Mini-SGLang: PrefillManager

A concise reference for understanding the prefill scheduling pipeline

---

1. Overview

PrefillManager sits inside the Scheduler process. It manages a queue of incoming requests that have not yet started generating tokens. Its single job is to take pending requests and form efficient **prefill batches** to send to the GPU engine.

Scheduler

```
■■■ PrefillManager  ← pending requests, batch formation
■■■ DecodeManager    ← running requests, token generation
■■■ CacheManager     ← KV cache pages (Radix Cache)
■■■ TableManager     ← request slots & page tables
```

2. Request Lifecycle

Stage	What Happens
1. Arrival	`add_one_req(msg)` wraps the tokenized request into a `PendingReq` and appends it to `pending_list`.
2. Batch Formation	`schedule_next_batch()` creates a `PrefillAdder` and iterates through pending requests, trying to fit each one within the token budget.
3. Cache Match	For each request, `cache_manager.match_req()` checks how many prefix tokens already have KV cache (e.g. shared system prompt). Only the unmatched tokens need computation.
4. Resource Alloc	A table slot is allocated, page table entries are set up for cached tokens (pointing to existing KV pages), and the token budget is decremented.
5. Forward Pass	The batch is sent to the Engine for prefill. All new tokens are processed in parallel, KV cache is computed.
6. Transition	Fully prefilled requests move to **DecodeManager**. Chunked requests stay in `pending_list` for the next round.

3. Key Concept: Chunked Prefill

Long requests are split across multiple scheduling rounds to control GPU memory usage. The token budget (default 8192) caps how many tokens can be prefilled in one batch.

Example

```
Request X: 12,000 input tokens, budget = 8,192

Round 1: prefill tokens [0 .. 8,191]
  → returns ChunkedReq (remaining = 3,808)
  → stays in pending_list at the FRONT

Round 2: prefill tokens [8,192 .. 11,999]
  → returns normal Req
  → moves to DecodeManager
```

ChunkedReq safety guards: A ChunkedReq overrides `can_decode()` → `False` and `append_host()` → `raises error`. This prevents the system from accidentally starting decode or appending a meaningless sampled token when the model hasn't seen all input tokens yet.

4. Key Concept: Memory Reservation

PrefillAdder is initialized with `reserved_size = decode_manager.inflight_tokens`. This protects already-decoding requests from being starved of KV cache pages by new prefills.

```
Total KV cache = 20,000 pages
In-flight decode requests need ~440 more pages

PrefillAdder(
  token_budget = 8,192   # max tokens this round
  reserved_size = 440     # memory decode still needs
)

As each new prefill request is added:
  reserved_size += extend_len + output_len
  # extend_len = tokens to prefill (after cache match)
  # output_len = future decode pages this request will need
```

Two-layer reservation: Layer 1 (initialization) protects existing decode requests. Layer 2 (per-request increment) reserves space for each newly added request's future output.

5. Key Concept: Prefix Cache Reuse

Before allocating new pages, `cache_manager.match_req()` checks the Radix Cache for prefix matches. Shared prefixes (like system prompts) reuse existing KV cache pages — no recomputation needed.

Example

```
Request 1: [SYS, SYS, SYS, Hello, World]  (first time)
  cached_len = 0  → extend_len = 5  (compute all)

Request 2: [SYS, SYS, SYS, Tell, Me]     (same prefix)
```

```
  cached_len = 3  → extend_len = 2  (only compute new tokens)

The page table for Request 2 points to Request 1's
existing KV pages for positions 0-2:
  page_entry.copy_(match_indices)  # [42, 43, 44]
```

6. Key Concept: Pending List Ordering

After batch formation, the pending list is rebuilt with a specific priority:

```
self.pending_list = chunked_list + self.pending_list[len(reqs):]

Before: [A, B, C, D, E]
  A → ChunkedReq (partial), B → done, C → done, D → failed

After: [A, D, E]
  A at front (has resources held, finish it first)
  B, C removed (moved to decode)
  D, E remain (not yet attempted)
```

Why chunked requests go first: They already hold a table slot and locked cache pages. Pushing them to the back risks resource waste and starvation — their held resources sit idle while new requests grab even more.

7. Subtle Detail: Double Availability Check

```
# Check 1: before locking
if estimated_len + self.reserved_size > available_size:
  return None

self.cache_manager.lock(handle)

# Check 2: after locking
if estimated_len + self.reserved_size > available_size:
  return self.cache_manager.unlock(handle)
```

Locking a cache handle may change available memory (it pins pages that were previously evictable). The first check is an optimistic fast-path; the second check catches the case where locking reduced available space below what's needed.

8. Quick Reference: Key Methods

Method	Purpose
`add_one_req(msg)`	Wrap incoming request as PendingReq and append to queue
`schedule_next_batch(budget)`	Form the next prefill batch from pending requests
`PrefillAdder.try_add_one(req)`	Try to fit one request into the batch (allocate or resume chunk)

`_try_allocate_one(req)`	Cache match → memory check → lock → slot allocation
`_add_one_req(...)`	Chunk sizing, budget decrement, token ID copy to GPU