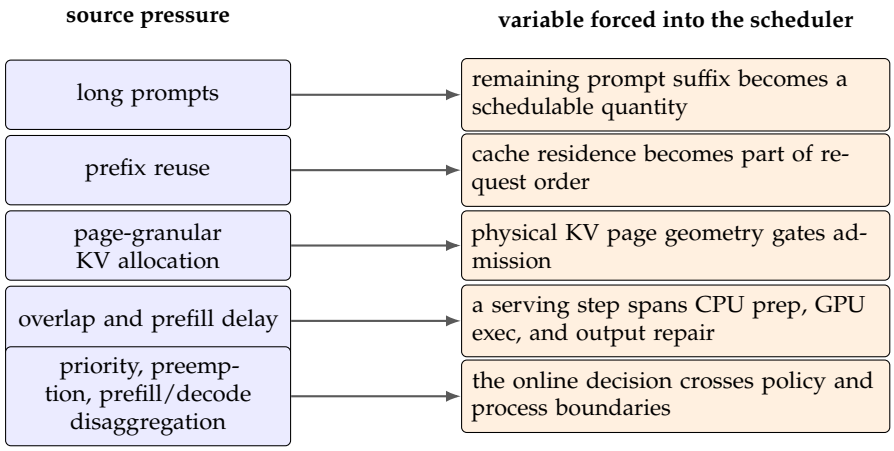
## 2.10. Scheduler Source Pressure and Runtime Scope

The chapter closes by checking the problem model against SGLang's scheduler sources. In `python/sglang/srt/managers/scheduler.py`, a serving step no longer sees only a FIFO list of requests. It must read remaining prompt work, prefix-hit length, available KV pages, decode rows that must not stall, output repair work, and policy state that can cross prefill/decode boundaries. The batch record in `schedule_batch.py` and the policy code in `schedule_policy.py` decide which visible rows can run. Prefill-delay and overlap support then add preparation, completion, and repair boundaries through `prefill_delayer.py` and `overlap_utils.py`.

Source pressure marks implementation boundaries; it is not a new scheduling objective.

### Variables Forced into the Scheduler

Here *evolution* means the scheduler being forced to name variables that a simpler batching loop could ignore. Long prompts made remaining prompt suffix a schedulable quantity. Prefix reuse made cache residence and safe prefix length part of request order. Page-granular allocation made physical KV geometry visible to admission. Overlap and prefill delay made a serving step include CPU preparation, GPU execution, output repair, future-token handling, and decode-latency protection. Priority, preemption, and prefill/decode disaggregation then moved the same online decision across policy and process boundaries [50]. Figure 2.22 pairs each pressure with the variable it forced the scheduler to name.




Each pressure makes a first-come, whole-prompt batching rule wrong, and so becomes part of the Chapter 2 serving problem.

**Figure 2.22.:** Scheduler source pressure. Each workload or deployment pressure on the left forced the scheduler to track a variable on the right that a simpler first-come, whole-prompt batching loop could ignore. These forced variables are exactly the quantities the chapter's serving problem must reason about.

Read Figure 2.22 with those source roles in mind. `schedule_batch.py` carries the request rows, token counts, prefix positions, and page-allocation state that make admission meaningful. `schedule_policy.py` changes which visible requests are considered first. `prefill_delayer.py` and `overlap_utils.py` protect decode latency by splitting a serving step into preparation, launch, completion, and repair work. Those cases are the concrete pressure behind the serving problem defined in this chapter: choose feasible work from visible request state while preserving token state, capacity, and user-visible latency.

## 2.11. Summary

Chapter 2 turned online LLM serving into a concrete control problem. The runtime sees arriving requests, preserves their state across steps, accounts for prompt and decode work separately, and admits only work that satisfies resource and policy constraints. The scheduler therefore reasons about token work, resident token state, prefix reuse, latency metrics, and feasible executable steps rather than request count alone.

The pancake request from Chapter 1 is the same obligation under different subsystem pressures: first as positions, then as schedulable work, then as state that must survive ownership changes. The active-work invariant (Invariant 2.2.3) is the first version of that rule: completed generation may still need bounded handoff, but it must stop competing for model work.

Prefill, decode, and resident token state remain separate quantities. Future token-state obligations stay accounted for even when the visible next action is only one prompt or decode slice (Invariant 2.3.6). Continuous batching and chunked prefill use that separation to decide which work may run now. KV allocation and prefix reuse use it to decide which state must stay resident and which prefix work can be skipped.

The rest of Volume I follows those variables where they become operational. Chapter 3 turns the decision into a ledger of scheduled token rows, attention-pair units, dense-token units, resident KV slots, and request rows. Later chapters lower the same serving question into scheduler state, runtime messages, memory pages, backend contracts, cache identity, and measurement boundaries.

## 2.12. Exercises

1. Define an online serving policy (Definition 2.6.2) in mathematical terms. For the running example, list the information available to the policy at 8 ms and the information that remains unknown.
2. For requests A, B, C, and D, identify which request has the largest raw prompt work, which has the longest expected decode lifetime, and which may have the smallest effective prompt work after prefix reuse.
3. Suppose a serving instance can admit at most 2048 uncached prompt tokens in one step. Show two feasible admission plans for the running example by 12 ms, and explain the latency tradeoff between them.
4. Give a formal definition of a live request. Then explain why a completed request should not remain in the active decode set.
5. Construct a small example where static batching causes head-of-line blocking (Definition 2.7.2) but dynamic batching (Definition 2.7.1) avoids it. Use arrival times, prompt lengths, and output lengths in your example.
6. Let a model family require $m$ bytes of fast-memory token state per token. If the cache budget is $C$ bytes and currently live requests own token states for $n_1, n_2, \ldots, n_k$ tokens, write the feasibility condition for admitting $q$ additional prompt tokens.

7. In a benchmark run, 500 requests generate 250,000 output tokens over 80 seconds. Compute output throughput. If the sum of request latencies is 640 seconds, compute observed concurrency. Then state whether the concurrency number is a request-population average or a token-throughput metric.
8. Explain why time to first token and inter-token latency can move in different directions when long prompt work is mixed with short decode work.
9. Consider two requests with identical raw prompt length, but one has a reusable prefix and the other does not. Define effective prompt work (Definition 2.4.3) for both and explain how this changes admission decisions.
10. List three claims about serving performance that would require experimental evidence rather than source reading alone. For each claim, name the metric that should be measured.