nano-vLLM

Scheduler

BrassinAI

How nano-vLLM decides what runs, when, and why.

1. Why Schedulers Exist
2. Mental Model
3. Sequences & Queues
4. Prefill vs Decode
5. Memory & Blocks
6. Reading schedule()
7. Prefix Caching
8. Preemption
9. Scheduler → GPU

58 slides · 9 chapters

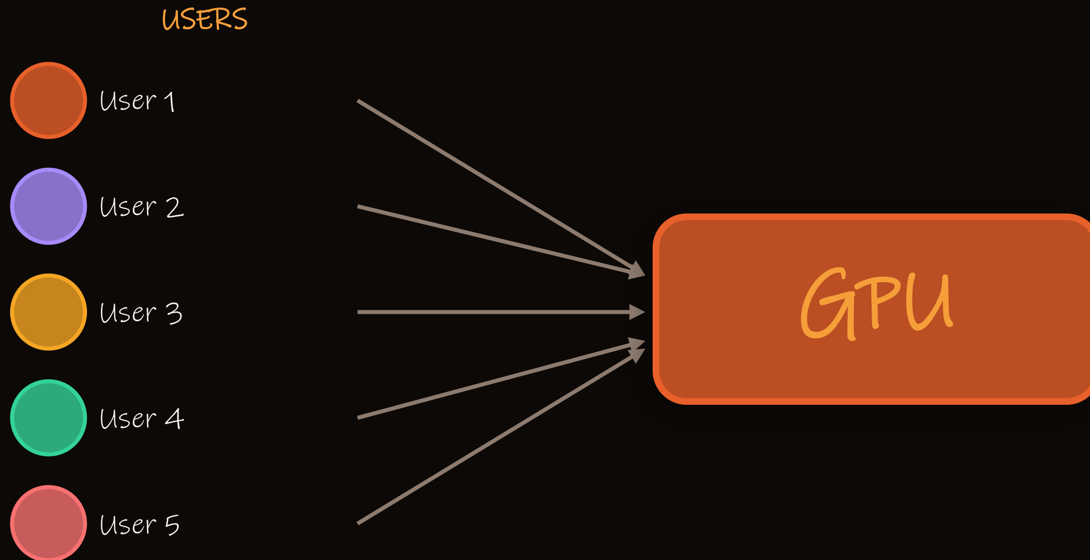
1

Why Schedulers Exist

Build intuition before discussing implementation

Why Does an LLM Server Need a Scheduler?

One GPU. Thousands of users. Something has to decide who goes first.



The core tension

Requests
arrive continuously

GPU
can only do so much

Memory
is finite

Without order
chaos ensues

Takeaway: Many requests compete for one limited resource. The scheduler decides who gets it.

The Naive Server: One at a Time

Process each request fully before starting the next.

What goes wrong

Poor throughput

The GPU only works on one sequence at a time. Most of its capacity is unused.

Poor latency

Short requests wait behind long ones. A 5-second request waits behind a 60-second one.

Wasted hardware

You paid for a fast GPU. You're using it like a single-threaded processor.

The Cost of Waiting

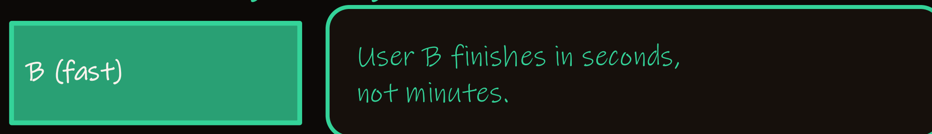
A short request trapped behind a long one — even when the GPU could help.

WITHOUT scheduling

User B's wait time:

60+ seconds

WITH scheduling (coming up)



User B finishes in seconds,
not minutes.

Continuous Batching: The Key Idea

Serve multiple sequences simultaneously — interleaved, one token at a time.

Instead of finishing one request before starting the next, we advance ALL running requests together, step by step.

When Seq B finishes at t5, a new request immediately fills that slot. The GPU is always busy.

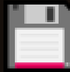
The Scheduler's Mission

One job: keep the GPU busy doing useful work.

Maximise useful GPU utilisation
by serving as many requests as possible, as fast as possible.

Admit

Decide which waiting requests get GPU time next



Manage

Track how much GPU memory each request consumes

Execute

Hand the right sequences to the GPU at the right time

2

Mental Model of the Scheduler

Understand the system before any implementation

The Restaurant Analogy

A mental model you already understand.

	Customers	→	Requests / Users
	Kitchen	→	GPU
	Manager	→	Scheduler
	Table	→	GPU Memory
	Menu item	→	Token generation

The manager decides which customers get seated, in what order, and when the kitchen should start cooking.

The Three Things the Scheduler Tracks

Everything else is derived from these three.

1

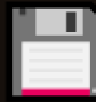
Waiting Queue

Requests that have arrived but haven't started yet. No GPU memory allocated.

2

Running Queue

Requests actively generating tokens. GPU memory is allocated to them.



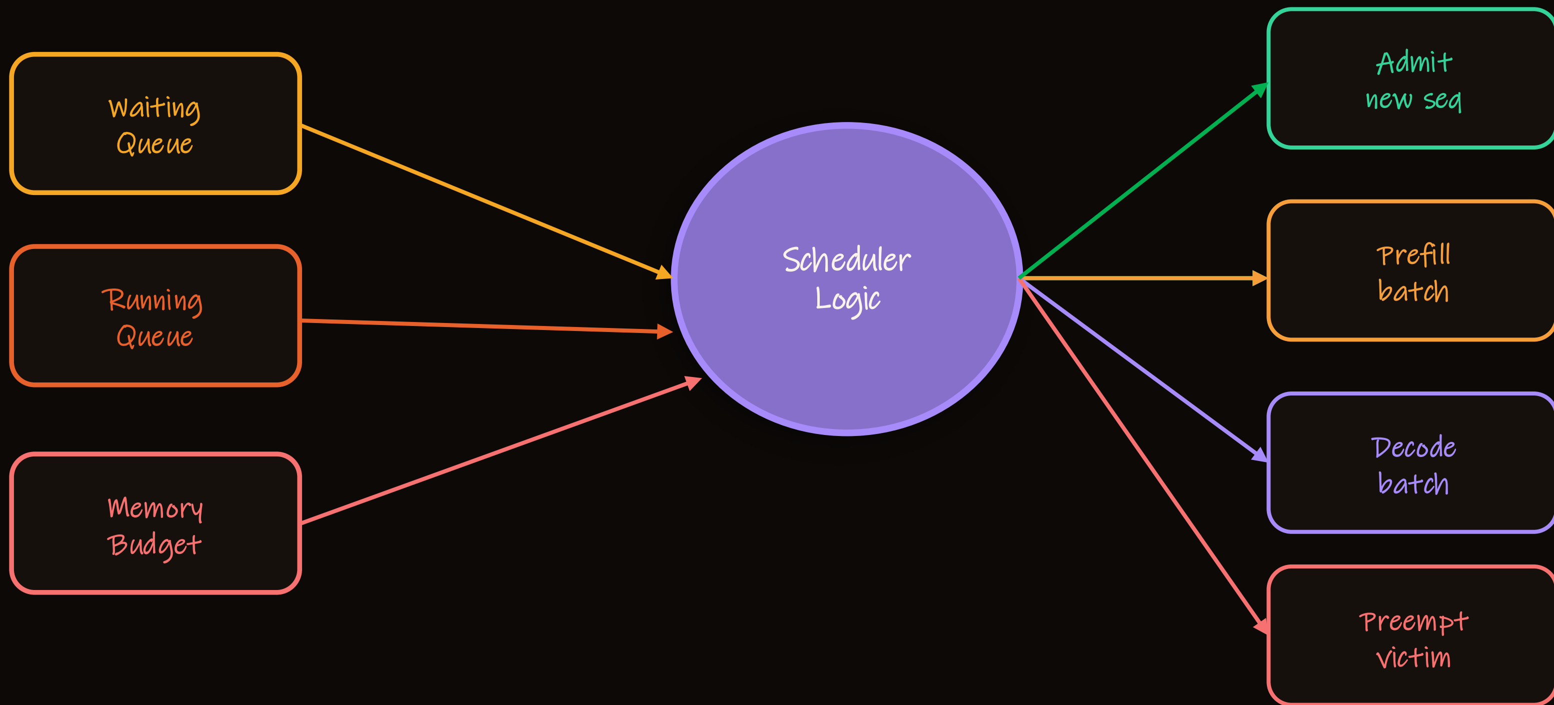
3

Memory Budget

How much GPU memory is available. When this runs out, no new requests can start.

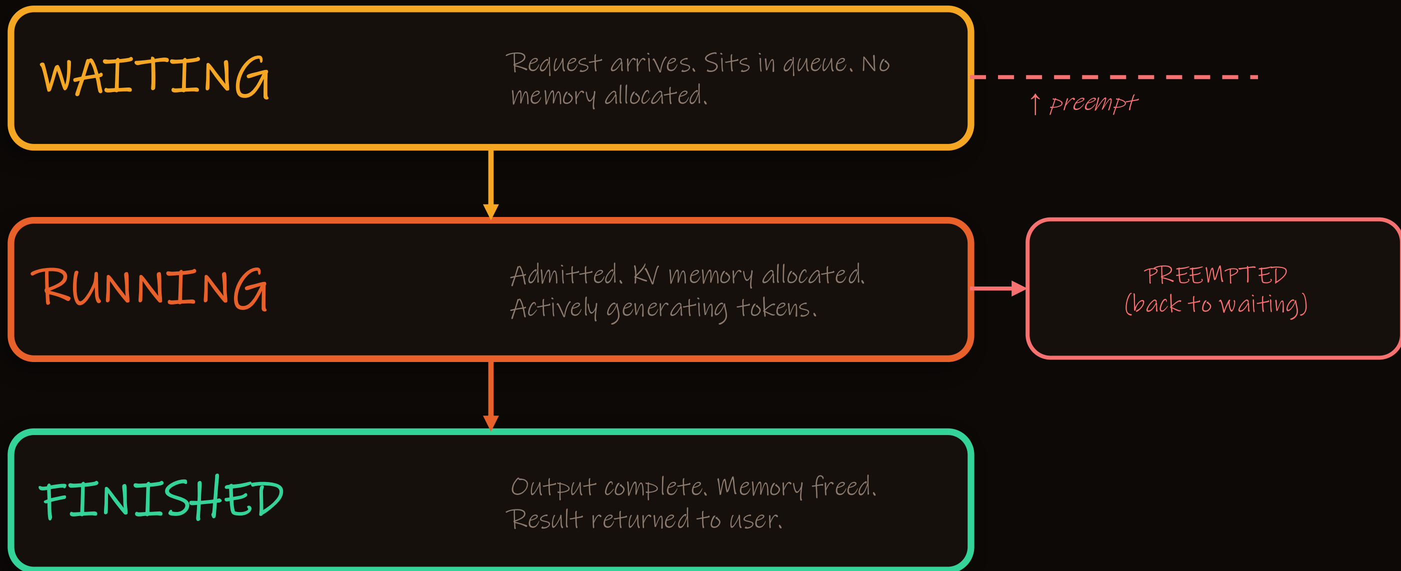
Everything Else Is Derived

Complex behaviour emerges from three simple things.



The Scheduler Lifecycle

Every request follows the same path.



Optional path: if memory runs out, a RUNNING sequence can be pushed back to WAITING (preemption). More on this in Chapter 8.

3

Sequences and Queues

Introduce the actual scheduler objects

What Is a Sequence?

A sequence is one inference request plus all the state needed to generate its response.

What the scheduler sees:

- **token_ids** — [1342, 374, 11782, 11478, 30] ← prompt tokenised into numbers
- **status** — WAITING → RUNNING → FINISHED
- **block_table** — [ ] ← empty until admitted; fills with memory block IDs
- **tokens_generated** — 0 ← counter incremented each decode step
- **max_tokens** — 256 ← hard cap on how long the response can be

A sequence is the scheduler's unit of work — one request, tracked from arrival to completion.

Sequence Status: Three States

A sequence is always in exactly one of these states.

WAITING

The request has arrived and is queued. No GPU memory is allocated. The sequence is waiting for the scheduler to admit it.

Set when: request first arrives. Also set when preempted.

RUNNING

The sequence has been admitted. GPU memory blocks are reserved. The model is actively generating tokens for this request.

Set when: scheduler allocates KV blocks and begins prefill.

FINISHED

Generation is complete (EOS token or max_tokens reached). All GPU memory blocks are immediately freed.

Set when: EOS token generated or max_tokens limit hit.

The Waiting Queue

New requests arrive here. They wait patiently until the scheduler has room.

Key properties: FIFO order · No KV memory allocated · Sequences wait here on arrival AND after preemption (pushed to front)

The Running Queue

Active sequences — they hold GPU memory and generate tokens every step.

Seq #7

384 tokens generated

KV blocks:

#3 #7 #12

⚡ RUNNING

Seq #5

256 tokens generated

KV blocks:

#1 #4

⚡ RUNNING

Seq #9

128 tokens generated

KV blocks:

#9

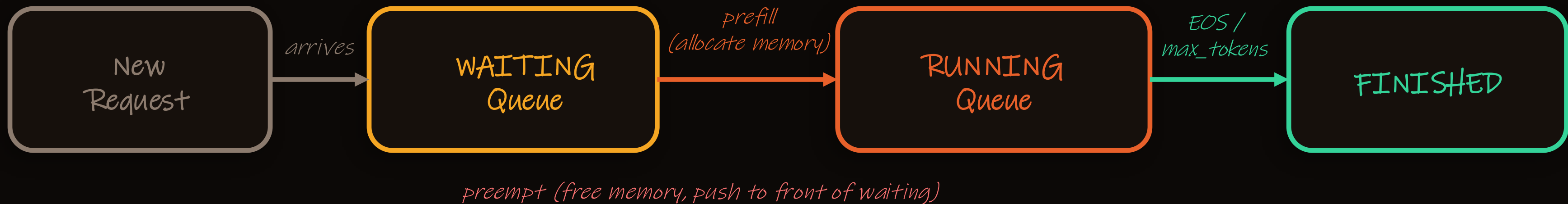
⚡ RUNNING

Key properties: each sequence holds physical KV memory blocks · generates one token per decode step · can be preempted if memory pressure rises

This is where the GPU work happens — every sequence in the running queue generates one token per inference step.

How Requests Move Through the System

The scheduler moves sequences between queues based on memory and compute decisions.



- → Waiting: Any new request enters the waiting queue first, regardless of priority.
- → Running: Scheduler admits request when token budget and memory budget both allow it. Prefill runs.
- → Finished: Scheduler detects EOS token or max_tokens hit. Blocks freed immediately.
- ↩ Preempt: If memory runs out, a running sequence is sent back to waiting (front of queue).

4

Prefill vs Decode

The most important mental model in this presentation

What Happens When a Request Starts?

There are two very different phases of LLM generation.

When a request moves from WAITING → RUNNING, the model does two distinct things in sequence:

1 PREFILL

Read and understand the entire prompt. Build a memory structure (the KV cache) that lets the model 'remember' the context.

Done once, at the start.

2 DECODE

Generate the response, one token at a time. Each step produces exactly one new word or sub-word.

Repeated until EOS or max_tokens.

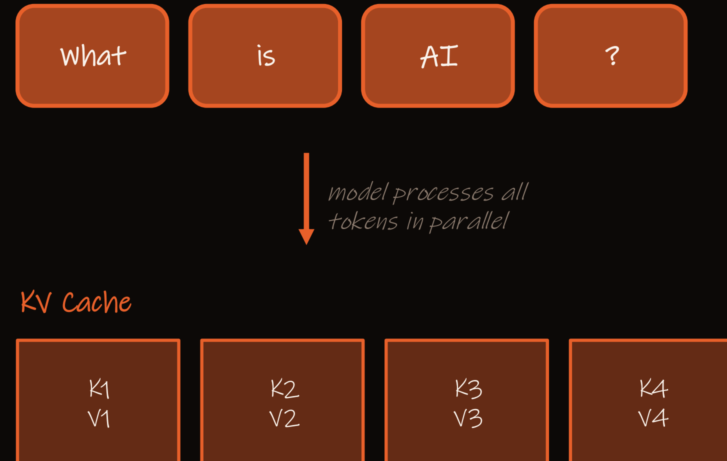
Prefill: Building the KV Cache

The model reads the entire prompt and stores what it learned.

What prefill does:

1. Take all prompt tokens at once
2. Run them through every transformer layer
3. For each layer, compute Key and Value vectors
4. Store K and V in GPU memory (= the KV cache)
5. After this, the model 'knows' the full context

Prefill is compute-intensive: parallel matrix multiply over all prompt tokens at once.



Decode: One Token at a Time

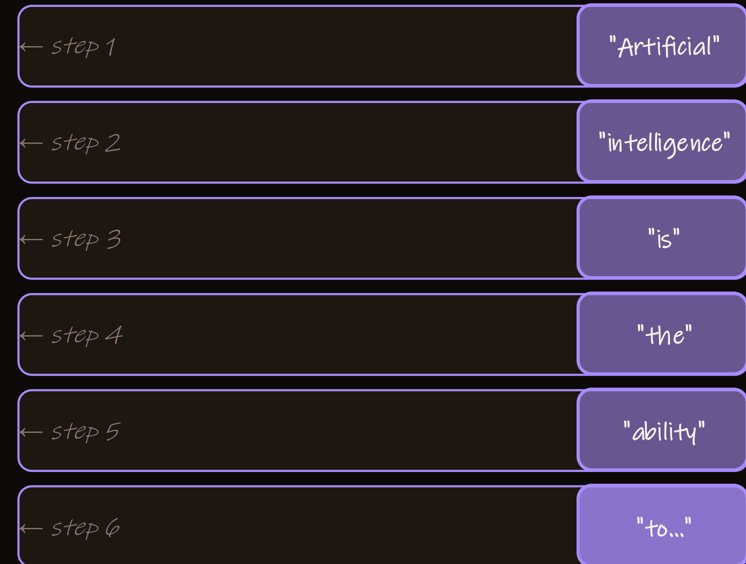
After prefill, the model generates the response token by token.

What decode does:

1. Take only the last generated token
2. Compute attention over entire KV cache
3. Predict probability of each next token
4. Sample one token (the next word)
5. Append it to KV cache, repeat

Decode is memory-bandwidth-bound: one token, but reads the entire KV cache.

Token-by-token generation



PREFILL

creates memory

DECODE

uses memory

This is why memory management is the central challenge of LLM serving.

Why the Scheduler Separates Prefill and Decode

Two fundamentally different workloads need different optimisation strategies.

	PREFILL	DECODE
Tokens per step	All prompt tokens (e.g. 512)	Exactly 1 token
Compute profile	Matrix multiply-bound (fast)	Memory-bandwidth-bound
GPU shape	Variable batch size	Fixed batch size → CUDA graphs
KV cache access	Writes only (storing new K/V)	Reads entire cache + 1 write
Latency priority	Starts new request — latency-sensitive	Continues existing — throughput-sensitive

Scheduling them separately lets the GPU be optimised for each workload. Mixing them wastes both.

5

Memory, Blocks, and BlockManager

Understand scheduler constraints

Why Memory Becomes the Boss

KV cache grows with every token — and it never shrinks until the request finishes.

Here is the challenge:

Every token generated adds more data to the KV cache

That memory stays reserved for the entire lifetime of the request

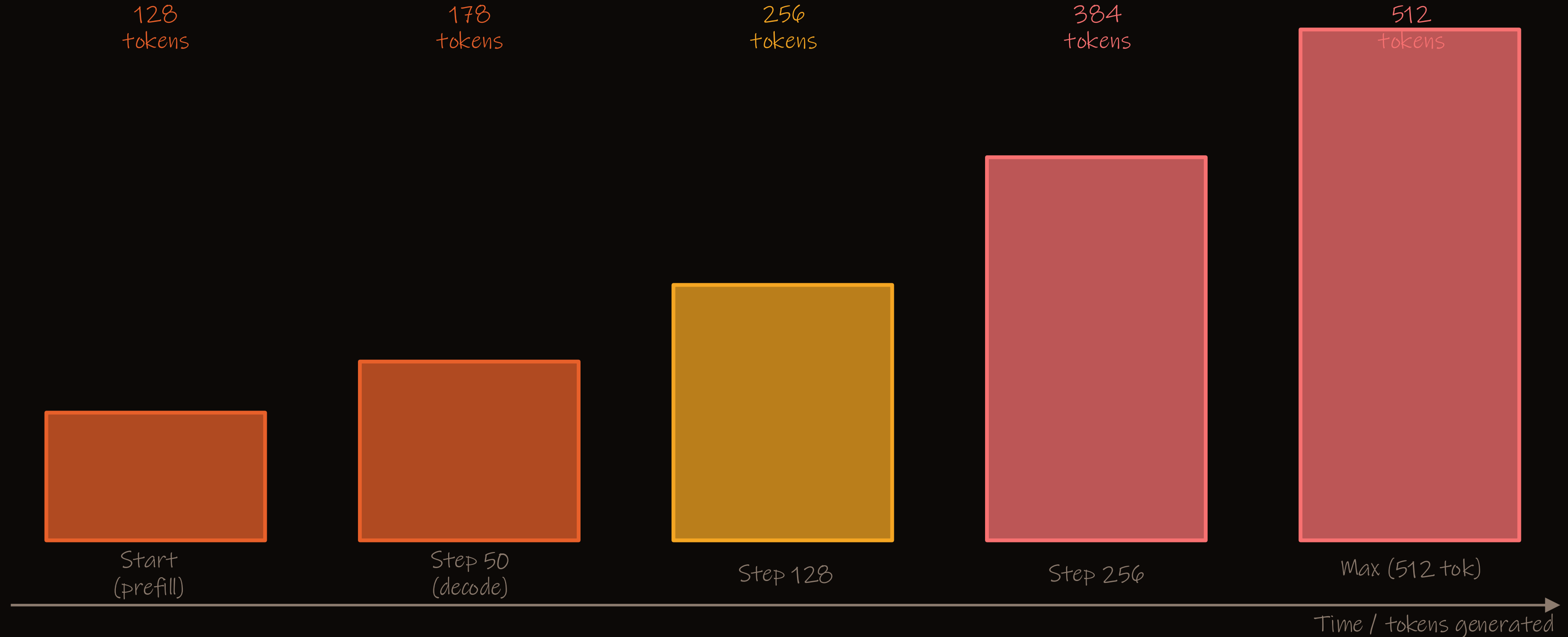
Once GPU memory is full, no new requests can start

Even one very long request can starve the GPU of capacity

Memory is finite. Compute is abundant. This makes GPU memory — not raw compute — the true bottleneck.

Memory Grows as Tokens Are Generated

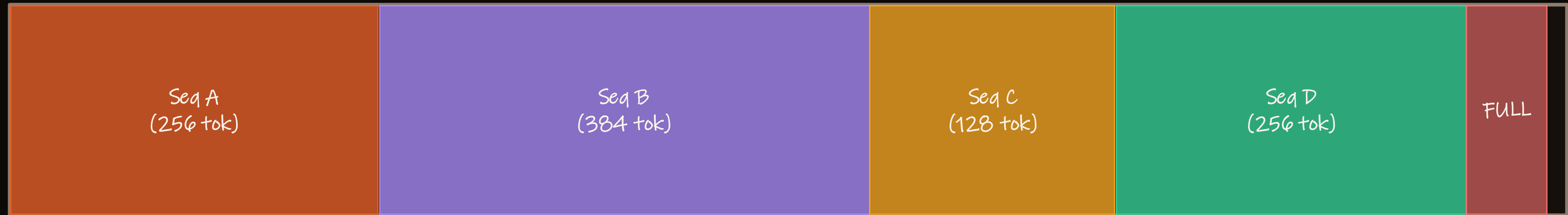
A request's memory footprint increases with every decode step.



Memory never shrinks until the request finishes (or is preempted).

GPU Full: The Problem Statement

What happens when there is no more room?



New requests queue up in WAITING indefinitely

Latency grows for all new users

Parts of GPU compute sit idle (decode only uses some of the hardware)

Solution: we need a way to manage memory predictably. That solution is blocks.

Blocks: Fixed-Size Memory Chunks

KV cache memory is divided into equal-size blocks of 256 tokens each.

Instead of allocating memory freely (like malloc), nano-vLLM carves GPU memory into fixed 256-token blocks.

🅿️ **Like a parking garage**

Cars park in fixed bays. You rent whole bays — not arbitrary square footage. The garage always knows exactly how many bays are free.

Predictable

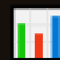
Every block is exactly 256 tokens. No fragmentation, no guessing.

O(1) ops

Allocate = pop from free list. Free = push back. Both are instant.

Hashable

Fixed-size blocks have consistent content. Easy to fingerprint for prefix caching.

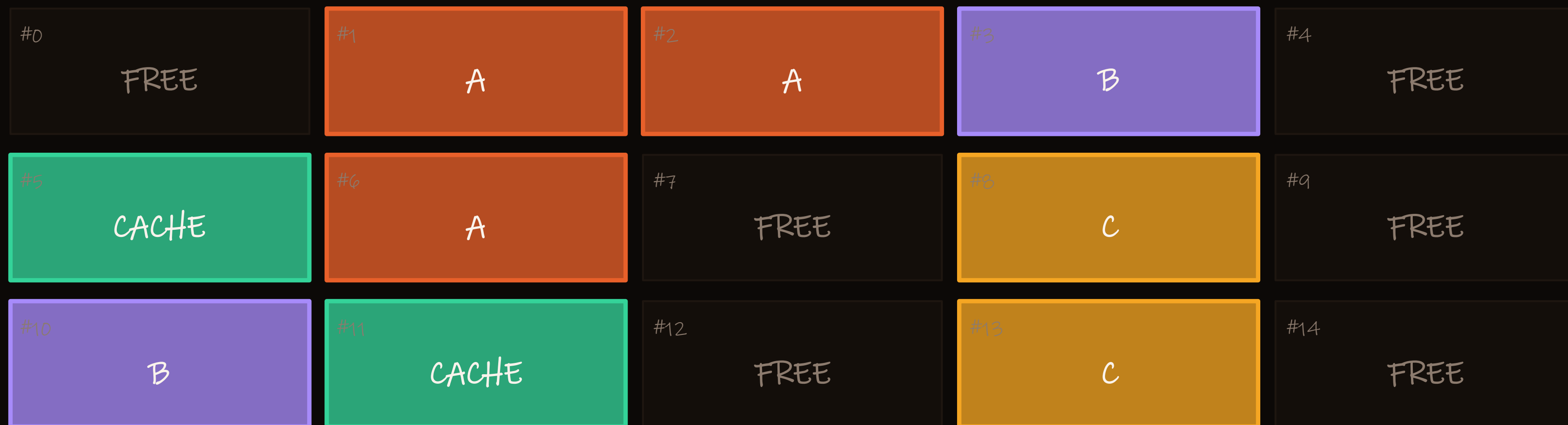


Countable

Always know exactly how many blocks are free. Budget management is trivial.

What Blocks Look Like in GPU Memory

A flat array of blocks — each holds 256 tokens' worth of KV vectors.



Each block is exactly 256 tokens. A sequence can span many non-contiguous blocks — the block_table maps logical positions to physical block IDs.

BlockManager: The Memory Accountant

Knows exactly what is free, what is in use, and what can be reused.

The BlockManager is the scheduler's dedicated memory sub-system. The scheduler asks it questions; it answers.

?

Can I admit this sequence?

can_allocate(seq) — Yes if free_blocks $\geq$ seq.num_blocks

?

Do I need a new block this step?

can_append(seq) — Yes if sequence just crossed a 256-token boundary

?

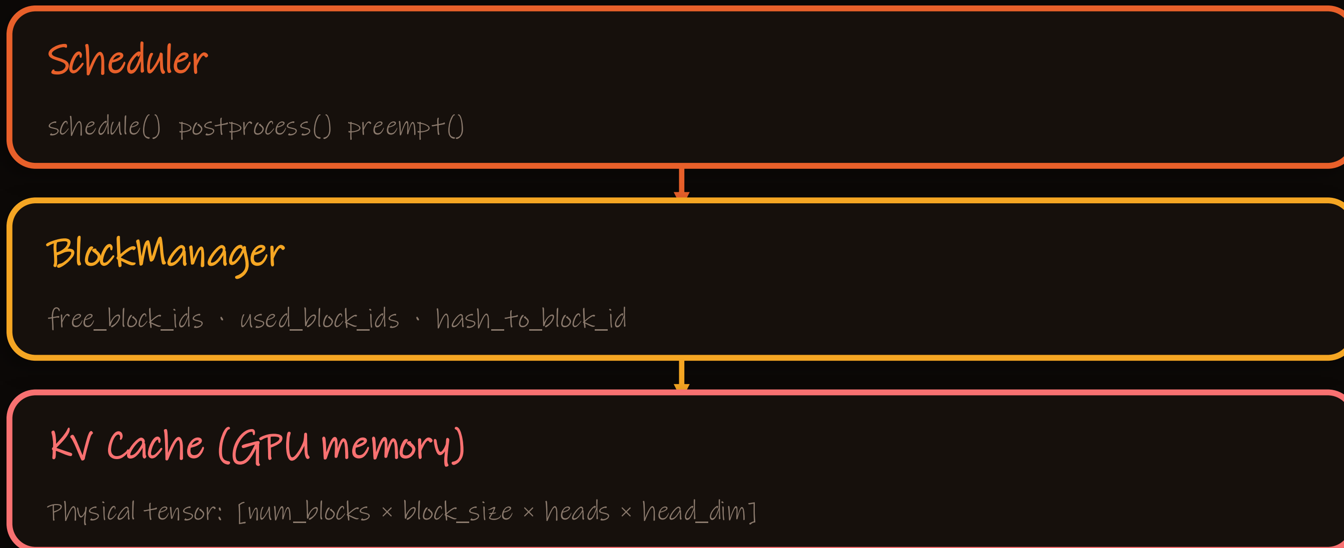
This sequence is done — clean up.

deallocate(seq) — Decrements ref_counts, returns blocks to free list

The BlockManager is called by the scheduler every single inference step. It never touches the GPU directly — it just tracks IDs.

BlockManager Architecture

Where it sits, what it holds, what it knows.



Inside BlockManager:

- free_block_ids: deque of available block IDs → O(1) pop for allocation
- used_block_ids: set of in-use block IDs → O(1) membership check
- hash_to_block_id: dict from hash → block ID → the prefix cache lookup table

BlockManager: Three Core Operations

Every interaction between the scheduler and memory goes through one of these.

allocate(seq)

WHEN

When a sequence is admitted to running.

WHAT

Reserve blocks from the free list. Build the sequence's block_table. Count cached blocks (prefix cache hits).

EFFECT

free_block_ids shrinks. sequence.block_table populated.

may_append(seq)

WHEN

Each decode step — but only if the sequence just crossed a 256-token block boundary.

WHAT

Pop one block from free list. Append to sequence's block_table.

EFFECT

free_block_ids shrinks by 1 (at most once per 256 tokens).

deallocate(seq)

WHEN

When a sequence finishes (or is preempted).

WHAT

Decrement ref_count of each block in block_table. Return blocks with ref_count=0 to free list.

EFFECT

free_block_ids grows. Memory available for new sequences.

Reading schedule()

Understand the scheduler loop — decision by decision

What Does schedule() Actually Do?

Called once per inference step. Returns a list of sequences and a phase flag.

Every inference step, the engine calls schedule(). It returns:

sequences	A list of Sequence objects to process in this step
is_prefill	True = run prefill for these sequences · False = run decode

The high-level decision tree:

Is anyone waiting AND can we admit them?

YES → run PREFILL for admitted sequences

NO → run DECODE for all running sequences

schedule() — The Entire Logic

Simpler than you think. Two phases, strict priority.

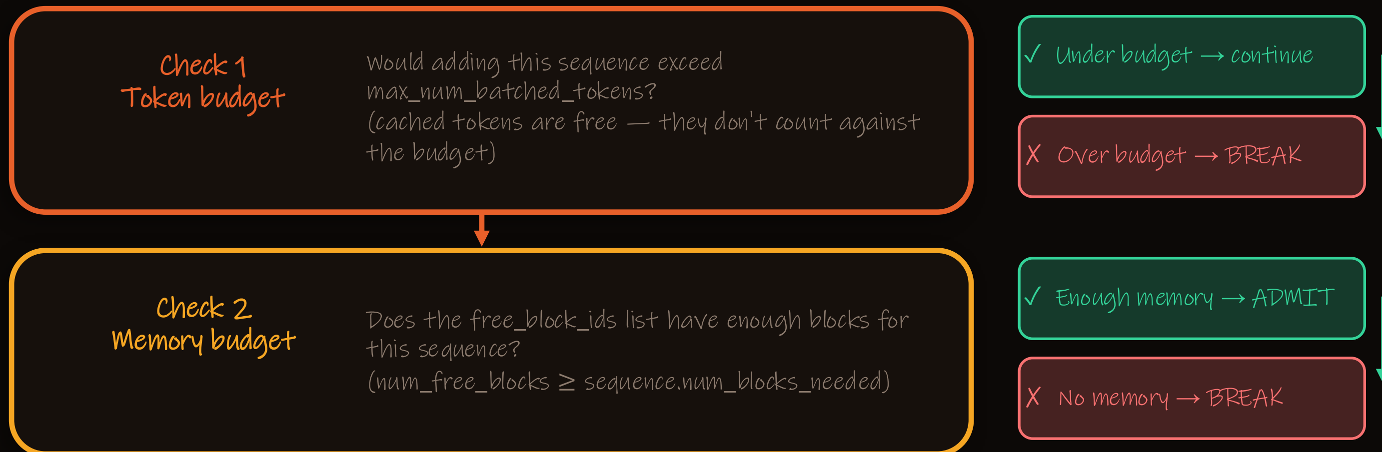
PSEUDOCODE

```
function schedule():

    # —— PHASE 1: PREFILL ————————————————
    admitted = []
    for each seq in waiting_queue:
        if token_budget_ok AND memory_ok:
            admit seq → alloc blocks → move to running
        else:
            break ← STOP, do not skip
    if admitted:
        return (admitted, is_prefill=True)

    # —— PHASE 2: DECODE ————————————————
    for each seq in running_queue:
        if can_append(seq):
            may_append(seq) → add to batch
        else:
            preempt someone → free memory → retry
    return (decode_batch, is_prefill=False)
```

The Prefill Path: Admitting Sequences

Two checks must pass before any sequence is admitted.

For each sequence in the waiting queue:



ADMIT: allocate KV blocks → move to running queue → add to prefill batch

break not continue: if any check fails, stop the loop entirely. Smaller waiting requests do NOT jump the queue.

Admission Checks: Two Gates, Both Must Pass

Think of it as a nightclub bouncer with two independent rules.

Capacity rule: the club holds 200 people total.
→ Gate 1: total_tokens + this_request ≤ max_batched_tokens

ID check: you must be able to hold a table.
→ Gate 2: enough free memory blocks for this sequence

If either rule fails: the whole queue stops. No jumping ahead.

Why BREAK and not CONTINUE?

Starvation prevention. If the scheduler could skip a large request and admit smaller ones, a large request could wait forever.

Fairness. FIFO order is simple and predictable. Users know their request will run in roughly the order it arrived.

Simplicity. No priority inversion, no starvation logic needed.

FIFO is a deliberate design choice — not a limitation.

The Decode Path: Advancing All Running Sequences

When nothing new can be admitted, advance every running sequence by one token.

After confirming no prefill is needed, the scheduler prepares a decode batch:

Step	Action	Description
1	Pop sequence from running queue (front)	Process sequences in admission order.
2	Check can_append(seq)	Does this step cross a 256-token block boundary? If yes, we need one free block.
3a	YES: may_append(seq)	Allocate a new block if needed. Add sequence to decode batch.
3b	NO: preempt someone	Memory pressure. Evict the lowest-priority running sequence. Retry.
4	Return decode batch	All sequences that passed step 2 are included. Model runner executes one token per sequence.

After the loop, restored sequences go back to the front of the running queue in original order. No sequence is forgotten.

One Decode Step: What Happens

Every running sequence generates exactly one token. Then the loop restarts.

Seq A

256 tokens so far

→ "intelligence"

at boundary → new block!

Seq B

100 tokens so far

→ "photosynthesis"

mid-block → no alloc

Seq C

511 tokens so far

→ "[EOS]"

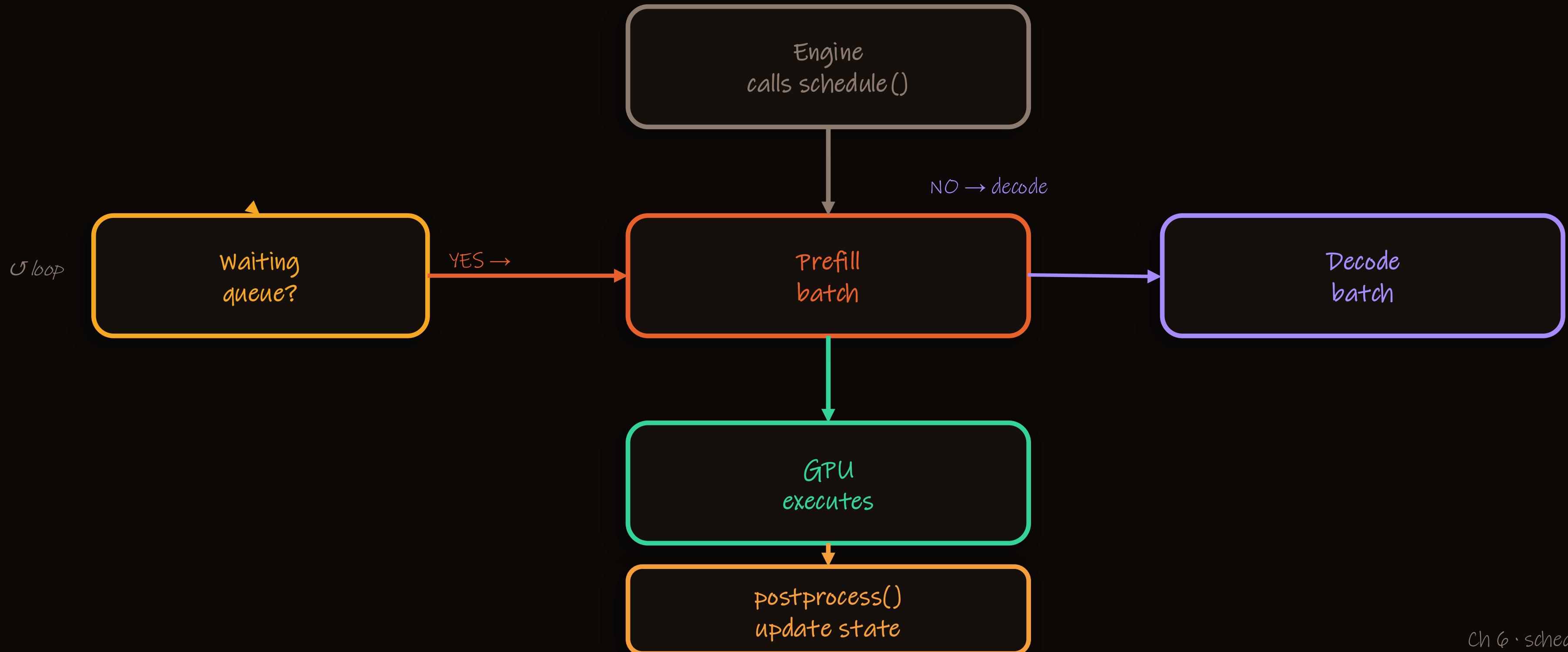
→ FINISHED, blocks freed

After the GPU runs this batch, postprocess() runs:

1. Append new token to each sequence's token_ids
2. Check: is the new token == EOS? → mark FINISHED
3. Check: tokens_generated == max_tokens? → mark FINISHED
4. For finished sequences: deallocate all KV blocks immediately

The Scheduler Loop: Complete Picture

The engine calls schedule() every step. This diagram shows the full cycle.



7

Prefix Caching

Why compute the same thing twice?

The Problem: Redundant Computation

Most requests share a common opening — the system prompt.

System prompt (same for ALL users): "You are a helpful assistant. Answer clearly and concisely."

system prompt (128 tokens) ← RECOMPUTED	User 1: "What is quantum computing?"
system prompt (128 tokens) ← RECOMPUTED	User 2: "Explain the water cycle."
system prompt (128 tokens) ← RECOMPUTED	User 3: "Write a haiku about autumn."

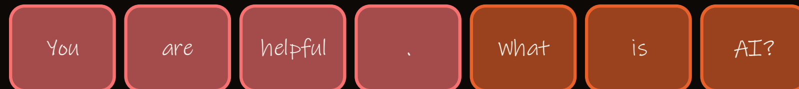
↑ *This 128-token block is computed from scratch every single time.*

In a real deployment: thousands of requests × same system prompt = enormous wasted compute.

Without Prefix Cache: Repeated Work

Two requests. Same prompt. Two full computations.

Request A



← ALL computed from scratch (7 tokens)

Request B

← SAME 4 system tokens computed again!

Total tokens computed: 13
Necessary tokens: 9
Wasted: 4 tokens (31%)

The cost at scale

If your system prompt is 256 tokens and you serve 10,000 requests per hour:

→ 2,560,000 tokens/hour of wasted KV computation

→ Longer latency for every request

→ Higher GPU utilisation just to recompute what you already know

Prefix caching eliminates this entirely.

With Prefix Cache: Reuse the Work

If we've seen this exact token sequence before, retrieve the stored KV — no recomputation.

Request A (first time)

← 7 tokens computed → system prefix CACHED (blocks #3, #7)

Request B (cache hit!)

← 4 tokens skipped (cache hit!), 2 tokens computed

Total tokens computed: 9
Necessary tokens: 9
Wasted: 0 tokens (0%)

How it works

1. When a block is full (256 tokens), compute a fingerprint (hash) of its content.

2. Store: hash → block_id in a lookup table.

3. When a new request arrives, hash its first block.

4. If hash is found: skip computation, reuse existing block.

5. Continue to next block with chained hash.

Shared Prefix Blocks: Visual

Two sequences sharing the same opening blocks.

Seq A block_table: [5, 5, 5, 7, 12] ← blocks 5 shared with B

Seq B block_table: [5, 5, 5, 4, 9] ← blocks 5 shared with A

Physical GPU memory:

How sharing works:

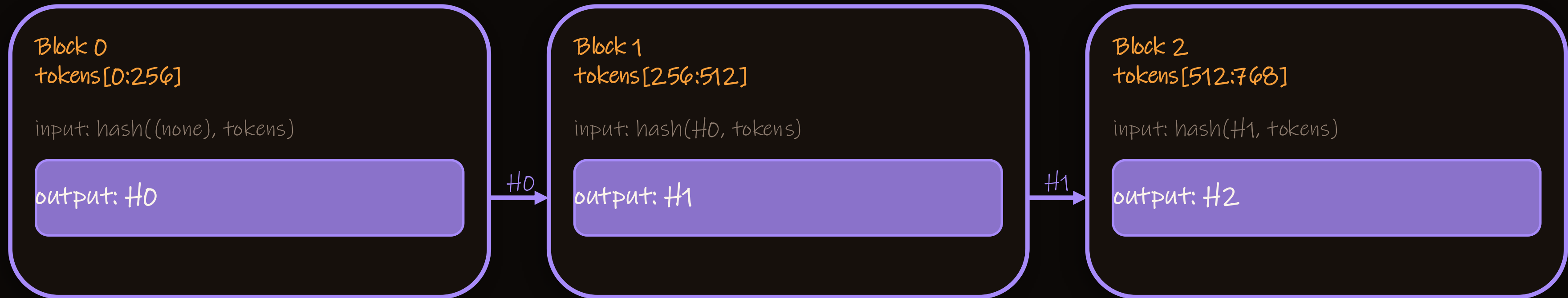
1. Block #5 appears in both sequences' block_tables — it is genuinely shared.
2. It has ref_count = 2 — the BlockManager will not free it while either sequence holds it.
3. When Seq A finishes, ref_count drops to 1. Block still held by Seq B.
4. When Seq B finishes, ref_count drops to 0. Block returned to free list.

Hash Chaining: Position-Sensitive Fingerprinting

A block's hash depends on its content AND every block before it.

Why chaining? Without it, two sequences with identical Block 1 but different Block 0 would get a false cache hit.

Hash chain for a 3-block prompt:



Like a git commit hash: it depends on the commit content AND the parent commit. Moving a commit to a different branch changes its hash — even if the content is identical.

Reference Counting: Safe Shared Ownership

A block shared by two sequences must not be freed when only one finishes.

ref_count tracks how many sequences currently hold a block. A block is only returned to the free list when ref_count reaches zero.

Time	Event	ref_count
t=0	Seq A allocated. Shared block #5 created.	ref_count(#5) = 1
t=5	Seq B arrives. Cache hit on block #5.	ref_count(#5) = 2
t=12	Seq A finishes. Deallocate called.	ref_count(#5) = 1 ← NOT freed
t=18	Seq B finishes. Deallocate called.	ref_count(#5) = 0 → FREED ✓

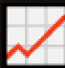
Reference counting makes shared blocks safe. Memory is freed exactly when the last holder releases it — never earlier.

Why Prefix Caching Matters in Practice

The benefits are measurable and significant.

Lower latency

Requests with cache hits skip prefill for shared tokens. Response starts faster.



Higher throughput

Skipped tokens don't count against the max_batched_tokens budget. More requests admitted per step.

Less compute

KV blocks that already exist are reused. GPU does less work for the same output.

Lower cost

For cloud deployments: fewer GPU cycles = lower inference cost per request.

Real-world impact:

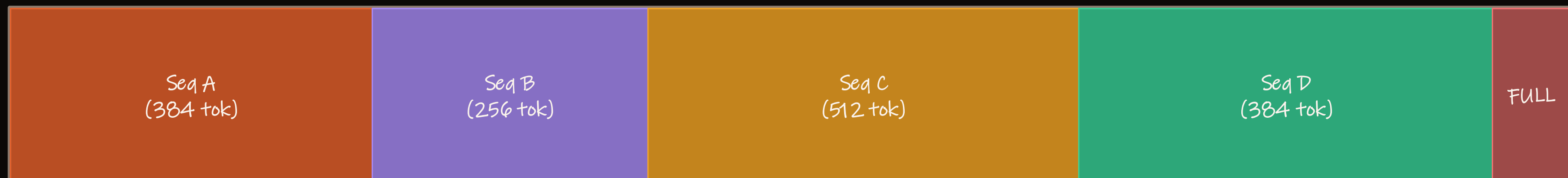
In a typical deployment with 256-token system prompts:
→ 100% of requests get a cache hit after the first request
→ Effective throughput on system-prompt tokens: near-infinite (no recomputation)
→ TTFT (time-to-first-token) improves proportionally to the cache hit fraction

Preemption

What to do when memory runs out mid-generation

Memory Full: The Crisis

All GPU memory is in use. A running sequence needs more space. What now?



⚠️ Seq A just crossed a 256-token boundary. Needs a new block. Zero free blocks.

Options:

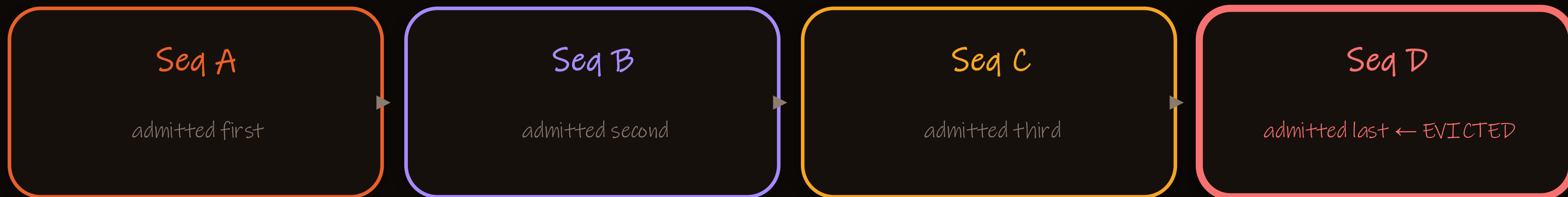
- ❌ Block Seq A: Tell it to wait. But it already has blocks allocated — wasteful and complex.
- ❌ Fail the request: Return an error. Terrible user experience.
- ✅ Preempt a lower-priority sequence: Temporarily remove one running sequence, free its blocks, use them for Seq A.

Who Gets Preempted?

The sequence at the back of the running queue — the lowest admission priority.

nano-vLLM's rule: evict the sequence that was admitted most recently.

running queue (front → back):



Why the most recently admitted sequence?

- It has generated the fewest tokens — redoing its prefill costs the least.
- It was admitted last, so other sequences have more invested work to protect.
- The evicted sequence is pushed to the FRONT of waiting — it gets priority when memory is available.
- This prevents starvation: an evicted sequence cannot be skipped by new arrivals.

Preemption: What Actually Happens

Temporarily remove a sequence to free its GPU memory.

Preemption = stop a running sequence + free its memory + put it back in waiting

BEFORE preemption

- Seq D is RUNNING
- Seq D holds blocks #8, #9
- All GPU memory occupied
- Seq A cannot grow

preempt() →

AFTER preemption

- Seq D is WAITING (front of queue)
- Blocks #8 and #9 returned to free list
- 2 free blocks available
- Seq A can allocate and continue

Three Lines That Do Everything

preempt() is surprisingly simple — each line has a specific purpose.

```
def preempt(self, seq):
    seq.status = SequenceStatus.WAITING
    self.block_manager.deallocate(seq)
    self.waiting.appendleft(seq)
```

Line 1: seq.status = WAITING

The sequence is no longer running. Status change prevents it from being included in the next decode batch. No lingering references.

Line 2: deallocate(seq)

Every block in block_table has its ref_count decremented. Blocks with count=0 return to free_block_ids. This is the actual memory being freed.

Line 3: waiting.appendleft(seq)

appendleft pushes to the FRONT of waiting — not the back. The evicted sequence gets higher priority than new arrivals. Anti-starvation.

Preemption Walkthrough: Step by Step

Running queue = [A, B, C, D]. Memory full. Seq A needs a block.

1. Seq A popped from front of running queue. can_append(A) = False. No free blocks.
2. running queue not empty → pick victim = running.pop() = Seq D (back of queue).
3. preempt(D): status=WAITING, deallocate(D) frees blocks #8 and #9, appendleft to waiting.
4. can_append(A)? Yes — 2 free blocks now available. may_append(A) allocates new block.
5. Seq A added to decode batch. B, C also added. Seq D is at front of waiting queue.
6. Next step: Seq D is first to be admitted in prefill (if budget allows).

The Cost of Preemption

Preemption is correct — but it forces a redo of expensive work.

Full prefill redo

The KV cache was discarded. Re-admission means reprocessing all prompt tokens AND all generated tokens so far.

Latency hit

The re-admitted sequence will experience a delay proportional to its total token count at the time of eviction.

Prefix cache helps

If the sequence's prompt was previously cached, num_cached_tokens will be non-zero — reducing the redo cost.

vs. full vLLM

vLLM supports swap-to-CPU: blocks are evicted to DRAM and swapped back, avoiding the full prefill redo. nano-vLLM omits this for simplicity.

9

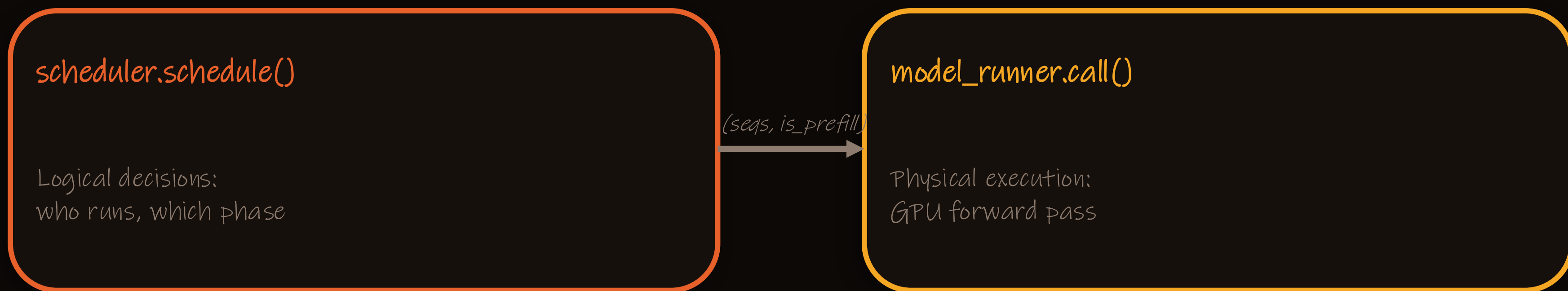
Scheduler to GPU Execution

How scheduling decisions become actual GPU work

Where schedule() Ends and Execution Begins

The scheduler decides. The model runner executes. They are separate systems.

schedule() returns a list of sequences and a phase flag. It does not execute anything on the GPU. That is the model runner's job.



The scheduler passes:

seqs	A list of Sequence objects. Each carries its block_table (list of physical block IDs).
is_prefill	True → run prefill forward pass. False → run decode forward pass (CUDA graph if possible).

Scheduler → ModelRunner Architecture

The full call chain from engine to GPU.

Block Tables: Logical Memory Map

How a sequence's block_table becomes a CUDA tensor.

The block_table is a list of integers. Each integer is a physical block ID. The model runner converts this into a GPU tensor so the attention kernel can find each token.

Seq A block_table =

3	7	12	2

slot_mapping calculation

For each token at position p:

block_id = block_table[p // 256]

offset = p % 256

slot = block_id × 256 + offset

Example: p = 775

block_id = block_table[3] = 2

offset = 775 % 256 = 7

slot = 2 × 256 + 7 = 519

Physical KV Cache: What Lives on GPU

A flat 4D tensor — indexed by block ID, position, head, dimension.

k_cache shape: [num_blocks, block_size, num_heads, head_dim]

num_blocks	~1,000+	Total blocks in the KV cache. Set at startup from GPU memory budget.
block_size	256	Tokens per block. Fixed at compile time. Hardware aligned.
num_heads	8–32	Number of attention heads (model architecture parameter).
head_dim	64–128	Dimension of each head's key/value vectors.

Concrete example (Qwen3-0.6B on FP16):

28 layers × 2 (K+V) × 8 heads × 128 dim × 2 bytes × 256 block_size = 0.875 MB per block

CUDA Graphs: Why Decode Gets Special Treatment

Fixed batch shape enables pre-recorded kernel replay — zero Python overhead.

Python dispatch overhead is significant for decode steps. CUDA graphs eliminate it by pre-recording the entire forward pass.

Without CUDA graphs

- Python: schedule()
- Python: build tensors
- CUDA: attention kernel
- Python: check outputs
- Python: postprocess()
- → Python overhead every step

With CUDA graphs

- Python: schedule()
- Python: build tensors
- GPU: replay graph
- (entire forward pass)
- (zero Python steps)
- → much faster decode

Why only decode? Decode has a fixed batch size (padded to power-of-2). Prefill varies in length every step — can't be graphed.

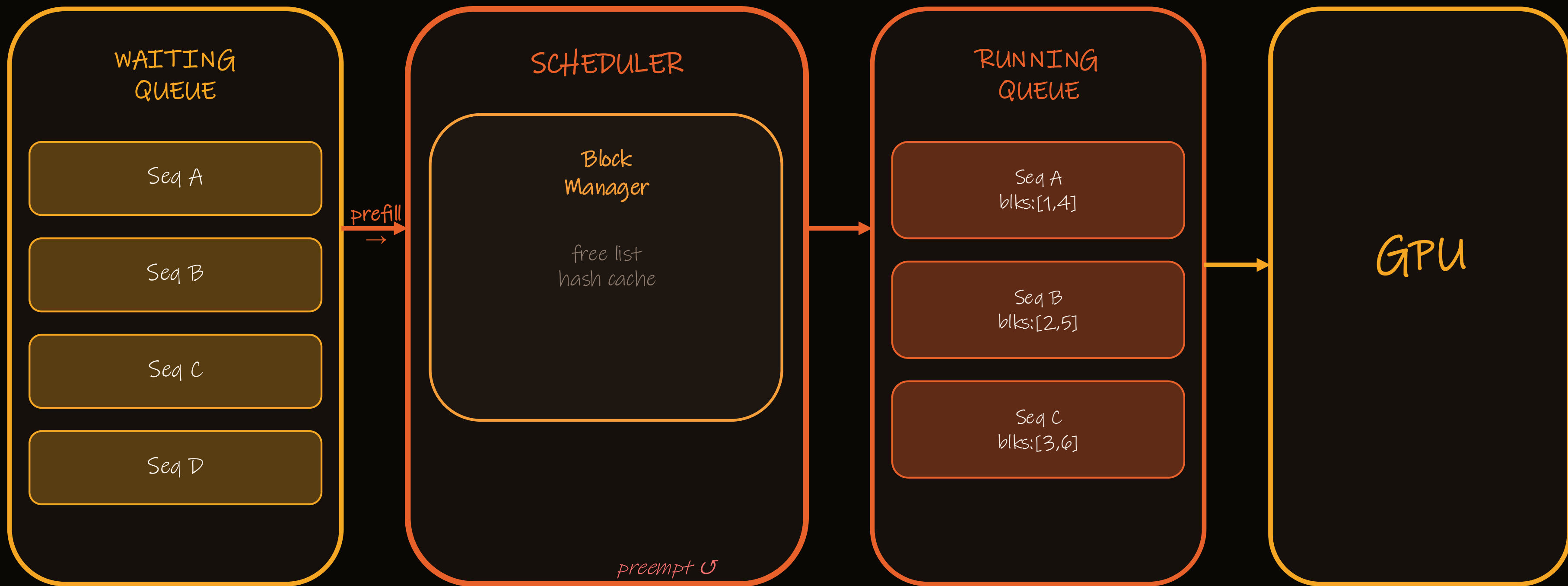
End-to-End Request Journey

From user message to generated response — every step.

#	Step	Details
1	Request arrives	add_request() → Sequence created → appended to waiting queue
2	Scheduler admits	can_allocate()? → allocate blocks → move to running → status=RUNNING
3	Prefill executes	model_runner forward pass (all prompt tokens) → KV cache built → first token returned
4	postprocess()	token appended → EOS? → max_tokens? → if not done: stay RUNNING
5	Decode × N steps	schedule() → decode → may_append at block boundaries → one token per step
6	EOS token generated	postprocess() marks FINISHED → deallocate all blocks → result returned

The Whiteboard Version

Everything on one diagram



Memory full? → Preempt

Repeated prompt? → Prefix cache hit

Final Takeaways

If you remember nothing else from this deck:

1. The scheduler exists to keep the GPU busy.
2. Prefill builds KV cache. Decode uses it.
3. Memory — not compute — is the real constraint.
4. Blocks make memory manageable.
5. Prefix caching reuses work across requests.
6. Preemption protects progress when memory is tight.
7. schedule() is three simple ideas: admit, decode, preempt.
8. Everything revolves around waiting, running, and memory.
9. The scheduler never touches the GPU. ModelRunner does.