nano-vllm-omni
Self-Study Handbook

From implementing Qwen-Omni to a multi-stage concurrent inference service

Part I Exercises: 26 one-hour core implementations

Part II Project: nano-vllm-omni on a single RTX 5090

Part III Reference: a deep dive into vLLM-Omni's concurrency design

Source versions: vllm-omni `fa8b4199c` (main) · vLLM v0.31.0 · transformers 4.57.1
Exercise code: `nano-vllm-omni-drills/` (`mine/` is what you write, `tests/` the acceptance tests, `solutions/` the reference solutions)
`path:line` references point to files and lines in the vllm-omni repo at that commit; open them in a local checkout.

Source & latest version: https://github.com/kzhou003/nano-vllm-omni-drills
© 2026 kzhou003. Please credit this repository when sharing or adapting.

How to use this handbook

1. **Part I (about 25 hours).** One exercise per sitting; the goal is to get one core mechanism right within an hour. Start with Track M to learn what each Qwen-Omni layer computes, then do the serving exercises starting from the orchestrator. Each exercise ends with a notes area for the pitfalls you hit.
2. **Part II (about 8–10 weeks).** Assemble the pieces from Part I into a real, usable service. Milestones go strictly in order; the parity tests you write early stay in place as regression tests for every later optimization.
3. **Part III (reference, any time).** Whenever you want to know how the real system does it and why, turn to the matching section and read the vllm-omni source at the given `path:line`.

How the parts map: Part I exercises ⇄ Part II milestones ⇄ Part III sections. For example, the P02/P03 orchestrator maps to M6/M8 in Part II and section 6 of Part III; the P12/P13 data plane maps to M7/M8 and section 8; the M05 talker MTP maps to sections 1 and 9.

Changes from the original HTML for print: all collapsed content is expanded; Part I hints and full test code are moved to the appendices at the end of Part I; links are shown as `path:line` text.

© 2026 kzhou003

Contents

Exercises: 26 one-hour core implementations 5
- How to use the exercises 6
- Big picture: Orchestrator → EngineCore → Scheduler → Sampler 7
- Exercise list (26 exercises, about 23 hours) 12
- A Design Framework for Concurrent Programs: T · SBWC · LIFE 14
- Track M · Qwen-Omni model implementation 26
  - #1 · M01 RMSNorm and multimodal RoPE (interleaved / chunked) 27
  - #2 · M02 Thinker attention: QK-norm + GQA + KV cache 31
  - #3 · M03 Sparse MoE: routing, per-expert grouped compute, shared expert 35
  - #4 · M04 Multimodal merge: placeholder counts, embedding merge, DeepStack 39
  - #5 · M05 Talker MTP: predicting 15 residual codebooks within one frame 43
  - #6 · M06 Code2Wav streaming causal conv and transposed conv 46
  - #7 · M07 Per-stage weight loading and packing 49
  - #8 · M08 Putting it together: KV-cache greedy decoding for a tiny thinker 52
- Track A · Frontend & Orchestrator 55
  - #9 · P02 Orchestrator v1: linear multi-stage routing 56
  - #10 · P03 Orchestrator v2: async_chunk prewarm + terminal gating 59
  - #11 · P01 OutputSink: batched wakeups from thread → asyncio 62
  - #12 · P04 StagePool: sticky binding, two-phase abort, fault isolation 65
  - #13 · P05 RpcRouter: RPC correlation over a shared channel 69
- Track B · Stage process & scheduling 72
  - #14 · P10 EngineCore busy loop 73
  - #15 · P11 WAITING_FOR_CHUNK: park and restore 76
  - #16 · P07 Scheduler: continuous batching + chunked prefill + preemption 80
  - #17 · P06 BlockManager: paged KV + prefix caching 85
  - #18 · P08 Batch-invariant sampler (per-request seed) 90
  - #19 · P09 Paged KV write + decode attention (reference implementation) 94
- Track C · Data plane 98
  - #20 · P12 SharedMemoryConnector: cross-process put/get protocol 99
  - #21 · P13 ChunkAdapter: background send/recv threads with ordering/cleanup semantics 103
- Track D · Model glue (pure functions) 108
  - #22 · P15 Thinker→Talker input construction (including "off by one row") 109
  - #23 · P14 Codec window planning: small first chunk, large later chunks, left context 113
  - #24 · P18 MRoPE 3D positions 116
- Track E · Startup & GPU 119
  - #25 · P16 Startup planning: device groups, parallel init, rollback, admission 120
  - #26 · P17 CUDA graph bucket runner (CPU simulation) 124
- Capstone: assemble the pieces into a mini omni service (half a day to a day) 127
- Appendix A · Three-level hints (look only after 15+ minutes stuck; L1 → L2 → L3 give away progressively more) 128
- Appendix B · Full test code 186
- Appendix C · Reference solutions (commented; Pitfall: marks easy-to-get-wrong spots) 225
- Appendix D · P02 Orchestrator ↔ vllm-omni source mapping 261

Project: build nano-vllm-omni from scratch 269
- 0. Assignment and final deliverables 270
- 1. Target model and hardware budget 270
- 2. Target architecture overview 272
- 3. Write-it-yourself vs borrow list 272
- 4. Recommended code layout 273
- 5. Milestone roadmap 274
  - M0 · Environment and golden reference 274
  - M1 · Weight splitting and per-stage loading 274
  - M2 · Single-stage LLM engine core (nano-vllm level) 275
  - M3 · Multimodal input and MRoPE 276
  - M4 · Talker stage (single-process chained version) 277
  - M5 · Token2Wav stage 277

© 2026 kzhou003

M6 · Multi-process topology and Orchestrator 278
M7 · Data-plane Connector (bypassing the orchestrator) 279
M8 · async_chunk streaming pipeline 280
M9 · OpenAI-compatible server 281
M10 · Performance: CUDA graph, IO/compute overlap 281
M11 · Correctness and robustness 282
6. Benchmarks and acceptance 283
7. Grading rubric (100 points + bonus) 283
8. Bonus items (+3 to +8 each) 284
9. Common pitfalls (look here when you hit a problem) 284
10. References 285
Reference: a deep dive into vLLM-Omni's concurrency design 286
0. How to read this 287
1. Background: the Qwen3-Omni model 287
2. vLLM core concepts crash course 290
3. Big picture: processes / threads / event loops 291
4. The life of a request (async_chunk mode) 293
5. Frontend ↔ Orchestrator: cross-thread queues 293
6. Orchestrator event-driven loop 294
7. Inside a stage process 296
8. async_chunk data plane 298
9. GPU-level concurrency: CUDA streams / events / graphs 301
10. Correctness: abort / failure / shutdown / RPC 302
11. Startup-time concurrency 303
12. MRv2 native data plane (brief) 304
13. Design trade-offs and known risks 304
14. Recommended source reading path 304

© 2026 kzhou003

PART I

Exercises: 26 one-hour core implementations

Practice the pieces first: 8 Qwen-Omni model exercises checked against HF transformers, and 18 serving-mechanism exercises covering the orchestrator, scheduling, the data plane, start-up and GPU. Each comes with a full spec, interface skeleton, test checklist and follow-up questions; hints and test code are in the appendices at the end of this part.

© 2026 kzhou003

How to use the exercises

1. Code lives in `nano-vllm-omni-drills/`: implement in `mine/` and check with `pytest tests/test_<id>.py`. Reference solutions are in `solutions/`; `DRILL_IMPL=solutions pytest` shows them all passing.
2. Time-box each exercise: 5 minutes to read the spec and list the edge cases you can think of, 45 minutes to implement, 15 minutes to run the tests and fix bugs.
3. Suggested order: **Track M (understand the model first)** → A → B → C → D → E. Dependencies: M08 uses your M01, M02, M03 and M07; P03 extends your P02; everything else is independent.
4. The print edition is ordered by how much it gives away: each exercise page has only the spec, the skeleton, what the tests check, **analogies** and a **framework-view table**; **Appendix A** has three-level hints (L1 I/O examples and key questions → L2 approach and pitfalls → L3 walkthrough), **Appendix B** the full tests, **Appendix C** the commented reference solutions, and **Appendix D** the P02 ↔ vllm-omni source mapping.
5. Read the next chapter, the **big picture**, first: it places all 26 exercises on one layered map (Orchestrator → EngineCore → Scheduler → Sampler, with the data plane alongside); the colored exercise tags link straight to each exercise.
6. Interactive versions (React Flow step-throughs with hidden answers; Chinese only) are in `nano-vllm-omni-drills/hints/`: one `<id>_overview.html` per exercise, plus `p02_flow.html` for P02; start at `hints/index.html`. The "Interactive" line on each exercise page names the file.
7. Environment: Python ≥ 3.10, torch, pytest; everything runs on CPU. Track M also needs `transformers==4.57.1`, which the tests use as the parity reference.

© 2026 kzhou003

Big picture: Orchestrator → EngineCore → Scheduler → Sampler

In one sentence: the Orchestrator conducts several stages; each stage has its own engine, EngineCore; on every turn of the engine (a step), the Scheduler first decides which requests get on board this step, the model forward computes logits, and then the Sampler picks the next token for each request. Bulk data between stages bypasses the conductor and travels on the side-channel data plane (connector).

Problem colors: A Frontend & Orchestrator B Stage process & scheduling C Data plane D Model glue E Startup & GPU M Model implementation

1. Layered overview

User / API server

add_request(prompt) → async-iterate OrchOutput until finished=True

▼

Frontend process · Orchestrator (one asyncio event loop managing all stages)

Owns cross-stage concerns: which stages a request passes through in order, when to submit downstream, when to send the single terminal message, and notifying every stage on abort. It does not care about the KV cache or how batches are formed.

Routing & terminal message

P02 Linear multi-stage routing, single-writer dispatcher

P03 async_chunk: prewarm downstream stages, hold back the terminal message until every stage is done

Replicas & abort

P04 StagePool: sticky request-to-replica binding, two-phase abort, fault isolation

Thread ↔ event loop

P01 OutputSink: the output thread uses `call_soon_threadsafe` to wake the event loop with coalesced wakeups

P05 RpcRouter: on a shared reply channel, route results by key back to the blocked caller

▼ StageClient: submit / get_output / abort / utility (ZMQ, cross-process, one per stage)

Stage process · EngineCore (one per stage: thinker / talker / code2wav ...)

Input thread → input_queue → **main-thread busy loop** → output thread. The main thread is the single writer of engine state. It does not know other stages exist.

busy loop

P10 Block while idle; drain ADD / ABORT / UTILITY / WAKEUP in FIFO order; if there is work, step once and emit outputs; handle fatal and utility errors separately

▼ Each engine.step() goes through, in order:

© 2026 kzhou003

① Scheduler: which requests run this step, and how many tokens each

P07 continuous batching, chunked prefill, preemption

P06 BlockManager: paged KV block allocation, prefix caching (called by P07)

P11 ChunkAwareScheduler: wraps P07; requests whose next chunk hasn't arrived are parked, and restored once it arrives

② Model forward: logits [B, V]

M01–M08 thinker / talker / MTP / code2wav layers

P09 Write KV via slot_mapping; decode attention gathers KV via the block table

P18 MRoPE 3D positions

P17 decode uses bucketed CUDA graph replay

③ Sampler: what is each request's next token

P08 Row by row: repetition penalty → temperature → top-k/top-p → multinomial; each seeded request has a private generator, so results are independent of batch composition

Data plane (parallel to the control plane, bypasses the Orchestrator)

Upstream stage → P13 ChunkAdapter background save thread → P12 ShmConnector (shared memory, keyed put/get, read-once consumption) → downstream stage's P13 background recv thread → chunk ready → P11 restores the request into scheduling.

How payloads are built: P15 thinker hidden states into talker prefill input, P14 codec frame windows handed to code2wav.

Startup (before everything above runs)

P16 Group stages that share a GPU; initialize serially within a group and in parallel across groups, roll back on failure, admission-check GPU memory · M07 Load weights per stage · P17 capture CUDA graphs

2. What each of the four core components owns

Component	Where it runs	Responsible for	Not responsible for	Problems	vllm-omni counterpart
Orchestrator	Frontend process, one event loop	Cross-stage routing; exactly one finished=True per request, and it is the last; fan out abort to all stages; replica selection	Batching, KV, sampling; in async_chunk mode it doesn't move data either	P01 P02 P03 P04 P05	`vllm_omni/engine/orchestrator.py` `stage_engine_core_client.py`
EngineCore	One process per stage; main thread runs the busy loop	Receive messages, drive steps, hand outputs to the output thread; distinguish fatal vs utility errors; orderly shutdown	Knowing other stages exist; how exactly batches are formed (delegated to the scheduler)	P10	`vllm_omni/engine/stage_engine_core_proc.py` (subclasses vLLM `EngineCoreProc`)

© 2026 kzhou003

Scheduler	Inside the EngineCore main thread	Each step, pick requests within the token budget and assign tokens per request; allocate KV blocks; preempt when out of space; under async_chunk, park requests whose data hasn't arrived	Computing logits, picking tokens	P07 P06 P11	`vllm_omni/core/sched/` (`omni_ar_scheduler.py`, `omni_generation_scheduler.py`)
Sampler	Inside the EngineCore main thread, after the forward	logits [B, V] → token [B]; each request uses its own sampling params and random stream	Which requests take part in this step (the scheduler already decided)	P08	vLLM v1 Sampler (`vllm/v1/sample/`)

> Both Scheduler and Sampler "select", but different things: the Scheduler selects **which requests** take part in this step and how many tokens each computes; the Sampler selects the **next token** for each participating request.

3. One request end to end (tying the problems together)

Example: thinker(S0) → talker(S1) → code2wav(S2), final={0,2}, async_chunk mode. Blue is the control plane, teal is the data plane, red is the error path.

1. **Startup**: P16 plans the stage initialization order and checks GPU memory; each stage process loads its share of weights with M07, captures CUDA graphs with P17, then enters the P10 busy loop and blocks while idle.
2. **Request arrives**: the user calls add_request. P03 registers request state before any await and sets submitted to {0,1,2} in one go; then S0 gets the real prompt while S1 and S2 get placeholder inputs (prewarm). If a stage has several replicas, P04 first picks a replica and binds stickily.
3. **Cross-process**: StageClient sends ADD over ZMQ. The stage process's input thread puts the message into input_queue; the P10 main loop wakes up and handles it before the next step.
4. **Scheduling**: P07 admits requests within the token budget and does chunked prefill; it requests KV blocks from P06, reusing them on a prefix hit. S1 and S2 have no data yet, so P11 parks them and the underlying scheduler never sees them.
5. **Forward**: S0 thinker computes M01–M04 (RMSNorm/RoPE, attention, MoE, multimodal merge) with P18 MRoPE positions; P09 writes KV and runs decode attention; the decode phase uses P17 graph replay.
6. **Sampling**: P08 picks a token for each row. P10 emits the results to the output thread, and they return to the frontend over ZMQ.
7. **Data plane · send**: for each segment S0 produces, the P13 save thread builds talker input using the P15 rules (trimming the stop row) and writes it to P12 shared memory under key `{rid}_{stage}_{chunk_id}`.

© 2026 kzhou003

8 **Data plane · receive**: S1's P13 recv thread fetches data in chunk_id order into the ready queue → P11 restores the request → S1 talker generates codec frames with M05 MTP → sent to S2 via the connector the same way; S2 follows the P14 window plan (small first chunk, with left context) and calls M06 streaming code2wav to output audio.

9 **Back to the frontend**: each stage's output thread wakes the event loop with coalesced wakeups via P01 OutputSink → reader → the P02/P03 single-writer dispatcher. Outputs from S0 and S2 go to the user; S1 is not a final stage, so its outputs are not sent to the user and only record completion.

10 **Termination**: when all_final holds but S1 hasn't reported completion yet, P03 holds back the last message; only once `submitted ⊆ finished` holds does it release the single finished=True, then clean up. Any later outputs are dropped.

11 **abort**: P04 two-phase: first collect partials read-only, then commit after every stage's P10 has processed ABORT and acknowledged; unread segments left in the connector are reclaimed by P12's cleanup_prefix and P13's cleanup logic.

12 **Admin calls and crashes**: UTILITY enters P10 with a call_id; when the result returns to the frontend, P05 hands it back to the matching caller; a failing utility only affects that call. On a CUDA error during step, P10 emits fatal → the frontend's P05 `fail_all` fails every in-flight call immediately, and P04 isolates the faulty replica.

4. The same patterns recurring across layers

Many problems solve the same class of race at different layers. Once you recognize these patterns, later problems go much faster.

Pattern	Why it's needed	Where it appears
Register first, then yield or send	The reply may come back faster than the "send" returns. Without registering in advance, the reply is treated as unknown or late and dropped, and the caller hangs	P02/P03 register ReqState and mark submitted before await · P05 register before send · P13 register via load_async before polling
Single writer	If only one task or thread modifies the state, no locks are needed and no interleaving can occur	P02 the sole dispatcher · P10 engine state modified only on the main thread · P04 sync methods on the same event loop
A blocked party must be woken explicitly	Changing a shared flag alone is invisible to a party that is blocked. You must put something into the channel it is waiting on	P01 `call_soon_threadsafe` (writes to a self-pipe under the hood to wake select) · P10 push a WAKEUP on shutdown · P05 fail_all puts an exception into every mailbox
Exactly once	Terminal messages and result delivery may each happen only once	P02/P03 one finished=True per request · P05 pop guarantees each mailbox is delivered once · P12 delete on read
Late messages after cleanup: drop, never resurrect	Recreating state for a late message leads to a second terminal message or a state leak	P02/P03 dropped_outputs · P05 late_results · P05 unregister compares by identity so it never removes a new waiter with the same key
Latch fatal errors, fail fast	Once the process can't be trusted, fail everyone immediately instead of letting each wait for its own timeout	P10 emits fatal · P05 `_fatal` latch · P16 short-circuit initialization and roll back

© 2026 kzhou003

Independent of batch composition	Under continuous batching the batch changes every step; results must not change with it	P08 row-by-row sampling, private generator per request · P17 padding rows don't affect real rows

5. Common points of confusion

Q: Is EngineCore a layer above the Orchestrator, connecting the orchestrator to the scheduler?
The opposite. The Orchestrator is at the top and remotely drives several EngineCores through StageClient. EngineCore is the execution engine of a single stage, and the scheduler lives inside EngineCore. EngineCore doesn't know the orchestrator exists; it just receives messages, steps, and sends outputs.
Q: StageClient and the connector are both cross-process communication. What's the difference?
StageClient (ZMQ) is the **control plane**: it carries small messages like ADD/ABORT/UTILITY plus per-step outputs, and connects the orchestrator to stages. The connector (P12/P13) is the **data plane**: it moves bulk data such as hidden states and codec frames directly between stages, bypassing the orchestrator. The difference between P03 and P02 is exactly that data moves from "forwarded by the orchestrator" to "via the connector".
Q: How do P11 and P07 relate?
P11 is a layer wrapped around P07 (decorator-style). Before each scheduling pass it pulls requests whose next chunk hasn't arrived out of P07's queues, so P07 never needs to know chunks exist. After scheduling (including when it raises) it puts the requests back into their original queues with their original state.
Q: How does P03 know the data really arrived through the connector?
It doesn't. The orchestrator only looks at the finished status each stage reports. Data correctness is guaranteed by the connector protocol (key addressing, contiguous chunk_id, read-once consumption) and proven in tests by comparing outputs end to end. To detect it at runtime, you would need to add chunk-count checks and timeouts to the protocol.
Q: Why does the Sampler process row by row instead of one multinomial over the whole batch?
When sampling the whole batch at once, random numbers are assigned by row position. When other requests join or leave, the random numbers a seeded request receives change, and results become irreproducible. Real implementations vectorize on GPU but still keep an independent random stream per request.

6. Suggested problem order

Order	Problems	Rationale
1	P01 → P02 → P03	P03 builds directly on P02; P01 is the "thread to event loop" segment on P02's input side
2	P05 → P10 → P04	P10's UTILITY and fatal match P05's correlation and fail_all; P04's abort relies on P10's ABORT semantics
3	P06 → P07 → P11	P07 calls P06 to allocate blocks; P11 wraps P07
4	P09 → P08	Forward and sampling inside a step; P09's slot definition matches P06
5	P12 → P13	P13 uses the connector interface (the problem substitutes an in-memory implementation for P12)
6	P15 P14 P18, P16 P17	Pure-function and startup problems, relatively independent; can be interleaved

© 2026 kzhou003

In parallel	`M01–M08`	The model itself, i.e. the "forward" box in a step; M08 uses M01/M02/M03/M07

Exercise list (26 exercises, about 23 hours)

#	Track	Exercise	Min	Level	File
#1	M	RMSNorm and multimodal RoPE (interleaved / chunked)	45	medium	`m01_rmsnorm_mrope`
#2	M	Thinker attention: QK-norm + GQA + KV cache	60	medium	`m02_thinker_attention`
#3	M	Sparse MoE: routing, per-expert grouped compute, shared expert	60	medium	`m03_sparse_moe`
#4	M	Multimodal merge: placeholder counts, embedding merge, DeepStack	45	medium	`m04_mm_merge_deepstack`
#5	M	Talker MTP: predicting 15 residual codebooks within one frame	60	hard	`m05_talker_mtp`
#6	M	Code2Wav streaming causal conv and transposed conv	60	hard	`m06_streaming_causal_conv`
#7	M	Per-stage weight loading and packing	45	medium	`m07_stage_weight_loading`
#8	M	Putting it together: KV-cache greedy decoding for a tiny thinker	60	medium	`m08_tiny_thinker_e2e`
#9	A	Orchestrator v1: linear multi-stage routing	60	medium	`p02_orchestrator`
#10	A	Orchestrator v2: async_chunk prewarm + terminal gating	45	medium	`p03_async_chunk_orchestrator`
#11	A	OutputSink: batched wakeups from thread → asyncio	45	medium	`p01_output_sink`
#12	A	StagePool: sticky binding, two-phase abort, fault isolation	60	medium	`p04_stage_pool_abort`
#13	A	RpcRouter: RPC correlation over a shared channel	40	easy	`p05_rpc_router`
#14	B	EngineCore busy loop	45	medium	`p10_engine_core_loop`
#15	B	WAITING_FOR_CHUNK: park and restore	45	medium	`p11_chunk_aware_scheduler`
#16	B	Scheduler: continuous batching + chunked prefill + preemption	75	hard	`p07_scheduler`
#17	B	BlockManager: paged KV + prefix caching	75	hard	`p06_block_manager`
#18	B	Batch-invariant sampler (per-request seed)	45	medium	`p08_sampler`
#19	B	Paged KV write + decode attention (reference implementation)	45	medium	`p09_paged_attention`
#20	C	SharedMemoryConnector: cross-process put/get protocol	60	hard	`p12_shm_connector`

© 2026 kzhou003

#21	C	ChunkAdapter: background send/recv threads with ordering/cleanup semantics	75	hard	`p13_chunk_adapter`
#22	D	Thinker→Talker input construction (including "off by one row")	40	easy	`p15_talker_inputs`
#23	D	Codec window planning: small first chunk, large later chunks, left context	40	easy	`p14_codec_windows`
#24	D	MRoPE 3D positions	40	medium	`p18_mrope_positions`
#25	E	Startup planning: device groups, parallel init, rollback, admission	45	medium	`p16_init_planner`
#26	E	CUDA graph bucket runner (CPU simulation)	40	easy	`p17_graph_runner`

© 2026 kzhou003

A Design Framework for Concurrent Programs: T · SBWC · LIFE

A checklist to walk through in order when writing asyncio / multithreaded concurrent programs · Running example throughout: `mine/p02_orchestrator.py` (multi-stage Orchestrator)

First fix the venue T, then build the skeleton S B W C, then bring it to life L I F E

T · Topology
Which processes / threads / event loops does the code run in? Where are the boundaries?

S · State (the ledger)
What state is shared? Who is allowed to write it?

B · Blocking points
Where will the program sit and wait?

W · Workers (roles)
Who waits on what, does what, and is forbidden from doing what?

C · Channels
What carries messages between workers? What do the messages look like?

L · Lifecycle
How does it start? How does it stop cleanly?

I · Interface
What does the public API do, and nothing more?

F · Flow + Invariants
Write the rules that must always hold first, then the logic

E · Edges + Errors
What about late, out-of-order, unknown, or failing messages?

T decides **where things run and which synchronization primitives you may use**; SBWC decides **what each execution environment looks like**; LIFE ensures it **starts, stops, and doesn't break**.

0. Example: what the P02 Orchestrator must do

A request passes through several stages in order (analogous to Qwen3-Omni: thinker → talker → code2wav). Each stage is a separate process; the orchestrator can only interact with it through `StageClient`:

```
class StageClient(Protocol):
    async def submit(self, request_id, inputs) -> None          # hand the request to this stage
    async def get_output(self) -> list[StageOutput]            # block until this stage emits a batch of outputs
```

What the Orchestrator must do:

- After the user calls `add_request(rid, prompt)`, hand it to stage 0;
- When stage s finishes a request, build the input with `forward_fn(s+1, prompt, all of stage s's data)` and hand it to stage s+1;
- Outputs of stages in `final_stage_ids` are converted to `OrchOutput` and delivered to the user (`next_output()`);
- Each request gets **exactly one** `finished=True`, and it is the last one; then its state is cleaned up.

Concrete example used throughout: 3 stages, `final_stage_ids = {0, 2}`, and `forward_fn` returns the previous stage's data unchanged.

stage	Receives	Chunks emitted	Visible to user?
S0 thinker	`"hi there"`	`"hi"`, `"there"`	Yes (final)
S1 talker	`["hi", "there"]`	`["HI", "THERE"]` (1 chunk)	No (intermediate)

© 2026 kzhou003

S2 code2wav	`[["HI", "THERE"]]`	`"wav:HI"`, `"wav:THERE"`	Yes (final)

The user ends up receiving 4 `OrchOutput`s in order: two from stage 0 and two from stage 2; only the last one has `finished=True`.

© 2026 kzhou003

Step 0, T: fix the venue first — where does the code run?

T Topology · Execution topology

Ask yourself

How many **processes** are needed? Which **threads** live in each process? Which thread runs the **asyncio event loop**? Which boundaries are **already fixed** by the problem or the system?

Rule

T decides the **kinds of execution environments and their boundaries**; W decides **how many workers live in each environment**. Settle T first: every boundary you cross changes which communication mechanisms and synchronization primitives are available, and S, B, and C are all constrained by it.

How to choose

Execution unit	When you need it	Cost	In this problem set
Process	Fault isolation (one crash doesn't take down the rest); CPU-bound work that must bypass the GIL; exclusive resources per GPU / per model	No shared memory; cross-process data must be serialized or go through shared memory; slow to start	stage processes; P12 / P13 cross-process data plane; P16 launch planning
Thread	You must call a **blocking synchronous API** (socket / zmq recv, file locks, C extensions) without stalling the event loop	Shared memory but needs locks; GIL; prone to races	P01 producer thread; P13 background send/recv threads; P10 busy loop
asyncio task	Lots of I/O waiting, cooperating within one event loop	Switches only at `await`; any blocking call stalls the whole loop	All workers in P02 / P03 / P04 / P05

Hierarchy & boundaries

```
process ⊃ thread ⊃ event loop (at most one per thread) ⊃ task
```

Boundary crossed	Communication	Synchronization primitives
process ↔ process	pipe / zmq / shared memory (`multiprocessing.shared_memory`)	File locks, protocol conventions (who creates, who frees)
thread ↔ thread	`queue.Queue`, `collections.deque` + lock	`threading.Lock` / `Event` / `Condition`
thread → event loop	`loop.call_soon_threadsafe(...)`, `asyncio.run_coroutine_threadsafe(...)`	These two "bridges" are the only way into the loop
task ↔ task (same loop)	`asyncio.Queue`	`asyncio.Event` / `Lock`, or simply a single writer with no lock

P02 answer

© 2026 kzhou003

In P02 this step is **already fixed by the problem**:

- The orchestrator runs on **one** asyncio event loop in the API process;
- Each stage is a separate process, but `StageClient` wraps it in an `async` interface (`submit` / `get_output` can be awaited directly), so the orchestrator never sees the process boundary;
- So P02 needs **no threads at all**; every worker is a task on the same loop, and you can safely use "single writer, no locks".

Compared with the real vllm-omni: API process (asyncio: orchestrator, external HTTP) + one process per stage (running the EngineCore busy loop, see P10) + background send/recv threads inside stage processes (moving hidden states / codec across processes, see P13) + an OutputSink that carries results from threads back into the event loop (see P01). Every problem is really practice for communication across **one particular boundary**.

What happens if T is wrong: call a blocking `zmq.recv()` or `time.sleep()` directly in an asyncio task and the whole event loop (including every other request) stops; call `asyncio_queue.put_nowait()` directly from a plain thread and you occasionally lose wakeups or get errors, and it is very hard to reproduce.

Skeleton SBWC: decide what each execution environment looks like

S State · The ledger

Ask yourself What **mutable shared state** is there? **Who** writes each piece?

Rule

Give each piece of mutable state **a single writer** wherever possible. Reach for locks only when you can't.

P02 answer

State	Contents	Writer
`request_states: dict[rid, ReqState]`	Progress of each in-flight request: `submitted` , `finished` , `stage_data`	Registration: request_handler; everything else: **dispatcher**
`dropped_outputs: int`	Count of dropped outputs for unknown requests	dispatcher

asyncio is single-threaded, so two tasks never execute Python code "at the same time"; request_handler's registration is a dict assignment with no `await` , so it cannot interleave with the dispatcher. The real risk lives at `await` points; see F.

What happens if you violate it: if every reader updates `ReqState` itself on receiving an output, you have N writers. Outputs from S0 and S2 can interleave across an `await` ; someone cuts in between the read and the write of "have all final stages finished?", and you get two `finished=True` or none at all.

© 2026 kzhou003

B Blocking · Blocking points

Ask yourself

Where will the program **sit and wait**? List every `await xxx()` / `.get()`. Which ones are **independent of each other**?

Rule

Give each independent blocking source a dedicated worker that watches it. Never have one worker take turns waiting on several blocking sources.

P02 answer

Blocking point	When it returns	Count
`stages[i].get_output()`	When stage i emits output; timing is unpredictable and it may not return for a long time	N (one per stage)
`_req_q.get()`	When the user submits a new request	1
`_ready_q.get()`	When any stage's output has been moved in	1
`_out_q.get()` (inside `next_output`)	When there is a result for the user	1 (the caller is waiting)

What happens if you violate it (counterexample): have the dispatcher poll the stages itself in turn:

```
while True:
    for sid, client in enumerate(self.stages):
        outs = await client.get_output()     # ← can only wait on one at a time
        ...
```

Time	What the dispatcher is doing	S0	S1
t1	`await S0.get_output()`, gets `"hi"`	Has emitted "hi"	No work yet
t2	`await S1.get_output()`	**"there" emitted, nobody reads it**	No work, **will never produce output**
t3...	Stuck on S1 forever	Still waiting	S1's work is only submitted after S0's "there" is processed, but nobody reads "there"

The result is a **deadlock**: the dispatcher waits on S1, S1 waits on the dispatcher to forward, and forwarding requires first reading S0's "there".

© 2026 kzhou003

W Workers · Roles

Ask yourself What does each worker **wait on**? What does it **do**? What is it **forbidden from doing**?

Rule

The narrower each responsibility, the better. **Movers only move; the decision-maker only decides.** Writing down what each worker "must not do" prevents more bugs than writing what it does.

P02 answer

worker	Count	Waits on	Does	Must not
`_reader(sid)`	N	`stages[sid].get_output()`	Tag with `sid`, put into `_ready_q`	Inspect contents, touch `request_states`
`_request_handler`	1	`_req_q.get()`	Register `ReqState`, submit to stage 0	Handle stage outputs
`_dispatcher`	1	`_ready_q.get()`	The sole decision-maker: record data, check final, forward, deliver to user, clean up	Wait on stages directly (that's the readers' job)

> **Office analogy:** the dispatcher is the manager, and only the manager may edit the ledger; each phone (stage) has a secretary (reader) next to it who, when it rings, writes the message on a slip and drops it into the manager's inbox (`_ready_q`). The manager watches only the inbox and handles slips one at a time.

C Channels

Ask yourself

What carries data between workers? What is the **message format** on each channel? Who produces, who consumes?

Rule

With multiple producers and one consumer, merge into **a single queue (fan-in)** and include a **source tag** in each message. Fix the message format up front and document it in a comment.

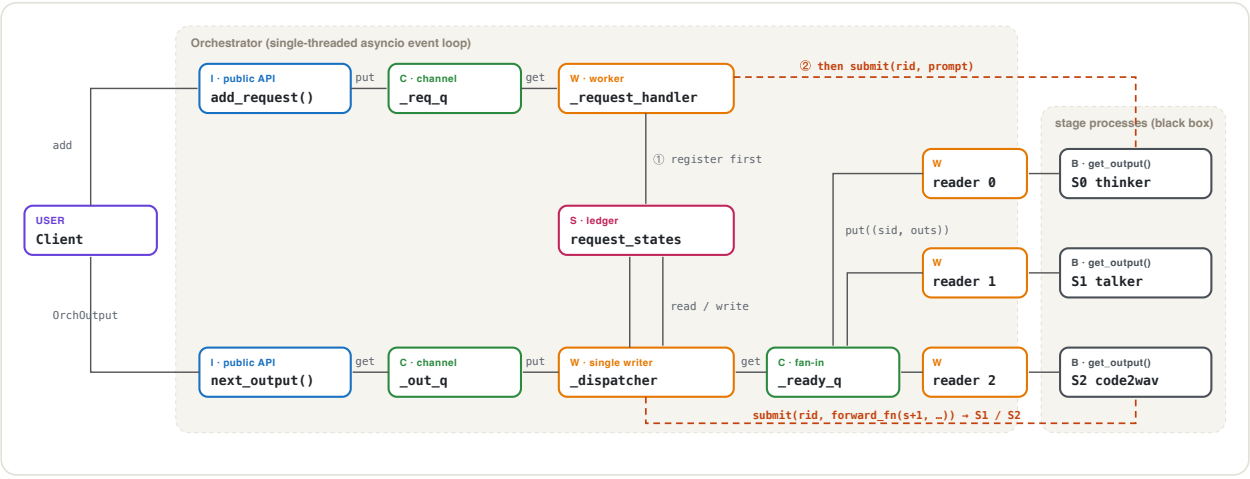
P02 answer

Channel	Producer → consumer	Message format
`_req_q`	`add_request` (caller) → request_handler	`("add", rid, prompt)`
`_ready_q`	N readers → dispatcher (**fan-in**)	`(sid, list[StageOutput])`
`_out_q`	dispatcher → `next_output` (caller)	`OrchOutput`

> **Easy to forget:** if messages in `_ready_q` don't carry `sid`, the dispatcher can't tell which stage an output came from: it can neither decide whether it is final nor where to forward it.

© 2026 kzhou003

Big picture: what the system looks like once SBWC is filled in



Orange dashed = submit to a stage; gray solid = queue put / get. Only the dispatcher and (at registration) the request_handler touch the ledger.

LIFE: make it start, stop, and not break

L Lifecycle

Ask yourself

When are workers created? How do they stop? When stopping, will any task be left hanging on an `await`?

Rule

`start`: create_task and **keep the handles**. `shutdown`: first `cancel()`, then `gather(..., return_exceptions=True)` to wait until they actually exit.

General pattern

```
async def start(self):
    for sid, client in enumerate(self.stages):          # one worker per blocking source (B → W)
        self._tasks.append(asyncio.create_task(self._reader(sid, client)))
    self._tasks.append(asyncio.create_task(self._request_handler()))
    self._tasks.append(asyncio.create_task(self._dispatcher()))

async def shutdown(self):
    for t in self._tasks:
        t.cancel()                                       # raises CancelledError at each task's await
    await asyncio.gather(*self._tasks, return_exceptions=True)   # wait until they really finish, swallowing
CancelledError
```

> **Common pitfalls:** ① not keeping task references, so tasks may be garbage-collected and shutdown can't find them; ② cancel without gather, so the test ends with "Task was destroyed but it is pending"; ③ awaiting something inside `start()` that never returns, so `start()` itself hangs.

© 2026 kzhou003

I Interface

Ask yourself

What can the caller do? Do public methods touch internal state? Whose context do they run in?

Rule

Keep the public API **thin**: it only puts into a channel or gets from one. Leave every decision to internal workers.

P02 answer

API	Only does	Does not
`add_request(rid, prompt)`	`_req_q.put(("add", rid, prompt))`	Register state, submit
`next_output(timeout)`	`wait_for(_out_q.get(), timeout)`	Assemble results, modify state

> Benefit: callers may come from any coroutine, or even from another thread via a bridge. Because the interface only posts messages, it doesn't care about the caller's context, and the ledger is only ever modified internally.

© 2026 kzhou003

F Flow + Invariants · Core logic and invariants

Ask yourself

Write down the rules that **must always hold** before writing the handling logic. Check each `await` in turn: **if another worker runs right now, is the state it sees correct?**

Rule `await` = yielding control. **State must be self-consistent before the await.**

P02 invariants

1. Each request gets **exactly one** `OrchOutput.finished=True`, and it is that request's **last** output.
2. `OrchOutput.finished` means "the **whole request** is done", i.e. `final_stage_ids ⊆ ReqState.finished`; `StageOutput.finished` only means "**this stage** is done". They mean different things.
3. Only outputs from stages in `final_stage_ids` reach the user.
4. When stage s finishes and `s < max(final_stage_ids)`, forward **all** of stage s's data to s+1. "Deliver to user" and "forward" are independent actions, not either/or.
5. Once the terminal message is sent, delete the request's state immediately.

Classic example of an await switch point: why must you "register ReqState first, then `await stages[0].submit()`"?

Order	Timeline	Result
❌ submit first, register later	`await submit()` yields control → S0 quickly emits "hi" → reader moves it into `_ready_q` → dispatcher looks up `request_states`: **r1 not found** → dropped as an unknown request → back in the handler, registration only happens now	The user never receives "hi", `dropped_outputs=1`
✅ register first, then submit	Register (no await, cannot be interrupted) → `await submit()` → whoever runs next, the dispatcher can find r1	Correct

For more step-by-step state changes, see the interactive sequence diagram `hints/p02_flow.html`. This page covers design only and does not give the `_handle_output` implementation.

© 2026 kzhou003

E Edges + Errors · Edge cases and failures

Ask yourself

What if a message is **late / out of order / duplicated / for a request that doesn't exist**? If a step raises, does it kill the whole worker?

Rule

Workers are long-running loops and **must not exit because of one bad message**. Every "shouldn't happen but might" case needs an explicit handling path, ideally with a counter.

P02 answer

Case	Handling	Why
The output's request_id is not in `request_states`	Drop it, `dropped_outputs += 1`	Raising would kill the dispatcher and **every** request would hang
Output arrives for a request after its terminal message was sent and its state cleaned up	Same as above: treat as unknown id	Guarantees "the terminal message is the last one"
Outputs of several requests interleave in `_ready_q`	No special handling needed	State is stored per rid and the dispatcher processes serially
Multiple chunks from the same stage	Append to `stage_data[sid]` in arrival order	Each stage has exactly one reader, so per-stage order is preserved naturally

P02 worksheet (filled in)

When you get a new problem, spend 5 minutes filling in this table starting from T before writing code. Here is the answer for P02:

	Question	P02
T	How many processes / threads / event loops? Where are the boundaries?	Fixed by the problem: orchestrator = 1 event loop in the API process; stage = separate process, wrapped by `StageClient` into an async interface; no threads needed
S	What shared state is there? Who writes each piece?	`request_states` (mainly the dispatcher; the handler only registers), `dropped_outputs` (dispatcher)
B	What are the blocking points? Which are independent?	N × `get_output()` (independent, unpredictable timing), `_req_q.get()`, `_ready_q.get()`, `_out_q.get()`
W	Which workers are needed? Wait on / do / must not?	reader × N (move only), request_handler (register + submit to S0), dispatcher (sole decision-maker)
C	What channels are there? What are the message formats?	`_req_q: ("add", rid, prompt)`; `_ready_q: (sid, list[StageOutput])` (fan-in); `_out_q: OrchOutput`
L	How does it start? How does it stop?	`start`: N+2 create_task calls, handles kept; `shutdown`: cancel + gather(return_exceptions=True)

© 2026 kzhou003

I	What does the public API do, and nothing more?	`add_request` only puts; `next_output` only gets (with timeout)
F	What are the invariants? Is state self-consistent before each await?	Exactly one terminal message and it is the last; terminal condition is `final ⊆ finished`; register before submit; delivering and forwarding are independent; clean up after the terminal message
E	Late / unknown / duplicate / exceptions?	Unknown id: drop and count, never raise; messages arriving after cleanup are treated as unknown

Three rules that apply throughout

> **1. await means yielding control.** Whenever you see `await`, ask: "If another worker runs right now, is the state it sees correct?" Finish state changes **before** the await.

> **2. Cross threads only through dedicated bridges.** To enter asyncio from a plain thread, use `loop.call_soon_threadsafe(...)` or `asyncio.run_coroutine_threadsafe(...)`; never call `asyncio.Queue` or `Event` methods directly from a thread (that is exactly what P01 drills).

> **3. If "queue + single writer" works, don't use locks.** Locks deadlock easily and are easy to forget; queues turn concurrency problems into **serial processing**, making "who changed what, and when" obvious at a glance.

Common patterns cheat sheet

Pattern	When to use	In this problem set
fan-in (many readers → one queue → single consumer)	Several independent blocking sources whose results must be handled in one place	P02 / P03 readers + dispatcher
single writer	Shared state needs complex read-modify-write	P02 dispatcher, P10 EngineCore loop
thread → event loop bridge (batched wakeups)	Producer in a thread, consumer in asyncio	P01 OutputSink
request correlation (key → future)	RPC over a shared channel with results returning out of order	P05 RpcRouter
background send/recv thread + sequence numbers	Cross-process data must be delivered in order and cancellably	P13 ChunkAdapter

Blank worksheet (print it and fill one in per problem)

Problem: _____ Date: _____

	Question	My answer
T	How many processes / threads / event loops? Which boundaries are given? How does each boundary communicate?	
S	What shared state is there? Who writes each piece? Can each have a single writer?	

© 2026 kzhou003

B	What are the blocking points? Which are independent? Thread blocking or await blocking?	
W	Which workers (tasks / threads) are needed? Wait on / do / must not?	
C	What channels are there? Who produces, who consumes? What are the message formats?	
L	How does it start? How does it stop? Does anything hang when stopping?	
I	What does the public API do, and nothing more? Whose context (thread / coroutine) does it run in?	
F	What are the invariants? Check each await: is state self-consistent when control switches away?	
E	Late / out-of-order / duplicate / unknown / exceptions / cancellation?	

Self-check after writing the code:

- Does any asyncio task call a blocking synchronous function? Does any thread touch asyncio objects directly?
- Is state self-consistent before every `await`?
- Does every piece of mutable state have exactly one writer? If there are several, is it protected by a lock or another mechanism?
- Is any worker waiting on two or more blocking sources at once?
- Could any single bad message make a worker exit?
- After `shutdown`, is any task or thread still alive?

© 2026 kzhou003

Track M · Qwen-Omni model implementation

Master the model itself first: every drill is checked against the Qwen3-Omni implementation in HF transformers 4.57 (tiny random weights, seconds on CPU). After this track you will know exactly what each layer of thinker / talker / MTP / code2wav computes.

© 2026 kzhou003

#1 · M01 RMSNorm and multimodal RoPE (interleaved / chunked)

45 min | medium

□ Done Date _____

> Every thinker layer starts with RMSNorm, and every attention applies RoPE to q/k. Qwen-Omni's RoPE is three-dimensional (t,h,w), and Qwen3 and Qwen2.5 use different frequency layouts. Get one index wrong and the model won't error; it just quietly gets dumber.

Where it sits in the system

Every decoder layer of the thinker (stage 0): `h = h + Attn(RMSNorm(h))`, `h = h + MoE(RMSNorm(h))`; inside Attention, rotary position embeddings are applied to q and k. The talker (stage 1) uses the same components. You will reuse these functions in M02 and M08.

RMSNorm numerics

`y = w · x / sqrt(mean(x²) + eps)`. The HF/vLLM detail: normalize x in fp32, **cast back to the input dtype, then multiply by weight**. In bf16 a different order is not bitwise identical, and the parity test will catch it.

RoPE recap

With head_dim = D, split the vector into two halves [a, b]; each frequency j (0 ≤ j < D/2) has angular rate `inv_freq[j] = θ^(−2j/D)`. The angle at position p is `p·inv_freq[j]`; `cos/sin` have shape [T, D] (each half duplicated), and the rotation is `x·cos + rotate_half(x)·sin`, where `rotate_half([a,b]) = [−b, a]`.

MRoPE: 3D positions

Each token has three positions (t, h, w): text tokens have all three equal; image patches have their own grid coordinates; audio is linear in time. **The D/2 frequencies are split among the three axes**; `mrope_section` (e.g. `[24, 20, 20]` for the real Qwen3-Omni, summing to 64 = 128/2) gives how many frequencies each axis gets. There are two ways to split:

Model	Layout (example: D/2 = 8, section = [4,2,2])
Qwen2-VL / Qwen2.5-Omni: **chunked**	`T T T T H H W W`
Qwen3-VL / Qwen3-Omni: **interleaved**	`T H W T H W T T` (index j with j%3==1 and j<3·section[1] goes to H; j%3==2 and j<3·section[2] goes to W; the rest go to T)

Motivation for interleaved: every axis gets both high- and low-frequency components, whereas chunked leaves the W axis with only the lowest frequencies.

Functions to implement

- `rms_norm`, `rope_inv_freq`, `mrope_cos_sin(inv_freq, positions[3,T], section, interleaved)`, `rotate_half`, `apply_rotary(x[T,heads,D], cos, sin)`
- Layout convention: in this drill set q/k are `[T, heads, D]` (vLLM style, tokens first); HF uses `[B, heads, T, D]`; the tests convert for you.

© 2026 kzhou003

Edge cases

- For pure text the three axis positions are equal, and both layouts must reduce to ordinary 1D RoPE (the tests check this).
- With bf16 inputs, cast cos/sin to the input dtype before multiplying.

File: `mine/m01_rmsnorm_mrope.py` **Check:** `pytest tests/test_m01_rmsnorm_mrope.py` **Hints:** A-1 **Tests:** B-1 **Solution:** C-1

Interface skeleton

```
"""M01 · RMSNorm + multimodal RoPE (interleaved for Qwen3-Omni, chunked for Qwen2.5-Omni).

rms_norm(x, weight, eps): normalize in fp32, cast back to x.dtype, THEN multiply by weight.
rope_inv_freq(head_dim, theta): 1 / theta ** (arange(0, head_dim, 2) / head_dim)  (fp32).
mrope_cos_sin(inv_freq, positions[3, T], mrope_section, interleaved) -> cos, sin [T, head_dim]
  freqs[a] = positions[a, :, None] * inv_freq  for axis a in (t, h, w), each [T, head_dim // 2].
  Pick, for every frequency index j, which axis' position it rotates with:
    interleaved=True  (Qwen3):   j % 3 == 1 and j < 3*section[1] -> h;
                                 j % 3 == 2 and j < 3*section[2] -> w; otherwise t.
    interleaved=False (Qwen2.5): contiguous chunks: first section[0] indices t, next section[1]
                                 h, last section[2] w.
  Then emb = cat([f, f], -1) and return emb.cos(), emb.sin().
apply_rotary(x[T, heads, head_dim], cos, sin) = x * cos + rotate_half(x) * sin,
  rotate_half([a, b]) = [-b, a] on the two halves of the last dim.
"""
import torch


def rms_norm(x: torch.Tensor, weight: torch.Tensor, eps: float) -> torch.Tensor:
    raise NotImplementedError


def rope_inv_freq(head_dim: int, theta: float) -> torch.Tensor:
    raise NotImplementedError


def mrope_cos_sin(
    inv_freq: torch.Tensor,
    positions: torch.Tensor,
    mrope_section: list[int],
    interleaved: bool,
) -> tuple[torch.Tensor, torch.Tensor]:
    raise NotImplementedError


def rotate_half(x: torch.Tensor) -> torch.Tensor:
    raise NotImplementedError


def apply_rotary(x: torch.Tensor, cos: torch.Tensor, sin: torch.Tensor) -> torch.Tensor:
    raise NotImplementedError
```

What the tests check

- ✓ RMSNorm matches HF in both fp32 and bf16
- ✓ inv_freq matches HF
- ✓ Interleaved MRoPE cos/sin match HF Qwen3-Omni (including image grid positions)
- ✓ With text-only positions both layouts reduce to 1D RoPE
- ✓ q/k after apply_rotary match HF
- ✓ Chunked layout matches HF Qwen2.5-Omni's apply_multimodal_rotary_pos_emb

© 2026 kzhou003

Follow-up questions

- Why does vllm-omni force `custom_ops: ["+rotary_embedding"]` for the thinker on CUDA? (The compiled fusion of MRoPE with QK-norm changed the audio output.)
- In decode there is only one new token; how do you compute cos/sin incrementally and efficiently?

Intuition & analogies

Analogy · Volume normalization + equalizer

Every song in a playlist has a different volume. RMSNorm first brings each song (each token's hidden vector) to the same **overall loudness** (divide by the RMS), then an equalizer (`weight` , one knob per dimension) adjusts each band.

"Cast back to x.dtype, then multiply by weight" is like: **export to 16-bit** first, then run the equalizer; versus run the equalizer in the 32-bit project and export afterwards. Your ears can't tell the difference, but a byte-for-byte comparison of the two files says they're **not the same** - and the parity test is that byte-for-byte comparison.

Analogy · A row of clock hands spinning at different speeds

Split a head_dim = D vector into D/2 pairs; each pair is one **clock hand**. Hand j spins at `inv_freq[j] = θ^(−2j/D)` : j = 0 is the fastest (the second hand), and larger j is slower (hour hand, calendar). For a token at position p, hand j sits at angle `p · inv_freq[j]` .

When two tokens are dotted together, only the **angle difference** survives - so attention naturally sees **relative position**. Fast hands tell "next door vs two doors down"; slow hands tell "10 apart vs 1000 apart". Rotation doesn't change length, so RoPE doesn't disturb the norms QK-norm just normalized.

Analogy · Dealing cards: one pile each vs round-robin

Sort 64 cards from "fast → slow" and deal them to three players T, H, W.

chunked (Qwen2-VL / Qwen2.5-Omni): the first 24 cards go to T, the next 20 to H, the last 20 to W. W ends up with only the **slowest** hands - adjacent patch columns have almost the same angle, so column resolution is poor.

interleaved (Qwen3-VL / Qwen3-Omni): deal round-robin T H W T H W ...; once H's and W's shares are used up, all remaining cards go to T. Every axis gets **both fast and slow hands**.

Framework view · Data flow / shape trace

Notation: T tokens, D = head_dim, half = D/2, heads = number of q or k heads.

#	tensor	shape / dtype	What happens
1	`x` → `rms_norm`	`[..., H]` any dtype	Cast to fp32, take `mean(x²)` over the last dim (keepdim), multiply by `rsqrt(· + eps)` , **cast back to x.dtype, then multiply by weight**
2	`inv_freq`	`[half]` fp32	`θ^(−2j/D)` , j = 0 … half-1
3	`positions`	`[3, T]` int	The three rows are t, h, w; for text tokens all three rows are identical
4	`freqs`	`[3, T, half]` fp32	One broadcast: the angle for every axis, every token, every hand
5	`f`	`[T, half]`	For each column j, **pick one** of the three axes and copy it over (column selection only, no new math); interleaved / chunked differ only here
6	`emb`	`[T, D]`	`cat([f, f], −1)` : column j and column j+half have the same angle

© 2026 kzhou003

7	`cos, sin`	`[T, D]` fp32	`emb.cos()` , `emb.sin()`
8	`apply_rotary(x)`	`x[T, heads, D]`	Insert a head dim into cos/sin to get `[T, 1, D]` , cast to x.dtype; `x·cos + rotate_half(x)·sin`

Pitfalls

- In bf16, RMSNorm multiplies by weight in fp32 and then casts back: numerically "close enough", but not bitwise identical.
- `mean` without `keepdim=True` , or taking the mean over all dims.
- For interleaved, modifying `freqs[0]` directly (no clone), or writing the slice end as `section` instead of `3 · section` .
- For chunked, taking columns `0..s_h` of `freqs[1]` for the H segment instead of the same column range `s_t..s_t+s_h` .
- Duplicating each frequency into adjacent pairs (`repeat_interleave` , GPT-J style), which doesn't match the "first half / second half pairing" of `rotate_half` .
- Writing `rotate_half` as `[b, −a]` : the rotation goes the wrong way.
- Multiplying cos directly with `[T, heads, D]` : no error when T == heads, but the T dim gets wrongly aligned with the heads dim.

Interactive (Chinese): `nano-vllm-omni-drills/hints/m01_overview.html`

Source: `deploy/qwen3_omni_moe.yaml:84-91` `model_executor/models/qwen3_omni/qwen3_omni_moe_thinker.py:1694-1785` HF: Qwen3OmniMoeThinkerTextRotaryEmbedding.apply_interleaved_mrope HF: Qwen3OmniMoeThinkerTextRMSNorm

Notes / pitfalls I hit:

© 2026 kzhou003

#2 · M02 Thinker attention: QK-norm + GQA + KV cache

⏱ 60 min | medium | □ Done | Date _____

> Qwen3 attention has one extra step over Llama: RMSNorm on each head's q and k before RoPE. You also need one implementation that supports full prefill, chunked prefill and token-by-token decode, with all three matching row for row.

Where it sits in the system

Every layer of the thinker and talker has one of these attentions. In serving it almost never runs "whole": long prompts are split by chunked prefill, decode has only 1 new token per step, and historical K/V come from the cache (paged KV in vLLM, P06/P09; here a simple list concat).

Computation (one layer, T new tokens)

```
q = x @ Wq^T -> [T, H, D]      k = x @ Wk^T -> [T, KVH, D]      v = x @ Wv^T -> [T, KVH, D]
q = RMSNorm_head(q; q_norm)     k = RMSNorm_head(k; k_norm)       # over the last dim D, each head
independently
q, k = RoPE(q, k; cos, sin)                                       # cos/sin cover only these T new tokens
K, V = cat(cache.K, k), cat(cache.V, v)                           # S = past + T
score[h, t, s] = q[t, h] · K[s, h // (H/KVH)] / sqrt(D),  with s ≤ past + t (causal)
out = softmax_fp32(score) @ V -> [T, H*D] @ Wo^T
```

Why the three call patterns must agree

The scheduler (P07) decides how many tokens a request computes this step: the whole thing, part of the prompt, or 1 decode token. Model code must not care; it may rely only on the convention "absolute position of new token i = len(cache) + i". The causal mask must be computed on **absolute positions**, not on in-chunk indices.

GQA

There are fewer KV heads than Q heads (here 4 vs 2): each KV head is shared by `H/KVH` consecutive Q heads (head h uses kv head `h // (H/KVH)`, i.e. `repeat_interleave`, not `repeat`).

Edge cases

- cache None means plain prefill.
- T=1 decode and T>1 chunks use the same code.

File: `mine/m02_thinker_attention.py` **Check:** `pytest tests/test_m02_thinker_attention.py` **Hints:** A-2 **Tests:** B-2 **Solution:** C-2

Interface skeleton

```
"""M02 · Qwen3 thinker attention: QK-norm + MRoPE + GQA + causal, with a KV cache.

attention_forward(w, x[T, hidden], cos[T, D], sin[T, D], cache=None) -> [T, hidden]
  1. q = x @ q_proj.T -> [T, H, D];  k, v = ... -> [T, KVH, D]
  2. RMSNorm EACH HEAD of q and k over head_dim (q_norm / k_norm weights) -- before RoPE
  3. RoPE on q and k with the given cos/sin (rows belong to the NEW tokens)
  4. if cache is given: the new tokens sit at absolute positions len(cache) .. len(cache)+T-1;
     append k, v to the cache and attend over everything cached so far
  5. causal: query at absolute position p sees keys 0..p;  GQA: query head h uses kv head
     h // (H // KVH);  scale = D ** -0.5;  softmax in fp32
  6. out [T, H*D] @ o_proj.T
Prefill-all-at-once, chunked prefill and token-by-token decode must give identical rows.
"""
from dataclasses import dataclass, field
```


© 2026 kzhou003

```python
import torch


# ---------------------------------------------------------------- given types
@dataclass
class AttnWeights:
    q_proj: torch.Tensor  # [num_heads * head_dim, hidden]
    k_proj: torch.Tensor  # [num_kv_heads * head_dim, hidden]
    v_proj: torch.Tensor  # [num_kv_heads * head_dim, hidden]
    o_proj: torch.Tensor  # [hidden, num_heads * head_dim]
    q_norm: torch.Tensor  # [head_dim]  RMSNorm weight applied per head
    k_norm: torch.Tensor  # [head_dim]
    num_heads: int
    num_kv_heads: int
    head_dim: int
    eps: float = 1e-6


@dataclass
class KVCache:
    k: list[torch.Tensor] = field(default_factory=list)  # chunks of [t, num_kv_heads, head_dim]
    v: list[torch.Tensor] = field(default_factory=list)

    def __len__(self) -> int:
        return sum(x.shape[0] for x in self.k)


def from_hf(att) -> AttnWeights:
    """Pull the weights out of a HF Qwen3OmniMoeThinkerTextAttention."""
    return AttnWeights(
        att.q_proj.weight.detach(), att.k_proj.weight.detach(), att.v_proj.weight.detach(),
        att.o_proj.weight.detach(), att.q_norm.weight.detach(), att.k_norm.weight.detach(),
        att.config.num_attention_heads, att.config.num_key_value_heads, att.head_dim, att.config.rms_norm_eps,
    )


# ---------------------------------------------------------------- implement below
def attention_forward(
    w: AttnWeights,
    x: torch.Tensor,
    cos: torch.Tensor,
    sin: torch.Tensor,
    cache: KVCache | None = None,
) -> torch.Tensor:
    raise NotImplementedError
```

What the tests check

- ✓ Full prefill matches HF attention
- ✓ Prefill 5 tokens then decode 7 times one by one: rows equal the full pass
- ✓ Chunked prefill (4 + 7) equals the full pass, cache length correct

Follow-up questions

- How does the interface change if you replace list concat with P06 paged KV + P09 paged attention?
- Can QK-norm and RoPE be fused into one kernel? Why did vllm-omni once have problems from compiled fusion?

© 2026 kzhou003

Intuition & analogies

Analogy · A latecomer reads the meeting minutes

Everything said in the meeting goes into the **minutes** (the KV cache). A latecomer (a new token) doesn't need everyone to rerun the meeting; reading the minutes tells them what was said.

The minutes record each remark's timestamp **at the time** - matching "the k in the cache was already rotated with its absolute position back then", never to be recomputed.

The newcomer may only see remarks made **before they arrived (including their own)**, not what later people haven't said yet (the causal mask). The key is that they judge "before" by **the meeting's master clock**, not "I'm the 0th to arrive today" - if you use in-chunk indices, the decode token thinks it's at position 0 and can only see the first remark.

Analogy · GQA: deskmates sharing a reference book

Each Q head is a student, each KV head a reference book. There are fewer books than students (here H = 4, KVH = 2): H/KVH students in **adjacent seats** share one book - students 0, 1 use book 0, students 2, 3 use book 1. That's `h // (H // KVH)`, equivalent to `repeat_interleave` on the books. Use `repeat` by mistake and it becomes "every other seat shares a book" (0, 2 use book 0), so every head gets the wrong book - and a test with H == KVH will never notice.

Analogy · QK-norm: weigh-in before the match

Qwen3 has one step more than Llama: before RoPE, apply RMSNorm (over head_dim) to each head's q and k separately. Like a weigh-in before the match: every head's q and k are brought to the same "weight class", so dot products don't blow up because some heads have unusually large activations. The order is **proj → QK-norm → RoPE**: RMS is invariant to rotation, but the per-dimension `q_norm` weight is not, so swapping the order changes the numbers. v is not normed.

Framework view · Data flow / shape trace

Notation: T new tokens, past = `len(cache)` before the call, S = past + T, H / KVH / D are head counts and head_dim.

#	tensor	shape	What happens
1	`x`	`[T, hidden]`	New tokens only
2	`q` / `k, v`	`[T, H, D]` / `[T, KVH, D]`	view after `x @ proj.T`; q_proj rows are blocked by head, so each run of D contiguous values in the last dim belongs to one head
3	QK-norm	same	RMSNorm over the last dim D, with all heads sharing `q_norm` (k uses `k_norm`); v untouched
4	RoPE	same	cos/sin `[T, D]` → `[T, 1, D]` broadcast to every head
5	cache	`K, V: [S, KVH, D]`	Read past first, then append this call's k, v, then concatenate all blocks
6	GQA	`[S, H, D]`	Each KV head is copied to H/KVH adjacent Q heads
7	`scores`	`[H, T, S]` fp32	`q·K · D^-0.5`; positions with `k_pos > past + t` set to `-inf`
8	`probs @ V`	`[T, H, D]`	softmax over the key dim (fp32), weighted sum, then cast back to x.dtype
9	output	`[T, H·D] @ o_proj.T → [T, hidden]`	The reshape mirrors the view in step 2

© 2026 kzhou003

Pitfalls

- Reading `len(cache)` after appending: past is overcounted by T, the mask shifts right, and tokens can "peek" at later tokens in the same chunk.
- Using in-chunk indices `arange(T)` for the mask instead of `past + arange(T)`: the decode token can only see key 0.
- GQA with `repeat` / `expand` + a wrong reshape gives a 0,1,0,1 mapping; invisible when H == KVH.
- QK-norm placed after RoPE, or normalizing over all of `H·D` instead of each head's D.
- softmax over `dim=-2` (each column sums to 1); or softmax in bf16.
- Re-applying RoPE to old k in the cache (they were already rotated with the correct absolute positions).

Interactive (Chinese): `nano-vllm-omni-drills/hints/m02_overview.html`

Source: `model_executor/models/qwen3_omni/qwen3_omni_moe_thinker.py:588-596` HF: Qwen3OmniMoeThinkerTextAttention.forward

Notes / pitfalls I hit:

© 2026 kzhou003

#3 · M03 Sparse MoE: routing, per-expert grouped compute, shared expert

60 min | medium

□ Done Date _____

> The "A3B" in Qwen3-Omni-30B-A3B is MoE: each token activates only a few experts. The thinker is pure routed MoE; the talker adds a shared expert with a sigmoid gate.

Where it sits in the system

The FFNs of both thinker and talker are MoE. In serving, MoE efficiency determines throughput: all tokens of one expert should go through a single matmul (that is what vLLM's FusedMoE does), not a per-token loop.

Routing (must match HF)

```
logits = x @ router^T                      # [N, E]
probs  = softmax(logits, dim=-1, fp32)      # softmax over all E experts
w, ids = topk(probs, k)                    # softmax first, then top-k (not softmax after top-k)
if norm_topk_prob: w = w / w.sum(-1)        # Qwen3 renormalizes by default
out[n] = Σ_j w[n,j] · Expert_{ids[n,j]}(x[n]),   Expert(h) = (silu(h Wg^T) ⊙ (h Wu^T)) Wd^T
```

"The serving way": group by expert

1. Sort the N×k (token, slot) pairs by expert id (`argsort(stable=True)`);
2. `bincount` gives the token count per expert; skip zeros;
3. Each expert runs one SwiGLU over its batch of tokens;
4. Multiply by the corresponding weights and scatter back to the original tokens with `index_add_` .

This is exactly a CPU version of "moe_align_block_size + grouped GEMM" in the FusedMoE kernel.

The talker's shared expert

`out += sigmoid(x @ g^T) · SharedMLP(x)` : every token goes through this expert; the gate is a scalar.

Edge cases

- With N=1 (decode) only k experts have work.
- Batch invariance: computing a token alone gives the same result as in a batch.

File: `mine/m03_sparse_moe.py` **Check:** `pytest tests/test_m03_sparse_moe.py` **Hints:** A-3 **Tests:** B-3 **Solution:** C-3

Interface skeleton

```
"""M03 · Sparse MoE block (thinker) and MoE + gated shared expert (talker).

route(x[N, H], w) -> (weights [N, k] in x.dtype, expert ids [N, k])
  softmax(x @ router.T) over ALL experts in fp32 -> top-k -> renormalize to sum 1 if
  norm_topk_prob.
moe_forward(x[N, H], w) -> [N, H]
  out[n] = sum_j weight[n, j] * Expert_{id[n, j]}(x[n]),  Expert(h) = (silu(h Wg^T) * (h Wu^T)) Wd^T
  Do it the serving way: group (token, slot) pairs by expert, run each expert ONCE on all of
  its tokens, scatter-add the weighted results; skip experts that received no token.
  If w.shared is set (talker): out += sigmoid(x @ shared.gate.T) * SharedExpert(x).
"""
from dataclasses import dataclass

import torch
import torch.nn.functional as F
```


© 2026 kzhou003

```
# ---------------------------------------------------------------- given types
@dataclass
class SharedExpert:
    gate_proj: torch.Tensor   # [I_s, H]
    up_proj: torch.Tensor     # [I_s, H]
    down_proj: torch.Tensor   # [H, I_s]
    gate: torch.Tensor        # [1, H]   sigmoid gate of the shared expert


@dataclass
class MoeWeights:
    router: torch.Tensor      # [E, H]
    w_gate: torch.Tensor      # [E, I, H]   stacked expert gate_proj
    w_up: torch.Tensor        # [E, I, H]
    w_down: torch.Tensor      # [E, H, I]
    top_k: int
    norm_topk_prob: bool
    shared: SharedExpert | None = None


def from_hf(block) -> MoeWeights:
    """HF Qwen3OmniMoe{Thinker,Talker}TextSparseMoeBlock -> stacked tensors."""
    ex = block.experts
    shared = None
    if hasattr(block, "shared_expert"):
        s = block.shared_expert
        shared = SharedExpert(s.gate_proj.weight.detach(), s.up_proj.weight.detach(),
                              s.down_proj.weight.detach(), block.shared_expert_gate.weight.detach())
    return MoeWeights(
        block.gate.weight.detach(),
        torch.stack([e.gate_proj.weight.detach() for e in ex]),
        torch.stack([e.up_proj.weight.detach() for e in ex]),
        torch.stack([e.down_proj.weight.detach() for e in ex]),
        block.top_k, block.norm_topk_prob, shared,
    )


# ---------------------------------------------------------------- implement below
def route(x: torch.Tensor, w: MoeWeights) -> tuple[torch.Tensor, torch.Tensor]:
    raise NotImplementedError


def moe_forward(x: torch.Tensor, w: MoeWeights) -> torch.Tensor:
    raise NotImplementedError
```

What the tests check

- ✓ Routing (softmax→top-k→renorm) exactly matches the reference
- ✓ Thinker MoE matches HF
- ✓ Talker MoE (with shared expert) matches HF
- ✓ norm_topk_prob=False variant
- ✓ Single-token and batch results agree

Follow-up questions

- With Expert Parallel, tokens go all-to-all across GPUs; how are the routing results used?
- Why does vllm-omni's deploy config set `moe_backend: triton` for Qwen3-Omni?

© 2026 kzhou003

Intuition & analogies

> **Analogy · Hospital triage desk**
> The router is the **triage desk**: for each patient (token) it scores "how likely they belong here" for every department (expert), normalizes over **all departments** (softmax), then books the k highest-scoring departments (top-k). `norm_topk_prob` means "redistribute 100% of the say among just the booked departments".
> **The serving way**: each department **gathers all of today's patients and sees them together** (one matmul), instead of the doctor walking ward to ward; a department with no bookings today simply **takes the day off** (skipped).
> Finally each patient receives k reports, **merged with weights** into their own chart - and it must be **accumulated**: the second doctor's report must not overwrite the first's (`index_add_` , not assignment).

> **Analogy · shared expert = the family doctor**
> In the talker, every patient sees a specialist and **also always** sees the family doctor (the shared expert). How much the family doctor's opinion counts is set by a small switch that looks only at the patient: `sigmoid(x @ gate.T)` , one scalar in 0~1 per token.

Framework view · Data flow / shape trace

Notation: N tokens, H = hidden, E experts, I = expert intermediate dim, k = top_k, n_e = number of tokens routed to expert e.

#	tensor	shape	What happens
1	`logits`	`[N, E]`	`x @ router.T`
2	`probs`	`[N, E]` fp32	softmax over all E experts
3	`weights, ids`	`[N, k]`	top-k; optional renorm (fp32), weights cast back to x.dtype
4	flatten	`[N·k]` × 3	For each (token, slot) pair: expert id, weight, token id (aligned one-to-one, row-major)
5	group	`order[N·k]` , `counts[E]`	Stable sort by expert id; pair count per expert (sums to N·k)
6	expert e	`x[toks]: [n_e, H] → [n_e, I] → [n_e, H]`	Gather this batch of tokens and run one SwiGLU: `(silu(h Wg`^{T}`) ⊙ h Wu`^{T}`) Wd`^{T} ; skip if $n_e = 0$
7	scatter	`out[N, H]`	Multiply by the matching weights, then **accumulate** by token id
8	shared (talker)	`[N, 1] · [N, H]`	`sigmoid(x @ gate.T)` times a full SwiGLU, added to out

Pitfalls

- top-k on logits, then softmax: equivalent exactly when `norm_topk_prob=True` , completely different when `False` (weights should sum to < 1).
- token ids built with `arange(N).repeat(k)` (0,1,2,0,1,2) instead of repeating each k times (0,0,1,1,2,2), which doesn't line up with `ids.reshape(-1)` .
- scatter by assignment `out[toks] = y` : the same token gets overwritten by the second expert.
- gate/up projection written as `h @ w_gate[e]` (missing transpose), so shapes `[n_e, H] @ [I, H]` don't match.
- Routing weights left in fp32, so the result dtype gets promoted; or running a matmul for an empty expert.
- Treating the shared-expert gate as a vector, or forgetting to skip when `w.shared is None` (thinker).

© 2026 kzhou003

Interactive (Chinese): `nano-vllm-omni-drills/hints/m03_overview.html`

Source: `model_executor/models/qwen3_omni/qwen3_omni_moe_talker.py:356-393` `deploy/qwen3_omni_moe.yaml:28-31` HF: Qwen3OmniMoeThinkerTextSparseMoeBlock HF: Qwen3OmniMoeTalkerTextSparseMoeBlock

Notes / pitfalls I hit:

© 2026 kzhou003

#4 · M04 Multimodal merge: placeholder counts, embedding merge, DeepStack ⏱ 45 min

medium □ Done Date _____

> A user sends an audio clip and an image. The frontend must know in advance how many placeholder tokens to insert, the thinker must drop the encoder outputs into those slots, and visual features are additionally injected into the first few layers via DeepStack.

Where it sits in the system

The frontend (API process) uses the HF processor to expand `<audio>` and `<image>` into N placeholder tokens. **N must exactly equal the number of tokens the encoder actually outputs**, otherwise the merge on the thinker side fails. In the thinker (stage 0): `inputs_embeds = embed_tokens(ids)`, then the outputs of the audio encoder (AuT) and the ViT are written into the placeholder positions in order.

Audio token count

AuT uses 100 mel frames as a window; each full window goes through 3 stride-2 convolutions to give 13 tokens; the remaining r = n % 100 frames also go through 3 stride-2 convolutions:

```
f = (r - 1) // 2 + 1
tokens_r = ((f - 1) // 2 + 1 - 1) // 2 + 1
total = tokens_r + 13 * (n // 100)        # e.g. n = 100 -> 13; n = 250 -> 33
```

Embedding merge

For each modality, `mask = (input_ids == placeholder_id)`, `out[mask] = mm_embeds[modality]` (in order of appearance). A count mismatch must raise; in production this kind of error usually means the frontend and backend processor versions differ.

DeepStack

Qwen3-Omni's ViT exports one copy of visual features from each of several intermediate layers (`deepstack_visual_indexes`). In the first L LLM layers, **after layer i's output**, the i-th feature is **added** to the hidden state at the visual token positions:

```
for i, layer in enumerate(layers):
    h = layer(h)
    if i < len(deepstack): h[visual_mask] += deepstack[i]
```

Edge cases

- If a modality is absent, mm_embeds may omit that key; but if placeholders exist it must raise.
- Do not modify text_embeds in place (the caller may reuse it).

File: `mine/m04_mm_merge_deepstack.py` **Check:** `pytest tests/test_m04_mm_merge_deepstack.py` **Hints:** A-4 **Tests:** B-4 **Solution:** C-4

Interface skeleton

```
"""M04 · Placeholder bookkeeping, multimodal embedding merge, DeepStack injection.

audio_token_count(num_mel_frames) (int or LongTensor): number of <audio> placeholders the
  Qwen3-Omni audio encoder produces. Mel frames are processed in windows of 100 frames; a
  full window yields 13 tokens; the remainder r = n % 100 passes three stride-2 convolutions:
  f = (r - 1) // 2 + 1;  tokens_r = ((f - 1) // 2 + 1 - 1) // 2 + 1.  total = tokens_r + 13 * (n // 100)
merge_multimodal(input_ids[T], text_embeds[T, H], placeholder_ids, mm_embeds)
```


© 2026 kzhou003

```
  -> (inputs_embeds[T, H], masks{modality: bool[T]})
  Replace the rows at each modality's placeholder positions, in order, with that modality's
  encoder outputs. Count mismatch (or placeholders without embeddings) -> ValueError.
  Never mutate text_embeds.
apply_deepstack(hidden[T, H], visual_mask[T], visual_embeds[n_vis, H]) -> new tensor with
  visual_embeds ADDED at the visual rows.
run_layers(layers, hidden, visual_mask, deepstack_embeds): run the decoder layers in order;
  DeepStack level i is added right AFTER layer i (only for i < len(deepstack_embeds)).
"""
from collections.abc import Callable

import torch


def audio_token_count(num_mel_frames):
    raise NotImplementedError


def merge_multimodal(
    input_ids: torch.Tensor,
    text_embeds: torch.Tensor,
    placeholder_ids: dict[str, int],
    mm_embeds: dict[str, torch.Tensor],
) -> tuple[torch.Tensor, dict[str, torch.Tensor]]:
    raise NotImplementedError


def apply_deepstack(hidden: torch.Tensor, visual_mask: torch.Tensor, visual_embeds: torch.Tensor) -> torch.Tensor:
    raise NotImplementedError


def run_layers(
    layers: list[Callable[[torch.Tensor], torch.Tensor]],
    hidden: torch.Tensor,
    visual_mask: torch.Tensor,
    deepstack_embeds: list[torch.Tensor],
) -> torch.Tensor:
    raise NotImplementedError
```

What the tests check

- ✓ Audio token count matches HF `_get_feat_extract_output_lengths` (including the 100 boundary)
- ✓ Image/audio embeddings land in order, text rows unchanged, input not modified in place
- ✓ Raises on count mismatch or missing embeddings
- ✓ Full-model output after DeepStack matches HF Qwen3OmniMoeThinkerTextModel
- ✓ DeepStack affects only visual rows

Follow-up questions

- With chunked prefill, if an image is split across two chunks, how are encoder outputs and DeepStack features distributed?
- What is the key of vllm-omni's encoder cache?

© 2026 kzhou003

Intuition & analogies

Analogy · Saving seats with bags

The frontend is the friend who arrives early: based on "how many people are coming", they put bags (placeholder tokens) on a row of seats. The encoder is the bus that arrives later with the real passengers (embedding rows). merge is "go left to right along the seats and swap each bag for a person": the k-th bag's seat goes to the k-th passenger.

Images and audio are two tour groups, each ordered by its own bags; they may sit interleaved. If the head count doesn't match the bag count - one too many or one too few - **raise an error on the spot**: in production this usually means the frontend and backend processor versions disagree, and silently truncating just makes the model "look at the wrong picture".

One more courtesy: don't scribble on someone else's seating chart (`text_embeds`); make a photocopy first (`clone`) - the caller may still reuse it.

Analogy · Folding a paper strip: audio token count

AuT processes mel frames in windows of 100, and each window goes through 3 stride-2 convolutions. Each convolution is like **folding** the strip in half: length becomes `(len - 1) // 2 + 1` , i.e. ceil(len / 2), and an odd leftover half-cell still counts as a cell. 100 → 50 → 25 → 13, so every full window contributes 13 tokens; only the tail shorter than 100 actually needs folding three times. With a remainder of 0, `(0 - 1) // 2 + 1 = 0` - this relies on floor division rounding toward negative infinity.

Analogy · Adding ingredients to a stew in stages: DeepStack

Shallow ViT features lean toward "detail", deep ones toward "semantics". DeepStack doesn't add everything when the pot starts; following the recipe, it adds portion i **after the i-th stir** (level i is injected **after** layer i), and only into **the bowls that have vegetables** (the visual-token rows).

Add before stirring (inject before the layer) and the flavor changes - but the text rows got no ingredients at all, so tasting only the text rows will never catch this bug.

Framework view · Data flow / shape trace

#	tensor	shape	What happens
1	`num_mel_frames`	int / `[B]`	Three `(len-1)//2+1` on the remainder window + 13 per full window; uses only `% // + - *` , so it works for both int and tensor
2	`input_ids`	`[T]`	The frontend has already expanded placeholders to the token count
3	`masks[m]`	`bool[T]`	`input_ids == placeholder_ids[m]` ; returned for every modality (all False if absent)
4	validate	—	`mask.sum()` vs `mm_embeds[m].shape[0]` ; placeholders present but embedding missing is also an error
5	`inputs_embeds`	`[T, H]`	`text_embeds.clone()` ; boolean indexing writes rows 0, 1, 2... in ascending position order (cast to the output dtype)
6	layer i	`[T, H]`	`hidden = layer(hidden)`
7	DeepStack i	`hidden[visual_mask]: [n_vis, H]`	For i < K, add `deepstack[i]` on the visual rows and return a new tensor

© 2026 kzhou003

Pitfalls

- Using `if r == 0`, `int()`, `math.floor` in `audio_token_count`: unusable for multi-element tensors; the formula already gives 0 correctly.
- `out = text_embeds` without clone, so the writes mutate the caller's tensor in place.
- Looping over `mm_embeds` as the driver: misses the "placeholders present but no embedding given" error.
- Truncating / zero-padding on a count mismatch instead of raising.
- DeepStack injection placed **before** the layer call (off-by-one), or `zip(layers, deepstack)` skipping the extra layers.
- `hidden[visual_mask] += …` mutating the input in place; or `out += visual_embeds` (n_vis rows vs T rows).

Interactive (Chinese): `nano-vllm-omni-drills/hints/m04_overview.html`

Source: `model_executor/models/qwen3_omni/qwen3_omni_moe_thinker.py:1018`
`model_executor/models/qwen3_omni/qwen3_omni_moe_thinker.py:1137–1168`
`model_executor/models/qwen3_omni/qwen3_omni_moe_thinker.py:605–606` HF: _get_feat_extract_output_lengths HF: Qwen3OmniMoeThinkerTextModel._deepstack_process

Notes / pitfalls I hit:

© 2026 kzhou003

#5 · M05 Talker MTP: predicting 15 residual codebooks within one frame ⏱ 60 min hard

□ Done Date _____

> Each talker step samples only codebook 0; the remaining 15 residual codebooks are filled in autoregressively within the same step by a small transformer (the code predictor). That is the "1 big model + 15 small model" calls behind every talker token.

Where it sits in the system

The talker (stage 1) outputs an RVQ audio codec: G=16 codebooks per frame. After the talker LLM samples CB0, the code predictor starts from `[talker last-layer hidden, CB0 embedding]` and predicts CB1..CB15 one by one; the talker's input embedding for the next step is **the sum of this frame's 16 codec embeddings + the thinker text vector for this step**.

```
seq = [h_talker, E0(cb0)]                # length 2
for j in 0..G-2:                         # predict codebook j+1
    hj   = CP(seq)[:, -1]                # small transformer, full recompute
    code = argmax(lm_head[j](hj))
    seq.append(codec_embedding[j](code))  # note: the embedding table is also indexed by j
next_input = E0(cb0) + Σ_j codec_embedding[j](code_j) + text_step
```

Why vllm-omni chose re-prefill (no KV cache)

The sequence is at most 17 long, so $O(T^2)$ recompute is nearly free; no KV state means fixed shapes that can be wrapped as one CUDA graph (P17), and no per-request cache to manage. When the batch is large the re-prefill FLOPs become significant, so vllm-omni also offers an optional incremental-KV version; both must produce exactly the same codes.

Functions to implement

- `predict_residual_codes` (re-prefill, greedy): checked against HF `code_predictor.generate(do_sample=False)`, which uses a KV cache internally;
- `predict_residual_codes_kv` (incremental version with transformers.DynamicCache);
- `next_talker_input`: the next talker input, checked against HF talker's `prepare_inputs_for_generation`.

The real service samples residual codebooks with top-k 50 / top-p 0.8; this drill uses greedy so it can be compared exactly.

File: `mine/m05_talker_mtp.py` **Check:** `pytest tests/test_m05_talker_mtp.py` **Hints:** A-5 **Tests:** B-5 **Solution:** C-5

Interface skeleton

```
"""M05 · Talker MTP: predict the residual codebooks of one audio frame.

`cp` is a HF Qwen3OmniMoeTalkerCodePredictorModelForConditionalGeneration. You may only use:
  cp.config.num_code_groups (G), cp.model(inputs_embeds=..., [past_key_values=, use_cache=])
  .last_hidden_state, cp.lm_head[j] (j = 0..G-2), cp.model.codec_embedding[j] (j = 0..G-2).
predict_residual_codes(cp, last_talker_hidden[B, H], cb0_embed[B, H]) -> codes [B, G-1]
  Greedy. The sequence starts as [last_talker_hidden, cb0_embed]. For j = 0..G-2: run the
  transformer over the WHOLE sequence (re-prefill, no cache), take the last position, logits
  = lm_head[j], code = argmax, then append codec_embedding[j](code) to the sequence.
predict_residual_codes_kv(...): identical result with an incremental KV cache
  (transformers.DynamicCache): prefill the 2 initial positions, then feed 1 position per step.
next_talker_input(cp, cb0_embed, residual_codes, text_step) -> [B, H]
```


© 2026 kzhou003

```
  cb0_embed + sum_j codec_embedding[j](residual_codes[:, j]) + text_step
"""
import torch


def predict_residual_codes(cp, last_talker_hidden: torch.Tensor, cb0_embed: torch.Tensor) -> torch.Tensor:
    raise NotImplementedError


def predict_residual_codes_kv(cp, last_talker_hidden: torch.Tensor, cb0_embed: torch.Tensor) -> torch.Tensor:
    raise NotImplementedError


def next_talker_input(cp, cb0_embed: torch.Tensor, residual_codes: torch.Tensor, text_step: torch.Tensor) ->
torch.Tensor:
    raise NotImplementedError
```

What the tests check

- ✓ Re-prefill result exactly matches HF generate (KV cache)
- ✓ KV-cache version matches re-prefill version
- ✓ Rows in a batch do not affect each other
- ✓ Next talker input matches HF's summation

Follow-up questions

- Why does seeded per-row sampling prevent full CUDA graph replay? How does vllm-omni handle it?
- Why can MRv2's "eager frames" complete the whole frame in the same step that samples CB0?

Intuition & analogies

Analogy · Measuring height: meters → decimeters → centimeters → millimeters

CB0 is like first saying "about 1.7 m". CB1 doesn't measure again; it answers "how far off?" - "add 3 cm"; CB2 answers "how far off now?" - "subtract 2 mm". Each level only corrects **the error left by the previous level**, so it must know what the earlier levels already reported.

The big talker model = the person who reads "1.7 m" off a tape measure; the code predictor = the assistant with calipers, who takes the tape reading (`last_talker_hidden` + CB0) and refines it level by level. Each level uses **a different scale**: step j has its own `lm_head[j]` (taking the reading) and `codec_embedding[j]` (writing the reading down).

Analogy · Reciting a short poem vs taking notes

re-prefill: before each next line, recite the whole poem written so far from the top. With only a dozen or so lines this costs almost nothing, and the motion is **identical** every time (fixed shapes), so it's easy to "rehearse" - exactly what CUDA graphs like.

KV cache: jot the gist of every line you've read in a notebook, and next time read only the new line. It saves more effort when there are many lines and many people reciting at once (batch), but there's one more notebook to manage.

Both approaches must produce **exactly the same** next line - because under causal attention, the "gist" of each earlier line never changes when new lines are added after it.

Framework view: shape trace and pitfalls

G=5, B=2, H=64, vocab=12 (the same model as the example above).

© 2026 kzhou003

Step	Tensor	Shape	What happens
init	`seq`	`[2, 2, 64]`	Stack two `[B, H]` along a length dim: position 0 = talker hidden state, position 1 = CB0's embedding
j=0	`cp.model(inputs_embeds=seq).last_hidden_state`	`[2, 2, 64]`	The small transformer runs over the whole sequence; take the last position → `[2, 64]`
	`lm_head[?](h)` → argmax	`[2, 12]` → `[2]`	Gives CB1 = `[8, 7]`
	`codec_embedding[?](code)`	`[2, 64]` → `[2, 1, 64]`	Add the length dim back and append to the end of seq
j=1	`seq`	`[2, 3, 64]`	CB2 = `[0, 1]`
j=2	`seq`	`[2, 4, 64]`	CB3 = `[0, 4]`
j=3	`seq`	`[2, 5, 64]`	CB4 = `[6, 5]`; this is the last code
out	the 4 collected `[2]`	`[2, 4]`	Stack as columns: `[[8,0,0,6], [7,1,4,5]]`
next	`next_talker_input`	`[2, 64]`	Sum of the G codec embeddings + `text_step`

The `?` in the table is left for you to fill in (revealed in step 2 of section 4).

Interactive (Chinese): `nano-vllm-omni-drills/hints/m05_overview.html`

Source: `model_executor/models/qwen3_omni/qwen3_omni.py:1044-1069`
`model_executor/models/common/qwen3_code_predictor.py:706-722`
`model_executor/models/common/qwen3_code_predictor.py:1357-1440` `worker/gpu_model_runner.py:243-272` HF: Qwen3OmniMoeTalkerCodePredictorModelForConditionalGeneration HF: Qwen3OmniMoeTalkerForConditionalGeneration.prepare_inputs_for_generation

Notes / pitfalls I hit:

© 2026 kzhou003

#6 · M06 Code2Wav streaming causal conv and transposed conv ⏱ 60 min hard

□ Done Date _____

> Code2Wav upsamples codec to a waveform; its core is causal Conv1d and causal ConvTranspose1d. Streaming decode requires: feed chunks of any size, the concatenated output matches one-shot processing sample for sample, and every sample is emitted as soon as it is "final".

Where it sits in the system

Stage 2 (Code2Wav) turns `[B, 16, T]` codes into a `[B, 1, T·upsample]` waveform: embedding average → sliding-window transformer → **causal transposed-conv upsampling** → ConvNeXt → multi-level (causal conv + SnakeBeta) decoder. vllm-omni by default uses the stateless "recompute left context then slice" scheme (P14); this drill practices the other approach, **stateful streaming convolution**, which is key to understanding the trade-off between the two.

Causal Conv1d (stride 1)

The offline module left-pads the whole signal with `(K−1)·d` zeros (d is dilation), so output length = input length and output t depends only on inputs ≤ t. The streaming version only needs to keep the last `(K−1)·d` inputs as state, prepending it to each new chunk before convolving.

Causal ConvTranspose1d (stride s, kernel K)

```
Input frame i writes to output positions [i·s, i·s + K)    (overlap-add)
Offline: y = conv_transpose(x), length (L−1)·s + K, then trim K−s samples from the start and K−s from the end
Streaming: after receiving n frames, samples at positions < n·s will not be modified by later frames → can be
emitted
      (minus the K−s trimmed globally at the start); on flush emit the rest and trim K−s from the end
```

Pitfall: **add the bias only once, at emit time**. If every chunk's convolution includes bias, overlap regions get it twice.

"Emit as early as possible", as tested

After feeding frame n, the cumulative output must be exactly `max(0, n·s − (K−s))` samples. More means you emitted samples that are not final; fewer means you added unnecessary latency (directly affecting time-to-first-packet audio latency, TTFP).

File: `mine/m06_streaming_causal_conv.py` **Check:** `pytest tests/test_m06_streaming_causal_conv.py`
Hints: A-6 **Tests:** B-6 **Solution:** C-6

Interface skeleton

```
"""M06 · Streaming versions of Code2Wav's causal Conv1d and causal ConvTranspose1d.

StreamingCausalConv1d (stride 1, dilation d, kernel K)
  The offline module left-pads the whole signal with (K−1)*d zeros. step(x[B, C, t]) must
  return exactly t outputs and, concatenated over any chunking, equal the offline output.
  Keep the last (K−1)*d inputs as state.
StreamingCausalTransConv1d (stride s, kernel K, weight [C_in, C_out, K])
  Input frame i writes output positions [i*s, i*s + K) (overlap-add). The offline module trims
  K − s samples at the START and at the END. step(x) returns every sample that is already
  final (after n frames: positions < n*s, minus the global left trim); flush() returns the rest
  minus the right trim. Add the bias once per output position, not once per contribution.
from_hf(m) builds each class from the HF module (weights .conv.weight / .conv.bias).
"""
import torch
```


© 2026 kzhou003

```
class StreamingCausalConv1d:
    def __init__(self, weight: torch.Tensor, bias: torch.Tensor | None, dilation: int = 1) -> None:
        raise NotImplementedError

    @classmethod
    def from_hf(cls, m) -> "StreamingCausalConv1d":  # HF Qwen3OmniMoeCausalConvNet (m.conv, m.dilation, m.stride)
        raise NotImplementedError

    def step(self, x: torch.Tensor) -> torch.Tensor:
        raise NotImplementedError


class StreamingCausalTransConv1d:
    def __init__(self, weight: torch.Tensor, bias: torch.Tensor | None, stride: int) -> None:
        raise NotImplementedError

    @classmethod
    def from_hf(cls, m) -> "StreamingCausalTransConv1d":  # HF Qwen3OmniMoeCausalTransConvNet (m.conv)
        raise NotImplementedError

    def step(self, x: torch.Tensor) -> torch.Tensor:
        raise NotImplementedError

    def flush(self) -> torch.Tensor:
        raise NotImplementedError
```

What the tests check

- ✓ Causal conv: any chunking matches offline, output count per chunk equals input count (including dilation, K=1)
- ✓ Transposed conv: any chunking + flush matches offline (various K/s)
- ✓ Transposed conv: cumulative output after each single frame is exactly the theoretical earliest value

Follow-up questions

- Stateful streaming vs stateless "left-context recompute": what are the effects on batching, abort and CUDA graphs?
- How do you make Code2Wav's sliding-window attention streaming?

Intuition & analogies

> **Analogy · Causal conv = a copyist with a "sticky note"**
>
> For every character the copyist writes, they look back at the previous `(K−1)·d` characters (e.g. with K=3, d=2: "2 back" and "4 back"). Pages arrive one at a time, and the previous page is taken away when you turn it - so before turning, they **copy the last `(K−1)·d` characters onto a sticky note** and stick it at the front of the next page. The note before the first page is blank (all 0), which is exactly the offline version's "left-pad with 0".
>
> Note: if the new page is short (shorter than the note), the new note must be copied from the end of "**old note + new page**" joined together, not from the new page alone.

© 2026 kzhou003

Analogy · Transposed conv = a K-wide stamp every s cells

For each incoming frame, press a `K`-cell-wide stamp onto a long paper strip starting at cell `i·s` (ink darkness = frame value × stamp pattern). Because `K > s`, adjacent stamps **overlap** on `K−s` cells.

When can a cell be cut off and handed downstream? When **no future stamp will ever touch it**: after n frames, the next stamp starts no earlier than `n·s`, so cells `< n·s` are final.

The paper's background color (bias) may be painted only once - if each stamp brought its own background, overlapping regions would be twice as dark. The offline version also trims `K−s` cells off each end of the strip: the cut at the start happens only once (global position), and the cut at the end isn't known until you wrap up (flush).

Framework view: shape trace and pitfalls

Stage	Tensor	Shape	What happens
Conv · before first chunk	`state`	`[B, C_in, (K−1)·d]`	All 0; B / dtype / device are only known on the first step, so it's created lazily
Conv · step	`buf = cat([state, x], −1)`	`[B, C_in, (K−1)·d + t]`	"Note + new page"
	`F.conv1d(buf, w, b, dilation=d)`	`[B, C_out, t]`	No padding: length shrinks by (K-1)·d, leaving exactly t
	new `state`	`[B, C_in, (K−1)·d]`	Taken from the **end** of buf
TransConv · step	`F.conv_transpose1d(x, w, None, stride=s)`	`[B, C_out, (t−1)·s + K]`	Overlap within the chunk is already done; covers global `[frames·s, frames·s + len)`
	`acc` (starting at global position `base`)	`[B, C_out, ≤ K−s]` (after each step)	Overlap across chunks: add the new result into the matching positions of acc
	output	`[B, C_out, ·]`	Global `[base, frames·s)`, add bias once, then drop the part falling in `[0, K−s)`
TransConv · flush	output	`[B, C_out, ·]`	Finalize all remaining pending, drop the trailing K-s (usually that trims all of it)
Cumulative	output after feeding n frames	`max(0, n·s − (K−s))`	Example B: 0, 3, 6, 9, 12, 15 … (with K = 2s it's (n-1)·s)

Interactive (Chinese): `nano-vllm-omni-drills/hints/m06_overview.html`

Source: `model_executor/models/qwen3_omni/qwen3_omni_code2wav.py:124–218`

`model_executor/models/qwen3_omni/qwen3_omni_code2wav.py:447–508` HF: Qwen3OmniMoeCausalConvNet HF: Qwen3OmniMoeCausalTransConvNet

Notes / pitfalls I hit:

© 2026 kzhou003

#7 · M07 Per-stage weight loading and packing

45 min | medium | □ Done | Date _____

> One Qwen3-Omni checkpoint contains weights for thinker, talker and code2wav. Each stage process should open only its own shards, take only its own tensors, and pack q/k/v, gate/up and the experts into kernel-friendly layouts.

Where it sits in the system

The three stage processes each load the same checkpoint directory. vllm-omni's approach: one wrapper class builds only one submodule according to `model_stage`, and `load_weights` routes by prefix `thinker./talker./code2wav.` (see `qwen3_omni.py:1725-1783`). In this drill you go one step further: open only the relevant shards based on the `weight_map` in `model.safetensors.index.json`.

Packed layout (vLLM style)

```
qkv  = cat([q_proj, k_proj, v_proj], dim=0)     # [(H + 2·KVH)·D, hidden] one GEMM yields all three
w13  = stack_e cat([gate_e, up_e], dim=0)       # [E, 2I, hidden]   FusedMoE's w1/w3
w2   = stack_e down_e                           # [E, hidden, I]
router = mlp.gate.weight                        # [E, hidden]
```

Key format (same for the tiny model and the real checkpoint): `thinker.model.layers.{i}.self_attn.q_proj.weight`, `thinker.model.layers.{i}.mlp.experts.{e}.gate_proj.weight`, `thinker.model.layers.{i}.mlp.gate.weight`, etc.

Completeness checks

"Failing to load one tensor" does not error in PyTorch: that parameter stays randomly initialized and the model output quietly turns into noise. So missing keys, extra keys, non-contiguous expert numbering, or an expert missing gate/up/down must all raise `LoadError`.

File: `mine/m07_stage_weight_loading.py` **Check:** `pytest tests/test_m07_stage_weight_loading.py`

Hints: A-7 **Tests:** B-7 **Solution:** C-7

Interface skeleton

```
"""M07 · Stage-aware checkpoint loading and weight packing.

A Qwen3-Omni checkpoint holds every stage: thinker.*, talker.*, code2wav.*. Each stage process
must open only its shards and load only its tensors, then pack them for fast kernels.
shards_for_stage(weight_map, stage): sorted shard files containing at least one key of the stage.
select_stage(state, stage, strip): keep keys starting with `strip` (e.g. "thinker.model.") and
  remove that prefix.
pack_layer(sd, i) -> PackedLayer for "layers.{i}.*":
  qkv = cat(q_proj, k_proj, v_proj) along rows; w13[e] = cat(gate_proj_e, up_proj_e) along
  rows; w2[e] = down_proj_e; router = mlp.gate.weight. Expert ids must be exactly 0..E-1 and
  complete; any missing or unexpected key in the layer -> LoadError.
load_thinker_text(state, num_layers) -> ({"embed_tokens", "norm"}, [PackedLayer]); anything
  else under thinker.model. besides layers/embed_tokens/norm -> LoadError.
"""
import re
from dataclasses import dataclass

import torch

STAGE_PREFIX = {"thinker": "thinker.", "talker": "talker.", "code2wav": "code2wav."}


class LoadError(Exception):
    pass
```


© 2026 kzhou003

```
@dataclass
class PackedLayer:
    input_layernorm: torch.Tensor
    qkv: torch.Tensor               # [(H + 2*KVH) * D, hidden]  rows: q | k | v
    q_norm: torch.Tensor
    k_norm: torch.Tensor
    o_proj: torch.Tensor
    post_attention_layernorm: torch.Tensor
    router: torch.Tensor            # [E, hidden]
    w13: torch.Tensor               # [E, 2*I, hidden]  rows: gate | up
    w2: torch.Tensor                # [E, hidden, I]


def shards_for_stage(weight_map: dict[str, str], stage: str) -> list[str]:
    raise NotImplementedError


def select_stage(state: dict[str, torch.Tensor], stage: str, strip: str) -> dict[str, torch.Tensor]:
    raise NotImplementedError


def pack_layer(sd: dict[str, torch.Tensor], i: int) -> PackedLayer:
    raise NotImplementedError


def load_thinker_text(state: dict[str, torch.Tensor], num_layers: int) -> tuple[dict, list[PackedLayer]]:
    raise NotImplementedError
```

What the tests check

- ✓ Picks only this stage's shards via weight_map
- ✓ select_stage strips the prefix and drops other stages
- ✓ Packed qkv / w13 / w2 reproduce HF projections and expert outputs
- ✓ Both missing and extra keys raise

Follow-up questions

- How is qkv split for TP=2? Why can't you simply split the packed qkv in half by rows?
- Why does the talker process also need some thinker weights (Qwen2.5-Omni's embed_tokens)?

Intuition & analogies

Analogy · Moving out of a shared storage unit

Three households (thinker / talker / code2wav) keep their things in the same storage unit, packed into several big boxes (shards), with every item labeled "household.room.item" (the key). At the door is a master list (`weight_map`): which item is in which box.

On moving day each household hauls only the boxes **that contain its own things** (`shards_for_stage`) - some boxes mix two households' things, so both must haul them. After unpacking, take only items whose label starts with your household name, and tear the "household." off the label (`select_stage`), since once it's in your own home you don't need the name.

Labels must be compared **including the dot**: the names `thinker` and `thinker_extra` start with the same letters, but they're not the same household.

© 2026 kzhou003

Analogy · Key ring and packing list

Packing: the q, k, v keys open the same door (all take hidden as input), so put them on one ring (`qkv = cat(..., dim=0)`); one motion opens the door (one GEMM), then split the result by `[H·D, KVH·D, KVH·D]` . Each expert's gate / up are strung into `w13` the same way - but the order is a convention: **gate first, then up**. String them backwards and the key still fits; it just opens the wrong door.

Reconciliation: after the move, tick off the packing list item by item (the `used` set). Anything not ticked (extra keys) and anything on the list but not found (missing keys) must stop the move (`LoadError`), rather than quietly moving in.

Framework view: shape trace and pitfalls

Running `load_thinker_text` on a tiny random thinker (hidden=64, H=4, KVH=2, D=16, E=6, I=32), layer 0 (measured):

HF key (prefix stripped)	HF shape	After packing	What happens
`self_attn.q_proj.weight`	`[64, 64]`	`qkv [128, 64]`	Concatenated along dim 0: 64 + 32 + 32 = (4 + 2·2)·16 = 128 rows. The q from `(x @ qkv.T).split([64, 32, 32])` vs HF `q_proj(x)` : max diff 0.0
`self_attn.k_proj.weight`	`[32, 64]`		
`self_attn.v_proj.weight`	`[32, 64]`		
`mlp.experts.{e}.gate_proj / up_proj`	`[32, 64]` ×2 ×6	`w13 [6, 64, 64]`	cat within each expert, then stack over e = 0..5; after reconstructing SwiGLU, max diff vs the HF expert output is 0.0
`mlp.experts.{e}.down_proj`	`[64, 32]` ×6	`w2 [6, 64, 32]`	stack over e
`mlp.gate.weight`	`[6, 64]`	`router [6, 64]`	It's named gate, but it's the router, not gate_proj
norm × 4, `o_proj`	as is	as is	input_layernorm / post_attention_layernorm / q_norm / k_norm
outside layers	`embed_tokens.weight` , `norm.weight`		The tiny model has 56 keys total: 2 outside layers + 2 layers × 27

Interactive (Chinese): `nano-vllm-omni-drills/hints/m07_overview.html`

Source: `model_executor/models/qwen3_omni/qwen3_omni.py:1725-1783`
`model_executor/models/qwen3_omni/qwen3_omni_moe_talker.py:61-77`
`model_executor/models/qwen2_5_omni/qwen2_5_omni.py:1009-1022` HF: Qwen3OmniMoeThinkerTextModel.state_dict()

Notes / pitfalls I hit:

© 2026 kzhou003

#8 · M08 Putting it together: KV-cache greedy decoding for a tiny thinker

60 min | medium

□ Done Date _____

> Assemble M01/M02/M03/M07 into a text-only thinker: prefill once, then compute 1 token per step; the generated token sequence must exactly match HF greedy with full recompute at every step.

Where it sits in the system

This is what the stage 0 model runner does each step (minus multimodal and paged KV). After this drill you have a "real model" that can plug into the Track B scheduler and be driven by the Track A orchestrator.

Structure

```
h = embed_tokens[ids]                                  # [T, hidden]
for layer:  h = h + Attn(rms(h, ln1), cos, sin, cache)  # M02 (one cache per layer)
            h = h + MoE(rms(h, ln2))                    # M03
logits = rms(h, norm) @ lm_head^T                       # [T, vocab]
```

Positions: for pure text the three axes are equal, `positions = arange(start, start+T).expand(3, T)`, using M01's interleaved MRoPE. Greedy: one forward over the prompt (start=0), then each step feeds only the previous token, `start = len(prompt) + num_generated − 1`.

Dependencies

This drill uses your own M01/M02/M03/M07 directly via `from .m01_... import ...` etc. Any small bug in a prerequisite drill shows up here as "diverges starting at token k".

File: `mine/m08_tiny_thinker_e2e.py` **Check:** `pytest tests/test_m08_tiny_thinker_e2e.py` **Hints:** A-8 **Tests:** B-8 **Solution:** C-8

Interface skeleton

```
"""M08 · Assemble a text-only thinker from YOUR m01/m02/m03/m07 and decode greedily.

TinyThinker(state, dims): load with your load_thinker_text; lm_head = state["thinker.lm_head.weight"].
forward(ids[T], start, caches) -> logits [T, V] for tokens at absolute positions start..start+T-1:
  text-only MRoPE positions (the same index on all 3 axes, interleaved layout),
  h = embed[ids]; per layer: h += attn(rms(h, ln1)); h += moe(rms(h, ln2)); logits = rms(h, norm) @ lm_head.T
  One KVCache per layer.
greedy(prompt, max_new): prefill the prompt once, then decode ONE token per forward using the
  caches. Must equal HF full-recompute greedy decoding.
"""
from dataclasses import dataclass

import torch

from .m02_thinker_attention import KVCache  # noqa: F401  (re-exported for the tests)


@dataclass
class Dims:
    num_layers: int
    num_heads: int
    num_kv_heads: int
    head_dim: int
    top_k: int
    norm_topk_prob: bool
    rope_theta: float
    mrope_section: list[int]
    eps: float = 1e-6


class TinyThinker:
```


© 2026 kzhou003

```
    def __init__(self, state: dict[str, torch.Tensor], dims: Dims) -> None:
        raise NotImplementedError

    def forward(self, ids: torch.Tensor, start: int, caches: list) -> torch.Tensor:
        raise NotImplementedError

    def greedy(self, prompt: list[int], max_new: int) -> list[int]:
        raise NotImplementedError
```

What the tests check

✓ Prefill logits match HF

✓ KV-cache greedy for 12 tokens exactly matches HF full-recompute greedy (two sets of random weights)

Follow-up questions

- Replace KVCache with P06 BlockManager + P09 paged attention.
- Add M04's multimodal merge and switch positions to P18's MRoPE positions.

Intuition & analogies

Analogy · A new house needs a street number

The KV cache is a street: the houses already built (tokens in the cache) are numbered 0, 1, 2, ..., L-1, and each house's K was "rotated" by its own number when it was moved into (RoPE). Now you're building a new house, and its number must be **L** - the number of houses already on the street.

Attention looks at "how far apart two houses are" (RoPE's relative position). Number the new house L+1 and its distance to every neighbor is off by 1: no error, it looks about right, but the neighborhood has quietly changed - and often it only "takes a wrong turn" a few steps later.

Analogy · A word-chain game

prefill = read the whole prompt first and say the first chain word along the way. After that, each round only needs to **hear the latest word** (feed only `out[-1:]`), because everything before is written in the notebook (the cache).

To say `max_new` words you need only `max_new` rounds: the first word comes from reading the prompt, and after the last word is spoken there's no need to listen for another round. At the start of each round, also check how many words the notebook holds (`len(cache)`) - that's the slot this round's new word takes (`start`).

Framework view: shape trace and pitfalls

Stage	Tensor	Shape (prefill / decode)	What happens
positions	`positions`	`[3, 5]` / `[3, 1]`	Text only: all three rows are `start … start+T-1`. Here interleaved MRoPE is identical to plain 1D RoPE (measured allclose) - but still go through M01's MRoPE interface
	`cos, sin`	`[5, 16]` / `[1, 16]`	Cover only the **new** tokens; shared by all layers
embed	`h = embed[ids]`	`[5, 64]` / `[1, 64]`	Table lookup

© 2026 kzhou003

each layer	`h + attn(rms(h, ln1), cos, sin, cache)`	`[T, 64]`	Pre-norm residual; cache appends T in place: 0→5 / 5→6
	`h + moe(rms(h, ln2))`	`[T, 64]`	top-2 of 6 experts
output	`rms(h, norm) @ lm_head.T`	`[5, 97]` / `[1, 97]`	`lm_head [97, 64]` comes from `state["thinker.lm_head.weight"]`, not under `thinker.model.`
sampling	`int(logits[-1].argmax())`	scalar	Only the last row is used

Interactive (Chinese): `nano-vllm-omni-drills/hints/m08_overview.html`

Source: `model_executor/models/qwen3_omni/qwen3_omni.py:235-267` HF: Qwen3OmniMoeThinkerTextModel

Notes / pitfalls I hit:

© 2026 kzhou003

Track A · Frontend & Orchestrator

Concurrency in the API process: thread ↔ event loop, single-writer routing, terminal messages, abort, RPC.

© 2026 kzhou003

#9 · P02 Orchestrator v1: linear multi-stage routing ⏱ 60 min medium □ Done Date _____

> You are writing vllm-omni's brain. Three stages (thinker→talker→code2wav) each emit outputs; you must chain them correctly and guarantee every request finishes exactly once.

Where it sits in the system

The API process has a dedicated **orchestrator thread** running its own asyncio event loop. It holds a client for each stage (the peer is the EngineCore in the stage subprocess). It does three things: accept requests submitted by the frontend; read each stage's outputs; and decide whether an output goes to the client, is forwarded to the next stage, or both.

```
frontend add_request ─► [request handler] ─► stage0.submit
                                                  │
 get_output of stage0 / stage1 / stage2 ─► reader×3 ─► ready_q ─► [dispatcher] ─► output to frontend
                                                                        └─► stage(s+1).submit
```

Requirements

1. **Single writer**: one reader task per stage only moves data; the **only** dispatcher mutates request state. asyncio is cooperative, so as long as you don't await in the middle of a read-modify-write, you need no locks.
2. **Register before submit**: `request_states[rid]` must exist before the first `await stages[0].submit()`. The reason: submit may yield internally, and the stage may emit its first output very quickly.
3. **Forwarding**: when stage s reports `finished=True` for a request and s < max(final_stage_ids), build the next stage's input with `forward_fn(s+1, prompt, all outputs of stage s)` and submit it.
4. **Outputs and terminal message**: only outputs from stages in `final_stage_ids` go to the client (for Qwen3-Omni, {0: text, 2: audio}). Each request gets **exactly one** `finished=True`, and it is the last one. The condition is "all final stages have finished".
5. **Cleanup**: delete state after emitting the terminal message; outputs that arrive afterwards for an unknown request are dropped and counted (`dropped_outputs`).

Example

```
stage0: r:t0 r:t1 r:t2 r:t3 r:t4(fin)      → client gets 5 stage0 outputs (finished=False)
stage1: [R:T0..R:T2] [R:T3,R:T4](fin)      → not sent to client; on fin, forward to stage2
stage2: wav(R:T0) ... wav(R:T4)(fin)       → the last one is finished=True
```

File: `mine/p02_orchestrator.py` **Check:** `pytest tests/test_p02_orchestrator.py` **Hints:** A-9 **Tests:** B-9 **Solution:** C-9

Interface skeleton

```
"""P02 · Orchestrator v1: linear multi-stage routing with a single-writer dispatcher.

Requirements
- start(): one reader task per stage client (awaits client.get_output()) feeding ONE
  ready queue; one dispatcher task processes it serially; one request-handler task
  consumes add_request() messages.
- add: register ReqState in self.request_states BEFORE awaiting stages[0].submit().
- When stage s finishes a request and s < max(final_stage_ids): build the next stage's
  inputs with forward_fn(s+1, prompt, all data stage s produced) and submit to s+1.
- Outputs of stages in final_stage_ids go to the client (next_output()). Exactly one
  message per request has finished=True: the last one, sent once every final stage finished.
```


© 2026 kzhou003

```
- After the terminal message the request state is removed. Outputs for unknown request
  ids are dropped and counted in self.dropped_outputs.
"""
import asyncio
from collections.abc import Callable
from dataclasses import dataclass, field
from typing import Any, Callable, Protocol


# ---------------------------------------------------------------- given types
@dataclass
class StageOutput:
    request_id: str
    data: Any
    finished: bool


@dataclass
class OrchOutput:
    request_id: str
    stage_id: int | None
    data: Any
    finished: bool
    error: str | None = None


class StageClient(Protocol):
    async def submit(self, request_id: str, inputs: Any) -> None: ...
    async def get_output(self) -> list[StageOutput]: ...
    async def abort(self, request_ids: list[str]) -> None: ...


@dataclass
class ReqState:
    request_id: str
    prompt: Any
    submitted: set[int] = field(default_factory=set)
    finished: set[int] = field(default_factory=set)
    stage_data: dict[int, list] = field(default_factory=dict)
    pending_final: OrchOutput | None = None  # used by P03


# ---------------------------------------------------------------- implement below
class Orchestrator:
    def __init__(
        self,
        stages: list[StageClient],
        final_stage_ids: set[int],
        forward_fn: Callable[[int, Any, list], Any],
    ) -> None:
        self.stages = stages
        self.final_stage_ids = frozenset(final_stage_ids)
        self.last_stage = max(final_stage_ids)
        self.forward_fn: Callable[[int, Any, list[Any]], Any] = forward_fn
        self.request_states: dict[str, ReqState] = {}
        self.dropped_outputs = 0
        self._req_q: asyncio.Queue = asyncio.Queue()
        self._ready_q: asyncio.Queue = asyncio.Queue()
        self._out_q: asyncio.Queue = asyncio.Queue()
        self._tasks: list[asyncio.Task] = []

    async def start(self) -> None:
        raise NotImplementedError

    async def shutdown(self) -> None:
        raise NotImplementedError

    async def add_request(self, request_id: str, prompt: Any) -> None:
        raise NotImplementedError

    async def next_output(self, timeout: float | None = None) -> OrchOutput:
        raise NotImplementedError

    # Suggested internals (P03 overrides the last two):
    #   async def _reader(self, sid, client)
    #   async def _request_handler(self)
    #   async def _dispatcher(self)
    #   async def _handle_add(self, rid, prompt)
```


© 2026 kzhou003

```
#   async def _handle_output(self, sid, o: StageOutput)
#   def _cleanup(self, rid)
```

What the tests check

- ✓ Single request, 3 stages: stage0 text and stage2 audio both arrive in order; the only finished is the last
- ✓ 30 concurrent requests don't cross-talk; each stage receives each request exactly once
- ✓ When only the last stage is final, stage0 outputs don't leak to the client
- ✓ A stage emitting its first output inside submit() must not be lost (register before submit)
- ✓ Outputs for unknown requests are dropped and counted, no crash

Follow-up questions

- If a stage has multiple replicas, can output order still be guaranteed? Under what precondition? (→ P04 sticky binding)
- Why can the output queue be unbounded while the request queue should be bounded?
- What if you replaced the dispatcher with one task per request? Benefits and costs?

Interactive (Chinese): `nano-vllm-omni-drills/hints/p02_flow.html (timeline) · hints/p02_vs_vllm_omni.html (source mapping, see Appendix D)`

Source: `engine/orchestrator.py:1099-1320` `engine/orchestrator.py:1912-2046` `engine/orchestrator.py:3062-3128` `engine/orchestrator.py:232-271`

Notes / pitfalls I hit:

© 2026 kzhou003

#10 · P03 Orchestrator v2: async_chunk prewarm + terminal gating ⏱ 45 min medium

□ Done Date _____

> With async_chunk enabled, stages talk directly through connectors and the orchestrator no longer forwards data. But the control plane and data plane are two paths, and completion signals can arrive out of order: the audio stage says "I'm done" before the talker has reported completion.

Where it sits in the system

In async_chunk mode, the thinker sends a vector to the talker via a connector for every decoded token, and the talker sends codec to code2wav every time it has a full window; the data **does not go through** the orchestrator. The orchestrator does only two things: **prewarm** downstream stages when a request arrives (submit a placeholder request that waits in the scheduler for the first chunk), and **gate** the terminal message.

Why prewarm

If you submit the talker request only after the thinker produces its first chunk, you add a "stage→orchestrator→stage" round trip; with prewarm, the talker can start the moment the first chunk lands.

Why gate the terminal message

```
time →
stage2 (audio, final):  ... wav5(fin)                ← arrives first on the control plane
stage1 (talker):        ............. (fin)          ← arrives late (its IPC lagged a beat)
If you emit finished and clean up at wav5(fin):
  – cleanup reclaims resources stage1 is still using (SHM, bindings)
  – stage1's completion message becomes an output for an "unknown request"
```

Rule: when the last final stage finishes but `submitted ⊄ finished`, stash the terminal message in `pending_final`; emit it when the last submitted stage also finishes. Still exactly one terminal message, and it is the last.

Requirements checklist

- add: register state → mark 0..last all as submitted → submit stage 0 first (real prompt), then downstream stages (placeholder input from `prewarm_fn(sid, prompt)`).
- No forwarding at all.
- Outputs arriving after cleanup are dropped and counted; state is not resurrected.

This drill directly subclasses the `Orchestrator` you wrote in P02, overriding `_handle_add` and `_handle_output`.

File: `mine/p03_async_chunk_orchestrator.py` **Check:** `pytest tests/test_p03_async_chunk_orchestrator.py` **Hints:** A-10 **Tests:** B-10 **Solution:** C-10

Interface skeleton

```
"""P03 · Orchestrator v2 for async_chunk (builds on YOUR P02).

Stages now exchange data on a side channel (the connector). The orchestrator:
– on add: registers state, marks stages 0..last as submitted, submits the prompt to
  stage 0 and a placeholder prewarm_fn(sid, prompt) to every downstream stage;
– never forwards stage outputs itself;
```


© 2026 kzhou003

```
- emits outputs of final stages, but the terminal message (finished=True) is only
  released once EVERY submitted stage has finished. If the last final stage finishes
  first, hold the message in ReqState.pending_final and emit it when the remaining
  stage(s) finish. Exactly one terminal message, always the last one.
- outputs after cleanup are dropped (self.dropped_outputs), never resurrect state.
"""
from collections.abc import Callable
from typing import Any

from .p02_orchestrator import Orchestrator, OrchOutput, ReqState, StageClient, StageOutput

__all__ = ["AsyncChunkOrchestrator", "OrchOutput", "StageOutput", "ReqState"]


class AsyncChunkOrchestrator(Orchestrator):
    def __init__(
        self,
        stages: list[StageClient],
        final_stage_ids: set[int],
        prewarm_fn: Callable[[int, Any], Any],
    ) -> None:
        super().__init__(stages, final_stage_ids, forward_fn=lambda *_: None)
        self.prewarm_fn = prewarm_fn

    # Override the add path and the per-output handling of your P02.
    async def _handle_add(self, rid: str, prompt: Any) -> None:
        raise NotImplementedError

    async def _handle_output(self, sid: int, o: StageOutput) -> None:
        raise NotImplementedError
```

What the tests check

- ✓ Each stage is submitted only once; downstream stages get the prewarm placeholder input
- ✓ When non-final stage1 reports completion 200ms late, the terminal message must wait for it (no early finished)
- ✓ 20 requests + late completions: each request gets exactly one terminal message and it is the last
- ✓ Outputs arriving after cleanup are dropped; state is not resurrected

Follow-up questions

- Why does emitting the terminal message too early cause resource problems? (Hint: cleanup reclaims SHM the upstream is still writing.)
- What if the prewarm placeholder prompt's length depends on upstream results?

Intuition & analogies

Analogy · Shop-floor foreman and conveyor belts

P02: each station (stage) finishes **a whole batch** of parts and hands it to the foreman (orchestrator), who carries it to the next station. The next station must **wait for the previous one to finish everything** before starting.

P03: **conveyor belts** (connectors) run between stations. Each time the previous station finishes **one** part, it goes straight onto the belt and the next station can work on it immediately - the line runs in parallel. The foreman no longer carries parts; only two jobs remain:

① **At start**: tell **every** station "new order, get ready for parts on the belt" (**prewarm**);

② **At close**: confirm that **every** station has stopped before giving the customer the receipt (**terminal-message gate**).

© 2026 kzhou003

Analogy · Preheating the oven

No preheat: only once the dough is kneaded (the thinker's first token) do you turn on the oven (submit talker), and then you wait for it to heat up.

Preheat: turn on the oven the moment the customer orders (prewarm), and the dough goes straight in when it's ready. For the user, that means **the first audio chunk comes out sooner** (lower time-to-first-audio, TTFA).

Analogy · Parcels and delivery texts

Goods travel by freight (data plane); "shipped / delivered" travels by text message (control plane). The final stop's "delivered" text can reach your phone **before** the transit hub's "dispatched" text.

If you **archive and delete** the order the moment you see "delivered", then when the hub's "dispatched" text arrives a few seconds later, you can't find the order; worse, when archiving you may already have reclaimed the **pallets** (shared memory, bindings) the hub was still using.

Framework view · P03 through T · SBWC · LIFE: only two things changed

	Compared with P02
T	Unchanged: tasks in the same event loop. The new connector is a boundary between stage processes; the orchestrator doesn't touch it.
S	`ReqState` gains one field, `pending_final`; `submitted` now means "all prewarmed stages"; `stage_data` is no longer needed.
B / W / C	**Completely unchanged**: reader × N, request_handler, dispatcher, three queues. That's why P03 can inherit P02 directly.
L / I	Unchanged.
F	**Changed**: mark submitted all at once on add; the terminal condition adds `submitted ⊆ finished`; **a non-final stage's completion event can also trigger a send** (releasing pending_final). "Exactly one terminal message, and it's the last one" is unchanged.
E	Outputs arriving after cleanup are still dropped and counted; state must **never** be recreated because of S1's late completion message.

Interactive (Chinese): `nano-vllm-omni-drills/hints/p03_overview.html`

Source: `engine/orchestrator.py:1954-1967` `engine/orchestrator.py:2036-2040` `engine/orchestrator.py:2711-2870`

Notes / pitfalls I hit:

© 2026 kzhou003

#11 · P01 OutputSink: batched wakeups from thread → asyncio ⏱ 45 min medium

□ Done Date _____

> The orchestrator thread produces tens of thousands of outputs per second, and the frontend asyncio loop must consume them. call_soon_threadsafe per item is too expensive; waking only once easily loses wakeups.

Where it sits in the system

The API process has two event loops: the frontend loop (HTTP, SSE, tokenize) and the loop in the orchestrator thread. The orchestrator hands each token and each audio chunk to the frontend as a message. The only way to wake asyncio from another thread is `loop.call_soon_threadsafe`: it writes a byte to the loop's self-pipe and takes a lock, costing on the order of microseconds per call, which adds up at tens of thousands per second.

Target design

```
producer (any thread)              consumer (frontend loop)
put(x):                          get_batch():
  items.append(x)                  ① out = drain();  if out: return out
  if wakeup in flight: return      ② event.clear(); mark "no wakeup in flight"  ← arm
  mark "wakeup in flight"          ③ out = drain();  if out: return out  ← the key step
  call_soon_threadsafe(event.set)  ④ await event.wait() (with timeout)
```

How a wakeup gets lost

Without step ③: the consumer finds the queue empty at ① → the producer appends an item right then and sees the "wakeup in flight" flag (left over from the previous round), so it doesn't wake → the consumer arms at ② → sleeps at ④. That message sits in the queue until the next put or the timeout. The test uses the `after_empty_drain` hook to insert a put precisely between ① and ②.

Requirements

- put is thread-safe and never blocks; order from the same producer is preserved.
- At most one `call_soon_threadsafe` per "wakeup cycle", counted in `wakeups_scheduled`.
- get_batch returns 1..max_items items; returns [] on timeout.

File: `mine/p01_output_sink.py` **Check:** `pytest tests/test_p01_output_sink.py` **Hints:** A-11 **Tests:** B-11 **Solution:** C-11

Interface skeleton

```
"""P01 · OutputSink: many producer threads -> one asyncio consumer.

Requirements
- put(item): callable from ANY thread, never blocks, FIFO per producer.
- At most ONE loop.call_soon_threadsafe(...) per "wake cycle": while a wakeup is
  already in flight, further puts must not schedule another. Count every scheduled
  wakeup in self.wakeups_scheduled.
- get_batch(max_items, timeout): runs on the bound loop; returns 1..max_items items,
  or [] on timeout. It must never sleep while items are available (no lost wakeups).
- Test hook: call self.after_empty_drain() right after a drain that found nothing and
  BEFORE you re-arm the wakeup. Tests use it to inject a put() from another thread into
  exactly the window where naive implementations lose the wakeup.
"""
import asyncio
```


© 2026 kzhou003

```
class OutputSink:
    def __init__(self) -> None:
        self.wakeups_scheduled = 0
        self.after_empty_drain = lambda: None
        raise NotImplementedError

    def bind_loop(self, loop: asyncio.AbstractEventLoop | None = None) -> None:
        """Remember the consumer loop (default: the running loop) and create its wakeup Event."""
        raise NotImplementedError

    def put(self, item) -> None:
        raise NotImplementedError

    async def get_batch(self, max_items: int = 256, timeout: float | None = None) -> list:
        raise NotImplementedError
```

What the tests check

- ✓ Basic FIFO
- ✓ 1000 puts while the frontend isn't consuming schedule only 1 wakeup
- ✓ Timeout returns []
- ✓ 4 producer threads × 5000 items: per-producer order preserved, consumer not starved
- ✓ 300 ping-pong rounds without lost wakeups
- ✓ A put injected "after an empty drain, before re-arming" must be picked up immediately

Follow-up questions

- Does this still work with multiple consumers (multiple asyncio loops)?
- What does janus.Queue solve for you, and what doesn't it?

Intuition & analogies

Analogy · Mailbox and doorbell

You (the **frontend loop**) live upstairs; downstairs there's a **mailbox** (the queue). The mail carrier (the **orchestrator thread**) may drop off letters at any time.

You can't see the mailbox from upstairs; you only come down when the carrier **rings the doorbell** (`loop.call_soon_threadsafe`).

But every ring makes you drop what you're doing and run down the stairs - **expensive**. If the carrier rang once per letter, at tens of thousands of letters a second you'd spend all day on the stairs.

Analogy · The little flag on the door

Next to the mailbox stands a **little flag** (the "wakeup in flight" flag). The rules:

① After dropping off a letter, if the flag is **down**, the carrier **raises it** and rings once; if the flag is **already up**, the bell has been rung and you're on your way down, so he **doesn't ring again**.

② Once downstairs, you take **everything** out of the mailbox at once.

③ Only when you're about to go back upstairs to rest do you **lower** the flag, meaning "next letter needs a ring".

© 2026 kzhou003

Analogy · How a wakeup gets lost

You go downstairs and open the mailbox: **empty**. You turn to go back up and rest.

In the instant you turn, **before you've lowered the flag**, the carrier drops off a letter. He sees the flag is still **up** (left over from your last trip down), thinks "the bell's been rung, he'll come get it", and leaves **without ringing**.

You lower the flag and go to sleep upstairs. The letter **sits in the mailbox until morning** (until the next letter arrives, or your alarm - the timeout - goes off).

Walk through it with the T · SBWC framework (think first)

	Question	Hint
T	Where are the producers and the consumer? Which boundary is crossed?	Several ordinary threads → one asyncio loop. In the framework's "thread → event loop" row: which bridge is the only one allowed?
S	What shared state is there? Who reads, who writes?	At least two things: "data waiting to be taken" and "whether a wakeup is in flight". Which one is modified by multiple threads at once? Do you need a lock? Which operations are already atomic in CPython?
B	Who may block? Who must never block?	`put` never blocks; `get_batch` may only `await` inside the loop, never wait with `time.sleep` or a thread lock.
F	What's the invariant?	"Queue non-empty ⇒ either the consumer is awake, or a wakeup is in flight". At which step does the timeline above violate it?
E	Edge cases?	Timeouts; `max_items` smaller than the backlog (what happens to the rest? will the next `get_batch` wait for nothing?); multiple producers interleaving.

Interactive (Chinese): `nano-vllm-omni-drills/hints/p01_overview.html`

Source: `engine/async_engine_utils.py:40-97` `engine/omni_engine_base.py:973-991`

Notes / pitfalls I hit:

© 2026 kzhou003

#12 · P04 StagePool: sticky binding, two-phase abort, fault isolation ⏱ 60 min medium

□ Done Date _____

> Each stage can have multiple replicas. Once a request lands on a replica it must stay there (its KV is there). A client disconnect means abort; if a replica process dies, only the requests bound to it may fail.

Where it sits in the system

`num_replicas: 2` in the deploy config starts two independent EngineCore processes for a stage. The orchestrator keeps one `StagePool` per logical stage, responsible for choosing replicas.

Why sticky binding is required

A request's KV cache and the intermediate state of streaming input live on the replica it was first assigned to. Subsequent update / abort / next chunk must find the same replica, otherwise the peer doesn't even know the request. New requests are round-robined across **live** replicas.

Two-phase abort

```
① partial = processor.collect_partial(rid)   # read-only: get the partial text generated so far, for the final
output
② for pool: await pool.abort([rid])           # actually stop on the engine side; on failure raise, don't
enter ③
③ processor.commit_abort(rid)                # commit: drop detokenizer state
return OrchOutput(rid, partial, finished=True, error="abort")
```

If ③ happens before ②: when the engine abort fails, the processor has already forgotten the request while the engine keeps producing tokens for it, and those outputs become orphans. Also, multiple requests on the same replica must be **merged into one** abort call (one IPC).

Fault isolation

A replica process crashes (the reader receives EngineDead): mark it dead, end only the requests **bound to it** with an error; requests on other replicas proceed; new requests are no longer assigned to it. If all replicas are dead, `pick` raises `NoLiveReplica`.

File: `mine/p04_stage_pool_abort.py` **Check:** `pytest tests/test_p04_stage_pool_abort.py` **Hints:** A-12
Tests: B-12 **Solution:** C-12

Interface skeleton

```
"""P04 · StagePool (sticky replica binding) + two-phase abort + dead-replica isolation.

StagePool
- pick(rid): sticky -- a request keeps its replica; new requests round-robin over LIVE
  replicas; raise NoLiveReplica if none.
- evict(replica): mark dead, drop and return the request ids bound to it.
- abort(ids): ONE abort call per replica with the ids bound to it; ids without a binding
  are ignored; bindings are released only after that replica's abort succeeded
  (if it raises, propagate and keep the bindings so the abort can be retried).

abort_request(pools, processor, rid)  -- two phase
  1. partial = processor.collect_partial(rid)        (read only)
  2. await pool.abort([rid]) for every pool          (any failure propagates; no commit)
  3. processor.commit_abort(rid)
  return OrchOutput(rid, partial, finished=True, error="abort")

handle_dead_replica(pool, replica) -> one terminal error OrchOutput per affected request;
requests bound to other replicas are untouched.
"""
```


© 2026 kzhou003

```python
from dataclasses import dataclass
from typing import Any, Protocol


# ---------------------------------------------------------------- given types
class NoLiveReplica(Exception):
    pass


class ReplicaClient(Protocol):
    async def abort(self, request_ids: list[str]) -> None: ...


class PartialProcessor(Protocol):
    def collect_partial(self, request_id: str) -> Any: ...
    def commit_abort(self, request_id: str) -> None: ...


@dataclass
class OrchOutput:
    request_id: str
    data: Any
    finished: bool
    error: str | None = None


# ---------------------------------------------------------------- implement below
class StagePool:
    def __init__(self, stage_id: int, replicas: list[ReplicaClient]) -> None:
        self.stage_id = stage_id
        self.replicas = replicas
        raise NotImplementedError

    def live_replicas(self) -> list[int]:
        raise NotImplementedError

    def pick(self, request_id: str) -> int:
        raise NotImplementedError

    def bound(self, request_id: str) -> int | None:
        raise NotImplementedError

    def release(self, request_id: str) -> None:
        raise NotImplementedError

    def evict(self, replica_id: int) -> list[str]:
        raise NotImplementedError

    async def abort(self, request_ids: list[str]) -> None:
        raise NotImplementedError


async def abort_request(pools: list[StagePool], processor: PartialProcessor, request_id: str) -> OrchOutput:
    raise NotImplementedError


def handle_dead_replica(pool: StagePool, replica_id: int) -> list[OrchOutput]:
    raise NotImplementedError
```

What the tests check

- ✓ round-robin + sticky
- ✓ evict returns affected requests and nothing is assigned to the dead replica afterwards; raises NoLiveReplica when none are live
- ✓ abort is grouped by replica, one call per replica
- ✓ Two-phase order: collect → abort on each stage → commit
- ✓ On abort failure, don't commit and keep the binding for retry
- ✓ A dead replica fails only the requests bound to it

© 2026 kzhou003

Follow-up questions

- What happens if abort and ADD are reordered inside EngineCore? (→ P10)
- With multiple final stages, who should emit the abort terminal message, and how many times?

Intuition & analogies

Analogy · Luggage storage counters (sticky binding)

The train station has 3 luggage counters (replicas). On your first visit, staff assign you a counter in turn (round-robin), and your luggage (the KV cache) is stored at **that** counter. From then on, whether you add more, pick up, or check out, you **must return to the same counter** - the others have none of your things. Only new travelers are assigned in rotation; if a counter closes (evict), rotation continues among the open ones. When a counter closes, the luggage stored there is gone too: notify the travelers who stored things there (handle_dead_replica) and cross those lines out of the register - otherwise the next person holding an old ticket just finds a closed door.

Analogy · Cancelling an order already split across several factories (two-phase abort)

A customer's order has been split across three factories (the thinker / talker / code2wav replicas), each making a part. Now the customer cancels:

① **Take a photo** (`collect_partial` , read-only): first photograph the half-finished goods, to send to the customer at the end;

② **Tell every factory to stop** (`await pool.abort([rid])`): a call may not get through (IPC failure) - **if it doesn't, stop and don't close the account**;

③ **Close the account** (`commit_abort` , irreversible): only after every factory confirms it has stopped, delete the order from the ledger;

Finally, send the customer a "cancelled" notice with the photo of the partial goods: `OrchOutput(rid, partial, finished=True, error="abort")` .

If you close the account first and the call then fails: the factory keeps working, the finished goods come back, and the ledger has no such order (orphan output); and if you want to retry, the photo is gone too.

Framework view: T · SBWC · LIFE

	Dimension	P04's answer
T	Topology	Everything runs in the orchestrator's **single asyncio event loop**. Replicas are other processes, but `ReplicaClient.abort` wraps them as an awaitable interface. No threads, no locks needed.
S	State	Per pool: `alive` (or a dead set), `bindings: dict[rid → replica]` , a round-robin counter. Orchestrator side: each request's partial results in PartialProcessor. Single writer: all mutated only inside the event loop.
B	Blocking	The only yield point is `await replica.abort(ids)` . pick / bound / release / evict are all synchronous methods and never interleave with each other.
W	Workers	No new workers here: the callers are P02's request_handler (handling abort messages) and dispatcher (detecting a dead replica).
C	Channels	**add and abort travel on the same control queue**, so "add then abort" for the same request can't be reordered; each replica call is one IPC, batched per replica.
L	Lifecycle	A binding's life: created by `pick` → `release` on normal completion / released after a successful abort / deleted on `evict` . All three paths must exist, or `bindings` only grows.

© 2026 kzhou003

I	Interface	`abort_request` either returns **exactly one** terminal message with `error="abort"`, or raises and leaves the world retryable; `handle_dead_replica` sends **exactly one** error terminal message per affected request.
F	Flow + Invariants	① A binding exists ⇔ that replica (may) still hold state for this request; ② the irreversible commit always comes after every abort that could fail; ③ `pick` never returns a dead replica.
E	Edges + Errors	Ids with no binding (never dispatched to this stage / already released / already evicted) → silently ignored; some replicas fail → release the successful ones, keep the failed ones; all replicas dead → `NoLiveReplica`; the binding may be deleted elsewhere during the `await` → release must tolerate "already gone".

Interactive (Chinese): `nano-vllm-omni-drills/hints/p04_overview.html`

Source: `engine/stage_pool.py:1279-1339` `engine/stage_pool.py:333` `engine/orchestrator.py:1464-1515` `engine/orchestrator.py:645-731`

Notes / pitfalls I hit:

© 2026 kzhou003

#13 · P05 RpcRouter: RPC correlation over a shared channel ⏱ 40 min easy

□ Done Date _____

> Results of sleep / wake_up / collective_rpc share the same return queue with other control messages, and multiple threads are waiting for their own results at the same time.

Where it sits in the system

The frontend's `collective_rpc`, `abort` and `sleep` are synchronous APIs: the calling thread puts a request into the request queue and blocks waiting for the result. All results come back on the same `rpc_output_queue`, and a router thread dispatches them to waiters by key.

Three classic pitfalls

1. **Send before register**: the result may arrive before registration → it's treated as "unclaimed" and dropped → the caller waits forever. You must **register before sending**.
2. **Not unregistering after timeout**: a late result gets stuffed into a queue nobody reads; reusing the same key later reads the stale result. You must unregister in finally, and only your own waiter (compare with `is`).
3. **No broadcast on fatal error**: if the orchestrator thread dies, all waiters must get the error immediately rather than each waiting until timeout; new calls afterwards should also fail immediately (latched).

Interface

`register / unregister / deliver / fail_all / call(key, send, timeout)`; all methods may be called concurrently from different threads. Each waiter is a `queue.Queue(maxsize=1)`.

File: `mine/p05_rpc_router.py` **Check:** `pytest tests/test_p05_rpc_router.py` **Hints:** A-13 **Tests:** B-13 **Solution:** C-13

Interface skeleton

```
"""P05 · RpcRouter: correlate results arriving on one shared channel with blocked callers.

- register(key) -> waiter (a queue.Queue(maxsize=1)); duplicate live key -> ValueError.
- deliver(key, result) -> True if a waiter got it, else False and self.late_results += 1.
- call(key, send, timeout): register FIRST, then send(), then wait; always unregister in
  finally; raise TimeoutError on timeout; if the result is an exception, raise it.
- fail_all(err): wake every current waiter with err and latch it: later register()/call()
  raise err immediately.
All methods are called from arbitrary threads.
"""
import queue
from collections.abc import Callable, Hashable
from typing import Any


class RpcRouter:
    def __init__(self) -> None:
        self.late_results = 0
        raise NotImplementedError

    def register(self, key: Hashable) -> queue.Queue:
        raise NotImplementedError

    def unregister(self, key: Hashable, waiter: queue.Queue) -> None:
        raise NotImplementedError

    def deliver(self, key: Hashable, result: Any) -> bool:
```


© 2026 kzhou003

```
        raise NotImplementedError

    def fail_all(self, error: BaseException) -> None:
        raise NotImplementedError

    def call(self, key: Hashable, send: Callable[[], None], timeout: float) -> Any:
        raise NotImplementedError
```

What the tests check

- ✓ 50 threads call concurrently, results return out of order, each gets its own
- ✓ A result returned synchronously inside send() is not lost (register before send)
- ✓ Timeout → TimeoutError; late result dropped and counted; key reusable
- ✓ Duplicate key raises
- ✓ fail_all wakes all waiters immediately and latches the error

Follow-up questions

- Why does the orchestrator executing RPCs inline block admission of later requests? How would you change it?

Intuition & analogies

Analogy · The bubble-tea shop's pickup buzzer

When you order, the cashier first hands you a **pickup buzzer** (the `Queue(maxsize=1)` returned by `register(key)`), and **only then** passes the ticket to the kitchen (`send()`). The kitchen buzzes whichever number is ready (`deliver(key, result)`), regardless of order sequence. One buzzer holds one drink (maxsize=1).

① Why take the buzzer before ordering? The kitchen is fast; the drink may be ready the moment the ticket goes in - if you don't have a buzzer yet, the cashier has to put it under "unclaimed" (`late_results += 1`), and you wait at the counter for nothing.

② If you can't wait and leave (timeout), you **must return the buzzer** (`finally: unregister`). Any drink finished afterwards can only go to "unclaimed".

③ When returning it, you return **the one in your hand** (`unregister(key, waiter)` compares with `is`), not "cancel number 17" - number 17 may already have been handed to the next customer.

Analogy · Kitchen fire (fail_all)

The kitchen catches fire: the cashier doesn't let each customer wait until they give up; instead they **immediately announce** "all buzzers are void, please leave" (put the error into every waiting mailbox), and **hang a "temporarily closed" sign** on the door (the latch): customers arriving later don't even get to order - they're told at the door (`register / call` raise this error immediately, and `send` is never called).

Framework view: T · SBWC · LIFE

	Dimension	P05's answer
T	Topology	**Threads only**: N caller threads + 1 result-channel thread (in the real system, the router's own daemon thread) + an occasional thread calling fail_all. No event loop, so the sync primitives are `threading.Lock` and `queue.Queue`.
S	State	`waiters: dict[key → Queue(1)]`, the latched `fatal`, `late_results`. All three are read and written by multiple threads → protected by **one lock**; "check fatal + insert waiter" must happen under the same lock acquisition.

© 2026 kzhou003

B	Blocking	Callers block on `waiter.get(timeout)` - **never while holding the lock**; deliver / fail_all also put into the waiter outside the lock (capacity 1, put only once, so it never blocks).
W	Workers	Callers: register, send, wait, unregister their own; result thread: only pops the waiter by key and delivers, without looking at the result; fail_all: clears and broadcasts.
C	Channels	Out: `send()` (request_queue in the real system); back: one shared result channel; each call has a **private** `Queue(maxsize=1)` as its mailbox.
L	Lifecycle	A waiter's life: created by register → taken by deliver / cleared by fail_all / deleted by unregister in finally; exactly one of the three paths actually removes it from the table. The router's life: normal → permanently "locked" after fail_all.
I	Interface	`call` has only four outcomes: return the result / raise the exception in the result / `TimeoutError` / raise the latched fatal error; in every case, its waiter is no longer in the table when it returns.
F	Flow + Invariants	① At most one live waiter per key; ② each waiter receives at most one result (deliver uses pop, mutually exclusive with fail_all); ③ once fatal is set, no new waiter ever appears in the table; ④ registration always precedes sending.
E	Edges + Errors	Results for late / unknown keys → counted and dropped; duplicate key → ValueError; `send()` itself raises → finally still unregisters; a result arriving in the gap "already popped, caller just timed out" → lands in a mailbox nobody reads and is silently dropped (acceptable; it doesn't count as late).

Interactive (Chinese): `nano-vllm-omni-drills/hints/p05_overview.html`

Source: `engine/rpc_result_router.py:32-150` `engine/messages.py:125-148`

Notes / pitfalls I hit:

© 2026 kzhou003

Track B · Stage process & scheduling

EngineCore busy loop, park / restore, continuous batching, paged KV, sampling, attention.

© 2026 kzhou003

#14 · P10 EngineCore busy loop ⏱ 45 min medium

□ Done Date _____

> In each stage process, the IO thread pushes ZMQ messages into input_queue and the main thread runs schedule→execute→update. Requirements: zero CPU when idle, strict FIFO for ADD/ABORT, a signal can wake the blocked loop, and step exceptions are fatal.

Where it sits in the system

Stage subprocess = upstream vLLM's `EngineCoreProc`: an input IO thread (ZMQ → `input_queue`), the main busy loop, and an output IO thread (`output_queue` → ZMQ). vllm-omni doesn't change the busy loop itself, but much of its correctness depends on the busy loop's semantics.

```
while not shutdown:
    if not engine.has_work():  msg = input_queue.get()   # blocking: zero CPU when idle
    drain input_queue (FIFO): ADD / ABORT / UTILITY / WAKEUP
    if shutdown: break
    if engine.has_work():  for out in engine.step(): emit(("output", out))
```

Requirement details

- **FIFO**: ADD r1 → ABORT r1 → ADD r2 must be applied to the engine in this order; an aborted request must produce zero outputs.
- **Wakeable shutdown**: a signal handler or another thread calls `request_shutdown()`: set the flag and put WAKEUP on the queue; a loop blocked in get() wakes immediately, exits, and emits `("shutdown",)`.
- **UTILITY** (sleep/wake_up/collective_rpc, etc.): results return by call_id; a utility exception only affects that call, and the engine keeps running.
- **Fatal error**: step() raises (e.g. a CUDA illegal memory access) → emit `("fatal", repr(e))` and exit; the outer process monitor reports it as a stage death (P04).

File: `mine/p10_engine_core_loop.py` **Check:** `pytest tests/test_p10_engine_core_loop.py` **Hints:** A-14
Tests: B-14 **Solution:** C-14

Interface skeleton

```
"""P10 · The EngineCore busy loop (what runs inside every stage process).

IO threads (not part of this drill) push Msg objects into self.input_queue and ship
whatever you pass to emit(). Implement run_busy_loop() (it runs on its own thread):
- While the engine has no work, BLOCK on input_queue (no polling / spinning).
- Before every step, drain all queued messages in FIFO order:
    ADD     payload (req_id, data)          -> engine.add
    ABORT   payload [req_ids]               -> engine.abort
    UTILITY payload (call_id, name, args)   -> emit(("utility", call_id, result, None)),
            or emit(("utility", call_id, None, repr(exc))) if it raised (engine keeps running)
    WAKEUP  no-op (only unblocks the queue)
- If the engine has work: for out in engine.step(): emit(("output", out)).
- request_shutdown() may be called from any thread / a signal handler: the loop must exit
  promptly even when blocked, then emit(("shutdown",)).
- An exception from engine.step() is fatal: emit(("fatal", repr(exc))) and return.
"""
import queue
from collections.abc import Callable
from dataclasses import dataclass
from typing import Any, Literal, Protocol


# ---------------------------------------------------------------- given types
@dataclass
```


© 2026 kzhou003

```
class Msg:
    kind: Literal["ADD", "ABORT", "UTILITY", "WAKEUP"]
    payload: Any = None


class Engine(Protocol):
    def add(self, req_id: str, data: Any) -> None: ...
    def abort(self, req_ids: list[str]) -> None: ...
    def has_work(self) -> bool: ...
    def step(self) -> list[Any]: ...
    def utility(self, name: str, args: tuple) -> Any: ...


# ------------------------------------------------------------ implement below
class EngineCoreLoop:
    def __init__(self, engine: Engine, emit: Callable[[tuple], None]) -> None:
        self.engine = engine
        self.emit = emit
        self.input_queue: queue.Queue[Msg] = queue.Queue()

    def request_shutdown(self) -> None:
        raise NotImplementedError

    def run_busy_loop(self) -> None:
        raise NotImplementedError
```

What the tests check

- ✓ Idle 0.3s: step 0 times, has_work called < 10 times (no spinning)
- ✓ ADD then run to completion
- ✓ ADD→ABORT→ADD pre-queued: engine call order strictly FIFO, aborted request has zero outputs
- ✓ request_shutdown from another thread: exits within 0.5s and emits only ('shutdown',)
- ✓ step raises → ('fatal', ...) and exit
- ✓ utility results/exceptions returned by call_id without killing the engine

Follow-up questions

- Why does vllm-omni release overflowed requests only after the ABORT has gone through the FIFO?
- What is step_with_batch_queue doing when max_concurrent_batches>1?

Intuition & analogies

Analogy · The head chef and the ticket rail

Each "round" of cooking (`step()`), the head chef (the busy loop) advances **every** dish in the pans by one step. Servers (the input IO thread) clip tickets onto the ticket rail in order (`input_queue`).

① **Before firing up each round**, the chef processes **all tickets already on the rail** in order: new dishes, cancellations, customer questions (ADD / ABORT / UTILITY) - not just one per round.

② When there's nothing in the pans, the chef **dozes** next to the ticket printer (blocking `get()`): no walking back and forth to check (no polling); he wakes as soon as the printer goes off.

③ Closing time: the manager must hang the "closed" sign (set the shutdown flag) **and also print a blank ticket** (WAKEUP), or the dozing chef will never see the sign.

④ A customer asks "how much stock is left?" and the chef can't answer (utility raises) - just tell that customer "couldn't check", and the kitchen carries on; the stove catches fire (`step()` raises) - sound the alarm (`("fatal", ...)`) and evacuate, no more orders.

© 2026 kzhou003

> **Analogy · The cancellation ticket must come before the next round fires**
>
> The rail holds, in order: "one kung pao chicken (ADD q2)", "cancel the kung pao chicken (ABORT q2)". If the chef only read one ticket per round, he'd cook a round of kung pao chicken first and only see the cancellation next round - and what was cooked that round (one output) has already gone out. **Drain the rail first, then fire up**, and not a single bite of the cancelled dish ever leaves the kitchen.

Framework view: T · SBWC · LIFE

	Dimension	P10's answer
T	Topology	Inside one stage process: an input IO thread, the busy loop thread, an output IO thread, plus a signal handler / any thread calling `request_shutdown`. **No event loop**: threads communicate only via `queue.Queue` and `threading.Event`.
S	State	The engine (scheduler + KV + model) is **touched only by the busy loop thread**: single writer, no locks. The only cross-thread shared things are `input_queue` (has its own lock) and the shutdown flag (`threading.Event`).
B	Blocking	The only place allowed to block: `input_queue.get()` (no timeout) **when there's no work**. With work, drain only with `get_nowait()`, never blocking.
W	Workers	Busy loop: take messages, apply them to the engine in order, step, emit. IO threads only move data and never touch the engine. `request_shutdown` only sets the flag + enqueues WAKEUP; no emit, no join.
C	Channels	In: **one** FIFO `input_queue` (shared by ADD / ABORT / UTILITY / WAKEUP, so order is preserved naturally); out: `emit` (in the real system, output_queue → output IO thread). UTILITY results are correlated by the peer via `call_id` (→ the P05 approach).
L	Lifecycle	Start → run (block / drain / step loop) → two endings: shutdown received → `("shutdown",)`; step raises → return after `("fatal", ...)`. Mutually exclusive, exactly one per run.
I	Interface	The shape of the emitted tuples is the protocol and must be exact (`("shutdown",)` is a one-element tuple).
F	Flow + Invariants	① Before each step, all messages enqueued before that step began are processed; ② messages are applied to the engine in enqueue order; ③ engine has work ⇒ never block, engine has no work ⇒ never spin; ④ once the shutdown flag is seen, no more steps.
E	Edges + Errors	Utility exception → reply with an error for that call only; step / add exception → fatal; after processing messages the engine may still have no work (only WAKEUP / UTILITY / ABORT arrived) → don't step; the message returned by the blocking get is a normal message too and must be processed.

Interactive (Chinese): `nano-vllm-omni-drills/hints/p10_overview.html`

Source: `engine/stage_engine_core_proc.py:133-177` `engine/stage_engine_core_proc.py:147-154` `engine/stage_engine_core_proc.py:311-345` `vllm/v1/engine/core.py`

Notes / pitfalls I hit:

© 2026 kzhou003

#15 · P11 WAITING_FOR_CHUNK: park and restore ⏱ 45 min medium □ Done Date _____

> The upstream scheduler has no notion of a "waiting for chunk" state. You wrap it from the outside: a request without data must not exist as far as it can see.

Where it sits in the system

Under async_chunk, the talker's request enters the scheduler right after prewarm, but its input arrives from the thinker chunk by chunk. Until the next chunk arrives this request **must not be scheduled**: scheduling it would consume token budget and KV, while there is no input at forward time. vllm-omni's approach is to add a negative-valued status `WAITING_FOR_CHUNK` to `RequestStatus` and add hooks before and after each `schedule()`.

```
schedule():
    ① put ready parked requests back into their original queues
    ② remove from waiting/running requests that needs_chunk and are not ready, set status WAITING_FOR_CHUNK,
       record source queue and previous status; on first park call source.load_async(id)
    try:     ③ base.schedule()          # upstream only sees requests that have data
    finally: ④ restore ready ones again # restore even if upstream raises
```

Why "remove" rather than "skip"

We don't want to modify upstream scheduler code (vllm-omni invariant AR-INV-001: vLLM owns the base scheduling semantics). In this drill, `BaseScheduler` asserts if it sees a request in WAITING_FOR_CHUNK or ABORTED status, simulating upstream's intolerance of unknown states.

Other requirements

- Restore to the **source** queue and restore the previous status; multiple requests restored together are ordered by arrival.
- `abort(id)` can be called at any time: whether parked or in a queue, it must never be scheduled or restored afterwards.

File: `mine/p11_chunk_aware_scheduler.py` **Check:** `pytest tests/test_p11_chunk_aware_scheduler.py`
Hints: A-15 **Tests:** B-15 **Solution:** C-15

Interface skeleton

```
"""P11 · Park / restore around an upstream scheduler that knows nothing about chunks.

A request with needs_chunk=True receives its input from an upstream stage. Whenever its
next chunk is not ready (source.is_ready(id) is False) it must be invisible to
BaseScheduler.schedule():
- schedule(): (a) restore parked requests that became ready, (b) park every not-ready
  needs_chunk request from base.waiting / base.running (status "WAITING_FOR_CHUNK",
  remember origin queue and previous status, call source.load_async(id) once per parking),
  (c) call base.schedule() and (d) in a `finally`, restore any parked request that is ready
  -- this must also happen when base.schedule() raises.
- Restoring puts a request back into its ORIGIN queue with its previous status, in
  arrival order among the requests restored together.
- abort(id) works at any time: parked or queued, the request disappears and is never
  scheduled or restored again.
"""
from collections import deque
from dataclasses import dataclass
from typing import Protocol


# ---------------------------------------------------------------- given types
@dataclass
```


© 2026 kzhou003

```
class Req:
    req_id: str
    arrival: int
    needs_chunk: bool = False
    status: str = "WAITING"


class BaseScheduler:
    """Stand-in for the upstream vLLM scheduler: knows nothing about chunks."""

    def __init__(self) -> None:
        self.waiting: deque[Req] = deque()
        self.running: list[Req] = []

    def schedule(self) -> list[str]:
        for r in self.running + list(self.waiting):
            if r.status in ("WAITING_FOR_CHUNK", "ABORTED"):
                raise AssertionError(f"upstream saw {r.req_id} in state {r.status}")
        out = [r.req_id for r in self.running] + [r.req_id for r in self.waiting]
        for r in self.waiting:
            r.status = "RUNNING"
        self.running += list(self.waiting)
        self.waiting.clear()
        return out


class ChunkSource(Protocol):
    def load_async(self, req_id: str) -> None: ...
    def is_ready(self, req_id: str) -> bool: ...


# ------------------------------------------------------------ implement below
class ChunkAwareScheduler:
    def __init__(self, base: BaseScheduler, source: ChunkSource) -> None:
        self.base = base
        self.source = source
        self.parked: dict[str, tuple[Req, str, str]] = {}  # id -> (req, origin, previous status)

    def add(self, req: Req) -> None:
        self.base.waiting.append(req)

    def abort(self, req_id: str) -> None:
        raise NotImplementedError

    def schedule(self) -> list[str]:
        raise NotImplementedError
```

What the tests check

- ✓ Unready requests are never scheduled; load_async called only once
- ✓ Once ready, it is definitely scheduled on the next schedule
- ✓ Requests in running waiting for the next chunk are also parked
- ✓ finally still restores when upstream schedule raises
- ✓ abort while parked: never scheduled/restored again
- ✓ Multiple requests restored together keep arrival order

Follow-up questions

- Why doesn't vllm-omni call waiting_queue.prepend_requests in the middle of a step?
- Can the recv thread modify request.prompt_token_ids directly once it gets data?

© 2026 kzhou003

Intuition & analogies

Analogy · The kitchen's order board and the "awaiting ingredients" clip

The head chef (the **upstream scheduler**) only looks at the order board on the wall: a "to do" column (waiting) and an "in progress" column (running). He **cooks every ticket** on the board, and blows up (AssertionError) if he sees one marked "awaiting ingredients".

Some dishes need semi-finished ingredients from the kitchen next door (the **thinker**). You're the sous-chef (**ChunkAwareScheduler**): before each round, **take the "ingredients not here yet" tickets off the board** and clip them onto the **awaiting-ingredients clip** beside it (`parked`), writing two lines on the back: **which column it hung in** (origin) and **what its status was** (previous status); and call next door, "please send the ingredients for this ticket" (`load_async` , one call per removal).

When the ingredients arrive, **hang the ticket back in its original column** and restore its original status. When several tickets' ingredients arrive at once, hang them back in **the order customers placed them** (`Req.arrival`), not the order they were clipped.

Analogy · The chef throws a pan, a customer cancels

The chef throws a pan (`base.schedule()` raises): whatever happened this round, before clocking out the sous-chef must hang the clipped tickets whose "ingredients have arrived" back on the board (`finally`), and then report the incident truthfully (the exception keeps propagating; it must not be swallowed).

A customer cancels (`abort(id)`): right now the ticket may be in **three** places - the to-do column, the in-progress column, or the clip. Search all three and tear it up everywhere; tear up only the clipped one and the chef still cooks the one on the board; just write "cancelled" on the ticket without taking it down and the chef blows up again. Once it's torn up, even if next door sends the ingredients, there's no ticket to hang back.

Framework view · P11 through T · SBWC · LIFE

P11 is the **only fully single-threaded** problem among P11–P13: `add / abort / schedule` are all called on the same scheduler thread in the stage process. Its "concurrency" comes from just one thing - the answer of `is_ready()` is advanced by **another thread** (P13's recv thread), so you may get a different result every time you ask.

	Question	P11
T	Which processes / threads does it run in? Where are the boundaries?	The scheduler thread in the downstream stage process (EngineCore busy loop, see P10). Data comes from another process (thinker) via P12's shared memory and is received by this process's P13 recv thread; P11 crosses only the one "scheduler thread ↔ ChunkSource" boundary, and only with two **non-blocking** calls.
S	Shared state? Who writes?	`parked` , `base.waiting / base.running` , `req.status` : all written only by the scheduler thread (single writer, no locks). The upstream scheduler also mutates the queues, but only inside (c), serialized with you.
B	Where does it sit and wait?	**Nowhere is waiting allowed.** `schedule()` must never block waiting for a chunk (that would idle the whole GPU and stall every request in the batch) - hence "park + ask again next round", not "wait until it arrives".
W	Who does what?	No background worker of its own. The sous-chef (you) only takes tickets down and hangs them back; the "porter" delivering ingredients is P13's recv thread.
C	Channels?	`ChunkSource` is the channel: `load_async(id)` is the request, `is_ready(id)` is the answer.
L	Lifecycle?	A needs_chunk request goes back and forth between `WAITING/RUNNING` ⇄ `WAITING_FOR_CHUNK` many times: each chunk consumed may lead to another park. It ends in completion or `ABORTED` .
I	External interface?	`add` / `abort` / `schedule` ; `abort` may be called at any time.

© 2026 kzhou003

F	Invariants?	① While upstream (c) runs, the queues contain no `WAITING_FOR_CHUNK` / `ABORTED` requests; ② each request is in **at most one place** at any time: waiting, running, or parked; ③ each park makes exactly one `load_async`; ④ restore = original queue + original status, with a batch sorted by arrival.
E	Edges and failures?	Upstream raises (finally still restores, exception still propagates); a chunk arrives exactly during (c); the same request is parked first from waiting, later from running; `is_ready` still returns True after abort.

Interactive (Chinese): `nano-vllm-omni-drills/hints/p11_overview.html`

Source: `core/sched/omni_ar_scheduler.py:408-451`
`distributed/omni_connectors/transfer_adapter/chunk_transfer_adapter.py:1120-1186`
`distributed/omni_connectors/transfer_adapter/chunk_transfer_adapter.py:1488-1532` `patch.py:364-372`

Notes / pitfalls I hit:

© 2026 kzhou003

#16 · P07 Scheduler: continuous batching + chunked prefill + preemption ⏱ 75 min hard

□ Done Date _____

> The core of nano-vllm. Under three constraints (token budget, concurrency limit, KV block limit), decide each step who runs and how many tokens.

Where it sits in the system

The busy loop of each LLM stage process calls `schedule()` once per step, producing "how many tokens each sequence computes this step" for the model runner; after the runner finishes, `update()` writes the sampled results back.

Three constraints

- `max_num_batched_tokens`: cap on the total tokens of all sequences this step (determines activation memory and per-step latency);
- `max_num_seqs`: cap on the number of concurrently running sequences;
- KV blocks: a sequence holding n tokens (computed + to compute this step) needs `ceil(n / block_size)` blocks, with the total not exceeding `num_blocks`.

Scheduling rules

```
budget = max_num_batched_tokens
1) running (in order): n = min(remaining uncomputed tokens, budget)     # 1 in decode; more while chunked
prefill is unfinished
   not enough blocks → preempt the last one in running (most recently added): free blocks, num_computed=0, put
back at head of waiting
              → retry; if the preempted one is itself, skip it
2) waiting (strict FCFS): while len(running) < max_num_seqs and budget > 0, n = min(remaining, budget)
   stop at the first request whose blocks don't fit (no one behind may cut in line)
```

update

For each scheduled sequence, `num_computed += n`; only when it has computed all its tokens (the last prefill chunk or a decode) call `sample_fn` to append a token; on eos or reaching max_tokens, finish and free blocks. A preempted sequence next time recomputes prompt + generated tokens (recompute); outputs already generated stay unchanged.

Example (budget=16, three sequences with prompt length 10)

```
step1: s0:10  s1:6          # s1 did only part of its prefill
step2: s0:1   s1:4  s2:10   # s0 decode; s1 has 4 prefill left; s2 whole
step3: s0:1   s1:1  s2:1
```

File: `mine/p07_scheduler.py` **Check:** `pytest tests/test_p07_scheduler.py` **Hints:** A-16 **Tests:** B-16 **Solution:** C-16

Interface skeleton

```
"""P07 · Scheduler: continuous batching + chunked prefill + recompute preemption.

KV accounting is in blocks: a sequence holding n computed+scheduled tokens needs
ceil(n / block_size) blocks; self.blocks_used must never exceed num_blocks.

schedule() -> {seq_id: num_tokens_this_step}, total <= max_num_batched_tokens:
  1. RUNNING sequences first, in order: n = min(remaining tokens, budget)
     (1 for decode, more while a chunked prefill is still in progress). If the blocks do
```


© 2026 kzhou003

```
     not fit, preempt the LAST running sequence (free its blocks, num_computed = 0, put it
     at the FRONT of waiting, num_preemptions += 1) and retry; if the victim is the
     sequence itself, skip it.
  2. WAITING sequences strictly FCFS while len(running) < max_num_seqs and budget > 0:
     n = min(remaining, budget); stop at the first one whose blocks do not fit.
update(sample_fn, eos): for each scheduled seq add n to num_computed; if it now covers all
  tokens, append sample_fn(seq_id) to output; finish on eos or max_tokens (free blocks).
A preempted sequence recomputes prompt + already generated tokens; outputs are kept.
"""
from collections import deque
from collections.abc import Callable
from dataclasses import dataclass, field


@dataclass
class Seq:
    seq_id: str
    prompt_len: int
    max_tokens: int
    output: list[int] = field(default_factory=list)
    num_computed: int = 0
    num_blocks: int = 0
    status: str = "WAITING"  # WAITING | RUNNING | FINISHED

    @property
    def num_tokens(self) -> int:
        return self.prompt_len + len(self.output)


class Scheduler:
    def __init__(self, num_blocks: int, block_size: int, max_num_seqs: int, max_num_batched_tokens: int) -> None:
        self.num_blocks = num_blocks
        self.block_size = block_size
        self.max_num_seqs = max_num_seqs
        self.max_num_batched_tokens = max_num_batched_tokens
        self.waiting: deque[Seq] = deque()
        self.running: list[Seq] = []
        self.finished: list[Seq] = []
        self.blocks_used = 0
        self.num_preemptions = 0

    def add(self, seq: Seq) -> None:
        self.waiting.append(seq)

    def has_work(self) -> bool:
        return bool(self.waiting or self.running)

    def schedule(self) -> dict[str, int]:
        raise NotImplementedError

    def update(self, sample_fn: Callable[[str], int], eos_token: int) -> list[Seq]:
        raise NotImplementedError
```

What the tests check

- ✓ Exact schedule under token budget and chunked prefill
- ✓ max_num_seqs limit
- ✓ FCFS: when the head doesn't fit, smaller requests behind it can't cut in
- ✓ Preemption with tiny KV; all requests finish correctly with unchanged outputs
- ✓ EOS finishes early and frees blocks

Follow-up questions

- Trade-offs of recompute preemption vs swap preemption?
- How is encoder compute for multimodal requests counted in the budget?
- Under async scheduling num_computed includes placeholder tokens; how must update change?

© 2026 kzhou003

Intuition & analogies

> **Analogy · A restaurant's "serve per round" scheduling**
> **waiting queue** = guests lined up at the door (strictly in arrival order, no cutting in); **running** = tables already seated; **max_num_seqs** = how many tables in total;
> **KV block** = table surface: the longer guests eat (the more tokens generated), the more dishes pile up; the area only grows until they leave;
> **token budget** (`max_num_batched_tokens`) = how many dishes the kitchen can put out **per round**. If a big party of 10 (a long prompt) can't be served in one round, serve part of it first - that's **chunked prefill**;
> **preemption** = when there really isn't enough table space, ask the table that **sat down last** to go back to the **front** of the line and wait; what they've already eaten (generated tokens) stays eaten, but their dishes are cleared (KV is freed), and when they return everything has to be **set out again** (recompute).

> **Analogy · Preemption = sending the last one in back to the head of the line, not kicking them out of the theater**
> Preemption is **not** cancelling the request: it only throws away the **cache** (KV), not the **progress** (generated tokens).
> It's like being pulled out mid-movie for another security check: you **still remember the plot so far** (output is kept), but you lost your seat, so you must find a new one and **fast-forward replay** everything before (recompute all tokens of `prompt + generated`).
> So the user **can't tell at all** that preemption happened: the final output is token-for-token identical to a run without preemption, just a bit slower. This is exactly the property that's easiest to get wrong in this problem, and most worth verifying yourself.

Framework view: two phases · three budgets · six invariants

P07 is a single-threaded pure state machine (no concurrency, so T·SBWC·LIFE doesn't apply); the hard part is that **budget accounting** and **queue moves** must both be correct at the same time. Tabulate "where each budget is debited and when it's refunded", and the preemption branches won't get tangled:

Budget	Who holds it / how it's measured	Debit (when it decreases)	Refund (when it's added back)
token budget	A local variable, reset to `max_num_batched_tokens` on every `schedule()`	For each sequence successfully scheduled: `budget -= n` , where `n = min(num_tokens - num_computed, budget)`	No refund this step (the preempted sequence didn't run this step either). **Back to full next step**
running slots	`len(self.running)` , capped at `max_num_seqs`	Each admission in phase 2 appends to the **end** of running (end = lowest priority = first to preempt)	`running.remove` on preemption; `running.remove` on finish. Both free a slot for phase 2
KV block	Tracked in two places at once: `blocks_used` and each `seq.num_blocks`	`_try_grow(seq, num_computed + n)` : compute the increment `need` ; **update both places only if it fits**, otherwise change nothing and return False	Preemption: `blocks_used -= seq.num_blocks; seq.num_blocks = 0` ; finish: freed the same way

© 2026 kzhou003

Skeleton responsibilities of the two phases

Phase	What it iterates	n per sequence	What if it doesn't fit	Termination
Phase 1 running	`self.running`, by index with `while` (**not for**: preemption mutates this list)	`min(num_tokens - num_computed, budget)` : 1 for decode, the remaining chunk for an unfinished chunked prefill	Preempt `running[-1]`, then **retry the same sequence**; if the victim is itself → skip, index doesn't advance	Index reaches the end, or `budget == 0`
Phase 2 waiting	`self.waiting[0]` (head only, FCFS)	Same formula (for a previously preempted sequence this yields `prompt + generated`)	**break**; skipping it to schedule later ones is not allowed	`waiting` empty / `budget == 0` / `len(running) == max_num_seqs`

Invariants (write them as asserts and run every step)

	Invariant	Symptom when violated
I1	`blocks_used == sum(s.num_blocks for s in running)` and `<= num_blocks`; after all sequences finish, `blocks_used == 0`	Block leak: eventually nothing can be scheduled, or over-allocation causes OOM in the real system
I2	`sum(schedule().values()) <= max_num_batched_tokens`, and every `n >= 1`	The forward batch exceeds the budget → activation memory blows up; a sequence with `n == 0` wastes a row
I3	For any sequence: `0 <= num_computed <= num_tokens`; sampling is allowed only when `num_computed == num_tokens`	Sampling midway through chunked prefill → extra tokens, max_tokens reached early, output changes
I4	Preemption resets only `num_computed`, `num_blocks`, `status`; `output` **must be kept**	Final output differs from the no-preemption run (duplicated generation, or content drift) - the hardest class of bug to track down
I5	The preempted sequence goes to the **head** of `waiting`; `waiting` is always in arrival order	Preempted requests starve; under load request order scrambles and tail latency explodes
I6	Each sequence is in exactly one of `waiting`, `running`, `finished` at a time	The same request is scheduled twice (two rows in one forward), or still scheduled after finishing

Pitfall checklist

- **Computing blocks from** `num_tokens`: chunked prefill computed only part, but blocks are requested for the whole prompt → fills up early, needless preemption. Use `num_computed + n`.
- **Forgetting** `num_computed = 0` **after preemption**: the KV was freed with the blocks, yet only 1 token is scheduled → the model reads someone else's KV (and raises no error).
- **Using** `continue` **in phase 2 to skip a head that doesn't fit**: violates FCFS; large requests starve forever.
- **Iterating running with** `for`: `running.pop()` during preemption makes the iterator skip elements or visit an already-preempted sequence.
- **update doesn't know who was scheduled this step**: `update` takes no arguments, so the result of `schedule()` must be stored in an instance field (preferably returning a copy to the caller, so nobody else can edit your ledger).
- **Finish-check order**: append the sampled token to `output` first, then check `tok == eos or len(output) >= max_tokens` - eos itself is part of the output.

Interactive (Chinese): `nano-vllm-omni-drills/hints/p07_overview.html`

Source: `vllm/v1/core/sched/scheduler.py`

© 2026 kzhou003

Notes / pitfalls I hit:

© 2026 kzhou003

#17 · P06 BlockManager: paged KV + prefix caching ⏱ 75 min hard ☐ Done Date _____

> The memory management layer of PagedAttention: reference counting, hash-chain prefix matching, an LRU evictable pool, atomic allocation.

Where it sits in the system

The KV cache is split into fixed-size physical blocks (e.g. 16 tokens). Each sequence has a **block table**: logical block i → physical block number. The attention kernel finds a token's K/V with `slot = table[pos // bs] * bs + pos % bs`.

Prefix caching

Multi-turn conversations with the same system prompt or the same image have identical prefix KV. **Full** blocks are identified by a hash chain: `h_i = hash((h_{i-1}, tuple(tokens of block_i)))` (chained hashing ensures blocks with the same content but different prefixes don't falsely hit). On allocation, match block by block from the start; on a hit, share the physical block (refcount +1).

```
prompt: [0..9] (bs=4)  → blocks [0,1,2,3] [4,5,6,7] [8,9,_,_]
matchable blocks = (len − 1) // bs = 2     # keep at least 1 token to compute (otherwise no logits for the next
token)
num_cached_tokens = hit blocks × bs
```

Free and eviction

- On free, decrement refcounts in **reverse order**; blocks reaching zero: with a hash → go into the LRU evictable pool (still hittable for reuse); without a hash → back to the free list.
- When a new block is needed: prefer the free list, otherwise evict the oldest LRU block (also deleting its hash registration). Reverse-order freeing makes tail blocks get evicted first.
- `num_free_blocks` = free + evictable.

Atomicity

If allocate / append fails for lack of blocks, it must **change no state** (check before mutating). Note: hitting a block in the evictable pool also consumes an "available" slot.

append filling a block

When decode fills the last block, compute and register its hash too, so later requests can hit model-generated content (multi-turn conversations).

File: `mine/p06_block_manager.py` **Check:** `pytest tests/test_p06_block_manager.py` **Hints:** A-17
Tests: B-17 **Solution:** C-17

Interface skeleton

```
"""P06 · BlockManager: paged KV-cache allocator with ref counts and prefix caching.

- Physical blocks 0..num_blocks-1 of block_size token slots each.
- allocate(seq, tokens): all-or-nothing (raise OutOfBlocks without changing state).
  With prefix caching, FULL blocks are identified by a hash chain
  h_i = hash((h_{i-1}, tuple(block_i_tokens))). Reuse matching blocks from the start of the
  prompt, but only among the first (len(tokens)-1)//block_size blocks, so at least one
  token is always computed. num_cached_tokens(seq) = reused blocks * block_size.
- append_slot(seq, token): returns the slot (block*block_size + offset) of the new token;
  allocates a block when needed (OutOfBlocks leaves state unchanged); when a block becomes
  full it is registered in the prefix cache.
```


© 2026 kzhou003

```
- free(seq): decrement refs (walk the table in reverse). A block whose ref hits 0 goes
  back to the free list, or -- if it carries a hash -- to an LRU "evictable" pool that
  still serves cache hits until it is reused for a new allocation.
- num_free_blocks counts free + evictable blocks.
"""


class OutOfBlocks(Exception):
    pass


class BlockManager:
    def __init__(self, num_blocks: int, block_size: int, enable_prefix_cache: bool = False) -> None:
        self.num_blocks = num_blocks
        self.block_size = block_size
        self.enable_prefix_cache = enable_prefix_cache
        raise NotImplementedError

    @property
    def num_free_blocks(self) -> int:
        raise NotImplementedError

    def block_table(self, seq_id: str) -> list[int]:
        raise NotImplementedError

    def num_cached_tokens(self, seq_id: str) -> int:
        raise NotImplementedError

    def slot(self, seq_id: str, pos: int) -> int:
        raise NotImplementedError

    def can_allocate(self, num_tokens: int) -> bool:
        raise NotImplementedError

    def allocate(self, seq_id: str, token_ids: list[int]) -> list[int]:
        raise NotImplementedError

    def append_slot(self, seq_id: str, token_id: int) -> int:
        raise NotImplementedError

    def free(self, seq_id: str) -> None:
        raise NotImplementedError
```

What the tests check

- ✓ slot mapping
- ✓ append across block boundaries
- ✓ OutOfBlocks atomicity (failed allocate/append changes no state)
- ✓ free returns everything
- ✓ Prefix sharing and refcounts
- ✓ Cache still hits after free; the last token never hits
- ✓ No hit after eviction
- ✓ Blocks filled by append also enter the cache

Follow-up questions

- What must the block hash of a multimodal prompt include?
- When multiple stages share a GPU, how is each stage's num_blocks decided?

© 2026 kzhou003

Intuition & analogies

Analogy · Hotel rooms and the room-key list

A physical block = **a four-bed hotel room** (`block_size=4` beds = 4 token slots); `num_blocks` = how many rooms the hotel has in total, fixed at opening - you can't add a floor on the fly.

A request (sequence) = a tour group. The front desk gives it a **room-key list** (block table): the 1st batch of guests stays in room 2, the 2nd in room 7... Room numbers **don't need to be consecutive at all**, which is the whole point of paging - no reserving an entire floor because "a long sequence must stay in a contiguous suite".

`slot(seq, pos)` is "which bed guest number pos sleeps in": `room number × beds per room + bed number` .

Analogy · The library's "returns cart" (= the evictable LRU pool)

When a reader returns a book, it first goes on the **returns cart**, not yet reshelved. At this point:

① If another reader happens to want that book, they **can take it straight off the cart** (= a prefix-cache hit on a ref==0 block, "reviving" it);

② When the cart is full and space is needed, the librarian takes one from **the end where books were placed earliest**, wipes its contents, and rebinds it as a different book (= LRU eviction).

Key detail: the moment the contents are wiped, the **catalog card** (`hash → block`) must be torn up too. Otherwise the next reader follows the catalog to that spot and gets a book with **the right cover and completely wrong contents** - that's KV corruption: the model raises no error, it just quietly talks nonsense.

Analogy · Hash chain = the "full route" on a shipping label

If a block's label only says "this block contains [7, 8]", two sequences with completely different prefixes would wrongly think they can share it. But KV values depend on **all the context from the start up to here**, not just the few tokens in this block.

So the label must read "**previous block's label + this block's contents**" (`h_i = hash((h_{i-1}, this block's tokens))`), like a shipping label that nests the entire route. Only sequences whose "entire prefix is identical" compute the same `h_i` .

Framework view: three operations · one ledger · four invariants

P06 is a single-threaded pure data-structure problem with no concurrency (so T·SBWC·LIFE doesn't apply). All of its difficulty is in "**one ledger modified from different directions by three operations**". Write each operation as "what it reads → what it writes, in what order", and bugs basically have no room to get in:

Operation	Read (read-only, no side effects)	Write (order matters)	Where it may raise
`allocate`	① Hash chain of full blocks (computed only for the `len//bs` full blocks) ② Look up `hash→block` **contiguously from the start** within the first `(len-1)//bs` blocks, stopping at the first miss ③ Capacity: `new blocks needed <= num_free_blocks - number of hit blocks with ref==0`	④ Hit blocks: those with ref==0 are first removed from the evictable pool → ref+1 ⑤ Remaining blocks: take new blocks (free list first, otherwise evict the oldest + delete its hash entry) ⑥ Register hashes for full blocks ⑦ Save table / tokens / hash chain / hit token count	Only at ③ (plus the fallback in ⑤); **must come before any write**

© 2026 kzhou003

`append_slot`	`pos = len(tokens[seq])` ; is `pos // bs == len(table)` (need a new block?)	① Take the new block first (the only thing that can raise) ② Then append the token ③ If `(pos+1) % bs == 0` : this block just filled → use the chain tail as the parent hash, compute the new hash, append to chain, register	Only at ①; reversing the order leaves a dirty state with "one extra token"
`free`	That sequence's block table	Traverse in **reverse order**: ref−1; when it hits zero and `block_hash` is set → into the evictable pool (newest end), otherwise → free list; finally delete the sequence's four records	Never raises

Invariants (usable directly as asserts once written)

	Invariant	Symptom when violated
I1	`num_free_blocks == len(free list) + len(evictable pool)` , the free list and the evictable pool are **disjoint**, and every block in either has `ref == 0`	The same block is allocated to two sequences, and their contents overwrite each other
I2	`b in evictable pool ⟺ ref[b] == 0 and block_hash[b] is not None`	Forgot to remove from the pool after a hit → a block in use gets evicted as a candidate
I3	`hash_to_block[h] == b ⟹ block_hash[b] == h` (the mapping and the block's label always point at each other)	Mapping not deleted after eviction → an old prefix "hits" a block that has been overwritten
I4	`ref[b] ==` number of block tables containing b; after every sequence is freed, `num_free_blocks == num_blocks`	Block leak: OutOfBlocks after running for a while, fixed only by a restart
I5	`num_cached_tokens(seq) % block_size == 0` and `<` `len(token_ids)` (leave at least one token to compute)	On a full hit the model has no token to forward, gets no logits for the last position, and can't sample the next token

Pitfall checklist

- **Hashing the half-full last block too**: a later, longer prompt would hit a block holding only some of its tokens. Only full blocks have an identity.
- **Writing the matchable limit as** `len // bs` : when the prompt is an exact multiple of bs and fully hits, `num_cached_tokens == len(token_ids)` , violating I5.
- **Inconsistent hash-chain start**: `allocate` uses `None` as `h_{-1}` while `append_slot` uses a different sentinel, so blocks filled during decode are never hit by allocate (the cache becomes "write-only").
- **Unconditional** `del hash_to_block[h]` **on eviction**: when two blocks compute the same hash (same contents) and the mapping points to the **other** block, you delete someone else's valid mapping. First confirm the mapping still points to this block.
- **free in forward order**: the tail block (the least shared prefix) becomes "newest" and is evicted last, while the most shared head block dies first, lowering the hit rate.
- **Conflating the checks in** `can_allocate` **and** `allocate` : the former is coarse (ignores hits, conservative), the latter must be exact (see the atomicity trap above).

Interactive (Chinese): `nano-vllm-omni-drills/hints/p06_overview.html`

Source: `vllm/v1/core/kv_cache_manager.py` `vllm/v1/core/block_pool.py`

© 2026 kzhou003

Notes / pitfalls I hit:

© 2026 kzhou003

#18 · P08 Batch-invariant sampler (per-request seed)

45 min | medium | □ Done | Date _____

> The talker is the only stochastic stage; vllm-omni fixes its seed for reproducibility. But if the result for a given seed changes depending on who else is in the batch, the seed is meaningless.

Where it sits in the system

The last step of each LLM stage's model runner: logits → token. In the Qwen3-Omni deploy config the thinker has temperature 0 (greedy) and the talker `temperature 0.9, top_k 50, repetition_penalty 1.05, seed 42`; the comment explains that without a fixed seed a few samples "fluently talk nonsense", and only with a fixed seed is it reproducible.

Processing order (each row independent)

```
1. repetition penalty: for each token that has appeared, divide positive logits by p, multiply negative logits
by p
2. temperature < 1e-5 → argmax, done
3. logits /= T
4. top_k: keep only the k largest
5. top_p: after sorting descending, keep the smallest prefix whose cumulative probability reaches top_p (at
least 1)
6. draw one token with this request's own torch.Generator
```

Batch invariance

If all rows share one generator, request a's random numbers depend on how many rows precede it in the batch. Batch composition differs every step, so the same seed yields different audio. Therefore every seeded request owns its own generator (created on first use, reused across steps, `free`d when the request finishes).

File: `mine/p08_sampler.py` **Check:** `pytest tests/test_p08_sampler.py` **Hints:** A-18 **Tests:** B-18 **Solution:** C-18

Interface skeleton

```
"""P08 · Batch-invariant sampler with per-request seeds.

- apply_repetition_penalty: for every distinct previous token, positive logits are
  divided by the penalty and negative ones multiplied (do not mutate the input).
- sample(): per row -> repetition penalty -> temperature < 1e-5 means greedy argmax;
  otherwise divide by temperature, apply top_k (keep k largest) then top_p (keep the
  smallest prefix of the sorted distribution whose mass reaches top_p; always keep >= 1),
  then draw ONE token.
- Seeded requests own a torch.Generator created on first use from params.seed; the token
  drawn for a request must not depend on which other rows share the batch, or their order.
- free(req_id) forgets the generator (a new request with the same id restarts the stream).
"""
from dataclasses import dataclass

import torch


@dataclass
class SamplingParams:
    temperature: float = 1.0
    top_k: int = -1
    top_p: float = 1.0
    repetition_penalty: float = 1.0
    seed: int | None = None


def apply_repetition_penalty(logits: torch.Tensor, prev_tokens: list[int], penalty: float) -> torch.Tensor:
    raise NotImplementedError
```


© 2026 kzhou003

```
class Sampler:
    def __init__(self) -> None:
        raise NotImplementedError

    def free(self, req_id: str) -> None:
        raise NotImplementedError

    def sample(
        self,
        logits: torch.Tensor,  # [B, V]
        req_ids: list[str],
        params: list[SamplingParams],
        prev_tokens: list[list[int]],
    ) -> torch.Tensor:  # [B] long
        raise NotImplementedError
```

What the tests check

- ✓ greedy
- ✓ top_k=1 / tiny top_p are equivalent to greedy
- ✓ Requests with the same seed give identical results under different batch compositions/orders
- ✓ Empirical distribution with top_k=2 ≈ 0.7/0.3
- ✓ Repetition penalty values and effect
- ✓ Random stream restarts after free

Follow-up questions

- Vectorized version: use per-row generators to produce exponential noise for an exponential race.
- Why can't a batch with per-row generators replay a CUDA graph directly? (→ P17)

Intuition & analogies

Analogy · A talent show: deduct points → widen the gaps → shortlist → draw lots

A row of logits is the **judges' scores** for V contestants; sampling = picking one to perform. Four steps:
① **repetition penalty**: deduct points from singers who just performed (down-weight tokens that already appeared);
② **temperature**: T<1 **widens** the gaps (the strong get stronger, output is steadier), T>1 **flattens** them (more creative, and more prone to nonsense), T→0 just picks first place (greedy);
③ **top_k**: keep only the k highest scorers;
④ **top_p**: accumulate "vote share" from high to low, and close the list once it reaches p (a dynamic k);
Finally **draw lots by probability** from the shortlist - that draw is the only place a random number is used.

Analogy · A shared lottery machine vs everyone bringing their own die

Shared lottery machine (one generator for the whole batch): every draw by someone ahead of you advances the machine one notch. Third in line today, first tomorrow, and naturally you draw different numbers - even if "the machine's initial seed" is the same every day.
Bring your own die (per-request generator): only you roll your die, and how many times it's been rolled depends only on **how many times you've been scheduled**, completely independent of who else is in the batch or who's ahead of you.
Two more things: **keep the die with you** (reuse it across steps; don't grab a new one each step), and **hand it back** when done (`free` , or the next request with the same name continues your random stream).

© 2026 kzhou003

Framework view: shape trace and pitfalls

P08 is a pure-function problem (no concurrency); correctness rests entirely on three things: "**shape, order, side effects**". Write out the data flow for one row first, then wrap it in the loop over B rows:

Step	Tensor / shape	What it does	Pitfalls
0	`logits [B, V]`, dtype may be bf16/fp16	Take each row `logits[i]` and `.float()` it	Doing softmax / cumsum directly in bf16, which drifts from the reference by up to 1e-2
1	`row [V]` fp32	`apply_repetition_penalty(row, prev_tokens[i], p)`: for the **deduplicated** history tokens, `v > 0 ? v/p : v*p`	① Penalizing repeated tokens multiple times; ② dividing negative logits too (making them larger = a reward); ③ mutating the caller's tensor in place
2	`row [V]`	`if temperature < 1e-5: return argmax(row)`; otherwise `row = row / temperature`	Dividing before checking → `/0` gives inf/NaN; checking greedy before the penalty → the penalty has no effect on greedy rows
3	`row [V]`, partly set to `-inf`	Only when `0 < top_k < V`: take the k-th largest value `kth` and set `row < kth` to `-inf`	Not treating `top_k = -1` or `>= V` as "disabled"; ties at k-th place keep a few extra (acceptable, but know it)
4	Sorted view `sorted [V]`, `order [V]`	Sort descending → softmax → cumsum; **"mass accumulated before it" =** `cum - probs`; positions where that already exceeds top_p are set to `-inf`; always keep position 0	Also removing "the token that crosses the threshold" → with a tiny top_p everything is removed → softmax gives NaN; forgetting to map back to original positions with `order`
5	`probs [V]` → scalar	`torch.multinomial(probs, 1, generator=self._gen(req_id, p))`	One generator shared by the whole batch (batch invariance lost); rebuilding the generator each step (skewed distribution); forgetting `free`
6	`out [B]` `torch.long`	Write back to position i and return	Returning float / shape `[B,1]`; row order not matching `req_ids`

Invariants (can be written directly as a self-check script)

	Invariant	How to check
I1	**Batch invariance**: for a fixed-seed request, the token sequence over N consecutive steps is identical to when it "runs as a batch of one"	Put the same request into randomly shuffled batches for N steps and compare with the list from running it alone for N steps (see interactive figure B)
I2	**No side effects**: after `sample()` returns, the passed-in `logits` are bitwise unchanged	`before = logits.clone()`, then after the call `assert torch.equal(before, logits)`
I3	`temperature < 1e-5`, `top_k == 1`, and `top_p → 0` must all degenerate to **argmax** (that single token)	Run each of the three settings once; all results should equal the argmax of the penalized logits (token 0 in this page's example)

© 2026 kzhou003

I4	The kept set is non-empty: under any parameter combination at least 1 token has probability > 0 (no NaN)	Run with extremes like `top_p=1e-9`, `top_k=1`, and assert `probs.sum() > 0` with no NaN
I5	**Continuous random stream**: a request's n-th sample uses the n-th random number from its generator; reset after `free`	Draw 4 times → `free` → draw 4 more; the two lists must match (this page's example: `[0, 2, 0, 2]` both times)
I6	Empirical distribution ≈ filtered probabilities	Sample the same row a few thousand times and count frequencies (this page's example, 4000 draws: `0.666 / 0.189 / 0.146`, vs theoretical `0.6624 / 0.1898 / 0.1478`)

Interactive (Chinese): `nano-vllm-omni-drills/hints/p08_overview.html`

Source: `deploy/qwen3_omni_moe.yaml:49-62` `worker/gpu_model_runner.py:1998-2010`

Notes / pitfalls I hit:

© 2026 kzhou003

#19 · P09 Paged KV write + decode attention (reference implementation)

45 min | medium

□ Done Date _____

> First write a correct reference implementation in PyTorch; it becomes the parity baseline for a later Triton kernel.

Where it sits in the system

P06 decides which physical slot each token's K/V lives in. Each step the model runner first writes the new tokens' K/V into the cache according to `slot_mapping`, then attention reads history from the cache according to `block_tables`.

```
cache layout:   [num_blocks, block_size, num_kv_heads, head_dim]
write:          cache[slot // bs, slot % bs] = k[i]          (slot = -1 is padding, don't write)
decode read:    the first context_lens[b] tokens of sequence b:
                k_b = cache[block_tables[b, :ceil(n/bs)]].reshape(-1, KVH, D)[:n]
                out_b = softmax(q_b · k_bᵀ · scale) · v_b   (GQA: head h uses kv head h // (H/KVH))
```

Pitfalls

- Physical blocks are non-contiguous (the test shuffles allocation); unused entries at the tail of the block table are garbage and must not be read.
- In CUDA graph mode the batch is padded, and padded tokens have slot -1.
- Use fp32 for softmax.

File: `mine/p09_paged_attention.py` **Check:** `pytest tests/test_p09_paged_attention.py` **Hints:** A-19
Tests: B-19 **Solution:** C-19

Interface skeleton

```
"""P09 · Paged KV write + decode attention (PyTorch reference; Triton is the follow-up).

Cache layout: [num_blocks, block_size, num_kv_heads, head_dim].
- write_kv: scatter token i's K/V into slot slot_mapping[i] = block*block_size + offset;
  slot -1 marks a padding token that must NOT be written.
- paged_decode_attention: one query token per sequence. Sequence b attends to its first
  context_lens[b] cached tokens, found through block_tables[b] (entries past the last
  used block are garbage). Support GQA: num_heads is a multiple of num_kv_heads and query
  head h uses kv head h // (num_heads // num_kv_heads). Softmax in fp32.
"""
import torch


def write_kv(
    k_cache: torch.Tensor,
    v_cache: torch.Tensor,
    k: torch.Tensor,  # [num_tokens, num_kv_heads, head_dim]
    v: torch.Tensor,
    slot_mapping: torch.Tensor,  # [num_tokens]
) -> None:
    raise NotImplementedError


def paged_decode_attention(
    q: torch.Tensor,  # [B, num_heads, head_dim]
    k_cache: torch.Tensor,
    v_cache: torch.Tensor,
    block_tables: torch.Tensor,  # [B, max_blocks]
    context_lens: torch.Tensor,  # [B]
    scale: float,
```


© 2026 kzhou003

```
) -> torch.Tensor:  # [B, num_heads, head_dim]
    raise NotImplementedError
```

What the tests check

- ✓ Numerically matches dense attention (GQA, non-contiguous physical blocks, garbage at block table tail)
- ✓ Padding tokens with slot -1 are not written
- ✓ Random shapes and block_size

Follow-up questions

- Triton version: one program per (seq, head), loop over blocks doing online softmax.
- How do you write the varlen version for prefill?

Intuition & analogies

Analogy · A wall of lockers: the slot is the locker number

The KV cache is a **wall of lockers**: `num_blocks` groups in total, `block_size` lockers per group. `slot = group number × lockers per group + locker number`, so "the slot-th locker after flattening" and `cache[slot // bs, slot % bs]` are **the same location**.

A sequence doesn't get a contiguous stretch of wall; it gets a **list of pickup codes** (the block table): batch 1 goes in group 3, batch 2 in group 1... Out-of-order, non-consecutive group numbers are perfectly fine - that's the whole point of paging.

The most dangerous case is `slot = -1` (padding tokens added when CUDA graphs pad the batch). In Python/PyTorch `-1` **means "the last locker"**, so this token that should have been dropped **silently overwrites someone else's package in the last locker** - no error, no warning.

Analogy · A library call slip

The **block table** is a pre-printed call slip (a fixed `max_blocks` rows); you filled in only the first two rows, and **the remaining rows are garbage left by the previous reader**. Fetch books following the garbage and at best you get someone else's book (results silently wrong), at worst a shelf number that doesn't exist (IndexError).

Read only the first few pages of the last volume: in volume 2 you only use up to page 2, and the next two pages are someone else's content. The hope that "those two pages are blank anyway" doesn't hold here - a blank page has score = 0, which still takes a share of weight in softmax (`e^0 = 1`); only **truncation** (or setting `-inf`) truly excludes it.

Framework view: shape trace and pitfalls

This is a pure-function problem (no concurrency). Before writing, jot down the **shapes** of both data flows line by line; almost every bug shows up on paper:

`write_kv` (can be vectorized over the whole batch)

Step	Tensor / shape	What it does	Pitfalls
1	`slot_mapping [T]`	`keep = slot_mapping >= 0`, giving a boolean mask	Not filtering `-1`: padding tokens overwrite **the last slot**, with no error

© 2026 kzhou003

2	`s = slot_mapping[keep] [T']`	`blk = s // bs`, `off = s % bs`	Forgetting to cast to long when using `divmod`; int32 indices aren't accepted on some backends
3	`k[keep] [T', KVH, D]`	`k_cache[blk, off] = k[keep]` (same for v)	Both advanced indices must be given **together** (`[blk, off]`); writing `k_cache[blk] [off]` copies first and then writes, so the original cache isn't modified
4	—	Return `None` (in place)	Returning a new tensor → the caller never sees the writes

`paged_decode_attention` (a per-sequence loop is fine; get it correct first)

Step	Tensor / shape	What it does	Pitfalls
1	`n = int(context_lens[b])`	`nb = ceil(n / bs)`	Using `n // bs` → the last, partially filled block is dropped entirely
2	`tbl = block_tables[b, :nb].long() [nb]`	Take only the valid entries	Taking the whole row → reading garbage: IndexError, or silently reading someone else's KV
3	`k_cache[tbl] [nb, bs, KVH, D]`	Gather in table order; **the logical order comes out right automatically**	Thinking the table must be sorted first - no, the table's order **is** the logical order
4	`.reshape(nb*bs, KVH, D)[:n]` → `[n, KVH, D]`	Flatten into a "token dim", then **truncate** the extra positions of the last block	No truncation → a few extra positions are attended, results silently wrong (step 3 of section 5 on this page has real numbers comparing them)
5	`.repeat_interleave(H // KVH, dim=1)` → `[n, H, D]`	GQA: map head `h` to kv head `h // group`	Using `repeat` → becomes `h % KVH`; identical when `KVH=1`, so it's hard to notice
6	`scores [H, n]` fp32	`einsum("hd,nhd->hn", q[b].float(), kb) * scale`, then softmax over the `n` dim	Applying scale after softmax (wrong distribution); doing softmax in bf16 (1e-2 off from the reference)
7	`out[b] [H, D]`	`einsum("hn,nhd->hd", probs, vb).to(q.dtype)`	Forgetting to cast back to `q.dtype`; mixing up the head and token dims (shape right, values wrong)

Invariants / self-check list

	Invariant	How to check
I1	Values for a `-1` slot **appear nowhere in the cache**	Write a padding token whose elements are all a sentinel (e.g. 9.0), then `assert (cache == 9).sum() == 0`
I2	Only `block_tables[b, :ceil(n/bs)]` is accessed	Fill the tail of the block table with `99999` (beyond num_blocks); a correct implementation should not error
I3	The number of K/V rows in attention is exactly `== context_lens[b]`	Assert `kb.shape[0] == n`; and test specifically with a length where "`n` is not a multiple of bs"
I4	`probs.sum(-1) == 1`, and values match dense attention	Write your own dense version (compute directly on contiguous K/V) and diff against it: random lengths (including 1, multiples of bs, non-multiples), shuffled physical blocks, garbage in the tail. On this page's random test set the max absolute error is **0.0**
I5	Correct GQA mapping: head `h` uses kv head `h // group`	With `H=4, KVH=2`, `repeat_interleave` gives `[0,0,1,1]` and `repeat` gives `[0,1,0,1]` - it must be the former

Interactive (Chinese): `nano-vllm-omni-drills/hints/p09_overview.html`

© 2026 kzhou003

Source: `vllm/v1/attention/backends/triton_attn.py`

Notes / pitfalls I hit:

© 2026 kzhou003

Track C · Data plane

Cross-process payloads: shared-memory protocol, background send/recv threads, ordering and cleanup.

© 2026 kzhou003

#20 · P12 SharedMemoryConnector: cross-process put/get protocol ⏱ 60 min hard

□ Done Date _____

> The thinker's hidden states must go to the talker process for every token. Using POSIX shared memory with no extra metadata channel, both sides derive the name from the key alone and use a file lock to express "done writing".

Where it sits in the system

Bulk data between stages (hidden states, embeddings, codec) doesn't go through the orchestrator; workers write it directly into shared memory and a background thread of the downstream stage reads it. For single-host deployments SharedMemoryConnector is the default.

Protocol

```
key e.g. "req42_0_7" (request 42, chunk 7 sent by stage 0)
name = short_hash(key)                 # macOS POSIX shm names are at most 31 bytes
lock = <lock_dir>/<name>.lock

put(key, data):                          get(key):
  open(lock, O_CREAT); flock(EX)           open(lock)            → missing: None
  create shm (unlink stale segment first)  flock(EX | NB)        → BlockingIOError: still writing, None
  write 8-byte length + data               open shm              → missing: None
  flock(UN)                                read → close → unlink (reader frees) → delete lock file
```

Why the 8-byte length header: the shm segment size is rounded up to a page, so the reader can't learn the true length from the segment size.

The resource tracker trap (a real pitfall; the test reproduces it)

In Python < 3.13, `SharedMemory(create=True)` registers the segment with the **creator's** resource tracker. If the writer is a separate interpreter and exits first, its tracker treats the still-registered segment as "leaked" and deletes it; when the reader comes to read, the data is gone. Fix: after writing, transfer ownership to the reader (unregister from the tracker) and let the reader unlink.

Producer cleanup

When a request is aborted, the downstream may never come to read. The producer must remember the keys it wrote, and `cleanup_prefix("req42_")` deletes the segments among them not yet consumed.

File: `mine/p12_shm_connector.py` **Check:** `pytest tests/test_p12_shm_connector.py` **Hints:** A-20
Tests: B-20 **Solution:** C-20

Interface skeleton

```
"""P12 · ShmConnector: byte payloads through named POSIX shared memory (one consumer/key).

Protocol (no metadata channel — both sides derive names from the key):
- shm_name(key): a short, deterministic name (macOS caps POSIX shm names at 31 bytes).
- put(key, data): take an EXCLUSIVE flock on lock_path(key) (create the file), create the
  segment (replace a stale one), write an 8-byte length header + data, then unlock.
  Remember the key so cleanup_prefix() can find it.
- get(key): non-blocking. None if the lock file is missing, the lock is held by someone
  else (LOCK_NB -> BlockingIOError = "still being written"), or the segment is missing.
  Otherwise read, close, UNLINK the segment and remove the lock file (the reader frees it).
- cleanup_prefix(prefix): producer side; unlink every key it wrote with that prefix that
  was never consumed; return how many it removed.
- The payload must survive the WRITER PROCESS EXITING before the reader reads.
  (Hint: look at multiprocessing.resource_tracker.)
```


© 2026 kzhou003

```
"""


class ShmConnector:
    def __init__(self, lock_dir: str | None = None, namespace: str = "nvo") -> None:
        raise NotImplementedError

    def shm_name(self, key: str) -> str:
        raise NotImplementedError

    def lock_path(self, key: str) -> str:
        raise NotImplementedError

    def put(self, key: str, data: bytes) -> None:
        raise NotImplementedError

    def get(self, key: str) -> bytes | None:
        raise NotImplementedError

    def cleanup_prefix(self, prefix: str) -> int:
        raise NotImplementedError
```

What the tests check

- ✓ Round trip, consumable only once
- ✓ Empty payload and 4MB payload
- ✓ Reader can still read 1 second after a separate writer process exits (resource tracker trap)
- ✓ get returns None while the lock is held by someone else
- ✓ cleanup_prefix cleans only unconsumed keys with that prefix

Follow-up questions

- For same-GPU cases, zero-copy with CUDA IPC: how do you guarantee lifetime and cross-process stream synchronization?
- How can the producer reclaim its own bookkeeping without waiting for an ACK?

Intuition & analogies

Analogy · The apartment complex's parcel lockers

A shared-memory segment is a **parcel locker compartment**, and the key is the pickup code. The courier (the **writer**) and the recipient (the **reader**) have never met, but they use the same table to compute **the same compartment number** from the pickup code (`shm_name(key)`) - so nobody has to pass along "it's in locker number such-and-such".

Each compartment door has a **lock** (lock file + `flock`): the courier **locks the door first, then puts things in**, and unlocks only when done. When the recipient arrives, they first try to lock the door: if it won't lock (`LOCK_NB` → `BlockingIOError`), the courier is still loading, so they **turn around and leave** and come back later, never standing at the door waiting.

Every compartment is the standard large size (one page), even for a single pen. So the courier sticks a **note** at the very front of the compartment: "actual contents: N bytes" (the 8-byte length header).

After taking the item, the recipient **releases the compartment themselves** (`unlink`) and removes the lock from the door (deletes the lock file) - only they know "it's been picked up".

© 2026 kzhou003

Analogy · Building management's "move-out cleaner" (the resource tracker trap)

Every Python process has a building cleaner (the `resource_tracker` child process). When the courier **rents** a compartment (`SharedMemory(create=True)`), management registers it **in the courier's name**.

When the courier clocks out (**the writer process exits**), the cleaner checks: compartments registered in his name still not released? "He must have forgotten" - so it **empties and releases** them, and grumbles `leaked shared_memory objects` in the log. But the recipient hasn't picked up that parcel yet!

The fix: after loading the parcel, the courier goes to management and **cancels the registration** (`resource_tracker.unregister`), effectively **transferring** ownership of the compartment to the recipient; the recipient releases it after pickup.

Framework view · P12 through T · SBWC · LIFE

P12's T is completely different from the earlier problems: the participants are **processes**, not threads or tasks. Processes don't share Python objects, so `threading.Lock` is meaningless; all you have is what the kernel provides - shared-memory segments, files, `flock` - plus a **protocol** both sides agree on in advance (how names are computed, who creates, who releases).

	Question	P12
T	Processes / threads / boundaries?	The writer process (stage s's save thread), the reader process (stage s+1's recv thread), plus an easily forgotten third party: **each process's own resource_tracker child process**. The only cross-process sync primitive is `flock` (an advisory lock, effective only among participants that also use flock).
S	Shared state? Who writes?	Three things in the kernel: the segment (created by the writer, unlinked by the reader), the lock file (writer `O_CREAT`s, reader deletes; its **existence** is itself state: "there's an unconsumed key"), and the flock holder. Process-local: the writer's "keys I've written" set (used only by `cleanup_prefix`), and each process's tracker registry.
B	Where does it wait?	The writer's `flock(LOCK_EX)` may block (at most until the reader finishes one chunk, microseconds); the reader **never blocks**: `LOCK_EX | LOCK_NB` , returning None if it can't get the lock, and P13's recv thread retries later.
W	Who does what?	Writer: create + write + hand over ownership. Reader: read + copy + release. Tracker: on process exit, cleans up segments still registered - a worker **you didn't write but that will definitely run**.
C	Channels?	The "name" is the channel: `shm_name(key)` / `lock_path(key)` are computed independently on both sides. The only in-band metadata is the 8-byte length header.
L	Lifecycle?	Segment: writer create → write → close (**not** unlink) → unregister from tracker → reader attach → copy → close → **unlink**. Lock file: created by the writer → deleted by the reader. Aborted requests: the writer reclaims unread segments with `cleanup_prefix` .
I	External interface?	`put` / `get` / `cleanup_prefix` (+ two pure functions `shm_name` / `lock_path`). `get` is non-blocking and consuming.
F	Invariants?	① When the reader can lock the door, the segment either doesn't exist or is fully written (a half-written one is never read); ② each segment is unlinked exactly once; ③ lock file exists ⇔ this key was written and not yet consumed; ④ every return path of `get` unlocks and closes the fd.
E	Edges and failures?	A stale segment with the same name (`FileExistsError` → replace); lock file present but segment missing; the writer exits first; empty payload and a 4 MB payload (both measured to round-trip); the prefix `"spk3_"` must not hit `"spk30_…"` .

Interactive (Chinese): `nano-vllm-omni-drills/hints/p12_overview.html`

Source: `distributed/omni_connectors/connectors/shm_connector.py:195-364` `entrypoints/stage_utils.py:193-237`

© 2026 kzhou003

Notes / pitfalls I hit:

© 2026 kzhou003

#21 · P13 ChunkAdapter: background send/recv threads with ordering/cleanup semantics

75 min | hard | □ Done Date _____

> The heart of async_chunk: the scheduler thread only snapshots and enqueues, the save thread builds the payload and puts, the recv thread polls get. You must also handle cancel, None payloads, and "a chunk is being read exactly when cleanup happens".

Where it sits in the system

Each stage process's scheduler has a ChunkAdapter: a layer between the connector (transport only, P12) and the scheduler (P11), responsible for the full lifecycle of chunks.

```
sender (stage 0)                                    receiver (stage 1)
scheduler thread: save_async(req, snapshot, final)  scheduler thread: load_async(req) / poll_ready()
     | only deepcopy + enqueue (never blocks scheduling)   ▲ committed chunks, in order
save thread: build_payload(snapshot)                recv thread: get(f"{req}_{src}_{next_id}")
           put(f"{req}_{stage}_{chunk_id}")                    got it → check "still registered?" under lock →
commit
           chunk_id += 1 on success
```

The bug behind each rule

Rule	What happens if you break it
save_async deep-copies the snapshot	The scheduler keeps mutating the request object next step, so the background thread reads "future" data
build_payload and put both outside the lock, on the save thread	Slow Python reshaping and SHM writes block scheduling, and the GPU idles
chunk_id increments only after a successful put; a None payload takes no id	The receiver polls consecutive ids, so a hole leaves it stuck forever
cancel must not swallow the final chunk	The final chunk is the downstream's only EOS; lose it and the downstream request stays in WAITING_FOR_CHUNK forever
get outside the lock; before commit, check under the lock that the registration token is still the same	The request was just cleaned up, and the background thread writes a late chunk back, polluting the next request with the same id
Back off ~1ms when there's no progress; sleep on a Condition when idle	busy-poll pins a core and fights tokenize for the GIL

File: `mine/p13_chunk_adapter.py` **Check:** `pytest tests/test_p13_chunk_adapter.py` **Hints:** A-21
Tests: B-21 **Solution:** C-21

Interface skeleton

```
"""P13 · ChunkAdapter: the async-chunk data plane of one stage (sender + receiver halves).

Sender (scheduler thread calls save_async; a background SAVE thread does the work)
- save_async(req, snapshot, is_final) returns immediately; it must snapshot (deep-copy)
  its argument because the caller keeps mutating the live request.
- The save thread calls build_payload(snapshot) OUTSIDE any lock; None means "nothing to
  send this step" (skip, unless is_final). It puts {"data": ..., "finished": is_final} at
  key f"{req}_{stage_id}_{chunk_id}"; chunk_id starts at 0 and advances only after a
  successful put, so ids stay contiguous.
- cancel_sender(req): later non-final chunks are dropped, but the final chunk is still
```


© 2026 kzhou003

```
  sent (with data=None) because it is the downstream end-of-stream marker.
Receiver (background RECV thread polls the connector)
- load_async(req): start polling f"{req}_{src_stage_id}_{next_id}"; the get happens
  OUTSIDE your locks; back off (~1 ms) when nothing arrived, sleep on a Condition when idle.
- poll_ready() (scheduler thread): {req: [chunk dicts in order]} committed since last call.
- cleanup_receiver(req): from now on nothing for req appears in poll_ready -- including a
  chunk the recv thread is reading at this very moment (re-check a per-registration token
  under the lock before committing).
- start()/stop() manage both threads; stop() must return promptly.
"""
import copy
import threading
from collections import deque
from collections.abc import Callable
from typing import Any


# ---------------------------------------------------------------- given types
class InMemoryConnector:
    """Transport only: put/get by key, get consumes. Thread-safe. Hooks let tests add latency."""

    def __init__(self, on_put: Callable[[str], None] | None = None, on_get: Callable[[str], None] | None = None):
        self._store: dict[str, Any] = {}
        self._lock = threading.Lock()
        self.on_put, self.on_get = on_put, on_get
        self.puts: list[str] = []

    def put(self, key: str, obj: Any) -> None:
        if self.on_put:
            self.on_put(key)
        with self._lock:
            self._store[key] = obj
            self.puts.append(key)

    def get(self, key: str) -> Any | None:
        with self._lock:
            present = key in self._store
        if not present:
            return None
        if self.on_get:
            self.on_get(key)
        with self._lock:
            return self._store.pop(key, None)


# ---------------------------------------------------------------- implement below
class ChunkAdapter:
    def __init__(
        self,
        connector: InMemoryConnector,
        stage_id: int,
        src_stage_id: int | None,
        build_payload: Callable[[Any], Any | None],
    ) -> None:
        self.connector = connector
        self.stage_id = stage_id
        self.src_stage_id = src_stage_id
        self.build_payload = build_payload
        raise NotImplementedError

    def start(self) -> None:
        raise NotImplementedError

    def stop(self) -> None:
        raise NotImplementedError

    def save_async(self, req_id: str, snapshot: Any, is_final: bool) -> None:
        raise NotImplementedError

    def cancel_sender(self, req_id: str) -> None:
        raise NotImplementedError

    def load_async(self, req_id: str) -> None:
        raise NotImplementedError

    def poll_ready(self) -> dict[str, list]:
        raise NotImplementedError
```


© 2026 kzhou003

```
    def cleanup_receiver(self, req_id: str) -> None:
        raise NotImplementedError
```

What the tests check

- ✓ Single request, 50 chunks in order
- ✓ 10 requests × 30 chunks with random delays, each in order
- ✓ save_async never blocks the caller (when build_payload is slow)
- ✓ Snapshot isolation: mutating the original object after the call doesn't affect the payload
- ✓ After cancel the final chunk is still delivered, chunk ids contiguous
- ✓ None payloads are skipped with no id holes
- ✓ cleanup is a commit fence: a chunk mid-get never appears after cleanup

Follow-up questions

- With multiple save threads, how do you guarantee order?
- If build_payload needs a TP collective, can it run on a background thread?

Intuition & analogies

Setup

Building A is the main-course kitchen (upstream stage, thinker), **Building B is the plating kitchen** (downstream stage, talker). A table's (one request's) main course doesn't come out all at once; it comes out **portion by portion**, and Building B processes each portion as it arrives.

The two buildings **can't phone each other**; there's only a row of **parcel lockers** (connector, P12): each compartment has a number, locks when something is put in, and is empty once someone takes it.

Building A's **head chef** (the scheduler thread) watches 30 tables' pans at once and **can't leave the stove for a moment**: every second he steps away, all 30 pans go off the heat for a second (the GPU idles).

Analogy · The pass window and numbered tickets

Every time the kitchen (the **sender**) finishes a dish, it attaches a **numbered ticket**: 0, 1, 2..., and puts it on the pass window (the **connector**). The runner (the **recv thread**) collects by number: **until ticket k shows up, never go for k+1** - that's where ordering comes from, with no other coordination needed.

So the kitchen **tears off the next ticket only after the dish is actually on the window** (id+1 after a successful put). "No new dish this step" (payload is None) means don't tear off a ticket; if an empty plate also took a number, the runner would stand at the window forever waiting for a number that doesn't exist.

The last dish carries a sign: "**all dishes served for this table**" (`finished=True`). If the customer cancels midway (`cancel_sender`): dishes not yet made can be skipped, but the "all served" sign **must be delivered** (on an empty plate, `data=None`) - otherwise the waiter stands by the table forever.

Also: when cooking, the chef works from **a photocopy of the order** (`deepcopy`), not by looking at the customer who keeps adding dishes.

© 2026 kzhou003

Analogy · Turning a table and the table card (generation token)

Table 7's customers leave (`cleanup_receiver`) and a new party sits down right away, at what's **still table 7** (the same request id calls `load_async` again).

The runner is walking back from the window carrying a dish the previous party ordered. If he only asks "is anyone at table 7?" - yes! - he serves the old dish to the new customers.

The right way: give each party a **unique table card** when seated (`object()`). The runner notes the card before leaving and on return checks "is the card on the table **the same card**?" (`is`); if not, he throws the dish out. The check and the serving must happen **in one go** (under the same lock), so clearing the table and serving never interleave.

Framework view · P13 through T · SBWC · LIFE

P13 spans two layers of boundaries: **between processes** there's only the connector (P12's shared memory, or the problem's InMemoryConnector); **within a process** it's ordinary thread synchronization between the scheduler thread and background threads. Here T determines everything: threads share memory, so you need locks; there's a GIL, so slow pure-Python work in background threads competes with the scheduler thread for CPU - when idle they must truly sleep.

	Question	P13
T	Processes / threads / boundaries?	Each stage process: the scheduler thread (P10's busy loop) + a save thread + a recv thread (daemon threads). Between threads: `threading.Lock` / `Condition` / `Event` ; between processes: only keys on the connector.
S	Shared state? Who writes?	Sender: the pending-send queue (scheduler thread appends, save thread popleft), each request's next send id (written by the save thread), the cancelled set (written by the scheduler thread, read by the save thread → locked). Receiver: the registry `req → token` , the next receive id, the committed-but-not-yet-taken list (written by both the scheduler thread and the recv thread → one shared lock).
B	Where does it wait?	`build_payload` , `connector.put` , `connector.get` : slow, **all outside the lock**, on background threads. When idle, sleep on `Condition.wait(timeout)` ; with requests but no data, back off about 1 ms. Nothing blocks on the scheduler thread.
W	Who does what, and what's forbidden?	Scheduler thread: only snapshots and enqueues, registers, collects results; slow operations are **forbidden**. Save thread: only build + put, one at a time in FIFO order. Recv thread: only get + commit; get while holding the lock is **forbidden**.
C	Channels?	Scheduler thread → save thread: deque + Condition notify. Save thread → recv thread (cross-process): connector keys. Recv thread → scheduler thread: the ready dict, **swapped out wholesale** under the lock by `poll_ready` .
L	Lifecycle?	`start` : create the two threads; `stop` : set the Event → `notify_all` on both Conditions → join with a timeout. Per request: `load_async` issues a token → unregistered when final arrives or on `cleanup_receiver` ; the sender resets the id and cancel flag after sending final (for the next request with the same name).
I	External interface?	Apart from start / stop, every public method is called on the scheduler thread, and lock hold times must be microseconds.
F	Invariants?	① Each request's chunk ids are contiguous from 0, advance only after a successful put, and non-final None chunks take no id; ② the final chunk is always sent (with `data=None` if cancelled); ③ commit only if the token's **identity** hasn't changed; ④ each committed chunk appears in `poll_ready` exactly once, in order.
E	Edges and failures?	A chunk put before `load_async` (the connector keeps it, so it's still received); put raises (id doesn't advance); cleanup happens mid-read; a same-named request re-registers; threads asleep on wait at `stop()` .

Interactive (Chinese): `nano-vllm-omni-drills/hints/p13_overview.html`

© 2026 kzhou003

Source: `distributed/omni_connectors/transfer_adapter/base.py:20-140`

`distributed/omni_connectors/transfer_adapter/chunk_transfer_adapter.py:381-503`

`distributed/omni_connectors/transfer_adapter/chunk_transfer_adapter.py:539-555`

`distributed/omni_connectors/transfer_adapter/chunk_transfer_adapter.py:714-723`

`distributed/omni_connectors/transfer_adapter/chunk_transfer_adapter.py:991-1042`

Notes / pitfalls I hit:

© 2026 kzhou003

Track D · Model glue (pure functions)

Where "off by one row" bugs live: thinker→talker inputs, codec windows, MRoPE positions.

© 2026 kzhou003

#22 · P15 Thinker→Talker input construction (including "off by one row") ⏱ 40 min easy

□ Done Date _____

> vLLM appends the stop token to the output before checking stop, so the thinker's hidden states have one extra row. Feed that to the talker and you get noise at the end of the audio.

Where it sits in the system

In non-streaming (full payload) mode, when the thinker finishes it sends the accumulated hidden states to the talker (Qwen2.5-Omni layout; Qwen3-Omni is similar). The talker's prompt consists of the thinker's prompt part + two special positions; after that, each decode step consumes one row of thinker reply vectors.

"Off by one row"

```
prompt: P tokens, generated: [a, b, <stop>]
vLLM samples <stop> → appends to output first → then checks stop → FINISHED_STOPPED
accumulated hidden rows = P + 3 (including the <stop> step)
the talker should consume only rows a, b → drop the last row, and remove the stop id from output_ids
finish_reason = "length" (hit max_tokens): no extra row, keep everything
```

Talker prefill layout (Qwen2.5-Omni)

```
embeds = [prompt_embeds (P rows)] + [spk_embed + codec_pad_embed] + [first_reply + codec_bos_embed]
ids    = [mask_id] * P + [pad_id, start_id]                # P + 2 positions total
decode: input_k = reply_queue[k] + embed(codec sampled last step); once reply vectors run out, use fallback
(tts_pad)
```

Special ids come from the config; here they are passed as arguments.

File: `mine/p15_talker_inputs.py` **Check:** `pytest tests/test_p15_talker_inputs.py` **Hints:** A-22
Tests: B-22 **Solution:** C-22

Interface skeleton

```
"""P15 · Thinker -> Talker input construction (Qwen2.5-Omni style, pure tensor functions).

trim_thinker_payload(prompt_ids, output_ids, hidden[R, D], finish_reason)
  vLLM appends the sampled stop token BEFORE checking stop, so a "stop" finish carries one
  extra row/id the talker must not consume. Keep target = P + len(output_ids) - drop rows
  (drop = 1 for "stop", else 0); prefill = rows[:P], decode = rows[P:target]; drop the stop
  id from output_ids. Raise ValueError if fewer than P rows were captured.
build_talker_prefill(prompt_embeds[P, D], first_reply[D], spk, codec_pad, codec_bos, *ids)
  embeds = [prompt rows] + [spk + codec_pad] + [first_reply + codec_bos]   -> [P + 2, D]
  ids    = [mask_id] * P + [pad_id, start_id]
talker_decode_input(reply_queue[K, D], last_codec_embed[D], fallback[D])
  input = next thinker row (or fallback when exhausted) + embedding of the last codec id;
  return (input, remaining_queue).
"""
from dataclasses import dataclass

import torch


@dataclass
class ThinkerPayload:
    prefill: torch.Tensor
    decode: torch.Tensor
    output_ids: list[int]


def trim_thinker_payload(
```


© 2026 kzhou003

```
    prompt_ids: list[int],
    output_ids: list[int],
    hidden: torch.Tensor,
    finish_reason: str,
) -> ThinkerPayload:
    raise NotImplementedError


def build_talker_prefill(
    prompt_embeds: torch.Tensor,
    first_reply: torch.Tensor,
    spk_embed: torch.Tensor,
    codec_pad_embed: torch.Tensor,
    codec_bos_embed: torch.Tensor,
    *,
    mask_id: int,
    pad_id: int,
    start_id: int,
) -> tuple[list[int], torch.Tensor]:
    raise NotImplementedError


def talker_decode_input(
    reply_queue: torch.Tensor,
    last_codec_embed: torch.Tensor,
    fallback: torch.Tensor,
) -> tuple[torch.Tensor, torch.Tensor]:
    raise NotImplementedError
```

What the tests check

- ✓ Stop finish: drop one row and the stop id
- ✓ Length finish: keep everything
- ✓ Raises when prefill has too few rows
- ✓ Talker prefill ids/embeds layout
- ✓ Decode consumes one row per step, fallback after exhaustion

Follow-up questions

- In async_chunk mode, what should chunk 0 carry? And later chunks?

Intuition & analogies

Analogy · The tape recorder's "click"

You record a message on an old tape recorder, then reach over and press stop - but the recorder **records the "click" of the button first**, then stops. Before playing it back, you have to **trim** that click off the end (finish="stop" → drop 1).

If the recording stopped automatically because **the tape ran out** (finish="length"), the last word is something you really said, and **not a single word may be trimmed**.

There's one more case: extra blank tape at the end (extra captured rows). The correct trim is "**keep only the length there should be**", not "always cut the last second" - otherwise you cut the blank and the click is still there.

© 2026 kzhou003

> **Analogy · A seating chart: P regular seats + two special seats**
> The talker's prefill is like a seating chart with **P + 2** seats. The first P seats hold each row of the prompt (every name card just says `mask_id` - the talker doesn't read name cards, only the person, i.e. the embedding).
> Seat P is the "**host's seat**": the speaker (`spk_embed`) + the codec pad, name card `pad_id` . Seat P+1 is the "**opening seat**": the first reply row + the codec bos, name card `start_id` - this is where the talker starts speaking its first codec frame.
> After that, each decode step brings the next reply row "on stage" once; when all reply rows have gone up, a stand-in (`fallback` , tts_pad in the real system) fills in from then on.

Framework view: data flow (shape trace)

A pure-function problem, no concurrency. Walk the shapes through section 3's example (P = 3, 4 outputs including stop, R = 8, D = 2):

#	Tensor / quantity	Shape	What happens
1	`hidden`	`[8, 2]`	3 thinker prefill rows + 1 row per decode step (including the stop step) + 1 extra captured row
2	`P` , check `R ≥ P`	scalar	P = 3; 8 ≥ 3, continue (otherwise ValueError)
3	`drop` , expected row count	scalar	stop with outputs → drop = 1; 3 + 4 − 1 = 6
4	`rows = hidden[:6]`	`[6, 2]`	Row 6 (stop) and row 7 (extra) are both cut off
5	`prefill / decode`	`[3, 2]` / `[3, 2]`	Split at P; decode row count == the new `len(output_ids)` = 3
6	`spk+pad` , `first_reply+bos`	`[2]` → `[1, 2]`	A 1-D vector must become one row before it can be concatenated with `[P, D]` along dim 0
7	`embeds` / `ids`	`[5, 2]` / 5 items	P + 2 positions; ids and embeds correspond one-to-one
8	`reply_queue`	`[2, 2]` → `[1, 2]` → `[0, 2]`	Take the head row each step; once empty, use fallback, and the queue stays `[0, 2]`

Pitfalls

- **Dropping the stop row with** `hidden[:-1]` : when extra rows were captured, you cut the extra row and the stop row is still there (section 3's `[60,60]`).
- **Dropping for finish="length" too**: loses the real last word.
- **stop but output_ids is empty**: there's no "stop row" to drop; dropping anyway cuts off a prompt row.
- **Slicing before checking**: out-of-range slices don't raise, they just silently return a short tensor; the R < P check must come first.
- **Mutating the caller's output_ids in place**: the caller may still need it for detokenization; return a new list.
- `[D]` **vs** `[1, D]` : `cat` -ing a 1-D vector directly raises a dimension error; and don't swap the return order of `(ids, embeds)` .
- **Returning None when the queue is empty**: the caller passes it back in on the next step; return the empty `[0, D]` queue as is.

Interactive (Chinese): `nano-vllm-omni-drills/hints/p15_overview.html`

Source: `model_executor/stage_input_processors/qwen2_5_omni.py:261-317`
`model_executor/models/qwen2_5_omni/qwen2_5_omni.py:760-847`
`model_executor/stage_input_processors/qwen3_omni.py:409-542`

© 2026 kzhou003

Notes / pitfalls I hit:

© 2026 kzhou003

#23 · P14 Codec window planning: small first chunk, large later chunks, left context

⏱ 40 min

easy

□ Done Date _____

> TTFP must be low (first chunk only 4 frames), throughput high (25 frames per chunk afterwards), and seams lossless (recompute left context then slice).

Where it sits in the system

The talker produces 1 codec frame per step. It accumulates frames into windows and sends them to code2wav; code2wav decodes each window independently (stateless, easy to batch). Qwen3-Omni defaults: `initial_codec_chunk_frames=4, codec_chunk_frames=25, codec_left_context_frames=25`.

```
next_window(available, emitted, finished) -> (ctx_start, new_start, new_end) | None
  size = initial (first chunk and initial>0) else chunk
  available − emitted ≥ size  → new_end = emitted + size
  finished and remainder left → new_end = available (tail flush)
  otherwise                   → None (keep waiting)
  new_start = emitted;  ctx_start = max(0, new_start − left_ctx)

code2wav decodes codes[ctx_start:new_end], getting (new_end−ctx_start)·up samples,
and emits only the part belonging to [new_start, new_end) (left context recomputed but not re-emitted)
```

Why left context removes seams

The vocoder is causal with a finite receptive field: a sample depends only on the last r frames. As long as `left_ctx ≥ r`, new frames in the window see exactly the same context as in a full decode. The test proves this with a toy vocoder whose receptive field is controllable (and shows that seams do appear when it's insufficient).

File: `mine/p14_codec_windows.py` **Check:** `pytest tests/test_p14_codec_windows.py` **Hints:** A-23
Tests: B-23 **Solution:** C-23

Interface skeleton

```
"""P14 · Codec window planning for streaming vocoding (pure functions).

next_window(available, emitted, finished, *, initial, chunk, left_ctx)
  available: frames produced so far; emitted: frames already shipped.
  The first window holds `initial` frames (if initial > 0), later ones `chunk` frames.
  Return (ctx_start, new_start, new_end) when a full window is available; when finished,
  flush whatever remains; otherwise None. new_start = emitted and
  ctx_start = max(0, new_start − left_ctx) (left context is re-decoded, not re-emitted).
slice_new_samples(decoded, window, upsample)
  decoded = vocoder(codes[ctx_start:new_end]) has (new_end − ctx_start) * upsample samples;
  return only the samples that belong to frames [new_start, new_end).
"""
from collections.abc import Sequence


def next_window(
    available: int,
    emitted: int,
    finished: bool,
    *,
    initial: int,
    chunk: int,
    left_ctx: int,
) -> tuple[int, int, int] | None:
    raise NotImplementedError
```


© 2026 kzhou003

```
def slice_new_samples(decoded: Sequence[float], window: tuple[int, int, int], upsample: int) -> list[float]:
    raise NotImplementedError
```

What the tests check

- ✓ Exact windows for first chunk / steady state / tail flush
- ✓ Config without a first chunk
- ✓ Under random arrival patterns windows tile exactly, no overlap, no gaps
- ✓ Streaming == full decode when left context ≥ receptive field; seams when insufficient

Follow-up questions

- How do TTFP/RTF change with a ramp [1,2,4,8,16,25]?
- Why does stateless decoding let code2wav batch freely? (Compare with M06.)

Intuition & analogies

Analogy · Simultaneous interpretation

An interpreter doesn't wait for the speaker to finish a whole paragraph before speaking - the audience would sit waiting (high TTFP). They start translating the first sentence after hearing **two or three words** (a **small first chunk**), then settle into a rhythm, each time **hearing a full sentence** before translating (**large chunks**, smoother and less effort).

But every time they start a new sentence, they **recall the last few words of the previous one** so tone and references carry over (**left context**) - and they **don't say those words again** (decoded again, but not output twice).

When the speech ends, the leftover half-sentence still gets translated (**flush the tail when finished**).

Analogy · A TV series' "previously on"

Each episode (window) opens with a 1-minute "previously on" (`[ctx_start, new_start)`) to get the viewers (the vocoder) into it. But if you want to cut all the episodes into one complete movie, you must **cut out every episode's "previously on"**, or the same scenes appear twice. `slice_new_samples` is those scissors: trim the recap, keep only the new footage `[new_start, new_end)` .

How long should the recap be? At least long enough to **cover the memory the viewer needs** (the vocoder's receptive field). Too short, and the viewer "loses the thread" for the first few seconds - that's the seam.

Framework view: data flow (shape trace)

This is a pure-function problem, no concurrency. Walk through the unit conversions over "the life of one window" (using `(5, 9, 15)` , `upsample=4` as the example):

#	Quantity	Coordinates / shape	What happens
1	`available, emitted`	scalars, global frame counts	17, 9: 17 frames produced, the first 9 already emitted
2	`size`	frames	Not the first chunk → use `chunk = 6`
3	`(ctx_start, new_start, new_end)`	global frame indices	(5, 9, 15): context 5..8, new frames 9..14; 15 and 16 are left for next time
4	`codes[ctx_start:new_end]`	10 frames (in the real system each frame is `num_quantizers` codes)	The span sent to code2wav

© 2026 kzhou003

5	`decoded`	`[(15−5)·4] = [40]`, **local** sample coordinates	Vocoder output; index 0 is the first sample of global frame 5
6	slice start	`(9−5)·4 = 16`	Convert "global frame 9" to a local sample index
7	return value	`[24]` = 6 frames × 4	Only new frames are output; all windows' outputs concatenated end to end = 17×4 = 68 samples

Pitfalls

- **First-chunk check**: use "no frames emitted yet" (emitted is 0) and `initial > 0`; don't guess from the size of available - frames may arrive many at once.
- **Check order**: first "is there enough for a full window", then "finished with leftovers". finished does not mean "pack up everything that's left".
- **Never return an empty window**: return None when the stream has ended and everything has been emitted, or the caller's while loop never stops.
- **ctx_start must not be negative**: at the start of the stream there aren't that many frames to the left.
- **Local vs global coordinates**: `decoded` counts from `ctx_start`; slicing directly with `new_start·upsample` is the most common bug (it's only right by coincidence when ctx_start=0).
- **left_ctx ≥ receptive field** is required for sample-exact, seamless output; this relies on the vocoder being "causal + local".

Interactive (Chinese): `nano-vllm-omni-drills/hints/p14_overview.html`

Source: `model_executor/stage_input_processors/qwen3_omni.py:765-842`
`model_executor/models/qwen3_omni/qwen3_omni_code2wav.py:447-508`

Notes / pitfalls I hit:

© 2026 kzhou003

#24 · P18 MRoPE 3D positions ⏱ 40 min medium

□ Done Date _____

> The Qwen-VL/Omni thinker uses (t,h,w) 3D rotary positions. Compute the positions wrong and the model won't error; it just quietly gets dumber.

Where it sits in the system

M01 is "how to rotate given positions"; this drill is "how the positions themselves are computed". At prefill the model runner computes `[3, T]` positions for each request, plus a `delta` used in decode.

```
text, n tokens:       all three axes are start + arange(n)
image (t, h, w) patch: LLM grid (t, h/m, w/m), m = spatial_merge (2)
                    token order t-major, then h, then w; position = (t_idx, h_idx, w_idx) + start
end of each segment: start = current max position + 1   # note: NOT "number of tokens so far"!
delta = (max_pos + 1) - T                 # in decode a token's position = absolute index + delta (same on all
axes)
```

Example

```
text(2) | image(1,4,4) → 2×2 grid | text(3)
t: 0 1 | 2 2 2 2 | 4 5 6
h: 0 1 | 2 2 3 3 | 4 5 6
w: 0 1 | 2 3 2 3 | 4 5 6        T = 9, max = 6, delta = 7 − 9 = −2, decode position of token 9 = 7
```

The image takes 4 tokens but advances only 2 positions; that is why delta is negative, and the source of the "use len(tokens) as the decode position" bug.

File: `mine/p18_mrope_positions.py` **Check:** `pytest tests/test_p18_mrope_positions.py` **Hints:** A-24
Tests: B-24 **Solution:** C-24

Interface skeleton

```
"""P18 · MRoPE (Qwen2-VL style) 3-D positions for interleaved text and images.

segments: ("text", n) or ("image", (t, h, w)) with h, w in vision patches; an image
becomes t * (h // merge) * (w // merge) LLM tokens laid out t-major, then h, then w.
- text tokens: the same running index on the temporal, height and width axes;
- image tokens: (t_idx, h_idx, w_idx) + start;
- after each segment, start = max(position so far) + 1.
Return (positions[3, T] long, delta) with delta = (max position + 1) - T, so a token at
absolute index i >= T decodes at position i + delta on all three axes (decode_position).
"""
import torch

Segment = tuple[str, int | tuple[int, int, int]]


def mrope_positions(segments: list[Segment], spatial_merge: int = 2) -> tuple[torch.Tensor, int]:
    raise NotImplementedError


def decode_position(index: int, delta: int) -> torch.Tensor:
    raise NotImplementedError
```


© 2026 kzhou003

What the tests check

- ✓ Pure text reduces to ordinary RoPE
- ✓ Exact positions and delta for text+image+text
- ✓ Multi-frame rectangular grid
- ✓ Shape and dtype

Follow-up questions

- How does Qwen2.5-Omni's TMRoPE interleave audio and video by the second?

Intuition & analogies

Analogy · Cinema seat numbers

Text tokens are like **one ticket per person**: everyone sits in "row k, seat k" (all three axes equal), and the row number advances one by one.

An image is like a **tour group**: 6 people book a 3-row × 2-seat block. Their tickets read "row 2+r, seat 2+c" - people in the same row share a row number. The whole group uses up only **3 rows**, not 6.

Where do the individual moviegoers after the group start? At **the group's largest row number + 1**. If you hand them row 8 because "8 people have already come in", several rows are left empty in between - and the model thinks the two pieces of text are far apart.

Analogy · Floors and room numbers

Think of t as the floor, h and w as the room's row and column numbers. A video clip is a 3-story building with just 1 room per floor: take the elevator up 3 floors (t advances by 3), but the room number doesn't change (h and w both stay at the same value).

After leaving the building, the next piece of text continues from "the largest of the three coordinates + 1" - whether that max is the floor (video) or the column (a wide image). That's why start takes **the max over all three axes**.

Framework view: data flow (shape trace)

A pure-function problem, no concurrency. Trace every intermediate value with section 4's example:

#	Quantity	Shape / dtype	What happens
1	`segments`	5 segments	In sequence order, one-to-one with the placeholder tokens in the prompt
2	image `(1, 6, 4)` → `(t, gh, gw)`	(1, 3, 2)	`gh = h // m`, `gw = w // m`; no merging along t
3	the segment's `(t_idx, h_idx, w_idx)`	3 × `[6]`	Expand with t outermost, h next, w innermost: h = 0,0,1,1,2,2; w = 0,1,0,1,0,1
4	block = stack + start	`[3, 6]` long	start = 2, add 2 to all three rows
5	new start	int	Max of the block (over all three axes together) + 1 = 4 + 1 = 5
6	text block	`[3, n]`	`start + arange(n)` copied into three rows
7	`positions`	`[3, 14]` long	All blocks concatenated along dim=1
8	`delta`	int	`(10 + 1) − 14 = −3`, stored in the request state

© 2026 kzhou003

9	`decode_position(i, delta)`	`(3,)` long	i = 14 → [11, 11, 11], right after prefill's max position 10

Pitfalls

- **start += token count**: correct for text only, wrong once there's an image (in the example the second text segment would start at 8 instead of 5).
- **Swapping h and w / wrong expansion order**: invisible on square grids; always self-test with a **rectangular** grid (e.g. 3×2) and **t > 1**.
- **Looking at only one axis's max**: the max may come from t (video), h (a tall image), or w (a wide image).
- **Missing the +1 in delta**, or using the index directly as the position during decode: the first decode position should be max+1.
- **Empty segments** (n = 0 or a zero grid) have no max and must not update start; with no segments at all, return a `[3, 0]` long tensor.
- **dtype**: positions are integer indices; use `torch.long`.

Interactive (Chinese): `nano-vllm-omni-drills/hints/p18_overview.html`

Source: `model_executor/models/qwen2_5_omni/qwen2_5_omni.py:412-576`
`model_executor/models/qwen3_omni/qwen3_omni_moe_thinker.py:1694-1785`

Notes / pitfalls I hit:

© 2026 kzhou003

Track E · Startup & GPU

Multi-stage startup on shared GPUs, memory admission, CUDA graph buckets.

© 2026 kzhou003

#25 · P16 Startup planning: device groups, parallel init, rollback, admission

45 min | medium

□ Done Date _____

> The thinker owns GPU0, while the talker and code2wav share GPU1: the latter two must initialize serially (their memory measurements interfere), but the two groups can run in parallel. If any one fails, everything already started must be torn down.

Where it sits in the system

At startup each stage: loads weights → runs a profile at max batch → computes the KV block count from "memory budget – used" → captures CUDA graphs. Steps 2 and 3 depend on **measuring currently free GPU memory**: if another stage on the same GPU happens to be loading weights, the measurement is wrong. Too many KV blocks means OOM, too few wastes memory.

Requirements

- **Grouping**: stages that share devices (transitively: {0,1} and {1,2} are also one group) belong to the same group; within a group run serially by stage id, across groups run in parallel in a thread pool.
- **Fail fast**: after the first failure, start no new init; let in-flight ones finish; then call cleanup on all stages that succeeded; raise `InitError(stage_id)`.
- **Admission check**: before startup check for each GPU that `Σ(util × total + reserve) ≤ total`, rejecting deployments that can't succeed up front instead of waiting for the third stage to OOM.

```
Default Qwen3-Omni (2×H100):
  GPU0: thinker(0.9)            → group [0]
  GPU1: talker(0.6) code2wav(0.1) → group [1, 2] (serial)     the two groups run in parallel
```

File: `mine/p16_init_planner.py` **Check:** `pytest tests/test_p16_init_planner.py` **Hints:** A-25 **Tests:** B-25 **Solution:** C-25

Interface skeleton

```
"""P16 · Stage start-up planning: device groups, parallel init, rollback, admission.

init_groups(stage_devices): stages that share a device -- directly or transitively --
  form one group. Return groups as sorted stage lists, ordered by their first stage.
run_init(stage_devices, init_fn, cleanup_fn, max_workers):
  groups run in parallel (thread pool); stages inside a group run one after another in
  stage-id order. On the first failure: no NEW init_fn calls start, running ones finish,
  then cleanup_fn(stage, result) for every stage that succeeded, and raise
  InitError(failed_stage, cause). Returns {stage: init_fn(stage)} on success.
admission_check(device_total_bytes, stages, reserve_bytes_per_stage):
  for every device, sum(utilization * total + reserve) over stages placed on it must not
  exceed total; return one error string per offending device (mentioning "device <id>").
"""
from collections.abc import Callable
from typing import Any


class InitError(Exception):
    def __init__(self, stage_id: int, cause: BaseException) -> None:
        super().__init__(f"stage {stage_id} failed to initialize: {cause!r}")
        self.stage_id = stage_id


def init_groups(stage_devices: dict[int, set[int]]) -> list[list[int]]:
    raise NotImplementedError
```


© 2026 kzhou003

```
def run_init(
    stage_devices: dict[int, set[int]],
    init_fn: Callable[[int], Any],
    cleanup_fn: Callable[[int, Any], None],
    max_workers: int = 4,
) -> dict[int, Any]:
    raise NotImplementedError


def admission_check(
    device_total_bytes: dict[int, int],
    stages: dict[int, tuple[set[int], float]],
    reserve_bytes_per_stage: int,
) -> list[str]:
    raise NotImplementedError
```

What the tests check

- ✓ Connected components of device overlap
- ✓ Parallel across groups, serial within a group by stage id
- ✓ Fail fast + roll back stages that succeeded
- ✓ GPU memory admission check

Follow-up questions

- When truly initializing same-GPU stages in parallel, how do you use a cross-process read-write lock to make "measure free memory" exclusive?

Intuition & analogies

Analogy · Tour groups sharing tables (device grouping = union-find)

In the restaurant, each **table** is a GPU and each **tour group** is a stage. **Sharing tables is allowed**: one table can seat several groups at once (multiple stages sharing one card); a big group that doesn't fit at one table **takes two tables at once** (a TP=2 stage takes two cards).

The only thing that needs a queue is **being seated** (initialization): when a group sits down, it first checks "how many seats are left at this table", then sets out its own place settings (memory profiling + allocation). If two groups are seated at the same table **at the same time**, both think the free seats are theirs and set out too much. So **groups sharing a table must be seated one at a time** (serial within a group); once seated, eating together is totally fine. Groups that share no table can be seated simultaneously (parallel across groups).

The key is **transitivity**: group A sits at table 2, group B takes tables 2 and 3 at once, group C sits at table 3. A and C never shared a table, but B can only be seated when **nobody is being seated** at either table 2 or 3 - the simplest approach that's guaranteed correct is to have **the same host** seat A, B, C in order. (This is a conservative scheme: A and C could in theory be seated at the same time, but that would complicate B's arrangement.) Different hosts handling disjoint sets of tables can work simultaneously.

Union-find is "merging the hosts' seating lists": for each arriving group, check whether any of its tables is already on some host's list; if so, merge the two lists into one.

© 2026 kzhou003

Analogy · The moving company pulls out (failure rollback)

Three moving crews are carrying furniture into three buildings at once (parallel initialization). Crew B drops and breaks its second piece (init fails), and the dispatcher calls a halt:
① Pieces **not yet started** must not be started (short-circuit, no new inits); ② the piece **being carried right now** can't be dropped mid-air - finish carrying it first (let running inits finish); ③ only after **everyone has stopped**, carry **all** the furniture already moved in back out (cleanup every successful stage) - if you start the inventory the moment B breaks something, you'll miss the piece crew A "set down a bit later".

Analogy · Restaurant reservations (memory admission, continuing the table-sharing analogy)

An 80-seat table (one card): each tour group (stage) reserves "a percentage of the table size", plus a fixed number of spare chairs per table (reserve). Before service, do the math: `Σ(fraction × 80 + spare chairs) ≤ 80` ? If it's over, tell the guests on the spot "this table can't fit you", rather than kicking them out after two courses. For a big client spanning two tables (TP=2), its fraction is counted **at each table** separately.

Framework view: T · SBWC · LIFE

	Question	P16's answer
T	Where does the code run? What boundaries does it cross?	The main thread calls `run_init` ; a **thread pool** runs one task per group (not asyncio: `init_fn` is a blocking synchronous call - loading weights, waiting for a child process to become ready). In the real system each stage is also a separate process, and mutual exclusion for same-card measurement relies on a cross-process file lock.
S	Shared state? Who writes?	`results` (successful stage → handle), `failure` (the first failure, at most one). Written by multiple worker threads → one `threading.Lock` protects both.
B	Who sits and waits?	`init_fn(sid)` - very slow. It must be called **outside the lock**, or parallelism across groups degrades to global serialization. The main thread waits for all groups at "thread pool exit".
W	Roles	One worker per group: a for loop in stage-id order; check failure **before starting each new init**. Forbidden: parallelism within a group, cleanup at the moment of failure.
C	Channels	No queues; workers communicate only via lock-protected shared state (a "something went wrong" flag).
L	Lifecycle	**Join first, then roll back**: exiting the thread pool's with block = all groups done and results no longer changing; only then clean up everything, then raise.
I	External interface	On success, return the full dict; on failure there's only one exit: `InitError(the first failed stage)` , with the original exception attached as `__cause__` .
F	Invariants	① Inits of stages in the same group never overlap, and run in id order; ② after the first failure no new init is **started**; ③ on return/raise, every stage that succeeded is either in the results or has been cleaned up - exactly once; ④ the failed stage is not cleaned up (it has no handle).
E	Edges and errors	Several groups fail almost simultaneously (record only the first); a running init succeeds only after the failure (must be rolled back too); `max_workers` smaller than the number of groups (a queued group sees the failure when its turn comes and returns immediately); exceptions that aren't `Exception` , like `KeyboardInterrupt` , must also trigger rollback; an empty card doesn't error; need exactly equal to total is allowed.

Interactive (Chinese): `nano-vllm-omni-drills/hints/p16_overview.html`

Source: `engine/stage_runtime.py:879-956` `engine/stage_init_utils.py:1738-1762` `engine/stage_admission.py:205` `engine/stage_phase_lock.py:106-238`

© 2026 kzhou003

Notes / pitfalls I hit:

© 2026 kzhou003

#26 · P17 CUDA graph bucket runner (CPU simulation) ⏱ 40 min easy ☐ Done Date _____

> Decode's kernel launch overhead is eliminated with CUDA graphs. A graph only knows static buffers: pad, zero, replay, clone, and when you must fall back to eager.

Where it sits in the system

In decode each step has very little compute but launches hundreds or thousands of kernels; CPU launch speed becomes the bottleneck. A CUDA graph records one forward as a graph, and afterwards a single launch replays the whole graph. The cost: the graph only knows **the memory it was recorded on** and **the shapes it was recorded with**.

```
At startup: for each batch bucket (1,2,4,8,...) record one graph on the first b rows of the same static buffer
At runtime (bs requests):
  b = smallest bucket ≥ bs; none → eager
  static_in[:bs] = x;  static_in[bs:b] = 0      # dirty rows left by a previous larger batch
  graph[b].replay()                             # results written to static_out[:b]
  return static_out[:bs].clone()                # the next replay overwrites static_out!
```

This drill uses `CpuGraph` to simulate these semantics on CPU: replay only ever reads the static buffers and accepts no new arguments.

When you must fall back to eager

- bs exceeds the largest bucket;
- a request in the batch carries its own RNG generator (P08): replay doesn't re-run Python code, so per-row generators would be silently ignored.

File: `mine/p17_graph_runner.py` **Check:** `pytest tests/test_p17_graph_runner.py` **Hints:** A-26 **Tests:** B-26 **Solution:** C-26

Interface skeleton

```
"""P17 · GraphRunner: batch-size buckets over captured graphs (CPU emulation of CUDA graphs).

- __init__: allocate static_in / static_out of shape [max bucket, hidden] once.
- capture(): one CpuGraph per bucket b over the views static_in[:b] / static_out[:b]
  (largest bucket first), call .capture(), keep them in self.graphs[b].
- run(x, force_eager=False):
    pick the smallest bucket >= bs; if none, or force_eager (e.g. rows carry their own RNG
    generators, which a replay would silently ignore) -> stats["eager"] += 1, return fn(x);
    else copy x into static_in[:bs], ZERO static_in[bs:b] (stale rows), replay,
    stats["replays"] += 1, stats["last_bucket"] = b, and return a result that is NOT a view
    of static_out (the next replay overwrites it).
- Never capture at run time.
"""
from collections.abc import Callable

import torch


# ---------------------------------------------------------------- given types
class CpuGraph:
    """Emulates torch.cuda.CUDAGraph semantics on CPU: replay() re-runs the recorded work on
    the *same static buffers*; it never sees new Python arguments."""

    def __init__(self, fn: Callable[[torch.Tensor], torch.Tensor], static_in: torch.Tensor, static_out:
torch.Tensor):
        self._fn, self._in, self._out = fn, static_in, static_out
        self.captured = False

    def capture(self) -> None:
```


© 2026 kzhou003

```
        self._out.copy_(self._fn(self._in))
        self.captured = True

    def replay(self) -> None:
        assert self.captured
        self._out.copy_(self._fn(self._in))


# ---------------------------------------------------------- implement below
class GraphRunner:
    def __init__(self, fn: Callable[[torch.Tensor], torch.Tensor], buckets: list[int], hidden: int,
                 dtype: torch.dtype = torch.float32) -> None:
        self.fn = fn
        self.buckets = sorted(buckets)
        self.graphs: dict[int, CpuGraph] = {}
        self.stats = {"replays": 0, "eager": 0, "last_bucket": None}
        raise NotImplementedError

    def capture(self) -> None:
        raise NotImplementedError

    def run(self, x: torch.Tensor, *, force_eager: bool = False) -> torch.Tensor:
        raise NotImplementedError
```

What the tests check

- ✓ Results match eager for every batch size
- ✓ Bucket selection and both eager fallbacks
- ✓ Return value doesn't alias the static buffer
- ✓ Padding rows are zeroed
- ✓ Never re-captures at runtime

Follow-up questions

- Should graph memory count toward the KV budget? Where does vllm-omni's default path miss it?
- Why is the talker's MTP (M05: 15 small-model forwards) a good fit for wrapping into one FULL graph?

Intuition & analogies

> **Analogy · Fixed-size baking trays**
>
> The bakery has only three tray sizes: 1, 3, and 6 slots, and each tray's oven program was **set up in advance** (capture); it's never reprogrammed once work starts.
>
> 2 lumps of dough arrive: use the **3-slot** tray (round up to a bucket), and the empty slot must first **be swept clean of crumbs from the previous batch** (zero the padding rows), or it bakes into a lump of charcoal that may even smoke out the neighboring loaves (in a real model it would contaminate cross-row computation).
>
> Once baked, you must **take the bread off the tray and bag it** before handing it to the customer (clone) - if you just point the customer at "those slots on the tray", the moment the next batch starts baking, the bread in their hands becomes someone else's.
>
> 7 lumps arrive? The largest tray has only 6 slots → bake them by hand one by one (eager).

© 2026 kzhou003

> **Analogy · A karaoke backing track**
> A CUDA graph is like **a pre-recorded backing track**: press play (replay) and it plays back exactly the same way. You can swap the singer (copy new data into the same microphone = the static buffer), but you **can't have the band rearrange the song during playback**.
> If a customer asks "use my own random rhythm for this song" (the request brings its own `torch.Generator`), the backing track won't complain; it just **plays the rhythm it was recorded with** - your request is silently ignored. So such requests need a live band (`force_eager`).

Framework view: data flow (shape trace)

This is a "CPU-simulated GPU runner" with no real concurrency. Walk through one replay call (using step 3 of the interactive figure: `bs=2`, bucket 3, `hidden=2`):

#	Tensor	Shape	What happens
1	`x`	`[2, 2]`	The caller's new batch (a Python argument; the graph can't see it)
2	pick bucket	scalar 3	The smallest bucket ≥ 2; none / `force_eager` → just `fn(x)`
3	`static_in[:2] ← x`	`[2, 2]` written into `[6, 2]`	Copy in place into the memory used at capture time
4	`static_in[2:3] ← 0`	`[1, 2]`	Zero the padding row (previously the 5 left by the last bs=5 call); rows 3..5 are outside the bucket and untouched
5	`graphs[3].replay()`	reads `static_in[:3]`, writes `static_out[:3]`	Outputs 5, 5, -1 (padding row = fn(0))
6	`static_out[:2].clone()`	`[2, 2]`	Take only the real rows, and copy them out, decoupled from the static buffer

Pitfalls

- **Reassigning instead of copying**: replacing `self.static_in` with a new tensor while the graph keeps reading the old memory.
- **Forgetting to zero the padding rows**: in this problem fn is row-independent, so dirty rows only garble the padding output; real models have cross-row interactions (MoE routing statistics, NaN propagation, whole-batch reductions) that contaminate the real rows.
- **Returning a view**: invisible on a single call; after the second call, the first call's result is silently overwritten.
- **The eager branch touching the static buffer**, or updating `last_bucket` in eager mode.
- **Capturing on the fly at runtime**: capture is slow and needs warmup and synchronization, causing latency spikes; all graphs must be recorded at startup.
- **Capture order**: largest to smallest (on CUDA multiple graphs share one memory pool; the big graph claims memory first, and the small ones reuse it).

Interactive (Chinese): `nano-vllm-omni-drills/hints/p17_overview.html`

Source: `worker/gpu_ar_model_runner.py:634-690` `worker/gpu_model_runner.py:243-272` `worker/base.py:140-147`

© 2026 kzhou003

Notes / pitfalls I hit:

Capstone: assemble the pieces into a mini omni service (half a day to a day)

Using your own implementations, assemble a 3-stage toy pipeline on CPU and verify the concurrency semantics end to end:

1. **Processes and engines.** Start three `multiprocessing` (spawn) child processes, each running the P10 busy loop. Inside the engine use the P07 scheduler; the P06 block manager is optional. The "model" can be a toy function at first, or swap stage 0 for the M08 TinyThinker.
2. **Control plane.** In the parent process start an orchestrator thread that drives the three stages with P03 and hands results to the front end via P01. Use P04 for abort and P05 for RPC.
3. **Data plane.** Pass data directly between stages with P12 + P13, park requests on the receiving side with P11, and window the codec with P14. Use P16 for start-up.
4. **Acceptance.** Send 100 concurrent requests, abort a random 10% midway, and kill stage 2 once at random. Requirements: exactly one terminal message per request, nothing left in `/dev/shm`, no zombie processes after exit.

Then move on to the Part II project: replace the toy model with the real Qwen2.5-Omni-3B and run it on a single 5090.

© 2026 kzhou003

Appendix A · Three-level hints (look only after 15+ minutes stuck; L1 → L2 → L3 give away progressively more)

A-1 · RMSNorm and multimodal RoPE (interleaved / chunked)

Level 1 · I/O examples

rms_norm: `x = [[3, 4], [1, 1]]` (fp32), `weight = [1, 2]`, `eps = 0`.

```
row0: mean(x²) = (9+16)/2 = 12.5, rsqrt = 1/3.5355
      x̂ = [0.8485, 1.1314]  -> * weight -> [0.8485, 2.2627]
row1: mean(x²) = 1 -> x̂ = [1, 1] -> * weight -> [1.0000, 2.0000]
```

Note the mean is over the **last dim** (each row independently); the output has the same shape as x.

rope_inv_freq: `rope_inv_freq(8, 10000.0)` → `[1.0000, 0.1000, 0.0100, 0.0010]` (length D/2 = 4, i.e. 10000^(-0/8), 10000^(-2/8), 10000^(-4/8), 10000^(-6/8)); `rope_inv_freq(6, 100.0)` → `[1.0000, 0.2154, 0.0464]`.

Visualizing the "axis assignment" of mrope_cos_sin: take `inv_freq = ones(10)` (D = 20), a single token, `positions = [[1],[2],[3]]` (t=1, h=2, w=3). Each frequency's angle then equals the position of the axis it uses, so `acos(cos)` tells you directly which axis frequency j rotates with:

section / layout	acos(cos[0]) (length 20)
`[4,3,3]`, interleaved=True	`1 2 3 1 2 3 1 2 3 1 | 1 2 3 1 2 3 1 2 3 1`
`[4,3,3]`, interleaved=False	`1 1 1 1 2 2 2 3 3 3 | 1 1 1 1 2 2 2 3 3 3`
`[6,2,2]`, interleaved=True	`1 2 3 1 2 3 1 1 1 1 | (second half same as first)`

The last row shows: H/W only occupy the first 3·section "slots"; everything after is T. The second half is a copy of the first half (`cat([f, f])`).

rotate_half / apply_rotary: `rotate_half([1, 2, 3, 4]) = [-3, -4, 1, 2]`. Take D = 4, `inv_freq = rope_inv_freq(4, 10000) = [1, 0.01]`, a pure-text position p = 2 (all three axes are 2, section=[1,1,0], interleaved):

```
cos = [cos2, cos0.02, cos2, cos0.02] = [-0.4161, 0.9998, -0.4161, 0.9998]
sin = [ 0.9093, 0.0200, 0.9093, 0.0200]
apply_rotary(x=[[[1,0,0,0]]]) = [[[-0.4161, 0, 0.9093, 0]]]     # dims 0 and 2 form one rotation pair
apply_rotary(x=[[[1,2,3,4]]]) = [[[-3.1440, 1.9196, -0.3391, 4.0392]]]
```

x has shape `[T=1, heads=1, D=4]`; cos/sin are `[T=1, D=4]`.

Key questions

1. In RMSNorm, is there a difference between "cast back to x.dtype, then multiply by weight" and "multiply by weight in fp32, then cast back" for fp32 input? For bf16 input? Why?
2. What does `torch.arange(0, head_dim, 2)` give you? After dividing by head_dim, what is the range of exponents? What is the length of inv_freq?
3. `positions` has shape `[3, T]` and `inv_freq` is `[D/2]`. How do you get `freqs[3, T, D/2]` for all axes with a single broadcast?
4. In the interleaved layout, with frequency index j = 7 and section = [4,3,3], which axis does it belong to? What about section = [6,2,2]?
5. In rotate_half, is dim j paired with dim j+1 or with dim j + D/2? How does this relate to `cat([f, f])` (as opposed to repeating each frequency twice in adjacent slots)?
6. If all tokens are pure text (same position on all three axes), how should the interleaved and chunked cos/sin relate? Why?
7. In `apply_rotary`, x is `[T, heads, D]` and cos is `[T, D]`. How must you reshape to broadcast correctly across all heads?

Details

- **"normalize in fp32, cast back, THEN multiply by weight"**: compute `x * rsqrt(mean(x², -1) + eps)` entirely in fp32; the normalized result is first `.to(x.dtype)`, then multiplied by weight (usually the same dtype as x). In bf16 the two orders differ by about 1 ulp. For example, on one bf16 input the correct order gives `1.9140625` while "multiply by weight in fp32 first" gives `1.90625`: numerically "close" but not bitwise identical.
- **rope_inv_freq must be fp32**: `torch.arange` defaults to int64; `int / int` in PyTorch yields a float, but its dtype depends on the default dtype setting. Explicitly passing `dtype=torch.float32` is safest. `theta ** tensor` is an

© 2026 kzhou003

elementwise power with a Python scalar base.

- **What section means**: `mrope_section = [s_t, s_h, s_w]` sums to D/2 and is "how many **frequencies** each axis gets" (not dims; the dim count is 2x). In chunked it is just three contiguous segment lengths; in interleaved, H takes j = 1, 4, 7, … (s_h of them, i.e. j%3==1 and j < 3·s_h), W takes j = 2, 5, 8, … (j%3==2 and j < 3·s_w), and everything else goes to T.
- **The shape of cos/sin** is `[T, head_dim]` : do the axis selection on `[T, D/2]` first, then `cat([f, f], −1)` to get D, and finally take `.cos()` / `.sin()` . This corresponds to neox-style "first half / second half pairing".
- **dtype**: cos/sin are computed in fp32; when `apply_rotary` sees a bf16 x, it should cast cos/sin to x's dtype before multiplying (matching HF).
- **Qwen3 vs Qwen2.5**: inv_freq, rotate_half, and the apply formula are identical in both. The only difference is "which axis's position frequency j uses", and that is the entire meaning of the `interleaved` bool.

Level 2 · Approach & pitfalls

- **Decomposition**: the five functions are independent yet compose in layers: `rope_inv_freq` → `mrope_cos_sin` → `apply_rotary` (which calls `rotate_half`). `rms_norm` stands alone. Write them in this order and self-check each one against the numbers in the io examples as you go.
- **rms_norm**: three steps: `x32 = x.float()` ; `x32 * torch.rsqrt(x32.pow(2).mean(−1, keepdim=True) + eps)` ; `.to(x.dtype)` , then multiply by weight. Common mistakes: forgetting `keepdim=True` (wrong broadcast or an error), putting eps outside the sqrt, taking the mean over all dims.
- **The core of mrope_cos_sin is "selection", not "computation"**: first compute each axis's `freqs[a]` in one shot (shape `[T, D/2]`), then build a `[T, D/2]` result f where each column j is copied from some axis's freqs. Only the source of each column differs; there is no extra math.
- **Implementing interleaved**: start from the T axis as the base (clone it so you don't mutate the original tensor), then overwrite H's columns (start 1) and W's columns (start 2) with stride-3 slices whose end is determined by the corresponding section. Alternatively, build a length-D/2 "axis index" vector `axis_of_j` and select with `gather` /advanced indexing.
- **Implementing chunked**: keep a start cursor and successively slice column segments of length s_t, s_h, s_w out of freqs[0], freqs[1], freqs[2], then concatenate. Note you slice the **same column range**: the H segment takes columns s_t..s_t+s_h of freqs[1], not columns 0..s_h of freqs[1].
- **rotate_half**: use `chunk(2, dim=−1)` to split the last dim into front/back halves a, b, and return `cat([−b, a], −1)` . Common mistakes: writing `[b, −a]` (rotation direction reversed, the sign of sin flips everywhere), or splitting by even/odd positions (GPT-J style, which doesn't match cat([f,f])).
- **Broadcasting in apply_rotary**: cos/sin are `[T, D]` , x is `[T, heads, D]` ; you need to insert a head dim in the middle: `cos[:, None, :]` . If you multiply directly, PyTorch aligns cos's T dim with x's heads dim. When T == heads this doesn't error but the result is wrong: the sneakiest bug here.
- **Self-test trick**: run mrope_cos_sin with `inv_freq = ones` plus distinct small integer positions (e.g. 1/2/3), then take `acos` , and you can see by eye which axis each column comes from; use identical positions on all three axes to verify both layouts degenerate to plain RoPE.

Level 3 · Walkthrough (code: see the commented solutions in Appendix C)

Key question answers:

- For fp32 input, `.to(x.dtype)` is a no-op and the two orders give the same result; in bf16, "round to bf16 then multiply by the bf16 weight" and "multiply in fp32 then round" round at different points and can differ by 1 ulp.
- `arange(0, D, 2)` = `[0, 2, …, D−2]` ; after dividing by D the exponents lie in [0, 1); inv_freq has length D/2 and decreases monotonically from 1 to nearly 1/θ.
- `positions.float()[:, :, None] * inv_freq[None, None, :]` → `[3, T, D/2]` .
- j = 7: 7 % 3 == 1; with section=[4,3,3], 3·3 = 9 > 7 → H; with section=[6,2,2], 3·2 = 6 ≤ 7 → T.
- Dim j pairs with j + D/2; because cos/sin are `cat([f, f])` , column j and column j+D/2 share the same angle, and rotate_half combines exactly these two columns into a 2D rotation.
- Same position on all three axes → the three freqs are identical → no matter how you select columns, the result is plain 1D RoPE, and the two layouts are bitwise identical.
- `cos[:, None, :]` (`[T, 1, D]`).

mrope_cos_sin shape trace (T tokens, D = head_dim, half = D/2):

```
positions [3, T] --float--> [3, T, 1] * inv_freq [1, 1, half] = freqs [3, T, half]
select columns                                                  -> f     [T, half]
cat([f, f], -1)                                                 -> emb   [T, D]
emb.cos(), emb.sin()                                            -> [T, D], [T, D]
```

Skeleton Check: `slice(1, 9, 3)` selects 1, 4, 7 (3 of them = s_h); `slice(2, 9, 3)` selects 2, 5, 8 (3 of them = s_w).
rotate_half / apply_rotary skeleton Verify the io example: x = [1,2,3,4], cos = [c0,c1,c0,c1], sin = [s0,s1,s0,s1], rotate_half(x) = [-3,-4,1,2]: `out[0] = 1·c0 − 3·s0 = −0.4161 − 2.7279 = −3.1440` , `out[2] = 3·c0 + 1·s0 = −1.2484 + 0.9093 = −0.3391` : exactly a 2D rotation matrix acting on the pair (x0, x2).
Final checklist: (1) in bf16, rms_norm casts first, then multiplies by weight; (2) inv_freq is explicitly fp32; (3) interleaved uses `clone()` then overwrites, slice end is `3*section[axis]` ; (4) chunked column ranges use a cumulative offset; (5)

© 2026 kzhou003

`cat([f, f])`, not `repeat_interleave`; (6) apply_rotary inserts the head dim and matches dtype.

A-2 · Thinker attention: QK-norm + GQA + KV cache

Level 1 · I/O examples

A toy you can do by hand: hidden = 2, `num_heads = num_kv_heads = 1`, `head_dim = 2`, q/k/v/o_proj are all 2×2 identity matrices, `q_norm = k_norm = [1, 1]`, `eps = 0`; cos all 1, sin all 0 (i.e. no rotation).

```
x = [[1, 0],
     [0, 1],
     [3, 3]]                      # T = 3
after QK-norm: q = k = [[1.414, 0], [0, 1.414], [1, 1]]     # each row divided by its own RMS
v is not normed:  v = x  = [[1, 0], [0, 1], [3, 3]]
attention_forward(w, x, cos, sin) =
    [[1.0000, 0.0000],            # token0 can only see itself
     [0.1956, 0.8044],            # token1: scores [0, 1.414] -> softmax [0.1956, 0.8044]
     [1.5768, 1.5768]]            # token2: scores [1, 1, 1.414] -> softmax [0.2846, 0.2846, 0.4307]
```

scale = $D^{-0.5}$ = 1/1.414; e.g. token1's score against itself = (1.414·1.414)/1.414 = 1.414.

Same toy, split into two calls (chunked prefill / decode):

```
cache = KVCache()
a = attention_forward(w, x[:2], cos[:2], sin[:2], cache)   # -> [[1, 0], [0.1956, 0.8044]]
len(cache) == 2; cache.k == [tensor of shape [2, 1, 2]]
b = attention_forward(w, x[2:], cos[2:], sin[2:], cache)   # -> [[1.5768, 1.5768]]
len(cache) == 3; cache.k holds two chunks: [2,1,2] and [1,1,2]
```

`cat([a, b])` matches the one-shot prefill row by row. Note the second call's cos/sin has only 1 row: it belongs to the new token at absolute position 2.

What two typical bugs produce (same toy):

Bug	Result
No causal mask at all	`[[1.5665, 1.1690], [1.1690, 1.5665], [1.5768, 1.5768]]`: only the last row happens to be right
Decode mask uses the in-chunk index (q_pos = 0 instead of 2)	The third token only sees key 0 and outputs `[1, 0]` instead of `[1.5768, 1.5768]`

Key questions

1. q_proj has shape `[H·D, hidden]`. After `x @ q_proj.T` gives `[T, H·D]`, how do you view it as `[T, H, D]` so that "head h" is a contiguous run of D columns?
2. Over which dim is QK-norm applied? q_norm has shape `[D]`; how does it broadcast to `[T, H, D]`? Does v get normed?
3. cos/sin has only T rows (new tokens only), while K has S = past + T rows. Do the old K in the cache need RoPE re-applied? Why?
4. What is the absolute position of new token i (in-chunk index i)? Which range of key indices can it see?
5. To turn k of `[S, KVH, D]` into `[S, H, D]` with a matching kv head per Q head, should you use `repeat` or `repeat_interleave`? What mapping does each give for H=4, KVH=2?
6. Which steps does "softmax in fp32" cover? Where do you cast back to x.dtype?
7. Should you append the new k/v to the cache before attention, or attend first and append after? Which order lets a token see itself?

Details

- **"RMSNorm EACH HEAD ... over head_dim"**: QK-norm does not normalize the whole hidden; view q as `[T, H, D]` and compute RMS independently over the last dim D. All heads share the same `q_norm` weight `[D]` (likewise k uses `k_norm`). eps is `w.eps`.
- **Order: proj → QK-norm → RoPE**. RoPE then norm is numerically different (RMS is rotation-invariant, but the per-dim weight is not), and parity checks will fail.
- **"rows belong to the NEW tokens"**: the caller has already computed cos/sin by absolute position and passes in only the new tokens' rows; the function does not need to, and should not, compute positions itself. The cache stores k **already RoPE'd** (and v as-is).
- **KVCache layout**: `k` and `v` are lists of chunks, each `[t, KVH, D]`; `len(cache)` is the sum of t over all chunks. So "past = len(cache)" must be read before you append.
- **Causal by absolute position**: let past = cache length before appending; new token i has absolute position p = past + i and sees keys 0..p. For a full prefill (past=0) this reduces to the ordinary lower triangle; for decode (T=1) the new token sees all S keys.

© 2026 kzhou003

- **Output**: the attention result `[T, H, D]` is first reshaped to `[T, H·D]` (head dim first, D last, symmetric with how q was viewed), then `@ o_proj.T` gives `[T, hidden]`.

Level 2 · Approach & pitfalls

- **Step breakdown**: (1) three projections + view; (2) a small per-head RMSNorm helper (reuse the M01 idea: normalize in fp32 → cast back → multiply weight); (3) rotate_half + RoPE helper, noting cos/sin must broadcast over the head dim; (4) cache handling to get past and full K/V; (5) GQA expansion; (6) scores + mask + softmax + weighted sum; (7) reshape + o_proj.
- **Cache handling order**: first read `past = len(cache)`, then append this call's k/v, then `torch.cat` all chunks to get `[S, KVH, D]`. If you append first and then read len, past is over-counted by T, the whole mask shifts, and the first new token can "peek" at later tokens. When cache is None, past = 0 and K/V are just this call's k/v.
- **GQA expansion**: `k.repeat_interleave(H // KVH, dim=1)` turns `[S, KVH, D]` into `[S, H, D]` with the 0,0,1,1 mapping. `repeat` / `expand` +wrong reshape gives 0,1,0,1, which is invisible when H == KVH but completely wrong under real GQA.
  GQA mapping (symbolic example): `num_heads = 8, num_kv_heads = 2` → 4 Q heads per group:

q head h	0	1	2	3	4	5	6	7
kv head h // 4	0	0	0	0	1	1	1	1

  (Not the alternating 0,1,0,1,... of `h % 2`.)
- **Computing scores**: clearest via einsum: `scores[h, t, s] = Σ_d q[t,h,d]·K[s,h,d]`, i.e. `einsum("thd,shd->hts")`, then multiply by `D ** -0.5`. You can also transpose to `[H, T, D]` and `[H, S, D]` and use `@`. Do it in fp32 (call `.float()` on q and K first).
- **Building the mask**: `q_pos = arange(past, past+T)[:, None]`, `k_pos = arange(S)[None, :]`; fill `-inf` where `k_pos > q_pos`, shape `[T, S]`, which broadcasts automatically to `[H, T, S]`. Since every query sees at least key 0, there is never an all -inf row causing NaN.
- **Softmax and weighting**: `softmax(scores, dim=-1)` (over the key dim S), then multiply with V: `einsum("hts,shd->thd")`, then `.to(x.dtype)`. Common bug: softmax over dim=-2 (each column sums to 1 instead of each row); in the toy, token0's output would no longer be v0.
- **Self-testing that three call patterns agree**: take a random T=6 input and run (a) one call; (b) one cache, two calls of 3+3; (c) one cache, six T=1 calls; after cat, all three should be allclose. Pass cos/sin with the same slicing.

Level 3 · Walkthrough (code: see the commented solutions in Appendix C)

Key question answers:

- `(x @ q_proj.T).view(T, H, D)`: q_proj's rows are laid out in per-head blocks (the first D rows are head 0), so after the view the last dim's D contiguous values belong to the same head.
- Over the last dim D; `[D]` broadcasts automatically to `[T, H, D]`; v is not normed.
- No: the cache stores k that was already rotated with the correct absolute position at the time; positions never change, so it never needs recomputing.
- Absolute position = past + i; visible key indices 0..past+i (inclusive).
- `repeat_interleave`: H=4, KVH=2 → kv mapping [0,0,1,1] (correct); `repeat` → [0,1,0,1] (wrong).
- q·k, scaling, masking, softmax, and the multiply with V are all in fp32; cast back to x.dtype after multiplying V, then do o_proj.
- Append first, then attend: the new token's key is then in K, the mask allows s == p, and the token can see itself.

Shape trace (H=4, KVH=2, D=8, hidden=32, cached past=5, new T=2):

```
x           [2, 32]
q           [2, 4, 8]     k, v [2, 2, 8]
QK-norm     same shapes
RoPE        q, k with cos/sin [2, 8] -> [2, 1, 8] broadcast
cache cat   K, V [7, 2, 8]            (S = 5 + 2)
GQA         K, V [7, 4, 8]
scores      [4, 2, 7]                  mask [2, 7]:  row0 sees s ≤ 5, row1 sees s ≤ 6
probs @ V   [2, 4, 8] -> reshape [2, 32] -> @ o_proj.T -> [2, 32]
```

Check the toy row by row: token2 (past=2, T=1 decode): q = [1,1], K = [[1.414,0],[0,1.414],[1,1]], scores = [1.414, 1.414, 2]/1.414 = [1, 1, 1.414]; mask: q_pos = 2, k_pos = [0,1,2] all visible; softmax = [0.2846, 0.2846, 0.4307]; output = 0.2846·[1,0] + 0.2846·[0,1] + 0.4307·[3,3] = [1.5768, 1.5768]. If you get `[1, 0]`, q_pos used 0; if the first row is not `[1, 0]`, the mask is missing or softmax is on the wrong dim.
Link to serving: "past = len(cache), new token absolute position = past + i" is exactly the information carried by `seq_lens` / `query_start_loc` in vLLM; paged attention just replaces `torch.cat(cache.k)` with reads via the block table, and the mask logic is unchanged.

© 2026 kzhou003

A-3 · Sparse MoE: routing, per-expert grouped compute, shared expert

Level 1 · I/O examples

route: E = 4, H = 4, `router = eye(4)` (so logits equal x), `top_k = 2`.

```
x = [[2.0, 1.0, 0.0, -1.0],
     [0.0, 0.5, 3.0, -0.5]]
softmax over ALL 4 experts:
     [[0.6439, 0.2369, 0.0871, 0.0321],
      [0.0428, 0.0706, 0.8605, 0.0260]]
top-2 ids     = [[0, 1], [2, 1]]
norm_topk_prob=True  -> weights [[0.7311, 0.2689], [0.9241, 0.0759]]   # 0.6439/(0.6439+0.2369) ...
norm_topk_prob=False -> weights [[0.6439, 0.2369], [0.8605, 0.0706]]
```

The returned weights are in x.dtype (e.g. bf16 when x is bf16); ids are int64.

Per-expert grouping bookkeeping: N = 3, k = 2, E = 5; suppose routing gives `ids = [[2, 0], [1, 2], [2, 3]]`.

```
flat expert id (N·k = 6):  [2, 0, 1, 2, 2, 3]
flat token id:             [0, 0, 1, 1, 2, 2]
stable argsort by expert:  [1, 2, 0, 3, 4, 5]
per-expert counts:         [1, 1, 3, 1, 0]      # expert 4 receives 0 tokens -> skipped entirely
expert 0 -> pairs [1]        -> tokens [0]
expert 1 -> pairs [2]        -> tokens [1]
expert 2 -> pairs [0, 3, 4]  -> tokens [0, 1, 2]   # one matmul handles 3 rows
expert 3 -> pairs [5]        -> tokens [2]
```

moe_forward end to end (scalar version): H = 1, per-expert intermediate dim I = 1, E = 3, top_k = 2, norm_topk_prob = True.

```
router = [[1], [0], [-1]]
w_gate = [1, 2, 0.5],  w_up = [1, 1, 2],  w_down = [1, -1, 3]   # each is [E, 1, 1]
x = [[2.0], [-1.0]]

Expert_e(h) = silu(g·h) · (u·h) · d
         e=0        e=1        e=2
x=2:   3.5232    -7.8561     8.7727
x=-1:  0.2689    -0.2384     1.1326

token0: logits [2, 0, -2] -> ids [0, 1], w [0.8808, 0.1192]
        out = 0.8808·3.5232 + 0.1192·(-7.8561) = 2.1667
token1: logits [-1, 0, 1] -> ids [2, 1], w [0.7311, 0.2689]
        out = 0.7311·1.1326 + 0.2689·(-0.2384) = 0.7639
moe_forward(x, w) = [[2.1667], [0.7639]]
```

Add a shared expert (gate_proj = up_proj = down_proj = [[1]], `gate = [[0]]` → sigmoid = 0.5): SharedExpert(x) = silu(x)·x = [3.5232, 0.2689], and the result becomes `[[3.9283], [0.8984]]` (= 2.1667 + 0.5·3.5232, 0.7639 + 0.5·0.2689).

Key questions

1. Is softmax taken over all E experts and then top-k, or top-k first and softmax over the k? Are the results the same with norm_topk_prob=True? With False?
2. What dtype should the weights have after top-k? At which step do you convert?
3. With N tokens and top_k slots each, how many (token, slot) pairs are there after flattening? How do you build the "which token does each pair belong to" vector so it lines up one-to-one with `ids.reshape(-1)`?
4. After sorting, how do you know where expert e's segment starts and ends in the sorted array?
5. A token's two slots may land on different experts; how are the two results added back into the same row? What happens if you write back all (token, slot) pairs at once with `out[all_toks] += y_all`?
6. `w_gate` has shape `[E, I, H]`; for a batch `h[n_e, H]`, should the gate projection multiply by `w_gate[e]` or its transpose?
7. Is the shared expert's gate a scalar or a vector? What is the shape of `sigmoid(x @ gate.T)`, and how does it multiply the `[N, H]` SharedExpert output?

Details

- **"softmax … over ALL experts in fp32 → top-k → renormalize"**: probabilities are first normalized over all E experts, then the largest k are picked; if `norm_topk_prob`, divide those k by their sum so they sum to 1. Without renormalization the k weights sum to less than 1 (0.6439+0.2369 = 0.8808 in the example above).
- **dtype convention**: routing probabilities are computed in fp32 (call `.float()` on logits first or use softmax's `dtype=` argument), renorm is also in fp32, and finally weights are cast back to `x.dtype`. Expert ids are the int64 indices returned by topk.
- **SwiGLU expert**: `Expert(h) = (silu(h Wg^T) * (h Wu^T)) Wd^T`, with Wg/Wu `[I, H]` and Wd `[H, I]`, so `h[n, H] @ Wg.T → [n, I]`, elementwise product, then `@ Wd.T → [n, H]`. When stacked, use `w_gate[e]` to take

© 2026 kzhou003

expert e.

- **"the serving way"**: not "loop k times per token", and not "run every expert on all N tokens and select with a mask", but each expert computes **once** on just the batch of tokens routed to it; experts with no tokens do no work. This is the grouped-GEMM idea behind FusedMoE.
- **scatter-add**: each expert's output is first multiplied by the routing weight of the corresponding (token, slot), then added back into `out` by token index. A token appears at most once within one expert (top-k ids are distinct) but multiple times across experts, so it must be an add, using an accumulate API that supports duplicate indices.
- **shared expert (talker)**: every token goes through a plain SwiGLU (`gate_proj/up_proj/down_proj`), multiplied by a per-token scalar gate `sigmoid(x @ gate.T)` (`[N, 1]`), then added to the routed result. Skip it when `w.shared is None` (thinker).

Level 2 · Approach & pitfalls

- **route** in three lines: logits → fp32 softmax (dim=-1) → `torch.topk(probs, k, dim=-1)` ; optional renorm (sum with `keepdim=True`); cast weights back to x.dtype before returning. Pitfall: topk on logits first then softmax (mathematically equivalent when norm_topk_prob=True, but completely different when False, and the weight magnitudes change).
- **Flattening**: `flat_e = ids.reshape(-1)` , `flat_w = weights.reshape(-1)` , `flat_tok = arange(N).repeat_interleave(k)` . All three have length N·k, and element p describes the same (token, slot) pair. Pitfall: `arange(N).repeat(k)` gives 0,1,2,0,1,2, which does not match the row-major reshape.
- **Grouping**: `order = argsort(flat_e, stable=True)` puts each expert's pairs together; `counts = bincount(flat_e, minlength=E)` gives each segment's length; accumulate a cursor start to get each segment `order[start:start+n]` . You can also take indices per expert with `(flat_e == e).nonzero()` : more intuitive, but scans the whole array for every expert.
- **Per-expert compute**: `toks = flat_tok[sel]` , `h = x[toks]` (`[n_e, H]`), SwiGLU gives `y[n_e, H]` , then multiply by `flat_w[sel][:, None]` . Note the weights' dtype matches x (route already cast it back).
- **scatter-add**: `out = zeros_like(x)` , `out.index_add_(0, toks, y_weighted)` . Pitfall: `out[toks] += y` keeps only the last write when toks has duplicates; within a single expert toks has no duplicates so it "looks" right, but if you do one advanced-index assignment over all pairs you lose results.
- **Skip empty experts**: `if n == 0: continue` . This is both a performance requirement and avoids pointless matmuls on empty tensors. The cursor start need not advance when n == 0 (adding 0 is harmless anyway).
- **shared expert**: compute SwiGLU on all N tokens, multiply by `torch.sigmoid(x @ shared.gate.T)` (`[N, 1]` , broadcasts along H automatically), add to out.
- **Batch-invariance self-test**: random x[6, H]; compare `moe_forward(x)[i]` with `moe_forward(x[i:i+1])[0]` . A mismatch usually means flat_tok is built wrong or the scatter dropped accumulation.

Level 3 · Walkthrough (code: see the commented solutions in Appendix C)

Key question answers:

- Full softmax first, then top-k. With norm_topk_prob=True, softmax-then-renorm equals softmax over the top-k logits (the denominators cancel); with False they differ: the top-k weights from the full softmax sum to < 1.
- x.dtype; convert after renorm, before return.
- N·k; `arange(N).repeat_interleave(k)` → 0,0,1,1,2,2 (k=2).
- bincount gives each expert's count; the running sum of counts in expert order gives each segment's start (prefix sum).
- `index_add_` (or `scatter_add_`). `out[idx] += y` is read-add-write; with duplicate idx in one call, later writes overwrite earlier ones. In the per-expert loop toks happen to be distinct so it works, but once you write all (token, slot) pairs back at once you lose contributions; `index_add_` is correct in both cases.
- `h @ w_gate[e].T` : `[n_e, H] @ [H, I] → [n_e, I]` . down: `[n_e, I] @ w_down[e].T` (`[I, H]`) → `[n_e, H]` .
- One scalar per token: `x @ gate.T` is `[N, 1]` ; after sigmoid it broadcasts directly against `[N, H]` .

Shape trace (N=5, H=16, E=8, I=32, k=2):

```
logits    x[5,16] @ router.T[16,8]      -> [5, 8]
probs     softmax fp32                   -> [5, 8]
topk      weights [5, 2], ids [5, 2]
flatten   flat_e / flat_w / flat_tok     -> [10] each
order     argsort(flat_e, stable)        -> [10]
counts    bincount(minlength=8)          -> [8]   (sum = 10)
expert e  h = x[toks]                    -> [n_e, 16]
          gate/up                        -> [n_e, 32]
          down                           -> [n_e, 16]
scatter   out.index_add_(0, toks, ·)     -> out [5, 16]
shared    sigmoid(x @ gate.T) [5,1] * SwiGLU_s(x) [5,16]
```

Check with the scalar example: token0 (x=2) routes to experts 0, 1 with weights 0.8808 / 0.1192; token1 (x=-1) routes to experts 2, 1 with weights 0.7311 / 0.2689. After grouping, expert 0 handles [token0], expert 1 handles [token0, token1], expert 2 handles [token1]. Expert 1's single matmul outputs `[-7.8561, -0.2384]` , multiplied by weights `[0.1192, 0.2689]` and

© 2026 kzhou003

index_add'ed back into rows 0 and 1. Final `[[2.1667], [0.7639]]`. If token0 gets `-0.9365` (= 0.1192·(-7.8561), only the contribution of the last-processed expert 1), you used assignment `out[toks] = y` instead of accumulation.
Link to serving: argsort + bincount corresponds to `moe_align_block_size` in vLLM FusedMoE (align tokens by expert and pad to blocks); each per-expert matmul in the loop is one group of the grouped GEMM; under Expert Parallel, `pair_token` / `pair_expert` are exactly the routing table for the all-to-all dispatch.

A-4 · Multimodal merge: placeholder counts, embedding merge, DeepStack

Level 1 · I/O examples
audio_token_count:

num_mel_frames n	full windows n // 100	remainder r	f → second → third	total
0	0	0	0 → 0 → 0	**0**
50	0	50	25 → 13 → 7	**7**
99	0	99	50 → 25 → 13	**13**
100	1	0	0 → 0 → 0	**13**
101	1	1	1 → 1 → 1	**14**
330	3	30	15 → 8 → 4	**43**

With a LongTensor input it computes elementwise: `audio_token_count(torch.tensor([50, 100, 330]))` → `tensor([7, 13, 43])`. Note 99 and 100 both give 13, and 101 jumps to 14.
merge_multimodal: H = 2, image placeholder id = 90, audio placeholder id = 91.

```
input_ids   = [7, 90, 90, 8, 91, 91, 9]
text_embeds = [[0,1],[2,3],[4,5],[6,7],[8,9],[10,11],[12,13]]
mm_embeds   = {"image": [[100,100],[200,200]], "audio": [[-1,-1],[-2,-2]]}

inputs_embeds = [[  0,   1],
                 [100, 100],   # image #0
                 [200, 200],   # image #1
                 [  6,   7],
                 [ -1,  -1],   # audio #0
                 [ -2,  -2],   # audio #1
                 [ 12,  13]]
masks = {"image": [F, T, T, F, F, F, F],
         "audio": [F, F, F, F, T, T, F]}
```

After the call, `text_embeds` must still hold its original values. For text-only input (no placeholders, `mm_embeds = {}`), the output is a copy of text_embeds and every modality's mask is all False.
Error cases:

Input	Result
`ids = [90, 90, 5]`, `mm_embeds = {"image": 1 row}`	`ValueError` (2 placeholders vs 1 embedding row)
`ids = [91, 5]`, `placeholder_ids = {"audio": 91}`, `mm_embeds = {}`	`ValueError` (placeholders present but no embedding)
`ids = [7, 8]`, `mm_embeds = {}`	Returns normally (no placeholders, a missing key is legal)

apply_deepstack / run_layers:

```
apply_deepstack(hidden=zeros(4, 2), visual_mask=[F, T, T, F], visual_embeds=[[1,1],[2,2]])
  -> [[0,0],[1,1],[2,2],[0,0]]       # hidden itself is still all zeros

layers = [lambda t: t*2, lambda t: t+10, lambda t: t*3]
hidden = ones(3, 1);  visual_mask = [T, F, T]
deepstack_embeds = [[[1],[5]], [[100],[200]]]       # 2 levels, each [n_vis=2, 1]

row:          0      1      2
layer0 (*2):  2      2      2
+ level 0:    3      2      7
layer1 (+10): 13     12     17
```


© 2026 kzhou003

```
+ level 1:    113    12     217
layer2 (*3):  339    36     651      # only 2 levels, nothing injected after layer2
run_layers(...) = [[339], [36], [651]]
```

If you mistakenly inject **before** the layer, you get `[[342], [36], [666]]`: non-visual rows are unaffected, so checking only text rows will never catch this bug.

Key questions

1. audio_token_count must support both Python int and LongTensor. Which constructs (e.g. `math.floor`, `int(...)`, `if` branches) break on tensor input? Which operators alone are enough?
2. When r = 0 (n is an exact multiple of 100), what does `(r - 1) // 2 + 1` give? Which behavior of Python/PyTorch `//` on negative numbers does this rely on?
3. For one modality, what does replacing "in order" mean? Which row of `mm_embeds[modality]` should the k-th placeholder position take? Does boolean-mask assignment `out[mask] = emb` guarantee that order?
4. How do you ensure text_embeds is not modified? What happens if you do `out = text_embeds` and then `out[mask] = ...`?
5. If a modality is in placeholder_ids but not in mm_embeds, when should you raise and when should you silently skip?
6. How does the row count of `visual_embeds` relate to `visual_mask`? What is the shape of `hidden[visual_mask]`?
7. With L layers and K DeepStack levels (K < L), after which layers is injection done? How does level i map to layer i?

Details

- **Where the audio formula comes from**: the AuT encoder cuts mel frames into 100-frame windows, and three stride-2 convolutions each map length to `(len - 1) // 2 + 1` (i.e. ceil(len/2)): 100 frames → 50 → 25 → 13. Each full window contributes 13 directly; only the remainder window actually goes through the formula. The frontend (processor) must use the same formula to decide how many `<audio>` placeholders to insert.
- **int / tensor duality**: `%`, `//`, `+`, `-`, `*` all follow elementwise integer semantics for both Python int and LongTensor, so an expression using only these operators naturally supports both inputs. For negative numbers, both implement `//` as floor, so `(-1) // 2 = -1`.
- **Semantics of masks**: return `{modality: bool[T]}` with an entry for **every** modality in placeholder_ids (an all-False mask even if that modality doesn't appear). The visual_mask later used by DeepStack comes from here.
- **"in order"**: the encoder produces embeddings in the order the placeholders appear in the sequence. Boolean-index assignment `out[mask] = emb` writes rows 0, 1, 2... of emb into the True positions in ascending order, which is exactly "in order". Placeholders of different modalities may interleave (e.g. image, audio, image); each modality counts on its own.
- **dtype**: encoder output may have a different dtype from text_embeds (e.g. ViT in fp32, LLM in bf16); cast to the output dtype before writing. Same for the DeepStack add.
- **DeepStack is "add", not "replace"**, and returns a new tensor (`apply_deepstack` must not modify the hidden passed in). In `run_layers`, level i is injected **after** layer i; extra layers (i ≥ len(deepstack_embeds)) just do a plain forward.

Level 2 · Approach & pitfalls

- **audio_token_count**: a single expression suffices; three steps reads more clearly: `r = n % 100` → three rounds of `(len - 1) // 2 + 1` → add `13 * (n // 100)`. Don't write branches like `if r == 0` (unusable on tensors, and the formula already gives 0 correctly).
- **Loop structure of merge_multimodal**: start with `out = text_embeds.clone()`; for each (modality, token_id) in `placeholder_ids`: compute the mask, store it in masks, count n = mask.sum(); then check whether mm_embeds has this modality, validate, and write. Drive the main loop by placeholder_ids rather than mm_embeds, so "placeholders present but embedding missing" is covered naturally.
- **Validation**: (a) modality missing: n > 0 → ValueError, n == 0 → skip; (b) modality present: `emb.shape[0] != n` → ValueError. Putting the modality name and both counts in the error message is very useful for debugging in production.
- **Writing**: `out[mask] = emb.to(out.dtype)`. Pitfall: `out[mask.nonzero()] = emb` adds an extra dim; `torch.where(mask[:, None], emb, out)` has mismatched shapes (emb has only n rows).
- **Immutability**: `clone()` must come before any write. With `out = text_embeds`, every write modifies the caller's tensor, which the caller may reuse in the next request or in a CUDA graph.
- **apply_deepstack**: `out = hidden.clone()`, `out[visual_mask] = out[visual_mask] + visual_embeds` (or `index_add_` with nonzero indices). Pitfall: `hidden[visual_mask] += ...` modifies the input in place; `out += visual_embeds` has mismatched shapes (n_vis rows vs T rows).
- **run_layers**: `for i, layer in enumerate(layers)`: first `hidden = layer(hidden)`, then inject level i when `i < len(deepstack_embeds)`. Pitfall: injecting before the layer call (off-by-one; HF parity fails yet everything outside the visual rows matches exactly); or zip(layers, deepstack) causing the extra layers to be skipped.

Level 3 · Walkthrough (code: see the commented solutions in Appendix C)

Key question answers:

- `math.floor`, `int(n)`, `if r == 0` all fail on multi-element tensors; `% // + - *` alone are enough.
- r = 0: `(0 - 1) // 2 + 1 = -1 + 1 = 0`, and the next two steps also give 0; this relies on floor division (rounding toward negative infinity). C-style truncating division would give 1, and 100 frames would be miscounted as 14.

© 2026 kzhou003

- The k-th True position takes row k; boolean-index assignment writes in ascending position order, so yes.
- `clone()`; without it `out` and text_embeds share memory and writes are in-place modifications.
- Missing embedding and the modality has ≥1 placeholder → raise; 0 placeholders → skip (still return the mask).
- `visual_embeds.shape[0] == visual_mask.sum()`; `hidden[visual_mask]` is `[n_vis, H]`.
- One injection after each of layers 0 … K-1; level i ↔ layer i (0-based).

audio_token_count skeleton (The `((f - 1) // 2 + 1 - 1) // 2 + 1` in the docstring is just the b and c steps written together.)
merge_multimodal skeleton Check with the io example: the image mask's True positions are 1, 2, so row 1 ← image[0] = [100,100], row 2 ← image[1] = [200,200]; likewise audio at 4, 5.
apply_deepstack / run_layers skeleton Check the io example row by row: row 0: 1 →(*2) 2 →(+1) 3 →(+10) 13 →(+100) 113 →(*3) 339; row 1 is not a visual row: 1 → 2 → 12 → 36.
Link to serving: `audio_token_count` decides the prompt length in the API process (HF processor), while `merge_multimodal` consumes encoder output in the thinker stage. When the two formulas disagree, the error surfaces at merge's count check, which is why the check must be strict rather than papering over it with truncation or zero-padding. DeepStack's visual_mask is exactly the image (and video) mask returned by merge.

A-5 · Talker MTP: predicting 15 residual codebooks within one frame

Level 1 · I/O examples
Example 1: G=4 (1 CB0 + 3 residual codebooks), B=2, H=16, vocab=10 (tiny code predictor with random weights, 1 layer).

```
last_talker_hidden: [2, 16]   cb0_embed: [2, 16]
predict_residual_codes(cp, hid, cb0)    -> tensor([[4, 5, 6],
                                                   [7, 4, 9]])   # [B, G-1] = [2, 3]
predict_residual_codes_kv(cp, hid, cb0) -> tensor([[4, 5, 6],
                                                   [7, 4, 9]])   # must be elementwise identical
```

Column 0 is CB1 (predicted with `lm_head[0]`), column 1 is CB2 (`lm_head[1]`), column 2 is CB3 (`lm_head[2]`). Running row 1 alone (inputs `hid[1:2], cb0[1:2]`) gives `[[7, 4, 9]]`: rows in a batch do not affect each other.
Example 2: sequence length at each step of the re-prefill version (same G=4, B=2 model)

j	inputs_embeds fed to cp.model	row taken	logits	result	then append
0	[2, 2, 16] = [h_talker, E0(cb0)]	[:, -1]	lm_head[0] → [2, 10]	CB1 = [4, 7]	codec_embedding[0](CB1)
1	[2, 3, 16]	[:, -1]	lm_head[1] → [2, 10]	CB2 = [5, 4]	codec_embedding[1](CB2)
2	[2, 4, 16]	[:, -1]	lm_head[2] → [2, 10]	CB3 = [6, 9]	(loop ends; appending is optional)

The output concatenates each step's code as a column: `[[4,5,6],[7,4,9]]`. In the KV version, after the same 3 steps the cache length goes 2 → 3 → 4.
Example 3: shape and meaning of next_talker_input (G=4, B=2, H=16)

```
cb0_embed:       [2, 16]       # talker codec embedding of this frame's CB0 (given by caller)
residual_codes:  [2, 3]        # [[4,5,6],[7,4,9]] from Example 1
text_step:       [2, 16]       # thinker text vector aligned to this step
returns [2, 16] = cb0_embed
             + codec_embedding[0](residual_codes[:, 0])   # row 0 looks up index 4, row 1 index 7
             + codec_embedding[1](residual_codes[:, 1])   # looks up 5 / 4
             + codec_embedding[2](residual_codes[:, 2])   # looks up 6 / 9
             + text_step
```

G=4 codec embeddings are summed in total (1 CB0 + G-1 residuals), plus the text vector; the result is still `[B, H]`, with no sequence dimension.

Key questions

1. When predicting codebook j+1, which index of `lm_head` and which index of `codec_embedding` do you use? Why are the two indices the same rather than off by 1?
2. The initial sequence `[last_talker_hidden, cb0_embed]` is two `[B, H]` tensors; to turn them into `[B, 2, H]` should you use `stack` or `cat`? Along which dimension?
3. Each step the transformer outputs `[B, L, H]`; which position do you use to compute logits? Why not position 0?
4. After the last of the G-1 codes (CB_{G-1}) is computed, does the re-prefill version need to run the transformer again? Does next_talker_input need its embedding?
5. KV version: how many positions do you feed to `cp.model` on the first call? On the second? What happens if on the second call you still feed the full sequence into the cached model?
6. In the KV version, who increments the position id of new positions? Do you need to pass position_ids manually? (Hint: think about what HF's `DynamicCache` records.)

© 2026 kzhou003

7. How many embeddings are summed in next_talker_input? Why does the CB0 term use the passed-in `cb0_embed` directly instead of looking it up in the code predictor's table?

Details

- **G = num_code_groups** is the total number of codebooks per frame (16 in the real model, including CB0). The big talker model only samples CB0; the code predictor handles only the remaining **G-1**, so `lm_head` and `codec_embedding` are both `ModuleList`s of length G-1, with index j=0 corresponding to CB1.
- **re-prefill (no cache)**: each step hands the **full current sequence** to `cp.model(inputs_embeds=...)` without passing `past_key_values`. Because attention is causal, the last position's output is mathematically the same as incremental decoding; vllm-omni chooses this because the sequence is at most G+1 long and fixed shapes are convenient for CUDA graphs.
- **Greedy**: `code = logits.argmax(-1)`, a long tensor of shape `[B]` that can be used directly as an index into `nn.Embedding`. Real serving uses top-k/top-p sampling; this exercise uses argmax so results can be compared exactly.
- **codec_embedding[j]** is the code predictor's own embedding table (`[vocab, H]`), not the same table as CB0's in the main talker model. CB0's embedding is computed by the caller and given to you as `cb0_embed`.
- **DynamicCache** (`from transformers import DynamicCache`): create an empty object, pass it as `past_key_values=` with `use_cache=True`, and the model appends each layer's K/V into it in place; it also knows how many positions are cached, so RoPE positions continue automatically.
- **next_talker_input** is the input embedding for the talker's **next step**: "sum of the embeddings of all G codebooks of this frame + the aligned text vector". This is the Qwen3-Omni talker's "multi-codebook summed input" convention; returns `[B, H]`.

Level 2 · Approach & pitfalls

- **Breakdown**: the three functions share one "single step" pattern: *take last-position hidden → lm_head[j] → argmax → embed with codec_embedding[j]*. The two predict functions differ only in "what is fed to the transformer each step": re-prefill feeds the whole sequence, the KV version feeds only the new piece.
- **Data structures**: keep the sequence as a `[B, L, H]` tensor and append `[B, 1, H]` along the length dim each step with `torch.cat(..., dim=1)`; collect codes in a Python list and finally `torch.stack(codes, dim=1)` into `[B, G-1]`. Note the embedding output is `[B, H]`; add the length dim before appending (`[:, None]` or `unsqueeze(1)`).
- **Loop count**: `for j in range(G - 1)`, producing exactly one code per iteration. Do not write `range(G)` (out-of-bounds access to `lm_head[G-1]`) or `range(G-2)` (one code short, output becomes `[B, G-2]`).
- **Input switching in the KV version**: use a variable `x` for "the new positions to feed this step". The first time `x` is the `[B, 2, H]` initial sequence; afterwards `x` becomes the embedding of the code just produced, `[B, 1, H]`. Create the cache only once, outside the loop.
- **Common bug 1: using lm_head[0] at every step**. The first code is correct, later ones go wrong. In the G=4 example above, the result changes from `[[4,5,6],[7,4,9]]` to `[[4,5,4],[7,4,2]]`: only the first column is reliably correct, the telltale symptom of this bug.
- **Common bug 2: the KV version still feeds the full sequence each step**. The cache stores old positions again and position ids get pushed out, so results drift silently: the same example gives `[[4,5,6],[7,4,2]]`, some rows right and some wrong, very hard to track down. Check: after each step the cache length should be 2, 3, 4, ...
- **no_grad**: this is inference; wrap it in `@torch.no_grad()` or `with torch.no_grad():`, otherwise a useless autograd graph is built (same result, but slower and more memory).
- **Pitfalls in next_talker_input**: column j of the residual codes must be looked up with `codec_embedding[j]` (the same j as in prediction); do not modify the passed-in `cb0_embed` in place (use `a = a + b` rather than `a += b`, or `clone()` first), otherwise you corrupt the caller's tensor.

Level 3 · Walkthrough (code: see the commented solutions in Appendix C)

Answers to the key questions

- Predict CB(j+1) with `lm_head[j]`; embed the resulting code with `codec_embedding[j]`. Both tables cover only the residual codebooks (length G-1), and index 0 refers to CB1 in both, hence the same j.
- `torch.stack([last_talker_hidden, cb0_embed], dim=1)` → `[B, 2, H]`. `cat` would give `[2B, H]` or `[B, 2H]`, both wrong.
- Take `last_hidden_state[:, -1]` (`[B, H]`): a causal LM predicts the "next" token from the last position.
- After the last code is computed there is no need to run the transformer again; but next_talker_input needs its embedding (via `codec_embedding[G-2]`).
- The KV version feeds 2 positions the first time, then 1 each time. Re-feeding the whole sequence puts duplicate K/V in the cache and the positions are wrong too.
- No need to pass position_ids manually: `DynamicCache` records the cached length, and the HF model derives the new positions' `cache_position` from it.
- G codec embeddings in total (CB0 + G-1 residuals) plus text_step. CB0's embedding comes from the main talker model's table; the code predictor's tables have no CB0, so the caller passes it in.

Shape trace (G=4, B=2, H=16, vocab=10)

© 2026 kzhou003

```
seq0 = stack([hid, cb0], 1)            [2, 2, 16]
j=0: cp.model(seq0).last_hidden_state  [2, 2, 16] -> [:, -1] [2, 16]
     lm_head[0](.)                     [2, 10] -> argmax(-1) [2]   = [4, 7]
     codec_embedding[0](code)          [2, 16] -> [:, None] [2, 1, 16]
     seq1 = cat([seq0, e], 1)          [2, 3, 16]
j=1: ...                               [2, 4, 16]       code = [5, 4]
j=2: ...                                                code = [6, 9]
stack([c0, c1, c2], dim=1)             [2, 3]  = [[4,5,6],[7,4,9]]
```

predict_residual_codes_kv skeleton Self-check: print `cache.get_seq_length()` inside the loop; for G=4 it should be 2, 3, 4 in turn; both versions should be `torch.equal` on the same inputs.
`residual_codes[:, j]` is `[B]`, and the lookup gives `[B, H]`; do not throw the whole `[B, G-1]` into one table (that gives `[B, G-1, H]` with every column using the same table).
Why re-prefill and the KV version must agree: under causal attention, the output at position p depends only on inputs at positions 0..p. Re-prefill recomputes all old positions each step, but their K/V are exactly the same as in the previous step; the KV version just stores them. So as long as RoPE positions match (DynamicCache guarantees this automatically), the last-position hidden values are nearly identical and the argmax is the same.

A-6 · Code2Wav streaming causal conv and transposed conv

Level 1 · I/O examples
Example 1: causal Conv1d, K=3, d=1, C_in=C_out=1, weight=[1,2,3], bias=0

```
x = [1, 2, 3, 4, 5]                # [1, 1, 5]
offline: left-pad (K-1)*d = 2 zeros → [0, 0, 1, 2, 3, 4, 5]
      y[t] = 1·x[t-2] + 2·x[t-1] + 3·x[t]
      y = [3, 8, 14, 20, 26]

streaming, chunks [2, 3]:
step(x[0:2]=[1,2])   -> [3, 8]          state becomes [1, 2]
step(x[2:5]=[3,4,5]) -> [14, 20, 26]    state becomes [4, 5]
concatenated = [3, 8, 14, 20, 26]  ✓ matches offline
```

Outputs per call = inputs in this chunk; state is always "the most recent (K-1)·d = 2 inputs". Before the first chunk, state is 2 zeros (exactly equivalent to the offline left zero-padding).
Example 2: dilated causal Conv1d, K=2, d=3, weight=[1,1], bias=0

```
y[t] = x[t-3] + x[t], need to keep (K-1)·d = 3 past inputs
x = [1, 2, 3, 4, 5, 6], offline y = [1, 2, 3, 5, 7, 9]

chunks [1, 4, 1]:
step([1])          -> [1]            state = [0, 0, 1]
step([2, 3, 4, 5]) -> [2, 3, 5, 7]   state = [3, 4, 5]
step([6])          -> [9]            state = [4, 5, 6]
```

Note the first chunk has only 1 sample, shorter than state: state must be taken from the tail of the buffer "old state + new chunk", so it still contains two padded zeros.
Example 3: causal ConvTranspose1d, K=4, s=2, weight=[1,2,3,4] ([C_in=1, C_out=1, K]), bias=0.5

```
x = [1, 10, 100] (3 frames)
frame i writes positions [2i, 2i+4), values = x_i · [1,2,3,4]
pos:         0    1    2     3     4      5      6     7
frame0 (1):  1    2    3     4
frame1 (10):           10    20    30     40
frame2 (100):                100    200    300   400
sum+bias:  1.5  2.5  13.5  24.5  130.5  240.5  300.5  400.5   (length (3-1)·2+4 = 8)
offline: trim K-s=2 at the start and 2 at the end → [13.5, 24.5, 130.5, 240.5]

frame-by-frame streaming:
step(frame0) -> []              # positions <2 are final, but they are exactly the 2 left-trimmed ones
step(frame1) -> [13.5, 24.5]    # positions 2,3 final
step(frame2) -> [130.5, 240.5]  # positions 4,5 final; 6,7 still waiting (future frames would write there)
flush()      -> []              # 6,7 remain, exactly the ones trimmed on the right
chunks [2, 1] work the same: step(frame0..1) -> [13.5, 24.5], step(frame2) -> [130.5, 240.5], flush -> []
```

Cumulative output count after frame n = `max(0, n·s - (K-s))` = 0, 2, 4.
What wrong bias handling looks like: if each frame does its own transposed conv with bias and the overlaps are summed, bias is added twice at overlapping positions and the output becomes `[14.0, 25.0, 131.0, 241.0]` (each one 0.5 too high).

© 2026 kzhou003

Example 4: transposed conv with K == s (K=3, s=3, weight=[1,2,3], bias=0)

```
x = [1, 10]: frames don't overlap, K-s = 0, no trimming
offline = [1, 2, 3, 10, 20, 30]
step(frame0) -> [1, 2, 3]   step(frame1) -> [10, 20, 30]   flush() -> []
```

A good example for checking the "trim amount is 0" edge case.

Key questions

1. How many samples must the causal Conv1d state keep? How many when K=1? In that case what does your slice `buf[..., -n:]` return for n=0?
2. What should state be before the first call to `step`? Where do its batch / channel dims come from (B is not known in the constructor)?
3. Transposed conv: after n frames have arrived, outputs before which position will never be modified by any future frame? Why?
4. Global left trim of K-s samples: if every step trims the first K-s samples of its own output, what goes wrong? Think about Example 3 with chunks [2,1].
5. When should bias be added? If you pass bias into each chunk's `conv_transpose1d`, which positions go wrong, and by how much?
6. At flush, what should the total output length be (in terms of frame count L, s, K)? How much is left after the right trim?
7. In `from_hf`, which HF module attributes do dilation and stride come from? Is `nn.ConvTranspose1d.stride` an int or a tuple?

Details

- **Weight layout**: `nn.Conv1d` weight is `[C_out, C_in, K]`; `nn.ConvTranspose1d` weight is `[C_in, C_out, K]` (noted in the docstring). Both can be passed directly to `F.conv1d` / `F.conv_transpose1d`; no need to hand-write the convolution. With no padding, `F.conv1d(..., dilation=d)` output length = input length - (K-1)·d.
- **What "causal" means**: output t depends only on inputs ≤ t. The offline version achieves this by padding (K-1)·d zeros **on the left only**. The streaming version replaces "the zeros on the left" with "the real history left over from the previous chunk"; on the very first call that history is exactly zeros.
- **"Final"**: in a transposed conv, frame i affects [i·s, i·s+K). With n frames received, future frames (index ≥ n) only write positions ≥ n·s, so all positions < n·s are final and can be emitted immediately; this directly determines first-packet audio latency.
- **Global vs in-chunk coordinates**: the left trim applies to the first K-s samples of the **whole output stream** (global positions 0..K-s-1), not the start of each chunk. You need to track "which global position has been emitted so far".
- **flush**: once all frames are in, the total length is (L-1)·s + K; emit the part not yet output and drop the last K-s (the offline module's right trim). When K==s the trim is 0; beware that `t[..., :-0]` gives an empty tensor.
- **from_hf**: HF's `Qwen3OmniMoeCausalConvNet` has `m.conv` (`nn.Conv1d`), `m.dilation`, `m.stride` (only 1 is supported here); for `Qwen3OmniMoeCausalTransConvNet` the stride must be taken from `m.conv.stride` (a tuple, take [0]). Use `.detach()` on weights; bias may be None.

Level 2 · Approach & pitfalls

- **StreamingCausalConv1d state**: you only need `weight, bias, dilation`, `context = (K-1)·d`, and a `state` (initially None; on the first step create all-zeros `[B, C_in, context]` from the input's B, C). step = concatenate → update state → `F.conv1d` with no padding.
- **Order of state update**: first `buf = cat([state, x], -1)`, then take the last context samples of **buf** as the new state, and finally convolve buf. If you take the tail of x, state shrinks when the chunk is shorter than context, and the next chunk's output count is wrong (the first chunk of Example 2 is exactly this case).
- **K=1 edge case**: with context=0, `buf[..., -0:]` is the whole buf (Python's -0 is just 0), so state grows without bound and outputs keep getting longer. Use `buf[..., buf.shape[-1]-context:]` or check context==0 explicitly.
- **StreamingCausalTransConv1d state**: `frames` (frames received), `base` (next global position to emit), `acc` (accumulation buffer of not-yet-final samples starting at base). `trim = K - s`.
- **step flow**: (1) run `F.conv_transpose1d(x, w, None, stride=s)` on this chunk **without bias**, length (t-1)·s+K, covering global positions [frames·s, frames·s + len); (2) if needed, extend acc with zeros on the right, then add the result into the matching range (subtract base from indices); (3) frames += t; (4) emit the range [base, frames·s).
- **Emit function (shared by step and flush)**: slice the first (upto - base) samples off acc, keep the rest in acc; add bias now (once per position); then drop the part whose global positions fall in [0, trim): `skip = max(0, trim - old base)`; finally base = upto.
- **Common bugs and symptoms**: bias inside each chunk's conv → extra bias in overlap regions (in Example 3 every value is 0.5 too high); left trim per chunk → every later chunk is K-s samples short and the total length is wrong; emitting positions ≥ frames·s → those values get added to later, diverging from offline; emitting too little → result may still be correct, but the "emit as early as possible" counts won't match.
- **Chunking invariance**: when several frames are fed at once, a single `conv_transpose1d` already does the overlap-add inside the chunk; overlap between chunks is handled by acc. Combined, they match a single offline computation exactly (linear + no bias).

© 2026 kzhou003

Level 3 · Walkthrough (code: see the commented solutions in Appendix C)

Answers to the key questions

- state keeps (K-1)·d samples; 0 when K=1. `buf[..., -0:]` returns the whole buf, so it must be special-cased.
- Initial state is all-zeros `[B, C_in, context]`, created lazily on the first step with `x.new_zeros(x.shape[0], x.shape[1], context)` (which also inherits dtype/device).
- After n frames, positions < n·s are final: any future frame i ≥ n only writes [i·s, i·s+K), which starts at ≥ n·s.
- Per-chunk trimming: in Example 3 with chunks [2,1], the second chunk's output [130.5, 240.5] would wrongly lose 2 samples and become []. The left trim applies only to global positions 0..K-s-1.
- Add bias once, at the moment a final position is emitted. If added in each chunk's conv, positions where chunks overlap (each position is covered by ceil(K/s) frames; the cross-chunk part) get it added extra times.
- At flush the total length = (L-1)·s + K; removing K-s from each side leaves L·s - (K-s). Example 3: L=3 → 8 - 4 = 4.
- Conv: `m.dilation`, `m.stride`; TransConv: `m.conv.stride[0]` (tuple).

Internal state per frame for Example 3 (K=4, s=2, trim=2; acc holds values **without bias**)

Operation	Global range covered by this chunk's conv	frames	Emit range	Output	base after	acc after
step(frame0)	[0,4) = 1,2,3,4	1	[0,2) all within trim	[]	2	[3, 4] (positions 2,3)
step(frame1)	[2,6) += 10,20,30,40	2	[2,4)	[13.5, 24.5]	4	[30, 40]
step(frame2)	[4,8) += 100..400	3	[4,6)	[130.5, 240.5]	6	[300, 400]
flush()	—	3	[6,8), then drop 2 on the right	[]	8	[]

acc length is always ≤ K-s: it is exactly the tail "still waiting for later frames".

Self-check list

- Causal conv: with the hand-set weights of Examples 1 and 2, chunking as [5], [1,1,1,1,1], [2,3] should all give the same sequence; each chunk's output count == its input count.
- Transposed conv: feeding K=8, s=4 frame by frame, the cumulative output count should be 0, 4, 8, 12, … (= max(0, 4n-4)).
- K==s (Example 4): no trimming, each step outputs s samples per frame, flush is empty.
- cat all step outputs and the flush output together and compare against the HF module's offline forward with `torch.testing.assert_close`.

A-7 · Per-stage weight loading and packing

Level 1 · I/O examples

Example 1: shards_for_stage

```
weight_map = {
  "thinker.model.embed_tokens.weight":       "model-00001.safetensors",
  "thinker.model.layers.0.mlp.gate.weight":  "model-00002.safetensors",
  "talker.model.codec_embedding.weight":     "model-00002.safetensors",
  "talker.code_predictor.lm_head.0.weight":  "model-00003.safetensors",
  "code2wav.decoder.0.conv.weight":          "model-00003.safetensors",
  "thinker.lm_head.weight":                  "model-00004.safetensors",
}
shards_for_stage(weight_map, "thinker")  -> ["model-00001.safetensors", "model-00002.safetensors", "model-
00004.safetensors"]
shards_for_stage(weight_map, "talker")   -> ["model-00002.safetensors", "model-00003.safetensors"]
shards_for_stage(weight_map, "code2wav") -> ["model-00003.safetensors"]
```

The same shard (00002) can belong to two stages; the result is deduplicated and sorted. Note that `thinker.lm_head.*` also belongs to the thinker stage (it is not under `thinker.model.`, but it is under `thinker.`).

Example 2: select_stage

```
state keys:
  "thinker.model.norm.weight"
  "thinker.model.layers.0.input_layernorm.weight"
  "thinker.lm_head.weight"
  "talker.model.norm.weight"
  "thinkers.model.x"           # deliberately constructed "look-alike prefix"
keys of select_stage(state, "thinker", "thinker.model.")
  -> {"layers.0.input_layernorm.weight", "norm.weight"}
```


© 2026 kzhou003

`thinker.lm_head.weight` is not under the `strip` prefix, so it is dropped (M08 fetches it separately); values (tensors) are kept as is, only the keys change.

Example 3: shapes and row order of pack_layer (H=2, KVH=1, D=2, hidden=4, E=2, I=3; each tensor is filled with a constant so its origin is visible)

```
layers.0.self_attn.q_proj.weight  [4, 4] all 1
layers.0.self_attn.k_proj.weight  [2, 4] all 2
layers.0.self_attn.v_proj.weight  [2, 4] all 3
layers.0.mlp.experts.e.gate_proj.weight [3, 4] all 10e+1
layers.0.mlp.experts.e.up_proj.weight   [3, 4] all 10e+2
layers.0.mlp.experts.e.down_proj.weight [4, 3] all 10e+3
layers.0.mlp.gate.weight [2, 4]
(plus input_layernorm / q_norm / k_norm / o_proj / post_attention_layernorm)

pack_layer(sd, 0):
  qkv     [8, 4]     column 0 = [1, 1, 1, 1, 2, 2, 3, 3]         # (H + 2·KVH)·D = 8 rows: q | k | v
  w13     [2, 6, 4]  w13[:, :, 0] = [[1, 1, 1, 2, 2, 2],
                                     [11, 11, 11, 12, 12, 12]]  # per expert: 3 gate rows | 3 up rows
  w2      [2, 4, 3]  w2[:, 0, 0]  = [3, 13]
  router  [2, 4]
```

It doesn't matter if sd also contains `layers.1.*`: pack_layer(sd, 0) only looks at keys under `layers.0.`.

Example 4: cases that must raise LoadError (change one thing in Example 3's sd)

Change	pack_layer(sd, 0)
unchanged	succeeds
add the three weights of expert 3 (ids become 0,1,3)	LoadError (ids not contiguous)
delete `mlp.experts.1.down_proj.weight`	LoadError (expert 1 incomplete)
add `layers.0.self_attn.rotary_emb.inv_freq`	LoadError (unexpected key)
delete `layers.0.self_attn.k_norm.weight`	LoadError (missing key)
add `layers.10.input_layernorm.weight`	succeeds (that is layer 10, not layer 0)

load_thinker_text: an extra `thinker.model.rotary_emb.inv_freq` in state → LoadError; `num_layers` larger than the actual layer count (e.g. only layer 0 exists but you pass 2) → layer 1 has nothing → LoadError; the presence of `thinker.lm_head.weight` and `talker.*` is not an error (not under `thinker.model.`).

Key questions

1. Which prefix should you use to decide which stage a key belongs to? Why is `"thinker"` (no dot) not safe enough?
2. When select_stage strips the prefix, what does `"thinker.model.norm.weight"` become? What happens to `"thinker.lm_head.weight"`?
3. When collecting keys for layer i, why must the prefix be `"layers.{i}."` (with the trailing dot)? For i=1, which layers would be wrongly collected without the dot?
4. How many rows does qkv have (in terms of H, KVH, D)? Why `cat` along dim 0 rather than dim 1?
5. Is w13 built by first doing `cat(gate, up)` per expert and then `stack`, or by first `stack`ing all gates and concatenating with the stacked ups? Do the two approaches give the same shape/content?
6. How do you check that the expert ids are "exactly 0..E-1 and complete"? Where does E come from (it is not a function argument)?
7. How do you detect "unexpected keys"? Which keys do you need to record as used?

Details

- **weight_map** comes from `model.safetensors.index.json`, shaped like `{tensor_name: shard_file}`. A stage only needs to open shards that contain at least one key of that stage; return a **sorted, deduplicated** list of file names.
- **Stage prefix vs strip prefix**: `STAGE_PREFIX` gives the stage-level prefix (e.g. `"thinker."`); select_stage's `strip` is more specific (e.g. `"thinker.model."`): keep only keys that start with it and remove that segment. Use `startswith` + slicing `k[len(strip):]`, not `str.replace` (which may replace a substring in the middle).
- **Row concatenation (dim 0)**: `nn.Linear` weight is `[out, in]`. Merging several Linears that share an input into one means concatenating along the out dim (rows): the output columns of `x @ qkv.T` are q, k, v in order, and you split back with `[H·D, KVH·D, KVH·D]`. Same for gate/up.
- **The 9 fields of PackedLayer** map one-to-one to HF keys: `input_layernorm.weight`, `self_attn.{q,k,v,o}_proj.weight`, `self_attn.{q,k}_norm.weight`, `post_attention_layernorm.weight`, `mlp.gate.weight` (the router; note it is called gate but is not gate_proj), `mlp.experts.{e}.{gate,up,down}_proj.weight`.

© 2026 kzhou003

- **Completeness**: within a layer, a key that is "missing" or "extra" is a LoadError; in load_thinker_text, any key under `thinker.model.` other than `layers.*`, `embed_tokens.weight`, `norm.weight` is also a LoadError. The first returned element is `{"embed_tokens": ..., "norm": ...}` (no `.weight` in the keys).
- **Regex**: `re` is already imported; expert keys have the regular form `mlp.experts.<number>.<gate_proj|up_proj|down_proj>.weight`, well suited to an anchored regex (`^...$`) that extracts the id and projection name.

Level 2 · Approach & pitfalls

- **shards_for_stage**: a one-line set comprehension: iterate `weight_map.items()`, collect the value when the key starts with the stage prefix, finally `sorted(set)`.
- **pack_layer step 1: slice out this layer's sub-dict** `sub = {key with "layers.{i}." removed: tensor}`. All later operations work on sub, where keys become short names like `"self_attn.q_proj.weight"`, which makes checking easy.
- **Bookkeeping: a used set**. Write a small helper `take(name)`: raise LoadError if missing, otherwise add name to used and return the tensor. Expert keys are also added to used when the regex matches. At the end, a non-empty `set(sub) - used` means "unexpected keys".
- **Collecting experts**: `experts: dict[int, dict[str, Tensor]]`; after a regex match, `experts.setdefault(int(id), {})[proj] = v`. Then check: non-empty, `sorted(experts) == list(range(len(experts)))`, and each expert's proj set is exactly `{gate_proj, up_proj, down_proj}`. E is just `len(experts)`.
- **Packing**: `qkv = cat([q, k, v], 0)`; `w13 = stack([cat([gate_e, up_e], 0) for e in range(E)])` → `[E, 2I, hidden]`; `w2 = stack([down_e ...])` → `[E, hidden, I]`. Always go in order e = 0..E-1, not dict iteration order.
- **Ordering trap**: the "unexpected key" check must happen after all takes; otherwise legitimate keys not yet taken are falsely reported as unexpected.
- **load_thinker_text**: `sd = select_stage(state, "thinker", "thinker.model.")` → call pack_layer for `range(num_layers)` → check that the set of keys not starting with `layers.` is exactly `{embed_tokens.weight, norm.weight}` (extra or missing is a LoadError).
- **Common bugs and symptoms**: prefix without dot (`"layers.1"`) → layer 1 swallows the keys of layers.10..19, then gets reported as "unexpected key" or an expert conflict; w13 concatenated as `[up | gate]` → shape is right but SiLU is applied to the wrong half and expert outputs are all wrong; stacking experts in dict order → when ids are out of order, experts are mismatched and the router picks the wrong expert.

Level 3 · Walkthrough (code: see the commented solutions in Appendix C)

Answers to the key questions

- Use `STAGE_PREFIX[stage]` (with a dot, e.g. `"thinker."`). Without the dot, keys like `"thinkers.x"` or `"talker2.x"` are misclassified.
- `"thinker.model.norm.weight"` → `"norm.weight"`; `"thinker.lm_head.weight"` does not start with strip and is dropped.
- `"layers.1"` matches `layers.1x.*` (10–19), `layers.1xx.*`, etc. With the dot, `"layers.1."` matches only layer 1.
- qkv rows = (H + 2·KVH)·D. Linear weights are `[out, in]`; all three share the same in (hidden) with different out, so merging outputs means concatenating along dim 0.
- You must "cat(gate, up) within each expert, then stack", giving `[E, 2I, hidden]`, where the first I rows of each expert are gate and the last I rows are up. Stacking separately and then concatenating along dim 1 actually gives the same result; but concatenating along dim 0 gives `[2E, I, hidden]`, which is wrong.
- E = number of distinct ids collected; the condition is `sorted(ids) == list(range(E))` and E > 0, and every id has all three projs.
- Maintain a used set; after all fields are taken, compute `set(sub) - used`.

Self-check: is the packed form equivalent? (with your own small tensors)

```
x = torch.randn(5, hidden)
q, k, v = (x @ L.qkv.T).split([H*D, KVH*D, KVH*D], dim=-1)
# q should equal x @ q_proj.T; same for k, v
g, u = (x @ L.w13[e].T).chunk(2, dim=-1)
y = (F.silu(g) * u) @ L.w2[e].T      # should equal expert e's SwiGLU output
```

This is exactly how M08 turns a PackedLayer back into the weights M02/M03 need: split qkv by rows, cut w13 in half along dim 1.

A-8 · Putting it together: KV-cache greedy decoding for a tiny thinker

Level 1 · I/O examples

Example 1: arguments of each forward in greedy(prompt=[10, 20, 30, 40], max_new=4)

© 2026 kzhou003

Forward #	ids	start	positions ([3, T], three identical rows)	Returned logits	Per-layer cache length after forward	New token
1 (prefill)	[10,20,30,40]	0	[[0,1,2,3]]×3	[4, V]	4	out[0] = argmax(logits[-1])
2	[out[0]]	4	[[4]]×3	[1, V]	5	out[1]
3	[out[1]]	5	[[5]]×3	[1, V]	6	out[2]
4	[out[2]]	6	[[6]]×3	[1, V]	7	out[3]

max_new=4 needs only 4 forwards: prefill itself produces the first token. The last generated token does not need another forward. Rule: `start = len(prompt) + len(out) − 1`, and before every forward `start == len(cache)`.
Example 2: real numbers on a randomly initialized tiny thinker (vocab=97, hidden=64, 2 layers, H=4, KVH=2, D=16, 6 experts top-2, mrope_section=[4,2,2], random weights)

```
prompt = [10, 20, 30, 40]
forward(tensor(prompt), 0, caches)   -> logits [4, 97], last-row argmax = 34, cache lengths [4, 4]
forward(tensor([34]), 4, caches)     -> logits [1, 97], argmax = 75, cache lengths [5, 5]
greedy(prompt, 6)                    -> [34, 75, 75, 75, 17, 1]
HF full-recompute greedy each step   -> [34, 75, 75, 75, 17, 1]   ✓
```

Two bugs on the same model:

```
start off by +1 in decode (len(prompt)+len(out))    -> [34, 75, 75, 75, 75, 75]   # differs from the 5th token
on
feeding the full ids from start=0 into the same caches each step  -> [34, 77, 2], cache length already 15 after
3 steps
```

Position errors often don't change the argmax immediately; they "diverge" only after several steps, so compare the whole sequence, not just the first token.
Example 3: text-only MRoPE equals plain 1D RoPE

```
positions = arange(4, 6).expand(3, 2)  -> [[4, 5], [4, 5], [4, 5]]
cos, sin = mrope_cos_sin(rope_inv_freq(16, 10000.0), positions, [4, 2, 2], interleaved=True)
cos.shape -> [2, 16]
cos is identical to the cos of plain RoPE using only the t-axis positions [4, 5] (allclose = True)
```

When all three axes have the same position, the result is the same no matter which axis each frequency picks. Still, go through M01's MRoPE interface (interleaved=True) so you stay on the same code path as the multimodal case.

Key questions

1. PackedLayer only has the packed `qkv` and `w13`, while M02's `AttnWeights` wants separate q/k/v and M03's `MoeWeights` wants separate `w_gate` / `w_up`. What sizes do you split by in each case?
2. In `forward(ids, start, caches)`, what is the shape of the positions tensor? What goes into each of the three axes?
3. Do RoPE's cos/sin need to be recomputed on every forward? Do they correspond to the positions of the "new tokens" or of "all tokens in the cache"?
4. M02's attention decides the causal mask from `len(cache)` on its own; what relation must hold between `start` and `len(cache)`? What happens when it doesn't?
5. Which state key does lm_head come from? Why doesn't it go through `load_thinker_text`? Are the logits `h @ lm_head` or `h @ lm_head.T`?
6. How many times does greedy call forward in total (in terms of max_new)? In the k-th decode, which token is fed in and what is start?
7. Where should caches be created: in `__init__`, at the start of `greedy`, or in every forward? Why?

Details

- **Dims** provides every hyperparameter needed for splitting and RoPE: `num_heads` / `num_kv_heads` / `head_dim` determine how qkv is split; `top_k` / `norm_topk_prob` go into MoeWeights; `rope_theta` / `mrope_section` go to M01; `eps` is used for all RMSNorms (including q_norm/k_norm inside attention).
- **"text-only MRoPE positions"**: Qwen3-Omni's RoPE has three axes (t, h, w). For text only, all three axes equal the token's absolute position, i.e. every row of the `[3, T]` tensor is `start..start+T-1`. **interleaved layout** refers to the frequency assignment with `interleaved=True` in M01 (Qwen3 uses this; Qwen2.5 uses chunked).
- **Residual structure (pre-norm)**: `h = h + attn(rms(h, ln1))`, `h = h + moe(rms(h, ln2))`. Normalization applies only to the sublayer input; the residual adds the unnormalized h. After the last layer, do one more `rms(h, norm)` before multiplying by lm_head.
- **One KVCache per layer**: `caches` is a list of length num_layers; layer l only reads and writes `caches[l]`. M02's `attention_forward` appends K/V in place, so forward does not need to return the cache.

© 2026 kzhou003

- **"Must equal HF full-recompute greedy"**: the HF reference recomputes the full sequence from scratch each step and takes the argmax of the last row. Your KV-cache version will have tiny floating-point differences, but the argmax sequence must match exactly; any error in RoPE positions, mask, or normalization order will make the sequence diverge at some step.
- **prefill returns all rows**: forward returns all rows of `[T, V]` (not just the last), so it can also be used to compare prefill logits row by row; greedy uses only `logits[-1]`.

Level 2 · Approach & pitfalls

- **What __init__ does**: (1) `load_thinker_text(state, dims.num_layers)` gives `{embed_tokens, norm}` and a list of PackedLayers; (2) take `state["thinker.lm_head.weight"]`; (3) convert each PackedLayer into M02's `AttnWeights` and M03's `MoeWeights` (along with ln1/ln2) and store them; (4) precompute `inv_freq = rope_inv_freq(head_dim, rope_theta)`.
- **Recovering weights**: `q, k, v = L.qkv.split([H*D, KVH*D, KVH*D], dim=0)`; `I = L.w13.shape[1] // 2`, `w_gate = L.w13[:, :I]`, `w_up = L.w13[:, I:]`, `w_down = L.w2`. Don't use `chunk(2, dim=0)`; that splits along the expert dimension.
- **forward steps**: positions → cos/sin (once, shared by all layers) → `h = embed[ids]` (`[T, hidden]`) → per-layer (attn, moe) residuals → final norm → `@ lm_head.T`. cos/sin correspond only to this call's T new tokens.
- **Contract with M02**: M02 places the new tokens at absolute positions `len(cache) .. len(cache)+T-1` for the causal mask, while RoPE positions are determined by the cos/sin you pass in. The two must agree, i.e. `start == len(caches[0])` on every call. Adding an assert in forward helps debugging.
- **The greedy loop**: create caches at the start of greedy (every greedy call is a new request); one prefill gives the first token; then `while len(out) < max_new`: feed only `out[-1:]`, `start = len(prompt) + len(out) - 1`. Total forwards = max_new.
- **Common bugs and symptoms**: start off by one → diverges only after several steps (Example 2: differs from the 5th token); feeding the whole sequence each step → cache grows with duplicates, garbage soon; using the rms result as the residual trunk (`h = rms(h) + attn(...)`) → prefill logits wrong immediately; w13 halves swapped → MoE output entirely wrong, prefill logits wrong; forgetting the final norm → logits noticeably too large in magnitude.
- **Debugging order**: first check only whether prefill logits are allclose to the reference (checks weights, norm, MoE, RoPE); then check the greedy sequence (checks cache and start). If the former is wrong, the problem is most likely in one of the prerequisite drills M01–M03/M07.
- **dtype/no_grad**: greedy uses `@torch.no_grad()`; `torch.tensor(prompt)` is long and can index embed directly; convert tokens to Python ints with `int(logits[-1].argmax())` before putting them into the list.

Level 3 · Walkthrough (code: see the commented solutions in Appendix C)

Key question answers

- qkv is split by rows into `[H·D, KVH·D, KVH·D]`; w13 is halved on dim 1: the first I rows are gate, the last I rows are up (I = w13.shape[1] // 2).
- positions is a `[3, T]` long tensor whose three rows are all `arange(start, start+T)` (e.g. `torch.arange(start, start+T).expand(3, T)`).
- Recomputed on every forward, and only for the new tokens' positions; the K of old tokens in the cache was already rotated with their own positions before being written to the cache.
- Must have `start == len(cache)`. Otherwise RoPE's relative distances disagree with the positions the mask assumes, the output silently drifts, and often diverges after several steps.
- `state["thinker.lm_head.weight"]`, shape `[V, hidden]`; it is not under `thinker.model.`, so load_thinker_text doesn't handle it. logits = `h @ lm_head.T` → `[T, V]`.
- max_new forwards in total: 1 prefill + (max_new - 1) decodes. The k-th decode (k starting at 1) feeds `out[k-1]`, start = len(prompt) + k - 1.
- Create them at the start of greedy, one per layer; putting them in `__init__` makes the second greedy call inherit the previous history, and putting them in forward is the same as having no cache.

Shape trace (hidden=64, H=4, KVH=2, D=16, V=97, prompt length 4)

```
prefill: ids [4] -> positions [3, 4] -> cos/sin [4, 16]
         h = embed[ids]                      [4, 64]
         rms(h, ln1) -> attention_forward    [4, 64]   cache: 0 -> 4
         rms(h, ln2) -> moe_forward          [4, 64]
         (same for layer 2)
         rms(h, norm) @ lm_head.T            [4, 97]
decode:  ids [1], start=4 -> positions [3, 1] -> cos/sin [1, 16]
         ...                                 [1, 97]   cache: 4 -> 5
```

greedy skeleton Self-check: `assert start == len(caches[0])` inside the loop; also write a slow "full recompute each step" version (new caches each time, feeding prompt+out from start=0); the two token sequences must be identical.

© 2026 kzhou003

A-9 · Orchestrator v1: linear multi-stage routing

Level 1 · I/O examples

Example 1 (the one we keep using): 3 stages S0→S1→S2 (thinker→talker→code2wav), `final_stage_ids={0,2}`, `forward_fn = lambda nxt, prompt, prev: prev` (returns prev_data unchanged).

```
await orch.add_request("r1", "hi there")

S0.submit("r1", "hi there")          -> S0 emits "hi", "there"(finished)
S1.submit("r1", ["hi", "there"])     -> S1 emits ["HI","THERE"](finished)   (just one chunk)
S2.submit("r1", [["HI", "THERE"]])   -> S2 emits "wav:HI", "wav:THERE"(finished)
```

The user calls `await orch.next_output()` repeatedly and gets 4 items:

#	stage_id	data	finished
1	0	`"hi"`	False
2	0	`"there"`	**?** (your answer)
3	2	`"wav:HI"`	False
4	2	`"wav:THERE"`	True

None of stage 1's outputs appear on the user side. Afterwards `orch.request_states == {}`, `orch.dropped_outputs == 0`. Hint: there is exactly one `finished=True` overall, and it is the last item; use this rule to deduce row 2.

Example 2: only the last stage is final. Same three stages, `final_stage_ids={2}`, prompt is `"a b c"`.

```
S0 receives "a b c"            -> emits "a","b","c"(fin)       invisible to user
S1 receives ["a","b","c"]      -> emits ["A","B","C"](fin)     invisible to user
S2 receives [["A","B","C"]]    -> emits "wav:A","wav:B","wav:C"(fin)

User receives:
  OrchOutput("r1", stage_id=2, data="wav:A", finished=False)
  OrchOutput("r1", stage_id=2, data="wav:B", finished=False)
  OrchOutput("r1", stage_id=2, data="wav:C", finished=True)
```

Note: even though S0 is not final, its completion must still trigger forwarding to S1. "Whether to send to the user" and "whether to forward" are two independent decisions.

Example 3: outputs for unknown requests. Same configuration as Example 1, but:

```
1) Before add_request, a StageOutput("ghost", "?", True) pops out of S1's output channel
2) r1 runs normally (the user still receives the 4 items of Example 1)
3) After r1's terminal message is sent, S2 delivers a late StageOutput("r1", "wav:LATE", True)

Result: no extra messages on the user side; orch.request_states == {}; orch.dropped_outputs == 2
```

"ghost" was never registered, "r1" has already been cleaned up; to the orchestrator both are unknown ids and are handled the same way.

Key questions

1. In Example 1, S0's last chunk `"there"` carries `finished=True`. What should the `finished` of the `OrchOutput` sent to the user be? Is it determined by `StageOutput.finished`, or by something else?
2. S0's last chunk both belongs to a final stage (must go to the user) and is S0's completion event (must be forwarded to S1). Does this one output have to do both? If so, does the order matter?
3. Why is S2's input `[["HI","THERE"]]` rather than `["HI","THERE"]`? Which list exactly is `forward_fn`'s third argument, "all data produced by stage s", and when is it accumulated?
4. Why must `ReqState` be put into `request_states` **before** `await stages[0].submit()`? While that await yields control, what might other tasks do?
5. After a request is cleaned up, another output belonging to it arrives. What happens? If at that point you "conveniently" create a new ReqState with something like `setdefault`, what symptoms appear?
6. Why does the reader task, after reading a batch, not process it directly (change state, send to the user, forward), but instead drop it into `_ready_q` for the single dispatcher? There are no locks under asyncio; what guarantees correctness?
7. With `final_stage_ids={0}`, what is `last_stage`? Will S1 and S2 be submitted?

Details

- **Two output types, two directions**: `StageOutput(request_id, data, finished)` goes stage→orchestrator, obtained via `StageClient.get_output()`, and one call returns a `list[StageOutput]` (a batch that may mix several

© 2026 kzhou003

requests). It has **no** stage_id field; only the reader that read it knows which stage it came from. `OrchOutput(request_id, stage_id, data, finished, error)` goes orchestrator→user, generated only for final stages.

- `last_stage = max(final_stage_ids)` (already computed in the constructor): the pipeline only needs to run up to the last final stage; stages after it never need to be submitted. That's what "forward only if s finished and s < last_stage" means.
- **ReqState fields**: `submitted` = set of stages already submitted; `finished` = set of stages that reported finished; `stage_data[s]` = all data stage s produced for this request, in arrival order; `pending_final` is unused in P02 (P03 uses it).
- **"Exactly one finished=True, and it is the last"**: finished here is the **request-level** "the whole request is done", not "some stage is done". The docstring's condition is "every final stage finished", which can be expressed as the set inclusion `final_stage_ids ⊆ st.finished`.
- **Three queues**: `_req_q` (control messages posted by add_request → request handler), `_ready_q` (shared by all readers → dispatcher), `_out_q` (dispatcher → `next_output()`). `next_output(timeout)` with timeout=None means wait forever.
- **Single writer**: asyncio is cooperative scheduling; code between two awaits won't be interrupted by other tasks. So as long as there's no await between "read state" and "modify state", no lock is needed. "A single dispatcher processes serially" means putting all state modifications into one task and executing them in order.

Level 2 · Approach & pitfalls

- **Task breakdown** (created in `start()`, stored in `self._tasks`):

task	Count	What it does
`_reader(sid, client)`	One per stage	Infinite loop `await client.get_output()`, puts `(sid, batch)` into `_ready_q`. Only moves data, never touches state.
`_request_handler()`	1	Takes add messages from `_req_q`, calls `_handle_add`.
`_dispatcher()`	1	Takes a batch from `_ready_q` and calls `_handle_output(sid, o)` on each item **in order**.

  `shutdown()`: cancel all tasks, then `gather(..., return_exceptions=True)` to wait until they actually exit (otherwise you get "Task was destroyed but it is pending" warnings).
- **Why the reader attaches sid**: a StageOutput doesn't know which stage it came from. The reader is the only place that knows, so it packs `(sid, outs)` when enqueuing. One stage's outputs pass through "one reader → one FIFO queue → one dispatcher", so order within a stage is preserved naturally.
- **Decision order in `_handle_output`** (write it as comments first, then fill in code): ① look up the state; if missing, increment the drop count and return; ② append data to `stage_data[sid]`; ③ if `o.finished`, add sid to `finished`; ④ if sid is a final stage: compute request-level done and send an OrchOutput; if done, clean up and **return immediately**; ⑤ if `o.finished` and `sid < last_stage`: build the input with forward_fn, mark submitted, submit to sid+1.
- **Modify state first, then await**: in the dispatcher, `await self._out_q.put(...)` and `await stages[nxt].submit(...)` are both yield points. Update stage_data / finished / submitted before any await, so any task that cuts in sees consistent state.
- **The add path**: `add_request` only puts `("add", rid, prompt)` into `_req_q`; the actual registration + submission happens in the request handler. Registration (a synchronous dict assignment) must come before the first await.
- **Common mistake 1: using `o.finished` directly as OrchOutput.finished**. Symptom: in Example 1 the user gets two finished=True (S0's "there" and S2's "wav:THERE"), and the first isn't the last; if you clean up at the first finished, all of S2's outputs become dropped.
- **Common mistake 2: forwarding on every chunk** (instead of waiting for the stage to finish). Symptom: S1 is submitted multiple times for the same request, or the prev_data seen when forwarding is incomplete. **Common mistake 3: forgetting to return after cleanup**, continuing on to read deleted state (KeyError) or to forward.
- **Common mistake 4: submitting before registering**. If the stage emits its first output inside submit (or while submit yields), the dispatcher drops it as an unknown id. In Example 1 the symptom is: the user sees only stage 0's `"there"`, S1 receives `["there"]`, and S2 finally emits only `"wav:THERE"`, `dropped_outputs == 1`. No hang, but a chunk of data silently goes missing. Also: an exception raised inside a task isn't printed automatically; the task just dies quietly, showing up as a `next_output` timeout. When debugging, check `task.done() and task.exception()`.

Level 3 · Walkthrough (code: see the commented solutions in Appendix C)

Step-by-step state table for Example 1 (`final_stage_ids={0,2}`, forward_fn returns prev_data unchanged):

Step	Dispatcher handles	stage_data	finished	Action
0	(request handler) add r1	{}	{}	Register ReqState, submitted={0}, then `await S0.submit("r1","hi there")`

© 2026 kzhou003

1	`S0: ("hi", fin=False)`	`{0:['hi']}`	{}	0 ∈ final → send `(0,'hi',False)`; not finished, no forward
2	`S0: ("there", fin=True)`	`{0:['hi','there']}`	{0}	0 ∈ final; {0,2} ⊄ {0} → send `(0,'there',False)` (**False**!); 0 finished and 0 < 2 → `forward_fn(1,'hi there', ['hi','there'])`, submitted={0,1}, submit S1
3	`S1: (['HI','THERE'], fin=True)`	`{…, 1: [['HI','THERE']]}`	{0,1}	1 ∈ final → not sent to user; 1 finished and 1 < 2 → `forward_fn(2, …, [['HI','THERE']])`, submitted= {0,1,2}, submit S2
4	`S2: ("wav:HI", fin=False)`	`{…, 2:['wav:HI']}`	{0,1}	send `(2,'wav:HI',False)`
5	`S2: ("wav:THERE", fin=True)`	`{…, 2: ['wav:HI','wav:THERE']}`	{0,1,2}	{0,2} ⊆ finished → send `(2,'wav:THERE',True)`, clean up, return (2 == last_stage, so it wouldn't forward anyway)
6	`S2: ("wav:LATE", fin=True)` (suppose it arrives late)	—	—	r1 no longer in request_states → `dropped_outputs += 1`

Answers to the open questions

1. **finished for S0's last chunk**: False. OrchOutput.finished is request-level, with the condition `final_stage_ids ⊆ st.finished`; at that point finished={0}, which doesn't include 2. The condition first holds only at S2's last chunk, so "exactly one, and it's the last" is satisfied automatically.
2. **Do both?** Yes. "sid ∈ final → send to user" and "o.finished and sid < last_stage → forward" are two independent ifs, not if/elif. Sending to the user first and then forwarding is more natural (the user's text needn't wait for S1's submission); but the only hard constraint is: if this item is already terminal (done), clean up and return after sending; never forward.
3. **The forward input**: `forward_fn(s+1, st.prompt, st.stage_data[s])`. stage_data[s] is the list the dispatcher builds by appending once for every stage-s output it handles, so S1's input is the list of S0's two chunks `["hi","there"]`; S1 emits only one chunk, and that chunk is itself a list, so S2 receives "a one-element list" `[["HI","THERE"]]`. Unpacking/concatenating is forward_fn's job (in a real system, functions like thinker2talker); the orchestrator doesn't care what data looks like.
4. **Why register before awaiting submit**: await hands control back to the event loop. The stage may put its first output into its output channel inside submit, and the reader and dispatcher run right away; if rid isn't in request_states yet, that output is dropped as an unknown id (see common mistake 4 in Level 2). A dict assignment is synchronous; placing it before the await leaves no window at all.
5. **Late outputs after cleanup**: state not found → `dropped_outputs += 1` → return. Never recreate state: a recreated ReqState will never reach a terminal message, stays in request_states forever (a leak), and may even send an orphan output to the user.
6. **Why a single dispatcher**: if the three readers each modified state, they would interleave at their own await points, e.g. two readers both see "not done yet" and each send one, or one has just cleaned up while another writes. Putting all modifications in one task executed serially is equivalent to a global lock, without writing any lock.
7. `final_stage_ids={0}`: last_stage=0, S0 finishing is the terminal message, and S1/S2 are never submitted.

Skeleton (registration and output handling)P03 subclasses this class and overrides only `_handle_add` / `_handle_output`, so writing the three loops and cleanup as separate methods makes the next drill much easier.

A-10 · Orchestrator v2: async_chunk prewarm + terminal gating

Level 1 · I/O examples

Example 1: the talker's (S1, non-final) completion report arrives late. `final_stage_ids={0,2}`, `prewarm_fn = lambda sid, p: f"warm{sid}:{len(p)}"`, `add_request("r1", "hi there")`.

```
three submits after add (in order):
  S0.submit("r1", "hi there")
  S1.submit("r1", "warm1:8")
  S2.submit("r1", "warm2:8")
```


© 2026 kzhou003

The stages then report in the order below (the data itself travels through the connector; these reports are only the outputs the orchestrator sees):

Arriving StageOutput	User receives at this moment	ReqState.finished	pending_final
`S0 ("hi", False)`	`(0,'hi',False)`	{}	None
`S0 ("there", True)`	`(0,'there',False)`	{0}	None
`S2 ("wav:HI", False)`	`(2,'wav:HI',False)`	{0}	None
`S2 ("wav:THERE", True)`	**(nothing)**	{0,2}	`(2,'wav:THERE',True)`
`S1 (None, True)`	`(2,'wav:THERE',True)`	(cleaned up)	—
`S2 ("wav:EXTRA", False)`	(nothing)	—	—; `dropped_outputs == 1`

Example 2: normal order (S1 finishes before S2). Same config as above.

Arrives	User receives
`S0 ("hi", False)`	`(0,'hi',False)`
`S0 ("there", True)`	`(0,'there',False)`
`S1 ("tok", True)`	(nothing: S1 is not final)
`S2 ("wav:HI", False)`	`(2,'wav:HI',False)`
`S2 ("wav:THERE", True)`	`(2,'wav:THERE',True)`, then cleanup

No pending_final is needed here: the moment the last final stage finishes, every submitted stage has also finished.
Example 3: the last final stage to finish is S0 (S2 ends first, the text ends last). Same config as above.

Arrives	User receives	finished
`S0 ("hi", False)`	`(0,'hi',False)`	{}
`S2 ("wav:HI", True)`	`(2,'wav:HI',False)`	{2}
`S1 (None, True)`	(nothing)	{1,2}
`S0 ("there", True)`	`(0,'there',True)`, cleanup	—

The terminal message's stage_id is not necessarily last_stage: it is the stage of "the output that first makes the condition true".

Key questions

1. In P02, `submitted` grows bit by bit as outputs are forwarded (only {0} at add time); in P03 it must contain 0..last at add time. If you keep the P02 approach and only mark {0}, at which step of Example 1 does it go wrong, and how? What is the benefit of marking before the first await?
2. Example 1, row 4: the last final stage has finished, but S1 has not. What should the user receive at this moment? What do you store, and where?
3. When a **non-final** stage reports completion, what must the handler do? Under what condition does it do nothing?
4. "All final stages finished" and "all submitted stages finished" are two different set conditions. Write each one. What does each control (the value of OrchOutput.finished? whether to emit immediately? whether to clean up?)
5. Does a non-final stage's data ever reach the user? Is P02's `stage_data` still useful here?
6. This drill inherits your P02: which of the reader, dispatcher, request handler, `_cleanup`, and dropped counter can be reused unchanged?
7. After pending_final is stored, another non-finished S1 output for the same request arrives. How should it be handled?

Details

- **async_chunk mode**: data between stages travels through the connector (shared memory / side channel), and the orchestrator no longer calls forward_fn. The `forward_fn=lambda *_: None` passed to the constructor only satisfies the parent class signature.
- **prewarm**: as soon as a request arrives, each downstream stage receives a placeholder input `prewarm_fn(sid, prompt)`, so it can reserve a slot in its own scheduler and wait for the first chunk. Submit order: S0 first (real prompt), then S1, S2, ... (placeholders). Submit range is `0..last_stage`.
- **Two gates**: the value of `OrchOutput.finished` is still P02's "all final stages finished"; but **when** that finished=True message is emitted has one more condition: all submitted stages have finished (`submitted ⊆ finished`).
- `ReqState.pending_final`: stores an already-built OrchOutput with `finished=True`. It is "held back", not discarded: it is emitted later as is, and its content (stage_id, data) is that last final output.

© 2026 kzhou003

- **Why gate**: in the real system the request is cleaned up after the terminal message, reclaiming SHM, connector buffers, and replica bindings. If the talker is still writing to the connector, cleaning up early reclaims resources it is using; its later completion report also becomes an "unknown request".
- **Inheritance**: override only `_handle_add` and `_handle_output`. If your P02 hard-coded these two steps in the dispatcher instead of separate methods, go back and refactor P02 first.

Level 2 · Approach & pitfalls

- **add path**: create ReqState → register it in request_states → `submitted = {0..last}` (all done synchronously) → `await S0.submit(prompt)` → then `await S_k.submit(prewarm_fn(k, prompt))` in order. Each stage is submitted exactly once.
- **Decision table for the output path** (look up state first, drop if unknown; then if finished, add to st.finished):

Output comes from	all_final (final ⊆ finished)	all_done (submitted ⊆ finished)	Action
final stage	no	—	emit `OrchOutput(..., finished=False)`
final stage	yes	yes	emit the `finished=True` message, clean up
final stage	yes	no	build the finished=True message, store it in pending_final, **do not emit**
non-final stage	—	yes, and pending_final exists	emit pending_final, clean up
non-final stage	—	otherwise	do nothing

- **submitted is what the gate is based on**: in `submitted ⊆ finished`, every stage missing from submitted is one gate missing. For the consequence of keeping P02's {0}-only marking, see Level 3. Marking everything at once before the first await follows the same principle as P02's "register before submit": the state is complete before any yield point, so you never need to reason about which interleavings are safe.
- **Common mistake 1: not checking pending_final when a non-final stage finishes**. Symptom: in Example 1 the user never receives the terminal message, `next_output` times out, and the request lingers in request_states.
- **Common mistake 2: emitting the message while also storing it in pending_final**, or forgetting to clean up after emitting pending. Symptom: duplicate terminal messages (two finished=True), or leaked request state and later outputs not being dropped.
- **Common mistake 3: reusing P02's _handle_output directly** (the parent's version calls forward_fn and submits S1 when S0 finishes). Symptom: S1 is submitted twice (once for prewarm, once for forwarding), and the second input is None.
- **Common mistake 4: "resurrecting" state for an unknown id** (e.g., `setdefault`). Symptom: a late S2 output after cleanup creates a new ReqState (submitted is empty, so the gate is useless), the late output is sent to the user after the terminal message, and the state leaks permanently in request_states.

Level 3 · Walkthrough (code: see the commented solutions in Appendix C)

State evolution for Example 1 (✓ means the output has finished=True):

Event	submitted	finished	all_final	all_done	pending	Sent to user
add	{0,1,2}	{}			None	
`S0 hi`	{0,1,2}	{}	F	F	None	`(0,hi,F)`
`S0 there ✓`	{0,1,2}	{0}	F	F	None	`(0,there,F)`
`S2 wav:HI`	{0,1,2}	{0}	F	F	None	`(2,wav:HI,F)`
`S2 wav:THERE ✓`	{0,1,2}	{0,2}	**T**	F	`(2,wav:THERE,T)`	—
`S1 ✓`	{0,1,2}	{0,1,2}	T	**T**	→ emitted	`(2,wav:THERE,T)`, cleanup
`S2 wav:EXTRA`	—	—				dropped += 1

Answers to the Level 1 questions

1. With only {0} marked, Example 1 breaks the moment S2's `"wav:THERE"` arrives: finished={0,2}, `{0} ⊆ {0,2}` holds, so the terminal message is emitted and the request cleaned up immediately: exactly the "premature terminal" this drill guards against; S1's late completion report then hits an unknown id, so `dropped_outputs == 1`. If 0..last is marked before the

© 2026 kzhou003

first await, then no matter how fast the stages are or at which await point an output slips in, it always sees the complete submitted set.

2. Emit nothing. Build the `finished=True` OrchOutput, store it in `st.pending_final`, and return.
3. After adding sid to finished, check `submitted ⊆ finished` and `pending_final is not None`; if both hold, emit pending and clean up. If pending is empty (some final stage has not finished yet, like S1 in Example 2), do nothing: when that final stage finishes later it takes the "emit immediately" branch.
4. all_final = `final_stage_ids ⊆ finished`: decides the value of OrchOutput.finished. all_done = `submitted ⊆ finished`: decides whether the finished=True message is emitted immediately (with cleanup) or held in pending_final first.
5. No; a non-final stage's output is only used to update finished. stage_data is not needed here (no more forwarding); recording it or not does not affect the result.
6. start / shutdown / add_request / next_output / reader / dispatcher / request handler / _cleanup are all reused; only _handle_add and _handle_output are replaced.
7. It is not finished and changes no set, so per the decision table it falls into "do nothing".

SkeletonNote that `_cleanup` comes after `await self._out_q.put`: on an unbounded queue put never actually suspends, but even if it did, the state is already consistent (pending has been taken out).

A-11 · OutputSink: batched wakeups from thread → asyncio

Level 1 · I/O examples

Example 1: watching the wakeup count in a single thread (all in the same loop, so every step is visible)

```
s = OutputSink(); s.bind_loop()
s.put("a"); s.put("b"); s.put("c")
```

Call	Returns	wakeups_scheduled	Notes
`3 puts`	—	1	The first put schedules a wakeup; the next two see "wakeup in flight" and don't schedule
`await s.get_batch(max_items=2)`	`['a','b']`	1	Queue non-empty, returns immediately, no sleep
`await s.get_batch(max_items=2)`	`['c']`	1	Same as above
`s.put("d")`	—	1	The previous wakeup is still "in flight" (the consumer hasn't re-armed), so nothing is scheduled
`await s.get_batch()`	`['d']`	1	Got it on the first drain
`await s.get_batch(timeout=0.05)`	`[]`	1	Times out after ~0.05s; this time the consumer actually "armed"
`s.put("e")`	—	**2**	Only the first put after arming schedules another wakeup
`await s.get_batch()`	`['e']`	2	

Example 2: a put inserted "after an empty drain, before arming"

```
s = OutputSink(); s.bind_loop()
s.put("a")
await s.get_batch()            # -> ['a'], wakeups_scheduled == 1, the "in flight" flag was not cleared

def hook():                    # another thread puts, and we wait for it to finish
    t = threading.Thread(target=lambda: s.put("b")); t.start(); t.join()
    s.after_empty_drain = lambda: None
s.after_empty_drain = hook

await s.get_batch(timeout=2.0) # correct implementation: returns ['b'] in ~0 s; wakeups_scheduled still == 1
```

That put saw "wakeup in flight", so it didn't schedule a wakeup. An implementation that drains only once and then sleeps will sleep the full 2 seconds for nothing.

Example 3: consumer already waiting, another thread puts 5 items in a row

```
task = asyncio.create_task(s.get_batch(timeout=2))   # let it go to sleep first
await asyncio.sleep(0.01)
```


© 2026 kzhou003

```
threading.Thread(target=lambda: [s.put(i) for i in range(5)]).start()  # then join
await task          # -> [0, 1, 2, 3, 4]
s.wakeups_scheduled # -> 1
```

5 puts produce only 1 `call_soon_threadsafe`; when the loop wakes up, all 5 items are already in the queue and are taken in one batch. (How many batches you get depends on thread timing, but the wakeup count must be 1.)

Key questions

1. Who sets the "wakeup in flight" flag to True, and who sets it back to False? When a `get_batch` gets data straight from the queue and returns, is it reset? Is that a problem? (Compare row 4 of Example 1.)
2. Why can't `put()` call `event.set()` directly? What thread assumptions do `asyncio.Event` and `loop.call_soon_threadsafe` each make?
3. In `put()`, which comes first: "append data" or "check/set the flag"? What interleaving appears if you reverse them?
4. There is a window between the consumer "finding the queue empty" and "arming (getting ready to sleep)". If a put lands exactly in that window and sees the flag still True, what happens? How do you avoid missing it?
5. When should `event.clear()` be called? What happens if you never clear, or clear before draining?
6. `timeout` is the total duration of the whole `get_batch` call, while the loop may go around several times (e.g. woken up but the drain is empty). How do you guarantee the total doesn't exceed it?
7. Inside the `after_empty_drain` hook another thread may call `put()`, and the hook waits for that thread to finish. What happens if you call the hook while holding a lock that put also needs?

Details

- **Wake cycle**: starts when "the flag goes False→True (some put scheduled a wakeup)" and ends when "the consumer re-arms (sets the flag back to False)". Within one cycle, no matter how many puts, `call_soon_threadsafe` runs at most once; `wakeups_scheduled` is the number of cycles.
- **There is only one legal path for cross-thread wakeup**: `asyncio.Event` is not thread-safe and may only be set on its loop's thread. Other threads must use `loop.call_soon_threadsafe(event.set)`: it pushes the callback into the loop and writes the self-pipe to wake the selector. That step involves a lock and a syscall, so it should be coalesced. When the loop is closed it raises `RuntimeError`, which the producer may ignore.
- **"never blocks"**: put may take a very short `threading.Lock` (protecting only the flag's check-and-set), but must not wait for the consumer. `collections.deque`'s `append` / `popleft` are atomic in CPython, so they need no lock.
- **"1..max_items items or []"**: as long as the queue has something, return at least 1 and at most max_items, leaving the rest for next time; return an empty list only on timeout.
- **bind_loop**: with no argument, use the currently running loop (`asyncio.get_running_loop()`), and create the Event here. A put that happens before bind just needs to put the data into the queue (there's no loop to wake); the consumer will naturally pick it up on its first drain.
- **Where the hook is called**: strictly after "a drain finds the queue empty" and before "re-arming", once per empty drain; do not hold any lock that put also needs when calling it.

Level 2 · Approach & pitfalls

- **State**: a `deque` (data), a `threading.Lock` + `bool` (the "wakeup in flight" flag), `loop`, `asyncio.Event`, and the `wakeups_scheduled` counter. The flag's "check-and-set" must be one atomic action, so it goes inside the lock; the counter is incremented in the same critical section.
- **Order in put**: append first, then inside the lock "if the flag is False, set it True and count", and after leaving the lock (if needed) `call_soon_threadsafe(event.set)`. Appending first guarantees that whenever anyone sees "flag already set", the data is already in the queue.
- **get_batch is a loop**: drain → return if non-empty → call hook → arm (clear the Event, set the flag to False inside the lock) → **drain again** → return if non-empty → wait on the Event (with the remaining timeout) → back to the top. There must be no await between arming and the second drain.
- **Why the second drain closes the window**: for any concurrent put, its "look at the flag" happens either before or after arming. Before: it sees True and doesn't schedule a wakeup, but its append has also already happened (append precedes looking at the flag), so the second drain is sure to get it. After: it sees False and schedules a wakeup, the Event gets set, and the wait wakes up. Nothing is lost in either case.
- **Timeout**: on entry compute a deadline with `loop.time()`, and compute the remaining time before each wait; if remaining ≤ 0, return []. Use `asyncio.wait_for(event.wait(), remaining)`, and after catching `asyncio.TimeoutError` drain one last time (data might arrive right as the timeout fires). With timeout=None, remaining is also None, meaning wait forever.
- **Spurious wakeups**: the Event may already be set by a leftover set from the previous round (the previous round took data straight from the queue without waiting), so wait returns immediately and the drain is empty. That's not an error; one more trip around the loop will arm properly and go to sleep, provided your implementation is a loop and not "wait only once".
- **Common mistakes and symptoms**:
  - Draining once then sleeping: the Example 2 scenario stalls until timeout; ping-pong interactions (put one item at a time, wait for the reply) stall intermittently.
  - call_soon_threadsafe on every put: functionally correct, but 1000 puts produce 1000 wakeups.

© 2026 kzhou003

- Looking at the flag before appending: producer sees False → schedules wakeup → consumer wakes, drain empty → arms, sleeps → only now does the producer append, and it doesn't wake again; the data is stuck.
- Using `asyncio.Queue` to put across threads: not thread-safe; occasional data loss or the loop not waking.
- Calling the hook or awaiting while holding the lock: deadlock.

Level 3 · Walkthrough (code: see the commented solutions in Appendix C)

Interleaving timeline for Example 2:

Time	Consumer (loop thread)	Producer (other thread)	Flag	Queue
t0	Previous round took 'a' from the queue and returned directly (didn't arm)		True	[]
t1	drain → []		True	[]
t2	call hook	append('b'); sees flag True → no wakeup scheduled	True	['b']
t3	clear Event; set flag False (arm)		False	['b']
t4	**second drain → ['b'], return**		False	[]
t4'	(naive implementation) goes straight to await event.wait() → nobody will set it, sleeps until timeout		False	['b']

Note that after returning at t4 the flag is False and the Event is clear: the next put will schedule a wakeup normally, and the count becomes 2.

Answers to the Level 1 questions

1. The flag is set True by put and set False by the consumer when arming. It is not reset when data is taken straight from the queue; that's fine, because the consumer always drains first on its next entry and only arms right before actually sleeping. The only cost is that "puts during this period won't wake", and no wakeup is needed then anyway (row 4 of Example 1).
2. `asyncio.Event` may only be operated on its loop's thread; `call_soon_threadsafe` is the only thread-safe way to have the loop run a callback and wake it out of select.
3. You must append first, then look at the flag. For the reversed interleaving, see the last item in Level 2.
4. That put won't wake you; the only remedy is to drain once more after arming (t4 in the table above).
5. clear belongs inside "arming", next to "set flag False", followed immediately by the second drain and the await, with no yield point in between. That clears any leftover set from the previous round, and any put after arming will schedule a fresh set. Since callbacks queued by `call_soon_threadsafe` only run when you await, the order of clear and set-False doesn't affect correctness. Never clearing is serious: the Event stays set, `event.wait()` returns without suspending, and on an empty queue get_batch spins in a loop that never yields: with a timeout it burns CPU until the timeout and then returns [] (no other task on the loop gets to run meanwhile); with timeout=None it hangs forever.
6. Compute the deadline once on entry, and use `deadline − loop.time()` as the wait duration on each iteration.
7. Deadlock: the thread in the hook needs the same lock to put, while the hook waits for that thread to finish. Never hold the lock when calling the hook.

Delete the two "second drain" lines and rerun Example 2: you'll see it stall for the full 2 seconds. That's the fastest way to understand this drill.

A-12 · StagePool: sticky binding, two-phase abort, fault isolation

Level 1 · I/O examples

Example 1: sticky + round-robin + evict. One stage with 3 replicas: `p = StagePool(1, [A, B, C])` (the replica id is the index 0/1/2).

Call	Returns	Notes
`p.pick("x")`	0	new request, 0's turn
`p.pick("y")`	1	
`p.pick("z")`	2	
`p.pick("w")`	0	wraps back to 0
`p.pick("x")`, `p.pick("y")`	0, 1	already bound → returned as is (sticky), round-robin does not advance
`p.bound("q")`	None	read-only query, never creates a binding

© 2026 kzhou003

`p.evict(1)`	`['y']`	B died: return and delete the requests bound to 1
`p.live_replicas()`	`[0, 2]`	
`p.pick("y")`, `p.pick("v")`	∈ {0, 2}	y's old binding was deleted, so it is treated as a new request; rotation only covers live replicas (which one it lands on depends on your round-robin counter; any live replica is correct)

Example 2: abort batched per replica, and failure semantics. Continuing Example 1, the bindings are now `{x:0, y:0, z:2, w:0, v:2}`.

```
await p.abort(["x", "z", "w", "ghost", "v"])

replica calls (exactly one per replica):
  A.abort(["x", "w"])
  C.abort(["z", "v"])
"ghost" has no binding -> ignored. Afterwards bound() is None for x/z/w/v, and y is still 0.
```

Another pool `q = StagePool(0, [A, B])`, where B's abort raises `RuntimeError("B ipc broken")`; bindings are `{m:0, n:1, o:0}`:

```
await q.abort(["m", "n", "o"])   # -> raises RuntimeError("B ipc broken")
call log: A.abort(["m","o"]) succeeds; B.abort(["n"]) raises
afterwards: q.bound("m") is None, q.bound("o") is None, q.bound("n") == 1   # n is kept, can be retried
```

Example 3: two-phase abort and dead replicas. Three stage pools s0/s1/s2; request `"r9"` is bound to T0, K1, W0 respectively; the partial text r9 has generated so far in the processor is `"Hello wor"`.

```
out = await abort_request([s0, s1, s2], proc, "r9")
call order: collect_partial("r9") -> T0.abort(["r9"]) -> K1.abort(["r9"]) -> W0.abort(["r9"]) ->
commit_abort("r9")
out == OrchOutput("r9", "Hello wor", finished=True, error="abort")

Different pools: s0 = [T0], s1' = [K0], where K0.abort raises RuntimeError; r9 is bound in both pools
await abort_request([s0, s1'], proc, "r9")   # -> raises RuntimeError
call order: collect_partial -> T0.abort(["r9"]) -> K0.abort(["r9"]) raises   # no commit_abort
s0.bound("r9") is None (it succeeded, released); s1'.bound("r9") == 0 (kept, can be retried)
```

Dead replica: `d = StagePool(2, [W0, W1])`, and `a,b,c` pick 0,1,0 in order.

```
handle_dead_replica(d, 0)
-> [OrchOutput("a", None, finished=True, error="stage 2 replica 0 died"),
    OrchOutput("c", None, finished=True, error="stage 2 replica 0 died")]
d.bound("b") == 1 (unaffected); d.live_replicas() == [1]
after another d.evict(1), d.pick("new") raises NoLiveReplica
```

The error text is up to you, as long as it is non-empty and explains the cause.

Key questions

1. What shape should the binding table have? What does `pick` do for bound vs. unbound requests? Should the round-robin counter advance on a "sticky hit"?
2. Round-robin rotates only among **live** replicas. Do you "take the modulo over the live list", or "take the modulo over all replicas and skip dead ones"? How do the two behave differently after an evict?
3. Why must `evict` delete the bindings instead of just marking the replica dead? If you don't delete them, what happens on a later `pick` or `abort` for those requests?
4. In `abort(ids)`, 5 ids spread over 2 replicas: how many replica calls should there be? What about ids with no binding? If a replica has no ids at all, should it be called?
5. In Example 2, A succeeds and B raises. Which bindings should be released and which kept? Why is "release first, then await" wrong?
6. In two-phase abort, why must `collect_partial` come before the engine abort and `commit_abort` after it? If commit were moved to the front and some stage's abort failed, what mess would be left behind?
7. What happens when `abort_request` calls `pool.abort(['r9'])` on a pool where r9 is not bound at all? Do you need to check beforehand?

Details

- **replica id = index**: the i in `self.replicas[i]` is the integer used by pick / bound / evict / live_replicas. `live_replicas()` returns the live ids in ascending index order.

© 2026 kzhou003

- **sticky**: a request's KV cache and streaming intermediate state live on the replica it was first assigned to, so every later pick for the same request id must return the same replica, as long as it is still alive.
- `bound` **vs** `pick` **vs** `release` : bound is read-only (returns None if unbound, never creates); pick may create a binding; release deletes the binding when a request ends normally, and does not raise for unknown ids.
- **"ONE abort call per replica"**: within one `StagePool.abort` call, group the ids by binding and call `ReplicaClient.abort(list)` only once per replica that has ids (in the real system one call = one IPC).
- **two-phase**: phase 1 collects "the partial result generated so far" read-only (used for the last message to the user); phase 2 commits only after all engines confirm they have stopped (discarding front-end state such as the detokenizer). Same idea as a database's prepare/commit: the irreversible step goes last.
- **Note the shape of OrchOutput**: in this drill `OrchOutput(request_id, data, finished, error)` has no stage_id field, unlike P02. The terminal error for abort is always `"abort"` ; for a dead replica the terminal data is None and error is an explanatory string.

Level 2 · Approach & pitfalls

- **State**: `alive: list[bool]` (or a dead set), `bindings: dict[str, int]` (request → replica), and a round-robin counter. All methods are synchronous, so there are no concurrency issues; the only await is in abort.
- **pick**: bound and that replica is alive → return it directly. Otherwise: take the live list, raise `NoLiveReplica` if empty; pick one by taking the counter modulo the live list, increment the counter, write the binding. Taking the modulo over the live list guarantees that after an evict you never pick a dead replica and no replica gets picked twice in a row ("modulo then skip dead" gives the dead replica's next neighbor an extra share).
- **evict**: mark dead → find all requests bound to it → delete them from the binding table → return the list. Don't delete while iterating the dict (collect first, then delete).
- **StagePool.abort**: ① iterate ids and group them via the binding table into `{replica: [ids]}` , skipping unbound ones; ② for each group `await replicas[rep].abort(ids)` , and release that group's bindings **only after the group returns successfully**; ③ if any group raises, let it propagate: groups that already succeeded are released, the failed group and unprocessed groups keep their bindings. Within a group keep the caller's order.
- **abort_request**: `partial = processor.collect_partial(rid)` → for each pool `await pool.abort([rid])` (sequentially, no try/except swallowing errors) → `processor.commit_abort(rid)` → return `OrchOutput(rid, partial, True, "abort")` .
- **handle_dead_replica**: just `pool.evict(replica)` plus "one terminal error output per affected request". Don't touch requests on other replicas.
- **Common mistakes and symptoms**
  - Calling replica.abort once per id: the result is right, but the IPC count = number of requests.
  - Releasing bindings before the await: after a failed abort `bound()` becomes None, a retry can't find the replica, and the request keeps running in the engine as an orphan.
  - evict marks dead but doesn't delete bindings: pick may return a dead replica for old requests; aborting those requests after handle_dead_replica calls into a dead process.
  - Running all groups concurrently with `asyncio.gather` : when one group fails, the others may or may not have finished, making it hard to reason about when bindings are released (doable, but you must handle each group's result yourself).
  - Putting commit before abort, or guaranteeing commit with try/finally: when abort fails, the front end has forgotten the request while the engine keeps emitting tokens.

Level 3 · Walkthrough (code: see the commented solutions in Appendix C)

State table for Examples 1 + 2 (counter approach: modulo over the live list):

Operation	alive	bindings	rr counter	Returns / side effects
initial	[T,T,T]	{}	0	
`pick x`	[T,T,T]	{x:0}	1	0
`pick y`		{x:0,y:1}	2	1
`pick z`		{…,z:2}	3	2
`pick w`		{…,w:0}	4	0
`pick x`		unchanged	4	0 (sticky, no advance)
`evict 1`	[T,F,T]	{x:0,z:2,w:0}	4	`['y']`
`pick y`		{…,y:0}	5	live=[0,2], 4 % 2 = 0 → 0
`pick v`		{…,v:2}	6	5 % 2 = 1 → 2
`abort [x,z,w,ghost,v]`		{y:0}	6	`A.abort([x,w])` ; `C.abort([z,v])`

© 2026 kzhou003

Answers to the Level 1 questions

1. `dict[request_id, replica_id]`. Bound and alive → return it; otherwise choose by rotation and write it. A sticky hit does not advance the counter (advancing is not wrong, but it makes the distribution uneven).
2. Prefer modulo over the live list: after an evict it automatically rotates evenly among the live ones. The other approach has to handle "skipping" and its distribution is skewed.
3. A binding means "this request has state on that replica". When the replica dies the state is gone; keeping the binding makes pick return a dead replica and abort call into a dead process. Once deleted, those requests have already received an error terminal message from handle_dead_replica.
4. 2 calls (one per replica with ids); unbound ids are ignored; replicas with no ids are not called.
5. A's group (m, o) is released, B's group (n) is kept, and the exception propagates. Release-then-await: on failure the binding is already gone, so a retry has nothing to go on.
6. collect goes first because commit discards front-end state, after which the partial text is no longer available; commit goes last because only once all engines confirm they have stopped can the front end safely forget the request. Commit first + failed abort = the engine is still producing for it while the front end no longer knows it (orphan output), and the user has already received the abort terminal message.
7. That pool's grouping is empty, so no replica call is made and it returns immediately. No need to check beforehand.

A-13 · RpcRouter: RPC correlation over a shared channel

Level 1 · I/O examples

Example 1: two threads call concurrently, results come back out of order

```
r = RpcRouter()

# send() only "sends the request out"; here a delayed thread simulates the router thread returning the result
later
def send_for(key, val, delay):
    return lambda: threading.Timer(delay, lambda: r.deliver(key, val)).start()

thread 1: r.call("sleep", send_for("sleep", "ok-sleep", 0.05), timeout=1)   -> "ok-sleep"
thread 2: r.call("wake",  send_for("wake",  "ok-wake",  0.01), timeout=1)   -> "ok-wake"
("wake"'s result arrives first, but each thread only gets its own)

# even when the result comes back synchronously inside send(), it must not be lost:
r.call(("rpc", 7), lambda: r.deliver(("rpc", 7), {"gpu_mem": 3}), timeout=0.1)  -> {"gpu_mem": 3}
```

Example 2: timeout, late result, key reuse, exception result

```
r.call("ping", lambda: None, timeout=0.1)      # raises TimeoutError after about 0.1s
r.deliver("ping", "pong")                      # -> False (nobody is waiting anymore)
r.late_results                                 # -> 1
r.call("ping", lambda: r.deliver("ping", "pong2"), timeout=0.1)   # -> "pong2" (never reads the old "pong")

r.call("bad", lambda: r.deliver("bad", ValueError("no such method")), timeout=0.1)
# -> raises ValueError("no such method"): when the result itself is an exception object, raise instead of return

w = r.register("dup"); r.register("dup")       # second call -> ValueError (key still in use)
```

Example 3: fail_all broadcasts and latches

```
3 threads each r.call("w0"/"w1"/"w2", send=lambda: None, timeout=5) and block
main thread: r.fail_all(RuntimeError("orch dead"))
-> the 3 threads each raise RuntimeError("orch dead") almost immediately (far less than 5s)

afterwards: r.call("after", send, timeout=1)  -> immediately raises RuntimeError("orch dead"), send is never
called
            r.register("x")                   -> also raises immediately
r.late_results is not changed by fail_all
```

Key questions

1. Why must you `register` before `send()`? Construct a "send first, register later" timing that waits forever (hint: the last call in Example 1).
2. Why is the waiter a `queue.Queue(maxsize=1)`? Can `deliver` block when putting the result into it?
3. Who removes a waiter from the registry: `deliver`, `call`'s finally, or both? If both can, how do you make sure they don't step on each other?
4. Why does `unregister(key, waiter)` take an extra waiter argument, and delete only when "the one in the table is mine"? Construct a timing where A times out and B reuses the same key, and show whom you'd delete by mistake without the

© 2026 kzhou003

comparison.

5. Does `deliver` need a lock? What goes inside the lock and what outside? Why not put into the waiter while holding the lock?
6. What exception does `waiter.get(timeout=...)` raise on timeout? What do you raise to the caller?
7. After `fail_all`, how are calls currently waiting and calls arriving later handled, respectively? Where is the "latch" stored, and who checks it?

Details

- **key is any Hashable**: it can be a string or a tuple like `("rpc", 7)` (in the real system usually a call_id). At any moment a key can have only one live waiter.
- **deliver's return value and counter**: if there is a waiter, hand it the result and return True; if not (never registered, already unregistered after timeout, already delivered by someone else), return False and `late_results += 1`.
- **"always unregister in finally"**: whether send raises, the wait times out, or the result arrives normally, when call exits it must ensure its waiter is no longer in the table; otherwise the key can never be reused (registering again reports a duplicate).
- **Exceptions as results**: what comes back on the result channel may be an exception object (remote execution failed), or the fatal error broadcast by fail_all. Convention: when call gets a `BaseException` instance, it raises it.
- **"All methods are called from arbitrary threads"**: the registry (dict) and the latched error are shared state, and every read-modify-write must happen under the same `threading.Lock`; but a blocking operation (waiting for the result) must never hold the lock.

Level 2 · Approach & pitfalls

- **State**: `threading.Lock`, `waiters: dict[key, Queue]`, `fatal: BaseException | None`, `late_results`.
- **register**: under the lock: if fatal is set → raise fatal; if key is already in the table → ValueError; otherwise create a `Queue(maxsize=1)`, put it in the table and return it.
- **deliver**: under the lock `pop(key, None)`: pop rather than get, so a key is delivered at most once; if nothing is found, count it and return False. Outside the lock `put_nowait(result)`: capacity 1 and just taken out of the table, so it can't be full and won't block.
- **unregister**: under the lock, delete only when `waiters.get(key) is waiter`. This is safe both when deliver has already popped it and when the key has been re-registered by a new call.
- **call**: `w = register(key)` → try: `send()`; `w.get(timeout)`, converting `queue.Empty` into `TimeoutError`; raise if the result is an exception, else return → finally: `unregister(key, w)`. Note register is outside the try: if register fails, there is nothing to unregister.
- **fail_all**: under the lock set fatal, take out all waiters, clear the table; outside the lock `put_nowait(error)` to each one (defensively ignoring `queue.Full`). Afterwards register fails immediately because of fatal, so call fails before send.
- **Common mistakes and symptoms**
  - Send before register: a result delivered synchronously inside send is counted as late, and the caller waits out the full timeout.
  - Not unregistering after timeout: a second call with the same key reports "duplicate key"; or (if register allows overwriting) reads the previous round's stale late result.
  - unregister doesn't compare identity: A's finally after its timeout deletes the waiter B just registered, B's result is treated as late, and B hangs until timeout.
  - put under the lock / get under the lock: deadlock, or all calls serialized.
  - fail_all wakes waiters but doesn't latch: later new calls send as usual and wait out the full timeout.

Level 3 · Walkthrough (code: see the commented solutions in Appendix C)

Why unregister must compare identity:

Time	Thread A (call k, times out)	router thread	Thread B (call k)	Table k →
t1	register → wA			wA
t2	get times out, about to enter finally	deliver(k, resA): pops wA, puts		(empty)
t3			register → wB	wB
t4	`unregister(k, wA)`: the table holds wB, **no delete**			wB
t5		deliver(k, resB) → True	gets resB	(empty)

If at t4 you skip the comparison and just `del waiters[k]`, the deliver at t5 finds no waiter, resB is treated as late, and B waits in vain until timeout. (At t2 A's waiter received resA but A had already given up, so that result is silently dropped; it does not count as late, because a waiter did catch it.)

Answers to the Level 1 questions

© 2026 kzhou003

1. send may trigger deliver before it returns (even inside send). Send first: at deliver time the key is not in the table → counted as late; then register → waits forever.
2. Each waiter receives exactly one result (deliver uses pop to guarantee single delivery). Capacity 1 is enough, and put_nowait always succeeds, so the router thread is never stuck behind a slow consumer.
3. Both can: on normal delivery deliver pops first, and the unregister in finally finds it's not its own (or absent) and does nothing; on timeout deliver never came, and finally is responsible for deleting.
4. See the table above.
5. Yes; inside the lock only do the dict pop and the counting, and put outside. Holding the lock during anything that may block stalls other threads' register/deliver.
6. `queue.Empty`; convert it to the built-in `TimeoutError` for the caller (you can use `from None` to drop the irrelevant chained exception).
7. Currently waiting: fail_all puts the error straight into their waiters, and their call raises it on receipt. Later ones: fatal is stored in the router, and register raises immediately when it sees it.

A-14 · EngineCore busy loop

Level 1 · I/O examples

Example 1: pre-queued messages + messages arriving while running

```
# A toy engine: add(rid, k) means request rid still needs k steps; step() does k-=1 for every request,
# returns [(rid, remaining k), ...], and removes requests whose k hits zero; utility("div", (a, b)) returns a /
b.

# Before the thread starts, input_queue already holds:
Msg("ADD", ("a", 2)), Msg("ADD", ("b", 1)),
Msg("UTILITY", (7, "div", (1, 0))), Msg("ABORT", ["b"]),
Msg("UTILITY", (8, "div", (6, 3)))
# Start the run_busy_loop thread; after 200ms put Msg("ADD", ("c", 1)); after another 200ms request_shutdown()

Order of engine calls:
  add a, add b, abort (b,), step 1, step 2, add c, step 3
Sequence received by emit:
  ("utility", 7, None, "ZeroDivisionError('division by zero')")
  ("utility", 8, 2.0, None)
  ("output", ("a", 1))
  ("output", ("a", 0))
  ("output", ("c", 0))
  ("shutdown",)
```

All five pre-queued messages are handled before the first step, so both utility results come before every output; b is aborted before it is ever stepped and produces **no output at all**. After step 2 the engine is idle and the loop blocks until c arrives.

Example 2: an exception from step is fatal (same pre-queued messages, but the engine's 2nd step raises `RuntimeError("NCCL timeout")`)

```
Order of engine calls:  add a, add b, abort (b,), step 1, step 2
Sequence received by emit:
  ("utility", 7, None, "ZeroDivisionError('division by zero')")
  ("utility", 8, 2.0, None)
  ("output", ("a", 1))
  ("fatal", "RuntimeError('NCCL timeout')")
```

run_busy_loop returns right after emitting fatal: there is **no** `("shutdown",)`, and the ADD c put later is never processed. Compare utility 7's ZeroDivisionError: it only affects that one call, and the loop continues.

Example 3: idle + shutdown from another thread

```
loop = EngineCoreLoop(engine, emitted.append)
t = threading.Thread(target=loop.run_busy_loop); t.start()
time.sleep(1.0)           # meanwhile engine.step is called 0 times, has_work only 1-2 times, CPU ≈ 0
threading.Thread(target=loop.request_shutdown).start()
t.join(0.5)               # returns immediately
emitted                   # -> [("shutdown",)]
```

Key questions

1. When the engine has no work, which of `input_queue.get()`, `input_queue.get(timeout=0.01)`, `while q.empty(): time.sleep(0.001)` satisfies "BLOCK, no polling / spinning"?
2. Is the message returned by the blocking get() "just for waking up", or must it be handled? If you drop it, which message goes unprocessed in Example 1?

© 2026 kzhou003

3. While the loop is blocked in `get()`, how can another thread wake it "immediately"? Is setting a flag enough?
4. After waking and draining the queue, if shutdown has been requested, should this iteration still step?
5. Why must you "drain all messages first, then step"? If you took only one message per step, what would happen to b in Example 1?
6. Where should the try/except go for utility exceptions and for step exceptions? After which one does the loop continue, and after which one does it return?
7. After a fatal exit, should you still emit `("shutdown",)`? What is the last thing emitted on a normal shutdown?

Details

- **payload format**:

kind	payload	Action
ADD	`(req_id, data)`	`engine.add(req_id, data)`
ABORT	`[req_ids]`	`engine.abort(req_ids)`
UTILITY	`(call_id, name, args)`	`engine.utility(name, args)` : args is passed as one tuple, not *args
WAKEUP	none	Do nothing

- **The emitted tuple shapes must be exact**: `("output", out)` (emit once for **each** element of the list step returns), `("utility", call_id, result, None)`, `("utility", call_id, None, repr(exc))`, `("shutdown",)` (note: a one-element tuple), `("fatal", repr(exc))`.
- **"drain all queued messages in FIFO order"**: loop on `get_nowait()` until `queue.Empty`. `queue.Queue` is already FIFO; just handle each message immediately in the order taken. Don't group them first (e.g. all ADDs then all ABORTs), which would break the ADD→ABORT→ADD semantics.
- **"may be called from any thread / a signal handler"**: request_shutdown should be as simple and non-blocking as possible: set a thread-safe flag (`threading.Event`), then `put_nowait` a WAKEUP to wake the blocked get(). It should not emit anything itself, nor join the thread.
- **"engine keeps running"**: a failed utility is just a failed call; the result goes back to the caller through emit (matched by call_id), and the loop does not exit.

Level 2 · Approach & pitfalls

- **State**: you only need one extra `self._shutdown = threading.Event()` (created in __init__).
- **Extract a _handle(msg)**: dispatch on kind; the UTILITY branch does its own try/except and emits the result or the error. That way "the first message from the blocking get" and "the rest from draining" go through the same function.
- **Loop structure**: `while not shutdown:` (1) if `not engine.has_work()` : block on get for one message and handle it; (2) drain with get_nowait and handle; (3) if shutdown is set → break; (4) if has_work: step and emit each output.
- **Why (4) asks has_work again**: what (1)(2) handled may have been ABORT/UTILITY/WAKEUP, leaving the engine still without work; calling step then is pointless (some engines assert).
- **Exception boundary**: wrap the whole while in `try: ... except Exception as e: emit(("fatal", repr(e))); return` ; after exiting the loop normally, emit `("shutdown",)`. Utility exceptions are swallowed inside _handle and never reach the outer level.
- **Common bugs and symptoms**:

bug	symptom
`get(timeout=...)` polling	When idle, has_work is called tens to hundreds of times per second (spinning)
Setting only the flag without a WAKEUP	When idle, after request_shutdown the loop is stuck in get() forever (or until the timeout)
Dropping the message from the blocking get	The first ADD is lost; that request never produces output
Handling only one message per step	ABORT takes effect after a step; the aborted request still produces output
Not catching step exceptions	The thread dies silently; outside you only see "no more output", no fatal
Stepping once more after shutdown	Extra output during shutdown, or running on resources already being torn down

Level 3 · Walkthrough (code: see the commented solutions in Appendix C)

Answers to the key questions

1. Only `get()` without a timeout. The other two are polling.
2. It must be handled: it's the earliest message in the queue. If dropped, ADD a disappears in Example 1.
3. Put a message (WAKEUP) into the queue. A flag alone isn't enough; the blocked get() can't see the flag.

© 2026 kzhou003

4. Don't step; break immediately and emit shutdown.
5. With one message at a time: the first iteration handles only ADD a and steps, before b is added; later the ABORT may land after one of b's steps, so b produces output (or the order differs from what the client sent).
6. utility: around the single call, emit the error and continue. step: around the whole loop, emit fatal and return.
7. No. The last emit on a normal shutdown is `("shutdown",)`.

Trace: each loop iteration in Example 1 (fill in the ?? yourself)

Iter	(1) has_work	Blocking get returns	(2) messages drained and handled	(3) shutdown?	(4) step / emit
1	False	ADD a	ADD b, UTILITY 7, ABORT [b], UTILITY 8	No	has_work=True → step 1 → output (a,1)
2	True	(doesn't block)	(empty)	No	step 2 → output (a,0)
3	??	?? (after about 200ms)	(empty)	No	??
4	False	??	(empty)	??	break → emit ??

Answers: iteration 3 has_work=False, blocks until it gets ADD c, step 3 → output (c,0); iteration 4 has_work=False, blocks until the WAKEUP put by request_shutdown, shutdown=yes → break → emit `("shutdown",)`. Over the whole run has_work is called ?? times (answer: 4 times at (1) + 3 times at (4) = 7, counted with the loop structure of the skeleton below; with a different structure your count may differ, but it must be single-digit).

Self-check: (1) idle for 1 second with a counter wrapping has_work, and check the call count is single-digit; (2) call request_shutdown from another thread and time the join; (3) pre-queue ADD→ABORT→ADD and record the engine call order; (4) make step raise, and confirm the last emit is fatal and the thread has exited.

A-15 · WAITING_FOR_CHUNK: park and restore

Level 1 · I/O examples

Example 1: a normal request + a request waiting for a chunk (parked from waiting, then parked from running)

```
s = ChunkAwareScheduler(BaseScheduler(), src)      # we control src.is_ready by hand
s.add(Req("x", 0))                                  # normal request
s.add(Req("t1", 1, needs_chunk=True))               # t1's chunk hasn't arrived yet

s.schedule()  -> ["x"]
   parked == {"t1": (t1, "waiting", "WAITING")}, t1.status == "WAITING_FOR_CHUNK"
   load_async call log: ["t1"]
s.schedule()  -> ["x"]                              # still not ready; load_async log still ["t1"]
# t1's chunk arrives (is_ready("t1") becomes True)
s.schedule()  -> ["x", "t1"]                        # parked == {}; x, t1 both in running, both status "RUNNING"
# t1 consumed this chunk, the next one hasn't come yet (is_ready becomes False again)
s.schedule()  -> ["x"]
   parked == {"t1": (t1, "running", "RUNNING")}     # this time the origin is running!
   load_async call log: ["t1", "t1"]                 # a new park => a new load_async
# the next chunk arrives
s.schedule()  -> ["x", "t1"]                        # t1 back in running, status restored to "RUNNING"
```

Note that origin and previous status are recorded **at each park**: the same request is parked from waiting the first time and from running the second time.

Example 2: requests restored together are ordered by arrival, not by add order

```
s.add(Req("k", 5, needs_chunk=True))
s.add(Req("m", 3, needs_chunk=True))
s.add(Req("n", 4, needs_chunk=True))
s.schedule()  -> []                 # all three parked; load_async log ["k", "m", "n"] (parking walks queue
order)
# all three chunks arrive at once
s.schedule()  -> ["m", "n", "k"]    # arrival 3, 4, 5
```

Example 3: when the upstream schedule raises, finally still restores

```
class Exploding(BaseScheduler):
    def schedule(self):
        src.ready.add("z")          # z's chunk happens to arrive during upstream scheduling
        raise RuntimeError("boom")
```


© 2026 kzhou003

```
s = ChunkAwareScheduler(Exploding(), src)
s.add(Req("z", 0, needs_chunk=True))
s.add(Req("y", 1))
s.schedule()                        # RuntimeError("boom") propagates to the caller unchanged
# afterwards: base.waiting == [y(WAITING), z(WAITING)], parked == {}, load_async log ["z"]
```

On restore z is appended to the **end** of waiting, so it comes after y; the exception is not swallowed.

Key questions

1. `BaseScheduler.schedule()` asserts as soon as it sees `WAITING_FOR_CHUNK`. So is it enough for "parking" to change only `status`? Which container should the request object be in at that point?
2. If the same request is not ready for 3 consecutive `schedule()` calls, how many times should `load_async` be called? What if it gets restored and then is not ready again? What mechanism in your code naturally guarantees "once per park"?
3. When a running request is parked, what is its "previous status"? If you always write `"WAITING"` on restore, what happens (hint: upstream only modifies the status of requests in waiting)?
4. Why does `schedule()` restore first in (a) and then park in (b)? If you dropped (a) and relied only on the previous round's (d), how many rounds later would a request that became ready between two `schedule()` calls get scheduled?
5. Is a request restored in (d) included in this round's `base.schedule()` return value? In which round will it be scheduled?
6. What does "arrival" mean in "restore in arrival order": the `Req.arrival` field, the add order, or the insertion order of `parked`?
7. After `abort(id)`, `source.is_ready(id)` may still return True. What guarantees it won't be restored at the next (a)/(d)? Where could it be hiding?

Details

- **parked triple**: `parked[id] = (req, origin, previous_status)`. `origin` records which queue the request was taken from (the representation is up to you, e.g. the strings `"waiting"` / `"running"`); `previous_status` is `req.status` **at the moment** of parking; after parking `req.status` becomes `"WAITING_FOR_CHUNK"`.
- **BaseScheduler semantics**: returns `running ids + waiting ids`, then moves all of waiting into running and sets them to `"RUNNING"`. In other words it never changes the status of requests in running: restoring status is entirely your layer's job.
- **Only requests with `needs_chunk=True` are ever parked**; normal requests never query `is_ready`. `is_ready` means "has the next chunk arrived"; it can flip True → False → True repeatedly (each consumed chunk may turn it False again).
- **Where to restore**: put it back at the **end** of its origin queue (waiting is a `deque`, running is a `list`); "arrival order" only constrains the relative order among requests restored **in the same batch**, it doesn't require reinserting at the original position.
- **Meaning of finally**: (d) must run even when `base.schedule()` raises, but the exception must keep propagating. Normally `schedule()` returns the result of `base.schedule()`.
- **abort**: callable at any time; the request may be in parked, base.waiting, or base.running. An aborted request must never appear in a later `schedule()` result nor be restored; upstream also asserts on an `"ABORTED"` status, so it can't stay in the queues.

Level 2 · Approach & pitfalls

- **Split into two private helpers**: `_restore_ready()` (scan parked, put ready ones back into their origin queues in arrival order) and `_park_not_ready()` (scan both queues, move needs_chunk requests that aren't ready into parked). `schedule()` is then just a four-line skeleton: restore → park → `try: return base.schedule()` → `finally: restore`.
- **Deduplicating load_async**: a parked request is **no longer** in base's queues, so (b) in the next round can't see it at all and won't park it again. Therefore "call `load_async` once at the moment of parking" naturally satisfies "once per park". For extra safety you can keep a "load requested" set and clear it on restore/abort, but make sure you clear it, otherwise the second park won't call load_async again.
- **Deleting while iterating**: `remove(r)` inside `for r in self.base.waiting` skips elements (list) or raises `RuntimeError: deque mutated during iteration`. Take a snapshot with `list(...)` first and iterate over that. Likewise, don't `del` directly while iterating `parked`; collect the ready ones first, then process them.
- **Restore order**: first collect all parked entries where `is_ready`, sort by `req.arrival`, then one by one `del parked[id]`, restore status, and append to the origin queue. Restoring in dict insertion order would give `["k", "m", "n"]` in Example 2.
- **Common bug 1: missing (a)**. Symptom: the first `schedule()` after a request becomes ready doesn't schedule it; it has to wait another round (until (d) puts it back). **Common bug 2: missing (d) or not putting it in finally**. Symptom: after an upstream exception the request stays in parked forever (z gets stuck in Example 3).
- **Common bug 3: hard-coded restored status**. A request from running restored as `"WAITING"` is never changed again by upstream (which only touches waiting), so its status stays wrong forever. **Common bug 4: return inside finally**: swallows the exception.
- **Abort's three hiding places**: check and remove from parked, base.waiting, and base.running; status can be set to `"ABORTED"`. If you only delete from parked and it happens to be in a queue (e.g. a normal request aborted before being ready), upstream will schedule it next round; if you only set ABORTED without removing it, upstream asserts immediately.

Level 3 · Walkthrough (code: see the commented solutions in Appendix C)

© 2026 kzhou003

Step-by-step state for Example 1 (status in parentheses; R=RUNNING, W=WAITING, WFC=WAITING_FOR_CHUNK)

Call	is_ready(t1)	After (a)	After (b): waiting / running / parked	base.schedule() returns	After (d)	loads
#1	F	no change	[x(W)] / [] / {t1: (waiting, W)}, t1(WFC)	["x"]	running=[x(R)], t1 still parked	[t1]
#2	F	no change	[] / [x(R)] / {t1} (t1 not in a queue, (b) can't see it)	["x"]	no change	[t1]
#3	T	waiting=[t1(W)], parked={}	no park (already ready)	["x","t1"]	running=[x(R), t1(R)]	[t1]
#4	F	no change	[] / [x(R)] / {t1: (running, R)}	["x"]	no change	[t1, t1]
#5	T	running=[x(R), t1(R)]	no park	["x","t1"]	no change	[t1, t1]

Mini exercise: if between #2 and #3 the chunk arrives **while the upstream schedule is running**, t1 is restored at step (??) of call #??, and scheduled in call #??. (Answer: restored to waiting by (d) of #2, scheduled by #3.)
Abort trace: add a(0), b(1, needs_chunk), c(2); b not ready.

Operation	waiting	running	parked
schedule() → ["a","c"]	[]	[a, c]	{b}
abort("b")	[]	[a, c]	{} (b removed from the parked table)
abort("c")	[]	[a] (c removed from running)	{}
b's chunk arrives; schedule() → ["a"]	[]	[a]	{}: b isn't in parked, so (a)/(d) never see it

Answers to the Level-1 questions

- Changing only status isn't enough: upstream iterates waiting/running and asserts on WAITING_FOR_CHUNK. It must be **removed** from the queues and exist only in parked.
- 3 consecutive not-ready calls: 1 time. Parked again after a restore: 1 more time (2 total). Mechanism: once parked the request isn't in a queue, so (b) can't reach it.
- A running request's previous status is `"RUNNING"`; hard-coding "WAITING" leaves it sitting in running with the wrong status forever.
- Without (a), a request that became ready between two calls only returns to a queue at the next (d): 1 round late.
- A request restored by (d) is not in this round's return value (upstream has already run); it's scheduled next round.
- arrival means the `Req.arrival` field.
- Deleting from parked is enough to prevent restore (restore only scans parked); but it may also be in base.waiting or base.running, so delete from all three.

Skeleton (fill in the details yourself) Note the order in `_park_not_ready`: "record `r.status` first, then change it to WAITING_FOR_CHUNK"; if reversed, previous status will always be WAITING_FOR_CHUNK.

A-16 · Scheduler: continuous batching + chunked prefill + preemption

Level 1 · I/O examples

Example 1: token budget + chunked prefill (`num_blocks=100, block_size=4, max_num_seqs=8, max_num_batched_tokens=8` ; add a1(prompt 6, max_tokens 3), a2(prompt 5, max_tokens 2), a3(prompt 3, max_tokens 1); update immediately after each schedule)

step	schedule() returns	after update (num_computed / num_tokens / num_blocks)	blocks_used	finished this step
1	`{a1: 6, a2: 2}`	a1 6/7/2; a2 2/5/1 (prefill not done, no sampling); a3 still waiting	3	-
2	`{a1: 1, a2: 3, a3: 3}`	a1 7/8/2; a2 5/6/2 (only now samples its first token); a3 samples 1 and hits max_tokens	4	a3
3	`{a1: 1, a2: 1}`	a1 has 3 outputs, a2 has 2, both hit max_tokens	0	a1, a2

step 1: a1 takes 6 first, the remaining budget of 2 goes to a2 (only part of its prompt). step 2: running goes first: a1 decodes 1, a2's remaining 3; budget has 4 left, so a3's whole 3 tokens get in. Note the block count is based on "already computed + to

© 2026 kzhou003

compute this step" tokens: after step 1 a2 holds only 2 tokens → 1 block.

Example 2: preempting another sequence (`num_blocks=3, block_size=2, max_num_seqs=8, max_num_batched_tokens=64` ; a1(prompt 2, max_tokens 3), a2(prompt 3, max_tokens 2); assume the model's k-th output for aN is `100*N + k`)

step	schedule() returns	What happened	waiting afterwards	num_preemptions
1	`{a1: 2, a2: 3}`	a1 1 block, a2 2 blocks, blocks_used=3 (full)	`[]`	0
2	`{a1: 1}`	a1 will hold 3 tokens → needs a 2nd block, doesn't fit → preempt the last running = a2 (frees 2 blocks, num_computed=0, back to the front of waiting) → retry a1, succeeds. Phase 2: a2 needs ceil(4/2)=2 blocks, 2+2>3, stop	`[a2]`	1
3	`{a1: 1}`	a1's 3rd output, finishes and frees → blocks_used=0	`[a2]`	1
4	`{a2: 4}`	a2 recompute: prompt 3 + 1 generated = 4 tokens computed at once, samples the 2nd output, finishes	`[]`	1

Final a1.output = `[100, 101, 102]` , a2.output = `[200, 201]` : exactly the same as without preemption (outputs are kept, only the KV is recomputed).

Example 3: FCFS head-of-line blocking (`num_blocks=5, block_size=4, max_num_batched_tokens=100` ; add a1(prompt 9), a2(prompt 9), a3(prompt 1) in order)

```
s.schedule()       # -> {"a1": 9}
s.blocks_used      # -> 3
[q.seq_id for q in s.waiting]   # -> ["a2", "a3"]
```

a2 needs 3 blocks but only 2 are left → stop. a3 needs just 1 block and would fit, but it is **not allowed to jump the queue**.

Key questions

1. For a sequence in decode (it just sampled a token), what is `num_tokens − num_computed` at the next schedule? What about a sequence halfway through chunked prefill?
2. When deciding whether blocks fit, how many tokens do you compute the block count from: `num_tokens` or `num_computed + n` ? How many blocks does each give for a2 in step 1 of Example 1?
3. When running sequence X doesn't fit, who gets preempted? After preempting, do you keep processing X (retry) or skip X? What if the one preempted is X itself: should the iteration index advance?
4. For a preempted sequence: which fields must be reset and which must be kept? How many tokens does it compute when it re-enters? Which end of waiting does it go to, and why?
5. When phase 2 hits the first waiting sequence that doesn't fit: `continue` or `break` ? Does `max_num_seqs` limit phase 1, phase 2, or both?
6. In update, when does a scheduled sequence **not** call `sample_fn` ? When a sequence finishes, besides changing status, what two other things must you do?
7. update has no argument telling you "who was scheduled last step and with how many tokens". Where will you store that information?

Details

- **Remaining tokens**: `num_tokens = prompt_len + len(output)` . Remaining uncomputed = `num_tokens − num_computed` . During decode it's 1 (the just-sampled token has no KV yet), so "decode and prefill use the same formula": this is exactly what the vLLM v1 scheduler unifies.
- **Block accounting**: each Seq carries its own `num_blocks` counter; this step needs `ceil((num_computed + n) / block_size)` blocks, and only the delta `need − seq.num_blocks` matters (≤0 means nothing to do). Invariant: `blocks_used == sum(seq.num_blocks for running)` and `<= num_blocks` . Release = subtract from blocks_used and zero seq.num_blocks.
- **budget**: both phases share one `budget = max_num_batched_tokens` , decremented by n for each scheduled sequence. Once the budget is used up, later running sequences are simply not scheduled this step (not preemption, they just don't run this step).
- **"put it at the FRONT of waiting"**: the preempted one is the latest arrival in running, but it's still earlier than everything in waiting, so it goes to the front to preserve FCFS. It is also tried in phase 2 of the same step (step 2 of Example 2 tries it, but it doesn't fit).
- **"if the victim is the sequence itself, skip it"**: X is no longer in running, so it's not scheduled this step; since running shrank by one, don't advance the index (or the loop just ends naturally at the end of the list).
- **Finish condition in update**: append the sampled token to output first, then check `tok == eos_token or len(output) >= max_tokens` ; so eos itself is in output. On finish: status=FINISHED, remove from running, release blocks, put into

© 2026 kzhou003

finished.

Level 2 · Approach & pitfalls

- **Write three helpers first**: `_blocks_for(n_tokens)` (ceil), `_try_grow(seq, n_tokens) -> bool` (compute the delta; if it fits, update both blocks_used and seq.num_blocks, otherwise change nothing and return False), `_preempt(seq)` (remove from running, release blocks, num_computed=0, status=WAITING, waiting.appendleft, counter+1). The schedule logic becomes very short.
- **Use an index-based while loop for phase 1, not for**: preemption mutates `self.running` (removes the last element). Deleting while iterating with `for seq in self.running` skips elements or visits a sequence that was already preempted.
- **Inner "preempt-retry" loop in phase 1**: `while not _try_grow(seq, ...)`: preempt running[-1]; if it is seq itself, break out and mark "this sequence skipped". Only when _try_grow succeeds do you write to scheduled, deduct budget, and advance the index.
- **Phase 2**: `while waiting and budget > 0 and len(running) < max_num_seqs`: look at the head, n = min(remaining, budget); if _try_grow fails, break; on success popleft, status=RUNNING, append to the end of running, record n, deduct budget.
- **Remember this step's schedule**: store the dict returned by schedule in `self._scheduled` (return a copy so callers can't mutate it). update iterates over it and clears it at the end.
- **Finding sequences by id in update**: build `{seq_id: seq}` first (from running), because you'll be removing finished sequences from running during the loop.
- **Common bugs and symptoms**:

bug	symptom
Computing blocks from num_tokens instead of num_computed+n	Blocks get filled early during chunked prefill; blocks_used too large in Example 1, possibly causing unnecessary preemption
Not zeroing num_computed after preemption	Only 1 token is scheduled on re-entry: the KV was lost with the released blocks but never recomputed (step 4 of Example 2 becomes {a2: 1} instead of {a2: 4})
Clearing output on preemption	Final output differs from the no-preemption run (first token generated again)
Putting it back at the tail of waiting	The preempted sequence starves, or ordering breaks under pressure
Calling sample_fn before prefill is done	Extra tokens in output, max_tokens reached early
Using continue in phase 2 to skip ones that don't fit	Violates FCFS: a3 gets scheduled in Example 3

Level 3 · Walkthrough (code: see the commented solutions in Appendix C)

Answers to the key questions

1. decode: 1; chunked prefill: the uncomputed remainder of the prompt (plus 0 outputs).
2. Use `num_computed + n`. For a2 in step 1 of Example 1: num_tokens=5 gives 2 blocks, 0+2=2 gives 1 block; 1 block is correct.
3. The last one in running. After preempting someone else, retry X; if X itself is preempted, skip it and don't advance the index (running is already shorter).
4. Reset: num_computed=0, num_blocks=0 (and subtract from blocks_used), status=WAITING; keep: output. It recomputes prompt_len + len(output) tokens. Goes to the front.
5. break. max_num_seqs is only an admission condition in phase 2 (everything in phase 1 is already in running).
6. When num_computed is still < num_tokens after adding (the prefill chunk isn't finished), don't sample. On finish: remove from running and release blocks, put into finished (and return it).
7. In an instance field, e.g. `self._scheduled`.

Trace: preempting yourself (`num_blocks=3, block_size=2, budget=64` ; a1 and a2 both prompt 2, max_tokens 3; fill in the ?? yourself first, answers below)

step	Phase 1	Phase 2	Returns	blocks_used (after update)	a2.output
1	running is empty	a1 n=2 → 1 block; a2 n=2 → 1 block	`{a1:2, a2:2}`	2	`[200]`
2	a1 n=1, needs 2 blocks, +1 → 3 ✓; a2 n=1, needs 2 blocks, +1 → 4 ✗ → preempt running[-1] = ?? → ??	a2 at the head: n=??, needs ?? blocks → ??	??	2	??
3	a1 n=1, no new block needed	a2 still doesn't fit	`{a1:1}`	0 (a1 finishes)	`[200]`

© 2026 kzhou003

4	running is empty	a2 n=??	??	2	??
5	a2 decode	-	`{a2:1}`	0	`[200, 201, 202]`

Answers: in step 2 the victim is a2 itself → skip; in phase 2 a2 n=3 (2+1), needs 2 blocks, 2+2>3 → break; returns `{a1: 1}`; a2.output is still `[200]`. step 4: n=3, returns `{a2: 3}`, output becomes `[200, 201]`. num_preemptions = 1.

Self-check: write Examples 1, 2, 3 and the trace above as a few lines of script each and run them; at every step assert `blocks_used <= num_blocks` and `blocks_used == sum(s.num_blocks for s in running)`; after all sequences finish, blocks_used must be back to 0.

A-17 · BlockManager: paged KV + prefix caching

Level 1 · I/O examples

Example 1: prefix cache off, slot and append crossing a block boundary (`block_size=4, num_blocks=6` , free list hands out blocks FIFO)

```
bm = BlockManager(num_blocks=6, block_size=4)
bm.allocate("x", [11, 12, 13, 14, 15, 16])   # -> [0, 1]      (6 tokens -> ceil(6/4)=2 blocks)
bm.num_free_blocks                            # -> 4           free list [2, 3, 4, 5]
bm.slot("x", 5)                               # -> 5           = table[5//4]*4 + 5%4 = 1*4 + 1
bm.append_slot("x", 17)                       # -> 6           pos=6, block 1 still has room, no allocation
bm.append_slot("x", 18)                       # -> 7           pos=7, block 1 is now full
bm.append_slot("x", 19)                       # -> 8           pos=8, needs a 3rd block -> table [0, 1, 2]
bm.num_free_blocks                            # -> 3
bm.free("x")                                  # returns 2, 1, 0 in reverse -> free list [3, 4, 5, 2, 1, 0]
bm.allocate("y", [1, 2, 3])                   # -> [3]
```

Key point: append takes a new block only when "the current token lands in a logical block that doesn't exist yet"; free returns blocks in reverse order to the tail of the free list.

Example 2: prefix sharing + refcounts + evictable pool (`block_size=2, num_blocks=8, enable_prefix_cache=True`)

```
bm.allocate("a", [1, 2, 3, 4, 5])   # -> [0, 1, 2]   block0=[1,2] block1=[3,4] are full, hashes registered;
block2=[5] not full
bm.num_cached_tokens("a")           # -> 0
bm.allocate("b", [1, 2, 3, 4, 9])   # -> [0, 1, 3]   (5-1)//2 = 2 blocks matchable, both hit
bm.num_cached_tokens("b")           # -> 4
bm.num_free_blocks                  # -> 4           ref = [2, 2, 1, 1, 0, 0, 0, 0]
bm.free("a")                        # block2 ref->0, no hash -> free list; block1, block0 ref->1
bm.num_free_blocks                  # -> 5
bm.free("b")                        # reverse: block3 -> free list; block1, block0 ref->0 with hash -> evictable
pool [1, 0]
bm.num_free_blocks                  # -> 8           6 free + 2 evictable
bm.allocate("c", [1, 2, 3, 4])      # -> [0, 4]      only (4-1)//2 = 1 block matchable!
bm.num_cached_tokens("c")           # -> 2           block0 "revived" from the evictable pool; pool left with
[1]
```

Note c: the second block `[3,4]` is clearly in the cache, but the "at least 1 token must be computed" rule forbids hitting it, so it gets a fresh block 4.

Example 3: hitting an evictable block also consumes capacity (the atomicity trap) (`block_size=2, num_blocks=3, enable_prefix_cache=True`)

```
bm.allocate("a", [1, 2, 3, 4, 5]); bm.free("a")
# free list [2], evictable pool [1, 0], num_free_blocks = 3
bm.allocate("b", [1, 2, 3, 4, 5, 6, 7])   # needs 4 blocks; hits block0, block1 (both in the evictable pool)
# -> raise OutOfBlocks
bm.num_free_blocks                      # -> 3, free list [2], evictable pool [1, 0] (completely unchanged)
bm.allocate("b", [1, 2, 3, 4, 5, 6])    # -> [0, 1, 2], num_cached_tokens = 4, num_free_blocks -> 0
```

First allocate: it needs 4-2=2 new blocks, but 2 of the 3 "available" blocks are exactly the two being hit, so only 1 block can actually serve as a new block. It must fail, and it must not touch any state before failing.

Key questions

1. A prompt has 10 tokens, `block_size=4` : how many blocks does it need? Which of them are "full blocks" that get a hash? Does the last half-full block enter the prefix cache?

© 2026 kzhou003

2. A prompt has exactly 8 tokens, `block_size=4`, and both blocks' contents are already cached. How many blocks may hit? What is `num_cached_tokens`? If everything hit, what could the model still produce this step?
3. Two blocks both contain `[7, 8]`, but one is preceded by `[1, 2]` and the other by `[5, 6]`. Should their hashes be equal? What does the "previous hash" in the hash chain protect against?
4. A hit block may have ref>0 (another sequence is using it) or ref==0 (sitting in the evictable pool). What must allocate do in each case? Which one reduces the "available block count"?
5. When you take a block from the evictable pool to hold **new content**, besides removing it from the pool, which two records must you clean up? What weird behavior shows up if you don't?
6. What is the new token position `pos` in `append_slot` (in terms of which stored quantity)? Under what condition do you need a new block? If taking a block fails, which state have you already changed?
7. Why does `free` walk the block table in **reverse**? Think about "LRU evicts the oldest first": after reversing, who gets evicted first, and how does that help later hit rates?

Details

- **Start of the hash chain**: the docstring says `h_i = hash((h_{i-1}, tuple(block_i_tokens)))`. Block 0 has no predecessor, so pick a fixed sentinel (e.g. `None`) as `h_{-1}`. allocate and append_slot must compute hashes with the same rule, otherwise blocks filled by append will never be hit by allocate.
- **"Only match within the first (len(tokens)-1)//block_size blocks"**: this is the upper bound on matchable blocks, and matching is **contiguous from the start**: stop at the first miss (a prefix-cache hit must be a prefix). `num_cached_tokens = hit blocks * block_size`, always a multiple of block_size.
- **num_free_blocks = free + evictable**: evictable blocks have ref==0 and can be reclaimed any time, so they count as "available". `can_allocate(num_tokens)` is only the rough check `ceil(num_tokens/bs) <= num_free_blocks` (ignoring cache hits); allocate must do a more precise check internally (see Example 3).
- **slot**: `slot(seq, pos) = table[pos // bs] * bs + pos % bs`. It is the index into the flattened physical KV cache; P09's `write_kv` uses it.
- **enable_prefix_cache=False**: no hashes are ever registered, so freed blocks all go back to the free list and the evictable pool stays empty forever; num_cached_tokens is always 0.
- **Multiple copies of the same content**: in Example 2, c's second block (block 4) has the same content and hash as block 1. In the hash table one hash points to only one physical block; when registering, if it already exists, keep the old one. When evicting a block, delete the mapping only if the hash table entry **currently points to this block**; otherwise you delete someone else's valid mapping.

Level 2 · Approach & pitfalls

- **State design** (suggested):

Structure	Purpose
`free: deque[int]`	Free blocks never cached / without hash; take with popleft, return with append
`ref: list[int]`	Refcount per physical block
`block_hash: list[int|None]`	Hash currently registered for each physical block (None = not in cache)
`hash_to_block: dict[int,int]`	Hit lookup
`evictable: OrderedDict[int,None]`	Blocks with ref==0 that still have a hash; insertion order = LRU order
`tables / tokens / hashes / cached: dict[seq_id, ...]`	Per-sequence block table, all tokens, hash chain, hit token count

- **allocate = compute, then look up, then mutate**. (1) Compute the whole hash chain over full blocks; (2) within the matchable range, look up hash_to_block from the start, stopping at a miss; (3) check capacity; (4) actually mutate (hit blocks ref+1, removing from the evictable pool when needed; take a new block for each remaining block and register hashes for full ones). Any raise must happen before (4).
- **Capacity check formula**: `new blocks = total blocks - hits`; `truly available = num_free_blocks - number of hit blocks with ref==0`. If you only write `new blocks <= num_free_blocks`, Example 3 "passes the check" and then raises only when taking the 2nd new block; by then the hit blocks' refs have already been incremented and the state is corrupted.
- **Extract a take-block helper** (e.g. `_take_block()`): if the free list is non-empty, popleft; otherwise pop the **oldest** from the evictable pool and delete its hash registration (both hash_to_block and block_hash); otherwise raise OutOfBlocks; finally set ref to 1. Both allocate and append_slot use it.
- **Order in append_slot**: `pos = len(tokens[seq])`; if `pos // bs == len(table)`, take a new block first (this step can raise, so put it **before any mutation**); then append the token to tokens; if `(pos + 1) % bs == 0` the block just

© 2026 kzhou003

became full: use the last hash in the chain as parent to compute the new hash, append it to the chain, and register it. Finally return `slot(seq, pos)`.

- **free**: iterate in reverse, ref-=1; when it hits zero check block_hash: present → put into the evictable pool (at the "newest" end); absent → return to the free list. Don't forget to delete the sequence's table/tokens/hashes/cached records.
- **Common bugs and symptoms**:

bug	symptom
Hit a ref==0 block without removing it from the evictable pool	A later allocate evicts and overwrites this "in-use" block → two sequences share a block with different content; num_free_blocks too large
Didn't delete the hash registration on eviction	The evicted block holds new content, yet the old prefix can still "hit" it (see the eviction trace in level 3)
Hashing the half-full last block too	A new long prompt hits a block that holds only part of its tokens
Using len//bs as the matchable bound	When the prompt is an exact multiple of bs and fully hits, num_cached_tokens == len and the model has no token to compute
append appends the token before taking the block	After OutOfBlocks, tokens has one extra entry and all later slot computations are wrong

Level 3 · Walkthrough (code: see the commented solutions in Appendix C)

Answers to the key questions

1. 10 tokens → 3 blocks; the full blocks are blocks 0 and 1 (tokens 0..7), each with a hash; block 2 has only 2 tokens and doesn't enter the cache.
2. Only (8-1)//4 = 1 block may hit, num_cached_tokens = 4. If everything hit, no token would need a forward pass, there would be no logits at the last position, and no next token could be sampled.
3. No. The chained hash encodes the prefix, preventing a false hit on a block with the same content but a different prefix (KV depends on the whole prefix, not just this block's tokens).
4. ref>0: just ref+1. ref==0: first remove it from the evictable pool, then ref=1; it was counted in num_free_blocks, and once hit it's no longer available, so it reduces "the number usable as new blocks" by 1.
5. Set block_hash[b] to None, and remove the hash_to_block mapping pointing to b. Without cleanup: later requests with the same prefix "hit" a block already overwritten with other content, and KV gets corrupted.
6. pos = current token count (the index of the new token). Condition: pos // bs == len(table). Taking the block is the first step, so on failure nothing has been changed yet.
7. In reverse, the last block enters the evictable pool first and becomes the "oldest", so it's evicted first; earlier prefix blocks (more likely shared by many requests) are evicted later.

Trace: a block filled by append enters the cache and is later evicted (`block_size=2, num_blocks=3, enable_prefix_cache=True`)

Operation	Returns	Free list	Evictable pool (old→new)	Notes
`allocate("a",[7,8,9])`	`[0, 1]`	`[2]`	`[]`	block0=[7,8] full, register h0; block1=[9] not full
`append_slot("a",10)`	`3`	`[2]`	`[]`	pos=3 → block1 offset 1; block1=[9,10] just became full, compute h1 with parent h0 and register
`free("a")`		`[2]`	`[1, 0]`	Reverse: block1 enters the pool first (oldest)
`allocate("b", [7,8,9,10,11])`	`[0, 1, 2]`	`[]`	`[]`	2 blocks matchable, both hit, num_cached_tokens=4
`free("b")`		`[2]`	`[1, 0]`	block2=[11] has no hash → free list
`allocate("z",[50..55])`	`[2, 1, 0]`	`[]`	`[]`	Uses free block 2 first, then evicts 1, 0 and deletes the h1, h0 registrations
`free("z"); allocate("b2", [7,8,9,10,11])`	??			num_cached_tokens = ?? (hint: is h0 still there?)

Answer for the last row: returns 3 new blocks, num_cached_tokens = 0.

Self-check list: run Examples 1-3 line by line; then build your own scenario where "one block is shared by two sequences, free one, then free the other", and confirm that after the first free num_free_blocks only grows by the non-shared blocks.

© 2026 kzhou003

A-18 · Batch-invariant sampler (per-request seed)

Level 1 · I/O examples

Example 1: repetition penalty

```
logits = torch.tensor([3.0, -1.0, 0.5, -0.2])
apply_repetition_penalty(logits, prev_tokens=[1, 3, 3], penalty=1.5)
# -> tensor([ 3.0, -1.5, 0.5, -0.3])
logits                    # still [3.0, -1.0, 0.5, -0.2] (must not modify in place)

s = Sampler()
s.sample(torch.tensor([[3.0, -1.0, 2.5, -0.2]]), ["r"],
         [SamplingParams(temperature=0.0, repetition_penalty=1.5)], [[0]])
# -> tensor([2])          token 0: 3.0/1.5 = 2.0 < 2.5, greedy picks token 2 instead
```

token 3 appears twice in the history but is penalized only once ("every distinct previous token"); tokens 0 and 2, which never appeared, are unchanged.

Example 2: kept sets for top_k / top_p (T=1, one row of 4 tokens, probabilities after softmax are `[0.5, 0.25, 0.15, 0.1]`)

Params	Descending cumulative prob	Kept tokens	Renormalized probs after filtering
`top_p=0.4`	0.5 | 0.75 | 0.9 | 1.0	`{0}`	`[1, 0, 0, 0]`
`top_p=0.7`	same as above	`{0, 1}`	`[0.667, 0.333, 0, 0]`
`top_p=0.8`	same as above	`{0, 1, 2}`	-
`top_p=1e-9`	same as above	`{0}`	(keep at least 1)
`top_k=2`	-	`{0, 1}`	`[0.667, 0.333, 0, 0]`
`top_k=3, top_p=0.6`	renormalized after top_k: 0.556 | 0.833 | 1.0	`{0, 1}`	-

top_p=0.7: keeping only token 0 gives mass 0.5 < 0.7, not enough; adding token 1 reaches 0.75 ≥ 0.7 → stop. Note the last row: top_p is computed on the distribution **after** top_k. Effect of temperature on the distribution: T=0.5 → `[0.725, 0.181, 0.065, 0.029]`, T=2 → `[0.370, 0.262, 0.203, 0.166]`.

Example 3: batch invariance (one row of logits `row = [1.0, 0.5, 0.0, -0.5, 0.2, 0.9]`, request q uses `SamplingParams(seed=7)`, one draw per sample call)

```
run 6 steps alone (batch contains only q)          -> [0, 5, 4, 0, 4, 5]
run 6 steps mixed with other requests (q's position/peers change every step,
  some peers seeded, some not)                     -> [0, 5, 4, 0, 4, 5]   # must be identical
wrong: whole batch shares one seed=7 generator     -> [5, 4, 1, 0, 0, 0]
after free("q"), rerun 3 steps with seed=7         -> [0, 5, 4]           # random stream restarts
```

The exact numbers depend on how you sample (here `torch.multinomial(probs, 1, generator=g)` after softmax); it's fine if your numbers differ. The properties to satisfy are **"alone == mixed" and "replay after free is identical"**.

Key questions

1. `prev_tokens=[4, 4, 4]`, penalty=2: is token 4's positive logit 2.0 penalized to 1.0 or 0.25?
2. Why are negative logits **multiplied** rather than divided? Does dividing -2.0 by 2 make the token more or less likely?
3. With temperature=0, what happens if you do `logits / temperature` first? Does repetition penalty still apply to greedy rows?
4. top_p's "smallest prefix whose mass reaches top_p": is the token that crosses the threshold kept? How do you express this test using "the cumulative mass before it"?
5. How should `top_k=-1` and `top_k >= V` be handled? If there's a tie for the k-th largest value, how many will your implementation keep?
6. When is each seeded request's generator created, where is it stored, and is it reused across steps? What do unseeded requests use?
7. Why can't you call `torch.multinomial(probs, 1, generator=g)` once on the whole `[B, V]` probability matrix, even if g was built from the seed?

Details

- **"do not mutate the input"**: `clone()` first, then modify. In sample, logits is the caller's tensor; modifying it in place makes the caller (e.g. when computing logprobs afterwards) see penalized values.

© 2026 kzhou003

- **The processing order is fixed**: penalty → (greedy?) → /T → top_k → top_p → sample. top_k and top_p just set the dropped positions to `-inf`; after softmax their probabilities are exactly 0, and the rest renormalize automatically.
- **top_p's "keep at least 1"**: even if top_p is tiny (e.g. 1e-9), the most probable token must be kept, so it degenerates to greedy. In practice you force position 0 of the sorted mask to not be dropped.
- **"created on first use from params.seed"**: the first time you see a req_id with a non-None seed, do `torch.Generator().manual_seed(seed)` and store it in a dict; reuse the same object every step afterwards so the random stream keeps advancing. `free(req_id)` removes it from the dict (no error if absent).
- **dtype / return value**: logits may be bf16/fp16; convert each row to `float()` before processing. Return a `[B]` `torch.long` tensor whose i-th element corresponds to row i.

Level 2 · Approach & pitfalls

- **Process row by row**: the simplest way to guarantee batch invariance is `for i in range(B)`, with each row using only its own logits, params, prev_tokens, and generator. Get it correct first; vectorization is a follow-up.
- **apply_repetition_penalty**: dedupe to get an index tensor → gather the values at those positions → `torch.where(v > 0, v / p, v * p)` → write back into the clone. If penalty==1 or prev is empty you can return directly (values unchanged, which still counts as "not modifying the input").
- **top_k**: `kth = torch.topk(row, k).values[-1]`, set `row < kth` to -inf (ties keep a few extra, which is acceptable). Only enable when `0 < k < V`.
- **top_p**: sort descending → softmax → cumsum; "cumulative mass before" = `cum - probs`; drop positions where it is already ≥ / > top_p; force-keep position 0; map back through the sort indices to set -inf at the original positions.
- **Generator management**: `dict[req_id, torch.Generator]`; requests with seed None can use one shared default generator (or the global RNG), since they aren't required to be reproducible anyway.
- **Drawing a token**: `torch.multinomial(softmax(row), 1, generator=g)`. One call per row, so request q's generator is consumed only by q, and the number of draws is independent of batch composition.
- **Common bugs and symptoms**:

bug	symptom
Penalizing repeated tokens multiple times	token 3 in Example 1 becomes -0.45 instead of -0.3
Dividing negative logits too	Penalized negative-score tokens become more likely to be chosen
top_p drops the token that crosses the threshold	With top_p=0.7 only {0} is left; with tiny top_p everything may be dropped → softmax gives NaN
All rows share one generator	"Mixed" result in Example 3 differs from "alone"
Building a fresh manual_seed generator every step	The same request draws the same random number every step; the distribution is clearly skewed
Not deleting the generator on free	A new request reusing an old id continues the old random stream; not reproducible

Level 3 · Walkthrough (code: see the commented solutions in Appendix C)

Answers to the key questions

1. 1.0: distinct, so divide only once.
2. -2.0/2 = -1.0 is larger → more likely to be chosen, the opposite of a penalty; multiplying by 2 gives -4.0, which actually lowers it.
3. Division by 0 gives inf/NaN. The threshold check must come before the division; penalty comes before greedy, so it still applies to greedy rows (second part of Example 1).
4. Kept. The test: drop a token only if the cumulative mass before it (excluding itself) has already reached top_p; the 0-th token's "mass before" is 0, so it's naturally kept.
5. Treat both as disabled. When keeping "≥ the k-th largest value", ties keep extra tokens, which is acceptable.
6. Created the first time that req_id is sampled, stored in a dict on the Sampler instance, reused across steps, deleted on free. Unseeded ones use a shared generator.
7. A single call consumes the same generator row by row in order, so the random numbers request q gets depend on how many rows come before it; and one g can't represent several requests with different seeds at once.

Trace: top_k=3, top_p=0.6, T=1 (original probs `[0.5, 0.25, 0.15, 0.1]`)

Stage	token0	token1	token2	token3
logits = log(p)	-0.693	-1.386	-1.897	-2.303
after top_k=3	-0.693	-1.386	-1.897	-inf
descending softmax	0.556	0.278	0.167	0

© 2026 kzhou003

cumulative mass before, cum - p	0	0.556	0.833	1.0
> 0.6 → drop?	keep (forced)	??	??	drop
final probs	??	??	0	0

Answer: token1 kept (0.556 hasn't reached 0.6), token2 dropped; final probs `[0.667, 0.333, 0, 0]`.

Self-check: (1) for each row of Example 2, compute the kept set with your filter function; (2) Example 3: run a seeded request alone for N steps, then in randomly shuffled batches for N steps; the result lists must be equal; (3) build two tokens with probs 0.8/0.2, draw a few thousand times, and check the frequencies are close.

A-19 · Paged KV write + decode attention (reference implementation)

Level 1 · I/O examples

Example 1: write_kv and padding (cache shape `[3, 2, 1, 1]`, i.e. 3 blocks, block_size=2, 1 kv head, head_dim=1, initially all 0)

```
k = [[[10.]], [[20.]], [[30.]], [[40.]]]          # [4, 1, 1]
v = k * 0.1
slot_mapping = tensor([4, 5, -1, 1])
write_kv(k_cache, v_cache, k, v, slot_mapping)

k_cache[:, :, 0, 0]  # -> [[ 0., 40.],     block0: offset1 <- token3 (slot 1)
                     #     [ 0.,  0.],     block1: nobody writes
                     #     [10., 20.]]     block2: slot 4,5 <- token0, token1
v_cache[:, :, 0, 0]  # -> [[0., 4.], [0., 0.], [1., 2.]]
```

token2's slot is -1 (padding), so 30 must not appear anywhere in the cache, in particular not in the last slot (block2 offset1).

Example 2: paged decode attention (`block_size=2, num_blocks=4, num_kv_heads=1, num_heads=2, head_dim=2, scale=1.0`)

```
# seq0: 3 tokens, physical blocks [2, 0] (non-contiguous and out of order)
#   K0 = [[1,0], [0,1], [1,1]]   V0 = [[1,2], [3,4], [5,6]]   written to slots [4, 5, 0]
# seq1: 1 token, physical block [3]
#   K1 = [[2,2]]                 V1 = [[7,8]]                 written to slot [6]
block_tables = [[2, 0],
                [3, 77]]          # 77 is garbage (even beyond num_blocks)
context_lens = [3, 1]
q = [[[1,0], [0,2]],              # seq0 head0 / head1
     [[1,1], [0,0]]]              # seq1

paged_decode_attention(q, k_cache, v_cache, block_tables, context_lens, 1.0)
# -> [[[3.0, 4.0], [3.8099, 4.8099]],
#     [[7.0, 8.0], [7.0,    8.0   ]]]
```

seq0 head0: scores = $q \cdot K^T$ = [1, 0, 1] → softmax = [0.4223, 0.1554, 0.4223] → weighted V = [3.0, 4.0]. head1: scores = [0, 2, 2] → [0.0634, 0.4683, 0.4683] → [3.81, 4.81]. seq1 has only 1 token, so the output is V1 regardless of q; reading the 77 in the table would go straight out of bounds.

Example 3: GQA head mapping (`num_heads=8, num_kv_heads=2`, group = 4)

query head h	0	1	2	3	4	5	6	7
Correct: h // group	0	0	0	0	1	1	1	1
Wrong: h % num_kv_heads	0	1	0	1	0	1	0	1

When expanding K from `[n, KVH, D]` to `[n, H, D]`, which PyTorch repeat operation gives the first row, and which gives the second?

Key questions

1. slot 7, block_size 4: which physical block and which offset? Conversely, after flattening the cache with `view(-1, KVH, D)`, how does slot relate to the first-dimension index?
2. If you directly do `k_cache.view(-1, KVH, D)[slot_mapping] = k`, where does slot -1 get written? Does it raise?
3. context_len = 5, block_size = 4: how many leading entries of the block table are valid? After gathering those blocks, how many token positions are there in total, and how many must be cut off?
4. What dtype is block_tables? Can it be used directly for advanced indexing?

© 2026 kzhou003

5. Under GQA, query head h maps to kv head `h // (H // KVH)`: which of `repeat_interleave` and `repeat` matches this?
6. "Softmax in fp32": which tensors need to be converted to fp32? What dtype is the final output?
7. At which step is scale applied? What happens if you apply it after softmax?

Details

- **Cache layout** `[num_blocks, block_size, num_kv_heads, head_dim]`: the first two dims together form the "slot" dimension. `slot = block * block_size + offset`, so `cache[slot // bs, slot % bs]` and `cache_flat[slot]` (with the first two dims flattened) refer to the same position.
- **"slot -1 marks a padding token that must NOT be written"**: CUDA graphs pad the batch to a fixed size, and padding tokens have slot -1. Python/PyTorch negative indexing interprets -1 as "the last one", so you must filter them out with a mask first rather than rely on an out-of-bounds error.
- **"entries past the last used block are garbage"**: block_tables is padded into a `[B, max_blocks]` rectangle; for sequence b only the first `ceil(context_lens[b] / block_size)` entries are meaningful; later values can be anything, even beyond num_blocks.
- **The last block is not full**: concatenating nb blocks gives nb*block_size positions, but only the first context_len are this sequence's tokens; the rest are other sequences' or stale data and must be cut off (you can't rely on cache values being 0, because a zero vector's score is 0, not -inf).
- **one query token per sequence**: in decode each sequence has only its latest 1 query, which can see the entire context (including itself, already written by write_kv first), so no causal mask is needed.

Level 2 · Approach & pitfalls

- **write_kv (vectorized)**: `valid = slot_mapping >= 0` → take only valid slots and the corresponding k/v rows → split into `blocks = slots // bs, offs = slots % bs` → `k_cache[blocks, offs] = k[valid]` (same for v). Writes in place, returns None.
- **decode (write the loop version first)**: for each b: n = context_lens[b], nb = ceil(n / bs), `table = block_tables[b, :nb].long()`.
- **gather + reshape + truncate**: `k_cache[table]` has shape `[nb, bs, KVH, D]` → reshape to `[nb*bs, KVH, D]` → take the first n rows. The table gives the physical order of the blocks, so the logical order comes out right automatically.
- **GQA**: `k.repeat_interleave(H // KVH, dim=1)` expands `[n, KVH, D]` to `[n, H, D]`, so head h's K comes from kv head h // group. Alternatively reshape q to `[KVH, group, D]` and avoid copying K.
- **Attention**: `scores[h, j] = (q[b, h] · k[j, h]) * scale` (fp32), softmax over j; `out[b, h] = Σ_j probs[h, j] * v[j, h]`; finally `.to(q.dtype)` into the output tensor `[B, H, D]`. einsum `"hd,nhd->hn"` / `"hn,nhd->hd"` is convenient.
- **Common bugs and symptoms**:

bug	symptom
Not filtering slot -1	The last slot of the last physical block is overwritten by padding tokens; long sequences intermittently give wrong results
Reading the whole table row	IndexError (garbage out of bounds), or reading someone else's KV and mismatching dense
Not cutting off the extra positions of the last block	Wrong results for sequences whose context length isn't a multiple of bs (seq0 in Example 2 would see 1 extra position)
GQA using repeat instead of repeat_interleave	Correct when num_kv_heads=1, wrong when >1: very subtle
Indexing directly with int32 block_tables	Works on CPU in recent PyTorch, but not in older versions / some backends; converting to long is safest
Softmax done in bf16	Deviation on the order of 1e-2 from the fp32 reference

Level 3 · Walkthrough (code: see the commented solutions in Appendix C)

Answers to the key questions

1. Block 1, offset 3; after flattening, the first-dimension index is the slot.
2. It writes to the last flattened position (the last offset of the last block), with no error: that's exactly the danger.
3. The first 2 entries; 2*4=8 positions, cut off the last 3, keep only the first 5.
4. Usually int32 (vLLM uses it to save memory / feed kernels); recent PyTorch accepts int32 indices too, but converting to `long` before advanced indexing is safest.
5. `repeat_interleave` ([a,b] → [a,a,a,a,b,b,b,b]); `repeat` gives [a,b,a,b,...], which corresponds to h % KVH.
6. q, k, v can all be converted to fp32 to compute scores / softmax / weighted sum; convert the output back to q.dtype.
7. Multiply the scores before softmax; applying it afterwards just scales the whole output, and the attention distribution is wrong.

Trace: gather for seq0 in Example 2 (fill in the ?? yourself)

© 2026 kzhou003

Step	Value
n = context_lens[0]	3
nb = ceil(n / bs)	??
table = block_tables[0, :nb]	??
k_cache[table] shape	[??, 2, 1, 2]
after reshape (logical positions 0..3)	[[1,0], [0,1], [1,1], ??]
truncate to first n	[[1,0], [0,1], [1,1]]
head0: scores = q·K^T × 1.0	[1, 0, 1]
softmax	[0.4223, 0.1554, 0.4223]
out = probs @ V	[??, ??]

Answers: nb = 2; table = [2, 0]; shape [2, 2, 1, 2]; the 4th position holds block 0 offset 1, here the zero vector [0, 0] (never written), which doesn't belong to this sequence: without truncation its score is 0, not -inf, and it would receive attention; out = [3.0, 4.0].

Self-check: write your own dense version (attention directly over contiguous K/V tensors), randomly generate several sequence lengths (including 1, exact multiples of bs, and non-multiples), shuffle physical block assignment, fill the tail of the block table with large garbage values, and cross-check against the paged version with `torch.testing.assert_close`.

A-20 · SharedMemoryConnector: cross-process put/get protocol

Level 1 · I/O examples

Example 1: round trip + consume only once

```
c = ShmConnector(namespace="hv")
k = "req7_0_0"
c.put(k, b"abc")
os.path.exists(c.lock_path(k))   -> True        # the lock file remains after put (it means "written, not yet
read")
c.get(k)                         -> b"abc"
c.get(k)                         -> None        # the reader already unlinked the segment and removed the lock
file
os.path.exists(c.lock_path(k))   -> False
c.get("never_written")           -> None        # lock file doesn't exist

c.shm_name(k) == c.shm_name(k)   -> True        # deterministic: another process with the same key computes the
same name
len(c.shm_name(k))               -> e.g. 23     # namespace + short hash; must be well under 31
c.put(k, b"one"); c.put(k, b"two!")             # same key written twice (old segment not read)
c.get(k)                         -> b"two!"     # put finds the stale segment → replaces it
```

Example 2: lock held by someone else = "still writing"

```
c.put("job_0_3", b"xyz")
fd = os.open(c.lock_path("job_0_3"), os.O_RDWR)
fcntl.flock(fd, fcntl.LOCK_EX)       # simulate a writer that is still writing
c.get("job_0_3")   -> None           # doesn't block, returns immediately
fcntl.flock(fd, fcntl.LOCK_UN); os.close(fd)
c.get("job_0_3")   -> b"xyz"
```

Even if "someone else" is an fd from another `os.open` in the same process, flock still conflicts: a flock lock belongs to the open file description, not to the process.

Example 3: cleanup_prefix only cleans keys "I wrote, nobody has read yet, and the prefix matches"

```
for i in range(3): c.put(f"rA_0_{i}", bytes([i]))
c.put("rAx_0_0", b"q")
c.get("rA_0_1")             -> b"\x01"     # one got consumed
c.cleanup_prefix("rA_")     -> 2           # rA_0_0, rA_0_2; "rAx_0_0" doesn't start with "rA_"
c.cleanup_prefix("rA_")     -> 0           # already cleaned
c.get("rA_0_0")             -> None
c.get("rAx_0_0")            -> b"q"
```


© 2026 kzhou003

Writer exits first: another Python process puts a key and exits; 1 second later this process does get. If the writer didn't unregister the segment from its own resource tracker, the result is `None`, and on exit the writer prints `UserWarning: resource_tracker: There appear to be 1 leaked shared_memory objects to clean up at shutdown`; with unregistration you get the original data.

Key questions

1. There is no metadata channel between reader and writer. The reader knows only the key: how does it know which shm segment and which lock file to open? What does this require of `shm_name` (deterministic? length? charset?)
2. Why write an 8-byte length header? If you inferred the length from `shm.size` alone, how many bytes would the 3-byte payload in Example 1 read back as? And what about `put(k, b"")`?
3. `get` returns None for three reasons. What are they? Which stage of the writer's lifecycle does each correspond to?
4. In put, what's the difference between "take the lock, then create the segment" and "create the segment, then take the lock"? Which order lets a reader read half-written data?
5. Who is responsible for unlinking the segment? Why not the writer? When the writer process exits, who "kindly" unlinks it on its behalf?
6. How does `cleanup_prefix` tell that a key it wrote has "already been consumed"? (Hint: what does the reader delete after a successful get?)
7. Why must the reader copy the data out with `bytes(...)` before `close()`?

Details

- **POSIX shm names**: on POSIX, Python's `SharedMemory(name=...)` automatically prefixes the name with `/`, and macOS caps it at 31 bytes (including that `/`). A key like `"req-3f2a9c1e_0_17"` plus a namespace easily exceeds that, so typically use `namespace + "_" + first N chars of a hash`. You must use a deterministic hash (e.g. `hashlib`), not Python's `hash()`, which is randomly seeded per process.
- **Segment size is rounded up**: on macOS `SharedMemory(create=True, size=5).size` is `16384` (one page). The `buf` the reader sees is longer than the data written, hence the length header. Also `size=0` raises `ValueError: 'size' must be a positive number different from zero`; the length header conveniently gives an empty payload a valid size too.
- **flock semantics**: `fcntl.flock(fd, LOCK_EX)` blocks; `LOCK_EX | LOCK_NB` raises `BlockingIOError` when it can't acquire. The lock is advisory, effective only among participants that also use flock. The writer uses blocking mode, the reader non-blocking (get must not block).
- **Lock file = "an unconsumed key exists"**: put creates it with `O_CREAT`; the reader deletes it after reading. So "lock file doesn't exist" means either "never written" or "already read". When `lock_dir` is None pick a default directory (`/dev/shm` on Linux is an in-memory filesystem, otherwise use the temp dir).
- **resource tracker (Python < 3.13)**: each process has a tracker subprocess; `SharedMemory` registers the name on create (and, in 3.12, on attach), and `unlink()` unregisters it. On process exit, the tracker treats still-registered segments as leaks and unlinks them. The reader's "attach → unlink" is balanced; the writer's "create → no unlink" is not: ownership must be handed off explicitly.
- **"Stale segment"**: the same key was written before but not read (or left behind by a crashed process), so `create=True` raises `FileExistsError`. The spec requires replacing it, not failing or appending.

Level 2 · Approach & pitfalls

- **Only three pieces of state**: `lock_dir`, `namespace`, and a set of "keys I wrote" (dict or set, used by cleanup_prefix). `lock_path(key)` derives directly from `shm_name(key)`, e.g. `os.path.join(lock_dir, shm_name(key) + ".lock")`, so both sides can compute it independently.
- **Order in put**: `os.open(lock, O_CREAT|O_RDWR)` → `flock(LOCK_EX)` → create the segment (on `FileExistsError` attach the old one, close, unlink, then create) → unregister from tracker → write length header and data → `close()` → record the key → in `finally`, `LOCK_UN` and close the fd. The lock must be acquired **before** creating the segment: by the time a reader can see the segment the lock is guaranteed held, so the reader never reads a half-finished one.
- **Order in get**: `os.open(lock, O_RDWR)` (`FileNotFoundError` → None) → `flock(LOCK_EX|LOCK_NB)` (`BlockingIOError` → None) → attach the segment (`FileNotFoundError` → None) → read the length, copy the data → `close()` → `unlink()` → delete the lock file → in `finally` unlock and close the fd. **Every return-None branch must reach the unlock/close-fd**, otherwise the fd leaks and the lock stays held.
- **Why "lock file exists but the segment doesn't"** can happen: the writer has just created the lock file with `os.open` but hasn't flocked yet, and the reader grabs the lock first. Returning None here is correct (come back later); don't raise and don't delete the lock file.
- **The resource tracker fix**: after creating the segment, the writer calls `resource_tracker.unregister(shm._name, "shared_memory")` (`_name` is the real name with the `/` prefix), handing ownership to the reader; when the reader calls `unlink()` the library automatically unregisters its own registration. Wrap it in try/except for compatibility across Python versions.
- **BufferError trap**: `data = shm.buf[8:8+n]` gives a memoryview slice, and calling `shm.close()` right after raises `BufferError: cannot close exported pointers exist`. Copy with `bytes(...)` first. Writing is fine: `shm.buf[8:8+n] = data` is an assignment that keeps no view, so you can close directly.

© 2026 kzhou003

- **cleanup_prefix**: iterate over the keys you recorded (snapshot first); remove prefix matches from the record; if the lock file still exists (= nobody read it), attach + unlink the segment, delete the lock file, count +1; swallow `FileNotFoundError` if the segment or lock file is already gone. Already-consumed keys are also removed from the record, but not counted.

Level 3 · Walkthrough (code: see the commented solutions in Appendix C)
Lifecycle of one key (L = lock file, S = shm segment, T = registration in the writer's tracker)

Moment	L	flock	S	T	get(k) returns now
before put	no	-	no	-	None (lock file doesn't exist)
put: after os.open, before flock	yes	not held	no	-	None (segment doesn't exist)
put: writing while holding the lock	yes	writer	yes (incomplete)	registered → unregistered	None (BlockingIOError)
after put returns	yes	none	yes	no	data
writer process exits	yes	none	yes (because T doesn't have it)	-	data
after a successful get	no	-	no (reader unlinked)	-	None

If the writer did **not** unregister: S in row 5 becomes ?? (answer: no; the writer's tracker unlinked it on exit, and the reader gets None).
Answers to the Level-1 questions

- Each side derives the names with `shm_name(key)` / `lock_path(key)`: deterministic (same result across processes; use hashlib, not hash()), short (≤ about 30), safe characters only (hex is fine).
- Relying on `shm.size` reads a whole page (16384 bytes on macOS); an empty payload can't even be created because size=0.
- Three reasons for None: lock file doesn't exist (not written / already read); lock is held (being written); segment doesn't exist (writer just created the lock file but not the segment yet).
- Segment first, lock second: a reader may grab the lock in between and read an unfinished segment. Lock first, segment second makes that impossible.
- The reader unlinks: only the reader knows "I'm done with it". When the writer exits, its resource tracker unlinks on its behalf: exactly what we need to prevent.
- The reader deletes the lock file after a successful get, so cleanup checks `os.path.exists(lock_path(key))`.
- A memoryview slice holds a buffer export, so close raises BufferError; and after close/unlink the memory may already be invalid.

Skeleton Think about it: get still holds the flock while deleting the lock file. This is safe under the assumption "each key has one consumer and keys are never reused"; if the same key could be written repeatedly, what race would appear?

A-21 · ChunkAdapter: background send/recv threads with ordering/cleanup semantics

Level 1 · I/O examples
Example 1: a None payload takes no chunk id; final is sent even if its payload is None

```
conn = InMemoryConnector()
snd = ChunkAdapter(conn, stage_id=2, src_stage_id=None, build_payload=lambda n: n if n % 3 else None)
rcv = ChunkAdapter(conn, stage_id=3, src_stage_id=2,    build_payload=lambda s: s)
snd.start(); rcv.start()
rcv.load_async("u9")
for n in range(1, 7):
    snd.save_async("u9", n, is_final=(n == 6))
# after a short wait
rcv.poll_ready() -> {"u9": [{"data": 1, "finished": False},
                            {"data": 2, "finished": False},
                            {"data": 4, "finished": False},
                            {"data": 5, "finished": False},
                            {"data": None, "finished": True}]}   # n=6's payload is None, but it's final
conn.puts        -> ["u9_2_0", "u9_2_1", "u9_2_2", "u9_2_3", "u9_2_4"]   # 6 saves, 5 contiguous ids
```

Note the numbers in the key: the sender uses **its own** `stage_id` (2), and the receiver uses `src_stage_id` (2) to build the same key.
Example 2: snapshot isolation + poll_ready "takes everything since last time"

© 2026 kzhou003

```
# a fresh pair of adapters: snd(stage_id=0), rcv(src_stage_id=0), build_payload = lambda s: list(s)
live = ["a"]
rcv.load_async("v")
snd.save_async("v", live, is_final=False)
live.append("b")                      # the scheduler thread keeps mutating the live object
snd.save_async("v", live, is_final=True)
live.append("c")
# wait → received data in order: ["a"], ["a", "b"]; conn.puts == ["v_0_0", "v_0_1"]

# another fresh pair (build_payload = lambda s: s), rcv.load_async("q") already called:
snd.save_async("q", 1, False); snd.save_async("q", 2, False)   # wait
rcv.poll_ready() -> {"q": [{"data": 1, "finished": False}, {"data": 2, "finished": False}]}
rcv.poll_ready() -> {}                                         # already taken
snd.save_async("q", 3, True)                                   # wait
rcv.poll_ready() -> {"q": [{"data": 3, "finished": True}]}
```

Also: if a chunk was put before `load_async`, it is still received after `load_async`: the connector keeps it until it's fetched with get.

Example 3: cancel_sender swallows later non-final chunks, but the final chunk (data=None) always arrives

```
# a fresh pair of adapters (stage_id=0 / src_stage_id=0); rcv.load_async("w") already called
# build_payload blocks on a gate until we release it
snd.save_async("w", "p0", False)       # save thread starts building "p0", stuck on the gate
for p in ["p1", "p2", "p3"]:
    snd.save_async("w", p, False)      # queued
snd.save_async("w", "end", True)
snd.cancel_sender("w")
gate.set()                             # release
# the receiver eventually gets:
[{"data": "p0", "finished": False},   # the chunk already being built before cancel is sent as usual
 {"data": None, "finished": True}]    # p1..p3 dropped; final still sent, but data=None
conn.puts -> ["w_0_0", "w_0_1"]        # ids still contiguous
```

Whether p0 is sent depends on whether it had already passed the cancel check at cancel time; either is acceptable. But ids must be contiguous and final must arrive.

Key questions

1. `save_async` runs on the scheduler thread and must return immediately. What are the at most two things it may do? Why can't `build_payload` be called here?
2. When should chunk_id be incremented: at enqueue, after build, or after a **successful** put? Why does "increment at enqueue" create a hole in Example 1, and what happens when the receiver hits a hole?
3. When `build_payload` returns None, when do you skip and when must you still put? After the final chunk is cancelled, what should `data` be?
4. How does the receiver know the next key to get? What state does each request need to remember? After a request receives `finished=True`, should it keep being polled?
5. The recv thread calls `connector.get` outside the lock (possibly slow). During that window, the scheduler thread calls `cleanup_receiver(req)` and then `load_async` again for the same req id. When the recv thread comes back, how can it tell "this data belongs to the obsolete registration"? Is checking `req in registered` enough?
6. With 10 requests being polled but none of them having data, what should the recv thread do? With no requests being polled at all? What's the difference between the two?
7. `stop()` must "return promptly". If the save thread is sleeping on `Condition.wait()`, how do you wake it? If it starts waiting just an instant before your notify, could it sleep forever?

Details

- **Two background threads, two kinds of caller**: `save_async` / `cancel_sender` / `load_async` / `poll_ready` / `cleanup_receiver` are all called on the scheduler thread; the SAVE thread only does build + put; the RECV thread only does get + commit. One ChunkAdapter object owns both halves, but a given stage usually uses only one half (stage 0 only sends, the last stage only receives).
- **Key format**: sender `f"{req}_{stage_id}_{chunk_id}"`, receiver `f"{req}_{src_stage_id}_{next_id}"`. On both sides chunk_id starts at 0 and is counted per request; the receiver can only fetch one by one in id order, so the sender's ids must be **contiguous**.
- **Chunk dict**: the object put is always `{"data": payload, "finished": is_final}`; `poll_ready()` returns these dicts unchanged, in id order, inside `{req: [...]}`; requests with nothing new don't appear in the result.
- **"OUTSIDE any lock"**: both `build_payload` and `connector.get` can be slow. If you call them while holding a lock the scheduler thread also needs, the scheduler thread gets slowed down, which defeats the whole point of this layer. Locks only protect small shared dicts/sets, and should be held for microseconds.
- **per-registration token**: on `load_async`, create a unique object for this request (e.g. `object()`) and store it; delete it on cleanup; the recv thread records the token before get, and after get compares under the lock whether "the currently registered token is **the same object**" (using `is`). If not, discard the data.

© 2026 kzhou003

- **Condition and timeouts**: after `Condition.wait(timeout=...)` returns you must recheck the condition (it may be a timeout, a spurious wakeup, or a stop wakeup). Give wait a short timeout (e.g. 0.1 s): even if a notification is lost you just sleep a bit longer instead of hanging forever.

Level 2 · Approach & pitfalls

- **Sender data structures**: a FIFO queue of `(req, snapshot, is_final)` (a `deque`; append/popleft are themselves thread-safe) + a Condition to wake the save thread; `next_send_id: dict[req, int]`; `cancelled: set[req]`. Since there is only one save thread processing FIFO, chunks of the same request are naturally ordered.
- **One iteration of the save thread**: pop an item → check cancelled (briefly holding the lock) → cancelled and non-final: drop → cancelled and final: data=None → otherwise build outside the lock → data is None and non-final: skip (no id consumed) → put to the current id's key → on success id+1; if final, clear this request's id counter and cancelled flag (reset for the next request with the same name). Don't +1 if put raises.
- **Receiver data structures**: `registered: dict[req, token]`, `next_recv_id: dict[req, int]`, `ready: dict[req, list]` (committed results waiting to be polled), a queue of requests to poll + a Condition. `load_async` should be idempotent for already-registered requests (don't enqueue twice, don't reset the id).
- **One round of the recv thread**: poll each request currently in the queue once: under the lock read (token, next_id), drop it if it's no longer registered; get outside the lock; no data → put it back at the tail; data → compare the token **under the lock** → only if it matches append to `ready[req]` and id+1; on finished unregister it (don't put it back), otherwise put it back at the tail. After the round: queue empty → sleep on the Condition (with timeout); there are requests but none made progress this round → sleep ~1 ms; progress → next round immediately.
- **poll_ready atomic swap**: under the lock replace `ready` with a new empty dict and return the old one. Don't iterate `ready` outside the lock (the recv thread may be appending to it).
- **cleanup_receiver is a commit fence**: under the lock delete registered[req], next_recv_id[req], ready[req]. Because the recv thread's "compare token + commit" is under the same lock, after cleanup returns any commit has either already happened (and been deleted by cleanup) or will see a token mismatch and be dropped. **Common bug**: checking registration only before get: a chunk cleaned up during get then shows up in poll_ready after cleanup.
- **stop()**: set a `threading.Event`, do `with cond: cond.notify_all()` on each of the two Conditions, then join the threads (with timeout). Thread loop conditions check this Event; check the Event once more before wait so you don't miss the wakeup. Make the threads daemon so they don't hang the interpreter at the end of tests.
- **Symptoms of other common bugs**: no deepcopy → Example 2 receives `["a","b","c"]`; +1 at enqueue → Example 1 jumps from `u9_2_2` to `u9_2_4`, and the receiver is stuck forever waiting for `u9_2_3`; cancel swallows final too → downstream never receives finished; busy-loop when idle → 100% CPU, other threads (including tests) slow down.

Level 3 · Walkthrough (code: see the commented solutions in Appendix C)

Step-by-step trace of the save thread in Example 1 (`build = n if n % 3 else None`, final at n=6)

Dequeued item	build result	Action	Key put	next_send_id[u9] afterwards
(u9, 1, F)	1	put	u9_2_0	1
(u9, 2, F)	2	put	u9_2_1	2
(u9, 3, F)	None	skip	-	2
(u9, 4, F)	4	put	u9_2_2	3
(u9, 5, F)	5	put	u9_2_3	4
(u9, 6, T)	None	final → still put, data=None	u9_2_4	(deleted, reset)

The corresponding recv thread: next_recv_id goes 0→1→2→3→4→5; after getting `finished=True` from `u9_2_4`, u9 is unregistered and not put back in the poll queue.

Timeline of the cleanup race (connector get is slow)

Time	Scheduler thread	recv thread	registered[q]
t0	load_async(q)		tok1
t1		reads tok1, id=0 under the lock; enters get("q_0_0") outside the lock...	tok1
t2	cleanup_receiver(q)	(still in get)	none
t3	load_async(q) (new request, same name)	(still in get)	tok2
t4		get returns data; under the lock compare registered[q] is tok1 → ??	tok2

© 2026 kzhou003

Answer: False → discard. If you only checked `q in registered`, t4 would be True, and the old request's data would leak into the new request.

Answers to the Level-1 questions

- save_async only does: deepcopy snapshot + enqueue (and notify). build_payload may be slow and would block scheduling.
- Increment only after a successful put. Incrementing at enqueue makes the skipped None and failed puts each take an id; the receiver polls in id order and waits forever for a key that doesn't exist.
- None and non-final → skip; final → always put; cancelled final → data=None.
- Per request: registration token, next id, list of committed-but-not-taken chunks. After finished, stop polling and unregister.
- Compare token identity (`is`); only checking membership in registered isn't enough (see the table above).
- Requests but no data: back off briefly (~1 ms) then poll again; no requests: sleep on the Condition until load_async wakes it (with a timeout as a safety net).
- stop sets the Event and calls notify_all; wait has a timeout, so even if the notification is missed it wakes at most one timeout period late, then checks the Event and exits.

`start()` creates two daemon threads; `stop()` sets the Event → `notify_all` on both cvs → join(timeout).

A-22 · Thinker→Talker input construction (including "off by one row")

Level 1 · I/O examples

Example 1: finished by stop, one extra row (D=2; row r of hidden has value r, so you can see which row is which)

```
prompt_ids  = [11, 12, 13, 14]          # P = 4
output_ids  = [21, 22, 151645]          # 151645 = <|im_end|>, i.e. the stop token
hidden      = 7 rows, row r = [r, r]
finish      = "stop"

→ prefill    rows 0..3  (values 0,1,2,3)
→ decode     rows 4..5  (values 4,5)       # row 6 belongs to the stop token, drop it
→ output_ids [21, 22]
```

target = 4 + 3 - 1 = 6, so only hidden[:6] is used.

Example 2: finished by length + extra rows captured

```
prompt_ids = [11, 12]  (P=2)   output_ids = [5, 6]   hidden has 5 rows (0..4)   finish = "length"
→ prefill rows 0,1    decode rows 2,3    output_ids [5, 6]       # row 4 is beyond target=4, discard
prompt_ids = [1,2,3,4] (P=4)   hidden has only 3 rows  → ValueError (not even the prefill was fully captured)
```

Note: the extra rows aren't necessarily just "the stop row"; always truncate by target.

Example 3: prefill layout and decode consumption (D=2)

```
prompt_embeds = [[1,2],[3,4]]   first_reply = [0.5,0.5]
spk = [10,20]  codec_pad = [1,1]  codec_bos = [100,200]
mask_id=7, pad_id=5, start_id=6

ids    = [7, 7, 5, 6]
embeds = [[1,2], [3,4], [11,21], [100.5,200.5]]      # shape [P+2, D] = [4, 2]

decode: reply_queue = [[1,1],[2,2]], fallback = [-9,-9]
 step | last_codec_embed | output input      | queue rows left
 1    | [0.1,0.2]        | [1.1, 1.2]        | 1
 2    | [0.3,0.4]        | [2.3, 2.4]        | 0
 3    | [0.5,0.6]        | [-8.5, -8.4]      | 0   ← uses fallback
 4    | [0.7,0.8]        | [-8.3, -8.2]      | 0   ← fallback from now on
```

Key questions

1. Why does a request finished by stop have one more hidden row than "what the talker actually needs"? Which token's forward does that row correspond to?
2. When `finish_reason == "length"`, is the last generated token "normal content" or a "stop token"? Should it be dropped?
3. What decides the boundary between `prefill` and `decode`: the number of `hidden` rows, or `len(prompt_ids)`?
4. If `hidden` has > target rows, what do you do with the extra rows? What if it has < P? Why are the two cases handled differently?
5. The talker prefill has P+2 positions. For each of the last two positions, which two vectors are summed into its embedding? What are the corresponding ids?
6. `first_reply` is already consumed in prefill. So which generated token's row should the first row of `reply_queue` be during decode? (This is a caller convention, but you need to think it through.)
7. When the queue is empty, what shape is the "remaining queue" returned by `talker_decode_input`?

© 2026 kzhou003

Details

- **"Append first, then check stop"**: vLLM's scheduling loop is: sample token → append to output → check stop conditions. So the stop token also went through (or was at least recorded as) one decode output, and its hidden row was accumulated along with the rest. If the talker consumes it as a reply vector, the audio gets an extra tail that isn't part of the reply.
- **target row count**: `target = P + len(output_ids) − drop`. It's "the total number of rows the talker should see", not `hidden.shape[0] − drop`; the latter is wrong when extra rows were captured.
- **ThinkerPayload.output_ids** is a new list with the stop id removed; don't modify the caller's list in place (the caller may still use it for detokenizing / returning to the user).
- **mask_id / pad_id / start_id** are special ids in the talker (codec) vocabulary; in real code they come from the model config, here they're passed as keyword arguments. The P prompt positions have no "real token" in the codec vocabulary, so they are all filled with `mask_id`, and the model only looks at the embedding.
- **[D] vs [1, D]**: `spk_embed + codec_pad_embed` is a 1-D `[D]` vector; to concatenate with `[P, D]` it must first become `[1, D]` (`[None]` / `unsqueeze(0)`), or use `torch.stack`.
- **decode input = text side + audio side**: the talker's input at each step is the sum of "the thinker reply vector (text semantics)" and "the embedding of the codec id sampled at the previous step (audio autoregression)". Thinker replies are much shorter than codec frames; once they run out, the text side switches to `fallback` (in the real system, the tts_pad embedding).

Level 2 · Approach & pitfalls

- **Split into three pure, independent functions**: trim only slices and checks bounds; prefill only does cat; decode only does "take head + add". No state, no loops, 3-8 lines each.
- **Order in trim**: (1) compute P and check `hidden.shape[0] < P` → ValueError; (2) compute drop; (3) compute target and truncate rows; (4) then split into prefill / decode at P; (5) trim the tail of output_ids. Put the check first so the error isn't masked by later slicing (out-of-range slicing doesn't raise, it just returns a short tensor).
- **Edge case: stop but output_ids is empty**. Here drop=1 makes target = P-1, cutting off one prompt row. Think about it: with no generated tokens, is there a "stop row" to drop at all? A safe way is to drop only when output_ids is non-empty.
- **Common bug 1**: using `hidden[:-1]` to "remove the stop row". When hidden captured extra rows (Example 2), you remove the extra row rather than the stop row, and the stop row stays in decode.
- **Common bug 2**: decode uses `rows[P:]` without first truncating rows to target; equivalent to bug 1. Symptom: decode has 1 more row than output_ids, and the talker has an extra noisy frame at the end.
- **prefill concatenation**: a single `torch.cat` of three pieces (`[P,D]`, `[1,D]`, `[1,D]`), dim=0. For ids, use list multiplication and then append the two special ids. The return order is `(ids, embeds)`; don't swap them.
- **decode's "remaining queue"**: if there are rows, return `queue[1:]` (shape `[K−1, D]`; `[0, D]` when K=1); if empty, return the empty queue as is. Don't return None; the caller keeps passing it into the next step.

Level 3 · Walkthrough (code: see the commented solutions in Appendix C)

Answers to the key questions

- After the stop token is sampled it's appended before the stop check, so that step's hidden has already been accumulated; this row is recorded "after the forward that produced the stop token" and isn't part of the reply content.
- When finished by length, the last token is normal content (max_tokens was just reached); keep everything, drop=0.
- The boundary is `P = len(prompt_ids)`; the hidden row count is only used for checking and truncation.
- > target: truncate to target, silently discarding; < P: not even the prompt is complete, the talker prefill can't be built, so you must raise. Between P and target: slicing naturally yields a shorter decode (the spec doesn't require an error).
- Second to last: `spk_embed + codec_pad_embed`, id = `pad_id`; last: `first_reply + codec_bos_embed`, id = `start_id`.
- The queue usually starts from the 2nd generated token (the 1st already went into prefill as first_reply); this function doesn't care, it just takes the head of the queue.
- With an empty queue, return the original empty queue, shape `[0, D]`.

Deriving trim (Example 1 step by step)

Quantity	Value
P	4
hidden.shape[0] < P ?	7 < 4 → no, continue
drop	finish=="stop" and output non-empty → 1
target	4 + 3 − 1 = 6
rows = hidden[:target]	rows 0..5
prefill = rows[:P]	rows 0..3
decode = rows[P:]	rows 4,5
output_ids	[21, 22, 151645][: 3−1] = [21, 22]

© 2026 kzhou003

Self-check: run your implementation with the numbers from Examples 1/2/3; in particular check (1) with Example 2's 5 hidden rows, decode is exactly 2 rows; (2) the first P rows of the embeds returned by `build_talker_prefill` are exactly equal to prompt_embeds (nothing added to them); (3) from decode step 3 on, fallback is always used and the queue shape stays `[0, D]`.

A-23 · Codec window planning: small first chunk, large later chunks, left context

Level 1 · I/O examples

Example 1: calling next_window one by one with `initial=2, chunk=5, left_ctx=3`

```
next_window(1,  0,  False, initial=2, chunk=5, left_ctx=3) -> None          # first chunk needs 2 frames, only 1
next_window(2,  0,  False, ...)                            -> (0, 0, 2)     # first chunk
next_window(6,  2,  False, ...)                            -> None          # 5 frames per chunk after that,
only 4 new
next_window(7,  2,  False, ...)                            -> (0, 2, 7)     # ctx_start = max(0, 2-3) = 0
next_window(12, 7,  False, ...)                            -> (4, 7, 12)    # 3 frames of left context: 4,5,6
next_window(20, 12, True,  ...)                            -> (9, 12, 17)   # finished, but a full chunk is
still there → still cut at 5
next_window(20, 17, True,  ...)                            -> (14, 17, 20)  # 3 frames left → flush
next_window(20, 20, True,  ...)                            -> None          # everything sent
next_window(1,  0,  True,  ...)                            -> (0, 0, 1)     # very short stream: can't even fill
the first chunk, flush directly
```

Without a first-chunk setting: `next_window(6, 0, False, initial=0, chunk=6, left_ctx=10) -> (0, 0, 6)`, `next_window(12, 6, False, initial=0, chunk=6, left_ctx=10) -> (0, 6, 12)` (left context clipped at 0).

Example 2: slice_new_samples

```
window = (4, 7, 12), upsample = 2
decoded = vocoder(codes[4:12])        # 8 frames × 2 = 16 samples: s0 .. s15
                                      # s0..s5 belong to context frames 4,5,6; s6..s15 belong to new frames
7..11
slice_new_samples(decoded, (4, 7, 12), 2) -> [s6, s7, s8, s9, s10, s11, s12, s13, s14, s15]

slice_new_samples([0, 1, 2, 3], (0, 0, 2), 2) -> [0, 1, 2, 3]   # with no left context the whole thing is new
```

Example 3: one complete stream (initial=2, chunk=5, left_ctx=3, 14 frames total, arriving in 6 batches)

Batch	Frames in batch	available	finished	Windows from looping this batch	emitted afterwards
0	1	1	F	(none)	0
1	2	3	F	(0, 0, 2)	2
2	4	7	F	(0, 2, 7)	7
3	3	10	F	(none)	7
4	1	11	F	(none)	7
5	3	14	T	(4, 7, 12), (9, 12, 14)	14

On each batch the caller loops on next_window until it returns None, setting `emitted = new_end` after each window. The [new_start, new_end) of all windows exactly tile [0, 14).

Key questions

1. How do you tell "the first window" from the arguments? The function has no internal state; the only information it has is `available / emitted / finished`.
2. What does `initial=0` mean? What size is the first chunk then?
3. When `finished=True` but the remaining frames are ≥ one full window, should you send all remaining frames at once, or still cut by chunk size? (`(20, 12, True)` in Example 1)
4. What do you return when `finished=True` and `available == emitted`? If you returned an empty window like `(…, 20, 20)`, what would happen to the caller's while loop?
5. Why can't `ctx_start` be negative? How many times are the frames between it and `emitted` decoded? How many times are they output?
6. Which frame of the window does the i-th sample of decoded belong to? What is the index in decoded of the first sample of new frame `new_start`?
7. How large must left_ctx be for streaming decode to be **sample-for-sample identical** to decoding the whole thing at once? Which property of the vocoder does this depend on?

Details

© 2026 kzhou003

- **All three coordinates are frame indices in the global coordinate system of the whole stream**: `ctx_start ≤ new_start < new_end ≤ available`. `[ctx_start, new_start)` is the left context (only used to "warm up" the vocoder); `[new_start, new_end)` are the new frames actually sent this step.
- **new_start always equals emitted**: windows must be contiguous end to end, no overlaps, no holes. The caller advances emitted to new_end after each window.
- **Window size**: the first window (nothing sent yet, and `initial > 0`) uses `initial`; all later ones use `chunk`. "full window available" means `available − emitted` is at least that size; then take exactly that many, leaving the extra frames for the next call.
- **flush**: only when a full window can't be filled, `finished` is set, and there are unsent frames, send all the rest as the last (shorter) window.
- **Unit conversion in slice_new_samples**: the vocoder produces `upsample` samples per frame; decoded is in local coordinates starting at `ctx_start`, with length `(new_end − ctx_start) * upsample`. All you need is to convert the frame range `[new_start, new_end)` into a sample range within decoded.
- **Why the first chunk is small**: TTFP (time to first packet) depends on how many frames the first window waits for; after that, larger chunks raise vocoder throughput and reduce call count. The cost of left context is decoding left_ctx extra frames per window.

Level 2 · Approach & pitfalls

- **Split next_window into three steps**: (1) compute this window's `size` (first-chunk test: `emitted == 0 and initial > 0`); (2) decide new_end: full window / flush / None; (3) compute new_start and ctx_start from emitted and left_ctx.
- **The order of checks matters**: first check "is a full window available", then "finished with leftovers". If finished goes first, `(20, 12, True)` in Example 1 returns `(9, 12, 20)`: no frames are lost, but it violates "return a full window when one is available".
- **The flush condition requires** `available > emitted`: otherwise at end of stream you return an empty window `new_start == new_end`, and the caller's `while w := next_window(...)` loops forever (emitted no longer advances).
- **First-chunk test trap**: use `emitted == 0`, not something like `available <= initial`. The latter misjudges the first chunk as chunk-sized when many frames arrive at once (e.g. available=30, emitted=0).
- **Slice formula**: offset = `(new_start − ctx_start) * upsample`, end = `(new_end − ctx_start) * upsample` (i.e. len(decoded)). Common bug: slicing the local decoded with the global index `new_start * upsample`: correct only by coincidence when ctx_start=0; in Example 2 it gives `decoded[14:24]`, only 2 samples.
- **You can add an assertion** `len(decoded) == (new_end − ctx_start) * upsample`, so that a wrong window from the caller is exposed immediately instead of silently producing audio of the wrong length.
- **Left context ≥ receptive field**: if the vocoder is causal, the output for frame i depends only on `codes[i−r .. i]`; so as long as there are at least r frames to the left of the new frames in the window, its input is exactly what it would be in a full decode; with fewer than r frames (and ctx_start not at 0), the first few frames of the window see truncated context and produce a seam.

Level 3 · Walkthrough (code: see the commented solutions in Appendix C)

Plug Example 1's parameters into the formula step by step (fill in the blanks)

available, emitted, finished	size	available - emitted	Branch	new_end	ctx_start	Result
2, 0, F	2 (first chunk)	2	full	2	max(0, 0-3)=0	(0, 0, 2)
6, 2, F	5	4	not enough, not finished	-	-	None
12, 7, F	5	5	full	12	4	(4, 7, 12)
20, 12, T	5	8	full (takes priority over flush)	17	9	(9, 12, 17)
20, 17, T	??	??	??	??	??	(14, 17, 20)
20, 20, T	5	0	nothing left	-	-	None

(Answer for the blank row: size=5, difference 3, flush, new_end=20, ctx_start=14.)

Left context vs receptive field: an example you can compute by hand Toy vocoder: frame i outputs 2 samples `[10·S, 10·S+1]`, where S = sum of `codes[max(0,i−2) .. i]` within the window (receptive field r=2, looking only inside the window). `codes = [3,1,4,1,5,9,2,6]`, `initial=3, chunk=3`, all available at once (finished=True).

© 2026 kzhou003

left_ctx	Window	decoded for this window	New samples sliced out
2	(0,0,3)	30,31,40,41,80,81	30,31,40,41,80,81
2	(1,3,6)	10,11,50,51,60,61,100,101,150,151	60,61,100,101,150,151
2	(4,6,8)	50,51,140,141,160,161,170,171	160,161,170,171
1	(2,3,6)	40,41,50,51,100,101,150,151	**50,51**,100,101,150,151
1	(5,6,8)	90,91,110,111,170,171	**110,111**,170,171

Full decode: `30,31,40,41,80,81,60,61,100,101,150,151,160,161,170,171`. With left_ctx=2 it matches exactly; with left_ctx=1 the first frame of each window (bold) has a seam: frame 3 should see codes[1..3]=1+4+1=6, but only sees codes[2..3]=4+1=5.

Answers to the Level-1 questions

- First chunk ⇔ `emitted == 0` (no frame sent yet) and `initial > 0`.
- `initial=0` means no special first chunk; the first chunk also uses chunk.
- Still cut by chunk; flush only when less than one chunk remains.
- Return None; an empty window leaves emitted where it is and the loop never ends.
- At the start of the stream new_start - left_ctx can be negative, and frame indices can't be. Context frames are decoded more than once (once as new frames of the previous window, again as context of the next), but output only once.
- decoded[i] belongs to local frame `i // upsample`, i.e. global frame `ctx_start + i // upsample`; the first sample of new_start is at `(new_start − ctx_start) * upsample`.
- left_ctx ≥ the vocoder's (causal) receptive field; it relies on the two properties "causal + local".

A-24 · MRoPE 3D positions

Level 1 · I/O examples

Example 1: image at the start + rectangular grid (spatial_merge=2)

```
segments = [("image", (2, 4, 6)), ("text", 2)]
image LLM grid = (t=2, h=4//2=2, w=6//2=3) → 12 tokens
             |---------- t=0 ---------|---------- t=1 ---------| text
t (axis0):   0  0  0  0  0  0          1  1  1  1  1  1         3  4
h (axis1):   0  0  0  1  1  1          0  0  0  1  1  1         3  4
w (axis2):   0  1  2  0  1  2          0  1  2  0  1  2         3  4
max position of the image segment = 2 (from the w axis) → next segment start = 3
T = 14, max = 4, delta = (4+1) − 14 = −9
decode_position(14, −9) = [5, 5, 5]    decode_position(15, −9) = [6, 6, 6]
```

Example 2: an image with a long time dimension (video frames)

```
segments = [("text", 1), ("image", (4, 2, 2)), ("text", 1)]
grid (4, 1, 1) → 4 tokens
t:  0 | 1 2 3 4 | 5
h:  0 | 1 1 1 1 | 5
w:  0 | 1 1 1 1 | 5
T = 6, max = 5, delta = 6 − 6 = 0       # the t axis advanced by 4, exactly the token count
```

delta being 0 doesn't mean "no image"; it only means this image's position span happens to equal its token count.

Example 3: two adjacent images + merge=1

```
[("text",1), ("image",(1,4,4)), ("image",(1,2,4)), ("text",1)], merge=2
t: 0 | 1 1 1 1 | 3 3 | 5
h: 0 | 1 1 2 2 | 3 3 | 5
w: 0 | 1 2 1 2 | 3 4 | 5          T=8, max=5, delta=−2

[("image",(1,2,4))], merge=1  → grid (1,2,4), 8 tokens
t: 0 0 0 0 0 0 0 0
h: 0 0 0 0 1 1 1 1
w: 0 1 2 3 0 1 2 3                T=8, max=3, delta=−4
```

The second image's start = 3: the first image's max position is 2, not "5 tokens so far".

Key questions

1. How many tokens does a `(t, h, w)` image segment take in the LLM sequence? What happens when `h` isn't divisible by `spatial_merge` (the docstring uses `//`)?
2. "t-major, then h, then w": how do you compute the k-th image token's `(t_idx, h_idx, w_idx)` from k?

© 2026 kzhou003

3. What does `start` become after each segment? Why isn't it "the number of tokens so far"?
4. On which axis can an image segment's "max position" occur? Which one is it in Example 1?
5. When is `delta` negative, zero, positive? (Is positive even possible?)
6. Why are the positions of the i-th decode token (i ≥ T) the same on all three axes? What shape and dtype does decode_position return?
7. How should an empty segment like `("text", 0)` be handled so it doesn't affect later starts?

Details

- **Three axes**: rows 0/1/2 of positions are temporal / height / width. MRoPE splits head_dim's rotary frequencies into three parts, each rotated with one of these three position rows (M01). For pure text the three rows are identical, degenerating to ordinary 1-D RoPE.
- **Vision patches vs LLM tokens**: the vision encoder outputs `t×h×w` features by patch, then merges each adjacent `merge×merge` group of patches into one LLM token. So h, w are in patch units, and the LLM grid is `h//merge × w//merge`; the t direction is not merged.
- **Meaning of start**: start is "the position origin of the next segment", equal to **the max over all positions so far (on all three axes) + 1**. An image's position span is `max(t, gh, gw)`, usually much smaller than its token count `t·gh·gw`: this is how MRoPE "compresses" positions.
- **delta**: `delta = (max_pos + 1) − T`. After prefill the sequence continues from absolute index T, but positions should continue from `max_pos + 1`; their difference is delta, kept as a constant for the request's lifetime.
- **dtype**: positions are integer indices; return `torch.long` with shape `[3, T]`. `torch.arange` is int64 by default.
- **decode_position(index, delta)**: returns a long tensor of shape `(3,)`, all three elements equal to `index + delta`. Decode generates only text, so the three axes are identical.

Level 2 · Approach & pitfalls

- **Overall structure**: keep `start = 0` and a `parts` list; per segment build a `[3, n_seg]` block and append it; after each segment update start with the block's max; finally `torch.cat(parts, dim=1)`, then compute delta.
- **Text segment**: `idx = start + arange(n)`, then make 3 rows (`expand(3, n)` / `repeat` / `stack([idx]*3)`). Note `expand` gives a view sharing storage; the later `cat` copies, so it's fine, but clone it if you plan to modify it in place.
- **Image segment**: first compute `gh, gw = h // m, w // m`. The three indices can come from `torch.meshgrid(arange(t), arange(gh), arange(gw), indexing="ij")` followed by `flatten()` on each: `ij` indexing + row-major flatten is naturally t-major, then h, then w. Or broadcast with view/expand.
- **Updating start**: `start = block.max() + 1` (as a Python int). For a text block the max is `start + n − 1`; for an image block max = `start + max(t, gh, gw) − 1`. An empty block (n=0 or a zero grid) has no max; skip the update.
- **Common bug 1**: `start += n_tokens`. Correct for pure text, wrong with images (the second image in Example 3 would start at 6 instead of 3). The model won't raise, it just gets "dumber".
- **Common bug 2**: meshgrid with default/`xy` indexing, or the wrong flatten order, swapping h and w. Invisible on a square grid (2×2), obvious on a rectangular grid (2×3 in Example 1): always self-test with a rectangular grid and t>1.
- **Common bug 3**: delta as `max_pos − T` (missing +1), or computing decode positions from the token count. Check with Example 1: the first decode position should be 5.
- **Empty input**: with empty segments `torch.cat([])` raises; returning `torch.zeros(3, 0, dtype=torch.long)` and delta=0 is more robust.

Level 3 · Walkthrough (code: see the commented solutions in Appendix C)

Answers to the key questions

- `t · (h//m) · (w//m)` tokens; when not divisible it rounds down (in real models h, w are always multiples of merge).
- Let `hw = gh·gw`: `t_idx = k // hw`, `h_idx = (k % hw) // gw`, `w_idx = k % gw`.
- `start = max of all positions generated so far + 1`. An image's position span is max(t, gh, gw), not its token count; using the token count makes subsequent text positions "jump too far".
- Depends on which of t, gh, gw is largest; in Example 1 gw=3 is largest, so the max position 2 comes from the w axis.
- Negative when there is an image with `t·gh·gw > max(t,gh,gw)`; zero with only text, or when the image degenerates into a line (Example 2: gh=gw=1); never positive: each token contributes at most as much position span as tokens.
- Decode produces only text tokens, and text has identical axes; returns `(3,)` long.
- Skip it; don't update start (no need to add it to parts either).

Segment-by-segment trace (first input of Example 3)

Segment	start (in)	Grid/length	Block (t / h / w)	Block max	start (out)
text 1	0	1	0 / 0 / 0	0	1
image (1,4,4)	1	1×2×2	1111 / 1122 / 1212	2	3
image (1,2,4)	3	1×1×2	33 / 33 / 34	4	5
text 1	5	1	5 / 5 / 5	5	6

© 2026 kzhou003

T = 1+4+2+1 = 8, max = 5, delta = 6 − 8 = −2; the next decode token (index 8) has position 6.
Self-check: use Example 1 (rectangular + multiple frames) to check h/w aren't swapped; use Example 3 to check the second image starts at 3; check `pos.dtype == torch.long`, `pos.shape == (3, T)`; for any input, `decode_position(T, delta)` should equal `pos.max() + 1`.

A-25 · Startup planning: device groups, parallel init, rollback, admission

Level 1 · I/O examples

Example 1: init_groups (with transitivity)

```
{0: {2}, 1: {0}, 2: {0, 1}, 3: {1}, 4: {5}}
  stage1 and stage2 share dev0; stage2 and stage3 share dev1 → 1,2,3 form one group
→ [[0], [1, 2, 3], [4]]

{3: {7}, 1: {7}, 0: {9}, 2: {9, 4}}          # dict key order is shuffled
→ [[0, 2], [1, 3]]                            # ascending within a group, groups sorted by first element

{0: {0}, 1: {1}, 2: {2}, 3: {2, 0}}           # stage3 "bridges" 0 and 2
→ [[0, 2, 3], [1]]
```

Example 2: run_init short-circuits on failure (`max_workers=1`, to make the order deterministic)

```
stage_devices = {0: {0}, 1: {0}, 2: {1}}     → groups [[0, 1], [2]]
init_fn: stage1 raises RuntimeError("boom"), the others return "e<sid>"

Timeline:
  group[0,1]: init(0) → "e0" ✓ ; init(1) → raises, recorded as the first failure
  group[2]  : sees an existing failure before starting → doesn't call init(2)
Afterwards: cleanup_fn(0, "e0"); raise InitError(stage_id=1)
stages init_fn was called for: [0, 1]      cleaned up: [(0, "e0")]

If stage2 fails instead: calls [0, 1, 2], cleanup (0,"e0"),(1,"e1"), InitError.stage_id = 2
All succeed: {0: {0}, 1: {0}, 2: {1}}, init_fn = lambda s: s*10 → {0: 0, 1: 10, 2: 20}
```

Example 3: admission_check (GiB = 2^{30}, two 80 GiB cards)

```
reserve = 2 GiB
  {0: ({0}, 0.9), 1: ({1}, 0.6), 2: ({1}, 0.3)}
  dev0: 72+2 = 74 ≤ 80 ✓    dev1: (48+2)+(24+2) = 76 ≤ 80 ✓        → []
  change stage2 to 0.35: dev1 = 50 + 30 = 80 ≤ 80 ✓ (exactly equal is allowed)    → []
  then change reserve to 4 GiB: dev1 = 52 + 32 = 84 > 80                  → 1 entry, containing "device 1"

cross-card stage (TP=2): reserve = 3 GiB
  {0: ({0, 1}, 0.5), 1: ({1}, 0.45)}
  dev0: 40+3 = 43 ✓     dev1: (40+3)+(36+3) = 82 > 80              → 1 entry, containing "device 1"
  add a dev2 with no stage on it → no error produced
```

Key questions

1. Why must "shared devices" be grouped by **transitive closure**? Give an example where looking only at pairwise relations goes wrong.
2. Why must stages within a group be serial? What's the risk of running groups in parallel with each other (hint: memory measurement)?
3. When the first failure happens, what about an `init_fn` in another group that is **currently running**? Can you forcibly interrupt the thread?
4. When should cleanup be called: right at the moment of failure, or after all threads have finished? Why?
5. If several groups fail almost simultaneously, which stage does `InitError` report? What guarantees "the first" is recorded only once?
6. In `admission_check`, a stage placed on two cards `{0, 1}`: which card's total is its utilization computed against? How many times is it counted?
7. Compare with `>` or `≥`? Should a card with no stage on it produce output?

Details

- **Canonical group form**: each group is a list of stage ids in **ascending** order; groups are sorted by their **first (smallest) stage**. This makes the output independent of dict insertion order and set iteration order.
- **"no NEW init_fn calls start"**: short-circuiting only blocks inits that **haven't started yet**; an init_fn already running is left to return naturally (Python threads can't be killed safely). If it succeeds, it also counts as "succeeded" and must be cleaned up too.

© 2026 kzhou003

- **cleanup_fn(stage, result)**: the second argument is that stage's `init_fn` return value (e.g. an engine handle), needed during rollback to release resources.
- **InitError(failed_stage, cause)**: the constructor is given; `stage_id` is the failed stage. `raise ... from cause` is recommended to keep the original traceback.
- **Admission formula**: for each card, `Σ_{stages on this card} (utilization × this card's total + reserve)` ≤ total. In vLLM, `gpu_memory_utilization` means "the maximum fraction of this card this process may use"; a cross-card stage takes that fraction on each of its cards.
- **Error strings**: one per over-subscribed card, and it must contain the substring `device <id>`; the rest (which stages, how many bytes needed) is up to you, but the more specific the easier to debug.

Level 2 · Approach & pitfalls

- **init_groups = connected components**. Two approaches: (1) union-find: iterate over each device of each stage, record "who first occupied this device", and union when you see it again; (2) build a bipartite graph (stage-device) and run BFS/DFS. The size is tiny, either works; the key is transitivity.
- **Structure of run_init**: call `init_groups` first; each group is one task submitted to `ThreadPoolExecutor(max_workers)`; inside the task, a for loop calls init_fn in stage id order. Exiting the `with` block joins all tasks, which is exactly "wait for the running ones to finish".
- **Shared state**: `results: dict`, `failure` (the first failure's (sid, exc)). Both protected by one `threading.Lock`. Each group checks failure **before calling each init_fn**, and returns if set.
- **Don't leave exceptions in the Future**: if init_fn's exception propagates straight out of the task, `ex.map` / `future.result()` re-raises it on the main thread, but you lose "which stage" and the control of "cleanup first, then raise". try/except inside the task and write (sid, e) into the shared state.
- **Put cleanup after the thread pool shuts down**: by then all threads are done and `results` no longer changes, so a successful stage that finished late is also collected and reclaimed. Cleaning up at the moment of failure misses stages that complete afterwards.
- **Common bugs**: (1) parallelizing within a group too (submitting each stage separately) → same-card stages initialize concurrently; (2) after a failure only checking whether "this group" failed, so other groups keep starting new stages; (3) cleaning up the failed stage too (it has no result); (4) catching only `Exception`: think about `KeyboardInterrupt` / `SystemExit` inside init.
- **admission_check**: the outer loop iterates `device_total_bytes` (sorted, for stable output), the inner loop finds all stages with `dev in devices`, sums `int(util * total) + reserve`, and only reports when `need > total`.

Level 3 · Walkthrough (code: see the commented solutions in Appendix C)

Answers to the key questions

- {0:{0,1}, 1:{1,2}, 2:{2}}: stage0 and stage2 don't share a card directly, but both share with stage1; if 0 and 2 run in parallel while 1 is in a different group from them, you get concurrent init on the same card. So the whole connected component must be serial.
- Each stage's init does "measure free memory → decide the KV block count". If another stage on the same card is loading weights/profiling, the measurement is skewed: overestimating KV means OOM, underestimating wastes memory. Different cards don't affect each other, so parallelism across groups shortens startup.
- Don't interrupt; let it finish. If it succeeds it goes into results and is cleaned up later with the rest.
- After the thread pool has fully joined. See level2.
- Report the first recorded failure; `if not failure: failure.append(...)` under the lock guarantees it's recorded only once.
- Against the total of each card it sits on, counted once per card (each with its own reserve).
- Report only when `need > total` (equal is allowed); an empty card has need=0, no output.

Union-find derivation (first input of Example 1): iterate stages in ascending order, `owner[device]` records the first occupant

stage	devices	Action	parent afterwards
0	{2}	owner[2]=0	0→0 1→1 2→2 3→3 4→4
1	{0}	owner[0]=1	unchanged
2	{0,1}	dev0 already owned by 1 → union(2,1); owner[1]=2	2→1
3	{1}	dev1 already owned by 2 → union(3, find(2)=1)	3→1
4	{5}	owner[5]=4	unchanged

Collect by root: {0:[0], 1:[1,2,3], 4:[4]} → sort by first element → `[[0],[1,2,3],[4]]`. Making the smaller id the root on union makes the root exactly the group's minimum (not required).

Going further: when `max_workers` is less than the number of groups, later groups queue up; when their turn comes and a failure already exists, they return immediately. That's why stage2 is never called in Example 2. If max_workers is large enough,

© 2026 kzhou003

group[2] may already be running, in which case it runs to completion and gets cleaned up. Both behaviors satisfy the spec, so unit tests should control timing with Events rather than rely on sleep.

A-26 · CUDA graph bucket runner (CPU simulation)

Level 1 · I/O examples

Example 1: state changes over a sequence of calls

```
fn = lambda x: x * 2 + 1
r = GraphRunner(fn, buckets=[16, 1, 4], hidden=3)   # self.buckets = [1, 4, 16]
r.capture()                                         # graphs keys: {1, 4, 16}
static_in / static_out shape: [16, 3]
each input x = a [bs, 3] filled entirely with float(bs)
```

Call	bucket	replays	eager	last_bucket	static_in[:6, 0]	static_out[:6, 0]
run(bs=10)	16	1	0	16	10 10 10 10 10 10	21 21 21 21 21 21
run(bs=3)	4	2	0	4	3 3 3 **0** 10 10	7 7 7 **1** 21 21
run(bs=17)	none → eager	2	1	4	unchanged	unchanged
run(bs=4)	4	3	1	4	4 4 4 4 10 10	9 9 9 9 21 21
run(bs=1, force_eager)	eager	3	2	4	unchanged	unchanged
run(bs=1)	1	4	2	1	1 4 4 4 10 10	3 9 9 9 21 21

Observe: at bs=3, row 3 is zeroed (it held the 10 left over from the previous bs=10), and its output is fn(0)=1; rows 4 and 5 are outside the bucket and nobody touches them.

Example 2: why you must clone

```
y1 = r.run(x1)        # if you return static_out[:bs] directly (a view)
y2 = r.run(x2)        # replay writes fn(x2) into static_out
y1                    # → now equals fn(x2)! y1 was "silently overwritten"
```

With a clone returned, y1 stays fn(x1).

Key questions

1. `CpuGraph.replay()` takes no arguments. So how does new input "get into" the graph? And where do you read the output from?
2. Why are the graphs for all buckets built on the first b rows of the **same** static buffer, instead of each bucket allocating its own?
3. With bs=3 and bucket=4, what might be in row 3 (by index)? What happens if it isn't zeroed (to this batch's outputs, to NaNs, to MoE routing/statistics)?
4. Zeroing all of `static_in[bs:]` vs only `static_in[bs:b]` : which is necessary? What's the cost of zeroing more?
5. The typical `force_eager` case is a request with its own `torch.Generator` . Why does replay "silently ignore" it instead of raising?
6. Should `last_bucket` be updated on the eager path?
7. If `run` finds a bucket that hasn't been captured, can it capture one on the spot? Why does the spec say never?

Details

- **Static buffers**: a CUDA graph records kernels and the **device addresses** they read and write. On replay the addresses don't change, so new data must be `copy_` 'd into the same memory; you can't reassign `self.static_in` to a new tensor (the graph would still read the old memory).
- **view**: `static_in[:b]` is a view sharing storage, not a copy. The b=4 graph reads and writes the first 4 rows of the big buffer; all graphs share the same memory, so data from a previous large batch lingers in later rows.
- **"largest bucket first"**: on CUDA multiple graphs share one memory pool; capturing the largest graph first allocates the largest intermediate tensors, and smaller graphs can reuse those blocks, saving memory. In the CPU simulation the order doesn't affect results, but follow the spec.
- **Bucket selection**: the smallest `b ≥ bs` . `buckets` is already sorted in __init__, so a linear search is fine (or use `bisect`).
- **stats**: `replays` and `eager` are counters; `last_bucket` records the bucket used, only on replay.
- **Return shape**: return only the first bs rows (the padding rows' results are garbage from "computing on 0" and must not be given to the caller).

Level 2 · Approach & pitfalls

© 2026 kzhou003

- **__init__**: `max_bs = buckets[-1]`; allocate `static_in`, `static_out` with shape `[max_bs, hidden]` and the given `dtype`. Initializing to 0 is safer than `empty` (fn runs once during capture). Note: delete the `raise NotImplementedError` at the end of the stub, and keep the existing fields.
- **capture**: iterate `sorted(buckets, reverse=True)`; for each b build a `CpuGraph` with `static_in[:b]` / `static_out[:b]`, call `.capture()`, store it in `self.graphs[b]`.
- **Decision order in run**: (1) compute bs; (2) force_eager or no bucket found (or that bucket isn't in graphs) → eager branch: count +1, `return self.fn(x)`; (3) otherwise take the replay branch. The eager branch doesn't touch the static buffers.
- **Four steps of the replay branch**: copy input → zero [bs, b) → replay → update stats → return a clone of the first bs rows. Order-wise, zeroing must happen before replay; the clone must happen before any next replay (i.e. before returning).
- **Pitfall: assignment instead of copy**. `self.static_in[:bs] = x` is actually an in-place write (equivalent to copy_), which is fine; but `self.static_in = x` or swapping tensors inside the graph is wrong: the graph can't see it.
- **Pitfall: forgetting to zero**. With this exercise's fn (row-wise), dirty rows only affect padding-row outputs, so "the results look right"; but real models have cross-row interactions (the batch dimension in attention, MoE load balancing, BN-like statistics, NaN/Inf propagation), and dirty rows contaminate real rows. So zero explicitly, in a way that's observable on `static_in`.
- **Pitfall: returning a view**. A single call won't catch it; after two consecutive calls the first result is overwritten (Example 2).

Level 3 · Walkthrough (code: see the commented solutions in Appendix C)

Answers to the key questions

- Input: first `copy_` into static_in; replay reads static_in and writes static_out; read the output from static_out.
- Sharing one buffer saves memory, and run needs only one "write location": whichever bucket is chosen, it writes the first bs rows of static_in.
- Possibly real data from a previous larger batch (the 10 in Example 1). In this exercise fn is row-independent, so not zeroing just makes the padding outputs garbage; in a real model it can spread NaNs, affect MoE routing and expert load statistics, and affect ops that reduce over the whole batch.
- Only [bs, b) needs zeroing: that's the range this graph reads; rows beyond b aren't read by this graph at all, so zeroing them just wastes bandwidth.
- replay just replays the kernels recorded at capture time and doesn't re-execute Python; capture used the global RNG state/a fixed philox offset, so the per-row generator's Python logic is never executed. Hence no error, just results that aren't reproducible / don't honor the seed.
- No; its meaning is "the bucket used by the most recent replay".
- No: capture is slow (needs warmup, synchronization), and on CUDA it conflicts with running streams/the memory pool; capturing at runtime causes latency spikes. All graphs must be recorded at startup.

Step-by-step trace of run (2nd call of Example 1, bs=3)

Step	Operation	static_in[:5,0]	static_out[:5,0]
0	before the call (previous bs=10)	10 10 10 10 10	21 21 21 21 21
1	b = smallest ≥3 → 4	same	same
2	static_in[:3] ← x	3 3 3 10 10	same
3	static_in[3:4] ← 0	3 3 3 0 10	same
4	graphs[4].replay()	same	7 7 7 1 21
5	stats: replays=2, last_bucket=4		
6	return static_out[:3].clone()	a new [3,3] tensor of all 7s	

Self-check list: (1) after two consecutive runs, the first return value is unchanged; (2) run a large batch then a small one, and check rows [bs, b) of `static_in` are all 0; (3) after capture, the `id()` of every object in `self.graphs` is unchanged across runs; (4) when bs > max bucket or force_eager, the static buffers are not modified at all.

© 2026 kzhou003

Appendix B · Full test code

B-0 · tests/_tiny.py (tiny HF builders shared by Track M)

```
"""Tiny random Qwen3-Omni configs/modules from HF transformers, used as the golden reference."""
import pytest
import torch

transformers = pytest.importorskip("transformers")
C = pytest.importorskip("transformers.models.qwen3_omni_moe.configuration_qwen3_omni_moe")
M = pytest.importorskip("transformers.models.qwen3_omni_moe.modeling_qwen3_omni_moe")

HIDDEN, HEADS, KV_HEADS, HEAD_DIM = 64, 4, 2, 16
MROPE_SECTION = [4, 2, 2]  # sums to HEAD_DIM // 2, same interleave pattern as the real [24, 20, 20]


def thinker_text_config(**kw):
    cfg = C.Qwen3OmniMoeTextConfig(
        vocab_size=97, hidden_size=HIDDEN, intermediate_size=96, num_hidden_layers=2,
        num_attention_heads=HEADS, num_key_value_heads=KV_HEADS, head_dim=HEAD_DIM,
        moe_intermediate_size=32, num_experts=6, num_experts_per_tok=2, norm_topk_prob=True,
        rope_theta=10000.0, max_position_embeddings=256, rms_norm_eps=1e-6,
        rope_scaling={"rope_type": "default", "mrope_section": MROPE_SECTION, "interleaved": True},
        **kw,
    )
    cfg._attn_implementation = "eager"
    return cfg


def talker_text_config():
    cfg = C.Qwen3OmniMoeTalkerTextConfig(
        vocab_size=50, hidden_size=HIDDEN, intermediate_size=96, num_hidden_layers=1,
        num_attention_heads=HEADS, num_key_value_heads=KV_HEADS, moe_intermediate_size=32,
        num_experts=6, num_experts_per_tok=2, norm_topk_prob=True, shared_expert_intermediate_size=48,
    )
    cfg._attn_implementation = "eager"
    return cfg


def code_predictor_config(num_code_groups=6, vocab=40):
    cfg = C.Qwen3OmniMoeTalkerCodePredictorConfig(
        vocab_size=vocab, hidden_size=HIDDEN, intermediate_size=96, num_hidden_layers=2,
        num_attention_heads=HEADS, num_key_value_heads=KV_HEADS, head_dim=HEAD_DIM,
        num_code_groups=num_code_groups, max_position_embeddings=64, rms_norm_eps=1e-6,
    )
    cfg._attn_implementation = "eager"
    return cfg


def randomize_(module, seed=0, std=0.2):
    """HF inits RMSNorm weights to 1 and some linears to ~0; randomize so bugs are visible."""
    g = torch.Generator().manual_seed(seed)
    with torch.no_grad():
        for p in module.parameters():
            p.copy_(torch.randn(p.shape, generator=g) * std + (1.0 if p.ndim == 1 else 0.0))
    return module.eval()


def causal_mask(t, past=0):
    m = torch.full((t, past + t), float("-inf"))
    m = torch.triu(m, diagonal=past + 1)
    return m[None, None]
```


© 2026 kzhou003

B-1 · tests/test_m01_rmsnorm_mrope.py

```
import torch

from drill import load
from tests._tiny import HEAD_DIM, HIDDEN, MROPE_SECTION, M, thinker_text_config

m = load("m01_rmsnorm_mrope")


def mm_positions():
    # text(3) | image grid t=1,h=2,w=3 starting at 3 | text(2)
    t = [0, 1, 2, 3, 3, 3, 3, 3, 3, 5, 6]
    h = [0, 1, 2, 3, 3, 3, 4, 4, 4, 5, 6]
    w = [0, 1, 2, 3, 4, 5, 3, 4, 5, 5, 6]
    return torch.tensor([t, h, w])


def test_rms_norm_matches_hf_fp32_and_bf16():
    norm = M.Qwen3OmniMoeThinkerTextRMSNorm(HIDDEN, eps=1e-6)
    with torch.no_grad():
        norm.weight.copy_(torch.randn(HIDDEN))
    x = torch.randn(3, 7, HIDDEN) * 5
    torch.testing.assert_close(m.rms_norm(x, norm.weight, 1e-6), norm(x))
    xb = x.bfloat16()
    torch.testing.assert_close(m.rms_norm(xb, norm.weight.bfloat16(), 1e-6), norm.bfloat16()(xb))


def test_inv_freq():
    cfg = thinker_text_config()
    rot = M.Qwen3OmniMoeThinkerTextRotaryEmbedding(cfg)
    torch.testing.assert_close(m.rope_inv_freq(HEAD_DIM, cfg.rope_theta), rot.inv_freq)


def test_interleaved_mrope_matches_hf_qwen3_omni():
    cfg = thinker_text_config()
    rot = M.Qwen3OmniMoeThinkerTextRotaryEmbedding(cfg)
    pos = mm_positions()
    x = torch.zeros(1, pos.shape[1], HIDDEN)
    cos_hf, sin_hf = rot(x, pos[:, None, :])  # [1, T, D]
    cos, sin = m.mrope_cos_sin(rot.inv_freq, pos, MROPE_SECTION, interleaved=True)
    torch.testing.assert_close(cos, cos_hf[0])
    torch.testing.assert_close(sin, sin_hf[0])


def test_text_only_positions_reduce_to_plain_rope():
    inv = m.rope_inv_freq(HEAD_DIM, 10000.0)
    pos = torch.arange(9).expand(3, 9)
    cos_i, _ = m.mrope_cos_sin(inv, pos, MROPE_SECTION, interleaved=True)
    cos_c, _ = m.mrope_cos_sin(inv, pos, MROPE_SECTION, interleaved=False)
    freqs = torch.arange(9).float()[:, None] * inv[None]
    plain = torch.cat([freqs, freqs], -1).cos()
    torch.testing.assert_close(cos_i, plain)
    torch.testing.assert_close(cos_c, plain)


def test_apply_rotary_matches_hf_qwen3_omni():
    cfg = thinker_text_config()
    rot = M.Qwen3OmniMoeThinkerTextRotaryEmbedding(cfg)
    pos = mm_positions()
    T = pos.shape[1]
    q, k = torch.randn(T, 4, HEAD_DIM), torch.randn(T, 2, HEAD_DIM)
    cos_hf, sin_hf = rot(torch.zeros(1, T, HIDDEN), pos[:, None, :])
    q_hf, k_hf = M.apply_rotary_pos_emb(q.transpose(0, 1)[None], k.transpose(0, 1)[None], cos_hf, sin_hf)
    cos, sin = m.mrope_cos_sin(rot.inv_freq, pos, MROPE_SECTION, interleaved=True)
    torch.testing.assert_close(m.apply_rotary(q, cos, sin), q_hf[0].transpose(0, 1))
    torch.testing.assert_close(m.apply_rotary(k, cos, sin), k_hf[0].transpose(0, 1))


def test_chunked_mrope_matches_hf_qwen2_5_omni():
    from transformers.models.qwen2_5_omni import modeling_qwen2_5_omni as Q25

    pos = mm_positions()
    T = pos.shape[1]
    inv = m.rope_inv_freq(HEAD_DIM, 10000.0)
    freqs = pos.float()[:, :, None] * inv  # [3, T, half] -> full per-axis cos/sin like HF
```


© 2026 kzhou003

```
    emb = torch.cat([freqs, freqs], -1)[:, None]  # [3, 1, T, D]
    q = torch.randn(1, 4, T, HEAD_DIM)
    q_hf, _ = Q25.apply_multimodal_rotary_pos_emb(q, q, emb.cos(), emb.sin(), MROPE_SECTION)
    cos, sin = m.mrope_cos_sin(inv, pos, MROPE_SECTION, interleaved=False)
    torch.testing.assert_close(m.apply_rotary(q[0].transpose(0, 1), cos, sin), q_hf[0].transpose(0, 1))
```

B-2 · tests/test_m02_thinker_attention.py

```
import torch

from drill import load
from tests._tiny import HIDDEN, M, causal_mask, randomize_, thinker_text_config

m = load("m02_thinker_attention")


def setup(T=10, seed=0):
    cfg = thinker_text_config()
    att = randomize_(M.Qwen3OmniMoeThinkerTextAttention(cfg, layer_idx=0), seed)
    rot = M.Qwen3OmniMoeThinkerTextRotaryEmbedding(cfg)
    torch.manual_seed(seed)
    x = torch.randn(T, HIDDEN)
    pos = torch.arange(T).expand(3, T).clone()
    pos[1, 3:7] += torch.tensor([0, 0, 1, 1])  # a tiny 2x2 "image" grid in the middle
    pos[2, 3:7] += torch.tensor([0, 1, 0, 1])
    cos, sin = rot(x[None], pos[:, None, :])
    with torch.no_grad():
        ref, _ = att(x[None], (cos, sin), causal_mask(T))
    return att, x, cos[0], sin[0], ref[0]


def test_prefill_matches_hf():
    att, x, cos, sin, ref = setup()
    out = m.attention_forward(m.from_hf(att), x, cos, sin)
    torch.testing.assert_close(out, ref, atol=1e-5, rtol=1e-4)


def test_decode_one_token_at_a_time_with_cache():
    att, x, cos, sin, ref = setup(T=12, seed=1)
    w, cache = m.from_hf(att), m.KVCache()
    out = m.attention_forward(w, x[:5], cos[:5], sin[:5], cache)
    rows = [out]
    for t in range(5, 12):
        rows.append(m.attention_forward(w, x[t:t + 1], cos[t:t + 1], sin[t:t + 1], cache))
    torch.testing.assert_close(torch.cat(rows), ref, atol=1e-5, rtol=1e-4)
    assert len(cache) == 12


def test_chunked_prefill_with_cache():
    att, x, cos, sin, ref = setup(T=11, seed=2)
    w, cache = m.from_hf(att), m.KVCache()
    a = m.attention_forward(w, x[:4], cos[:4], sin[:4], cache)
    b = m.attention_forward(w, x[4:], cos[4:], sin[4:], cache)  # a chunk of 7 after 4 cached tokens
    torch.testing.assert_close(torch.cat([a, b]), ref, atol=1e-5, rtol=1e-4)
```


© 2026 kzhou003

B-3 · tests/test_m03_sparse_moe.py

```
import torch

from drill import load
from tests._tiny import HIDDEN, M, randomize_, talker_text_config, thinker_text_config

m = load("m03_sparse_moe")


def run_hf(block, x):
    with torch.no_grad():
        y, logits = block(x[None])
    return y[0], logits


def test_routing_matches_hf_softmax_topk_renorm():
    block = randomize_(M.Qwen3OmniMoeThinkerTextSparseMoeBlock(thinker_text_config()), seed=0, std=0.5)
    x = torch.randn(17, HIDDEN)
    w = m.from_hf(block)
    topw, topi = m.route(x, w)
    probs = torch.softmax(x @ w.router.T, -1)
    ref_w, ref_i = torch.topk(probs, 2, -1)
    ref_w = ref_w / ref_w.sum(-1, keepdim=True)
    assert torch.equal(topi, ref_i)
    torch.testing.assert_close(topw, ref_w)
    torch.testing.assert_close(topw.sum(-1), torch.ones(17))


def test_thinker_moe_matches_hf():
    block = randomize_(M.Qwen3OmniMoeThinkerTextSparseMoeBlock(thinker_text_config()), seed=1)
    x = torch.randn(23, HIDDEN)
    ref, _ = run_hf(block, x)
    torch.testing.assert_close(m.moe_forward(x, m.from_hf(block)), ref, atol=1e-5, rtol=1e-4)


def test_talker_moe_with_shared_expert_matches_hf():
    block = randomize_(M.Qwen3OmniMoeTalkerTextSparseMoeBlock(talker_text_config()), seed=2)
    x = torch.randn(9, HIDDEN)
    ref, _ = run_hf(block, x)
    torch.testing.assert_close(m.moe_forward(x, m.from_hf(block)), ref, atol=1e-5, rtol=1e-4)


def test_no_renorm_variant():
    cfg = thinker_text_config()
    cfg.norm_topk_prob = False
    block = randomize_(M.Qwen3OmniMoeThinkerTextSparseMoeBlock(cfg), seed=3)
    x = torch.randn(8, HIDDEN)
    ref, _ = run_hf(block, x)
    torch.testing.assert_close(m.moe_forward(x, m.from_hf(block)), ref, atol=1e-5, rtol=1e-4)


def test_single_token_and_batch_invariance():
    block = randomize_(M.Qwen3OmniMoeThinkerTextSparseMoeBlock(thinker_text_config()), seed=4)
    w = m.from_hf(block)
    x = torch.randn(6, HIDDEN)
    full = m.moe_forward(x, w)
    for i in range(6):
        torch.testing.assert_close(m.moe_forward(x[i:i + 1], w)[0], full[i], atol=1e-5, rtol=1e-4)
```


© 2026 kzhou003

B-4 · tests/test_m04_mm_merge_deepstack.py

```
import pytest
import torch

from drill import load
from tests._tiny import HIDDEN, M, causal_mask, randomize_, thinker_text_config

m = load("m04_mm_merge_deepstack")
IMG, AUD = 90, 91


def test_audio_token_count_matches_hf():
    frames = torch.tensor([1, 2, 7, 99, 100, 101, 199, 200, 250, 1234, 3000])
    torch.testing.assert_close(m.audio_token_count(frames), M._get_feat_extract_output_lengths(frames))
    assert m.audio_token_count(100) == 13


def test_merge_places_embeddings_in_order():
    ids = torch.tensor([1, IMG, IMG, 2, AUD, AUD, AUD, 3, IMG])
    text = torch.randn(9, HIDDEN)
    img, aud = torch.randn(3, HIDDEN), torch.randn(3, HIDDEN)
    text_before = text.clone()
    out, masks = m.merge_multimodal(ids, text, {"image": IMG, "audio": AUD}, {"image": img, "audio": aud})
    torch.testing.assert_close(out[[1, 2, 8]], img)
    torch.testing.assert_close(out[[4, 5, 6]], aud)
    torch.testing.assert_close(out[[0, 3, 7]], text[[0, 3, 7]])
    assert masks["image"].tolist() == [False, True, True, False, False, False, False, False, True]
    assert torch.equal(text, text_before), "do not mutate the text embeddings in place"


def test_merge_count_mismatch_raises():
    ids = torch.tensor([IMG, IMG, 5])
    with pytest.raises(ValueError):
        m.merge_multimodal(ids, torch.randn(3, HIDDEN), {"image": IMG}, {"image": torch.randn(3, HIDDEN)})
    with pytest.raises(ValueError):
        m.merge_multimodal(ids, torch.randn(3, HIDDEN), {"image": IMG}, {})


def test_deepstack_matches_hf_text_model():
    cfg = thinker_text_config()
    model = randomize_(M.Qwen3OmniMoeThinkerTextModel(cfg), seed=0)
    T = 10
    emb = torch.randn(1, T, HIDDEN)
    vis = torch.zeros(T, dtype=torch.bool)
    vis[3:7] = True
    deep = [torch.randn(4, HIDDEN), torch.randn(4, HIDDEN)]
    pos = torch.arange(T).expand(3, 1, T)
    with torch.no_grad():
        ref = model(inputs_embeds=emb, position_ids=pos, visual_pos_masks=vis[None, :, None].expand(1, T, HIDDEN),
                    deepstack_visual_embeds=deep).last_hidden_state[0]
        cos_sin = model.rotary_emb(emb, pos)
        mask = causal_mask(T)
        layers = [lambda h, L=L: L(h[None], position_embeddings=cos_sin, attention_mask=mask)[0] for L in
model.layers]
        out = model.norm(m.run_layers(layers, emb[0].clone(), vis, deep))
    torch.testing.assert_close(out, ref, atol=1e-5, rtol=1e-4)


def test_deepstack_only_touches_visual_rows():
    h = torch.randn(6, HIDDEN)
    vis = torch.tensor([0, 1, 0, 1, 0, 0], dtype=torch.bool)
    v = torch.randn(2, HIDDEN)
    out = m.apply_deepstack(h, vis, v)
    torch.testing.assert_close(out[vis], h[vis] + v)
    torch.testing.assert_close(out[~vis], h[~vis])
```


© 2026 kzhou003

B-5 · tests/test_m05_talker_mtp.py

```
import torch

from drill import load
from tests._tiny import HIDDEN, M, code_predictor_config, randomize_

m = load("m05_talker_mtp")


def setup(G=6, B=3, seed=0):
    cp = randomize_(M.Qwen3OmniMoeTalkerCodePredictorModelForConditionalGeneration(code_predictor_config(G)), seed,
0.3)
    g = torch.Generator().manual_seed(seed + 100)
    hid, cb0 = torch.randn(B, HIDDEN, generator=g), torch.randn(B, HIDDEN, generator=g)
    return cp, hid, cb0


def hf_greedy(cp, hid, cb0):
    with torch.no_grad():
        res = cp.generate(inputs_embeds=torch.stack([hid, cb0], 1), max_new_tokens=cp.config.num_code_groups - 1,
                          do_sample=False, output_hidden_states=True, return_dict_in_generate=True)
    return res


def test_reprefill_matches_hf_generate():
    cp, hid, cb0 = setup()
    ref = hf_greedy(cp, hid, cb0).sequences
    codes = m.predict_residual_codes(cp, hid, cb0)
    assert codes.shape == (3, 5)
    assert torch.equal(codes, ref)


def test_kv_cache_version_matches_reprefill():
    for seed in range(3):
        cp, hid, cb0 = setup(G=7, B=2, seed=seed)
        assert torch.equal(m.predict_residual_codes_kv(cp, hid, cb0), m.predict_residual_codes(cp, hid, cb0))


def test_rows_are_independent():
    cp, hid, cb0 = setup(B=4, seed=5)
    full = m.predict_residual_codes(cp, hid, cb0)
    for i in range(4):
        assert torch.equal(m.predict_residual_codes(cp, hid[i:i + 1], cb0[i:i + 1])[0], full[i])


def test_next_input_matches_hf_summation():
    cp, hid, cb0 = setup(seed=7)
    res = hf_greedy(cp, hid, cb0)
    text_step = torch.randn(3, HIDDEN)
    # HF Qwen3OmniMoeTalkerForConditionalGeneration.prepare_inputs_for_generation
    mid = [h[0] for h in res.hidden_states[1:]]                 # embeddings of residual codes 1..G-2
    last = cp.get_input_embeddings()[-1](res.sequences[..., -1:])  # embedding of residual code G-1
    with torch.no_grad():
        ref = torch.cat([cb0[:, None]] + mid + [last], dim=1).sum(1) + text_step
        got = m.next_talker_input(cp, cb0, res.sequences, text_step)
    torch.testing.assert_close(got, ref, atol=1e-5, rtol=1e-4)
```


© 2026 kzhou003

B-6 · tests/test_m06_streaming_causal_conv.py

```
import random

import torch

from drill import load
from tests._tiny import M, randomize_

m = load("m06_streaming_causal_conv")


def random_chunks(total, seed):
    rnd = random.Random(seed)
    sizes, left = [], total
    while left:
        sizes.append(min(left, rnd.randint(1, 6)))
        left -= sizes[-1]
    return sizes


def test_causal_conv_streaming_equals_offline():
    for seed, (k, d) in enumerate([(3, 1), (5, 2), (7, 1), (1, 1)]):
        conv = randomize_(M.Qwen3OmniMoeCausalConvNet(4, 6, kernel_size=k, dilation=d), seed)
        x = torch.randn(2, 4, 37)
        with torch.no_grad():
            ref = conv(x)
        s = m.StreamingCausalConv1d.from_hf(conv)
        outs, pos = [], 0
        for n in random_chunks(37, seed):
            y = s.step(x[..., pos:pos + n])
            assert y.shape[-1] == n, "a causal stride-1 conv emits exactly one output per input"
            outs.append(y)
            pos += n
        torch.testing.assert_close(torch.cat(outs, -1), ref, atol=1e-5, rtol=1e-4)


def test_trans_conv_streaming_equals_offline():
    for seed, (k, st) in enumerate([(4, 2), (10, 5), (6, 3), (3, 3)]):
        conv = randomize_(M.Qwen3OmniMoeCausalTransConvNet(4, 5, kernel_size=k, stride=st), seed)
        x = torch.randn(2, 4, 23)
        with torch.no_grad():
            ref = conv(x)
        s = m.StreamingCausalTransConv1d.from_hf(conv)
        outs, pos = [], 0
        for n in random_chunks(23, seed + 10):
            outs.append(s.step(x[..., pos:pos + n]))
            pos += n
        outs.append(s.flush())
        torch.testing.assert_close(torch.cat(outs, -1), ref, atol=1e-5, rtol=1e-4)


def test_trans_conv_emits_as_early_as_possible():
    k, st = 8, 4
    conv = randomize_(M.Qwen3OmniMoeCausalTransConvNet(3, 3, kernel_size=k, stride=st), 0)
    s = m.StreamingCausalTransConv1d.from_hf(conv)
    emitted = 0
    for n in range(1, 8):
        emitted += s.step(torch.randn(1, 3, 1)).shape[-1]
        # positions < n*stride are final; the first (k - stride) are trimmed
        assert emitted == max(0, n * st - (k - st)), (n, emitted)
```


© 2026 kzhou003

B-7 · tests/test_m07_stage_weight_loading.py

```
import pytest
import torch
import torch.nn.functional as F

from drill import load
from tests._tiny import HIDDEN, M, randomize_, thinker_text_config

m = load("m07_stage_weight_loading")


def checkpoint():
    cfg = thinker_text_config()
    model = randomize_(M.Qwen3OmniMoeThinkerTextModel(cfg), seed=0)
    state = {f"thinker.model.{k}": v.detach().clone() for k, v in model.state_dict().items()}
    state["thinker.lm_head.weight"] = torch.randn(cfg.vocab_size, HIDDEN)
    state["talker.model.layers.0.self_attn.q_proj.weight"] = torch.randn(8, 8)
    state["code2wav.decoder.0.conv.weight"] = torch.randn(4, 4, 3)
    return cfg, model, state


def test_shard_planning():
    wm = {"thinker.model.a": "s1", "thinker.model.b": "s2", "talker.x": "s2", "code2wav.y": "s3", "thinker.lm_head":
"s1"}
    assert m.shards_for_stage(wm, "thinker") == ["s1", "s2"]
    assert m.shards_for_stage(wm, "code2wav") == ["s3"]


def test_select_stage_strips_prefix_and_drops_other_stages():
    _, model, state = checkpoint()
    sd = m.select_stage(state, "thinker", "thinker.model.")
    assert set(sd) == set(model.state_dict())


def test_packed_tensors_reproduce_hf_projections():
    cfg, model, state = checkpoint()
    globals_, layers = m.load_thinker_text(state, cfg.num_hidden_layers)
    assert torch.equal(globals_["embed_tokens"], model.embed_tokens.weight)
    x = torch.randn(5, HIDDEN)
    D, H, KVH = cfg.head_dim, cfg.num_attention_heads, cfg.num_key_value_heads
    for i, layer in enumerate(layers):
        hf = model.layers[i]
        q, k, v = (x @ layer.qkv.T).split([H * D, KVH * D, KVH * D], dim=-1)
        with torch.no_grad():
            torch.testing.assert_close(q, hf.self_attn.q_proj(x))
            torch.testing.assert_close(k, hf.self_attn.k_proj(x))
            torch.testing.assert_close(v, hf.self_attn.v_proj(x))
            for e, ex in enumerate(hf.mlp.experts):
                g, u = (x @ layer.w13[e].T).chunk(2, dim=-1)
                torch.testing.assert_close((F.silu(g) * u) @ layer.w2[e].T, ex(x))
        assert torch.equal(layer.router, hf.mlp.gate.weight)


def test_missing_and_unexpected_keys_are_errors():
    cfg, _, state = checkpoint()
    broken = dict(state)
    del broken["thinker.model.layers.1.mlp.experts.3.up_proj.weight"]
    with pytest.raises(m.LoadError):
        m.load_thinker_text(broken, cfg.num_hidden_layers)
    extra = dict(state)
    extra["thinker.model.layers.0.self_attn.rotary.inv_freq"] = torch.zeros(4)
    with pytest.raises(m.LoadError):
        m.load_thinker_text(extra, cfg.num_hidden_layers)
```


© 2026 kzhou003

B-8 · tests/test_m08_tiny_thinker_e2e.py

```
import torch

from drill import load
from tests._tiny import HIDDEN, MROPE_SECTION, M, randomize_, thinker_text_config

m = load("m08_tiny_thinker_e2e")


def build(seed=0):
    cfg = thinker_text_config()
    model = randomize_(M.Qwen3OmniMoeThinkerTextModel(cfg), seed=seed, std=0.3)
    lm_head = torch.randn(cfg.vocab_size, HIDDEN, generator=torch.Generator().manual_seed(seed)) * 0.5
    state = {f"thinker.model.{k}": v.detach().clone() for k, v in model.state_dict().items()}
    state["thinker.lm_head.weight"] = lm_head
    dims = m.Dims(cfg.num_hidden_layers, cfg.num_attention_heads, cfg.num_key_value_heads, cfg.head_dim,
                  cfg.num_experts_per_tok, cfg.norm_topk_prob, cfg.rope_theta, MROPE_SECTION, cfg.rms_norm_eps)
    return model, lm_head, state, dims


@torch.no_grad()
def hf_logits(model, lm_head, ids):
    return model(input_ids=torch.tensor([ids])).last_hidden_state[0] @ lm_head.T


@torch.no_grad()
def hf_greedy(model, lm_head, prompt, n):
    ids = list(prompt)
    for _ in range(n):
        ids.append(int(hf_logits(model, lm_head, ids)[-1].argmax()))
    return ids[len(prompt):]


def test_prefill_logits_match_hf():
    model, lm_head, state, dims = build()
    tt = m.TinyThinker(state, dims)
    ids = [3, 14, 15, 92, 65, 35, 8]
    caches = [m.KVCache() for _ in range(dims.num_layers)]
    with torch.no_grad():
        got = tt.forward(torch.tensor(ids), 0, caches)
    torch.testing.assert_close(got, hf_logits(model, lm_head, ids), atol=1e-4, rtol=1e-4)


def test_greedy_with_kv_cache_matches_hf_full_recompute():
    for seed in range(2):
        model, lm_head, state, dims = build(seed)
        tt = m.TinyThinker(state, dims)
        prompt = [1, 2, 3, 4, 5, 6][: 3 + seed * 3]
        assert tt.greedy(prompt, 12) == hf_greedy(model, lm_head, prompt, 12)
```


© 2026 kzhou003

B-9 · tests/test_p02_orchestrator.py

```
import asyncio
import random

from drill import load

m = load("p02_orchestrator")
StageOutput = m.StageOutput


class FakeStage:
    """Emits len(fn(rid, inputs)) outputs asynchronously after each submit."""

    def __init__(self, fn, max_delay=0.002, emit_first_in_submit=False):
        self.fn = fn
        self.max_delay = max_delay
        self.emit_first_in_submit = emit_first_in_submit
        self.q: asyncio.Queue = asyncio.Queue()
        self.submitted: list = []
        self.aborted: list = []

    async def submit(self, rid, inputs):
        self.submitted.append((rid, inputs))
        chunks = self.fn(rid, inputs)
        start = 0
        if self.emit_first_in_submit:
            await self.q.put([StageOutput(rid, chunks[0], len(chunks) == 1)])
            start = 1
            await asyncio.sleep(0.01)  # let readers + dispatcher run before we return
        asyncio.get_running_loop().create_task(self._emit(rid, chunks, start))

    async def _emit(self, rid, chunks, start):
        for i in range(start, len(chunks)):
            await asyncio.sleep(random.random() * self.max_delay)
            await self.q.put([StageOutput(rid, chunks[i], i == len(chunks) - 1)])

    async def get_output(self):
        return await self.q.get()

    async def abort(self, request_ids):
        self.aborted += request_ids


def thinker(rid, prompt):
    return [f"{rid}:t{i}" for i in range(5)]


def talker(rid, inputs):
    up = [x.upper() for x in inputs]
    return [up[:3], up[3:]]


def vocoder(rid, inputs):
    flat = [x for part in inputs for x in part]
    return [f"wav({x})" for x in flat]


def forward(next_stage, prompt, prev_data):
    return list(prev_data)


def make(final={0, 2}, **kw):
    stages = [FakeStage(thinker, **kw), FakeStage(talker), FakeStage(vocoder)]
    return stages, m.Orchestrator(stages, final, forward)


async def collect(orch, n_requests, timeout=5.0):
    per = {}
    done = 0
    while done < n_requests:
        o = await orch.next_output(timeout=timeout)
        per.setdefault(o.request_id, []).append(o)
        if o.finished:
            done += 1
    return per
```


© 2026 kzhou003

```
def check_request(rid, outs, final={0, 2}):
    assert sum(o.finished for o in outs) == 1, "exactly one terminal message"
    assert outs[-1].finished, "terminal message must be the last one"
    assert {o.stage_id for o in outs} == final, "only final stages reach the client"
    if 0 in final:
        assert [o.data for o in outs if o.stage_id == 0] == thinker(rid, None)
    expected_wav = [f"wav({x.upper()})" for x in thinker(rid, None)]
    assert [o.data for o in outs if o.stage_id == 2] == expected_wav


def test_single_request_three_stages():
    async def main():
        stages, orch = make()
        await orch.start()
        await orch.add_request("r0", "hello")
        per = await collect(orch, 1)
        check_request("r0", per["r0"])
        assert orch.request_states == {}
        await orch.shutdown()

    asyncio.run(main())


def test_many_concurrent_requests():
    async def main():
        stages, orch = make()
        await orch.start()
        rids = [f"r{i}" for i in range(30)]
        for rid in rids:
            await orch.add_request(rid, "p")
        per = await collect(orch, len(rids))
        for rid in rids:
            check_request(rid, per[rid])
        # each stage got each request exactly once
        for s in stages:
            assert sorted(r for r, _ in s.submitted) == sorted(rids)
        await orch.shutdown()

    asyncio.run(main())


def test_only_last_stage_is_final():
    async def main():
        stages, orch = make(final={2})
        await orch.start()
        await orch.add_request("a", "p")
        per = await collect(orch, 1)
        check_request("a", per["a"], final={2})
        await orch.shutdown()

    asyncio.run(main())


def test_state_registered_before_submit():
    async def main():
        stages, orch = make(emit_first_in_submit=True)
        await orch.start()
        await orch.add_request("early", "p")
        per = await collect(orch, 1)
        check_request("early", per["early"])
        assert orch.dropped_outputs == 0
        await orch.shutdown()

    asyncio.run(main())


def test_unknown_request_output_is_dropped():
    async def main():
        stages, orch = make()
        await orch.start()
        await stages[0].q.put([StageOutput("ghost", "x", True)])
        await orch.add_request("r", "p")
        per = await collect(orch, 1)
        check_request("r", per["r"])
        assert orch.dropped_outputs == 1
        await orch.shutdown()
```


© 2026 kzhou003

```
    asyncio.run(main())
```


© 2026 kzhou003

B-10 · tests/test_p03_async_chunk_orchestrator.py

```
import asyncio
import random

from drill import load

m = load("p03_async_chunk_orchestrator")
StageOutput = m.StageOutput


class Channel:
    """Side channel between two fake stages (stands in for the SHM connector)."""

    def __init__(self):
        self.qs: dict[str, asyncio.Queue] = {}

    def q(self, rid):
        return self.qs.setdefault(rid, asyncio.Queue())


class ChunkStage:
    """Stage that reads chunks from `inp` (or the prompt for stage 0), transforms
    each one, forwards it on `out`, and reports outputs to the orchestrator.

    late_finish: report finished=True to the orchestrator only after `late_finish` s
    (simulates a non-final stage whose terminal report arrives after the final stage).
    """

    def __init__(self, transform, inp=None, out=None, late_finish=0.0, n_tokens=6):
        self.transform, self.inp, self.out = transform, inp, out
        self.late_finish, self.n_tokens = late_finish, n_tokens
        self.q: asyncio.Queue = asyncio.Queue()
        self.submitted: list = []

    async def submit(self, rid, inputs):
        self.submitted.append((rid, inputs))
        asyncio.get_running_loop().create_task(self._run(rid))

    async def _run(self, rid):
        if self.inp is None:
            items = [(f"{rid}:t{i}", i == self.n_tokens - 1) for i in range(self.n_tokens)]
        else:
            items = None
        i = 0
        while True:
            if items is not None:
                if i == len(items):
                    break
                data, last = items[i]
                i += 1
            else:
                data, last = await self.inp.q(rid).get()
            await asyncio.sleep(random.random() * 0.002)
            y = self.transform(data)
            if self.out is not None:
                await self.out.q(rid).put((y, last))
            if last and self.late_finish:
                await self.q.put([StageOutput(rid, y, False)])
                await asyncio.sleep(self.late_finish)
                await self.q.put([StageOutput(rid, None, True)])
            else:
                await self.q.put([StageOutput(rid, y, last)])
            if last:
                break

    async def get_output(self):
        return await self.q.get()

    async def abort(self, ids):
        pass


def build(late_finish=0.0):
    c01, c12 = Channel(), Channel()
    stages = [
```


© 2026 kzhou003

```
        ChunkStage(lambda x: x, out=c01),
        ChunkStage(lambda x: x.upper(), inp=c01, out=c12, late_finish=late_finish),
        ChunkStage(lambda x: f"wav({x})", inp=c12),
    ]
    orch = m.AsyncChunkOrchestrator(stages, {0, 2}, prewarm_fn=lambda sid, p: ("placeholder", sid))
    return stages, orch


async def collect(orch, n, timeout=5.0):
    per, done = {}, 0
    while done < n:
        o = await orch.next_output(timeout=timeout)
        per.setdefault(o.request_id, []).append(o)
        done += o.finished
    return per


def test_prewarm_and_no_forwarding():
    async def main():
        stages, orch = build()
        await orch.start()
        await orch.add_request("r", "p")
        per = await collect(orch, 1)
        outs = per["r"]
        assert [o.data for o in outs if o.stage_id == 0] == [f"r:t{i}" for i in range(6)]
        assert [o.data for o in outs if o.stage_id == 2] == [f"wav(R:T{i})" for i in range(6)]
        assert outs[-1].finished and sum(o.finished for o in outs) == 1
        # every stage submitted exactly once; downstream got the placeholder
        assert stages[0].submitted == [("r", "p")]
        assert stages[1].submitted == [("r", ("placeholder", 1))]
        assert stages[2].submitted == [("r", ("placeholder", 2))]
        await orch.shutdown()

    asyncio.run(main())


def test_terminal_waits_for_non_final_stage():
    async def main():
        stages, orch = build(late_finish=0.2)
        await orch.start()
        loop = asyncio.get_running_loop()
        t0 = loop.time()
        await orch.add_request("r", "p")
        per = await collect(orch, 1)
        elapsed = loop.time() - t0
        assert elapsed >= 0.18, "terminal was emitted before stage 1 reported finished"
        outs = per["r"]
        assert outs[-1].finished and outs[-1].stage_id == 2
        assert sum(o.finished for o in outs) == 1
        assert all(o.stage_id in (0, 2) for o in outs)
        assert orch.request_states == {}
        await orch.shutdown()

    asyncio.run(main())


def test_many_requests_with_late_finish():
    async def main():
        stages, orch = build(late_finish=0.01)
        await orch.start()
        rids = [f"r{i}" for i in range(20)]
        for rid in rids:
            await orch.add_request(rid, "p")
        per = await collect(orch, len(rids))
        for rid in rids:
            outs = per[rid]
            assert outs[-1].finished and sum(o.finished for o in outs) == 1
            assert [o.data for o in outs if o.stage_id == 2] == [f"wav({rid.upper()}:T{i})" for i in range(6)]
        await orch.shutdown()

    asyncio.run(main())


def test_late_output_after_cleanup_is_dropped():
    async def main():
        stages, orch = build()
        await orch.start()
        await orch.add_request("r", "p")
```


© 2026 kzhou003

```
        await collect(orch, 1)
        await stages[1].q.put([StageOutput("r", "zombie", True)])
        await asyncio.sleep(0.05)
        assert orch.dropped_outputs == 1
        assert "r" not in orch.request_states
        await orch.shutdown()

    asyncio.run(main())
```


© 2026 kzhou003

B-11 · tests/test_p01_output_sink.py

```
import asyncio
import random
import threading
import time

from drill import load

m = load("p01_output_sink")


def test_basic_fifo():
    async def main():
        s = m.OutputSink()
        s.bind_loop()
        s.put(1)
        s.put(2)
        assert await s.get_batch() == [1, 2]

    asyncio.run(main())


def test_wakeups_are_coalesced():
    async def main():
        s = m.OutputSink()
        s.bind_loop()
        t = threading.Thread(target=lambda: [s.put(i) for i in range(1000)])
        t.start()
        t.join()
        # nobody consumed yet -> exactly one wakeup was scheduled for 1000 puts
        assert s.wakeups_scheduled == 1
        got = []
        while len(got) < 1000:
            batch = await s.get_batch(max_items=256)
            assert 1 <= len(batch) <= 256
            got += batch
        assert got == list(range(1000))

    asyncio.run(main())


def test_timeout_returns_empty():
    async def main():
        s = m.OutputSink()
        s.bind_loop()
        t0 = time.monotonic()
        assert await s.get_batch(timeout=0.05) == []
        assert time.monotonic() - t0 < 1.0

    asyncio.run(main())


def test_multi_producer_per_producer_order():
    async def main():
        s = m.OutputSink()
        s.bind_loop()
        n, producers = 5000, 4

        def produce(pid):
            for i in range(n):
                s.put((pid, i))
                if i % 97 == 0:
                    time.sleep(random.random() * 0.001)

        threads = [threading.Thread(target=produce, args=(p,)) for p in range(producers)]
        for t in threads:
            t.start()
        got = []
        while len(got) < n * producers:
            batch = await s.get_batch(timeout=2.0)
            assert batch, "consumer starved -> lost wakeup?"
            got += batch
        for t in threads:
            t.join()
        for p in range(producers):
```


© 2026 kzhou003

```
            assert [i for q, i in got if q == p] == list(range(n))

    asyncio.run(main())


def test_ping_pong_never_loses_a_wakeup():
    async def main():
        s = m.OutputSink()
        s.bind_loop()
        go = threading.Event()

        def produce():
            for i in range(300):
                go.wait()
                go.clear()
                time.sleep(random.random() * 0.0005)
                s.put(i)

        t = threading.Thread(target=produce)
        t.start()
        for i in range(300):
            go.set()
            assert await s.get_batch(timeout=2.0) == [i]
        t.join()

    asyncio.run(main())


def test_put_between_empty_drain_and_rearm_is_not_lost():
    async def main():
        s = m.OutputSink()
        s.bind_loop()
        s.put("warm")  # leaves a wakeup "in flight"
        assert await s.get_batch() == ["warm"]
        await asyncio.sleep(0.01)  # let the stale wakeup callback run
        injected = []

        def racing_put():
            if not injected:
                injected.append(True)
                t = threading.Thread(target=s.put, args=("raced",))
                t.start()
                t.join()

        s.after_empty_drain = racing_put
        t0 = time.monotonic()
        assert await s.get_batch(timeout=0.5) == ["raced"]
        assert time.monotonic() - t0 < 0.2, "the item waited for the timeout: lost wakeup"

    asyncio.run(main())
```


© 2026 kzhou003

B-12 · tests/test_p04_stage_pool_abort.py

```
import asyncio

import pytest

from drill import load

m = load("p04_stage_pool_abort")


class Rep:
    def __init__(self, name, log, fail=False):
        self.name, self.log, self.fail = name, log, fail

    async def abort(self, ids):
        self.log.append(("abort", self.name, sorted(ids)))
        if self.fail:
            raise RuntimeError("engine refused abort")


class Proc:
    def __init__(self, log):
        self.log = log
        self.state = {"x": "partial-text"}

    def collect_partial(self, rid):
        self.log.append(("collect", rid))
        return self.state.get(rid)

    def commit_abort(self, rid):
        self.log.append(("commit", rid))
        self.state.pop(rid, None)


def test_sticky_round_robin():
    pool = m.StagePool(0, [Rep("a", []), Rep("b", []), Rep("c", [])])
    picks = [pool.pick(f"r{i}") for i in range(6)]
    assert picks == [0, 1, 2, 0, 1, 2]
    assert all(pool.pick(f"r{i}") == picks[i] for i in range(6)), "binding must be sticky"
    pool.release("r0")
    assert pool.bound("r0") is None


def test_evict_and_no_live():
    pool = m.StagePool(1, [Rep("a", []), Rep("b", [])])
    for i in range(4):
        pool.pick(f"r{i}")
    affected = pool.evict(0)
    assert sorted(affected) == ["r0", "r2"]
    assert pool.live_replicas() == [1]
    assert all(pool.pick(f"n{i}") == 1 for i in range(3))
    assert pool.pick("r1") == 1
    pool.evict(1)
    with pytest.raises(m.NoLiveReplica):
        pool.pick("z")


def test_abort_groups_by_replica():
    log = []
    pool = m.StagePool(0, [Rep("a", log), Rep("b", log)])
    for i in range(4):
        pool.pick(f"r{i}")

    asyncio.run(pool.abort(["r0", "r1", "r2", "r3", "unknown"]))
    assert sorted(log) == [("abort", "a", ["r0", "r2"]), ("abort", "b", ["r1", "r3"])]
    assert all(pool.bound(f"r{i}") is None for i in range(4))


def test_abort_two_phase_order():
    log = []
    pools = [m.StagePool(0, [Rep("s0", log)]), m.StagePool(1, [Rep("s1", log)])]
    pools[0].pick("x")
    pools[1].pick("x")
    proc = Proc(log)
    out = asyncio.run(m.abort_request(pools, proc, "x"))
```


© 2026 kzhou003

```
    assert log == [("collect", "x"), ("abort", "s0", ["x"]), ("abort", "s1", ["x"]), ("commit", "x")]
    assert out.finished and out.error == "abort" and out.data == "partial-text"


def test_abort_failure_does_not_commit():
    log = []
    pools = [m.StagePool(0, [Rep("s0", log, fail=True)])]
    pools[0].pick("x")
    proc = Proc(log)
    with pytest.raises(RuntimeError):
        asyncio.run(m.abort_request(pools, proc, "x"))
    assert ("commit", "x") not in log
    assert proc.state == {"x": "partial-text"}
    assert pools[0].bound("x") == 0, "binding kept so abort can be retried"


def test_dead_replica_only_fails_bound_requests():
    pool = m.StagePool(2, [Rep("a", []), Rep("b", [])])
    for i in range(4):
        pool.pick(f"r{i}")
    errs = m.handle_dead_replica(pool, 1)
    assert sorted(e.request_id for e in errs) == ["r1", "r3"]
    assert all(e.finished and e.error for e in errs)
    assert pool.bound("r0") == 0 and pool.bound("r2") == 0
```


© 2026 kzhou003

B-13 · tests/test_p05_rpc_router.py

```
import queue
import random
import threading
import time

import pytest

from drill import load

m = load("p05_rpc_router")


def test_concurrent_calls_out_of_order():
    router = m.RpcRouter()
    wire: queue.Queue = queue.Queue()

    def server():
        pending = []
        while True:
            item = wire.get()
            if item is None:
                break
            pending.append(item)
            if len(pending) >= 5 or wire.empty():
                random.shuffle(pending)
                for key in pending:
                    router.deliver(key, ("ok", key))
                pending.clear()

    srv = threading.Thread(target=server)
    srv.start()
    results = {}

    def client(i):
        key = ("rpc", i)
        results[i] = router.call(key, lambda: wire.put(key), timeout=2.0)

    threads = [threading.Thread(target=client, args=(i,)) for i in range(50)]
    for t in threads:
        t.start()
    for t in threads:
        t.join()
    wire.put(None)
    srv.join()
    assert results == {i: ("ok", ("rpc", i)) for i in range(50)}


def test_result_delivered_inside_send_is_not_lost():
    router = m.RpcRouter()
    assert router.call("k", lambda: router.deliver("k", 42), timeout=0.5) == 42


def test_timeout_and_late_result():
    router = m.RpcRouter()
    with pytest.raises(TimeoutError):
        router.call("slow", lambda: None, timeout=0.05)
    assert router.deliver("slow", "too late") is False
    assert router.late_results == 1
    # the key can be reused after the timed-out call cleaned up
    assert router.call("slow", lambda: router.deliver("slow", 1), timeout=0.5) == 1


def test_duplicate_key_rejected():
    router = m.RpcRouter()
    w = router.register("dup")
    with pytest.raises(ValueError):
        router.register("dup")
    router.unregister("dup", w)


def test_fail_all_wakes_waiters_and_latches():
    router = m.RpcRouter()
    errors = []
```


© 2026 kzhou003

```
    def client(i):
        try:
            router.call(i, lambda: None, timeout=5.0)
        except Exception as e:  # noqa: BLE001
            errors.append(e)

    threads = [threading.Thread(target=client, args=(i,)) for i in range(5)]
    for t in threads:
        t.start()
    time.sleep(0.1)
    t0 = time.monotonic()
    router.fail_all(RuntimeError("orchestrator died"))
    for t in threads:
        t.join()
    assert time.monotonic() - t0 < 1.0
    assert len(errors) == 5 and all("orchestrator died" in str(e) for e in errors)
    with pytest.raises(RuntimeError):
        router.call("new", lambda: None, timeout=5.0)
```


© 2026 kzhou003

B-14 · tests/test_p10_engine_core_loop.py

```
import threading
import time

from drill import load

m = load("p10_engine_core_loop")
Msg = m.Msg


class FakeEngine:
    def __init__(self, steps_per_req=3, fail_on_step=None):
        self.reqs: dict[str, int] = {}
        self.calls: list = []
        self.steps = 0
        self.has_work_calls = 0
        self.steps_per_req = steps_per_req
        self.fail_on_step = fail_on_step

    def add(self, rid, data):
        self.calls.append(("add", rid))
        self.reqs[rid] = self.steps_per_req

    def abort(self, ids):
        self.calls.append(("abort", tuple(ids)))
        for r in ids:
            self.reqs.pop(r, None)

    def has_work(self):
        self.has_work_calls += 1
        return bool(self.reqs)

    def step(self):
        self.steps += 1
        if self.fail_on_step == self.steps:
            raise RuntimeError("CUDA error: illegal memory access")
        outs = []
        for r in list(self.reqs):
            self.reqs[r] -= 1
            outs.append((r, self.reqs[r] == 0))
            if self.reqs[r] == 0:
                del self.reqs[r]
        return outs

    def utility(self, name, args):
        if name == "boom":
            raise ValueError("bad utility")
        return (name, args)


def start(engine):
    emitted = []
    loop = m.EngineCoreLoop(engine, emitted.append)
    t = threading.Thread(target=loop.run_busy_loop, daemon=True)
    return loop, t, emitted


def wait_for(pred, timeout=2.0):
    end = time.monotonic() + timeout
    while time.monotonic() < end:
        if pred():
            return True
        time.sleep(0.005)
    return False


def test_idle_loop_blocks_instead_of_spinning():
    eng = FakeEngine()
    loop, t, _ = start(eng)
    t.start()
    time.sleep(0.3)
    assert eng.steps == 0
    assert eng.has_work_calls < 10, f"busy-spinning: has_work called {eng.has_work_calls} times"
    loop.request_shutdown()
    t.join(1.0)
```


© 2026 kzhou003

```
    assert not t.is_alive()


def test_add_runs_to_completion():
    eng = FakeEngine(steps_per_req=3)
    loop, t, emitted = start(eng)
    t.start()
    loop.input_queue.put(Msg("ADD", ("r1", None)))
    assert wait_for(lambda: ("output", ("r1", True)) in emitted)
    assert [e for e in emitted if e[0] == "output"] == [("output", ("r1", False))] * 2 + [("output", ("r1", True))]
    loop.request_shutdown()
    t.join(1.0)
    assert emitted[-1] == ("shutdown",)


def test_add_then_abort_are_applied_in_fifo_order_before_stepping():
    eng = FakeEngine()
    loop, t, emitted = start(eng)
    loop.input_queue.put(Msg("ADD", ("r1", None)))
    loop.input_queue.put(Msg("ABORT", ["r1"]))
    loop.input_queue.put(Msg("ADD", ("r2", None)))
    t.start()
    assert wait_for(lambda: ("output", ("r2", True)) in emitted)
    assert eng.calls[:3] == [("add", "r1"), ("abort", ("r1",)), ("add", "r2")]
    assert not any(e[0] == "output" and e[1][0] == "r1" for e in emitted)
    loop.request_shutdown()
    t.join(1.0)


def test_shutdown_from_another_thread_while_idle_is_fast():
    eng = FakeEngine()
    loop, t, emitted = start(eng)
    t.start()
    time.sleep(0.05)
    assert t.is_alive()
    t0 = time.monotonic()
    threading.Thread(target=loop.request_shutdown).start()
    t.join(1.0)
    assert not t.is_alive() and time.monotonic() - t0 < 0.5
    assert emitted == [("shutdown",)]


def test_step_exception_is_fatal():
    eng = FakeEngine(steps_per_req=10, fail_on_step=2)
    loop, t, emitted = start(eng)
    t.start()
    loop.input_queue.put(Msg("ADD", ("r1", None)))
    t.join(2.0)
    assert not t.is_alive()
    assert emitted[-1][0] == "fatal" and "illegal memory access" in emitted[-1][1]


def test_utility_results_and_errors():
    eng = FakeEngine()
    loop, t, emitted = start(eng)
    t.start()
    loop.input_queue.put(Msg("UTILITY", (1, "sleep", (2,))))
    loop.input_queue.put(Msg("UTILITY", (2, "boom", ())))
    assert wait_for(lambda: len([e for e in emitted if e[0] == "utility"]) == 2)
    utils = [e for e in emitted if e[0] == "utility"]
    assert utils[0] == ("utility", 1, ("sleep", (2,)), None)
    assert utils[1][1] == 2 and utils[1][2] is None and "bad utility" in utils[1][3]
    assert t.is_alive(), "a failing utility call must not kill the engine"
    loop.request_shutdown()
    t.join(1.0)
```


© 2026 kzhou003

B-15 · tests/test_p11_chunk_aware_scheduler.py

```
import pytest

from drill import load

m = load("p11_chunk_aware_scheduler")


class Source:
    def __init__(self):
        self.ready: set[str] = set()
        self.loads: list[str] = []

    def load_async(self, rid):
        self.loads.append(rid)

    def is_ready(self, rid):
        return rid in self.ready


def make():
    src = Source()
    return m.ChunkAwareScheduler(m.BaseScheduler(), src), src


def test_not_ready_is_never_scheduled_and_loaded_once():
    s, src = make()
    s.add(m.Req("plain", 0))
    s.add(m.Req("c1", 1, needs_chunk=True))
    for _ in range(3):
        assert "c1" not in s.schedule()
    assert src.loads == ["c1"]
    assert "c1" in s.parked and s.parked["c1"][0].status == "WAITING_FOR_CHUNK"


def test_ready_request_is_scheduled_on_next_call():
    s, src = make()
    s.add(m.Req("c1", 0, needs_chunk=True))
    assert s.schedule() == []
    src.ready.add("c1")
    assert s.schedule() == ["c1"]
    assert not s.parked


def test_running_request_waiting_for_next_chunk():
    s, src = make()
    src.ready.add("c1")
    s.add(m.Req("c1", 0, needs_chunk=True))
    assert s.schedule() == ["c1"]
    src.ready.discard("c1")  # consumed chunk k, chunk k+1 not here yet
    assert s.schedule() == []
    src.ready.add("c1")
    assert s.schedule() == ["c1"]
    assert s.base.running[0].status == "RUNNING"


class Boom(Exception):
    pass


def test_restore_happens_even_if_upstream_raises():
    src = Source()

    class Exploding(m.BaseScheduler):
        def schedule(self):
            src.ready.add("c1")  # the chunk lands while upstream is scheduling...
            raise Boom("upstream scheduler blew up")  # ...and upstream fails

    s = m.ChunkAwareScheduler(Exploding(), src)
    s.add(m.Req("c1", 0, needs_chunk=True))
    with pytest.raises(Boom):
        s.schedule()
    assert src.loads == ["c1"], "c1 must have been parked before upstream ran"
    assert "c1" not in s.parked, "the finally-restore must run even when upstream raises"
    assert [r.req_id for r in s.base.waiting] == ["c1"]
```


© 2026 kzhou003

```
def test_abort_while_parked():
    s, src = make()
    s.add(m.Req("c1", 0, needs_chunk=True))
    s.schedule()
    s.abort("c1")
    src.ready.add("c1")
    assert s.schedule() == []
    assert "c1" not in s.parked


def test_restored_in_arrival_order():
    s, src = make()
    for i, rid in enumerate(["a", "b", "c"]):
        s.add(m.Req(rid, i, needs_chunk=True))
    s.schedule()
    src.ready.update({"c", "b", "a"})
    assert s.schedule() == ["a", "b", "c"]
```


© 2026 kzhou003

B-16 · tests/test_p07_scheduler.py

```
from drill import load

m = load("p07_scheduler")
EOS = -1


def tok(seq_id, k):
    """Deterministic 'model': k-th output token of a sequence."""
    return 1000 * int(seq_id[1:]) + k


def run_step(s):
    sched = s.schedule()
    by_id = {q.seq_id: q for q in s.running}
    s.update(lambda sid: tok(sid, len(by_id[sid].output)), EOS)
    return sched


def test_token_budget_and_chunked_prefill():
    s = m.Scheduler(num_blocks=100, block_size=4, max_num_seqs=8, max_num_batched_tokens=16)
    for i in range(3):
        s.add(m.Seq(f"s{i}", prompt_len=10, max_tokens=5))
    assert run_step(s) == {"s0": 10, "s1": 6}
    assert run_step(s) == {"s0": 1, "s1": 4, "s2": 10}
    assert run_step(s) == {"s0": 1, "s1": 1, "s2": 1}


def test_max_num_seqs():
    s = m.Scheduler(num_blocks=100, block_size=4, max_num_seqs=2, max_num_batched_tokens=100)
    for i in range(4):
        s.add(m.Seq(f"s{i}", prompt_len=3, max_tokens=2))
    assert set(run_step(s)) == {"s0", "s1"}
    assert set(run_step(s)) == {"s0", "s1"}  # s0/s1 finish here (2 tokens)
    assert set(run_step(s)) == {"s2", "s3"}


def test_fcfs_head_of_line():
    s = m.Scheduler(num_blocks=4, block_size=4, max_num_seqs=8, max_num_batched_tokens=100)
    s.add(m.Seq("s0", prompt_len=12, max_tokens=50))  # 3 blocks
    s.add(m.Seq("s1", prompt_len=8, max_tokens=1))    # 2 blocks -> does not fit
    s.add(m.Seq("s2", prompt_len=2, max_tokens=1))    # would fit, but FCFS forbids skipping
    assert run_step(s) == {"s0": 12}


def test_preemption_eventually_finishes_everything_correctly():
    s = m.Scheduler(num_blocks=6, block_size=4, max_num_seqs=8, max_num_batched_tokens=64)
    seqs = [m.Seq(f"s{i}", prompt_len=5, max_tokens=8) for i in range(4)]
    for q in seqs:
        s.add(q)
    steps = 0
    while s.has_work():
        run_step(s)
        assert s.blocks_used <= s.num_blocks
        steps += 1
        assert steps < 500, "no progress"
    assert s.num_preemptions > 0
    assert s.blocks_used == 0
    for q in seqs:
        assert q.status == "FINISHED"
        assert q.output == [tok(q.seq_id, k) for k in range(8)]


def test_eos_stops_early():
    s = m.Scheduler(num_blocks=10, block_size=4, max_num_seqs=8, max_num_batched_tokens=64)
    s.add(m.Seq("s1", prompt_len=3, max_tokens=100))
    s.schedule()
    s.update(lambda sid: EOS, EOS)
    assert not s.has_work() and s.finished[0].output == [EOS] and s.blocks_used == 0
```


© 2026 kzhou003

B-17 · tests/test_p06_block_manager.py

```
import pytest

from drill import load

m = load("p06_block_manager")


def test_allocate_and_slots():
    bm = m.BlockManager(num_blocks=8, block_size=4)
    table = bm.allocate("a", list(range(10)))
    assert len(table) == 3 and len(set(table)) == 3
    assert bm.num_free_blocks == 5
    for pos in range(10):
        assert bm.slot("a", pos) == table[pos // 4] * 4 + pos % 4


def test_append_crosses_block_boundary():
    bm = m.BlockManager(num_blocks=3, block_size=4)
    bm.allocate("a", list(range(7)))  # 2 blocks, 1 free slot in the 2nd
    s7 = bm.append_slot("a", 7)
    assert bm.num_free_blocks == 1
    s8 = bm.append_slot("a", 8)  # needs a 3rd block
    assert bm.num_free_blocks == 0
    t = bm.block_table("a")
    assert s7 == t[1] * 4 + 3 and s8 == t[2] * 4 + 0


def test_out_of_blocks_is_atomic():
    bm = m.BlockManager(num_blocks=2, block_size=4)
    with pytest.raises(m.OutOfBlocks):
        bm.allocate("big", list(range(9)))
    assert bm.num_free_blocks == 2
    bm.allocate("a", list(range(8)))
    with pytest.raises(m.OutOfBlocks):
        bm.append_slot("a", 8)
    assert bm.block_table("a") and len(bm.block_table("a")) == 2
    bm.free("a")
    assert bm.num_free_blocks == 2


def test_free_returns_everything():
    bm = m.BlockManager(num_blocks=6, block_size=2)
    bm.allocate("a", [1, 2, 3])
    bm.allocate("b", [4, 5, 6, 7])
    bm.free("a")
    bm.free("b")
    assert bm.num_free_blocks == 6


def test_prefix_sharing_and_refcount():
    bm = m.BlockManager(num_blocks=10, block_size=4, enable_prefix_cache=True)
    prefix = list(range(8))
    ta = bm.allocate("a", prefix + [100, 101])
    tb = bm.allocate("b", prefix + [200])
    assert bm.num_cached_tokens("b") == 8
    assert tb[:2] == ta[:2] and tb[2] != ta[2]
    assert bm.num_free_blocks == 10 - 4
    bm.free("a")
    assert bm.num_free_blocks == 10 - 3, "shared blocks are still referenced by b"


def test_cache_survives_free_and_last_token_is_never_cached():
    bm = m.BlockManager(num_blocks=10, block_size=4, enable_prefix_cache=True)
    bm.allocate("a", list(range(8)) + [9])
    bm.free("a")
    bm.allocate("c", list(range(8)) + [42])
    assert bm.num_cached_tokens("c") == 8
    bm.free("c")
    # exactly 8 tokens: only the first block may be reused, the last token must be computed
    bm.allocate("d", list(range(8)))
    assert bm.num_cached_tokens("d") == 4


def test_eviction_drops_cached_blocks():
```


© 2026 kzhou003

```
    bm = m.BlockManager(num_blocks=4, block_size=4, enable_prefix_cache=True)
    bm.allocate("a", list(range(9)))
    bm.free("a")
    bm.allocate("filler", list(range(1000, 1016)))  # needs all 4 blocks -> evicts
    bm.free("filler")
    bm.allocate("b", list(range(9)))
    assert bm.num_cached_tokens("b") == 0


def test_append_completing_a_block_registers_it():
    bm = m.BlockManager(num_blocks=8, block_size=4, enable_prefix_cache=True)
    bm.allocate("e", [5, 6, 7])
    bm.append_slot("e", 8)  # block [5,6,7,8] is now full
    bm.free("e")
    bm.allocate("f", [5, 6, 7, 8, 9])
    assert bm.num_cached_tokens("f") == 4
```


© 2026 kzhou003

B-18 · tests/test_p08_sampler.py

```
import math
import random

import torch

from drill import load

m = load("p08_sampler")
SP = m.SamplingParams
V = 50


def rand_logits(seed, n=1):
    g = torch.Generator().manual_seed(seed)
    return torch.randn(n, V, generator=g) * 3


def test_greedy():
    s = m.Sampler()
    logits = rand_logits(0, 4)
    out = s.sample(logits, list("abcd"), [SP(temperature=0.0)] * 4, [[]] * 4)
    assert out.tolist() == logits.argmax(-1).tolist()


def test_top_k_1_and_tiny_top_p_are_greedy():
    s = m.Sampler()
    logits = rand_logits(1, 3)
    greedy = logits.argmax(-1).tolist()
    assert s.sample(logits, list("abc"), [SP(top_k=1, seed=1)] * 3, [[]] * 3).tolist() == greedy
    assert s.sample(logits, list("xyz"), [SP(top_p=1e-6, seed=2)] * 3, [[]] * 3).tolist() == greedy


def test_seeded_result_is_batch_invariant():
    row = rand_logits(7)[0]
    alone = m.Sampler()
    ref = [int(alone.sample(row[None], ["a"], [SP(temperature=1.0, seed=123)], [[]])[0]) for _ in range(30)]

    mixed = m.Sampler()
    got = []
    for step in range(30):
        others = random.randint(0, 5)
        ids = [f"o{step}_{j}" for j in range(others)] + ["a"]
        random.shuffle(ids)
        logits = torch.stack([row if r == "a" else rand_logits(step * 10 + j)[0] for j, r in enumerate(ids)])
        params = [SP(temperature=1.0, seed=123) if r == "a" else SP(temperature=1.0) for r in ids]
        out = mixed.sample(logits, ids, params, [[] for _ in ids])
        got.append(int(out[ids.index("a")]))
    assert got == ref


def test_top_k_distribution():
    s = m.Sampler()
    logits = torch.full((1, V), -1e9)
    logits[0, 3], logits[0, 9] = math.log(0.7), math.log(0.3)
    logits[0, 20] = math.log(0.2)  # removed by top_k=2
    counts = {3: 0, 9: 0}
    for _ in range(4000):
        t = int(s.sample(logits, ["r"], [SP(top_k=2, seed=5)], [[]])[0])
        assert t in counts
        counts[t] += 1
    frac = counts[3] / 4000
    assert abs(frac - 0.7) < 0.04, frac


def test_repetition_penalty_values_and_effect():
    logits = torch.tensor([2.0, -2.0, 1.5])
    pen = m.apply_repetition_penalty(logits, [0, 1, 1], 2.0)
    assert torch.allclose(pen, torch.tensor([1.0, -4.0, 1.5]))
    s = m.Sampler()
    out = s.sample(logits[None], ["r"], [SP(temperature=0.0, repetition_penalty=2.0)], [[0]])
    assert int(out[0]) == 2


def test_free_resets_generator():
```


© 2026 kzhou003

```
    s = m.Sampler()
    row = rand_logits(3)
    first = [int(s.sample(row, ["r"], [SP(seed=9)], [[]])[0]) for _ in range(10)]
    s.free("r")
    again = [int(s.sample(row, ["r"], [SP(seed=9)], [[]])[0]) for _ in range(10)]
    assert first == again
```

B-19 · tests/test_p09_paged_attention.py

```
import random

import torch

from drill import load

m = load("p09_paged_attention")


def dense_attention(q, k, v, scale):
    # q [H,D], k/v [n,KVH,D]
    group = q.shape[0] // k.shape[1]
    k = k.repeat_interleave(group, 1)
    v = v.repeat_interleave(group, 1)
    p = torch.softmax(torch.einsum("hd,nhd->hn", q, k) * scale, -1)
    return torch.einsum("hn,nhd->hd", p, v)


def build_case(seed, lens, block_size=4, num_blocks=64, H=8, KVH=2, D=16):
    torch.manual_seed(seed)
    random.seed(seed)
    k_cache = torch.zeros(num_blocks, block_size, KVH, D)
    v_cache = torch.zeros_like(k_cache)
    free = list(range(num_blocks))
    random.shuffle(free)  # non-contiguous physical blocks
    max_blocks = max(-(-n // block_size) for n in lens)
    tables = torch.full((len(lens), max_blocks), 999, dtype=torch.int32)  # junk in unused slots
    dense = []
    for b, n in enumerate(lens):
        nb = -(-n // block_size)
        blocks = [free.pop() for _ in range(nb)]
        tables[b, :nb] = torch.tensor(blocks, dtype=torch.int32)
        k, v = torch.randn(n, KVH, D), torch.randn(n, KVH, D)
        slots = torch.tensor([blocks[p // block_size] * block_size + p % block_size for p in range(n)])
        m.write_kv(k_cache, v_cache, k, v, slots)
        dense.append((k, v))
    q = torch.randn(len(lens), H, D)
    return q, k_cache, v_cache, tables, torch.tensor(lens), dense


def test_matches_dense_attention():
    lens = [1, 4, 5, 13, 32, 7]
    q, kc, vc, tables, ctx, dense = build_case(0, lens)
    scale = 16 ** -0.5
    out = m.paged_decode_attention(q, kc, vc, tables, ctx, scale)
    for b, (k, v) in enumerate(dense):
        torch.testing.assert_close(out[b], dense_attention(q[b], k, v, scale), atol=1e-5, rtol=1e-5)


def test_padding_slots_are_not_written():
    kc = torch.zeros(4, 4, 1, 2)
    vc = torch.zeros_like(kc)
    k = torch.ones(3, 1, 2)
    m.write_kv(kc, vc, k, k, torch.tensor([5, -1, 6]))
    assert kc.sum() == 4  # two tokens x 2 dims
    assert kc[1, 1].sum() == 2 and kc[1, 2].sum() == 2


def test_random_shapes():
    for seed in range(5):
        lens = [random.randint(1, 40) for _ in range(random.randint(1, 6))]
        q, kc, vc, tables, ctx, dense = build_case(seed + 10, lens, block_size=random.choice([2, 4, 8]))
        out = m.paged_decode_attention(q, kc, vc, tables, ctx, 0.25)
        for b, (k, v) in enumerate(dense):
            torch.testing.assert_close(out[b], dense_attention(q[b], k, v, 0.25), atol=1e-5, rtol=1e-5)
```


© 2026 kzhou003

B-20 · tests/test_p12_shm_connector.py

```
import fcntl
import os
import subprocess
import sys
import time
import uuid

from drill import load

m = load("p12_shm_connector")


def _uniq(tag):
    return f"{tag}-{uuid.uuid4().hex[:8]}"


def test_roundtrip_and_consume_once():
    c = m.ShmConnector(namespace="t1")
    k = _uniq("req_0_0")
    c.put(k, b"hello")
    assert c.get(k) == b"hello"
    assert c.get(k) is None, "the reader consumes (unlinks) the segment"
    assert c.get(_uniq("missing")) is None


def test_empty_and_large_payloads():
    c = m.ShmConnector(namespace="t2")
    k1, k2 = _uniq("e"), _uniq("big")
    c.put(k1, b"")
    assert c.get(k1) == b""
    blob = os.urandom(4 * 1024 * 1024 + 3)
    c.put(k2, blob)
    assert c.get(k2) == blob


def test_independent_writer_process_exits_before_read():
    """Writer is a separate interpreter (its own resource tracker) that exits first."""
    ns = "t3"
    keys = [_uniq(f"r_0_{i}") for i in range(3)]
    code = (
        "from drill import load; c = load('p12_shm_connector').ShmConnector(namespace=%r)\n"
        "for k in %r: c.put(k, ('payload-' + k).encode())" % (ns, keys)
    )
    root = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
    subprocess.run([sys.executable, "-c", code], check=True, cwd=root, env={**os.environ, "PYTHONPATH": root})
    time.sleep(1.0)  # give the writer's resource tracker time to "clean up"
    c = m.ShmConnector(namespace=ns)
    for k in keys:
        assert c.get(k) == f"payload-{k}".encode(), "segment vanished when the writer exited (resource tracker?)"


def test_not_ready_while_lock_is_held():
    c = m.ShmConnector(namespace="t4")
    k = _uniq("locked")
    c.put(k, b"data")
    fd = os.open(c.lock_path(k), os.O_RDWR)
    try:
        fcntl.flock(fd, fcntl.LOCK_EX)  # pretend a writer is still writing
        assert c.get(k) is None
    finally:
        fcntl.flock(fd, fcntl.LOCK_UN)
        os.close(fd)
    assert c.get(k) == b"data"


def test_cleanup_prefix_only_touches_unread_keys_with_prefix():
    c = m.ShmConnector(namespace="t5")
    a = _uniq("reqA")
    b = _uniq("reqB")
    c.put(f"{a}_0_0", b"1")
    c.put(f"{a}_0_1", b"2")
    c.put(f"{b}_0_0", b"3")
    assert c.get(f"{a}_0_0") == b"1"  # consumed already
    assert c.cleanup_prefix(f"{a}_") == 1
```


© 2026 kzhou003

```
    assert c.get(f"{a}_0_1") is None
    assert c.get(f"{b}_0_0") == b"3"
```


© 2026 kzhou003

B-21 · tests/test_p13_chunk_adapter.py

```
import random
import threading
import time

from drill import load

m = load("p13_chunk_adapter")


def pair(build=lambda snap: list(snap), **conn_kw):
    conn = m.InMemoryConnector(**conn_kw)
    sender = m.ChunkAdapter(conn, stage_id=0, src_stage_id=None, build_payload=build)
    receiver = m.ChunkAdapter(conn, stage_id=1, src_stage_id=0, build_payload=build)
    sender.start()
    receiver.start()
    return conn, sender, receiver


def drain_until_finished(receiver, req_ids, timeout=5.0):
    got = {r: [] for r in req_ids}
    done = set()
    end = time.monotonic() + timeout
    while done != set(req_ids):
        assert time.monotonic() < end, f"timed out; done={done}"
        for rid, chunks in receiver.poll_ready().items():
            for c in chunks:
                assert rid not in done, "chunk after the final one"
                got[rid].append(c)
                if c["finished"]:
                    done.add(rid)
        time.sleep(0.001)
    return got


def stop(*adapters):
    for a in adapters:
        a.stop()


def test_ordered_delivery_single_request():
    conn, s, r = pair()
    r.load_async("q")
    for i in range(50):
        s.save_async("q", [i], is_final=(i == 49))
    got = drain_until_finished(r, ["q"])
    assert [c["data"] for c in got["q"]] == [[i] for i in range(50)]
    assert conn.puts == [f"q_0_{i}" for i in range(50)]
    stop(s, r)


def test_many_requests_with_random_latency():
    jitter = lambda key: time.sleep(random.random() * 0.0005)  # noqa: E731
    conn, s, r = pair(on_put=jitter, on_get=jitter)
    rids = [f"r{i}" for i in range(10)]
    for rid in rids:
        r.load_async(rid)
    for step in range(30):
        for rid in random.sample(rids, len(rids)):
            s.save_async(rid, [rid, step], is_final=(step == 29))
    got = drain_until_finished(r, rids)
    for rid in rids:
        assert [c["data"] for c in got[rid]] == [[rid, k] for k in range(30)]
    stop(s, r)


def test_save_async_never_blocks_the_caller():
    def slow_build(snap):
        time.sleep(0.05)
        return snap

    conn, s, r = pair(build=slow_build)
    t0 = time.monotonic()
    for i in range(10):
        s.save_async("q", [i], is_final=(i == 9))
```


© 2026 kzhou003

```
    assert time.monotonic() - t0 < 0.02
    r.load_async("q")
    drain_until_finished(r, ["q"])
    stop(s, r)


def test_snapshot_isolation():
    conn, s, r = pair()
    live = [1, 2, 3]
    r.load_async("q")
    s.save_async("q", live, is_final=True)
    live.append(4)  # scheduler keeps mutating the live request
    got = drain_until_finished(r, ["q"])
    assert got["q"][0]["data"] == [1, 2, 3]
    stop(s, r)


def test_cancel_still_delivers_final_and_ids_stay_contiguous():
    gate = threading.Event()

    def gated_build(snap):
        gate.wait()
        return snap

    conn, s, r = pair(build=gated_build)
    r.load_async("q")
    s.save_async("q", ["first"], is_final=False)
    time.sleep(0.05)  # save thread is now blocked building "first"
    for i in range(20):
        s.save_async("q", [i], is_final=False)
    s.save_async("q", ["end"], is_final=True)
    s.cancel_sender("q")
    gate.set()
    got = drain_until_finished(r, ["q"])
    assert got["q"][-1]["finished"], "the final chunk is the downstream EOS: it must survive cancel"
    assert len(got["q"]) <= 2
    assert conn.puts == [f"q_0_{i}" for i in range(len(conn.puts))], "chunk ids must be contiguous"
    stop(s, r)


def test_none_payload_is_skipped_without_gap():
    conn, s, r = pair(build=lambda snap: None if snap[0] % 2 else snap)
    r.load_async("q")
    for i in range(6):
        s.save_async("q", [i], is_final=(i == 5))
    got = drain_until_finished(r, ["q"])
    assert [c["data"] for c in got["q"]] == [[0], [2], [4], None]
    assert conn.puts == ["q_0_0", "q_0_1", "q_0_2", "q_0_3"]
    stop(s, r)


def test_cleanup_is_a_commit_barrier_for_in_flight_reads():
    in_get = threading.Event()

    def slow_get(key):
        in_get.set()
        time.sleep(0.2)

    conn, s, r = pair(on_get=slow_get)
    conn.put("q_0_0", {"data": "late", "finished": False})
    r.load_async("q")
    assert in_get.wait(1.0)
    r.cleanup_receiver("q")  # request aborted while the recv thread is inside get()
    time.sleep(0.3)
    assert r.poll_ready() == {}, "a chunk read before cleanup must not be committed after it"
    stop(s, r)
```


© 2026 kzhou003

B-22 · tests/test_p15_talker_inputs.py

```
import pytest
import torch

from drill import load

m = load("p15_talker_inputs")
D = 4


def rows(n, base=0):
    return torch.arange(base, base + n, dtype=torch.float32)[:, None].repeat(1, D)


def test_stop_finish_drops_one_row_and_id():
    p = m.trim_thinker_payload([1, 2, 3], [7, 8, 151645], rows(7), "stop")  # one extra row captured
    assert p.prefill.shape == (3, D) and p.decode.shape == (2, D)
    assert p.output_ids == [7, 8]
    assert torch.equal(p.decode[:, 0], torch.tensor([3.0, 4.0]))


def test_length_finish_keeps_everything():
    p = m.trim_thinker_payload([1, 2], [5, 6, 7], rows(5), "length")
    assert p.decode.shape == (3, D) and p.output_ids == [5, 6, 7]


def test_under_captured_prefill_raises():
    with pytest.raises(ValueError):
        m.trim_thinker_payload([1, 2, 3], [4], rows(2), "length")


def test_build_prefill_layout():
    P = 3
    prompt = rows(P)
    first = torch.full((D,), 100.0)
    spk, pad, bos = torch.full((D,), 1.0), torch.full((D,), 10.0), torch.full((D,), 1000.0)
    ids, emb = m.build_talker_prefill(prompt, first, spk, pad, bos, mask_id=8296, pad_id=8292, start_id=8293)
    assert ids == [8296, 8296, 8296, 8292, 8293]
    assert emb.shape == (P + 2, D)
    assert torch.equal(emb[:P], prompt)
    assert torch.equal(emb[P], spk + pad) and torch.equal(emb[P + 1], first + bos)


def test_decode_consumes_one_row_then_falls_back():
    q = rows(2, base=50)
    last = torch.full((D,), 0.5)
    fb = torch.full((D,), -1.0)
    x, q = m.talker_decode_input(q, last, fb)
    assert torch.equal(x, torch.full((D,), 50.5)) and q.shape[0] == 1
    x, q = m.talker_decode_input(q, last, fb)
    assert torch.equal(x, torch.full((D,), 51.5)) and q.shape[0] == 0
    x, q = m.talker_decode_input(q, last, fb)
    assert torch.equal(x, torch.full((D,), -0.5)) and q.shape[0] == 0
```


© 2026 kzhou003

B-23 · tests/test_p14_codec_windows.py

```
import random

from drill import load

m = load("p14_codec_windows")
UP = 3


def toy_vocoder(codes, receptive_field):
    """Causal and local: sample j of frame i depends on codes[i-rf .. i] *within the given window*."""
    out = []
    for i in range(len(codes)):
        ctx = codes[max(0, i - receptive_field):i + 1]
        out += [sum(ctx) * 10 + j for j in range(UP)]
    return out


def stream(codes, arrivals, *, initial, chunk, left_ctx, rf):
    """Simulate the talker producing frames in bursts and the window planner reacting."""
    emitted, produced, audio, windows = 0, 0, [], []
    for k, burst in enumerate(arrivals):
        produced += burst
        finished = k == len(arrivals) - 1
        while True:
            w = m.next_window(produced, emitted, finished, initial=initial, chunk=chunk, left_ctx=left_ctx)
            if w is None:
                break
            ctx_start, new_start, new_end = w
            decoded = toy_vocoder(codes[ctx_start:new_end], rf)
            audio += m.slice_new_samples(decoded, w, UP)
            windows.append(w)
            emitted = new_end
    return audio, windows


def test_initial_then_steady_then_flush():
    assert m.next_window(3, 0, False, initial=4, chunk=25, left_ctx=25) is None
    assert m.next_window(4, 0, False, initial=4, chunk=25, left_ctx=25) == (0, 0, 4)
    assert m.next_window(20, 4, False, initial=4, chunk=25, left_ctx=25) is None
    assert m.next_window(29, 4, False, initial=4, chunk=25, left_ctx=25) == (0, 4, 29)
    assert m.next_window(60, 29, False, initial=4, chunk=25, left_ctx=25) == (4, 29, 54)
    assert m.next_window(60, 54, True, initial=4, chunk=25, left_ctx=25) == (29, 54, 60)
    assert m.next_window(60, 60, True, initial=4, chunk=25, left_ctx=25) is None


def test_no_initial_chunk():
    assert m.next_window(10, 0, False, initial=0, chunk=10, left_ctx=2) == (0, 0, 10)
    assert m.next_window(20, 10, False, initial=0, chunk=10, left_ctx=2) == (8, 10, 20)


def test_windows_tile_the_stream_exactly():
    for seed in range(20):
        random.seed(seed)
        total = random.randint(1, 200)
        arrivals, left = [], total
        while left:
            b = random.randint(1, 7)
            arrivals.append(min(b, left))
            left -= arrivals[-1]
        codes = [random.randint(0, 9) for _ in range(total)]
        _, windows = stream(codes, arrivals, initial=4, chunk=13, left_ctx=5, rf=5)
        covered = [f for (_, s, e) in windows for f in range(s, e)]
        assert covered == list(range(total)), "gap or overlap in shipped frames"


def test_streaming_equals_full_decode_iff_left_context_covers_receptive_field():
    random.seed(0)
    codes = [random.randint(0, 9) for _ in range(97)]
    arrivals = [1] * 97
    full = toy_vocoder(codes, receptive_field=4)
    ok, _ = stream(codes, arrivals, initial=4, chunk=10, left_ctx=4, rf=4)
    assert ok == full
```


© 2026 kzhou003

```
    bad, _ = stream(codes, arrivals, initial=4, chunk=10, left_ctx=2, rf=4)
    assert bad != full, "too little left context must produce seams"
```

B-24 · tests/test_p18_mrope_positions.py

```
import torch

from drill import load

m = load("p18_mrope_positions")


def test_text_only_is_plain_rope():
    pos, delta = m.mrope_positions([("text", 5)])
    assert pos.tolist() == [[0, 1, 2, 3, 4]] * 3
    assert delta == 0
    assert m.decode_position(5, delta).tolist() == [5, 5, 5]


def test_text_image_text():
    pos, delta = m.mrope_positions([("text", 2), ("image", (1, 4, 4)), ("text", 3)])
    assert pos.tolist() == [
        [0, 1, 2, 2, 2, 2, 4, 5, 6],
        [0, 1, 2, 2, 3, 3, 4, 5, 6],
        [0, 1, 2, 3, 2, 3, 4, 5, 6],
    ]
    assert delta == 7 - 9
    assert m.decode_position(9, delta).tolist() == [7, 7, 7]


def test_multi_frame_rectangular_grid():
    pos, delta = m.mrope_positions([("image", (2, 2, 6)), ("text", 1)], spatial_merge=2)
    # llm grid t=2, h=1, w=3 -> 6 tokens
    assert pos[:, :6].tolist() == [
        [0, 0, 0, 1, 1, 1],
        [0, 0, 0, 0, 0, 0],
        [0, 1, 2, 0, 1, 2],
    ]
    assert pos[:, 6].tolist() == [3, 3, 3]
    assert delta == 4 - 7


def test_shapes_and_dtype():
    pos, _ = m.mrope_positions([("text", 3), ("image", (1, 8, 8)), ("text", 2), ("image", (1, 2, 2))])
    assert pos.shape == (3, 3 + 16 + 2 + 1)
    assert pos.dtype == torch.long
```


© 2026 kzhou003

B-25 · tests/test_p16_init_planner.py

```
import threading
import time

import pytest

from drill import load

m = load("p16_init_planner")
GiB = 1 << 30


def test_groups_are_connected_components():
    devs = {0: {0}, 1: {1}, 2: {1}, 3: {2, 3}, 4: {3}, 5: {4}}
    assert m.init_groups(devs) == [[0], [1, 2], [3, 4], [5]]
    assert m.init_groups({0: {0, 1}, 1: {1, 2}, 2: {2}}) == [[0, 1, 2]]


def test_groups_run_in_parallel_and_sequential_within_group():
    devs = {0: {0}, 1: {1}, 2: {1}}
    started = {s: threading.Event() for s in devs}
    active = {"g1": 0, "max_g1": 0}
    order = []
    lock = threading.Lock()

    def init(sid):
        started[sid].set()
        with lock:
            order.append(sid)
            if sid in (1, 2):
                active["g1"] += 1
                active["max_g1"] = max(active["max_g1"], active["g1"])
        if sid == 0:
            # only returns if stage 1 starts while stage 0 is still initializing
            assert started[1].wait(2.0), "groups did not run in parallel"
        time.sleep(0.05)
        with lock:
            if sid in (1, 2):
                active["g1"] -= 1
        return f"engine{sid}"

    res = m.run_init(devs, init, cleanup_fn=lambda s, r: None)
    assert res == {0: "engine0", 1: "engine1", 2: "engine2"}
    assert active["max_g1"] == 1, "stages sharing a device must not init concurrently"
    assert order.index(1) < order.index(2)


def test_failure_short_circuits_and_rolls_back():
    devs = {0: {0}, 1: {1}, 2: {1}}
    calls, cleaned = [], []
    stage0_started = threading.Event()

    def init(sid):
        calls.append(sid)
        if sid == 0:
            stage0_started.set()
        if sid == 1:
            stage0_started.wait(2.0)
            raise RuntimeError("OOM while capturing graphs")
        time.sleep(0.05)
        return sid

    with pytest.raises(m.InitError) as ei:
        m.run_init(devs, init, cleanup_fn=lambda s, r: cleaned.append(s))
    assert ei.value.stage_id == 1
    assert 2 not in calls, "stage 2 must not start after stage 1 failed in the same group"
    assert cleaned == [0], "successfully initialized stages are cleaned up"


def test_admission_check():
    totals = {0: 32 * GiB}
    ok = {0: ({0}, 0.5), 1: ({0}, 0.3)}
    assert m.admission_check(totals, ok, reserve_bytes_per_stage=GiB) == []
    bad = {0: ({0}, 0.6), 1: ({0}, 0.3), 2: ({0}, 0.1)}
```


© 2026 kzhou003

```
    errs = m.admission_check(totals, bad, reserve_bytes_per_stage=GiB)
    assert len(errs) == 1 and "device 0" in errs[0]
```

B-26 · tests/test_p17_graph_runner.py

```
import torch

from drill import load

m = load("p17_graph_runner")
H = 8
W = torch.randn(H, H, generator=torch.Generator().manual_seed(0))


def fn(x):
    return torch.tanh(x @ W) + 1.0


def make():
    r = m.GraphRunner(fn, buckets=[1, 2, 4, 8], hidden=H)
    r.capture()
    return r


def test_matches_eager_for_every_batch_size():
    r = make()
    for bs in range(1, 9):
        x = torch.randn(bs, H)
        torch.testing.assert_close(r.run(x), fn(x))
    assert r.stats["eager"] == 0 and r.stats["replays"] == 8


def test_bucket_selection_and_eager_fallbacks():
    r = make()
    r.run(torch.randn(3, H))
    assert r.stats["last_bucket"] == 4
    r.run(torch.randn(9, H))  # larger than every bucket
    assert r.stats["eager"] == 1
    r.run(torch.randn(2, H), force_eager=True)  # e.g. a batch with per-row RNG generators
    assert r.stats["eager"] == 2


def test_result_does_not_alias_static_buffers():
    r = make()
    x1, x2 = torch.randn(2, H), torch.randn(2, H)
    y1 = r.run(x1)
    expected = fn(x1)
    r.run(x2)
    torch.testing.assert_close(y1, expected, msg="y1 was overwritten by the next replay")


def test_padding_rows_are_zeroed():
    r = make()
    r.run(torch.full((4, H), 7.0))
    r.run(torch.randn(3, H))
    assert torch.count_nonzero(r.static_in[3]) == 0


def test_captured_once_per_bucket():
    r = make()
    assert sorted(r.graphs) == [1, 2, 4, 8]
    graphs = dict(r.graphs)
    for bs in (1, 3, 5, 8):
        r.run(torch.randn(bs, H))
    assert all(r.graphs[b] is graphs[b] for b in graphs), "never re-capture at run time"
```


© 2026 kzhou003

Appendix C · Reference solutions (commented; `Pitfall:` marks easy-to-get-wrong spots)

C-1 · solutions/m01_rmsnorm_mrope.py

```
import torch


# ---------------------------------------------------------------------------
# rms_norm: the shared implementation of every RMSNorm in the Qwen3 family (input_layernorm /
# post_attention_layernorm / q_norm / k_norm / final norm). Algorithm: y = weight * (x / sqrt(mean(x², dim=-1) + eps)).
# Same order as HF Qwen3OmniMoe*RMSNorm: upcast to fp32 to normalize, cast back to the input dtype, then multiply by
# weight.
# ---------------------------------------------------------------------------
def rms_norm(x: torch.Tensor, weight: torch.Tensor, eps: float) -> torch.Tensor:
    # fp32 upcast: summing x² in bf16/fp16 easily loses precision or even overflows, so the statistic must be computed
    # in fp32
    x32 = x.float()
    # mean only over the last dim (hidden / head_dim): each token (each head) is normalized independently; keepdim for
    # broadcasting
    x32 = x32 * torch.rsqrt(x32.pow(2).mean(-1, keepdim=True) + eps)
    # Pitfall: you must .to(x.dtype) BEFORE multiplying by weight (cast-back-before-weight).
    # Multiplying by weight in fp32 and casting afterwards can differ from HF by 1 ulp in bf16, failing element-wise
    # comparison.
    return weight * x32.to(x.dtype)  # normalize in fp32, scale in the input dtype (HF order)


# ---------------------------------------------------------------------------
# rope_inv_freq: the standard RoPE inverse-frequency table, inv_freq[i] = theta^(-2i / head_dim), i = 0..head_dim/2-1.
# Each frequency owns one pair of head_dim dims (in the rotate_half layout: dim i and dim i + D/2).
# ---------------------------------------------------------------------------
def rope_inv_freq(head_dim: int, theta: float) -> torch.Tensor:
    # arange(0, D, 2) = [0, 2, 4, ...], length D/2; explicit float32 to avoid integer math / low precision
    return 1.0 / (theta ** (torch.arange(0, head_dim, 2, dtype=torch.float32) / head_dim))


# ---------------------------------------------------------------------------
# mrope_cos_sin: Multimodal RoPE (M-RoPE). Each token has 3 position components (t, h, w);
# the D/2 frequencies are split across the three axes by mrope_section: angle of freq j = pos[axis owning j] *
# inv_freq[j].
# Two assignment layouts:
#   interleaved=True  (Qwen3-Omni / Qwen3-VL): T H W T H W ... round-robin; once H/W run out, the rest all go to T
#   interleaved=False (Qwen2-VL / Qwen2.5-Omni): contiguous chunks [T...T | H...H | W...W]
# For pure text t = h = w, and both layouts reduce to plain 1D RoPE.
# ---------------------------------------------------------------------------
def mrope_cos_sin(
    inv_freq: torch.Tensor,     # [head_dim // 2]
    positions: torch.Tensor,    # [3, T]  (temporal, height, width)
    mrope_section: list[int],   # sums to head_dim // 2
    interleaved: bool,
) -> tuple[torch.Tensor, torch.Tensor]:  # cos, sin: [T, head_dim]
    # First compute each of the 3 axes' angles over all D/2 frequencies, then "pick" which axis each frequency uses per
    # layout
    freqs = positions.float()[:, :, None] * inv_freq[None, None, :]  # [3, T, half]
    if interleaved:  # Qwen3-Omni / Qwen3-VL: T H W T H W ... T T
        # Start from the T axis (clone so freqs[0] itself is not modified), then overwrite the frequency slots owned by
        # H / W
        f = freqs[0].clone()  # [T, half]
        # axis=1 (H) owns indices 1, 4, 7, ...; axis=2 (W) owns indices 2, 5, 8, ... (the offset equals the axis id)
        for axis, offset in ((1, 1), (2, 2)):
            # Pitfall: the slice stop is mrope_section[axis] * 3, not half --
            # each axis takes only section[axis] frequencies; the tail after the interleaved region stays on the T axis
            idx = slice(offset, mrope_section[axis] * 3, 3)
            f[..., idx] = freqs[axis][..., idx]
    else:  # Qwen2-VL / Qwen2.5-Omni: contiguous chunks [TTTT HH WW]
        # Segment `axis` takes that axis's angles over frequency range [start, start + n); the three segments concat
        # back to half
        parts, start = [], 0
        for axis, n in enumerate(mrope_section):
            parts.append(freqs[axis][..., start:start + n])
            start += n
        f = torch.cat(parts, dim=-1)  # [T, half]
    # Duplicate to [T, D]: the first and second halves share the same angles, matching rotate_half's "first half /
    # second half"
    # pairing layout (not the layout that pairs adjacent dims (2i, 2i+1))
    emb = torch.cat([f, f], dim=-1)  # [T, head_dim]
    return emb.cos(), emb.sin()
```


© 2026 kzhou003

```
# rotate_half: turn [x1, x2] (each D/2) into [-x2, x1], i.e. the term multiplied by sin in a 2D rotation.
def rotate_half(x: torch.Tensor) -> torch.Tensor:
    x1, x2 = x.chunk(2, dim=-1)  # split head_dim in half
    return torch.cat([-x2, x1], dim=-1)


# apply_rotary: x' = x * cos + rotate_half(x) * sin, i.e. rotate each pair (x_i, x_{i+D/2}) by angle θ_i.
def apply_rotary(x: torch.Tensor, cos: torch.Tensor, sin: torch.Tensor) -> torch.Tensor:
    """x: [T, num_heads, head_dim]; cos/sin: [T, head_dim]."""
    # [T, D] -> [T, 1, D] to broadcast over the head dim;
    # cast cos/sin to x.dtype before multiplying (HF's rotary embedding returns cos/sin in x.dtype)
    cos, sin = cos[:, None, :].to(x.dtype), sin[:, None, :].to(x.dtype)
    return x * cos + rotate_half(x) * sin
```

C-2 · solutions/m02_thinker_attention.py

```
from dataclasses import dataclass, field

import torch


# ------------------------------------------------------------------ given types
# AttnWeights: all weights of one thinker self-attention layer (bare tensors pulled out of the HF module).
# GQA: num_heads query heads share num_kv_heads groups of K/V (num_heads % num_kv_heads == 0).
@dataclass
class AttnWeights:
    q_proj: torch.Tensor  # [num_heads * head_dim, hidden]
    k_proj: torch.Tensor  # [num_kv_heads * head_dim, hidden]
    v_proj: torch.Tensor  # [num_kv_heads * head_dim, hidden]
    o_proj: torch.Tensor  # [hidden, num_heads * head_dim]
    q_norm: torch.Tensor  # [head_dim]  RMSNorm weight applied per head
    k_norm: torch.Tensor  # [head_dim]
    num_heads: int
    num_kv_heads: int
    head_dim: int
    eps: float = 1e-6


# KVCache: one layer's KV cache, appended by chunk (prefill is one big chunk, each decode step one [1, ...] chunk).
# Invariant: len(cache) = number of cached tokens = absolute start position of the next new tokens (cache offset).
@dataclass
class KVCache:
    k: list[torch.Tensor] = field(default_factory=list)  # chunks of [t, num_kv_heads, head_dim]
    v: list[torch.Tensor] = field(default_factory=list)

    def __len__(self) -> int:
        return sum(x.shape[0] for x in self.k)


def from_hf(att) -> AttnWeights:
    """Pull the weights out of a HF Qwen3OmniMoeThinkerTextAttention."""
    return AttnWeights(
        att.q_proj.weight.detach(), att.k_proj.weight.detach(), att.v_proj.weight.detach(),
        att.o_proj.weight.detach(), att.q_norm.weight.detach(), att.k_norm.weight.detach(),
        att.config.num_attention_heads, att.config.num_key_value_heads, att.head_dim, att.config.rms_norm_eps,
    )


# ------------------------------------------------------------------ implementation
# _rms: same as m01.rms_norm (normalize in fp32 -> cast back to the original dtype -> multiply by weight), over each
# head's head_dim.
def _rms(x: torch.Tensor, w: torch.Tensor, eps: float) -> torch.Tensor:
    x32 = x.float()
    return w * (x32 * torch.rsqrt(x32.pow(2).mean(-1, keepdim=True) + eps)).to(x.dtype)


# _rotate_half: [a, b] -> [-b, a]; used by _rope for the RoPE rotation (same as m01.rotate_half).
def _rotate_half(x: torch.Tensor) -> torch.Tensor:
    a, b = x.chunk(2, dim=-1)
    return torch.cat([-b, a], dim=-1)


# _rope: x [T, heads, D], cos/sin [T, D] -> insert a 1 on the head dim to broadcast.
def _rope(x: torch.Tensor, cos: torch.Tensor, sin: torch.Tensor) -> torch.Tensor:
    return x * cos[:, None] + _rotate_half(x) * sin[:, None]


# ------------------------------------------------------------------------------
# attention_forward: one Qwen3-Omni thinker self-attention layer (QK-norm + GQA + KV cache).
# Flow: q/k/v projection -> split into heads -> per-head RMSNorm on q, k (QK-norm) -> RoPE
#       -> append new k/v to the cache -> GQA-expand k/v -> causal-masked softmax(QK^T/√D) V -> o_proj.
# The same function does prefill (T new tokens, empty cache) and decode (T=1, cache already holds `past` tokens).
# ------------------------------------------------------------------------------
def attention_forward(
    w: AttnWeights,
    x: torch.Tensor,    # [T, hidden]  new tokens only
```


© 2026 kzhou003

```
    cos: torch.Tensor,  # [T, head_dim] for the new tokens
    sin: torch.Tensor,
    cache: KVCache | None = None,
) -> torch.Tensor:      # [T, hidden]
    T = x.shape[0]
    H, KVH, D = w.num_heads, w.num_kv_heads, w.head_dim
    # After projection, view as [T, heads, D], then RMSNorm over D.
    # Pitfall: QK-norm must come BEFORE RoPE (HF order); V is not normed.
    q = _rms((x @ w.q_proj.T).view(T, H, D), w.q_norm, w.eps)  # QK-norm BEFORE rope
    k = _rms((x @ w.k_proj.T).view(T, KVH, D), w.k_norm, w.eps)  # [T, KVH, D]
    v = (x @ w.v_proj.T).view(T, KVH, D)  # [T, KVH, D]
    # cos/sin cover only the new tokens' absolute positions; old k in the cache was rotated when written, don't rotate
    # it again
    q, k = _rope(q, cos, sin), _rope(k, cos, sin)

    # past = number of tokens cached before this call = absolute start position of the new tokens (cache offset)
    past = 0
    if cache is not None:
        past = len(cache)  # Pitfall: take the length BEFORE append, otherwise the offset is over by T
        # The cache stores k "after RoPE, before GQA expansion" (only KVH groups, saves memory)
        cache.k.append(k)
        cache.v.append(v)
        k, v = torch.cat(cache.k), torch.cat(cache.v)  # [S, KVH, D], S = past + T

    # GQA: each kv head serves `group` consecutive q heads (q head h uses kv head h // group).
    # Pitfall: must be repeat_interleave ([k0,k0,k1,k1]), not repeat ([k0,k1,k0,k1]), or the head mapping is scrambled
    group = H // KVH
    k = k.repeat_interleave(group, dim=1)  # [S, H, D]
    v = v.repeat_interleave(group, dim=1)
    # Scores in fp32 (softmax is precision-sensitive); scores[h, t, s] = q_t · k_s / √D
    scores = torch.einsum("thd,shd->hts", q.float(), k.float()) * D ** -0.5  # [H, T, S]
    S = k.shape[0]
    # Build the causal mask from ABSOLUTE positions: new token t is at past + t, key s is at s.
    # Pitfall: a relative-index form like triu(ones(T, S), 1) wrongly masks out history
    # when T < S (decode / chunked prefill)
    q_pos = torch.arange(past, past + T)[:, None]  # [T, 1]
    k_pos = torch.arange(S)[None, :]  # [1, S]
    scores = scores.masked_fill(k_pos > q_pos, float("-inf"))  # causal w.r.t. absolute positions
    probs = torch.softmax(scores, dim=-1)  # normalize over the key dim S
    # The weighted sum stays in fp32; cast the result back to the input dtype before o_proj
    out = torch.einsum("hts,shd->thd", probs, v.float()).to(x.dtype)  # [T, H, D]
    # Concat heads back to [T, H*D] (head-major, D minor, matching o_proj's column layout), then project back to hidden
    return out.reshape(T, H * D) @ w.o_proj.T
```

C-3 · solutions/m03_sparse_moe.py

```
from dataclasses import dataclass

import torch
import torch.nn.functional as F


# ---------------------------------------------------------------- given types
# SharedExpert: the talker's MoE block also has a shared expert (SwiGLU MLP) that "every token goes through",
# whose output is scaled by a per-token sigmoid gate. The thinker's MoE has no shared expert (shared=None).
@dataclass
class SharedExpert:
    gate_proj: torch.Tensor   # [I_s, H]
    up_proj: torch.Tensor     # [I_s, H]
    down_proj: torch.Tensor   # [H, I_s]
    gate: torch.Tensor        # [1, H]   sigmoid gate of the shared expert


# MoeWeights: the weights of the E experts stacked into big [E, ...] tensors, indexed by expert id.
@dataclass
class MoeWeights:
    router: torch.Tensor      # [E, H]
    w_gate: torch.Tensor      # [E, I, H]   stacked expert gate_proj
    w_up: torch.Tensor        # [E, I, H]
    w_down: torch.Tensor      # [E, H, I]
    top_k: int
    norm_topk_prob: bool
    shared: SharedExpert | None = None


def from_hf(block) -> MoeWeights:
    """HF Qwen3OmniMoe{Thinker,Talker}TextSparseMoeBlock -> stacked tensors."""
    ex = block.experts
    shared = None
    # Only the talker's block has the shared_expert / shared_expert_gate attributes
    if hasattr(block, "shared_expert"):
        s = block.shared_expert
        shared = SharedExpert(s.gate_proj.weight.detach(), s.up_proj.weight.detach(),
                              s.down_proj.weight.detach(), block.shared_expert_gate.weight.detach())
    return MoeWeights(
        block.gate.weight.detach(),  # HF calls the router `gate` (don't confuse it with the experts' gate_proj)
```


© 2026 kzhou003

```
        torch.stack([e.gate_proj.weight.detach() for e in ex]),
        torch.stack([e.up_proj.weight.detach() for e in ex]),
        torch.stack([e.down_proj.weight.detach() for e in ex]),
        block.top_k, block.norm_topk_prob, shared,
    )


# ---------------------------------------------------------- implementation
# ---------------------------------------------------------------------------
# route: router logits -> softmax (over all E experts) -> top-k -> optional renormalization -> cast back to x.dtype.
# Pitfall: the order is "softmax first, then top-k" (not softmax over the top-k logits);
#       with norm_topk_prob=False the top-k weights sum to < 1; don't normalize on your own.
# ---------------------------------------------------------------------------
def route(x: torch.Tensor, w: MoeWeights) -> tuple[torch.Tensor, torch.Tensor]:
    """-> (weights [N, k] in x.dtype, expert ids [N, k]). softmax over ALL experts, then top-k."""
    # logits: [N, E]; softmax in fp32 (dtype=torch.float), same as HF
    probs = F.softmax(x @ w.router.T, dim=-1, dtype=torch.float)
    topw, topi = torch.topk(probs, w.top_k, dim=-1)  # [N, k] weights / expert ids, sorted by weight descending
    if w.norm_topk_prob:
        # Renormalize only among the k selected so each token's k weights sum to 1 (still in fp32)
        topw = topw / topw.sum(-1, keepdim=True)
    # Cast back to x.dtype only at the end: routing weights multiply the expert outputs (x.dtype)
    return topw.to(x.dtype), topi


# ---------------------------------------------------------------------------
# moe_forward: sparse MoE forward (the "group by expert" dispatch / combine formulation).
# 1) Flatten N tokens × k slots into N*k (token, expert, weight) triples;
# 2) stable-sort by expert id and count how many triples each expert gets;
# 3) per expert, gather its tokens, run one SwiGLU MLP, scale by routing weights, index_add_ back to their tokens;
# 4) if there is a shared expert, add sigmoid(gate) * shared_mlp(x) to every token.
# ---------------------------------------------------------------------------
def moe_forward(x: torch.Tensor, w: MoeWeights) -> torch.Tensor:
    N, H = x.shape  # x: [N, H], N tokens (batch/seq already flattened)
    topw, topi = route(x, w)  # [N, k], [N, k]
    flat_e = topi.reshape(-1)                       # [N*k]
    # Triple i belongs to token i // k: [0,0,..,1,1,..], matching reshape(-1)'s row-major order
    flat_tok = torch.arange(N).repeat_interleave(w.top_k)  # [N*k]
    flat_w = topw.reshape(-1)  # [N*k]
    # Stable sort: tokens keep their original order within an expert, so results are reproducible
    order = torch.argsort(flat_e, stable=True)     # group (token, slot) pairs by expert
    # counts[e] = triples assigned to expert e; minlength=E gives unselected experts an entry too (value 0)
    counts = torch.bincount(flat_e, minlength=w.w_gate.shape[0]).tolist()
    out = torch.zeros_like(x)  # [N, H] accumulator
    # start points at the beginning of the current expert's segment in order; segments are contiguous in ascending e
    start = 0
    for e, n in enumerate(counts):
        if n == 0:
            continue                                # idle expert: no work at all
        sel = order[start:start + n]  # indices into flat_* of the triples belonging to expert e
        start += n
        toks = flat_tok[sel]  # [n] token ids (top-k has no duplicates: a token appears at most once per expert)
        h = x[toks]  # [n, H] gather
        # SwiGLU: down( silu(gate(h)) * up(h) ); w_gate[e]: [I, H], h @ w_gate[e].T -> [n, I]
        y = (F.silu(h @ w.w_gate[e].T) * (h @ w.w_up[e].T)) @ w.w_down[e].T  # [n, H]
        # combine: scale by routing weights, then scatter-add per token.
        # Pitfall: a token is processed by k different experts, so you must accumulate with index_add_;
        # out[toks] = ... would overwrite
        out.index_add_(0, toks, y * flat_w[sel, None])
    if w.shared is not None:
        s = w.shared
        # The shared expert runs on all tokens (no router)
        y = (F.silu(x @ s.gate_proj.T) * (x @ s.up_proj.T)) @ s.down_proj.T  # [N, H]
        # The gate is a per-token scalar: x @ gate.T -> [N, 1], broadcast over H after sigmoid
        out = out + torch.sigmoid(x @ s.gate.T) * y
    return out
```

C-4 · solutions/m04_mm_merge_deepstack.py

```
from collections.abc import Callable

import torch


# ---------------------------------------------------------------------------
# audio_token_count: the number of <audio> placeholders preprocessing puts in the prompt must exactly equal
# the number of tokens the AuT audio encoder actually outputs, or merge_multimodal won't line up later.
# Rule: every full 100 mel frames -> 13 tokens; the remaining r frames go through three stride-2 convs, each L ->
# (L-1)//2 + 1.
# Uses only // % + -, so it works on both int and LongTensor (element-wise).
# ---------------------------------------------------------------------------
def audio_token_count(num_mel_frames):
    """How many <audio> placeholder tokens the AuT encoder produces for `num_mel_frames` frames.

    The encoder works on 100-frame windows (each -> 13 tokens after three stride-2 convs)
    plus a remainder window that goes through the same three convs.
```


© 2026 kzhou003

```
    """
    rem = num_mel_frames % 100  # the tail shorter than one window
    feat = (rem - 1) // 2 + 1  # length after the 1st stride-2 conv
    # The 2nd and 3rd convs are nested in one expression: ((feat-1)//2+1) is the 2nd, the outer one is the 3rd;
    # then add the full-window part (n // 100) * 13 (100 -> 50 -> 25 -> 13).
    # Edge case: with rem == 0, feat = (-1)//2 + 1 = 0 (floor division), the next two are also 0, so the tail adds 0
    # tokens.
    # Pitfall: must use floor division //; a truncate-toward-zero form like int((rem-1)/2) gives 1 when rem == 0.
    return ((feat - 1) // 2 + 1 - 1) // 2 + 1 + (num_mel_frames // 100) * 13


# ------------------------------------------------------------------------
# merge_multimodal: replace, in order, the rows at each modality's placeholder positions in the text embedding with
# encoder outputs, giving the thinker's inputs_embeds. Also returns a bool mask per modality (DeepStack needs the visual
# mask).
# ------------------------------------------------------------------------
def merge_multimodal(
    input_ids: torch.Tensor,                # [T]
    text_embeds: torch.Tensor,              # [T, H]  embed_tokens(input_ids)
    placeholder_ids: dict[str, int],        # modality -> placeholder token id
    mm_embeds: dict[str, torch.Tensor],     # modality -> [n_tokens, H] encoder outputs, in order
) -> tuple[torch.Tensor, dict[str, torch.Tensor]]:
    # clone: the spec forbids modifying the caller's text_embeds in place
    out = text_embeds.clone()
    masks: dict[str, torch.Tensor] = {}
    for modality, tok in placeholder_ids.items():
        mask = input_ids == tok  # [T] bool, this modality's placeholder positions
        # Record the mask even if this modality has no embeddings this time, so the caller can index uniformly by
        # modality
        masks[modality] = mask
        emb = mm_embeds.get(modality)
        n = int(mask.sum())  # number of placeholders
        if emb is None:
            # Placeholders without embeddings is an error; having neither is a normal skip
            if n:
                raise ValueError(f"{n} {modality} placeholders but no {modality} embeddings")
            continue
        # Counts must match exactly (bool-index assignment also requires RHS rows == number of True in the mask)
        if emb.shape[0] != n:
            raise ValueError(f"{modality}: {n} placeholders vs {emb.shape[0]} embeddings")
        # Bool-mask assignment fills rows 0, 1, 2... of emb in order along T (the "in order" semantics);
        # encoder output dtype may differ from the text embedding, so cast first
        out[mask] = emb.to(out.dtype)
    return out, masks


# ------------------------------------------------------------------------
# apply_deepstack: DeepStack ADDS (does not replace) visual-encoder intermediate-layer features at the visual token
# positions of some LLM layer's output.
# ------------------------------------------------------------------------
def apply_deepstack(hidden: torch.Tensor, visual_mask: torch.Tensor, visual_embeds: torch.Tensor) -> torch.Tensor:
    """Add one DeepStack level to the hidden states at visual positions (returns a new tensor)."""
    # Return a new tensor, don't modify the input (hidden may still be referenced elsewhere, e.g. the residual)
    out = hidden.clone()  # [T, H]
    # Only the n_vis rows where visual_mask is True get visual_embeds[n_vis, H] added; text rows are unchanged
    out[visual_mask] = out[visual_mask] + visual_embeds.to(out.dtype)
    return out


# ------------------------------------------------------------------------
# run_layers: decoder layer loop with DeepStack. Level i features are injected AFTER layer i runs;
# there are usually fewer levels than layers, so only the first len(deepstack_embeds) layers get an injection.
# ------------------------------------------------------------------------
def run_layers(
    layers: list[Callable[[torch.Tensor], torch.Tensor]],
    hidden: torch.Tensor,
    visual_mask: torch.Tensor,
    deepstack_embeds: list[torch.Tensor],
) -> torch.Tensor:
    """Decoder stack with DeepStack: level i is injected right AFTER layer i."""
    for i, layer in enumerate(layers):
        hidden = layer(hidden)
        # Pitfall: injection happens after layer i (not before); level 0 is added to layer 0's output
        if i < len(deepstack_embeds):
            hidden = apply_deepstack(hidden, visual_mask, deepstack_embeds[i])
    return hidden
```

C-5 · solutions/m05_talker_mtp.py

```
import torch


# ------------------------------------------------------------------------
# Talker MTP (multi-token prediction) / code predictor:
# each audio frame has G codebooks (CB0 + G-1 residual codebooks). The talker trunk only samples CB0;
# the code predictor (a small transformer) starts from [talker last hidden state, E(CB0)] and autoregressively predicts
```


© 2026 kzhou003

```
# CB1..CB{G-1}:
# step j predicts CB{j+1} with lm_head[j], then embeds it with codec_embedding[j] and appends it to the sequence.
# The two implementations below must match element-wise: re-prefill (recompute the whole sequence each step) and
# incremental KV cache.
# ------------------------------------------------------------------------
@torch.no_grad()
def predict_residual_codes(cp, last_talker_hidden: torch.Tensor, cb0_embed: torch.Tensor) -> torch.Tensor:
    """Greedy residual codebooks 1..G-1 by RE-PREFILL (no KV cache).

    cp: HF Qwen3OmniMoeTalkerCodePredictorModelForConditionalGeneration
        cp.model(inputs_embeds=[B, L, H]).last_hidden_state, cp.lm_head[j], cp.model.codec_embedding[j]
    last_talker_hidden: [B, H]  talker's last-layer hidden state at the step that sampled CB0
    cb0_embed:          [B, H]  talker codec embedding of the sampled CB0
    returns codes [B, G-1]
    """
    G = cp.config.num_code_groups  # total codebooks per frame (including CB0)
    # Initial sequence length 2: position 0 = talker hidden state, position 1 = CB0's embedding
    seq = torch.stack([last_talker_hidden, cb0_embed], dim=1)  # [B, 2, H]
    codes = []
    for j in range(G - 1):  # predicts codebook j+1
        # Each step feeds the whole sequence again (no cache) and takes only the last position's hidden state; length 2
        # + j
        h = cp.model(inputs_embeds=seq).last_hidden_state[:, -1]  # [B, H]
        # Pitfall: step j uses head j (lm_head[j] predicts CB{j+1}); each residual codebook has its own head and
        # embedding
        tok = cp.lm_head[j](h).argmax(-1)  # [B] greedy
        codes.append(tok)
        # Embed the just-predicted code with codec_embedding[j] (same codebook as lm_head[j]); sequence length +1
        seq = torch.cat([seq, cp.model.codec_embedding[j](tok)[:, None]], dim=1)
    # One [B] per step, stacked as columns into [B, G-1] (column 0 is CB1)
    return torch.stack(codes, dim=1)


# KV version: the first step prefills length 2, then each step feeds just 1 new position; cache length goes 2 -> 3 ->
# ...,
# and HF infers the new position from the cache length, so it's mathematically equivalent to the re-prefill version.
@torch.no_grad()
def predict_residual_codes_kv(cp, last_talker_hidden: torch.Tensor, cb0_embed: torch.Tensor) -> torch.Tensor:
    """Same result with an incremental KV cache: prefill 2 positions, then 1 position per step."""
    from transformers import DynamicCache

    G = cp.config.num_code_groups
    # New cache per call: each frame's residual prediction is an independent short sequence; never reuse the previous
    # frame's cache
    cache = DynamicCache()
    x = torch.stack([last_talker_hidden, cb0_embed], dim=1)  # first step [B, 2, H]
    codes = []
    for j in range(G - 1):
        # HF appends this x's K/V into the cache in place inside forward
        h = cp.model(inputs_embeds=x, past_key_values=cache, use_cache=True).last_hidden_state[:, -1]
        tok = cp.lm_head[j](h).argmax(-1)
        codes.append(tok)
        # Pitfall: next step feeds only the "new token" embedding [B, 1, H]; don't re-feed the whole sequence (it writes
        # the cache twice)
        x = cp.model.codec_embedding[j](tok)[:, None]
    return torch.stack(codes, dim=1)


# ------------------------------------------------------------------------
# next_talker_input: the talker's input embedding for the next step =
#   sum of this frame's embeddings for all G codebooks (CB0 via the talker's codec embedding, CB1.. via the code
#   predictor's)
#   + the text-side vector for the current step, text_step.
# ------------------------------------------------------------------------
@torch.no_grad()
def next_talker_input(cp, cb0_embed: torch.Tensor, residual_codes: torch.Tensor, text_step: torch.Tensor) -> torch.Tensor:
    """Talker input of the NEXT step: sum of all G codec embeddings of this frame + the text row."""
    total = cb0_embed.clone()  # [B, H]; accumulate starting from CB0's embedding, without modifying the caller's tensor
    # residual_codes: [B, G-1]; column j (CB{j+1}) is embedded with codec_embedding[j]
    for j in range(residual_codes.shape[1]):
        total = total + cp.model.codec_embedding[j](residual_codes[:, j])
    return total + text_step  # [B, H]
```

C-6 · solutions/m06_streaming_causal_conv.py

```
import torch
import torch.nn.functional as F


# ------------------------------------------------------------------------
# StreamingCausalConv1d: streaming version of Code2Wav's causal Conv1d (stride 1).
# Offline = left-pad (K-1)*d zeros, then a valid conv. The streaming version replaces "left zero-padding" with a state:
# always keep the most recent context = (K-1)*d past inputs; for each chunk x, concat [state | x] and run a valid conv;
# the output length is exactly the chunk length t, and all chunk outputs concatenated match offline element-wise.
# ------------------------------------------------------------------------
class StreamingCausalConv1d:
```


© 2026 kzhou003

```
    """Chunk-by-chunk version of Code2Wav's causal Conv1d (stride 1, any dilation)."""

    def __init__(self, weight: torch.Tensor, bias: torch.Tensor | None, dilation: int = 1) -> None:
        # weight: [C_out, C_in, K]
        self.weight, self.bias, self.dilation = weight, bias, dilation
        # Receptive-field span (K-1)*d: output position t needs inputs t-(K-1)*d .. t
        self.context = (weight.shape[-1] - 1) * dilation  # inputs of the past this conv still needs
        self.state: torch.Tensor | None = None  # [B, C_in, context]; B / dtype / device known only at the first chunk

    @classmethod
    def from_hf(cls, m) -> "StreamingCausalConv1d":  # HF Qwen3OmniMoeCausalConvNet
        assert m.stride == 1  # the streaming-state derivation only holds for stride 1
        return cls(m.conv.weight.detach(), None if m.conv.bias is None else m.conv.bias.detach(), m.dilation)

    def step(self, x: torch.Tensor) -> torch.Tensor:  # [B, C_in, t] -> [B, C_out, t]
        if self.state is None:
            # The "history" before the first chunk is the offline version's left zero-padding
            self.state = x.new_zeros(x.shape[0], x.shape[1], self.context)  # causal zero padding
        buf = torch.cat([self.state, x], dim=-1)  # [B, C_in, context + t]
        # New state = the last `context` samples of buf.
        # Pitfall: take them from the end of buf (old state + new chunk), not from the end of x -- when t < context, x
        # is too short
        #       and the state must still keep some old samples (possibly including padded zeros).
        # When context == 0 (K=1, no history needed), state is explicitly set to an empty length-0 tensor.
        self.state = buf[..., buf.shape[-1] - self.context:] if self.context else buf[..., :0]
        # valid conv (no padding): output length (context + t) - context = t
        return F.conv1d(buf, self.weight, self.bias, dilation=self.dilation)


# ---------------------------------------------------------------------------
# StreamingCausalTransConv1d: streaming version (overlap-add) of the causal ConvTranspose1d used for Code2Wav
# upsampling.
# Input frame i writes output range [i*s, i*s + K); adjacent frames' write ranges overlap by K-s samples.
# Keep an accumulation buffer acc (samples from global position base onward that are not yet "final"):
#   - step: add this chunk's transposed-conv result into acc at the right offset; after n frames,
#     positions < n*s will never be written again and can be emitted;
#   - the offline module trims K-s samples at the start and K-s at the end; streaming does the global left trim in _emit
#   and the right trim in flush.
# ---------------------------------------------------------------------------
class StreamingCausalTransConv1d:
    """Chunk-by-chunk version of Code2Wav's causal ConvTranspose1d (upsampling by `stride`).

    Input frame i writes output positions [i*s, i*s + K). Position p is final once every frame
    that touches it has arrived, i.e. p < n*s after n frames. The full (offline) module trims
    K - s samples on both ends.
    """

    def __init__(self, weight: torch.Tensor, bias: torch.Tensor | None, stride: int) -> None:
        self.weight, self.bias, self.stride = weight, bias, stride  # weight: [C_in, C_out, K]
        self.K = weight.shape[-1]
        self.trim = self.K - stride  # samples trimmed at each end in the offline version
        self.acc: torch.Tensor | None = None  # pending (not final) samples, starting at self.base
        self.base = 0  # global output position of acc[..., 0] (= total samples emitted so far, including trimmed ones)
        self.frames = 0  # total input frames received

    @classmethod
    def from_hf(cls, m) -> "StreamingCausalTransConv1d":  # HF Qwen3OmniMoeCausalTransConvNet
        c = m.conv
        return cls(c.weight.detach(), None if c.bias is None else c.bias.detach(), c.stride[0])

    # _emit: slice the finalized samples at global positions [base, upto) out of acc, add bias, apply the global left
    # trim, return.
    def _emit(self, upto: int) -> torch.Tensor:
        assert self.acc is not None
        n = upto - self.base  # number of samples finalized this time
        out, self.acc = self.acc[..., :n], self.acc[..., n:]  # acc keeps only the not-yet-final tail
        start = self.base  # global position of out[..., 0]
        self.base = upto
        # Pitfall: bias may be added only once per position, at finalization. Passing bias to conv_transpose1d
        # would add it once per overlapping frame in the overlap region, diverging from offline.
        if self.bias is not None:  # bias once per output position (never inside the overlap-add)
            out = out + self.bias[None, :, None]
        # Global left trim: samples at global positions < trim belong to the start the offline version trims;
        # they may span several _emit calls, so compute how much is left to trim from start (skip is always 0 once start
        # >= trim)
        skip = max(0, self.trim - start)  # global left trim
        return out[..., skip:]

    def step(self, x: torch.Tensor) -> torch.Tensor:
        # x: [B, C_in, t], the t newly arrived frames
        s = self.stride
        # Transposed conv on this chunk alone (no bias), length (t-1)*s + K
        y = F.conv_transpose1d(x, self.weight, None, stride=s)  # covers [frames*s, frames*s + len)
        start = self.frames * s  # global start written by this chunk's first frame
        end = start + y.shape[-1]
        if self.acc is None:
            self.acc = y.new_zeros(y.shape[0], y.shape[1], 0)
        # Invariant: base == frames*s at the end of every step, so here start == base,
        # and acc holds exactly the K-s overlap tail left by the previous chunk (at [base, base+K-s))
```


© 2026 kzhou003

```
        need = end - self.base
        if self.acc.shape[-1] < need:
            # Right-pad with zeros so it can hold all of this chunk's writes
            self.acc = F.pad(self.acc, (0, need - self.acc.shape[-1]))
        # overlap-add: the part overlapping the previous chunk's tail is summed
        self.acc[..., start - self.base:end - self.base] += y
        self.frames += x.shape[-1]
        # After `frames` frames, positions < frames*s won't be written by later frames, so they can be finalized and
        # emitted
        return self._emit(self.frames * s)

    def flush(self) -> torch.Tensor:
        # End of stream: finalize all remaining pending samples, then trim the offline version's trailing `trim` samples
        assert self.acc is not None
        total = (self.frames - 1) * self.stride + self.K  # total length of the untrimmed offline output
        out = self._emit(total)
        # Pitfall: the right trim can only happen in flush -- mid-stream you don't know where the end is.
        # Normally exactly K-s = trim samples remain pending and are all trimmed, so flush returns length 0; with trim
        # == 0 it returns as is
        return out[..., :out.shape[-1] - self.trim] if self.trim else out
```

C-7 · solutions/m07_stage_weight_loading.py

```
import re
from dataclasses import dataclass

import torch

# One Qwen3-Omni checkpoint holds the weights of all three stages, distinguished by the top-level key prefix.
# Pitfall: the prefix includes the dot, "thinker.", otherwise look-alike prefixes such as "thinkers.xxx" also match.
STAGE_PREFIX = {"thinker": "thinker.", "talker": "talker.", "code2wav": "code2wav."}


class LoadError(Exception):
    pass


# ---------------------------------------------------------------------------
# shards_for_stage: in a multi-stage deployment each stage process loads only its own weights,
# so first find the shards to open from index.json's weight_map (param key -> shard file name).
# ---------------------------------------------------------------------------
def shards_for_stage(weight_map: dict[str, str], stage: str) -> list[str]:
    """model.safetensors.index.json 'weight_map' -> the shard files this stage must open (sorted)."""
    prefix = STAGE_PREFIX[stage]
    # set dedups (a shard may contain keys of several stages); sorted makes the result deterministic
    return sorted({f for k, f in weight_map.items() if k.startswith(prefix)})


# ---------------------------------------------------------------------------
# select_stage: filter a submodule's weights (e.g. "thinker.model.") out of the full state dict and strip the prefix,
# giving the submodule-local names ("layers.0.xxx", "norm.weight"...).
# ---------------------------------------------------------------------------
def select_stage(state: dict[str, torch.Tensor], stage: str, strip: str) -> dict[str, torch.Tensor]:
    """Keep only keys under `strip` (e.g. "thinker.model.") and drop that prefix."""
    # strip must sit under this stage, guarding against the caller using another stage's prefix
    assert strip.startswith(STAGE_PREFIX[stage])
    # Keys not under strip (e.g. thinker.lm_head.weight) are dropped; the caller fetches them separately if needed
    return {k[len(strip):]: v for k, v in state.items() if k.startswith(strip)}


# PackedLayer: the "packed" weight layout inference engines commonly use -- q/k/v concatenated into one qkv matrix, each
# expert's gate/up into w13,
# so one matmul computes several projections.
@dataclass
class PackedLayer:
    input_layernorm: torch.Tensor
    qkv: torch.Tensor                # [(H + 2*KVH) * D, hidden]  rows: q | k | v
    q_norm: torch.Tensor
    k_norm: torch.Tensor
    o_proj: torch.Tensor
    post_attention_layernorm: torch.Tensor
    router: torch.Tensor             # [E, hidden]
    w13: torch.Tensor                # [E, 2*I, hidden]  rows: gate | up
    w2: torch.Tensor                 # [E, hidden, I]


# Matches in-layer expert weight names, e.g. "mlp.experts.12.up_proj.weight" -> group(1)="12", group(2)="up_proj"
_EXPERT = re.compile(r"^mlp\.experts\.(\d+)\.(gate_proj|up_proj|down_proj)\.weight$")


# ---------------------------------------------------------------------------
# pack_layer: pack layer i's loose HF weights into a PackedLayer, with strict validation:
# missing keys, non-contiguous expert ids, an expert lacking gate/up/down, or unknown extra keys all raise LoadError.
# `used` records consumed keys; at the end "all keys - used" gives the unexpected ones.
# ---------------------------------------------------------------------------
def pack_layer(sd: dict[str, torch.Tensor], i: int) -> PackedLayer:
```


© 2026 kzhou003

```
    p = f"layers.{i}."
    # Pitfall: the prefix includes the trailing dot, otherwise besides "layers.1." it would also match "layers.10.",
    # "layers.11." ...
    sub = {k[len(p):]: v for k, v in sd.items() if k.startswith(p)}  # in-layer relative keys
    used: set[str] = set()

    # take: fetch a required weight; if absent report missing, otherwise record it in used
    def take(name: str) -> torch.Tensor:
        if name not in sub:
            raise LoadError(f"missing {p}{name}")
        used.add(name)
        return sub[name]

    # Collect expert weights: experts[e]["gate_proj" | "up_proj" | "down_proj"] = tensor
    experts: dict[int, dict[str, torch.Tensor]] = {}
    for k, v in sub.items():
        mm = _EXPERT.match(k)
        if mm:
            experts.setdefault(int(mm.group(1)), {})[mm.group(2)] = v
            used.add(k)
    # Expert ids must be exactly 0..E-1 (non-empty, no gaps).
    # Pitfall: sorted(experts) sorts integers; sorting string keys would give 0, 1, 10, 11, 2 ...
    if not experts or sorted(experts) != list(range(len(experts))):
        raise LoadError(f"layer {i}: expert ids must be 0..E-1, got {sorted(experts)}")
    for e, parts in experts.items():
        if set(parts) != {"gate_proj", "up_proj", "down_proj"}:
            raise LoadError(f"layer {i} expert {e}: incomplete {sorted(parts)}")

    layer = PackedLayer(
        input_layernorm=take("input_layernorm.weight"),
        # Concat along dim 0 (output dim / rows): q | k | v; can be split back later by [H*D, KVH*D, KVH*D]
        qkv=torch.cat([take("self_attn.q_proj.weight"), take("self_attn.k_proj.weight"),
                       take("self_attn.v_proj.weight")], dim=0),
        q_norm=take("self_attn.q_norm.weight"),
        k_norm=take("self_attn.k_norm.weight"),
        o_proj=take("self_attn.o_proj.weight"),
        post_attention_layernorm=take("post_attention_layernorm.weight"),
        router=take("mlp.gate.weight"),  # in HF the MoE router is called mlp.gate
        # For each expert, concat gate [I, hidden] and up [I, hidden] along rows into [2I, hidden], then stack in
        # expert-id order
        w13=torch.stack([torch.cat([experts[e]["gate_proj"], experts[e]["up_proj"]], 0) for e in range(len(experts))]),
        w2=torch.stack([experts[e]["down_proj"] for e in range(len(experts))]),  # [E, hidden, I]
    )
    # After all takes, check for extra keys (e.g. shared_expert.*, misspelled names)
    unexpected = set(sub) - used
    if unexpected:
        raise LoadError(f"layer {i}: unexpected {sorted(unexpected)}")
    return layer


# ---------------------------------------------------------------------------
# load_thinker_text: load the thinker text trunk: pack layer by layer, and check that outside the layers there are only
# the
# two global weights embed_tokens / norm. Returns ({"embed_tokens", "norm"}, [PackedLayer] * num_layers). lm_head is not
# under thinker.model.; the caller fetches it separately.
# ---------------------------------------------------------------------------
def load_thinker_text(state: dict[str, torch.Tensor], num_layers: int) -> tuple[dict, list[PackedLayer]]:
    sd = select_stage(state, "thinker", "thinker.model.")
    layers = [pack_layer(sd, i) for i in range(num_layers)]
    # Non-layers.* keys may only be these two global weights;
    # note: extra layers with index >= num_layers are not caught here (they start with "layers.")
    extra = {k for k in sd if not k.startswith("layers.")} - {"embed_tokens.weight", "norm.weight"}
    if extra:
        raise LoadError(f"unexpected {sorted(extra)}")
    for k in ("embed_tokens.weight", "norm.weight"):
        if k not in sd:
            raise LoadError(f"missing {k}")
    return {"embed_tokens": sd["embed_tokens.weight"], "norm": sd["norm.weight"]}, layers
```

C-8 · solutions/m08_tiny_thinker_e2e.py

```
from dataclasses import dataclass

import torch

from .m01_rmsnorm_mrope import mrope_cos_sin, rms_norm, rope_inv_freq
from .m02_thinker_attention import AttnWeights, KVCache, attention_forward
from .m03_sparse_moe import MoeWeights, moe_forward
from .m07_stage_weight_loading import load_thinker_text


# Dims: model architecture hyperparameters (a subset of the config).
@dataclass
class Dims:
    num_layers: int
    num_heads: int
```


© 2026 kzhou003

```
    num_kv_heads: int
    head_dim: int
    top_k: int
    norm_topk_prob: bool
    rope_theta: float
    mrope_section: list[int]
    eps: float = 1e-6


# ----------------------------------------------------------------------------
# TinyThinker: combines m01 (RMSNorm / M-RoPE), m02 (attention + KV cache), m03 (MoE), m07 (weight loading)
# into an end-to-end text-only thinker: embed -> N × [pre-norm attention + residual, pre-norm MoE + residual]
# -> final norm -> lm_head, with KV-cached greedy decoding. Results should match HF's full-recompute-per-step greedy.
# ----------------------------------------------------------------------------
class TinyThinker:
    """Text-only Qwen3-Omni thinker built from YOUR m01/m02/m03/m07 pieces."""

    def __init__(self, state: dict[str, torch.Tensor], dims: Dims) -> None:
        self.d = dims
        g, layers = load_thinker_text(state, dims.num_layers)
        self.embed, self.norm = g["embed_tokens"], g["norm"]  # embed: [V, hidden]
        # lm_head lives under "thinker.lm_head.", not "thinker.model.", so load_thinker_text doesn't return it
        self.lm_head = state["thinker.lm_head.weight"]  # [V, hidden]
        D, H, KVH = dims.head_dim, dims.num_heads, dims.num_kv_heads
        self.layers = []
        for L in layers:
            # Unpack m07's packed layout into the form m02 / m03 need: split qkv rows as q | k | v
            q, k, v = L.qkv.split([H * D, KVH * D, KVH * D], dim=0)
            I2 = L.w13.shape[1]  # = 2 * I
            attn = AttnWeights(q, k, v, L.o_proj, L.q_norm, L.k_norm, H, KVH, D, dims.eps)
            # w13: [E, 2I, hidden]; the first I rows are gate, the last I rows are up (matching m07's concat order)
            moe = MoeWeights(L.router, L.w13[:, : I2 // 2], L.w13[:, I2 // 2:], L.w2, dims.top_k, dims.norm_topk_prob)
            self.layers.append((L.input_layernorm, attn, L.post_attention_layernorm, moe))
        # The inverse-frequency table depends only on head_dim and theta; compute it once at init
        self.inv_freq = rope_inv_freq(D, dims.rope_theta)

    def forward(self, ids: torch.Tensor, start: int, caches: list[KVCache]) -> torch.Tensor:
        """ids: [T] new tokens at absolute positions start..start+T-1 -> logits [T, V]."""
        T = ids.shape[0]
        # Pure text: the t/h/w components are equal -> [3, T] (M-RoPE reduces to 1D RoPE, but still takes the
        # interleaved path to match HF)
        # Invariant: when called, start must equal every layer's current cache length
        pos = torch.arange(start, start + T).expand(3, T)
        cos, sin = mrope_cos_sin(self.inv_freq, pos, self.d.mrope_section, interleaved=True)  # [T, D]
        h = self.embed[ids]  # [T, hidden], embedding lookup
        # strict=True: number of layers and caches must match; one independent KVCache per layer
        for (ln1, attn, ln2, moe), cache in zip(self.layers, caches, strict=True):
            # pre-norm: the norm applies only to the sublayer input; the residual adds the un-normed h
            h = h + attention_forward(attn, rms_norm(h, ln1, self.d.eps), cos, sin, cache)
            h = h + moe_forward(rms_norm(h, ln2, self.d.eps), moe)
        return rms_norm(h, self.norm, self.d.eps) @ self.lm_head.T  # [T, V]

    @torch.no_grad()
    def greedy(self, prompt: list[int], max_new: int) -> list[int]:
        caches = [KVCache() for _ in self.layers]
        # prefill: the whole prompt starts at position 0; the last row of logits directly gives the first new token
        logits = self.forward(torch.tensor(prompt), 0, caches)
        out = [int(logits[-1].argmax())]
        # max_new forwards in total: prefill yields the 1st, then each decode yields 1; the last token needs no further
        # forward
        while len(out) < max_new:
            # decode: feed only the 1 token generated in the previous step; its absolute position is
            # len(prompt) + len(out) - 1 (== current cache length).
            # Pitfall: writing len(prompt) + len(out) shifts all RoPE positions by 1, and the output soon diverges from
            # HF
            logits = self.forward(torch.tensor(out[-1:]), len(prompt) + len(out) - 1, caches)
            out.append(int(logits[-1].argmax()))
        return out
```

C-9 · solutions/p02_orchestrator.py

```
import asyncio
from collections.abc import Callable
from dataclasses import dataclass, field
from typing import Any, Callable, Protocol


# ---------------------------------------------------------------- given types
# StageOutput: stage -> orchestrator message (every stage produces these, including non-final stages)
@dataclass
class StageOutput:
    request_id: str
    data: Any
    finished: bool  # only means "this stage is done with this request", not that the whole request is done
```


© 2026 kzhou003

```
# OrchOutput: orchestrator -> user message (only from stages in final_stage_ids)
@dataclass
class OrchOutput:
    request_id: str
    stage_id: int | None
    data: Any
    finished: bool  # the whole request's terminal flag: exactly one True per request, and it is the last one
    error: str | None = None


class StageClient(Protocol):
    async def submit(self, request_id: str, inputs: Any) -> None: ...
    async def get_output(self) -> list[StageOutput]: ...
    async def abort(self, request_ids: list[str]) -> None: ...


# ---------------------------------------------------------- implementation
# Routing state per in-flight request; read/written only by the orchestrator's internal tasks (same event loop, no
# lock).
@dataclass
class ReqState:
    request_id: str
    prompt: Any
    submitted: set[int] = field(default_factory=set)  # stages already submitted
    finished: set[int] = field(default_factory=set)  # stages that reported finished=True
    # All outputs accumulated per stage; when stage s finishes, the whole list goes to forward_fn to build s+1's input
    stage_data: dict[int, list] = field(default_factory=dict)
    pending_final: OrchOutput | None = None  # unused in P02; P03 uses it to hold back the terminal message


# Linear multi-stage routing (e.g. S0 -> S1 -> S2, final_stage_ids={0,2}).
# Task structure (all on the same event loop):
#   N _reader tasks (one per stage) --(sid, outs)--> _ready_q --> 1 _dispatcher task
#   add_request() --> _req_q --> 1 _request_handler task
#   _dispatcher --> _out_q --> next_output() (user side)
# Single writer: all StageOutputs funnel into one _ready_q and are processed serially by the only dispatcher,
# so for a given request the finished check / forwarding / cleanup can't be interleaved by another output.
class Orchestrator:
    def __init__(
        self,
        stages: list[StageClient],
        final_stage_ids: set[int],
        forward_fn: Callable[[int, Any, list], Any],
    ) -> None:
        self.stages = stages
        self.final_stage_ids = frozenset(final_stage_ids)
        # The last stage that needs to run: stages after it are meaningless to the user and are never submitted
        self.last_stage = max(final_stage_ids)
        self.forward_fn: Callable[[int, Any, list[Any]], Any] = forward_fn
        self.request_states: dict[str, ReqState] = {}
        self.dropped_outputs = 0
        # Three unbounded queues: put never blocks, so readers / the dispatcher can't deadlock each other on a slow
        # downstream
        self._req_q: asyncio.Queue = asyncio.Queue()
        self._ready_q: asyncio.Queue = asyncio.Queue()
        self._out_q: asyncio.Queue = asyncio.Queue()
        self._tasks: list[asyncio.Task] = []

    # ---------------------------------------------------------- public API
    async def start(self) -> None:
        for sid, client in enumerate(self.stages):
            # One reader per stage: a blocking get_output on one stage doesn't hold up the others
            self._tasks.append(asyncio.create_task(coro=self._reader(sid, client)))
        self._tasks.append(asyncio.create_task(self._request_handler()))
        # Exactly one dispatcher -- this is the "single writer"
        self._tasks.append(asyncio.create_task(self._dispatcher()))

    async def shutdown(self) -> None:
        for t in self._tasks:
            t.cancel()
        # cancel only requests cancellation; gather waits for them to actually exit. return_exceptions=True swallows
        # CancelledError,
        # otherwise the first CancelledError makes shutdown itself raise, leaving the other tasks unawaited
        await asyncio.gather(*self._tasks, return_exceptions=True)

    # Called from the user side: only posts a message; the real registration/submission happens in the request handler
    # task
    async def add_request(self, request_id: str, prompt: Any) -> None:
        await self._req_q.put(item=("add", request_id, prompt))

    async def next_output(self, timeout: float | None = None) -> OrchOutput:
        return await asyncio.wait_for(self._out_q.get(), timeout)

    # ---------------------------------------------------------- internals
    # One reader task per stage: only moves data, never touches request_states (avoids multiple writers)
    async def _reader(self, sid: int, client: StageClient) -> None:
        while True:
            outs = await client.get_output()
            # Enqueue together with sid: the dispatcher needs to know which stage the output came from
```


© 2026 kzhou003

```
            await self._ready_q.put((sid, outs))

    # Runs only in the request handler task
    async def _request_handler(self) -> None:
        while True:
            kind, rid, payload = await self._req_q.get()
            if kind == "add":
                await self._handle_add(rid, prompt=payload)

    async def _handle_add(self, rid: str, prompt: Any) -> None:
        st: ReqState = ReqState(rid, prompt)
        # Pitfall: register BEFORE the first await. await submit yields control to the dispatcher;
        # a fast stage may emit output before submit returns, and when reader -> _ready_q -> dispatcher
        # processes it without finding the ReqState, the output is dropped as an unknown request (dropped_outputs++),
        # and the user never gets finished=True -> hang.
        self.request_states[rid] = st  # register BEFORE the first await
        # Likewise, mark submitted before the await so the dispatcher always sees a consistent state
        st.submitted.add(0)
        await self.stages[0].submit(rid, prompt)

    # Runs only in the dispatcher task (single writer): processes one by one in arrival order; a request's outputs are
    # never handled concurrently
    async def _dispatcher(self) -> None:
        while True:
            sid, outs = await self._ready_q.get()
            for o in outs:
                # await one at a time: preserves order within a batch (e.g. "hi" is always handled before
                # "there"(finished))
                await self._handle_output(sid, o)

    # Runs only in the dispatcher task.
    # Handle one StageOutput: record -> (if final stage) send to user -> (if the stage finished) forward downstream
    async def _handle_output(self, sid: int, o: StageOutput) -> None:
        st = self.request_states.get(o.request_id)
        if st is None:
            # Request already cleaned up (terminal message sent) or never registered: drop and count.
            # Pitfall: never create state here -- that would "resurrect" a finished request, causing duplicate terminal
            # messages or leaked state
            self.dropped_outputs += 1
            return
        # Accumulate data first: when the stage finishes, forward_fn needs all its data (including this finished last
        # chunk)
        st.stage_data.setdefault(sid, []).append(o.data)
        if o.finished:
            st.finished.add(sid)
        if sid in self.final_stage_ids:
            # Pitfall: OrchOutput.finished means "all final stages are done", not o.finished.
            # E.g. final={0,2}: when S0's "there"(finished=True) arrives, finished={0}, and {0,2} isn't all done
            # -> the user gets finished=False; only S2's last chunk is True.
            done = self.final_stage_ids <= st.finished
            await self._out_q.put(OrchOutput(o.request_id, sid, o.data, finished=done))
            if done:
                # Terminal message sent: remove state immediately; any later output for this request takes the dropped
                # branch above,
                # guaranteeing "exactly one finished=True"
                self._cleanup(o.request_id)
                # Skip the forwarding logic: the request is over; submitting downstream now would create orphan work
                return
        # Non-final stage (e.g. S1) outputs are not sent to the user; they only trigger downstream.
        # Forward only when the stage finished and there is still a stage to run after it; stages after last_stage are
        # never submitted
        if o.finished and sid < self.last_stage:
            nxt = sid + 1
            # Give forward_fn ALL of stage sid's data (not just the last o.data)
            inputs = self.forward_fn(nxt, st.prompt, st.stage_data[sid])
            # Mark submitted before the await, same reason as _handle_add
            st.submitted.add(nxt)
            # The dispatcher is suspended during this await; other outputs wait in _ready_q and are not interleaved
            await self.stages[nxt].submit(o.request_id, inputs)

    # Remove request state; pop(..., None) makes repeated calls harmless
    def _cleanup(self, rid: str) -> None:
        self.request_states.pop(rid, None)
```

C-10 · solutions/p03_async_chunk_orchestrator.py

```
from collections.abc import Callable
from typing import Any

from .p02_orchestrator import Orchestrator, OrchOutput, ReqState, StageClient, StageOutput

__all__ = ["AsyncChunkOrchestrator", "OrchOutput", "StageOutput"]


# async_chunk mode: stages pass data directly over a side channel (connector); the orchestrator no longer forwards.
# It is responsible for only two things:
```


© 2026 kzhou003

```
#   1. on add, prewarm all downstream stages at once (so they are ready early, waiting for the connector to feed data);
#   2. guard the terminal message: finished=True may only be sent after "every submitted stage has finished".
# Task structure, the single-writer dispatcher, readers etc. are all reused from P02; only _handle_add / _handle_output
# are overridden.
class AsyncChunkOrchestrator(Orchestrator):
    """Stages exchange data on a side channel; the orchestrator only prewarms
    downstream stages and gates the terminal message."""

    def __init__(
        self,
        stages: list[StageClient],
        final_stage_ids: set[int],
        prewarm_fn: Callable[[int, Any], Any],
    ) -> None:
        # forward_fn is unused (the orchestrator no longer forwards), so pass a placeholder
        super().__init__(stages, final_stage_ids, forward_fn=lambda *_: None)
        self.prewarm_fn = prewarm_fn

    # Runs only in the request handler task
    async def _handle_add(self, rid: str, prompt: Any) -> None:
        st = ReqState(rid, prompt)
        # Same as P02: register before any await, otherwise early outputs get dropped -> hang
        self.request_states[rid] = st
        # Pitfall: mark all of 0..last_stage as submitted in one go, and before any await.
        # submitted is the baseline for everything_done; if you add while submitting, the dispatcher may see (mid-way,
        # during some await submit) an incomplete submitted set and wrongly conclude "all stages are done",
        # releasing the terminal message too early (later completion reports from the remaining stages can only be
        # dropped).
        st.submitted.update(range(self.last_stage + 1))  # mark before any await
        await self.stages[0].submit(rid, prompt)
        # Downstream stages receive a placeholder input (prewarm); the real data arrives later via the connector
        for sid in range(1, self.last_stage + 1):
            await self.stages[sid].submit(rid, self.prewarm_fn(sid, prompt))

    # Runs only in the dispatcher task (single writer).
    # Note: no stage_data accumulation and no forwarding here -- data goes through the connector.
    async def _handle_output(self, sid: int, o: StageOutput) -> None:
        st = self.request_states.get(o.request_id)
        if st is None:
            # Already cleaned-up request: drop it, never resurrect state (or a second terminal message might be sent)
            self.dropped_outputs += 1
            return
        if o.finished:
            st.finished.add(sid)
        # Two different notions of "done":
        #   all_final       = all final stages done -> determines the value of OrchOutput.finished
        #   everything_done = all submitted stages (including non-final ones, e.g. S1 talker) done -> decides whether
        #   the terminal message may go out
        everything_done = st.submitted <= st.finished
        if sid in self.final_stage_ids:
            all_final = self.final_stage_ids <= st.finished
            msg = OrchOutput(o.request_id, sid, o.data, finished=all_final)
            if all_final and not everything_done:
                # E.g. final={0,2}: S2's last chunk arrives first, but the non-final S1 hasn't reported completion yet.
                # Sending and cleaning up now would drop S1's completion report while S1 may still hold resources;
                # so hold back the terminal message and send it once the remaining stages finish.
                st.pending_final = msg  # hold the terminal until every stage finished
                return
            # Non-terminal messages go out directly; so does the terminal message when all_final and everything_done
            await self._out_q.put(msg)
            if all_final:
                # Terminal message sent: cleanup; all later outputs take the dropped branch -> exactly one finished=True
                self._cleanup(o.request_id)
            return
        # Reaching here means a non-final stage (e.g. S1): its outputs are never sent to the user.
        # If its completion makes everything_done true and a terminal message is being held, release it now.
        # Invariant: everything_done implies all_final; when the last final stage finished, if not yet everything_done
        # pending_final must have been set; if already everything_done, it was sent and cleaned up directly above.
        if everything_done and st.pending_final is not None:
            # What gets released is still that last final-stage output -> "the terminal message is always the last one"
            await self._out_q.put(st.pending_final)
            self._cleanup(o.request_id)
```

C-11 · solutions/p01_output_sink.py

```
import asyncio
import threading
from collections import deque


# OutputSink: many producer threads (e.g. each stage's output thread) -> one asyncio consumer.
# Two core difficulties:
#   1. Waking the event loop from another thread can only use loop.call_soon_threadsafe (asyncio.Event is not
#   thread-safe);
#   2. "coalesced wakeup": at most one call_soon_threadsafe per wake cycle,
#      while never losing a wakeup (lost wakeup) -- the consumer must not go to sleep while data is present.
```


© 2026 kzhou003

```
# Division of state:
#   _items          the data itself; deque append / popleft are atomic in CPython, so no lock
#   _wake_scheduled "wakeup in flight" flag; True means the consumer will drain again on its own, so put needn't wake it
#   _lock           protects only the _wake_scheduled + wakeups_scheduled pair (and the loop/event snapshot read)
class OutputSink:
    """Many producer threads -> one asyncio consumer, with coalesced wakeups."""

    def __init__(self) -> None:
        self._items: deque = deque()
        self._lock = threading.Lock()
        # Starts False: the first put always schedules a wakeup
        self._wake_scheduled = False
        self._loop: asyncio.AbstractEventLoop | None = None
        self._event: asyncio.Event | None = None
        self.wakeups_scheduled = 0
        # test hook: called right after a drain found nothing, before re-arming
        self.after_empty_drain = lambda: None

    # Call on the consumer's loop (defaults to the current running loop).
    # The Event must be created/used on this loop; afterwards only the loop thread may touch it directly.
    def bind_loop(self, loop: asyncio.AbstractEventLoop | None = None) -> None:
        self._loop = loop or asyncio.get_running_loop()
        self._event = asyncio.Event()

    # Called from any producer thread; non-blocking (the lock covers only a few assignments, held very briefly).
    def put(self, item) -> None:
        # Pitfall: append FIRST, then look at the flag. The consumer's invariant is "after clearing the flag, always
        # drain once more",
        # so as long as the data is enqueued first, it can't be missed whether the flag is True or False right now:
        #   - sees True: the consumer hasn't re-armed yet; its later drains (including the one after arming) will see
        #   item
        #   - sees False: this put is responsible for scheduling the wakeup
        # The other way round (check the flag, then append): see True and return -> consumer arms and drains again
        # before item is enqueued -> consumer sleeps -> item is stuck (lost wakeup / hang).
        self._items.append(item)  # append BEFORE looking at the flag
        with self._lock:
            if self._wake_scheduled:
                # Wakeup already in flight (or consumer busy, not yet re-armed): coalesce, don't schedule again
                return
            self._wake_scheduled = True
            self.wakeups_scheduled += 1
            # Snapshot inside the lock so a concurrent bind_loop can't give us a half-updated (loop, event)
            loop, event = self._loop, self._event
        # call_soon_threadsafe goes outside the lock: don't call into loop internals while holding it (shorter hold, no
        # potential deadlock)
        if loop is not None and event is not None:
            try:
                # Pitfall: never call event.set() directly on the producer thread -- asyncio.Event is not thread-safe
                # and won't wake a loop blocked in the selector. call_soon_threadsafe writes the self-pipe to wake the
                # loop,
                # then runs event.set on the loop thread.
                loop.call_soon_threadsafe(event.set)
            except RuntimeError:  # loop already closed
                # Loop already closed during shutdown: nobody will consume anymore, so silently drop the wakeup
                pass
        # If bind_loop hasn't happened yet: the flag is True but no real wakeup was sent; fine, get_batch's first drain
        # gets the data

    # Only called in the consumer (loop thread); single consumer, so popleft after "while self._items" can't fail
    def _drain(self, max_items: int) -> list:
        out = []
        while self._items and len(out) < max_items:
            out.append(self._items.popleft())
        return out

    # Runs only on the bound loop (single consumer). Returns 1..max_items items, or [] on timeout.
    async def get_batch(self, max_items: int = 256, timeout: float | None = None) -> list:
        if self._loop is None:
            self.bind_loop()
        loop, event = self._loop, self._event
        assert loop is not None and event is not None
        # Absolute deadline: after a spurious wakeup (stale set) the loop runs again without resetting the remaining
        # time
        deadline = None if timeout is None else loop.time() + timeout
        while True:
            # Fast path: data present -> return immediately, don't arm, don't clear the flag.
            # _wake_scheduled stays True, so producers send no extra wakeups while the consumer is busy;
            # the next get_batch always drains first, so this data is not lost.
            out = self._drain(max_items)
            if out:
                return out
            self.after_empty_drain()
            # arm: the next put() must schedule a wakeup
            # clear first: drop the set left over from the previous cycle, otherwise the wait below returns immediately
            # (a wasted spin).
            # There is no await between clear and clearing the flag, so the loop's event.set callback can't slip in;
            # this order is safe.
            event.clear()
            with self._lock:
```


© 2026 kzhou003

```
                # Clearing the flag = re-arming: the first put from now on schedules a wakeup
                self._wake_scheduled = False
            # drain again: closes the "put between drain and arm" race
            # Pitfall: this second drain is mandatory. Window: first drain is empty -> producer appends, sees the flag
            # still True
            # and returns (no wakeup scheduled) -> we clear the flag -> if we wait now, we sleep forever with data
            # queued (hang).
            # The test injects a put into exactly this window via after_empty_drain.
            out = self._drain(max_items)
            if out:
                # On return the flag is False (armed), so the next put schedules one more wakeup -- just one extra
                # wakeup,
                # correctness unaffected.
                return out
            remaining: float | None = None if deadline is None else deadline - loop.time()
            if remaining is not None and remaining <= 0:
                return []
            try:
                # While awaiting, a producer's put wakes us via call_soon_threadsafe(event.set)
                await asyncio.wait_for(event.wait(), remaining)
            except asyncio.TimeoutError:
                # Data may arrive right at timeout (set callback not yet run), so drain one last time instead of
                # returning [] directly
                return self._drain(max_items)
            # After wakeup, go back to the top and drain; if it was spurious (no data), re-arm and sleep again
```

C-12 · solutions/p04_stage_pool_abort.py

```
from dataclasses import dataclass
from typing import Any, Protocol


# ------------------------------------------------------------------ given types
class NoLiveReplica(Exception):
    pass


class ReplicaClient(Protocol):
    async def abort(self, request_ids: list[str]) -> None: ...


class PartialProcessor(Protocol):
    """Detokenizer-like state holder living on the orchestrator side."""

    def collect_partial(self, request_id: str) -> Any: ...  # read-only, no state change
    def commit_abort(self, request_id: str) -> None: ...    # drop the request's state


@dataclass
class OrchOutput:
    request_id: str
    data: Any
    finished: bool
    error: str | None = None


# ------------------------------------------------------------------ implementation
# A multi-replica pool (replica pool) for one stage. All methods are called on the orchestrator's event loop:
# sync methods (pick / evict / release) never interleave; the only yield point is the await inside abort.
# State:
#   _alive     whether each replica is alive
#   _bindings  sticky request_id -> replica_id binding (the request's KV cache / generation state lives on that replica)
#   _rr        round-robin cursor
class StagePool:
    def __init__(self, stage_id: int, replicas: list[ReplicaClient]) -> None:
        self.stage_id = stage_id
        self.replicas = replicas
        self._alive = [True] * len(replicas)
        self._bindings: dict[str, int] = {}
        self._rr = 0

    def live_replicas(self) -> list[int]:
        return [i for i, ok in enumerate(self._alive) if ok]

    # Pick a replica for a request: bound and alive -> stick to it; otherwise round-robin over live replicas and bind.
    def pick(self, request_id: str) -> int:
        bound = self._bindings.get(request_id)
        # Pitfall: stickiness. Later calls for the same request must go to the same replica, otherwise its state isn't
        # there
        if bound is not None and self._alive[bound]:
            return bound
        # The bound replica is dead (normally evict already removed the binding; this is a fallback): fall through to
        # re-pick
        live = self.live_replicas()
        if not live:
            raise NoLiveReplica(f"stage {self.stage_id} has no live replica")
```


© 2026 kzhou003

```
        # Take the modulus over the "live list", not over all replicas, so a dead replica is never chosen
        rid = live[self._rr % len(live)]
        self._rr += 1
        self._bindings[request_id] = rid
        return rid

    def bound(self, request_id: str) -> int | None:
        return self._bindings.get(request_id)

    # Release the binding when a request finishes normally; not releasing makes _bindings grow forever (leak)
    def release(self, request_id: str) -> None:
        self._bindings.pop(request_id, None)

    # A replica died: first mark it dead (pick won't choose it again), then remove and return the requests bound to it,
    # so the caller can send them terminal errors. Requests bound to other replicas are unaffected.
    def evict(self, replica_id: int) -> list[str]:
        self._alive[replica_id] = False
        # Collect first, then delete: you can't delete from a dict while iterating it (RuntimeError: dict changed size
        # during iteration)
        affected = [r for r, rep in self._bindings.items() if rep == replica_id]
        for r in affected:
            del self._bindings[r]
        return affected

    # Group by replica: one (batched) abort per replica, not one RPC per request.
    async def abort(self, request_ids: list[str]) -> None:
        groups: dict[int, list[str]] = {}
        for r in request_ids:
            rep = self._bindings.get(r)
            # Ids without a binding (never dispatched to this stage, or already released / evicted) are simply ignored
            if rep is not None:
                groups.setdefault(rep, []).append(r)
        for rep, ids in groups.items():
            # Pitfall: release the bindings only after the replica confirms the abort succeeded.
            # If abort raises, the exception propagates and these ids keep their bindings, so the caller can retry;
            # releasing before abort means after a failure you no longer know which replica they're on -> engine-side
            # leak, generation continues.
            # (Bindings on replicas that already succeeded are released; on retry they're ignored as unbound, so no
            # duplicate abort.)
            await self.replicas[rep].abort(ids)  # raises -> bindings are kept
            for r in ids:
                # pop(r, None): evict etc. may have removed this binding during the await
                self._bindings.pop(r, None)


# Two-phase abort: first collect the partial read-only -> engines of all stages confirm the abort -> only then commit.
async def abort_request(pools: list[StagePool], processor: PartialProcessor, request_id: str) -> OrchOutput:
    # Phase 1: read-only. Even if a later abort fails, processor state stays intact and a retry gets the same partial
    partial = processor.collect_partial(request_id)  # phase 1: read only
    for pool in pools:
        # Any pool failure raises immediately, skipping commit
        await pool.abort([request_id])  # any failure propagates before commit
    # Pitfall: commit must come after all aborts succeed. If you commit_abort first and abort then fails:
    # the engine keeps generating, later outputs arrive when the processor no longer has state for the request (error,
    # or re-created = resurrected),
    # and the partial is already lost for the retry.
    processor.commit_abort(request_id)  # phase 2: commit only after engines confirmed
    # Terminal message: finished=True + error="abort", carrying the partial result generated so far to the user
    return OrchOutput(request_id, partial, finished=True, error="abort")


# Dead-replica isolation: send one terminal error only to the requests bound to that replica; other requests keep
# running.
# Exactly one terminal message per affected request -- evict already deleted their bindings, so a repeated call won't
# return them.
def handle_dead_replica(pool: StagePool, replica_id: int) -> list[OrchOutput]:
    affected = pool.evict(replica_id)
    return [
        OrchOutput(r, None, finished=True, error=f"stage {pool.stage_id} replica {replica_id} died")
        for r in affected
    ]
```

C-13 · solutions/p05_rpc_router.py

```
import queue
import threading
from collections.abc import Callable, Hashable
from typing import Any


# RpcRouter: many caller threads share one result channel (e.g. one thread collects all workers' replies),
# and a key correlates each reply with the caller blocked waiting for it. Every method may be called from any thread.
# State (all protected by _lock):
#   _waiters  key -> a queue.Queue holding exactly one result (one per in-flight call)
#   _fatal    the fatal error latched by fail_all; once set, later register / call fail immediately
class RpcRouter:
```


© 2026 kzhou003

```
    """Correlates results coming back on one shared channel with blocked callers."""

    def __init__(self) -> None:
        self._lock = threading.Lock()
        self._waiters: dict[Hashable, queue.Queue] = {}
        self._fatal: BaseException | None = None
        self.late_results = 0

    # Caller thread: register a waiter
    def register(self, key: Hashable) -> queue.Queue:
        with self._lock:
            # Pitfall: the _fatal check must be under the same lock as the insert into _waiters. Otherwise fail_all may
            # clear _waiters
            # right between "check passed" and "insert", and the newly inserted waiter is never woken, only timing out
            if self._fatal is not None:
                raise self._fatal
            # Two in-flight calls with the same key would steal each other's result, so reject outright
            if key in self._waiters:
                raise ValueError(f"duplicate rpc key {key!r}")
            # maxsize=1: exactly one result per call
            waiter: queue.Queue = queue.Queue(maxsize=1)
            self._waiters[key] = waiter
            return waiter

    # Caller thread (call's finally): remove its own waiter
    def unregister(self, key: Hashable, waiter: queue.Queue) -> None:
        with self._lock:
            # Pitfall: compare by identity (is) before deleting. This key may already have been popped by deliver /
            # fail_all
            # and re-registered by a new call; a blind pop(key) would delete someone else's waiter, hanging them until
            # timeout
            if self._waiters.get(key) is waiter:
                del self._waiters[key]

    # Result-channel thread: hand the result to the matching waiter
    def deliver(self, key: Hashable, result: Any) -> bool:
        with self._lock:
            # pop, not get: a waiter is delivered at most once; deliver and fail_all are mutually exclusive under the
            # same lock,
            # so only one of them can get it
            waiter = self._waiters.pop(key, None)
            if waiter is None:
                # Caller already timed out / unregistered, or key never registered: a late result, count it and drop it
                # (incremented under the lock, so no lost counts across threads)
                self.late_results += 1
                return False
        # put outside the lock: the waiter has been removed from the dict and only we hold it; maxsize=1 and a single
        # put, so never Full.
        # Edge case: if the caller timed out just before this, the result lands in a queue nobody reads and is dropped
        # -- acceptable
        waiter.put_nowait(result)
        return True

    # Fatal error (e.g. worker process died): wake all in-flight calls and latch the error so later calls fail fast
    def fail_all(self, error: BaseException) -> None:
        with self._lock:
            # Latching and clearing under the same lock: any later register is guaranteed to see _fatal
            self._fatal = error
            waiters = list(self._waiters.values())
            self._waiters.clear()
        # Wake each one outside the lock: what goes in is the exception object itself; the call side recognizes it and
        # raises
        for w in waiters:
            try:
                w.put_nowait(error)
            except queue.Full:
                # Defensive: the waiter was removed from the dict and won't be delivered again, so it is normally never
                # full
                pass

    # Caller thread: register -> send -> block waiting for the result
    def call(self, key: Hashable, send: Callable[[], None], timeout: float) -> Any:
        # Pitfall: register BEFORE send. The reply arrives on another thread and may beat send()'s return;
        # if you send first, the reply finds no waiter -> counted as late_results and dropped -> this call waits until
        # timeout for nothing
        waiter = self.register(key)  # register BEFORE sending
        try:
            send()
            try:
                result = waiter.get(timeout=timeout)
            except queue.Empty:
                # from None: hide the internal queue.Empty; the caller only sees TimeoutError
                raise TimeoutError(f"rpc {key!r} timed out") from None
            # fail_all puts in an exception object: re-raise it on the caller thread
            if isinstance(result, BaseException):
                raise result
            return result
        finally:
            # Clean up on success / timeout / send raising alike, otherwise _waiters leaks and calling again with the
```


© 2026 kzhou003

```
            # same key reports duplicate;
            # cleaning up after timeout also ensures later late results are correctly counted as late_results
            self.unregister(key, waiter)
```

C-14 · solutions/p10_engine_core_loop.py

```
import queue
import threading
from collections.abc import Callable
from dataclasses import dataclass
from typing import Any, Literal, Protocol


# ------------------------------------------------------------------ given types
@dataclass
class Msg:
    kind: Literal["ADD", "ABORT", "UTILITY", "WAKEUP"]
    payload: Any = None


class Engine(Protocol):
    def add(self, req_id: str, data: Any) -> None: ...
    def abort(self, req_ids: list[str]) -> None: ...
    def has_work(self) -> bool: ...
    def step(self) -> list[Any]: ...
    def utility(self, name: str, args: tuple) -> Any: ...


# ------------------------------------------------------------------ implementation
# EngineCoreLoop: the main loop (busy loop) of the EngineCore process.
# The input thread puts Msgs into input_queue; each iteration of the main loop:
#   block for a message when idle -> process all queued messages
#   in FIFO order -> if there is work, step once and emit output.
# All output goes through emit(tuple): ("output", x) / ("utility",
# call_id, result, err) / ("fatal", repr) / ("shutdown",).
class EngineCoreLoop:
    """The busy loop of an EngineCore process (IO threads feed input_queue / drain emit)."""

    def __init__(self, engine: Engine, emit: Callable[[tuple], None]) -> None:
        self.engine = engine
        self.emit = emit
        # Thread-safe FIFO queue; the only input channel between other threads and the main loop.
        self.input_queue: queue.Queue[Msg] = queue.Queue()
        # Shutdown flag: a threading.Event can be set/queried safely across threads.
        self._shutdown = threading.Event()

    # Request shutdown: set the flag first, then enqueue a WAKEUP so a
    # main loop blocked in input_queue.get() wakes up and sees the flag.
    def request_shutdown(self) -> None:
        """Safe from any thread or a signal handler."""
        self._shutdown.set()
        # Pitfall: if only the flag is set, an idle main loop stays stuck in get()
        # forever; put_nowait never blocks, so it is safe to call from a signal handler.
        self.input_queue.put_nowait(Msg("WAKEUP"))

    # Dispatch one input message to the engine.
    def _handle(self, msg: Msg) -> None:
        if msg.kind == "ADD":
            req_id, data = msg.payload
            self.engine.add(req_id, data)
        elif msg.kind == "ABORT":
            # payload may be any iterable; normalize it to a list.
            self.engine.abort(list(msg.payload))
        elif msg.kind == "UTILITY":
            call_id, name, args = msg.payload
            # utility call (RPC-style): returns (result, None) on success,
            # (None, repr(e)) on failure; call_id lets the caller match them up.
            try:
                self.emit(("utility", call_id, self.engine.utility(name, args), None))
            except Exception as e:  # utility failure is call-scoped, not fatal
                self.emit(("utility", call_id, None, repr(e)))
        # WAKEUP: nothing to do, it only unblocks get()

    def run_busy_loop(self) -> None:
        try:
            while not self._shutdown.is_set():
                # When idle (no running requests), block for the next message to avoid spinning the CPU.
                if not self.engine.has_work():
                    self._handle(self.input_queue.get())  # block while idle: no spinning
                # Drain all queued messages without blocking, so ADD/ABORT take effect before the next step.
                while True:  # drain everything already queued, in FIFO order
                    try:
                        self._handle(self.input_queue.get_nowait())
                    except queue.Empty:
                        break
                # A shutdown request may have arrived while processing messages: do not step, exit right away.
```


© 2026 kzhou003

```
                if self._shutdown.is_set():
                    break
                # Pitfall: re-check has_work -- what was just processed may
                # only be WAKEUP/UTILITY/ABORT, leaving nothing to step.
                if self.engine.has_work():
                    for out in self.engine.step():
                        self.emit(("output", out))
        # Exceptions from engine.add/step etc. are fatal: report fatal and exit, without sending shutdown.
        except Exception as e:
            self.emit(("fatal", repr(e)))
            return
        # Normal shutdown path: emit one final shutdown notification.
        self.emit(("shutdown",))
```

C-15 · solutions/p11_chunk_aware_scheduler.py

```
from collections import deque
from dataclasses import dataclass
from typing import Protocol


# ------------------------------------------------------------------ given types
@dataclass
class Req:
    req_id: str
    arrival: int
    needs_chunk: bool = False  # receives its input from an upstream stage chunk by chunk
    status: str = "WAITING"


class BaseScheduler:
    """Stand-in for the upstream vLLM scheduler: knows nothing about chunks."""

    def __init__(self) -> None:
        self.waiting: deque[Req] = deque()
        self.running: list[Req] = []

    def schedule(self) -> list[str]:
        for r in self.running + list(self.waiting):
            if r.status in ("WAITING_FOR_CHUNK", "ABORTED"):
                raise AssertionError(f"upstream saw {r.req_id} in state {r.status}")
        out = [r.req_id for r in self.running] + [r.req_id for r in self.waiting]
        for r in self.waiting:
            r.status = "RUNNING"
        self.running += list(self.waiting)
        self.waiting.clear()
        return out


class ChunkSource(Protocol):
    def load_async(self, req_id: str) -> None: ...
    def is_ready(self, req_id: str) -> bool: ...


# ------------------------------------------------------------------ implementation
# ChunkAwareScheduler: a layer wrapped around the chunk-unaware upstream BaseScheduler.
# Before each schedule, needs_chunk requests whose next chunk is not ready yet are taken out of base's queues (park),
# so upstream never sees them; once the chunk is ready they are put
# back into the original queue with the original status (restore).
# After the upstream schedule finishes (including on exception), restore runs once more in finally.
class ChunkAwareScheduler:
    def __init__(self, base: BaseScheduler, source: ChunkSource) -> None:
        self.base = base
        self.source = source
        # Parking area: records origin ("waiting"/"running") and the pre-park status, restored exactly on restore.
        self.parked: dict[str, tuple[Req, str, str]] = {}  # id -> (req, origin queue, previous status)
        # Requests that already called load_async during this park, so each park triggers only one load.
        self._requested: set[str] = set()

    def add(self, req: Req) -> None:
        self.base.waiting.append(req)

    # abort can be called at any time: whether the request is parked, waiting or
    # running, remove it completely; it must never be scheduled or restored afterwards.
    def abort(self, req_id: str) -> None:
        # Remove from the parking area so _restore_ready will not put it back.
        entry = self.parked.pop(req_id, None)
        if entry is not None:
            entry[0].status = "ABORTED"
        # Iterate over a copy (list(...)) to avoid deleting while iterating.
        for r in list(self.base.waiting):
            if r.req_id == req_id:
                self.base.waiting.remove(r)
                r.status = "ABORTED"
        for r in list(self.base.running):
            if r.req_id == req_id:
```


© 2026 kzhou003

```
                self.base.running.remove(r)
                r.status = "ABORTED"
        self._requested.discard(req_id)

    # park: scan waiting and running, moving needs_chunk requests whose chunk is not ready into parked.
    def _park_not_ready(self) -> None:
        # Snapshot both queues first, then remove from the originals, to avoid mutating a container while iterating.
        for origin, q in (("waiting", list(self.base.waiting)), ("running", list(self.base.running))):
            for r in q:
                if r.needs_chunk and not self.source.is_ready(r.req_id):
                    (self.base.waiting.remove if origin == "waiting" else self.base.running.remove)(r)
                    # Pitfall: save the original status before changing it, or restore will put back WAITING_FOR_CHUNK.
                    self.parked[r.req_id] = (r, origin, r.status)
                    r.status = "WAITING_FOR_CHUNK"
                    # Issue load_async only once per park (_requested is cleared on restore/abort).
                    if r.req_id not in self._requested:
                        self._requested.add(r.req_id)
                        self.source.load_async(r.req_id)

    # restore: put parked requests whose chunk is ready back into their original queue and restore their status.
    def _restore_ready(self) -> None:
        # Collect first, then delete: you cannot del while iterating over a dict.
        ready = [e for rid, e in self.parked.items() if self.source.is_ready(rid)]
        # Requests restored in the same batch are sorted by arrival to keep arrival order (FCFS), not park order.
        for r, origin, prev in sorted(ready, key=lambda e: e[0].arrival):
            del self.parked[r.req_id]
            # Clear the "requested" mark; the next time it is parked, load_async is called again to load the next chunk.
            self._requested.discard(r.req_id)
            r.status = prev
            if origin == "waiting":
                self.base.waiting.append(r)
            else:
                self.base.running.append(r)

    # Order: (a) restore ready -> (b) park not ready -> (c) upstream schedule -> (d) restore again in finally.
    def schedule(self) -> list[str]:
        # Restore before park: a request that just became ready can be
        # scheduled this round; still-unready ones get re-parked next step.
        self._restore_ready()
        self._park_not_ready()
        try:
            return self.base.schedule()
        # Pitfall: restore even when the upstream schedule raises, or ready requests stay stuck in parked forever.
        finally:
            self._restore_ready()  # runs even if upstream raised
```

C-16 · solutions/p07_scheduler.py

```
from collections import deque
from collections.abc import Callable
from dataclasses import dataclass, field


# Seq: one request as seen by the scheduler. It only tracks token and block counts; it holds no real KV.
@dataclass
class Seq:
    seq_id: str
    prompt_len: int
    max_tokens: int
    # Output tokens generated so far.
    output: list[int] = field(default_factory=list)
    # num_computed: tokens already written to the KV cache; num_tokens - num_computed = tokens still to compute.
    # > 1 while prefill is unfinished (chunked prefill), always 1 during decode.
    num_computed: int = 0
    # KV blocks currently held (only grows, until release/preempt resets it to 0).
    num_blocks: int = 0
    status: str = "WAITING"  # WAITING | RUNNING | FINISHED

    # Total tokens = prompt + generated; recompute after preemption must cover all of them.
    @property
    def num_tokens(self) -> int:
        return self.prompt_len + len(self.output)


# Scheduler: a continuous batching scheduler.
# Each step, within the token budget (max_num_batched_tokens): first
# serve running requests (decode or continued chunked prefill),
# then admit new requests from waiting in FCFS order; when KV blocks run
# short, preempt (recompute-style) the request that joined running last.
class Scheduler:
    """Continuous batching with chunked prefill and recompute preemption."""

    def __init__(self, num_blocks: int, block_size: int, max_num_seqs: int, max_num_batched_tokens: int) -> None:
        self.num_blocks = num_blocks
        self.block_size = block_size
        self.max_num_seqs = max_num_seqs
```


© 2026 kzhou003

```
        self.max_num_batched_tokens = max_num_batched_tokens
        # waiting: FCFS queue; preempted requests go back to the head and are re-admitted first.
        self.waiting: deque[Seq] = deque()
        # running: in admission order; later = lower priority (preemption victims are taken from the tail).
        self.running: list[Seq] = []
        self.finished: list[Seq] = []
        # Invariant: blocks_used == sum(s.num_blocks for s in running).
        self.blocks_used = 0
        self.num_preemptions = 0
        # Result of the last schedule() (seq_id -> tokens computed this step), used by update().
        self._scheduled: dict[str, int] = {}

    def add(self, seq: Seq) -> None:
        self.waiting.append(seq)

    def has_work(self) -> bool:
        return bool(self.waiting or self.running)

    # ------------------------------------------------------------ helpers
    # ceil(n_tokens / block_size): blocks needed to hold n_tokens tokens.
    def _blocks_for(self, n_tokens: int) -> int:
        return -(-n_tokens // self.block_size)

    # Try to grow seq's blocks to hold n_tokens tokens; either
    # fully succeed, or return False without modifying any state.
    def _try_grow(self, seq: Seq, n_tokens: int) -> bool:
        need = self._blocks_for(n_tokens) - seq.num_blocks
        # Existing blocks suffice (e.g. decode within the same block); nothing to add.
        if need <= 0:
            return True
        # Atomicity: check first, then modify.
        if self.blocks_used + need > self.num_blocks:
            return False
        self.blocks_used += need
        seq.num_blocks += need
        return True

    # Return all blocks held by seq.
    def _release(self, seq: Seq) -> None:
        self.blocks_used -= seq.num_blocks
        seq.num_blocks = 0

    # Recompute-style preemption: drop all of seq's KV, put it back at the
    # head of waiting, and re-prefill later (including generated output).
    def _preempt(self, seq: Seq) -> None:
        self.running.remove(seq)
        self._release(seq)
        # Pitfall: num_computed must be reset to 0, otherwise re-admission thinks
        # the KV is still there. output is kept, so no generated tokens are lost.
        seq.num_computed = 0  # recompute prompt + already generated tokens later
        seq.status = "WAITING"
        # appendleft: the preempted request arrived earlier than everything in waiting; this keeps FCFS fair.
        self.waiting.appendleft(seq)
        self.num_preemptions += 1

    # ------------------------------------------------------------ step
    # Return this step's schedule: seq_id -> tokens to compute this
    # step. The sum over all requests is <= max_num_batched_tokens.
    def schedule(self) -> dict[str, int]:
        budget = self.max_num_batched_tokens
        scheduled: dict[str, int] = {}

        # 1) running requests first (decode, or continuing a chunked prefill)
        i = 0
        while i < len(self.running) and budget > 0:
            seq = self.running[i]
            # chunked prefill: the remaining uncomputed tokens may
            # exceed the budget; take only the slice the budget fits.
            n = min(seq.num_tokens - seq.num_computed, budget)
            # When blocks run short, repeatedly preempt the tail of running
            # (lowest priority) until it fits or seq itself gets preempted.
            while not self._try_grow(seq, seq.num_computed + n):
                victim = self.running[-1]  # lowest priority = most recently admitted
                self._preempt(victim)
                # The victim is at the tail of running (index >= i) and
                # has not been scheduled this step; the pop is defensive,
                # guaranteeing no preempted request is ever left in scheduled.
                scheduled.pop(victim.seq_id, None)
                # Preempted ourselves: seq is no longer in running; give up on this seq.
                if victim is seq:
                    break
            # while...else: the else runs only when the loop condition
            # becomes False normally (i.e. _try_grow succeeded, no break).
            else:
                scheduled[seq.seq_id] = n
                budget -= n
                i += 1
                continue
            # Pitfall: reaching here means seq itself was preempted; running has
```


© 2026 kzhou003

```
            # shrunk and index i now holds the next request, so do not i += 1.
            # seq itself was preempted; self.running shrank, do not advance i

        # 2) waiting requests, strictly FCFS
        # Bounded by both the token budget and max_num_seqs (the running concurrency cap).
        while self.waiting and budget > 0 and len(self.running) < self.max_num_seqs:
            seq = self.waiting[0]
            n = min(seq.num_tokens - seq.num_computed, budget)
            # Strict FCFS: if the head does not fit, stop; do not skip it to
            # schedule smaller requests behind it (avoids starving the head).
            # Admitting a new request also never preempts running requests for its sake.
            if not self._try_grow(seq, seq.num_computed + n):
                break
            self.waiting.popleft()
            seq.status = "RUNNING"
            self.running.append(seq)
            scheduled[seq.seq_id] = n
            budget -= n

        self._scheduled = scheduled
        # Return a copy so caller modifications do not affect what update() relies on.
        return dict(scheduled)

    # update: advance state after the model forward. Only seqs whose prefill
    # is complete (num_computed caught up with num_tokens) sample a new token.
    def update(self, sample_fn: Callable[[str], int], eos_token: int) -> list[Seq]:
        """Apply the last schedule(); sample_fn is called only for seqs that produced a token."""
        done: list[Seq] = []
        by_id = {s.seq_id: s for s in self.running}
        for sid, n in self._scheduled.items():
            seq = by_id[sid]
            seq.num_computed += n
            # Logits of an intermediate chunk are meaningless; do not sample.
            if seq.num_computed < seq.num_tokens:
                continue  # chunked prefill not finished yet
            tok = sample_fn(sid)
            # After the append num_tokens exceeds num_computed by 1, so the next step decodes exactly that 1 token.
            seq.output.append(tok)
            # Stop condition: sampled EOS or reached max_tokens; release blocks on completion.
            if tok == eos_token or len(seq.output) >= seq.max_tokens:
                seq.status = "FINISHED"
                self.running.remove(seq)
                self._release(seq)
                self.finished.append(seq)
                done.append(seq)
        # Clear it so the same schedule cannot be updated twice.
        self._scheduled = {}
        return done
```

C-17 · solutions/p06_block_manager.py

```
from collections import OrderedDict, deque


# Raised when allocation fails; the caller (scheduler) uses it to decide whether to preempt or wait.
class OutOfBlocks(Exception):
    pass


# Hash chain: each full block's hash depends on the previous block's hash,
# so one hash identifies "the whole prefix from the start of the sequence up to this block", not just its own tokens.
# Pitfall: hashing only this block's tokens makes identical fragments under different prefixes falsely hit.
def _block_hash(parent: int | None, tokens: tuple[int, ...]) -> int:
    return hash((parent, tokens))


# BlockManager: a vLLM-style paged KV cache allocator.
# - Each seq owns a block table (logical block index -> physical block id).
# - Physical blocks carry a refcount; with prefix cache enabled, multiple seqs can share the same prefix blocks.
# - A block whose refcount drops to 0 but still has a hash is not freed immediately; it enters the evictable LRU pool
#   and can be reused by later requests. Blocks are evicted from that pool in LRU order only when space runs out.
class BlockManager:
    """Paged KV-cache allocator with ref counts and optional prefix caching."""

    def __init__(self, num_blocks: int, block_size: int, enable_prefix_cache: bool = False) -> None:
        self.num_blocks = num_blocks
        self.block_size = block_size
        self.enable_prefix_cache = enable_prefix_cache
        # free list: fully free blocks (no hash, ref==0), FIFO order; initially 0..N-1, taken from the left.
        self._free: deque[int] = deque(range(num_blocks))
        # Refcount of each physical block (how many seqs' block tables contain it).
        self._ref = [0] * num_blocks
        # Prefix hash currently registered for each physical block;
        # None = not full or not registered (no prefix-cache hit).
        self._hash: list[int | None] = [None] * num_blocks
        # Reverse index prefix hash -> physical block, used for prefix-cache lookup.
```


© 2026 kzhou003

```
        self._hash_to_block: dict[int, int] = {}
        # Invariant: blocks in _evictable have ref==0 and _hash[b]
        # not None; the OrderedDict head is oldest (evicted first).
        self._evictable: OrderedDict[int, None] = OrderedDict()  # ref==0 but still cached (LRU)
        # seq_id -> block table (list of physical block ids).
        self._tables: dict[str, list[int]] = {}
        # seq_id -> all tokens of the seq so far (prompt + appended tokens), used to hash subsequent full blocks.
        self._tokens: dict[str, list[int]] = {}
        # seq_id -> chained hash of each "full" block (length = len(tokens) // block_size).
        self._seq_hashes: dict[str, list[int]] = {}
        # seq_id -> tokens hit in the prefix cache at allocate time (= hit blocks * block_size).
        self._cached_tokens: dict[str, int] = {}

    # ---------------------------------------------------------------- queries
    # Available blocks = truly free + evictable (evictable blocks can be reclaimed when needed).
    @property
    def num_free_blocks(self) -> int:
        return len(self._free) + len(self._evictable)

    # Return a copy so the caller cannot modify the internal block table.
    def block_table(self, seq_id: str) -> list[int]:
        return list(self._tables[seq_id])

    def num_cached_tokens(self, seq_id: str) -> int:
        return self._cached_tokens.get(seq_id, 0)

    # Logical position pos -> flat slot number in the KV cache: physical block id * block_size + offset within block.
    def slot(self, seq_id: str, pos: int) -> int:
        table = self._tables[seq_id]
        return table[pos // self.block_size] * self.block_size + pos % self.block_size

    # -(-a // b) is the integer form of ceil(a / b).
    # Note: this ignores prefix-cache hits; it is a conservative estimate.
    def can_allocate(self, num_tokens: int) -> bool:
        return -(-num_tokens // self.block_size) <= self.num_free_blocks

    # ---------------------------------------------------------------- internals
    # Take a new physical block: prefer the free list, otherwise evict the oldest one from the evictable pool.
    def _take_block(self) -> int:
        if self._free:
            b = self._free.popleft()
        elif self._evictable:
            # popitem(last=False) pops the LRU head, i.e. the least recently used cached block.
            b, _ = self._evictable.popitem(last=False)
            h = self._hash[b]
            # Pitfall: delete only if the index still points at this block; the same hash
            # may have been registered by another block (setdefault keeps the first one).
            if h is not None and self._hash_to_block.get(h) == b:
                del self._hash_to_block[h]
            # After eviction this block's contents will be overwritten;
            # its hash must be cleared or it will falsely hit later.
            self._hash[b] = None
        else:
            raise OutOfBlocks
        self._ref[b] = 1
        return b

    # Register a full block in the prefix-cache index (no-op when prefix cache is disabled).
    def _register(self, b: int, h: int) -> None:
        if not self.enable_prefix_cache:
            return
        self._hash[b] = h
        # setdefault: if a block is already registered for the same
        # prefix, keep the old one; do not overwrite the index.
        self._hash_to_block.setdefault(h, b)

    # ---------------------------------------------------------------- mutation
    # allocate: allocate a block table for a new seq's prompt.
    # Algorithm: 1) compute chained hashes of all full blocks; 2)
    # with prefix cache on, match the longest contiguous hit prefix;
    # 3) check capacity first (all or nothing), then in one go:
    #    reuse hit blocks (ref+1) + take new blocks for the rest.
    def allocate(self, seq_id: str, token_ids: list[int]) -> list[int]:
        assert seq_id not in self._tables
        bs = self.block_size
        # Total blocks needed (including the last partial block).
        n_blocks = -(-len(token_ids) // bs)
        hashes: list[int] = []
        parent = None
        # Hash only full blocks; the partial tail block's contents will still change and cannot be cached.
        for i in range(len(token_ids) // bs):
            parent = _block_hash(parent, tuple(token_ids[i * bs:(i + 1) * bs]))
            hashes.append(parent)

        hits: list[int] = []
        if self.enable_prefix_cache:
            # Pitfall: even if the prompt fully hits, at least 1 token must be left for
            # the model to compute, otherwise there are no logits for the next token.
            # So match at most (len-1)//bs blocks; e.g. len=8, bs=4 can hit only 1 block.
```


© 2026 kzhou003

```
            matchable = (len(token_ids) - 1) // bs  # always leave >= 1 token to compute
            for h in hashes[:matchable]:
                b = self._hash_to_block.get(h)
                # Prefix matching must be contiguous: once a block misses, later blocks are
                # unusable even if they hit (with a hash chain they would not hit anyway).
                if b is None:
                    break
                hits.append(b)

        # Some hit blocks are in the evictable pool (ref==0); once reused they are no longer available for allocation,
        # so subtract them from the count available for new blocks to avoid double counting.
        hits_from_evictable = sum(1 for b in hits if self._ref[b] == 0)
        # Atomicity: check capacity before any state change; on failure the internal state is completely unchanged.
        if n_blocks - len(hits) > self.num_free_blocks - hits_from_evictable:
            raise OutOfBlocks  # check first -> allocation is all-or-nothing

        table: list[int] = []
        for b in hits:
            # Reuse a block from evictable: take it out of the LRU pool (back to "in use").
            if self._ref[b] == 0:
                self._evictable.pop(b, None)
            self._ref[b] += 1
            table.append(b)
        # Missed part: take new blocks; if the position is a full block, register its hash for later requests to reuse.
        for i in range(len(hits), n_blocks):
            b = self._take_block()
            table.append(b)
            if i < len(hashes):
                self._register(b, hashes[i])

        self._tables[seq_id] = table
        self._tokens[seq_id] = list(token_ids)
        self._seq_hashes[seq_id] = hashes
        self._cached_tokens[seq_id] = len(hits) * bs
        return list(table)

    # append_slot: during decode, append a slot for the new token and return that token's slot in the KV cache.
    # If the current last block is full, take a new block first;
    # when a block fills up, compute and register its chained hash.
    def append_slot(self, seq_id: str, token_id: int) -> int:
        bs = self.block_size
        tokens, table = self._tokens[seq_id], self._tables[seq_id]
        # Position of the new token = current token count.
        pos = len(tokens)
        # A new block is needed when pos lands exactly at the start of a new block (block table is full).
        if pos // bs == len(table):
            # Atomicity: _take_block comes before tokens.append, so if it raises OutOfBlocks the seq state is intact.
            table.append(self._take_block())  # may raise OutOfBlocks before any mutation
        tokens.append(token_id)
        # The block is exactly full after the append: compute its hash (parent
        # = previous full block's hash) and register it in the prefix cache.
        if (pos + 1) % bs == 0:
            hashes = self._seq_hashes[seq_id]
            parent = hashes[-1] if hashes else None
            h = _block_hash(parent, tuple(tokens[pos + 1 - bs:pos + 1]))
            hashes.append(h)
            self._register(table[pos // bs], h)
        return self.slot(seq_id, pos)

    # free: release all of a seq's blocks. Once ref hits 0, hashed blocks
    # go to evictable (cache kept), unhashed ones go back to the free list.
    def free(self, seq_id: str) -> None:
        table = self._tables.pop(seq_id)
        self._tokens.pop(seq_id)
        self._seq_hashes.pop(seq_id)
        self._cached_tokens.pop(seq_id, None)
        # Release in reverse: tail blocks enter the LRU first and
        # are thus evicted first; blocks at the start of the prefix
        # enter later and are kept longer (blocks at the start are more likely to be shared by other requests).
        for b in reversed(table):  # tail blocks become the first eviction candidates
            self._ref[b] -= 1
            # Only actually release when ref hits 0; shared blocks still referenced by other seqs stay put.
            if self._ref[b] == 0:
                if self._hash[b] is not None:
                    self._evictable[b] = None
                else:
                    self._free.append(b)
```

C-18 · solutions/p08_sampler.py

```
from dataclasses import dataclass

import torch


# Per-request sampling params. seed=None means reproducibility is not needed; use the shared global default generator.
```


© 2026 kzhou003

```python
@dataclass
class SamplingParams:
    temperature: float = 1.0
    top_k: int = -1
    top_p: float = 1.0
    repetition_penalty: float = 1.0
    seed: int | None = None


# repetition penalty: lower the logits of tokens that have already appeared.
# Positive logits are divided by penalty, negative ones multiplied; both lower the token's probability (penalty > 1).
def apply_repetition_penalty(logits: torch.Tensor, prev_tokens: list[int], penalty: float) -> torch.Tensor:
    """HF/vLLM convention: positive logits are divided, negative ones multiplied."""
    if penalty == 1.0 or not prev_tokens:
        return logits
    # Pitfall: do not modify the caller's logits in place.
    out = logits.clone()
    # Dedup: a token appearing several times in the history is
    # penalized only once (penalty does not stack per occurrence).
    idx = torch.tensor(sorted(set(prev_tokens)), dtype=torch.long)
    vals = out[idx]
    out[idx] = torch.where(vals > 0, vals / penalty, vals * penalty)
    return out


# Filter one row of logits (shape [V]) with top_k then top_p;
# filtered positions are set to -inf (probability 0 after softmax).
def _filter_top_k_top_p(logits: torch.Tensor, top_k: int, top_p: float) -> torch.Tensor:
    out = logits.clone()
    # top_k <= 0 or >= V means no filtering.
    if 0 < top_k < out.numel():
        # The k-th largest value is the threshold; values strictly
        # below it become -inf (ties with the k-th value are kept).
        kth = torch.topk(out, top_k).values[-1]
        out[out < kth] = float("-inf")
    if top_p < 1.0:
        # Nucleus sampling on the distribution after top_k: sort by probability descending and take the cumulative sum.
        sorted_logits, order = torch.sort(out, descending=True)
        probs = torch.softmax(sorted_logits, dim=-1)
        cum = torch.cumsum(probs, dim=-1)
        # cum - probs is the mass "before this token"; remove only when that preceding mass already exceeds top_p,
        # so the token that crosses the threshold is kept: we keep "the shortest prefix whose mass reaches top_p".
        remove = (cum - probs) > top_p  # keep the token that crosses the threshold
        # Pitfall: always keep at least the single most likely token,
        # so not everything becomes -inf and softmax gives NaN.
        remove[0] = False
        # order maps sorted positions back to original vocab indices.
        out[order[remove]] = float("-inf")
    return out


# Sampler: row-wise sampling that guarantees batch invariance --
# the token a request samples depends only on its own logits, params
# and generator state, not on other rows in the batch or their order.
class Sampler:
    """Row-wise sampler whose result for a request never depends on the rest of the batch."""

    def __init__(self) -> None:
        # req_id -> the request's private torch.Generator (only requests with a seed have one).
        self._generators: dict[str, torch.Generator] = {}
        # Default generator shared by requests without a seed (no reproducibility guarantee).
        self._default = torch.Generator().manual_seed(0)

    # Get a request's generator: seeded requests create it from the seed on
    # first use, then keep advancing the same random stream across decode steps.
    def _gen(self, req_id: str, p: SamplingParams) -> torch.Generator:
        if p.seed is None:
            return self._default
        g = self._generators.get(req_id)
        # Pitfall: do not recreate the generator from the seed every step, or every step draws the same random numbers.
        if g is None:
            g = torch.Generator().manual_seed(p.seed)
            self._generators[req_id] = g
        return g

    # Drop the generator when the request finishes; a new request
    # with the same id restarts the random stream from the seed.
    def free(self, req_id: str) -> None:
        self._generators.pop(req_id, None)

    # logits: [B, V]; returns [B] long.
    # Per row: repetition penalty -> (greedy | temperature -> top_k/top_p -> multinomial).
    def sample(
        self,
        logits: torch.Tensor,
        req_ids: list[str],
        params: list[SamplingParams],
        prev_tokens: list[list[int]],
    ) -> torch.Tensor:
```


© 2026 kzhou003

```
        out = torch.empty(logits.shape[0], dtype=torch.long)
        for i, (rid, p) in enumerate(zip(req_ids, params, strict=True)):
            # Compute in float32 to avoid softmax/cumsum numerical error with low-precision logits.
            row = apply_repetition_penalty(logits[i].float(), prev_tokens[i], p.repetition_penalty)
            # temperature≈0 is treated as greedy: plain argmax, consuming no randomness from the generator.
            if p.temperature < 1e-5:
                out[i] = int(torch.argmax(row))
                continue
            # Divide by temperature first, then apply top_k/top_p (same order as the spec).
            row = _filter_top_k_top_p(row / p.temperature, p.top_k, p.top_p)
            probs = torch.softmax(row, dim=-1)
            # Draw exactly 1 sample per row with the request's own generator; this is the key to batch invariance.
            out[i] = int(torch.multinomial(probs, 1, generator=self._gen(rid, p)))
        return out
```

C-19 · solutions/p09_paged_attention.py

```
import torch


# write_kv: scatter this step's newly computed K/V into the paged KV cache according to slot_mapping.
# slot = physical block id * block_size + offset within block (same definition as BlockManager.slot).
def write_kv(
    k_cache: torch.Tensor,  # [num_blocks, block_size, num_kv_heads, head_dim]
    v_cache: torch.Tensor,
    k: torch.Tensor,  # [num_tokens, num_kv_heads, head_dim]
    v: torch.Tensor,
    slot_mapping: torch.Tensor,  # [num_tokens], -1 = padding token, do not write
) -> None:
    bs = k_cache.shape[1]
    # Pitfall: padding tokens have slot -1 and must be filtered out; otherwise
    # -1 is treated as a negative index and written into the last block.
    valid = slot_mapping >= 0
    slots = slot_mapping[valid]
    # Split the flat slot into a (block, offset) 2-D index.
    blocks, offsets = slots // bs, slots % bs
    # Advanced-indexing assignment: k_cache[blocks[j], offsets[j]]
    # = k[valid][j]; each element is [num_kv_heads, head_dim].
    k_cache[blocks, offsets] = k[valid]
    v_cache[blocks, offsets] = v[valid]


# paged_decode_attention: decode-phase attention; each sequence has only 1 query token.
# Per sequence: use the block table to gather K/V scattered across physical blocks into a contiguous [n, KVH, D],
# truncate to context_len, under GQA replicate KV heads up to the
# query head count, then do standard softmax(QK^T * scale) V.
def paged_decode_attention(
    q: torch.Tensor,  # [B, num_heads, head_dim] (one query token per sequence)
    k_cache: torch.Tensor,
    v_cache: torch.Tensor,
    block_tables: torch.Tensor,  # [B, max_blocks], unused tail entries may hold any value
    context_lens: torch.Tensor,  # [B]
    scale: float,
) -> torch.Tensor:
    bsz, num_heads, _ = q.shape
    block_size, num_kv_heads = k_cache.shape[1], k_cache.shape[2]
    # GQA: every group of query heads shares one KV head.
    group = num_heads // num_kv_heads
    out = torch.empty_like(q)
    for b in range(bsz):
        # n: this sequence's context length (including the current token, whose KV write_kv has already written).
        n = int(context_lens[b])
        # Blocks actually used = ceil(n / block_size).
        nb = -(-n // block_size)
        # Pitfall: take only the first nb entries; unused trailing entries may
        # hold arbitrary values (garbage/out of range) and must not be gathered.
        table = block_tables[b, :nb].long()
        # k_cache[table]: [nb, block_size, KVH, D] -> flatten to [nb*block_size, KVH, D],
        # then truncate to n, dropping unwritten slots in the last block.
        k = k_cache[table].reshape(-1, num_kv_heads, k_cache.shape[-1])[:n]  # [n, KVH, D]
        v = v_cache[table].reshape(-1, num_kv_heads, v_cache.shape[-1])[:n]
        # repeat_interleave (not repeat): query head h maps to KV head h // group; the order must match.
        k = k.repeat_interleave(group, dim=1)  # GQA: [n, H, D]
        v = v.repeat_interleave(group, dim=1)
        # scores[h, j] = q[h] · k[j, h] * scale -> [H, n]; computed in float32 for numerical stability.
        scores = torch.einsum("hd,nhd->hn", q[b].float(), k.float()) * scale
        # softmax over the key dim (n); in decode the query is the last token, so no causal mask is needed.
        probs = torch.softmax(scores, dim=-1)
        # out[h] = sum_j probs[h, j] * v[j, h] -> [H, D], then cast back to the input dtype.
        out[b] = torch.einsum("hn,nhd->hd", probs, v.float()).to(q.dtype)
    return out
```


© 2026 kzhou003

C-20 · solutions/p12_shm_connector.py

```
import fcntl
import hashlib
import os
import tempfile
from multiprocessing import resource_tracker
from multiprocessing.shared_memory import SharedMemory

# Segment layout: [ 8-byte little-endian length n | n-byte payload | (possible page-alignment padding) ]
# Pitfall: do not use shm.size as the payload length -- macOS / Linux round the
# segment size up to the page size, so the real length must come from the header.
_HEADER = 8  # little-endian payload length; shm segments may be rounded up to a page


# _untrack: unregister a just-created segment from this process's
# resource_tracker, i.e. "transfer ownership to the reader".
# Flow: writer create → write → close (no unlink) → reader opens → reads → unlink.
# Without unregistering, the tracker unlinks the "leaked" segment when
# the writer process exits, possibly before the reader gets to read it.
def _untrack(shm: SharedMemory) -> None:
    # Python < 3.13 registers every created segment with the creator's resource tracker.
    # When that tracker shuts down (e.g. an independent writer process exits) it unlinks
    # every still-registered segment as "leaked" -- possibly before the consumer read it.
    # Ownership moves to the reader, who unlinks after reading.
    try:
        # shm._name has a leading "/" (POSIX name) and is what the
        # tracker registered, so _name, not name, must be used here
        resource_tracker.unregister(shm._name, "shared_memory")  # type: ignore[attr-defined]
    except Exception:
        # Best effort: on some platforms/versions it was never registered, so a failed unregister does not matter
        pass


# ShmConnector: the data plane's "shared-memory mailbox" between stages. One key = one shm segment + one lock file.
# Protocol:
#   - lock file exists   = someone wrote and it has not been read yet (or is being written)
#   - lock file held via flock = writer is writing; reader should return None immediately (non-blocking)
#   - after reading, the reader unlinks the segment + deletes the lock file (consume = delete; one consumer per key)
#   - the producer records keys it wrote (_pending); on request abort, cleanup_prefix cleans up unread segments
class ShmConnector:
    """put/get of byte payloads through named POSIX shared memory, one consumer per key."""

    def __init__(self, lock_dir: str | None = None, namespace: str = "nvo") -> None:
        # On Linux the lock file lives in /dev/shm (tmpfs, in memory alongside the
        # segment); macOS has no /dev/shm, so fall back to the system temp dir
        default = "/dev/shm" if os.path.isdir("/dev/shm") else tempfile.gettempdir()
        self.lock_dir = lock_dir or default
        self.namespace = namespace
        # dict used as an "ordered set": only keys matter, not values
        self._pending: dict[str, None] = {}  # keys this producer wrote and may need to clean up

    # macOS limits POSIX shm names to 31 bytes -> hash the logical key
    # The name must be a deterministic function of key: a reader in another
    # process only knows key and must derive the same segment name on its own.
    # E.g. namespace="nvo" → "nvo_" + 20 hex chars = 24 bytes (< 31 even with the leading "/")
    def shm_name(self, key: str) -> str:
        return f"{self.namespace}_" + hashlib.sha1(key.encode()).hexdigest()[:20]

    def lock_path(self, key: str) -> str:
        return os.path.join(self.lock_dir, self.shm_name(key) + ".lock")

    # put: writer path. Take an exclusive lock first (tells readers
    # "writing"), then create the segment, write header + data, close.
    # After writing, the lock file stays (meaning "data pending")
    # and so does the segment (ownership is with the reader).
    def put(self, key: str, data: bytes) -> None:
        # O_CREAT: the writer creates the lock file; the reader only opens an existing one
        fd = os.open(self.lock_path(key), os.O_CREAT | os.O_RDWR, 0o600)
        try:
            # Blocking exclusive lock: if a reader holds the lock while reading, the writer waits until it is done
            fcntl.flock(fd, fcntl.LOCK_EX)  # "being written" marker for readers
            name, size = self.shm_name(key), _HEADER + len(data)
            try:
                shm = SharedMemory(name=name, create=True, size=size)
            except FileExistsError:  # stale segment from an earlier incarnation
                # A segment with the same name already exists (previous write never read
                # / leftover from a crash): delete the old segment first, then recreate.
                # Pitfall: do not reuse the old segment directly -- it may be smaller than the new data.
                old = SharedMemory(name=name)
                old.close()
                old.unlink()
                shm = SharedMemory(name=name, create=True, size=size)
            # Pitfall: skip this and the tracker unlinks the not-yet-read segment when the writer process exits
            _untrack(shm)
            shm.buf[:_HEADER] = len(data).to_bytes(_HEADER, "little")
            shm.buf[_HEADER:size] = data
```


© 2026 kzhou003

```
            # Only close (unmap from this process), do not unlink -- the segment is left for the reader
            shm.close()
            self._pending[key] = None
        finally:
            # Only after the lock is released can the reader's LOCK_NB succeed → readers never see half-written data
            fcntl.flock(fd, fcntl.LOCK_UN)
            os.close(fd)

    # get: reader path, non-blocking. Three None cases: no lock file / lock held by writer / no segment.
    # On success, read the data and "consume" it: unlink the segment +
    # delete the lock file, so a second get on the same key returns None.
    def get(self, key: str) -> bytes | None:
        """Non-blocking. Returns None if absent or still being written. Consumes the key."""
        try:
            # No O_CREAT: no lock file = never written (or already consumed); do not create it on the writer's behalf
            fd = os.open(self.lock_path(key), os.O_RDWR)
        except FileNotFoundError:
            return None
        try:
            try:
                # LOCK_NB: if the lock is unavailable, raise BlockingIOError
                # immediately instead of waiting for the writer.
                # Note: flock locks belong to the open file description; another
                # fd in the same process holding the lock also conflicts.
                fcntl.flock(fd, fcntl.LOCK_EX | fcntl.LOCK_NB)
            except BlockingIOError:
                return None  # writer still holds the lock
            try:
                # Attach the existing segment (create=False). On Python <
                # 3.13 attach also registers with this process's tracker,
                # but shm.unlink() below unregisters internally, so the reader side needs no _untrack
                shm = SharedMemory(name=self.shm_name(key))
            except FileNotFoundError:
                # Lock file present but no segment: e.g. the writer crashed
                # before creating it, or someone just cleaned it up
                return None
            n = int.from_bytes(shm.buf[:_HEADER], "little")
            # bytes(...) makes a copy: the memoryview is invalid after close, so copy out first
            data = bytes(shm.buf[_HEADER:_HEADER + n])
            shm.close()
            shm.unlink()  # the reader frees the memory
            # Deleting the lock file = declaring "consumed"; we still hold the lock, so a writer cannot cut in midway
            try:
                os.unlink(self.lock_path(key))
            except FileNotFoundError:
                pass
            return data
        finally:
            fcntl.flock(fd, fcntl.LOCK_UN)
            os.close(fd)

    # _drop: force-delete a key's segment and lock file (without reading data); every step tolerates "already gone".
    def _drop(self, key: str) -> None:
        try:
            shm = SharedMemory(name=self.shm_name(key))
            shm.close()
            shm.unlink()
        except FileNotFoundError:
            pass
        try:
            os.unlink(self.lock_path(key))
        except FileNotFoundError:
            pass

    # cleanup_prefix: called on the producer side when a request
    # finishes/aborts (keys look like f"{req_id}_{stage}_{chunk}",
    # the prefix is usually f"{req_id}_"). Cleans only keys that are
    # "written by me + prefix match + not yet consumed"; returns the count.
    # Pitfall: the prefix must include the separator; "rA_" will not hit "rAx_...", but "rA" will.
    def cleanup_prefix(self, prefix: str) -> int:
        """Producer-side: unlink every key we wrote that starts with prefix and was never consumed."""
        n = 0
        # Copy to a list first: the loop does del self._pending[key], and a dict cannot be modified while iterating
        for key in [k for k in self._pending if k.startswith(prefix)]:
            del self._pending[key]
            # Lock file still present = not consumed by a reader (the
            # reader deletes it on consume) → only then clean up and count
            if os.path.exists(self.lock_path(key)):
                self._drop(key)
                n += 1
        return n
```

C-21 · solutions/p13_chunk_adapter.py

```
from threading import Condition
```


© 2026 kzhou003

```
import copy
import threading
from collections import deque
from collections.abc import Callable
from typing import Any


# ---------------------------------------------------------------- given types
# InMemoryConnector: the transport layer given by the problem (replacing
# p12's shm connector); it only stores/fetches by key, and get consumes.
class InMemoryConnector:
    """Transport only: put/get by key, get consumes. Thread-safe. Hooks let tests add latency."""

    def __init__(self, on_put: Callable[[str], None] | None = None, on_get: Callable[[str], None] | None = None):
        self._store: dict[str, Any] = {}
        self._lock = threading.Lock()
        self.on_put, self.on_get = on_put, on_get
        self.puts: list[str] = []

    def put(self, key: str, obj: Any) -> None:
        # The hook is called outside the lock: tests can sleep here
        # to simulate slow transport without locking the whole store
        if self.on_put:
            self.on_put(key)
        with self._lock:
            self._store[key] = obj
            self.puts.append(key)

    def get(self, key: str) -> Any | None:
        with self._lock:
            present = key in self._store
        if not present:
            return None
        if self.on_get:
            self.on_get(key)
        # pop with a default: in theory someone may take it between the two
        # lock acquisitions; return None then instead of raising KeyError
        with self._lock:
            return self._store.pop(key, None)


# ---------------------------------------------------------------- implementation
# ChunkAdapter: one stage's "sender + receiver" in the async-chunk data plane.
#   Sender: the scheduler thread calls save_async to submit a snapshot → the background save thread runs build_payload,
#           then puts to the connector under key = f"{req_id}_{self.stage_id}_{chunk_id}" (chunk_id contiguous from 0).
#   Receiver: load_async registers req → the background recv thread polls get on f"{req_id}_{src_stage_id}_{chunk_id}",
#           commits in order to _ready, and the scheduler thread takes them via poll_ready.
# The two background threads keep slow transport/serialization from blocking the
# scheduler thread; all shared state is read and written under its own lock.
class ChunkAdapter:
    """Sender + receiver halves of the async-chunk data plane for one stage."""

    def __init__(
        self,
        connector: InMemoryConnector,
        stage_id: int,
        src_stage_id: int | None,
        build_payload: Callable[[Any], Any | None],
    ) -> None:
        self.connector = connector
        # stage_id: this stage's id written into keys when sending; src_stage_id: the upstream's id when receiving
        self.stage_id = stage_id
        self.src_stage_id = src_stage_id
        self.build_payload = build_payload
        self._stop = threading.Event()
        # sender
        # _save_q elements: (req_id, deep-copied snapshot, is_final). deque append/popleft are themselves thread-safe
        self._save_q: deque = deque()
        # _save_cond is only used to "wake the save thread"; it does not protect data
        self._save_cond: Condition = threading.Condition()
        # _send_lock protects _send_chunk and _cancelled (accessed concurrently
        # by the scheduler thread's cancel_sender and the save thread)
        self._send_lock = threading.Lock()
        self._send_chunk: dict[str, int] = {}  # req_id → next chunk id to send
        self._cancelled: set[str] = set()
        # receiver
        self._load_q: deque[str] = deque()  # req_ids to poll (round-robin queue)
        self._recv_cond = threading.Condition()
        # _recv_lock protects the three dicts below (recv thread
        # vs the scheduler thread's poll_ready / cleanup_receiver)
        self._recv_lock = threading.Lock()
        # req_id → generation token: a fresh object() per registration, compared with `is` to tell
        # "whether it was cleaned up or re-registered during the read"
        self._registered: dict[str, object] = {}
        self._recv_chunk: dict[str, int] = {}  # req_id → next expected chunk id
        self._ready: dict[str, list] = {}  # req_id → chunks committed in order, waiting for poll_ready to take them
        self._threads: list[threading.Thread] = []
```


© 2026 kzhou003

```
    def start(self) -> None:
        for target, name in ((self._save_loop, "save"), (self._recv_loop, "recv")):
            # daemon=True: even if stop is never called, these two threads will not hang process exit
            t = threading.Thread(target=target, name=f"chunk-{name}-{self.stage_id}", daemon=True)
            t.start()
            self._threads.append(t)

    def stop(self) -> None:
        # Set stop first, then notify_all: threads wake, check _stop and
        # exit; in the reverse order they may sleep an extra timeout in wait
        self._stop.set()
        for cond in (self._save_cond, self._recv_cond):
            with cond:
                cond.notify_all()
        for t in self._threads:
            t.join(2.0)

    # ---------------------------------------------------------- sender
    # save_async: called by the scheduler thread; must return immediately.
    # Pitfall: always deepcopy -- the caller keeps mutating snapshot in place (e.g. appending to the token list);
    # storing only a reference means the background thread's build_payload sees "future" state.
    def save_async(self, req_id: str, snapshot: Any, is_final: bool) -> None:
        self._save_q.append((req_id, copy.deepcopy(snapshot), is_final))  # caller keeps mutating
        with self._save_cond:
            self._save_cond.notify()

    # cancel_sender: called when a request is aborted. All further non-final
    # data for the req is dropped, but the final marker must still be sent,
    # otherwise the downstream receiver never sees finished=True and keeps polling this req forever.
    def cancel_sender(self, req_id: str) -> None:
        with self._send_lock:
            self._cancelled.add(req_id)

    def _save_loop(self) -> None:
        while not self._stop.is_set():
            if not self._save_q:
                with self._save_cond:
                    # Re-check after taking the lock: avoids the lost wakeup
                    # "check empty → producer append+notify → only then we wait"
                    if not self._save_q and not self._stop.is_set():
                        self._save_cond.wait(timeout=0.1)  # timeout = lost-notify fallback
                continue
            # A single save thread processes FIFO → chunk order within a req matches the order of save_async calls
            req_id, snap, is_final = self._save_q.popleft()
            with self._send_lock:
                cancelled = req_id in self._cancelled
            if cancelled and not is_final:
                continue  # drop data, but never the end-of-stream marker
            # build_payload can be slow (serializing/copying tensors);
            # keep it outside the lock so it does not block cancel_sender
            data = None if cancelled else self.build_payload(snap)  # outside any lock
            # payload is None (nothing new to send this step) and not final → skip without consuming a chunk id.
            # E.g. 6 saves, the 3rd with payload=None → chunks 0..4 sent, 5 contiguous ids
            if data is None and not is_final:
                continue  # nothing to send this step; chunk ids stay contiguous
            with self._send_lock:
                cid = self._send_chunk.get(req_id, 0)
            try:
                # The key uses our own stage_id; downstream builds the same key with src_stage_id
                self.connector.put(f"{req_id}_{self.stage_id}_{cid}", {"data": data, "finished": is_final})
            except Exception:
                # put failed: cid does not advance; this chunk is lost but later
                # chunk ids stay contiguous (the receiver will not stall on a hole)
                continue
            with self._send_lock:  # advance only after a successful put
                self._send_chunk[req_id] = cid + 1
                if is_final:
                    # End of stream: clear the req's send state so the dict does not grow without bound in long runs
                    self._send_chunk.pop(req_id, None)
                    self._cancelled.discard(req_id)

    # ---------------------------------------------------------- receiver
    # load_async: register a req to receive. Idempotent: return if already
    # registered (prevents the same req appearing twice in _load_q,
    # where two pollers would race for the same chunk id).
    def load_async(self, req_id: str) -> None:
        with self._recv_lock:
            if req_id in self._registered:
                return
            self._registered[req_id] = object()  # generation token
            self._recv_chunk.setdefault(req_id, 0)
        self._load_q.append(req_id)
        with self._recv_cond:
            self._recv_cond.notify()

    # poll_ready: atomically "swap out" _ready as a whole -- take every chunk
    # committed since the last poll (ordered by chunk id within each req)
    def poll_ready(self) -> dict[str, list]:
        """Called on the scheduler thread: committed chunks, in order, since the last poll."""
```


© 2026 kzhou003

```
        with self._recv_lock:
            out, self._ready = self._ready, {}
        return out

    # cleanup_receiver: called by the scheduler thread after a request is aborted /
    # finished. With its registration token deleted, the recv thread on its next turn will
    # find the entry is None and not re-queue it; an in-flight get is discarded via the token comparison.
    def cleanup_receiver(self, req_id: str) -> None:
        with self._recv_lock:
            self._registered.pop(req_id, None)
            self._recv_chunk.pop(req_id, None)
            self._ready.pop(req_id, None)

    def _recv_loop(self) -> None:
        while not self._stop.is_set():
            # Each round handles only the n reqs in the queue "at the start of this round";
            # those that received nothing are appended back to the tail and retried next round
            n = len(self._load_q)
            progressed = False
            for _ in range(n):
                req_id = self._load_q.popleft()
                # Snapshot (token, expected chunk id) inside the lock, then do the possibly slow get outside it
                with self._recv_lock:
                    entry = self._registered.get(req_id)
                    cid = self._recv_chunk.get(req_id)
                if entry is None or cid is None:
                    continue  # cleaned up -> stop polling it
                # Fetch strictly by cid: chunk cid+1 is never fetched before
                # chunk cid arrives → in-order commit by construction
                obj = self.connector.get(f"{req_id}_{self.src_stage_id}_{cid}")  # outside the lock
                if obj is None:
                    self._load_q.append(req_id)
                    continue
                progressed = True
                with self._recv_lock:
                    # Pitfall: during get the scheduler thread may have
                    # cleaned it up (or even re-registered it via load_async).
                    # Compare tokens with `is`: a new registration is a different object(); stale
                    # data must be dropped, never written into the new generation's _ready.
                    if self._registered.get(req_id) is not entry:
                        continue  # cleaned up (or re-registered) while we were reading: discard
                    self._ready.setdefault(req_id, []).append(obj)
                    self._recv_chunk[req_id] = cid + 1
                    if obj["finished"]:
                        # Got the end-of-stream marker: unregister and do not re-queue
                        # into _load_q (data in _ready is kept for poll_ready)
                        self._registered.pop(req_id, None)
                        self._recv_chunk.pop(req_id, None)
                        continue
                # Got a chunk but not finished → back to the tail, fetch
                # cid+1 next round (round-robin keeps multiple reqs fair)
                self._load_q.append(req_id)
            with self._recv_cond:
                if self._stop.is_set():
                    break
                if not self._load_q:
                    # No reqs to receive: sleep until load_async's notify (or a 0.1s timeout as a fallback)
                    self._recv_cond.wait(timeout=0.1)
                elif not progressed:
                    # Reqs exist but nothing arrived this round: back off briefly for 1ms to avoid spinning at 100% CPU
                    self._recv_cond.wait(timeout=0.001)  # back off instead of spinning
```

C-22 · solutions/p15_talker_inputs.py

```
from dataclasses import dataclass

import torch


# ThinkerPayload: data the thinker stage hands to the talker stage (stop-token row already trimmed).
# prefill rows correspond to prompt tokens; decode rows map
# one-to-one to output_ids (decode.shape[0] == len(output_ids)).
@dataclass
class ThinkerPayload:
    prefill: torch.Tensor  # [P, D]   rows for the prompt
    decode: torch.Tensor   # [K, D]   rows for generated tokens (stop row removed)
    output_ids: list[int]  # generated ids (stop id removed)


# trim_thinker_payload: split the hidden rows the thinker accumulated over prefill + decode into prompt / reply parts,
# and when finish_reason == "stop", drop the last generated token (stop/EOS) and its row.
# E.g. P=3, output_ids=[a, b, EOS], finish="stop" → drop=1, target_rows=3+3-1=5;
#     prefill=hidden[0:3], decode=hidden[3:5] (rows a, b), output_ids=[a, b]
def trim_thinker_payload(
    prompt_ids: list[int],
    output_ids: list[int],
```


© 2026 kzhou003

```
    hidden: torch.Tensor,  # [R, D] accumulated rows over prefill + decode steps
    finish_reason: str,    # "stop" | "length"
) -> ThinkerPayload:
    """vLLM appends the sampled stop token *before* checking stop conditions, so a
    stop-finished request carries one row the talker must not consume."""
    P = len(prompt_ids)
    # If not even all prefill rows were captured, the hidden-capture logic
    # is buggy (e.g. chunked prefill recorded only the last chunk); raise
    if hidden.shape[0] < P:
        raise ValueError(f"under-captured prefill: {hidden.shape[0]} rows < prompt {P}")
    # On a "length" finish the last token is normal content and must
    # not be dropped; with empty output_ids there is nothing to drop
    drop = 1 if (finish_reason == "stop" and output_ids) else 0
    outs = output_ids[:len(output_ids) - drop]
    target_rows = P + len(output_ids) - drop
    # Pitfall: truncate to target_rows rather than blindly "dropping
    # the last row" -- nothing should be dropped on a "length" finish;
    # if R exceeds target_rows (extra rows captured) the excess is cut too. E.g. P=3, [a, b, EOS], R=6 → keep 5 rows
    rows = hidden[:target_rows]
    return ThinkerPayload(prefill=rows[:P], decode=rows[P:], output_ids=list(outs))


# build_talker_prefill: build the Qwen2.5-Omni talker's prefill input (two parallel streams: ids and inputs_embeds).
# Sequence layout (P+2 rows total):
#   [0, P)  prompt rows: the thinker's prompt embedding, id is a mask_id placeholder (talker does not look up by id)
#   P       spk_embed + codec_pad_embed, id = pad_id
#   P+1     first_reply + codec_bos_embed, id = start_id (sampling of the first codec token starts here)
def build_talker_prefill(
    prompt_embeds: torch.Tensor,  # [P, D]
    first_reply: torch.Tensor,    # [D]  thinker row of the first generated token
    spk_embed: torch.Tensor,      # [D]  speaker text-token embedding
    codec_pad_embed: torch.Tensor,
    codec_bos_embed: torch.Tensor,
    *,
    mask_id: int,
    pad_id: int,
    start_id: int,
) -> tuple[list[int], torch.Tensor]:
    """Qwen2.5-Omni talker prompt: [prompt rows] + [spk+pad] + [first_reply+bos]."""
    # [None] turns [D] into [1, D] so it can be concatenated with [P, D] along dim=0 → result [P+2, D]
    # Each position = "text-side" embedding + "codec-side" embedding, added elementwise (dual-stream sum)
    embeds = torch.cat(
        [prompt_embeds, (spk_embed + codec_pad_embed)[None], (first_reply + codec_bos_embed)[None]],
        dim=0,
    )
    # len(ids) == embeds.shape[0] == P + 2
    ids = [mask_id] * prompt_embeds.shape[0] + [pad_id, start_id]
    return ids, embeds


# talker_decode_input: the talker's input at each decode step = the next
# thinker reply hidden row + the codec embedding sampled at the previous step.
# Thinker rows form a FIFO queue consumed one row per step; once exhausted
# (the talker is usually longer than the text), use fallback (e.g. tts_pad).
# Returns (this step's input [D], remaining queue [K-1, D] or the empty queue unchanged).
def talker_decode_input(
    reply_queue: torch.Tensor,     # [K, D] remaining thinker rows
    last_codec_embed: torch.Tensor,  # [D] embedding of the codec id sampled last step
    fallback: torch.Tensor,        # [D] used once thinker rows are exhausted (e.g. tts_pad)
) -> tuple[torch.Tensor, torch.Tensor]:
    if reply_queue.shape[0] > 0:
        # reply_queue[0] is [D] (not [1, D]); rest is a [K-1, D] view
        step, rest = reply_queue[0], reply_queue[1:]
    else:
        # Queue already empty: fallback can be used any number of times; the queue stays [0, D]
        step, rest = fallback, reply_queue
    return step + last_codec_embed, rest
```

C-23 · solutions/p14_codec_windows.py

```
from collections.abc import Sequence


# next_window: in streaming TTS, decide "when to send the next block of codec frames, and which span, to the vocoder".
# window = [ctx_start, new_start) left context (only lets the vocoder join smoothly; its output is discarded)
#       + [new_start, new_end)  the frames actually new this time.
# The first block uses a smaller initial (lower first-packet latency), then
# chunk frames per block; at end of stream the partial tail is flushed.
# E.g. initial=2, chunk=5, left_ctx=3
#   (available=2,  emitted=0)          → (0, 0, 2)    first block, 2 frames
#   (available=12, emitted=7)          → (4, 7, 12)   left context 4,5,6
#   (available=20, emitted=17, finished) → (14, 17, 20) 3-frame tail flushed
def next_window(
    available: int,
    emitted: int,
    finished: bool,
```


© 2026 kzhou003

```
    *,
    initial: int,
    chunk: int,
    left_ctx: int,
) -> tuple[int, int, int] | None:
    """Decide the next codec window to ship downstream.

    available: frames produced so far by the talker
    emitted:   frames already shipped (end of the previous window)
    returns (ctx_start, new_start, new_end) or None if we should wait for more frames.
    """
    # Use initial only for the "first block" (emitted == 0) when
    # initial is configured; initial=0 means no special first block
    size = initial if (emitted == 0 and initial > 0) else chunk
    if available - emitted >= size:
        # A full block is available: cut by whole blocks even if
        # finished; never merge several blocks into one big window
        new_end = emitted + size
    elif finished and available > emitted:
        # Finished with unsent frames left (less than a block) → send
        # them all. E.g. a very short stream available=1 → (0, 0, 1)
        new_end = available  # flush the tail
    else:
        # Not enough frames and not finished (wait for more), or finished with available == emitted (all sent)
        return None
    new_start = emitted
    # Left context clipped at 0: e.g. new_start=2, left_ctx=3 → ctx_start=0 (only 2 frames of context)
    ctx_start = max(0, new_start - left_ctx)
    return ctx_start, new_start, new_end


# slice_new_samples: after the vocoder decodes codes[ctx_start:new_end], each frame maps to upsample waveform samples;
# drop the leading samples belonging to context frames and keep only the new frames'
# samples; concatenated, they form continuous audio with no overlap and no gaps.
# E.g. window=(4, 7, 12), upsample=2 → decoded length (12-4)*2=16;
#     context frames 4..6 occupy samples [0, 6), new frames 7..11 occupy samples [6, 16) → return decoded[6:16]
def slice_new_samples(decoded: Sequence[float], window: tuple[int, int, int], upsample: int) -> list[float]:
    """decoded = vocoder(codes[ctx_start:new_end]); keep only the samples of the new frames."""
    ctx_start, new_start, new_end = window
    # Pitfall: decoded is indexed relative to ctx_start (not absolute frame numbers), so subtract ctx_start below
    assert len(decoded) == (new_end - ctx_start) * upsample
    return list(decoded[(new_start - ctx_start) * upsample:(new_end - ctx_start) * upsample])
```

C-24 · solutions/p18_mrope_positions.py

```
import torch

# Segment: ("text", token count) or ("image", (t, h, w)); image t/h/w are in patches (before merge)
Segment = tuple[str, int | tuple[int, int, int]]  # ("text", n) | ("image", (t, h, w)) in patches


# mrope_positions: build Qwen2-VL-style 3-D MRoPE positions [3, T] for
# a prefill sequence (axis 0 = time t, 1 = height h, 2 = width w).
#   - text: all three axes take the same increasing index (degenerates to plain 1-D RoPE)
#   - image: LLM grid (t, h//merge, w//merge); the three axes are the grid coordinates + segment start start
#   - after each segment, start = this segment's max position + 1 (not "number of tokens so far")
# Because an image segment's position span is usually smaller than its token
# count, final max position + 1 < T; the difference delta is used in decode:
#   decode position = absolute sequence index + delta (same on all three axes)
def mrope_positions(segments: list[Segment], spatial_merge: int = 2) -> tuple[torch.Tensor, int]:
    """Qwen2-VL style 3-D rotary positions (temporal, height, width).

    text  : the same running index on all three axes
    image : an LLM grid of (t, h//merge, w//merge) tokens, offset by the running start
    after every segment the next start is max(position so far) + 1
    returns (positions [3, T], mrope_position_delta) with decode position = index + delta
    """
    parts: list[torch.Tensor] = []  # one [3, n_seg] tensor per segment, concatenated along dim=1 at the end
    start = 0  # start position of the next segment
    for kind, spec in segments:
        if kind == "text":
            n = int(spec)  # type: ignore[arg-type]
            idx = torch.arange(n) + start
            # [n] → expand to [3, n]: three identical rows (expand is a view; cat makes the actual copy)
            parts.append(idx.expand(3, n))
        elif kind == "image":
            t, h, w = spec  # type: ignore[misc]
            # The vision encoder merges merge×merge patches into one LLM
            # token; e.g. (2, 4, 6), merge=2 → grid (2, 2, 3) = 12 tokens
            gh, gw = h // spatial_merge, w // spatial_merge
            # Tokens are laid out row-major with t outermost, then h,
            # w innermost; each coordinate is flattened to [t*gh*gw]
            # E.g. (t, gh, gw) = (2, 2, 3):
            #   ti = 0 0 0 0 0 0 1 1 1 1 1 1
            #   hi = 0 0 0 1 1 1 0 0 0 1 1 1
            #   wi = 0 1 2 0 1 2 0 1 2 0 1 2
```


© 2026 kzhou003

```
            ti = torch.arange(t).view(-1, 1).expand(t, gh * gw).flatten()
            hi = torch.arange(gh).view(1, -1, 1).expand(t, gh, gw).flatten()
            wi = torch.arange(gw).view(1, 1, -1).expand(t, gh, gw).flatten()
            # [3, t*gh*gw]; add the segment start to all three axes
            parts.append(torch.stack([ti, hi, wi]) + start)
        else:
            raise ValueError(kind)
        # Pitfall: the next segment's start is this segment's max
        # position + 1 (max over three axes), not start + token count.
        # E.g. the 12-token image segment above has max=2 → next segment
        # start=3. Empty segments (n=0 or a 0 grid) do not advance start
        if parts[-1].numel():
            start = int(parts[-1].max()) + 1
    # With no segments, return a [3, 0] long tensor to keep shape/dtype consistent
    pos = torch.cat(parts, dim=1) if parts else torch.zeros(3, 0, dtype=torch.long)
    # delta = (next position after prefill) − T. E.g. image (2,4,6) + text 2 → T=14, max=4 → delta=5−14=−9
    # For pure text max+1 == T → delta=0; delta=0 does not mean no image,
    # only that the position span happens to equal the token count
    delta = (int(pos.max()) + 1 if pos.numel() else 0) − pos.shape[1]
    return pos, delta


# decode_position: in decode every new token is text; all three axes share position = index + delta.
# E.g. continuing the example T=14, delta=−9 → the first decode
# token (index=14) is at [5, 5, 5], right after prefill's max=4
def decode_position(index: int, delta: int) -> torch.Tensor:
    """Position (3,) of the token at absolute sequence index `index` during decode."""
    return torch.full((3,), index + delta, dtype=torch.long)
```

C-25 · solutions/p16_init_planner.py

```
import threading
from collections.abc import Callable
from concurrent.futures import ThreadPoolExecutor
from typing import Any


# InitError: carries "the first failing stage_id"; the original exception hangs off __cause__ via `raise ... from cause`
class InitError(Exception):
    def __init__(self, stage_id: int, cause: BaseException) -> None:
        super().__init__(f"stage {stage_id} failed to initialize: {cause!r}")
        self.stage_id = stage_id


# init_groups: step one of multi-stage startup planning —— group "stages that share a GPU" together.
# Stages on the same card must initialize serially (memory profiling
# / allocation interfere); different groups can run in parallel.
# Sharing is transitive (A–B share dev0, B–C share dev1 → A, B,
# C in one group); union-find gives the connected components.
# E.g. {0: {2}, 1: {0}, 2: {0, 1}, 3: {1}, 4: {5}} → [[0], [1, 2, 3], [4]]
# Deterministic output: stages ascending within a group, groups ordered by first element ascending.
def init_groups(stage_devices: dict[int, set[int]]) -> list[list[int]]:
    """Stages that share (transitively) a device form one group -> must init sequentially."""
    parent = {s: s for s in stage_devices}

    def find(x: int) -> int:
        while parent[x] != x:
            # Path halving: hang x on its grandparent; amortized, the tree stays very shallow
            parent[x] = parent[parent[x]]
            x = parent[x]
        return x

    # owner: device → the first stage that occupies it; a later
    # stage hitting an occupied device is merged with the owner
    owner: dict[int, int] = {}
    # Iterate stages in ascending order: input dict key order may be
    # arbitrary; sorting makes the result independent of insertion order
    for s in sorted(stage_devices):
        for d in stage_devices[s]:
            if d in owner:
                a, b = find(s), find(owner[d])
                if a != b:
                    # Always make the smaller root the parent → each
                    # component's root = smallest stage id in the component
                    parent[max(a, b)] = min(a, b)
            else:
                owner[d] = s
    groups: dict[int, list[int]] = {}
    # Ascending iteration then append → each group is naturally sorted
    for s in sorted(stage_devices):
        groups.setdefault(find(s), []).append(s)
    return sorted(groups.values(), key=lambda g: g[0])


# run_init: initialize by group concurrently (parallel across groups, serial within a group); if any stage fails:
#   1) short-circuit: other groups see the failure before starting their next stage
```


© 2026 kzhou003

```
#      and stop (an init already running cannot be interrupted and runs to completion)
#   2) roll back: after all threads finish, call cleanup_fn on every stage that succeeded (free GPU memory/processes)
#   3) raise InitError whose stage_id is the "first" failure recorded
def run_init(
    stage_devices: dict[int, set[int]],
    init_fn: Callable[[int], Any],
    cleanup_fn: Callable[[int, Any], None],
    max_workers: int = 4,
) -> dict[int, Any]:
    groups = init_groups(stage_devices)
    # lock protects results and failure (read and written concurrently by multiple worker threads)
    lock = threading.Lock()
    results: dict[int, Any] = {}
    failure: list[tuple[int, BaseException]] = []  # at most one element: the first failure

    # One task per group; the for loop within a group is naturally serial
    def run_group(group: list[int]) -> None:
        for sid in group:
            with lock:
                if failure:
                    return  # short-circuit: do not start new work after a failure
            # init_fn can be slow (loading weights) and must be called
            # outside the lock, or groups lose their parallelism
            try:
                res = init_fn(sid)
            except BaseException as e:  # noqa: BLE001
                # Catch BaseException: KeyboardInterrupt etc. must also
                # trigger rollback; already-started stages must not leak
                with lock:
                    if not failure:
                        failure.append((sid, e))
                # Later stages in this group are not initialized
                return
            with lock:
                results[sid] = res

    with ThreadPoolExecutor(max_workers=max_workers, thread_name_prefix="stage-init") as ex:
        # list(...) forces the iterator to be consumed → waits for all groups;
        # run_group swallows its own exceptions, so nothing is raised here
        list(ex.map(run_group, groups))

    # Pitfall: roll back only after the thread pool exits (all inits finished),
    # otherwise a stage still initializing that succeeds later may be missed
    if failure:
        for sid, res in results.items():
            cleanup_fn(sid, res)  # roll back everything that did come up
        sid, cause = failure[0]
        raise InitError(sid, cause) from cause
    return results


# admission_check: pre-launch GPU memory admission check (read-only, no allocation). For each card:
#   need = Σ_{stages on this card} ( int(gpu_memory_utilization × total) + reserve_bytes_per_stage )
#   need > total is an error; exactly equal is allowed. A
#   cross-card stage (e.g. TP=2) counts once on every card it uses.
# E.g. (80 GiB card, reserve=2 GiB): two stages on dev1 with util=0.6/0.3 → (48+2)+(24+2)=76 ≤ 80 → passes
def admission_check(
    device_total_bytes: dict[int, int],
    stages: dict[int, tuple[set[int], float]],  # stage -> (devices, gpu_memory_utilization)
    reserve_bytes_per_stage: int,
) -> list[str]:
    errors = []
    # Output in ascending device order so error order is deterministic
    for dev, total in sorted(device_total_bytes.items()):
        on_dev = [s for s, (devs, _) in stages.items() if dev in devs]
        # Pitfall: utilization is a fraction of "this card's total", so
        # multiply by this card's total; int() truncates down to whole bytes
        need = sum(int(stages[s][1] * total) + reserve_bytes_per_stage for s in on_dev)
        if need > total:
            errors.append(f"device {dev}: stages {sorted(on_dev)} need {need} B > {total} B")
    return errors
```

C-26 · solutions/p17_graph_runner.py

```
from collections.abc import Callable

import torch


# ---------------------------------------------------------------- given types
# CpuGraph: simulates the key semantics of a CUDA graph on CPU -- capture binds static input/output buffers,
# and replay only reruns on "the same" buffers; it cannot see new Python arguments.
class CpuGraph:
    """Emulates torch.cuda.CUDAGraph semantics on CPU: replay() re-runs the recorded work on
    the *same static buffers*; it never sees new Python arguments."""
```


© 2026 kzhou003

```
    def __init__(self, fn: Callable[[torch.Tensor], torch.Tensor], static_in: torch.Tensor, static_out: torch.Tensor):
        self._fn, self._in, self._out = fn, static_in, static_out
        self.captured = False

    def capture(self) -> None:
        self._out.copy_(self._fn(self._in))
        self.captured = True

    def replay(self) -> None:
        assert self.captured
        self._out.copy_(self._fn(self._in))


# ------------------------------------------------------------ implementation
# GraphRunner: bucketed CUDA graph executor for the decode phase.
#   - capture one graph per bucket size b; all graphs share the first
#     b rows (a view) of one pair of [max_bs, hidden] static buffers
#   - at run time pick the smallest bucket >= bs, copy the input into
#     static_in[:bs], zero the padding rows, replay, then copy out the first bs rows
#   - bs larger than the largest bucket, or force_eager (e.g. prefill / debugging), goes eager
# E.g. buckets=[1, 4, 16], bs=3 → bucket 4 (the 4th row is padding); bs=17 → eager
class GraphRunner:
    def __init__(self, fn: Callable[[torch.Tensor], torch.Tensor], buckets: list[int], hidden: int,
                 dtype: torch.dtype = torch.float32) -> None:
        self.fn = fn
        # After sorting, the first hit in _bucket_for's linear scan is the smallest usable bucket
        self.buckets = sorted(buckets)
        self.max_bs = self.buckets[-1]
        # Static buffers are allocated once for the largest bucket; shape [max_bs, hidden]
        self.static_in = torch.zeros(self.max_bs, hidden, dtype=dtype)
        self.static_out = torch.zeros(self.max_bs, hidden, dtype=dtype)
        self.graphs: dict[int, CpuGraph] = {}
        self.stats = {"replays": 0, "eager": 0, "last_bucket": None}

    def capture(self) -> None:
        # Capture from largest to smallest: on CUDA the graphs share a memory pool; capture
        # the largest first so smaller graphs can reuse the memory it already reserved
        for b in sorted(self.buckets, reverse=True):  # largest first (shares a memory pool on CUDA)
            # static_in[:b] is a view (shared storage), not a copy → later
            # writes to static_in's first bs rows are "seen" by the graph
            g = CpuGraph(self.fn, self.static_in[:b], self.static_out[:b])
            g.capture()
            self.graphs[b] = g

    def _bucket_for(self, bs: int) -> int | None:
        # Round up to the smallest bucket; e.g. buckets=[1, 4, 16]: bs=1→1, bs=2..4→4, bs=5..16→16, bs=17→None
        for b in self.buckets:
            if bs <= b:
                return b
        return None

    def run(self, x: torch.Tensor, *, force_eager: bool = False) -> torch.Tensor:
        bs = x.shape[0]
        b = None if force_eager else self._bucket_for(bs)
        # `b not in self.graphs`: calling run before capture also safely falls back to eager
        if b is None or b not in self.graphs:
            self.stats["eager"] += 1
            return self.fn(x)
        # Pitfall: you must copy_ into the static buffer, not
        # self.static_in = x (the graph is bound to the original memory)
        self.static_in[:bs].copy_(x)
        # Pitfall: padding rows [bs, b) may hold leftover data from an earlier larger
        # batch; zeroing them keeps results deterministic and history-independent.
        # E.g. last time bs=10 wrote 10 rows, now bs=3, bucket=4 → the 4th row
        # (index 3) is zeroed; index 4.. is outside the bucket and can be ignored
        self.static_in[bs:b].zero_()  # stale rows from a previous, larger batch
        self.graphs[b].replay()
        self.stats["replays"] += 1
        self.stats["last_bucket"] = b
        # Pitfall: must clone. Returning the static_out[:bs] view
        # lets the next replay silently overwrite the caller's result
        return self.static_out[:bs].clone()  # next replay overwrites static_out
```

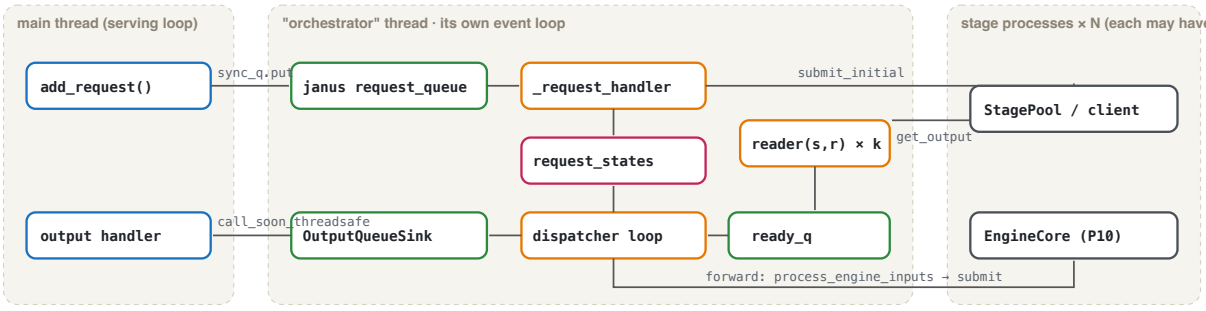

© 2026 kzhou003

Appendix D · P02 Orchestrator ↔ vllm-omni source mapping

Exercise: `mine/p02_orchestrator.py` · Source: vllm-project/vllm-omni @ fa8b419 · vllm_omni/engine/orchestrator.py (all links are pinned to this commit, so line numbers won't drift)

Bottom line

P02's skeleton is **identical** to the real code: `_request_handler` handles control messages, **one reader per stage replica** fans into a single `ready_q`, **one dispatcher** routes serially, the terminal message is decided by `final_output_stage_ids ⊆ finished`, and state is cleaned up after it. The biggest difference is in **T (execution topology)**: the real orchestrator runs in a **separate thread** and talks to the main thread through janus queues and `OutputQueueSink` (P01).



Item-by-item mapping

P02	vllm-omni	Source
`class Orchestrator`	`OrchestratorBase` (generic skeleton) + `Orchestrator(OrchestratorBase)` (turn-based subclass that handles add)	orchestrator.py:322 orchestrator.py:3033
`ReqState`	`OrchestratorRequestState`: `final_output_stage_ids`, `finished_final_output_stage_ids`, `finished_stage_ids`, `pending_final_output`	orchestrator.py:233-272
`request_states`	`self.request_states: dict[str, OrchestratorRequestState]`	orchestrator.py:390
`_req_q`	`request_async_queue`: a **janus** queue; the main thread does `sync_q.put`, the orchestrator thread does `async_q.get`	omni_engine_base.py:308 add_request → sync_q.put
`_out_q`	`output_async_queue` = `OutputQueueSink`: one `call_soon_threadsafe` wakeup per batch of outputs (**this is P01**)	omni_engine_base.py:309-310 async_engine_utils.py:40-50
`start()`	`run()`: create_task the request handler and output handler, then gather	orchestrator.py:479-512
`shutdown()`	`run()`'s `finally`: cancel + `gather(*tasks, return_exceptions=True)`, then a final sweep over `all_tasks`	orchestrator.py:530-555
`_request_handler`	`_request_handler`: dispatches add / abort / collective_rpc / register / shutdown by `msg.type`	orchestrator.py:557-592

© 2026 kzhou003

`_handle_add`	`_handle_add_request` : **first** `self.request_states[request_id] = req_state` , **then** `submit_initial`	orchestrator.py:3062 register :3117 · submit :3128
`_reader(sid)`	`_llm_replica_reader(stage_id, replica_id, client)` : one reader **per (stage, replica)**; errors are also handed to the dispatcher as `("error", …)` messages	orchestrator.py:1117-1136
`_ready_q`	`ready_q: asyncio.Queue[(kind, stage_id, replica_id, payload)]`	orchestrator.py:1112
`_dispatcher`	The main loop of `_orchestration_loop_event_driven` : consumes `ready_q` serially	docstring main loop :1264
`_handle_output`	`_handle_processed_outputs` → `_route_output`	orchestrator.py:1381-1412 orchestrator.py:1912-2043
Unknown id: `dropped_outputs += 1`	`req_state is None` : log a warning and `continue` (no counter)	orchestrator.py:1385-1393
Terminal condition `final_stage_ids ⊆ finished`	`request_finished = final_output_stage_ids.issubset(req_state.finished_final_output_stage_ids)`	orchestrator.py:1936-1940
Deliver to user `_out_q.put(OrchOutput)`	`OutputMessage(…, finished=request_finished)` → `output_async_queue.put`	orchestrator.py:1944-1967
Forward `forward_fn(s+1, prompt, data)`	`_forward_to_next_stage` → `next_client.process_engine_inputs(source_outputs, req_state.prompt)` → `custom_process_input_func` (e.g. `thinker2talker_full_payload`)	trigger condition :1988 process_engine_inputs :2602 stage_engine_core_client.py:455-471 thinker2talker_full_payload
`_cleanup(rid)`	`_cleanup_request_ids` : release bindings and transfer resources, `self.request_states.pop(...)`	orchestrator.py:1699-1762 pop :1756
`pending_final` (P03)	`pending_final_output` : under async_chunk, hold back the terminal message and release it only once every submitted stage has finished	orchestrator.py:1954-1965 orchestrator.py:2036-2040
abort (P04)	`_handle_abort` → `_cleanup_request_ids(abort=True)` → `_abort_request_ids` calls `abort_requests` on each pool	orchestrator.py:645-683 orchestrator.py:685-731 stage_pool.py:1279
dead replica (P04)	reader sends `("error", …EngineDeadError)` → dispatcher `_handle_dead_replica` evicts it and keeps serving	orchestrator.py:1297-1312 orchestrator.py:1464

© 2026 kzhou003

Suggested reading path (about 300 lines)

① The thread / event-loop boundary (T)

The main thread (HTTP / serving loop) calls `add_request()`, which does `sync_q.put` of the message into a janus queue; a background thread named `"orchestrator"` calls `new_event_loop()` and runs `orchestrator.run()` there. **This is the layer P02 spares you**: in P02 the caller and the orchestrator share one event loop.

omni_engine_base.py:303-330 · janus queues + orchestrator thread

```
303  # Janus queues are constructed eagerly here (not deferred to the
304  # orchestrator thread) so the master server's ROUTER thread always
305  # sees a non-None ``self.request_queue`` when on_register fires.
306  # ``async_q`` lazily binds to whatever event loop first awaits on
307  # it (the orchestrator loop), so cross-thread use stays correct.
308  self.request_queue: janus.Queue[EngineQueueMessage] = janus.Queue(maxsize=_REQUEST_QUEUE_MAXSIZE)
309  self.output_queue: janus.Queue[EngineQueueMessage] = janus.Queue()
310  self._output_sink = OutputQueueSink(self.output_queue.sync_q)
311  self.rpc_output_queue: janus.Queue[EngineQueueMessage] = janus.Queue()
312  self._shutdown_called = False
313  self._weak_finalizer: weakref.finalize | None = None
314  self._correlated_rpc_client: CorrelatedRpcClient | None = None
315  self._running_counter = OmniRequestCounter()
316  self._engines_waiting_counter = OmniRequestCounter()
317
318  logger.info(f"[OmniEngine] Launching Orchestrator thread with {self.num_stages} stages")
319
320  # Launch orchestrator background thread
321  startup_future: concurrent.futures.Future = concurrent.futures.Future()
322
323  self.orchestrator_thread = threading.Thread(
324      target=self._bootstrap_orchestrator,
325      args=(
326          stage_init_timeout,
327          startup_future,
328      ),
329      daemon=True,
330      name="orchestrator",
```

omni_engine_base.py:440-452 · create a new event loop inside the thread

```
440  def _bootstrap_orchestrator(
441      self,
442      stage_init_timeout: int,
443      startup_future: concurrent.futures.Future,
444  ) -> None:
445      """Create loop, initialize stages, then run Orchestrator."""
446
447      loop = asyncio.new_event_loop()
448      asyncio.set_event_loop(loop)
449
450      async def _run_orchestrator() -> None:
451          self._initialize_stages(stage_init_timeout)
452
```


© 2026 kzhou003

② Register first, then submit

Find the order of these two statements in `_handle_add_request`: registering in `request_states` happens before any `await`.

orchestrator.py:3107-3138

```
3107 req_state = OrchestratorRequestState(
3108     request_id=request_id,
3109     prompt=original_prompt,
3110     sampling_params_list=sampling_params_list,
3111     final_stage_id=final_stage_id,
3112     final_output_stage_ids=final_output_stage_ids,
3113     request_timestamp=float(msg.request_timestamp or _time.time()),
3114     mm_features=getattr(prompt, "mm_features", None),
3115     request_artifact_dirs=set(msg.request_artifact_dirs or ()),
3116 )
3117 self.request_states[request_id] = req_state
3118 self._maybe_attach_native_kv_transfer_params(req_state, prompt)
3119 self._register_running_request(req_state)
3120 req_state.streaming.enabled = bool(getattr(prompt, "resumable", False))
3121 req_state.stage_submit_ts[stage_id] = _time.time()
3122 enqueue_ts = msg.enqueue_ts
3123 if enqueue_ts > 0:
3124     req_state.pipeline_timings["queue_wait_ms"] = (_time.perf_counter() - enqueue_ts) * 1000.0
3125 preprocess_ms = msg.preprocess_ms
3126 if preprocess_ms > 0:
3127     req_state.pipeline_timings["preprocess_ms"] = preprocess_ms
3128 if not await self._dispatch_or_fail_request(
3129     lambda: self.stage_pools[stage_id].submit_initial(
3130         request_id,
3131         req_state,
3132         prompt,
3133         prompt_text=msg.output_prompt_text,
3134     ),
3135     req_id=request_id,
3136     stage_id=stage_id,
3137     operation="add_request",
3138 ):
```

③ reader: only moves things, even errors

Note the `except BaseException`: when the reader itself fails it **modifies no state**; it wraps the error in a message and hands it to the dispatcher. This extends the single-writer principle.

orchestrator.py:1117-1136

```
1117 async def _llm_replica_reader(stage_id: int, replica_id: int, client: Any) -> None:
1118     try:
1119         pool = self.stage_pools[stage_id]
1120         while not self._shutdown_event.is_set():
1121             # Read through the pool so this loop shares the legacy
1122             # poll's keep/drop rule and MR V2 payload rehydration. Only
1123             # a fully empty batch comes back as None. A poll-style
1124             # client returns one instead of blocking, so keep the
1125             # legacy 1 ms cadence for those; a blocking client only
1126             # lands here rarely, where 1 ms is irrelevant.
1127             raw_outputs = await pool.read_llm_raw_output(client)
1128             if raw_outputs is None:
1129                 await asyncio.sleep(0.001)
1130                 continue
1131             await ready_q.put(("llm", stage_id, replica_id, raw_outputs))
1132             self._orch_monitor.set_dispatch_queue_size(ready_q.qsize())
1133     except asyncio.CancelledError:
1134         raise
1135     except BaseException as e:  # noqa: BLE001 - routed to the dispatcher
1136         await ready_q.put(("error", stage_id, replica_id, e))
```


© 2026 kzhou003

④ dispatcher: consumes ready_q serially

The real code waits on three things at once: new outputs, the shutdown signal, and replica changes. Hence `asyncio.wait(FIRST_COMPLETED)`, rebuilding `pending_get` each round. This is exactly the "manage the tasks yourself with wait" pattern we discussed, used only in this one place, the dispatcher.

orchestrator.py:1264-1272

```
1264 while not self._shutdown_event.is_set():
1265     if pending_get is None:
1266         pending_get = asyncio.create_task(ready_q.get(), name="orch-dispatch-get")
1267     done, _ = await asyncio.wait(
1268         {pending_get, shutdown_task, reconcile_task},
1269         timeout=max(0.0, next_reconcile - _time.monotonic()),
1270         return_when=asyncio.FIRST_COMPLETED,
1271     )
1272     if shutdown_task in done:
```

orchestrator.py:1291-1293

```
1291 kind, stage_id, replica_id, payload = pending_get.result()
1292 pending_get = None
1293 # Record the queue after dequeue so the gauge reflects work
```


© 2026 kzhou003

⑤ _route_output: the real P02 _handle_output

Order: mark finished → compute `request_finished` → if it is a final stage, send `OutputMessage` → if `finished and stage_id < final_stage_id`, forward → if `request_finished`, clean up.

orchestrator.py:1933-1967 · terminal check + deliver to user

```
1933 if finished and not segment_finished:
1934     req_state.finished_stage_ids.add(stage_id)
1935
1936 request_finished = False
1937 if finished and self.stage_pools[stage_id].final_output and not segment_finished:
1938     req_state.finished_final_output_stage_ids.add(stage_id)
1939     final_output_stage_ids = req_state.final_output_stage_ids or {req_state.final_stage_id}
1940     request_finished = final_output_stage_ids.issubset(req_state.finished_final_output_stage_ids)
1941 if req_state.session_owned:
1942     # Session-owned outputs are delivered through the session interceptor.
1943     pass
1944 elif self.stage_pools[stage_id].final_output:
1945     message = OutputMessage(
1946         request_id=req_id,
1947         stage_id=stage_id,
1948         replica_id=replica_id,
1949         engine_outputs=output,
1950         metrics=stage_metrics,
1951         finished=request_finished,
1952         stage_submit_ts=submit_ts,
1953     )
1954     if (
1955         request_finished
1956         and self.async_chunk
1957         and not req_state.streaming.enabled
1958         and not set(req_state.stage_submit_ts).issubset(req_state.finished_stage_ids)
1959     ):
1960         # Direct chunk delivery can finish the decoder before the
1961         # upstream engine's terminal output/metrics reaches this loop.
1962         # Keep request state until those messages are routed, otherwise
1963         # the frontend publishes incomplete usage and loses stop reasons.
1964         req_state.pending_final_output = message
1965         request_finished = False
1966     else:
1967         await self.output_async_queue.put(message)
```

orchestrator.py:1988-1993 · forwarding condition

```
1988 if (
1989     (finished or segment_finished)
1990     and stage_id < req_state.final_stage_id
1991     and not req_state.skip_legacy_stage_forward
1992     and (not self.async_chunk or not self._stage_receives_async_chunks(stage_id + 1))
1993     and (not self._next_stage_already_submitted(stage_id, req_state) or req_state.streaming.enabled)
```

orchestrator.py:2036-2043 · release pending_final + clean up

```
2036 pending = req_state.pending_final_output
2037 if pending is not None and set(req_state.stage_submit_ts).issubset(req_state.finished_stage_ids):
2038     req_state.pending_final_output = None
2039     await self.output_async_queue.put(pending)
2040     request_finished = True
2041
2042 if request_finished and not req_state.session_owned:
2043     await self._cleanup_request_ids([req_id, *self._cfg_tracker.cleanup_parent(req_id)])
```


© 2026 kzhou003

⑥ run()'s finally: cancel + gather

The shutdown pattern we discussed is right here, plus a final safety net: sweep `asyncio.all_tasks(loop)` and cancel and await any task that slipped through.

orchestrator.py:530-555

```
530      finally:
531          self._shutdown_event.set()
532          for task in tasks:
533              if not task.done():
534                  task.cancel()
535          try:
536              await asyncio.gather(*tasks, return_exceptions=True)
537          except Exception:
538              pass
539          await self._shutdown_extensions()
540
541          if self._membership is not None:
542              await self._membership.drain_tasks(timeout=10.0)
543              self._membership.shutdown()
544
545          self._orch_monitor.flush()
546          self._shutdown_stages()
547
548          loop = asyncio.get_running_loop()
549          pending = [t for t in asyncio.all_tasks(loop) if t is not asyncio.current_task() and not
t.done()]
550          for task in pending:
551              task.cancel()
552          if pending:
553              await asyncio.gather(*pending, return_exceptions=True)
554
555  # ---- Request handling ----
```

What the real code has beyond P02

Extra	Why it's needed	Problem	Source
Multiple **replicas** per stage; `_reconcile_readers` adds/removes readers dynamically	Replicas register, die, and get replaced	P04	orchestrator.py:1159-1233
dispatcher uses `asyncio.wait({ready_q.get(), shutdown, reconcile})`	Besides outputs, it must react to shutdown and membership changes	—	orchestrator.py:1264-1289
On error the reader sends `("error", …)`; the dispatcher evicts that replica and continues	Fault isolation: one dead replica doesn't take down the whole system	P04	orchestrator.py:1297-1312
Forwarding uses `source_outputs = [output]` and doesn't accumulate chunks itself	The stage's output processor has already accumulated the output; P02 simulates this with `stage_data`	—	orchestrator.py:2366
The orchestrator runs in its own thread and talks to the main thread via janus and `OutputQueueSink`	Keeps routing logic from slowing down the main HTTP loop	P01	async_engine_utils.py:40-50

© 2026 kzhou003

diffusion stages are polled by a poller	The diffusion client only has a non-blocking poll interface	—	orchestrator.py:1138-1157
CFG companion, PD disaggregation, streaming, session, metrics	Product features	—	orchestrator.py:1912-1931

Questions to think about afterwards

1. The real orchestrator lives in **another thread**. When the main thread calls `add_request`, why must it go through janus's `sync_q.put` instead of directly calling `asyncio.Queue.put_nowait` on a queue in the orchestrator loop?
2. In `_route_output`, the user-facing `OutputMessage` is enqueued **before** forwarding to the next stage. If it were reversed, first `await _forward_to_next_stage` (a cross-process round trip) and then deliver to the user, what difference would the user perceive?
3. When a reader fails, why doesn't it call `_handle_dead_replica` directly, instead wrapping the error in a message for the dispatcher?

© 2026 kzhou003

PART II

Project: build nano-vllm-omni from scratch

Then put them together: build a complete multi-stage inference service for Qwen2.5-Omni-3B on a single RTX 5090. 12 milestones, each with acceptance criteria and points (100 total, plus bonus items).

© 2026 kzhou003

Capstone: Building nano-vllm-omni from Scratch

Goal: serve **Qwen2.5-Omni-3B** on a **single RTX 5090 (32 GB)** with text, image, and audio input, streaming text and speech output, concurrent requests, and an OpenAI-compatible API.
Reference implementations: vllm-omni `fa8b4199c` (source links point to this commit), vLLM v0.31.0, nano-vllm, HF transformers.

---

0. Assignment and final deliverables

The task is not "call vLLM" but to **implement a minimal yet complete multi-stage inference server yourself**. Along the way you will hit every core problem vllm-omni has solved: paged KV cache and continuous batching; passing data across stages; pipelined streaming output; correctness under multi-process concurrency; splitting GPU memory among several stages on one card; CUDA graphs.

Final deliverables

1. `nano_vllm_omni/` source. Aim for 4k–7k lines (excluding borrowed HF modules).
2. `nvo serve Qwen/Qwen2.5-Omni-3B`: starts an OpenAI-compatible server; `/v1/chat/completions` supports `modalities=["text","audio"]` and `stream=true`.
3. `tests/`: parity and concurrency tests for each milestone (see each milestone's "Acceptance" section).
4. `bench/`: reports TTFT, TPOT, TTFP (time to first audio packet), RTF, and throughput at concurrency 1, 4, and 8.
5. `REPORT.md`: describes the architecture, design trade-offs, memory budget, performance curves, and how you differ from vllm-omni.

> **How to use this document**
> Each milestone is organized as "Goal → What you write yourself → Interface skeleton → Acceptance criteria → Hints (collapsed) → Reference source". **Try not to expand the hints at first**: design it, write it, and pass acceptance on your own, then expand the hints and compare against the reference source. That is the only way to really test yourself.

---

1. Target model and hardware budget

1.1 Why Qwen2.5-Omni-3B rather than Qwen3-Omni

Model	Structure	Feasibility on a single 5090
Qwen3-Omni-30B-A3B	MoE thinker + MoE talker + 16-layer RVQ MTP + Code2Wav	bf16 weights exceed ~60 GB, **does not fit**. Still very tight after quantization; not suitable for this assignment.
Qwen2.5-Omni-3B ✅	Dense thinker (Qwen2.5-3B-class LLM + audio encoder + ViT) + small talker (single codebook) + Token2Wav (DiT flow-matching + BigVGAN)	bf16 weights are roughly a dozen-plus GB; the exact figure is `metadata.total_size` in `model.safetensors.index.json`. All three stages fit on one card with plenty of room left for KV.

© 2026 kzhou003

Qwen2.5-Omni-7B	Same as above, larger thinker	Bonus: fits in bf16 but tightly; requires a careful memory budget, or FP8 for the thinker.

It shares Qwen3-Omni's architectural idea (Thinker–Talker plus vocoder), so the concurrency design you learn transfers directly. The corresponding vllm-omni implementation: `omni(/) model_executor/models/qwen2_5_omni/pipeline.py:18-65`.

1.2 The model's three stages (read the HF reference implementation first)

Stage	Input	Output	Key points
0 Thinker	Text tokens + image/video/audio features	Text tokens (returned to the user); per-position embedding and hidden (for the talker)	MRoPE / TMRoPE (time-aligned audio/video positions); multimodal embedding merge
1 Talker	Thinker prompt embedding and hidden, plus per-step reply vectors	Codec tokens (single codebook). `START=8293`, `END=8294` (stop token), `PAD=8292`	`thinker_to_talker_proj` projects thinker dim to talker dim; per-step input = codec embedding + this step's thinker reply vector; tokens with id ≥ num_embeds+1 must be suppressed
2 Token2Wav	Codec tokens, speaker condition (cond / ref_mel, from `spk_dict.pt`, e.g. Chelsie / Ethan)	24k waveform (per config)	DiT solves flow matching with an RK4 ODE (vllm-omni uses 10 steps); each code maps to 2 mel frames, then BigVGAN turns them into a waveform

References: `omni(/) model_executor/models/qwen2_5_omni/qwen2_5_omni.py:241-410` (forward for the three stages), `omni(/) model_executor/models/qwen2_5_omni/qwen2_5_omni.py:760-847` (talker prefill and decode input construction), `omni(/) model_executor/models/qwen2_5_omni/qwen2_5_omni.py:895-982` (codec to audio), `omni(/) model_executor/models/qwen2_5_omni/qwen2_5_omni_token2wav.py:601` (RK4 solver). HF golden reference: `↗ modeling_qwen2_5_omni.py` (focus on how `Qwen2_5OmniForConditionalGeneration.generate` stitches the thinker to the talker).

1.3 RTX 5090 environment notes

- Blackwell architecture (sm_120), 32 GB of memory. Requires CUDA 12.8+ and a PyTorch build with sm_120 support (cu128 wheel).
- For third-party kernels such as FlashAttention and FlashInfer, **verify sm_120 support for your versions yourself**. Recommended main path: PyTorch SDPA or your own Triton varlen kernel for prefill, and **your own Triton paged attention** for decode (which is itself a core exercise).
- When multiple processes share one GPU, their kernels are **time-sliced** by default and do not truly run concurrently. With **CUDA MPS** enabled they can (bonus, see M10). vllm-omni also offers a `cuda_mps` switch: `omni(/) engine/cuda_mps.py`.

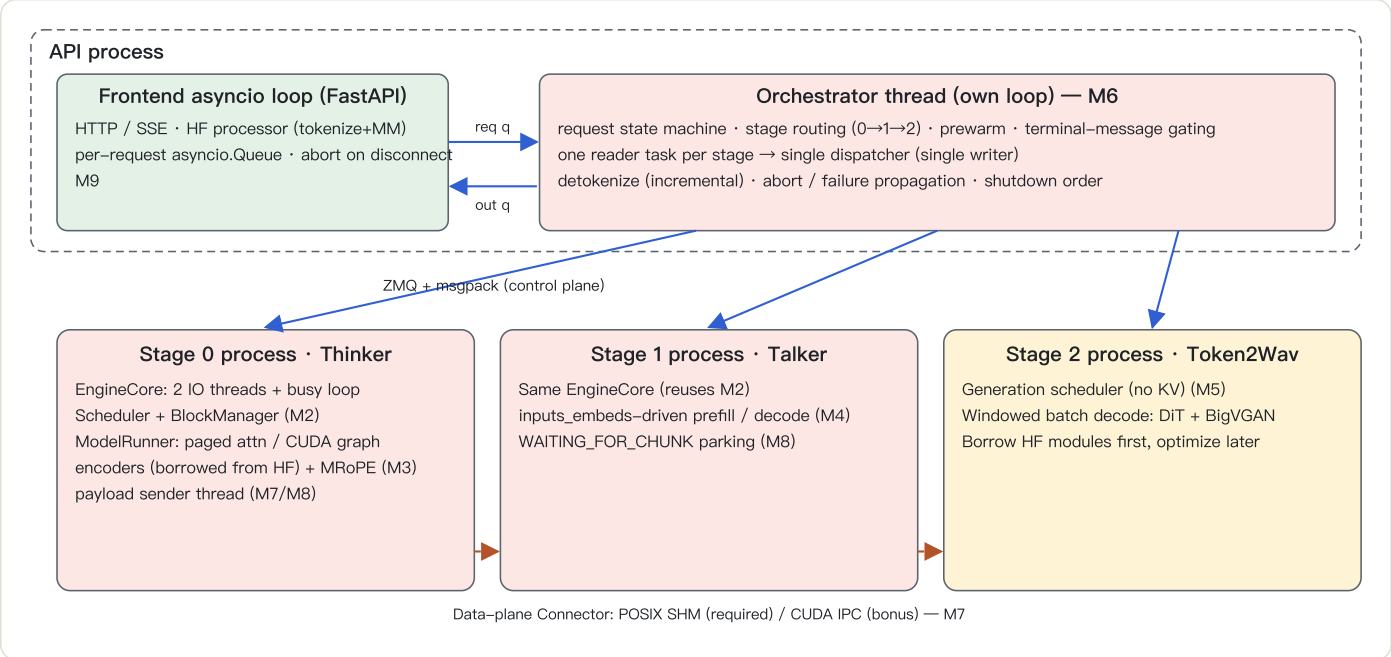
1.4 Memory budget (compute it yourself, put it in REPORT)

```
KV bytes / token = 2 (K,V) × num_layers × num_kv_heads × head_dim × dtype_bytes
e.g. if thinker text_config has 36 layers, 2 KV heads, head_dim=128, bf16
    → 2×36×2×128×2 = 36,864 B ≈ 36 KB/token → 8k tokens ≈ 0.29 GB
(use the actual values in config.json's thinker_config.text_config / talker_config)
```


© 2026 kzhou003

Item	Thinker	Talker	Token2Wav
Weights	LLM + audio encoder + ViT (the largest)	Small LLM + projection + needs the thinker's `embed_tokens` (see M4)	DiT + BigVGAN
KV cache	Main consumer	Compute from the talker config (codec sequences are long, several times the thinker's text length)	None (non-autoregressive)
Peak activations	Encoder activations during prefill of large images or long audio	Small	DiT attention over the whole mel, scales with audio length
CUDA graph	Every captured batch size takes its own graph pool and **must be counted in the budget** (vllm-omni's default path once missed the talker MTP graph, see `omni(/) worker/base.py:140-147`)		

2. Target architecture overview



Color legend: red = core, must write yourself green = may borrow yellow = borrow first, replace or optimize later . The overall structure matches vllm-omni: one API process and one EngineCore process per stage; the control plane goes through the orchestrator, while the data plane is transferred point-to-point between workers. The difference is that everything is on one card, so the three stage processes share `cuda:0`.

3. Write-it-yourself vs borrow list

Component	Ownership	Notes
Paged KV cache and BlockManager (prefix caching optional)	Write yourself	Understand vLLM's core abstraction

© 2026 kzhou003

Scheduler (continuous batching, chunked prefill, preemption)	Write yourself	All multi-stage waiting states hang off this
ModelRunner: input preparation (positions / slot_mapping / block_tables), sampling	Write yourself	per-request seed, top-k/p, repetition penalty
Paged decode attention kernel (Triton)	Write yourself	Prefill can use SDPA at first
Thinker LLM decoder and Talker LLM decoder (Qwen2 architecture, RMSNorm / RoPE / MRoPE / GQA / SwiGLU)	Write yourself	So they plug into your paged attention and CUDA graph
Per-stage weight splitting and loading	Write yourself	Streaming safetensors load by prefix
Thinker-to-Talker input construction (prefill concatenation, per-step reply vectors)	Write yourself	The most error-prone place for correctness
EngineCore process (IO threads, busy loop), ZMQ protocol	Write yourself	Follow vLLM's design
Orchestrator, request state machine, abort, failures, shutdown	Write yourself	The concurrency core of this assignment
SHM Connector and chunk adapter (background threads, keys, cleanup)	Write yourself	The foundation of async_chunk
Decode CUDA graph (static buffers, batch buckets, padding)	Write yourself	Performance-critical
Tokenizer, chat template, `Qwen2_5OmniProcessor` (audio features, image patches)	Borrow from HF	Frontend only
Audio encoder and vision encoder (ViT) modules	Borrow HF's `nn.Module`	You handle weight loading, device placement, invocation, and embedding merge
Token2Wav (DiT + BigVGAN)	Borrow HF's or vllm-omni's implementation first	Bonus: implement batching and CUDA graph yourself
FastAPI / uvicorn, pyzmq, msgspec or msgpack	Borrow	—

4. Recommended code layout

```
nano_vllm_omni/
  config.py            # PipelineSpec / StageSpec / DeploySpec (dataclass, YAML optional)
  weights.py           # streaming safetensors load by prefix, name mapping
  models/
    qwen2_decoder.py   # self-written: RMSNorm, RoPE/MRoPE, GQA attention (calls paged attn), MLP
    thinker.py         # LLM + borrowed audio/vision encoder + embedding merge + hidden export
    talker.py          # thinker_to_talker_proj + LLM + codec head + input construction
    token2wav.py       # wraps HF/omni DiT + BigVGAN, exposes decode(codes, spk)
  engine/
    sequence.py        # Request / Sequence state (incl. WAITING_FOR_CHUNK)
    block_manager.py   # paged KV allocation, refcounts, prefix hash
    scheduler.py       # AR scheduler + Generation scheduler
    model_runner.py    # input prep, forward, sampling, CUDA graph
    sampler.py
    core_proc.py       # EngineCore process: IO threads + busy loop
    core_client.py     # async ZMQ client used in the orchestrator loop
  kernels/
    paged_attention.py # Triton decode kernel (+ optional prefill)
```


© 2026 kzhou003

```
  connector/
    shm.py             # POSIX SHM put/get + lock + reader unlink
    chunk_adapter.py   # recv/save threads, key scheme, park and restore
  orchestrator.py      # request state machine, routing, prewarm, terminal-message gating, abort
  frontend/
    engine.py          # frontend ↔ orchestrator thread bridge (janus or hand-written)
    server.py          # OpenAI-compatible API, SSE
tests/  bench/  scripts/golden_hf.py
```

---

5. Milestone roadmap

Suggested pace: about 3–7 days per milestone, roughly 8–10 weeks overall. **Proceed strictly in order, and pass acceptance before each next step**. Earlier parity tests stay in place as regression tests for all later optimizations.

M0 · Environment and golden reference 5 pts

Goal: run Qwen2.5-Omni-3B text plus speech output with HF transformers, and **dump each stage's intermediate tensors to disk** as the ground truth for all later parity tests.

Write yourself: `scripts/golden_hf.py`. Fix 3–5 prompts (text only, with image, with audio, optionally with video), use greedy decoding, a fixed seed or greedy for the talker, and save:

- thinker: `input_ids`, MRoPE `position_ids` (3×T), hidden of every layer (or at least layer 0 and the last layer), logits (first few steps), generated tokens;
- talker: prefill `inputs_embeds`, per-step input vectors, generated codec ids;
- token2wav: codec input, mel, wav waveform;
- peak memory (`torch.cuda.max_memory_allocated`) and end-to-end latency.

Acceptance: HF output is reproducible; golden files load in `pytest`.

Hints

- Registering forward hooks to capture intermediate tensors is cleaner than editing HF source.
- Talker sampling is random. For parity, either use greedy or a fixed seed and reproduce **the same sampling order** in your implementation. A simpler approach is "teacher forcing": feed the golden codecs directly and compare only logits.
- Don't expect bitwise equality in bf16. Compare logits with `atol≈1e-2` plus argmax agreement rate; compare waveforms with SNR or mel-L1.

M1 · Weight splitting and per-stage loading 8 pts

Goal: implement `load_stage(model_dir, stage) -> nn.Module` so that each stage builds only its own modules and reads only the tensors it needs.

Write yourself:

- Read `model.safetensors.index.json`, open only the relevant shards by prefix `thinker.`, `talker.`, `token2wav.`, load tensors one by one with `safe_open`'s `get_tensor` and **copy them straight to GPU**, so the whole checkpoint never lands in CPU memory first.

© 2026 kzhou003

- Name mapping: HF names to your module names (see how vllm-omni writes `WeightsMapper`).
- Special dependencies: **the talker stage also needs the thinker's** `embed_tokens` to embed thinker text tokens into vectors; token2wav needs `spk_dict.pt`.
- Load completeness check: report missing and unexpected parameter names and **raise an error**, not just a warning.

```
def load_stage(model_dir: str, stage: Literal["thinker","talker","token2wav"],
               device: torch.device, dtype=torch.bfloat16) -> nn.Module: ...
def iter_stage_tensors(model_dir: str, prefixes: tuple[str, ...]) -> Iterator[tuple[str, torch.Tensor]]: ...
```

Acceptance: (1) the union of the three stages' parameter sets equals the full checkpoint; (2) each stage's memory after loading matches the byte count computed from safetensors metadata; (3) a teacher-forcing forward of the thinker with **your own decoder** produces logits matching golden.

Hints

- vllm-omni has every stage iterate the full checkpoint and then routes by prefix in `load_weights` (`omni(/) model_executor/models/qwen2_5_omni/qwen2_5_omni.py:983-1044`). You can do better: open only the relevant shards based on the index.
- Where the talker borrows the thinker embedding: `omni(/) model_executor/models/qwen2_5_omni/qwen2_5_omni.py:1009-1022`.
- Qwen2's q/k/v have bias; o_proj does not. You can pack qkv into one Linear to cut kernel count.

M2 · Single-stage LLM engine core (nano-vllm level) 20 pts

Goal: start with **thinker text only** and implement, in a single process, an inference engine with continuous batching and paged KV cache. The output of `LLMEngine.generate(prompts, sampling_params)` matches golden.

Write yourself:

1. **Sequence / Request**: state (WAITING / RUNNING / FINISHED_*, with WAITING_FOR_CHUNK and WAITING_FOR_INPUT **reserved**), token list, `num_computed_tokens`, block table.
2. **BlockManager**: fixed block_size (e.g. 16); free list; on-demand allocation; release; (optional) prefix caching by block-content hash with refcounts.
3. **Scheduler**: each step, within `max_num_batched_tokens` and `max_num_seqs`, schedule running (decode) first, then waiting (prefill, optionally chunked prefill); preempt (recompute) when KV runs out.
4. **ModelRunner**: flatten one step's schedule into `input_ids`, `positions`, `slot_mapping`, `block_tables`, `context_lens`, `cu_seqlens`; run forward; compute logits only at each sequence's last position.
5. **Paged attention**: write KV (scatter by slot_mapping); varlen causal attention for prefill; your own Triton kernel for decode, reading KV via the block table.
6. **Sampler**: greedy, temperature, top-k, top-p, repetition penalty; **per-request seed** (one `torch.Generator` per request; explain how results stay independent of batch composition).

```
class Scheduler:
    def add(self, req: Request) -> None
    def schedule(self) -> SchedulerOutput          # which requests run this step, and how many tokens each
    def update(self, out: RunnerOutput) -> list[RequestOutput]   # append tokens, check stop, free blocks
class ModelRunner:
    def execute(self, so: SchedulerOutput) -> RunnerOutput
```


© 2026 kzhou003

Acceptance:

- □ Single-request greedy tokens match golden exactly (at least the first 64 tokens).
- □ With 8 requests of different lengths submitted together, each result matches its standalone run (batch invariance).
- □ With the KV budget artificially shrunk to trigger preemption, results are still correct.
- □ Unit test: Triton decode kernel vs naive attention within tolerance.

Hints

- nano-vllm is an excellent reference at about 1200 lines: `↗ nano-vllm`. Write it yourself first and look only when stuck.
- The corresponding vLLM modules: `↗ scheduler.py` `↗ kv_cache_manager.py` `↗ gpu_model_runner.py`.
- Start with the simplest "each step is either all prefill or all decode", then support mixed batches.
- KV budget: profile one max-batch forward to measure peak activations, then compute the block count from `budget − weights − peak activations − graph reserve`. With multiple stages on one card, do **not** use a "fraction of free memory"; give each stage an **explicit byte budget** (same idea as vLLM's `kv_cache_memory_bytes`).

M3 · Multimodal input and MRoPE 8 pts

Goal: the thinker accepts image and audio (optionally video) input, with results matching golden.

Write yourself:

- The frontend calls the HF processor to get `input_ids` (with placeholder tokens), `pixel_values`, `image_grid_thw`, `input_features`, etc., and **sends them to stage 0 over IPC**. Design your own tensor serialization format (msgpack + raw bytes, or SHM).
- In stage 0, run the borrowed encoders and scatter their outputs into the placeholder positions to get `inputs_embeds`.
- Implement MRoPE's 3×T position computation: identical across the three dims for text, a (t, h, w) grid for images, linear for audio; interleaved video/audio (TMRoPE) is a bonus. Decode position = previous position + 1 + offset (`mrope_position_delta`).
- The scheduler counts encoder compute against the token budget, or uses a separate budget, so one large image can't blow up a step.

Acceptance: for prompts with images and with audio, `position_ids` match golden exactly; the first 32 generated tokens match.

Hints

- Reference: `omni(/) model_executor/models/qwen2_5_omni/qwen2_5_omni.py:412-576` (`get_mrope_input_positions`, including the `seconds_per_chunk` audio/video interleaving).
- With prefix caching and multimodal input, the block hash must include a hash of the multimodal content; otherwise different images hit the same KV.
- Under chunked prefill, run the encoder only once: use an encoder cache (keyed by request and input) and consume its results across multiple steps.

© 2026 kzhou003

M4 · Talker stage (single-process chained version) 10 pts

Goal: first chain it together **in one process**: run the thinker to completion, take the embeddings and hiddens, build the talker input, and have the talker generate codecs with your engine (reusing M2). Ignore multi-process for now.

Write yourself:

- At each step the thinker exports the vectors the talker needs (as defined by HF generate, usually some combination of the layer-0 embedding and the last-layer hidden), accumulated per request.
- Talker prefill `inputs_embeds`: the thinker's prompt part, plus `[spk_token_embed + codec_pad_embed]`, plus `[first reply vector + codec_bos_embed]`. Prompt token ids are filled with mask, pad, and start placeholders.
- Talker per-step decode input = embedding of the codec sampled in the previous step + the k-th thinker reply vector; once reply vectors run out, switch to the pad or eos text vector (follow HF).
- The engine must support requests "**driven by inputs_embeds**", so talker prefill and decode can both take external vectors (extend the runner you wrote in M2).
- Sampling constraints: stop token 8294; suppress tokens with id > token2wav `num_embeds`; strip START, END, PAD from the output.

Acceptance: under teacher forcing, talker logits match golden at every step; with greedy or a fixed seed, the codec sequence matches golden (or matches on >95% of steps, with the reason explained in the report).

Hints

- References: `omni(/) model_executor/models/qwen2_5_omni/qwen2_5_omni.py:686-847` (`talker_preprocess`, prefill concatenation, per-step decode), `omni(/) model_executor/models/qwen2_5_omni/qwen2_5_omni_talker.py:56-106` (`thinker_to_talker_proj`), `omni(/) model_executor/stage_input_processors/qwen2_5_omni.py:42-63` (stripping codec boundary tokens).
- When the thinker finishes, vLLM V1 appends the stop token to the output before checking stop, so the hidden has **one extra row** that must be trimmed: `omni(/) model_executor/stage_input_processors/qwen2_5_omni.py:261-317`. A classic off-by-one bug.
- Write "cross-stage input construction" as a **pure function**: `build_talker_prefill(payload) -> (ids, embeds)`. When M7 and M8 switch to cross-process transfer, this function needs no changes at all.

M5 · Token2Wav stage 7 pts

Goal: codec to waveform, first decoding the whole sequence at once, then **chunked streaming** decode.

Write yourself:

- `Token2WavRunner.decode(codes, speaker) -> wav`: load cond and ref_mel, initialize noise, integrate the DiT with RK4 for N steps to get mel, then BigVGAN to get the waveform.
- Chunked streaming: decode every W codes with left context L, emit only the waveform for the new part, and overlap or crossfade at the seams. **Determinism**: use a per-request seed for each request's noise so chunked and whole-sequence results stay as close as possible.
- A Generation scheduler: no KV, no sampling; pad the windows of requests that are ready at the same time and process them as a batch.

© 2026 kzhou003

Acceptance: whole-sequence decode reaches a mel-L1 or SNR threshold against the golden waveform (your choice, stated in the report); the chunked-vs-whole difference is quantified, and there are no audible seams.

Hints

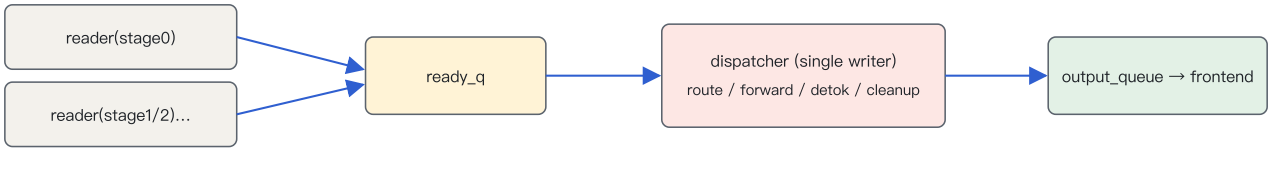
- References: `omni(/) model_executor/models/qwen2_5_omni/qwen2_5_omni.py:895-982` (cond and ref_mel fallback logic, `factor`, `steps=10`), `omni(/) model_executor/models/qwen2_5_omni/qwen2_5_omni_token2wav.py:1038-1180` (`process_chunk*`, including batched DiT and BigVGAN).
- Qwen3-Omni's Code2Wav uses a stateless "recompute left context, then slice" scheme (`omni(/) model_executor/models/qwen3_omni/qwen3_omni_code2wav.py:447-508`), a useful model for designing your window strategy.

M6 · Multi-process topology and Orchestrator 14 pts

Goal: split the three stages into three EngineCore child processes, chained by an orchestrator thread in the API process. Data first goes as "whole payload forwarded through the orchestrator" (slowest but simplest); M7 optimizes it.

Write yourself:

1. **EngineCore process** (modeled on vLLM's `EngineCoreProc`):
   - Input IO thread: ZMQ PULL or ROUTER, decode msgpack, put into a `queue.Queue`;
   - busy loop thread: take messages (ADD / ABORT / UTILITY) → `schedule()` → `execute()` → `update()` → put into the output queue; when there is no work, block on the input queue instead of spinning;
   - Output IO thread: serialize, then ZMQ PUSH;
   - On SIGTERM, set a flag and push a WAKEUP into the input queue so the busy loop never blocks forever.
2. **Startup and memory**: the parent spawns the three stages **sequentially** (same card), each with an explicit `kv_cache_bytes`; a child sends a READY handshake after "load → profile → allocate KV → capture graph"; if any stage fails, the others must be reaped. Optional: serialize the "measure free memory" step with a cross-process `flock`.
3. **Orchestrator thread** (its own asyncio loop):
   - One reader task per stage client waits on ZMQ output and puts it into a `ready_q`; a **single** dispatcher processes serially, guaranteeing request state has only one writer;
   - Request state: `stage_submit_ts`, `finished_stages`, `final_stage_ids={0(text),2(audio)}`;
   - When stage 0 finishes, build the stage 1 request (calling the M4 pure function); when stage 1 finishes, build the stage 2 request;
   - Incremental detokenization lives in the orchestrator (one offset per request).
4. **Frontend bridge**: bounded request queue (provides backpressure), unbounded output queue; cross-thread wakeups are **batched** (one `call_soon_threadsafe` per batch), and lost wakeups are handled.




```
@dataclass
class ReqState:
```


© 2026 kzhou003

```
    request_id: str
    prompt: Any
    stage_submit_ts: dict[int, float]
    finished_stages: set[int]
    final_stage_ids: frozenset[int]
    pending_final: Optional[OutputMsg] = None   # for M8 terminal-message gating
class Orchestrator:
    async def run(self): ...                      # start readers + dispatcher + request handler
    async def _on_stage_output(self, stage_id: int, outs: list[EngineOutput]): ...
    async def _forward(self, req: ReqState, src: int): ...
    async def _cleanup(self, req_ids: list[str], abort: bool): ...   # the only exit path
```

Acceptance:

- ☐ End to end: text plus audio output matches the M4/M5 single-process version.
- ☐ With 8 concurrent requests, every request's text tokens arrive in the right order, and `finished` appears exactly once, as the last message.
- ☐ When idle, CPU usage of the three child processes and the orchestrator is near 0 (no busy-polling).
- ☐ After Ctrl-C all processes exit, no zombies remain, and `nvidia-smi` shows all memory released.

Hints

- Reading order: `omni(/) engine/omni_engine_base.py:300-340` (threads and queues) → `omni(/) engine/async_engine_utils.py:40-97` (batched wakeups) → `omni(/) engine/omni_engine_base.py:973-991` (drain → arm → drain → wait) → `omni(/) engine/orchestrator.py:1099-1320` (readers plus dispatcher) → `omni(/) engine/orchestrator.py:3062-3162` (register state first, then submit).
- **Register request state before sending**. Otherwise output may arrive before the state exists.
- Don't `await` inside the dispatcher's critical section, or the single-writer guarantee breaks. If you truly need to await, finish mutating state first, then await.
- Use `spawn`, not `fork`, for child processes (CUDA does not support fork).

M7 · Data-plane Connector (bypassing the orchestrator) 8 pts

Goal: thinker-to-talker hiddens and embeddings (large) and talker-to-token2wav codecs are now transferred **point-to-point** directly between workers. The orchestrator only passes a "placeholder request" carrying length and sender info.

Write yourself: `SHMConnector.put(key, tensors)` / `get(key) -> tensors | None`:

- Keys are deterministic: `{req_id}_{src_stage}_{chunk_id}`, so both sides match up without an extra metadata channel;
- The writer first holds an exclusive `flock` on a same-named lockfile, then creates and writes the POSIX shm; the reader probes with `LOCK_EX|LOCK_NB` to tell whether writing is done;
- **The reader unlinks** (single consumer); the writer periodically reclaims consumed keys; on abort or shutdown, clean up in bulk by prefix;
- The receiving stage gets a new `WAITING_FOR_INPUT` state: a request that has arrived but whose payload hasn't **does not enter the scheduling queues**; once the background recv thread has the data, it hands it back to the scheduler thread, where it is applied **on the scheduler thread**.

Acceptance: results match M6; IPC bytes through the orchestrator process drop by an order of magnitude (instrumented); after 100 requests (including aborts) nothing is left in `/dev/shm`.

© 2026 kzhou003

Hints

- References: `omni(/) distributed/omni_connectors/connectors/shm_connector.py:195-364`, `omni(/) entrypoints/stage_utils.py:193-237`.
- Bonus: on a single card you can use CUDA IPC (`torch.multiprocessing` sharing CUDA tensors, or `cudaIpcGetMemHandle`) for zero D2H. The hard part is **lifetime**: the producer must not free the memory before the consumer is done with it, and cross-process streams must be synchronized with events.

M8 · async_chunk streaming pipeline 12 pts

Goal: each time the thinker decodes a token, it sends the corresponding reply vector to the talker; each time the talker accumulates W codecs, it sends them to token2wav. All three stages **work simultaneously**, and time to first audio packet (TTFP) drops from "full duration" to sub-second.

Write yourself:

1. **ChunkAdapter** (one per stage process):
   - `save_async(req, payload)`: on the scheduler thread, only **snapshot and enqueue**; the save thread builds the payload and does the `put`;
   - `load_async(req)`: the recv thread polls `get`, backing off on failure; the condition variable's timeout is the lost-wakeup fallback;
   - chunk_id is incremented under the lock only after a successful put; the terminal chunk (finished flag) **must not be swallowed by a cancel**;
   - Fence cleanup against "committing a received chunk" with a lock plus a generation check, so a late chunk can't be written into an already-cleaned-up request.
2. **Scheduler hooks**: before `schedule()`, **move** requests waiting for a chunk out of the waiting/running queues (set to WAITING_FOR_CHUNK); in `finally`, put requests whose chunk is ready back.
3. **Prewarm**: as soon as the orchestrator receives a request, it submits placeholder requests to stage 1 and stage 2. These park in WAITING_FOR_CHUNK and can start the moment the first chunk arrives.
4. **Terminal-message gating**: send `finished=True` to the frontend only when every submitted stage has finished. Until then, the terminal message is held in `pending_final`.
5. **Small first chunk, larger later chunks**: e.g. 4 codes in the first chunk, 25 per chunk after (or a configurable ramp). Give the TTFP vs RTF trade-off curve in the report.

Acceptance:

□ With async_chunk on, audio "sounds the same" as the non-streaming version, with the difference quantified (with identical talker sampling, codec sequences should match exactly).
□ At concurrency 1, TTFP drops by more than an order of magnitude vs M7 (give numbers).
□ Stress test: 1000 requests, a random 10% aborted midway; no request hangs forever, no SHM leaks, no process crashes.

Hints

- References: `omni(/) distributed/omni_connectors/transfer_adapter/base.py:20-140`, `omni(/) distributed/omni_connectors/transfer_adapter/chunk_transfer_adapter.py:381-503` (save_async snapshot), `omni(/) distributed/omni_connectors/transfer_adapter/chunk_transfer_adapter.py:1120-1186` (process_pending_chunks), `omni(/)`

© 2026 kzhou003

`core/sched/omni_ar_scheduler.py:408-451` (pre/post schedule hooks), `omni(/) engine/orchestrator.py:1954-1967` (terminal-message gating), `omni(/) engine/orchestrator.py:2711-2870` (prewarm).

- Qwen3-Omni's chunk processing functions are a good template: `omni(/) model_executor/stage_input_processors/qwen3_omni.py:409-542` (thinker to talker), `omni(/) model_executor/stage_input_processors/qwen3_omni.py:765-842` (talker-to-code2wav windows).
- The talker's prompt length depends on the thinker's prompt length, but at prewarm time the thinker hasn't started. Fix: the frontend has already tokenized, so the orchestrator knows the prompt length and the prewarm placeholder prompt can be `[0]*expected_len`.
- If you implement async scheduling (bonus), the sender may only send "confirmed" tokens: `num_computed - num_placeholders`.

M9 · OpenAI-compatible server 5 pts

Goal: `POST /v1/chat/completions`, supporting `image_url` and `input_audio` in `messages`, plus `modalities`, `audio: {voice, format}`, and `stream`.

Write yourself: **interleave** text deltas and audio deltas (base64 PCM or wav fragments) in SSE; on client disconnect (`CancelledError`) run abort under `asyncio.shield` with a timeout; `/health` returns 503 if any stage process has died.

Acceptance: both the `openai` Python SDK and curl stream text and audio; killing the client mid-stream makes all three stages receive the abort, and KV blocks and SHM are reclaimed (print the counts).

Hints

- References: `omni(/) entrypoints/async_omni.py:325-332` (shield abort), `omni(/) entrypoints/async_omni_base.py:460-493` (synthesize a terminal message after abort to prevent hangs).

M10 · Performance: CUDA graph, IO/compute overlap 10 pts

Goal: thinker and talker decode are no longer bound by kernel launch overhead, and cross-stage D2H copies don't block compute.

Write yourself:

1. **Decode CUDA graph**: preallocate static buffers (`input_ids`, `positions`, `slot_mapping`, `block_tables`, `context_lens`, output hidden), capture once per batch bucket (1, 2, 4, 8, 16, ...); at runtime pad to the nearest bucket and replay. Note: no Python branching, CPU syncs, or dynamic shapes inside the graph.
2. **Count graph memory in the budget**: measure memory before and after capture, and include it in the KV budget.
3. **Async payload D2H**: use a separate copy stream, `wait_stream(compute)` first, **clone before copying** (the graph buffer is overwritten by the next replay), copy into pinned host memory, call `record_stream`, then record an event; a background thread waits for the event before doing the put.
4. **(Optional) CUDA MPS**: when enabled, kernels from the three processes truly run concurrently. Compare RTF at concurrency 4 with it on and off.

© 2026 kzhou003

Acceptance: with CUDA graph on, results match eager exactly; thinker TPOT at batch=1 goes down (give numbers); a Nsight Systems screenshot shows D2H overlapping compute.

Hints

- References: `omni(/) worker/gpu_ar_model_runner.py:160-297` (async output snapshot: clone, pinned, record_stream, event, builder thread), `omni(/) worker/gpu_ar_model_runner.py:1730-1739` (lesson: pageable memory makes D2H degrade to synchronous), `omni(/) worker/gpu_ar_model_runner.py:634-690` (how to capture per bucket), `omni(/) worker/gpu_model_runner.py:243-272` (choosing static buffer sizes).
- A batch with per-request seeds can't simply replay a graph (the graph doesn't re-run Python generator logic). Either make the RNG state a tensor inside the graph, or run such batches eagerly: `omni(/) worker/gpu_model_runner.py:1998-2010`.

M11 · Correctness and robustness 8 pts

Goal: under an abort at any moment, a crash of any stage process, or a normal shutdown, the system converges to a consistent state.

Write and test yourself:

- **Two-phase abort**: first collect partial output from the detokenizer (without committing), then have the EngineCore abort, and commit the abort state only after it succeeds; with multiple final stages, only the last message carries finished.
- **Abort vs ADD ordering**: in the EngineCore, an abort may take effect only after the ADDs preceding it have been processed, so "abort then add" can't resurrect a request.
- **Fault isolation**: after `kill -9` on the talker process, requests using the talker get an error, text-only requests are unaffected, and `/health` reports unhealthy.
- **Shutdown order**: stop the orchestrator first, then the stage processes; every operation is idempotent.
- **Monotonic terminal state**: output arriving after cleanup is dropped; an update for a request that no longer exists does not resurrect it.

Acceptance: write a randomized fault-injection test (random aborts, random SHM delays, random stage kills) and run it for 30 minutes, such that every request gets **exactly one** terminal message and nothing leaks (KV block count returns to its initial value, `/dev/shm` is empty, no zombie processes).

Hints

- References: `omni(/) engine/stage_pool.py:1279-1339` (two-phase abort), `omni(/) engine/stage_engine_core_proc.py:147-154` (FIFO fence between abort and ADD), `omni(/) engine/orchestrator.py:1464-1515` (a dead process affects only requests bound to it), `omni(/) engine/omni_engine_base.py:1132-1220` (shutdown order).

© 2026 kzhou003

6. Benchmarks and acceptance

Metric	Definition	Suggested target (calibrate against your own HF baseline)
TTFT	Request arrival → first text token	Hundreds of milliseconds for short prompts at concurrency 1
TPOT	Interval between text tokens	Clearly lower than eager with CUDA graph on
TTFP	Request arrival → first audio chunk	Sub-second with async_chunk on (concurrency 1)
RTF	Generation time ÷ audio duration	< 1 (real time) at concurrency 1, ideally < 1 at concurrency 4
Throughput	req/s and audio seconds/s at concurrency 1, 4, 8	Provide curves
Memory	Per-stage weights, KV, graph, activations	Put in REPORT and explain the headroom

For reference: with async_chunk on, vllm-omni on Qwen3-Omni cuts TTFP at concurrency 1 from about 6.5 s to about 0.52 s (H800, `omni(/) docs/design/feature/async_chunk.md`). Your model is smaller; treat this as an order-of-magnitude comparison.

7. Grading rubric (100 points + bonus)

Milestone	Points	Key criteria
M0 Golden reference	5	golden is reproducible and loadable
M1 Loading	8	Load by prefix, completeness check, thinker logits parity
M2 Engine core	20	paged KV, continuous batching, preemption, batch invariance, Triton kernel
M3 Multimodal	8	Exact MRoPE positions, encoder cache
M4 Talker	10	Correct input construction, stop row trimmed, codec parity
M5 Token2Wav	7	Whole-sequence parity, chunked streaming without audible seams
M6 Multi-process	14	IO threads, single-writer dispatcher, backpressure, no busy-poll, clean exit
M7 Connector	8	Point-to-point transfer, reader unlink, no leaks
M8 async_chunk	12	Park and restore, prewarm, terminal-message gating, TTFP improvement
M9 API	5	SSE text plus audio, abort on disconnect
M10 Performance	10	CUDA graph parity, memory accounted for, D2H overlap
M11 Robustness	8	30-minute fault injection: exactly one terminal message per request, zero leaks
REPORT	—	Not scored separately, but if missing each milestone is capped at 70%

© 2026 kzhou003

8. Bonus items (+3 to +8 each)

- **Async scheduling**: overlap scheduling of step N+1 with GPU compute of step N, using placeholder tokens that the GPU side fills in with real ids (see `↗ async_scheduler.py`).
- **CUDA IPC data plane**: zero-copy on the same card, with lifetimes and cross-process events managed correctly.
- **Token2Wav batching and CUDA graph**: group windows from multiple requests by length, then replay (see `plan_decode_groups` in `omni(/) model_executor/models/qwen3_omni/qwen3_omni_code2wav.py:63-121`).
- **Prefix caching (multimodal-safe)**: reuse KV for system prompts and for multi-turn conversations about the same image.
- **FP8 thinker**: run Qwen2.5-Omni-7B on the 5090 and give a quality comparison.
- **Single-process multi-stream version**: run the three stages in one process on three CUDA streams, compare latency and complexity against the multi-process design, and write it up in the report.
- **Realtime / WebSocket duplex**: streaming audio input plus VAD.

9. Common pitfalls (look here when you hit a problem)

Correctness

- When the thinker stops there is one extra hidden row; if the talker consumes it, noise appears at the end of the audio. See the M4 hints.
- The talker's KV, head count, and max_len must come from **the talker's own config**; using the thinker's leads to wrong attention shapes (vllm-omni has the same comment: `omni(/) config/model.py:268-285`).
- MRoPE decode positions must add `mrope_position_delta`; you can't just use `len(tokens)`.
- Special ids emitted by the talker (START, END, PAD, MASK) reaching token2wav cause out-of-bounds indexing or abnormal output length: `omni(/) model_executor/stage_input_processors/qwen2_5_omni.py:42-63`.
- bf16 RMSNorm accumulates in fp32; some `torch.compile` fusions change numerics, so when parity fails, turn compile off first while debugging.

Concurrency

- Output arrives before request state: always register state before submitting.
- Lost wakeup: the standard pattern is "drain → arm → drain again → wait", or a condition variable with a timeout as fallback.
- The recv thread mutates request objects in the scheduler directly and races with the scheduler thread: hand the data to the scheduler thread and let it apply it.
- A cancel swallows the terminal chunk too, and the downstream request sits in WAITING_FOR_CHUNK forever.
- A blocking `queue.put` on the asyncio loop stalls the entire frontend (vllm-omni has this risk too: `omni(/) engine/async_omni_engine.py:584`).
- Child processes inherit `CUDA_VISIBLE_DEVICES` or other environment variables set by the parent: take a lock when setting env vars for spawn, and restore them right after spawning (`omni(/) engine/stage_engine_startup.py:911-934`).

Memory and GPU

- When the three stages initialize concurrently, their "measure free memory" steps interfere, overestimating KV and causing OOM: either initialize sequentially, use explicit byte budgets, or use a cross-process read-write lock (`omni(/) engine/stage_phase_lock.py:106-238`).

© 2026 kzhou003

- The CUDA graph pool is not counted in the budget.
- D2H from pageable memory degrades to a synchronous copy; async-copying a graph output buffer without cloning it first lets the next replay overwrite the data.
- Forgetting `record_stream` when using a tensor across streams lets the caching allocator reuse that memory early, causing random data corruption.

---

10. References

- Concepts and concurrency design primer (the previous document): `Part III of this book`
- nano-vllm (minimal reference for a single-stage engine): `↗ GeeeekExplorer/nano-vllm`
- vLLM V1: `↗ engine/core.py` `↗ engine/core_client.py` `↗ core/sched/scheduler.py` `↗ worker/gpu_model_runner.py`
- HF golden reference: `↗ modeling_qwen2_5_omni.py`, model card `↗ Qwen/Qwen2.5-Omni-3B`
- vllm-omni Qwen2.5-Omni implementation: `omni(/) model_executor/models/qwen2_5_omni/qwen2_5_omni.py` `omni(/) model_executor/models/qwen2_5_omni/qwen2_5_omni_token2wav.py` `omni(/) model_executor/stage_input_processors/qwen2_5_omni.py` `omni(/) deploy/qwen2_5_omni.yaml`
- vllm-omni design docs: `omni(/) docs/design/feature/async_chunk.md` `omni(/) docs/design/module/engine_orchestration.md` `omni(/) docs/design/module/stage_runtime.md`
- Triton tutorials (required reading before writing paged attention): `↗ triton-lang.org tutorials`
- ZeroMQ guide: `↗ zguide`

Note: model sizes and memory figures here are estimates only; the checkpoint's config.json and safetensors metadata are authoritative. vllm-omni's default Qwen2.5-Omni deployment is `max_num_seqs: 1`, `enforce_eager`, without async_chunk (`omni(/) deploy/qwen2_5_omni.yaml`), so M8 and M10 in this assignment actually go beyond what the reference implementation achieves for this model: they are the genuinely challenging part.

© 2026 kzhou003

PART III

Reference: a deep dive into vLLM-Omni's concurrency design

Using Qwen3-Omni as the example, how the real system runs fast and correctly: process/thread topology, the orchestrator, stage processes, the async_chunk data plane, GPU overlap, abort, failures and shutdown. Come back to it while doing the exercises and the project.

© 2026 kzhou003

A Deep Dive into vLLM-Omni Concurrency Design

Following Qwen3-Omni-MoE (thinker → talker → code2wav) as the main thread · source based on vllm-omni `fa8b4199c` (main) · upstream vLLM v0.31.0

---

0. How to read this

This document answers one question: **how does a multi-stage omni model service manage to be both "efficient" (stages overlap as a pipeline, IO overlaps with compute, the GPU never idles) and "correct" (no lost messages, no reordering, state converges after abort/failure)?**

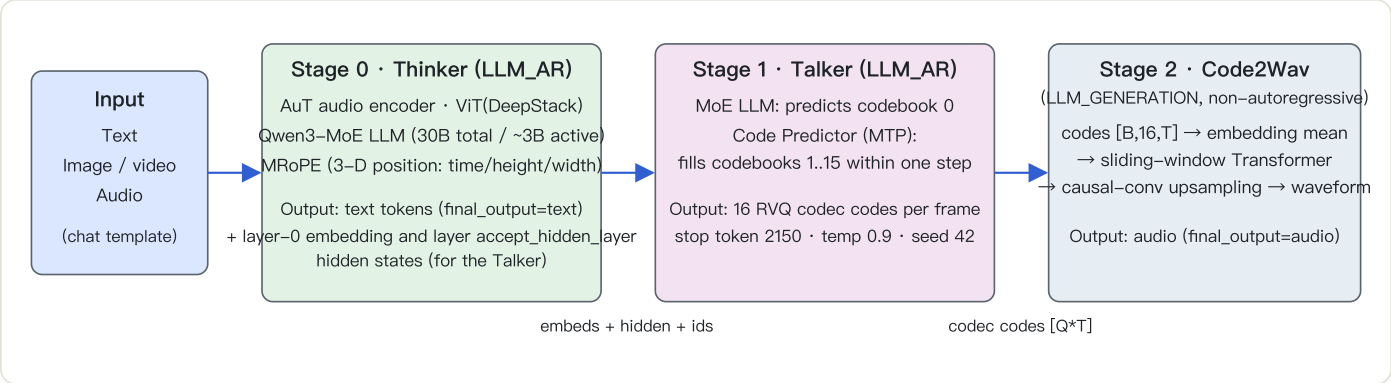
- Links styled like `⟨/⟩ path:line` point to specific lines of vllm-omni at a pinned commit on GitHub; click to read the source.
- Links styled like `vLLM ↗ file` point to files in upstream vLLM v0.31.0 (file-level only, since there is no local checkout of that version to verify line numbers against).
- Colored boxes: vLLM concept explains upstream fundamentals; Why explains design motivation; Note flags pitfalls or known risks.

The whole design in one sentence: **"one API process + one vLLM EngineCore subprocess per stage"**. Inside the API process, the frontend asyncio loop and a separate orchestrator thread (with its own event loop) communicate through janus queues. The orchestrator uses single-threaded cooperative scheduling to keep request state consistent. Control messages between stages go through the orchestrator (ZMQ), while bulk data and streaming chunks bypass the orchestrator: workers transfer them point-to-point via OmniConnector (shared memory, etc.), overlapped with compute by background threads and dedicated CUDA streams.

---

1. Background: the Qwen3-Omni model

Qwen3-Omni-30B-A3B is an "any modality in, text + speech out" model. It uses a **Thinker–Talker** architecture: the Thinker "thinks" (understands + generates text), the Talker "speaks" (turns semantics into speech codec codes), and finally a vocoder decodes the codec codes into a waveform. vllm-omni splits it into 3 stages; see `⟨/⟩ model_executor/models/qwen3_omni/pipeline.py:22-84`.



© 2026 kzhou003

1.1 Components

Component	What it does	Source
Unified wrapper	`Qwen3OmniMoeForConditionalGeneration`. The same class builds only the submodules its stage needs (thinker / talker / code2wav) based on `model_stage`, so the three processes load different "slices" of the same class.	⟨/⟩ model_executor/models/qwen3_omni/qwen3_omni.py:139 ⟨/⟩ model_executor/models/qwen3_omni/qwen3_omni.py:200–352
Thinker audio encoder (AuT)	`Qwen3OmniMoeAudioEncoder`, encodes mel spectrograms into LLM token embeddings.	⟨/⟩ model_executor/models/qwen3_omni/qwen3_omni_moe_thinker.py:1131
Thinker vision encoder	ViT + DeepStack: visual features from several intermediate layers are added into the first few LLM layers, not just the input layer. Optional encoder CUDA graph.	⟨/⟩ model_executor/models/qwen3_omni/qwen3_omni_moe_thinker.py:1137–1168 ⟨/⟩ model_executor/models/qwen3_omni/vision_encoder_cudagraph.py:26–58
Thinker LLM	Qwen3-MoE. Captures the layer-0 input (i.e. the word embedding) and the hidden states of layer `accept_hidden_layer` for the Talker.	⟨/⟩ model_executor/models/qwen3_omni/qwen3_omni.py:435–437 ⟨/⟩ model_executor/models/qwen3_omni/qwen3_omni_moe_thinker.py:588–596
Talker	MoE LLM that uses `codec_embedding` in place of the vocabulary embedding and `codec_head` to output codebook-0 logits. `text_projection` / `hidden_projection` project thinker dimensions to talker dimensions.	⟨/⟩ model_executor/models/qwen3_omni/qwen3_omni_moe_talker.py:356–393
Code Predictor (MTP)	Takes the sampled CB0 and the talker's last hidden state, and loops inside one talker step to predict CB1..CB15 (16 codebooks total, each of size 2048). The next step's input embedding is the sum of the 16 codec embeddings plus the text step.	⟨/⟩ model_executor/models/common/qwen3_code_predictor.py:1428 ⟨/⟩ model_executor/models/qwen3_omni/qwen3_omni.py:1062–1069
Code2Wav	Causal-conv vocoder: codes are first embedded and averaged, then go through a sliding-window Transformer, TransConv/ConvNeXt upsampling and SnakeBeta decoding; output is clamped to [-1,1].	⟨/⟩ model_executor/models/qwen3_omni/qwen3_omni_code2wav.py:124–385

1.2 What happens in one Talker step (key to understanding GPU concurrency)

```
# qwen3_omni.py:1062–1069  —— before each talker decode step: MTP first completes this frame
codes, summed = self.talker.code_predictor_forward(input_ids, inputs_embeds,
                                                    last_talker_hidden=last_talker_hidden)
inputs_embeds = summed.reshape(-1, H)                    # sum of the 16 codebook embeddings
inputs_embeds = (inputs_embeds + text_step).reshape(-1, H)  # plus the thinker text row for this step
return inputs_embeds, codes.squeeze(-1)
```


© 2026 kzhou003

`⟨/⟩ model_executor/models/qwen3_omni/qwen3_omni.py:1062–1069`. In other words, **each talker token is really "one large-model forward + 15 small-model forwards"**. To avoid being crushed by kernel-launch overhead, the whole `talker_mtp` call is wrapped in a FULL CUDA graph (when `talker_mtp_graph_safe` holds, `⟨/⟩ worker/gpu_model_runner.py:255–262`), and the small model itself uses `torch.compile(dynamic=False)` (`⟨/⟩ model_executor/models/common/qwen3_code_predictor.py:906–914`).

1.3 How the Talker prompt is assembled

The Talker does not read the user text directly. Its prompt is assembled from thinker outputs (`⟨/⟩ model_executor/models/qwen3_omni/qwen3_omni.py:1242–1328`):

- Drop the system turn; in the user turn, text positions use `text_projection(embed)` and audio/image/video positions use `hidden_projection(hidden)` (`⟨/⟩ model_executor/models/qwen3_omni/qwen3_omni.py:1448–1506`).
- The assistant turn is a 9-token bootstrap: 3 role tokens + 4× `tts_pad` + `tts_bos` + the first text token, added to the codec-side `[nothink, think_bos, think_eos, speaker, codec_pad, codec_bos]` (`⟨/⟩ model_executor/models/qwen3_omni/qwen3_omni.py:1515–1568`).
- The remaining thinker text is consumed one row per talker step; once the text runs out, one `tts_eos` is emitted, then `tts_pad` forever after (`⟨/⟩ model_executor/models/qwen3_omni/qwen3_omni.py:1364–1407`).

> **Why this matters for concurrency**
> Talker step k only needs the thinker's k-th text row. So **every time the thinker decodes a token, that row's embedding can be sent to the talker**; the talker does not have to wait for the thinker to finish. This is the model-level reason `async_chunk` can cut time-to-first-packet audio latency (TTFP) from seconds to sub-second.

1.4 What crosses stage boundaries

Edge	Non-streaming (full payload)	async_chunk (streaming)
Thinker → Talker	`embed.prefill` (layer 0), `hidden_states.output` (accept layer), `tts_bos/eos/pad` embeddings, `ids.all/prompt`, speaker, language. `⟨/⟩ model_executor/stage_input_processors/qwen3_omni.py:545–634`	chunk 0 carries all prefill data (buffered until its length exceeds the prompt before sending); each later chunk carries only `embed.decode` `[1, D]`. `⟨/⟩ model_executor/stage_input_processors/qwen3_omni.py:409–542`
Talker → Code2Wav	`codes.audio` filtered to valid codec ids [0,2048), flattened codebook-major into `[Q*T]`. `⟨/⟩ model_executor/stage_input_processors/qwen3_omni.py:843–915`	Sent in windows: chunk 0 = 4 frames (`initial_codec_chunk_frames`), then one send every 25 frames (`codec_chunk_frames`), with up to 25 frames of left context. `⟨/⟩ model_executor/stage_input_processors/qwen3_omni.py:807–827`

```
# stage_input_processors/qwen3_omni.py:807–815 —— return a payload only once a full window is buffered, else None (nothing sent
this step)
if configured_initial_chunk_size > 0:
    if chunk_id == 0:
        chunk_size_config = configured_initial_chunk_size
    else:
        length -= configured_initial_chunk_size
chunk_length = length % chunk_size_config
if chunk_length != 0 and not is_finished:
    return None
```


© 2026 kzhou003

Each Code2Wav chunk is decoded **statelessly**: it re-decodes the left context and then slices out only the waveform for the new frames (`⟨/⟩ model_executor/models/qwen3_omni/qwen3_omni_code2wav.py:447-508`). The upside is that code2wav requests need no cross-chunk state, so they can be batched freely (`max_num_seqs: 64`).

1.5 Deployment shape (default deploy)

`⟨/⟩ deploy/qwen3_omni_moe.yaml:15-77` : `async_chunk: true` , using SharedMemoryConnector; stage 0 on GPU 0 (memory utilization 0.9), stages 1 + 2 share GPU 1 (0.6 + 0.1). Thinker temperature 0; talker `seed: 42, temperature: 0.9, top_k: 50` (a comment explains why the seed is pinned: the talker is the only stochastic stage, and without a seed a few samples "fluently babble nonsense"; see `⟨/⟩ deploy/qwen3_omni_moe.yaml:49-57`).

Performance reference (A100, `⟨/⟩ docs/design/qwen3_omni_tts_performance_optimization.md:246-256`): at concurrency 1, stacking batching, CUDA graph, and async chunk + streaming brings TTFP from 426 s down to 1.16 s and RTF from 2.08 down to 0.21. Another data point (H800, `⟨/⟩ docs/design/feature/async_chunk.md`): at concurrency 1, enabling async_chunk cuts TTFP from 6.46 s to 0.52 s.

---

2. vLLM core concepts crash course

Each LLM stage in vllm-omni is essentially an upstream vLLM V1 engine. The concepts below are prerequisites for the rest of the document.

> **vLLM concept · EngineCore and process separation**
> vLLM V1 splits the engine in two. The **frontend** (`AsyncLLM` , in the API process) handles tokenize/detokenize and streaming output. **EngineCore** (a separate process, `EngineCoreProc`) handles scheduling and model execution. They talk over **ZMQ** sockets with **msgpack** serialization. EngineCoreProc has 3 threads inside: the **input IO thread** (ZMQ → `input_queue`), the **main busy loop thread** (fetch requests → `schedule()` → `execute_model()` → `update_from_output()`), and the **output IO thread** (`output_queue` → ZMQ). This way serialization and network IO never block GPU scheduling.
> `vLLM ↗ vllm/v1/engine/core.py` `vLLM ↗ vllm/v1/engine/core_client.py` `vLLM ↗ vllm/v1/engine/async_llm.py`

> **vLLM concept · AsyncMPClient**
> The frontend's client for EngineCore. It creates a background task on the current asyncio loop that keeps reading messages from the ZMQ output socket and puts them into an `asyncio.Queue` . `get_output_async()` simply awaits that queue. A separate monitor thread watches the child process's sentinel; when the process dies it marks the engine dead, and later calls raise `EngineDeadError` . With DP>1, `DPLBAsyncMPClient` does load balancing.
> `vLLM ↗ vllm/v1/engine/core_client.py`

> **vLLM concept · Continuous batching / Scheduler / KV cache**
> Each step, the Scheduler picks requests from the `waiting` and `running` queues and forms a batch within the token budget (`max_num_batched_tokens`) and the concurrency cap (`max_num_seqs`). New requests can join at any time and finished ones leave at any time; that is continuous batching. The KV cache is paged in blocks (PagedAttention). **Chunked prefill** splits long prompts across several steps so they can be mixed with decode. **Prefix caching** reuses KV blocks for identical prefixes.
> `vLLM ↗ vllm/v1/core/sched/scheduler.py` `vLLM ↗ vllm/v1/request.py`

© 2026 kzhou003

vLLM concept · Async scheduling and placeholder tokens

In synchronous mode, scheduling step N+1 must wait for step N's sampled results to be copied from GPU back to CPU, which leaves the GPU idle. With `async_scheduling` enabled, the scheduler assumes each running request will produce 1 token (recorded as an **output placeholder**) and schedules step N+1 ahead of time; the real token id is filled into the next step's input directly on the GPU (`-1` as placeholder). This overlaps CPU scheduling with GPU compute. The relevant pieces are `AsyncScheduler`, `num_output_placeholders`, and `AsyncGPUModelRunnerOutput` (which does the D2H copy on a dedicated `async_output_copy_stream`). When `max_concurrent_batches > 1` (e.g. PP), the busy loop uses `step_with_batch_queue`, allowing multiple batches in flight at once.

`vLLM ↗ vllm/v1/core/sched/async_scheduler.py` `vLLM ↗ vllm/v1/worker/gpu_model_runner.py`

vLLM concept · Executor / Worker / ModelRunner

EngineCore drives Workers through an Executor. With TP=1 it uses `UniProcExecutor` and the worker lives inside the EngineCore process; with TP>1 it uses a multiprocess executor with one worker process per rank. The ModelRunner prepares inputs, runs the forward pass (possibly a CUDA graph replay), and samples.

`vLLM ↗ vllm/v1/executor/abstract.py`

vLLM concept · CUDA graph

Records a sequence of GPU kernels into a graph, then replays the whole graph with a single launch, eliminating Python and kernel-launch overhead. The cost: input shapes must be fixed (captured per batch-size bucket), and the buffers the graph uses are reused over and over. So when "copying GPU outputs away asynchronously" you **must clone first**, or the next replay overwrites the data (see §9).

Python concurrency concepts · primitives used in this document

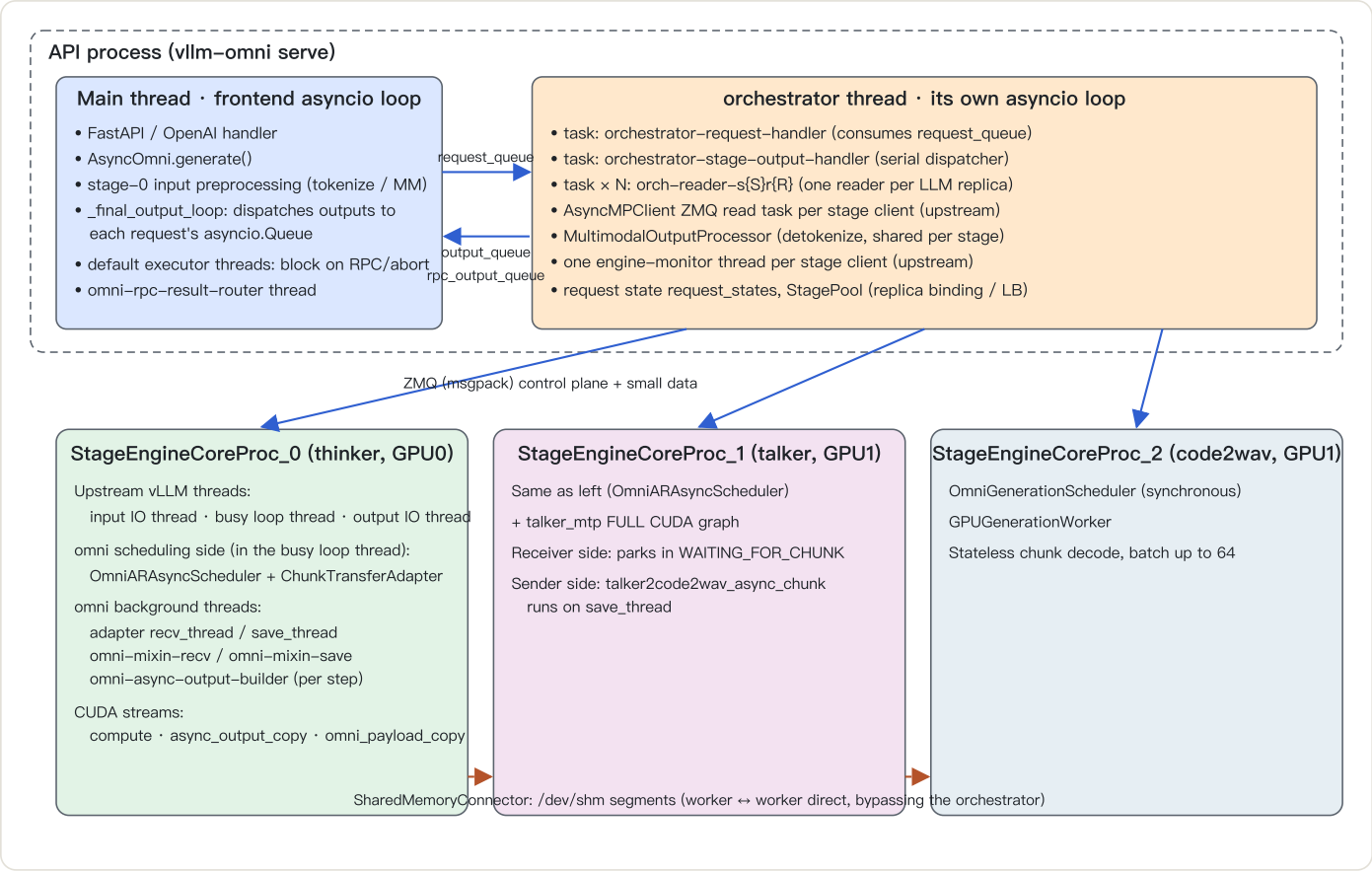
asyncio: single-threaded cooperative concurrency that switches only at `await`, so tasks on the same loop can access a shared dict without locks. **janus.Queue**: a queue with both a `sync_q` (thread side) and an `async_q` (coroutine side) interface, used to bridge "plain threads" and "an asyncio loop". **loop.call_soon_threadsafe**: safely wakes a loop from another thread. **threading.Condition**: a lock with wait and notify. **fcntl.flock**: a cross-process file lock. **POSIX shared memory**: named memory segments under `/dev/shm` that can be mapped across processes.

---

3. Big picture: processes / threads / event loops

Taking the default Qwen3-Omni deployment (2 GPUs, TP=1) as an example, a running service consists of **1 API process + 3 stage subprocesses**:

© 2026 kzhou003



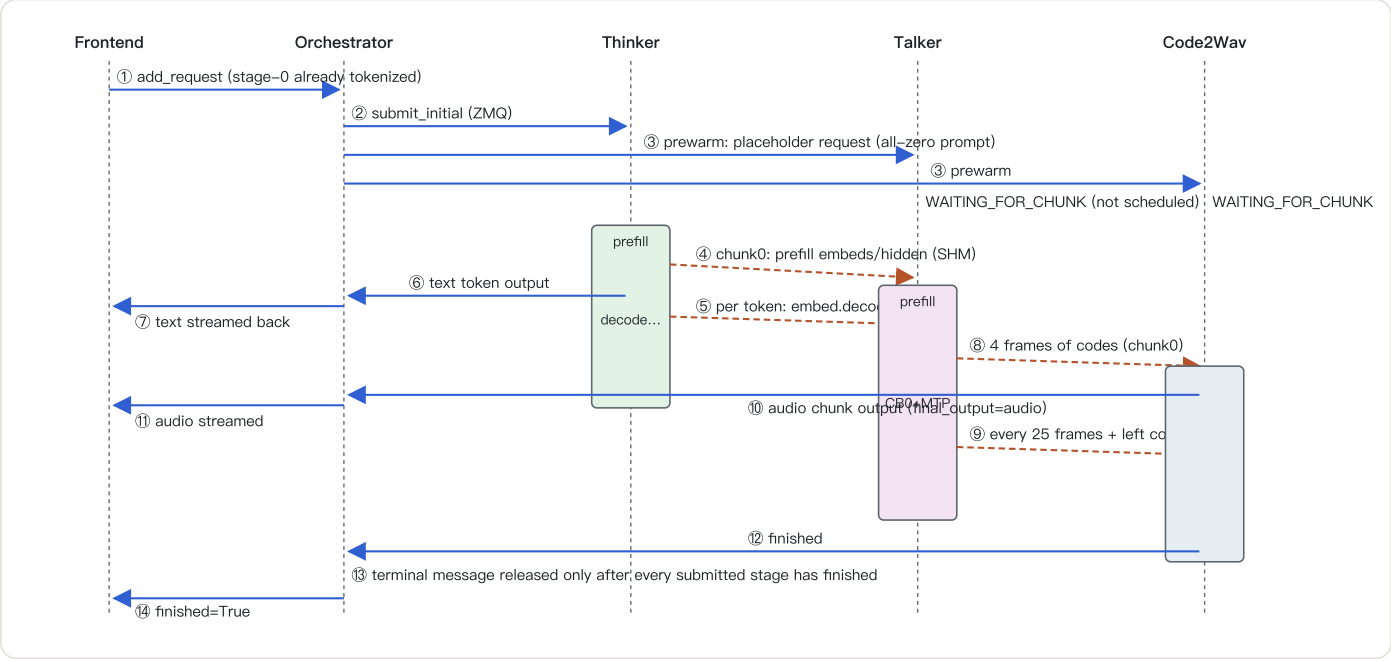
Executor	Responsibility	Source
Frontend loop (main thread)	HTTP; `AsyncOmni.generate`; stage-0 input processing done inline in `add_request`; `_final_output_loop`	⟨/⟩ engine/async_omni_engine.py:518-599 ⟨/⟩ entrypoints/async_omni_base.py:316-419
orchestrator thread	Daemon thread that internally runs its own loop with `run_until_complete`; handles stage initialization and `Orchestrator.run()`	⟨/⟩ engine/omni_engine_base.py:323-332 ⟨/⟩ engine/omni_engine_base.py:440-509 ⟨/⟩ engine/orchestrator.py:483-487
RPC router thread	Dispatches results from `rpc_output_queue` to waiters by key	⟨/⟩ engine/rpc_result_router.py:32-49
Stage process	`StageEngineCoreProc(EngineCoreProc)`, process name `StageEngineCoreProc_{stage}_replica{r}_DP{d}`	⟨/⟩ engine/stage_engine_core_proc.py:133 ⟨/⟩ engine/stage_engine_core_proc.py:242 ⟨/⟩ engine/stage_engine_core_proc.py:324

Why does the orchestrator get its own thread and loop?

1. **Isolation**: the frontend loop handles HTTP, SSE streams and tokenization (CPU-heavy), while the orchestrator must shuttle every step's output from every stage with low latency. With two loops, long synchronous work on one side does not directly stall the other side's await points.
2. **Single writer**: all cross-stage state (`request_states`, replica bindings, output processor state) is mutated only on the orchestrator loop. Because asyncio is cooperative, no locks are needed as long as nothing `await`s in the middle of a critical section.
3. **Swappable**: the same Orchestrator also serves the offline `Omni` (synchronous) entry point; the frontend just swaps in a synchronous queue interface.

© 2026 kzhou003

4. The life of a request (async_chunk mode)



1. **Frontend**: completes stage-0 input preprocessing on the frontend loop, then `request_queue.sync_q.put(msg)` (`⟨/⟩ engine/async_omni_engine.py:584`).
2. **Orchestrator**: `_handle_add_request` first registers an `OrchestratorRequestState` in `request_states`, then awaits `submit_initial`. This guarantees the request is already "known" by the time any output arrives (`⟨/⟩ engine/orchestrator.py:3107-3128`).
3. **Prewarm**: under async_chunk, placeholder requests are immediately submitted to every downstream stage that receives chunks (`⟨/⟩ engine/orchestrator.py:2711`, `⟨/⟩ engine/orchestrator.py:2762-2870`). So when the first chunk arrives, the downstream request is already waiting in the scheduler, saving one orchestrator round trip.
4. ④⑤⑧⑨ **Data plane**: chunks are written into shared memory by the worker and picked up by a background thread of the downstream scheduler, **without going through the orchestrator** (§8).
5. ⑥⑩ **Control plane**: each stage's outputs return to the orchestrator over ZMQ, are put into `ready_q` by a reader, and are then detokenized and routed by the serial dispatcher (§6).
6. ⑬ **Terminal-message gate**: even if the final stage reports finished first, as long as some submitted stage has not completed, the terminal message is held in `pending_final_output` (`⟨/⟩ engine/orchestrator.py:1954-1967`).

5. Frontend ↔ Orchestrator: cross-thread queues

All three queues are created in `OmniEngineBase.__init__` (`⟨/⟩ engine/omni_engine_base.py:308-311`):

Queue	Direction	Message types	Capacity
`request_queue` (janus)	frontend `sync_q` → orchestrator `async_q`	`StageSubmissionMessage`, `AbortRequestMessage`, `CollectiveRPCRequestMessage`, `ShutdownRequestMessage`, companion / interaction / remote replica registration, etc. See `⟨/⟩ engine/messages.py:29-91`	**256**, provides backpressure
`output_queue` (janus)	orchestrator → frontend	`OutputMessage`, `StageMetricsMessage`, `ErrorMessage`	Unbounded

© 2026 kzhou003

`rpc_output_queue` (janus)	orchestrator → router thread	`CollectiveRPCResultMessage`, `AbortResultMessage`, fatal `ErrorMessage`	Unbounded

5.1 Batched wakeups and lost-wakeup prevention on the output side

The orchestrator may produce thousands of outputs per second (every token, every audio chunk). If each one triggered `call_soon_threadsafe` to wake the frontend loop, cross-thread wakeup overhead would pile up. `OutputQueueSink` instead writes messages into `sync_q` but **schedules at most one wakeup per batch**.

```
# async_engine_utils.py:60-70
def put_nowait(self, item):
    self._sync_q.put_nowait(item)
    if self._wake_scheduled:          # a wakeup is already in flight; don't schedule another
        return
    self._wake_scheduled = True
    loop, ready = self._loop, self._ready
    if loop is not None and ready is not None:
        try:
            loop.call_soon_threadsafe(ready.set)
```

`⟨/⟩ engine/async_engine_utils.py:40-97`. The frontend's `get_outputs_async` uses the classic "**drain → arm → drain again → wait**" pattern: take up to 256 messages, re-arm the wakeup flag, drain once more (closing the lost-wakeup race where "a message arriving between drain and arm wakes nobody"), and finally wait on an `asyncio.Event` with a 1 s timeout as a liveness check (`⟨/⟩ engine/omni_engine_base.py:973-991`).

> **Why the output queue is unbounded**
> The orchestrator is the only mover on the hot path. If it blocked on put because the frontend was consuming slowly, every stage's output would stall and backpressure would propagate through ZMQ to the GPU. The design choice is that output never blocks the orchestrator; backpressure is applied only at the **entrance** (the 256 cap on request_queue).

5.2 Per-request output queues

The frontend has a single `_final_output_loop` task consuming `output_queue`, which puts messages into each request's own `asyncio.Queue` by request_id (`⟨/⟩ entrypoints/async_omni_base.py:380`, `⟨/⟩ entrypoints/client_request_state.py:23`); the async generator of `generate()` yields from there. Late messages for unknown requests are dropped (`⟨/⟩ entrypoints/omni_base.py:519-526`).

---

6. Orchestrator event-driven loop

Event-driven mode is on by default (True when `VLLM_OMNI_EVENT_DRIVEN_ORCH` is unset, `⟨/⟩ engine/orchestrator.py:99-103`). The structure is **"N readers + 1 serial dispatcher"**:

```
# orchestrator.py:1120-1136 — one reader per LLM replica
while not self._shutdown_event.is_set():
    raw_outputs = await pool.read_llm_raw_output(client)
    if raw_outputs is None:
        await asyncio.sleep(0.001); continue
    await ready_q.put(("llm", stage_id, replica_id,
raw_outputs))
...
except BaseException as e:
    await ready_q.put(("error", stage_id, replica_id, e))
```

```
# orchestrator.py:1264-1271 — the single dispatcher
while not self._shutdown_event.is_set():
    if pending_get is None:
        pending_get = asyncio.create_task(ready_q.get())
    done, _ = await asyncio.wait(
        {pending_get, shutdown_task, reconcile_task},
        timeout=max(0.0, next_reconcile - now),
        return_when=asyncio.FIRST_COMPLETED)
```


© 2026 kzhou003

- A reader awaits `client.get_output_async()` (`⟨/⟩ engine/stage_pool.py:1177`), i.e. the upstream AsyncMPClient's asyncio queue, rather than polling a socket. With no output, the reader sleeps entirely at zero CPU.
- All readers feed into **one** `ready_q` (`⟨/⟩ engine/orchestrator.py:1112`), which the dispatcher processes serially. This is "multiple producers / single consumer": reads can be concurrent, state mutation is serial.
- When replica membership changes (remote replicas come and go), `membership_listener` triggers `_wake_reconcile` via `call_soon_threadsafe` to add/remove readers; otherwise a fallback reconcile runs every 0.5 s (`⟨/⟩ engine/orchestrator.py:1243-1286`).
- Diffusion stages still use 0/1 ms polling (`⟨/⟩ engine/orchestrator.py:1138-1157`). The old polling mode (`⟨/⟩ engine/orchestrator.py:1040-1097`) shares `_process_llm_stage_outputs` and `_handle_processed_outputs` with event-driven mode, so the semantics are identical.

> **Why event-driven beats 1 ms polling**
> In polling mode, every 1 ms runs `asyncio.wait_for(read, 0.001)` on every replica (`⟨/⟩ engine/stage_pool.py:1252-1257`). That both wastes CPU (and competes with frontend tokenization for the GIL) and adds up to 1 ms latency per hop. The talker emits output every frame, and with 3 stages chained, this latency accumulates into TTFP and ITL.

6.1 Request state machine

Key fields of `OrchestratorRequestState` (`⟨/⟩ engine/orchestrator.py:232-271`):

- `final_stage_id` / `final_output_stage_ids` / `finished_final_output_stage_ids`: which stages' outputs are returned to the user (Qwen3-Omni has two: the thinker's text and code2wav's audio).
- `stage_submit_ts: dict[int, float]`: **the sole criterion for "has a given stage been submitted"** (`⟨/⟩ engine/orchestrator.py:2045-2046`), also used by the dead-replica scan.
- `finished_stage_ids`, `pending_final_output`: the terminal-message gate (see below).
- `streaming: StreamingInputState`: segment state for streaming input (e.g. realtime).

Forwarding rule (non-async_chunk, or when the downstream stage does not receive chunks); see `⟨/⟩ engine/orchestrator.py:1988-1996`:

```
if ((finished or segment_finished)
        and stage_id < req_state.final_stage_id
        and not req_state.skip_legacy_stage_forward
        and (not self.async_chunk or not self._stage_receives_async_chunks(stage_id + 1))
        and (not self._next_stage_already_submitted(stage_id, req_state) or req_state.streaming.enabled)):
    ...  # → _forward_to_next_stage → process_engine_inputs → submit_initial / submit_update
```

All cleanup goes through a single exit, `_cleanup_request_ids` (`⟨/⟩ engine/orchestrator.py:1699-1762`): pop the state, release replica bindings on all pools, and queue resource release (including SHM).

6.2 Ordering guarantees and terminal-message monotonicity

Each request's outputs are ordered, guaranteed by a FIFO chain:

1. On each stage, a request is **sticky-bound** to one replica (`_request_bindings`, `⟨/⟩ engine/stage_pool.py:623`).
2. Each replica has exactly one reader, reading in ZMQ order.
3. There is exactly one dispatcher, so `request_states` is never mutated concurrently.
4. output_queue is a single FIFO, dispatched by a single frontend task.

Terminal monotonicity (once finished, no new outputs and no "resurrection"):

© 2026 kzhou003

- `finished=True` is set only when **all** final-output stages have completed (`⟨/⟩ engine/orchestrator.py:1937–1940`).
- Under async_chunk, if the final message arrives before all upstream stages have completed, it is parked in `pending_final_output` and released once `stage_submit_ts ⊆ finished_stage_ids` (`⟨/⟩ engine/orchestrator.py:2036–2040`). The reason: the data plane and control plane are separate paths, so completion signals can arrive out of order; emitting the terminal message too early would let cleanup reclaim upstream resources that are still writing SHM.
- If the original terminal message is swallowed, an empty terminal message is synthesized so `generate()` never hangs forever (`⟨/⟩ engine/orchestrator.py:1858–1887`).
- Late outputs after cleanup are dropped (`⟨/⟩ engine/orchestrator.py:1385–1393`); a `streaming_update` for a request that no longer exists is also dropped rather than resurrecting it (`⟨/⟩ engine/orchestrator.py:3150–3162`).

```
# orchestrator.py:1954–1967 ── async_chunk terminal-message gate
if (request_finished and self.async_chunk
        and not req_state.streaming.enabled
        and not set(req_state.stage_submit_ts).issubset(req_state.finished_stage_ids)):
    req_state.pending_final_output = message
    request_finished = False
else:
    await self.output_async_queue.put(message)
```

> **Where detokenization lives**
>
> Each LLM stage's pool has one `MultimodalOutputProcessor` (`⟨/⟩ outputs/output_processor.py:384`), shared by all replicas of that stage and **running on the orchestrator thread**. `submit_initial` first calls `add_request`, then sends over ZMQ, and rolls back if the send fails (`⟨/⟩ engine/stage_pool.py:1053–1080`). This corresponds to the role of OutputProcessor in upstream AsyncLLM, just moved from the frontend loop to the orchestrator loop.

---

7. Inside a stage process

Each stage is a `StageEngineCoreProc`, inheriting upstream `EngineCoreProc`. **vllm-omni does not override `step`, `step_with_batch_queue` or `run_busy_loop`**; the main loop is entirely upstream. All omni extensions are done through "insertion points":

Extension point	What / why	Source
Request type swap	Before `__init__`, replaces the module-global `EngineCoreRequest` with `OmniEngineCoreRequest`. The reason: the upstream input IO thread creates `MsgpackDecoder(EngineCoreRequest)` inside `__init__`, so swapping later would make `additional_information` undecodable.	`⟨/⟩ engine/stage_engine_core_proc.py:255–258`
Payload field pass-through	`preprocess_add_request` copies `additional_information`, `external_req_id`, `payload_sender_info` onto the scheduler's `Request`	`⟨/⟩ engine/stage_engine_core_proc.py:171–177`
Abort fence	`release_overflowed_requests` is called only after ABORT goes through FIFO dispatch. Upstream's "eager abort queue" might run before an earlier ADD; only the FIFO position guarantees ordering and allows ID reuse.	`⟨/⟩ engine/stage_engine_core_proc.py:147–154`

© 2026 kzhou003

Signal wakeup	On SIGTERM, sets `shutdown_state` and pushes a `WAKEUP` into `input_queue` to wake the busy loop blocked on the queue	⟨/⟩ engine/stage_engine_core_proc.py:311–322
Same-process sinks	The first-audio sink writes the upstream `output_queue` directly, bypassing the scheduler; the native data plane's ready sink writes the scheduler inbox directly (only for UniProc, TP=1, PP=1, because only then are the runner and scheduler in the same process)	⟨/⟩ engine/stage_engine_core_proc.py:84–119
SHM reclamation	`omni_release_request_resources` on the UTILITY channel only enqueues; the actual unlink runs on the adapter's save thread	⟨/⟩ engine/stage_engine_core_proc.py:156–169

Scheduler selection: AR stages (thinker/talker) use `OmniARAsyncScheduler(OmniARScheduler, AsyncVLLMScheduler)` (⟨/⟩ core/sched/omni_ar_scheduler.py:1753); code2wav uses the synchronous `OmniGenerationScheduler` (⟨/⟩ config/stage_config.py:201–218).

7.1 Scheduler hooks and WAITING_FOR_CHUNK

omni adds two states to `RequestStatus` (⟨/⟩ patch.py:364–372): `WAITING_FOR_CHUNK = -1` and `WAITING_FOR_INPUT = -2`. They are negative so that upstream's existing `status > X` comparisons still treat them as "not finished".

Every `schedule()` call wraps upstream scheduling with pre/post hooks (⟨/⟩ core/sched/omni_ar_scheduler.py:408–451):

```
def schedule(self, throttle_prefills=False):
    self._drop_aborted_queued_requests()          # upstream raises on ABORTED status, so clear those first
    self._process_pending_omni_inputs(model_mode="ar")  # ① park: move requests without data out of the queues
    ...
    try:
        scheduler_output = super().schedule(throttle_prefills)  # ② upstream scheduling: only sees requests "with data"
    finally:
        ...
        self._restore_omni_wait_queues()           # ③ restore: put requests whose data arrived back into waiting/running
    self._postprocess_omni_schedule_output(...)    # ④ attach additional_information
```

The key is `process_pending_chunks` (⟨/⟩ distributed/omni_connectors/transfer_adapter/chunk_transfer_adapter.py:1120–1186): for requests that do not yet have a chunk, it calls `load_async` so a background thread fetches it, and **physically removes the request from waiting/running** into a private deque (⟨/⟩ distributed/omni_connectors/transfer_adapter/chunk_transfer_adapter.py:1431–1463).

> **Why "remove from the queue" instead of "skip"**
>
> - The upstream scheduler knows nothing about omni states. If a request without data stayed in the queue, it would be scheduled and consume token budget and KV, with no input at forward time. After removal, upstream `super().schedule()` only ever sees requests whose "data is ready", so upstream semantics need no changes (corresponding to AR-INV-001: vLLM owns the base scheduling semantics).
> - Restoring in `finally` guarantees no request is lost even if upstream scheduling raises. Aborts are cleaned once more before restoring, closing the race where "an abort lands exactly between process and restore" (#3736, ⟨/⟩ distributed/omni_connectors/transfer_adapter/chunk_transfer_adapter.py:1488–1532).
> - After receiving a chunk, the recv thread **does not modify the request's prompt directly**; instead it is finalized on the scheduler thread (`_apply_pending_ar_prompt_updates`, ⟨/⟩ distributed/omni_connectors/transfer_adapter/chunk_transfer_adapter.py:1281–1330). Only there are the request's token counters a consistent snapshot.

© 2026 kzhou003

`active_stream_window`: K = min(window, max_num_seqs) (`distributed/omni_connectors/transfer_adapter/chunk_transfer_adapter.py:162-166`). Inactive running requests are moved to `_held_non_active` rather than via `waiting_queue.prepend_requests`, because calling the latter mid-step trips an invariant assertion in the upstream LogitsProcessor (`distributed/omni_connectors/transfer_adapter/chunk_transfer_adapter.py:210-216`).

> **Interaction with async scheduling**
> The sender's "confirmed token count" = `num_computed_tokens - num_output_placeholders` (`distributed/omni_connectors/transfer_adapter/chunk_transfer_adapter.py:244-251`). In other words, **placeholder tokens that have not actually been sampled are never sent downstream as chunks**. This is the key line for correctness when async scheduling is combined with streaming transfer.

---

8. async_chunk data plane

Layering: the **Connector** handles only transport (`put/get`, no knowledge of request semantics); the **TransferAdapter** manages the chunk lifecycle (keys, ordering, park/restore, cleanup); the **stage input processor** (model-specific functions) decides what to send.

8.1 OmniChunkTransferAdapter: two background threads

The base class (`distributed/omni_connectors/transfer_adapter/base.py:20-46`) starts a `recv_thread` and a `save_thread`, each with a `deque` and a `threading.Condition`:

```
# transfer_adapter/base.py:59-82 —— recv_loop
while not self.stop_event.is_set():
    n = len(self._pending_load_reqs)
    for _ in range(n):
        request = self._pending_load_reqs.popleft()
        is_success = self._poll_single_request(request)   # non-blocking connector.get
        if not is_success:
            self._pending_load_reqs.append(request)       # not there yet: back to the tail, round-robin
    with self._recv_cond:
        if not self._pending_load_reqs and not self.stop_event.is_set():
            self._recv_cond.wait(timeout=0.1)             # idle: wait for notify
        elif not any_success and not self.stop_event.is_set():
            self._recv_cond.wait(timeout=0.001)           # nothing ready: 1 ms backoff to avoid spinning on shm_open
```

Enqueueing (append) does **not hold** the condition lock, so a race where "notify fires after append but before wait" is possible. The condition variable's timeout is the backstop for this race (`distributed/omni_connectors/transfer_adapter/base.py:77-82`). A typical trade-off: a small worst-case latency (≤0.1 s, ≤1 ms when there is work) in exchange for lock-free enqueue on the hot path.

Send path (who does what on which thread)

1. **GPU forward thread**: the runner puts `inter_stage_outputs` into `ModelRunnerOutput`.
2. **EngineCore thread**: `OmniARScheduler.update_from_output` calls `save_async(...)` (`core/sched/omni_ar_scheduler.py:880-901`). `save_async` only does **snapshot + enqueue**: shallow copies of the request, `prompt_token_ids`, `additional_information` and token lists (`distributed/omni_connectors/transfer_adapter/chunk_transfer_adapter.py:254-292`), because the scheduler will keep mutating the live request.

© 2026 kzhou003

3. **save thread**: runs the model's `custom_process_next_stage_input_func` (e.g. `talker2code2wav_async_chunk`), then `connector.put` (`distributed/omni_connectors/transfer_adapter/chunk_transfer_adapter.py:778-854`). Python-level payload shaping and SHM writes are entirely off the scheduling critical path.

> **Note · discrepancy with older docs**
> `docs/design/feature/async_chunk.md` says "the custom func builds the payload on the main thread". At the current HEAD, the V1 adapter path calls the processor on the save thread (the main thread only takes the snapshot). When reading the source, trust the code.

Locks and keys

- `_sender_state_lock`: protects `_SenderGeneration` (`in_flight`, `cancelled`, `terminal_pending`, `cleanup_deferred`, `distributed/omni_connectors/transfer_adapter/chunk_transfer_adapter.py:34-51`). Used only for short state transitions; neither payload building nor put happens inside the lock.
- `_receiver_state_lock`: serializes "register/cleanup" against "commit a received chunk". `connector.get()` runs outside the lock; after getting data, it **re-checks** `self._registered_load_entries.get(req_id) is entry` inside the lock (`distributed/omni_connectors/transfer_adapter/chunk_transfer_adapter.py:539-555`). This is a generation check: if the request was cleaned up or re-registered during the get, the late chunk is dropped. Cleanup thus acts as a "commit fence".
- Key rule: `{external_req_id}_{stage_id}_{chunk_id}` (send `distributed/omni_connectors/transfer_adapter/chunk_transfer_adapter.py:792`, receive `distributed/omni_connectors/transfer_adapter/chunk_transfer_adapter.py:516`). The chunk index is incremented inside the lock only after a successful put (`distributed/omni_connectors/transfer_adapter/chunk_transfer_adapter.py:856-874`). Deterministic keys let both sides match up without an extra metadata channel, and naturally guarantee chunk ordering: the receiver only fetches the next index.

Termination and cleanup ordering

- `cleanup_sender` never waits on IO: if the terminal chunk is still queued, it marks `cancelled + cleanup_deferred`; if a put is in progress, it leaves it to the sender's `finally` (`distributed/omni_connectors/transfer_adapter/chunk_transfer_adapter.py:991-1027`).
- `cancelled` does **not** block the terminal chunk, because it is the downstream's only end-of-stream signal. Blocking it would leave the downstream request stuck in WAITING_FOR_CHUNK forever (#6670, `distributed/omni_connectors/transfer_adapter/chunk_transfer_adapter.py:714-723`).
- The SHM unlink for an abort is queued as a `{"cleanup_shm": ...}` task on **the same save thread** (`distributed/omni_connectors/transfer_adapter/chunk_transfer_adapter.py:1029-1042`). Same-thread FIFO guarantees the unlink always happens after earlier puts, and the scheduler itself does no SHM IO.

8.2 SharedMemoryConnector

Step	Mechanism	Source
Write	Serialize the payload, take an exclusive `fcntl` lock on `/dev/shm/shm_{key}_lockfile.lock`, create a POSIX shared memory segment named after the key (unlink and recreate if it already exists)	`distributed/omni_connectors/connectors/shm_connector.py:195-228` `entrypoints/stage_utils.py:193-215`

© 2026 kzhou003

Read	Try to lock with `LOCK_EX|LOCK_NB`: `BlockingIOError` means the writer has not finished, treated as "not ready"; `ValueError("empty file")` means the segment has not been sized yet, also not ready	`distributed/omni_connectors/connectors/shm_connector.py:234-274`
Release	**The reader unlinks**: `shm_read_bytes` unlinks the segment immediately after copying the bytes out, then deletes the lockfile. The producer uses `reap_consumed` to round-robin check whether its pending keys have been consumed, bounding bookkeeping size	`entrypoints/stage_utils.py:223-237` `distributed/omni_connectors/connectors/shm_connector.py:358-364`
Wakeup	Each receiver has a named FIFO; on put, 1 byte is written to every FIFO of the target stage, and receivers wait in `select()`. The wakeup is only a hint; timed polling is the correctness backstop	`distributed/omni_connectors/connectors/shm_connector.py:81-174`
Fallback cleanup	Unconsumed segments are reclaimed by `cleanup`, `cleanup_prefix`, `close`; the orchestrator drives `release_shm_resources` after all stages complete	`distributed/omni_connectors/connectors/shm_connector.py:312-356` `distributed/omni_connectors/transfer_adapter/chunk_transfer_adapter.py:1060-1078`

> **Why "file lock + reader unlinks"**
> A shared memory segment has no atomic "write finished" signal of its own. Using an exclusive lock on a same-named lockfile as the "write in progress" marker, with readers probing via a non-blocking lock attempt, yields a cross-process ready signal that needs no extra channel. Having the reader unlink frees the memory right after consumption (single-consumer semantics), and the producer need not wait for an ACK.

Other connectors (Mooncake RDMA, NIXL, Mori, Yuanrong, etc., `vllm_omni/distributed/omni_connectors/connectors/`) implement the same put/get interface for cross-node or GPU-direct scenarios. Edges with no configured connector automatically get a SharedMemoryConnector (`distributed/omni_connectors/utils/initialization.py:391-421`).

8.3 Runner-side connector runtime

Non-async_chunk full-payload mode, as well as MRv2, goes through the `omni-mixin-recv` / `omni-mixin-save` threads inside the model runner (`distributed/omni_connectors/model_runner/omni_connector_runtime.py:317-341`):

- All transfer state uses one `_lock`, plus **two separate** events: `_work_available` and `_save_work_available`. A comment explains that when a single event was shared, the receiver would "swallow" the sender's wakeups.
- The receive loop snapshots the wakeup generation **before polling**, then calls `wait_for_change(generation, timeout)`. So data arriving during polling is not lost because its wakeup was already consumed (`distributed/omni_connectors/model_runner/omni_connector_payload_transport.py:959-995`). Another classic lost-wakeup-safe pattern.
- Received payloads first go into `_local_stage_payload_cache` and are later materialized into runtime state on the model thread (`distributed/omni_connectors/model_runner/omni_connector_runtime.py:281-289`). V1 calls recv/flush

© 2026 kzhou003

synchronously at the start of `execute_model` (`worker/gpu_ar_model_runner.py:963-971`).

---

9. GPU-level concurrency: CUDA streams / events / graphs

Every step, the Thinker has to copy hidden states back to CPU to send to the talker. A synchronous copy on the compute stream would interrupt the GPU. vllm-omni implements **asynchronous output materialization** in the V1 AR runner (`worker/gpu_ar_model_runner.py:1670-1796`):

```
# gpu_ar_model_runner.py:180-189
source_stream = torch.cuda.current_stream()
ready_event = torch.cuda.Event()
with torch.cuda.stream(copy_stream):                # dedicated _omni_payload_copy_stream
    copy_stream.wait_stream(source_stream)          # wait only until "the data is computed"
    cpu_payload = _copy_tensor_payload_to_cpu(cloned, pin_memory)   # copy the clone (graph buffers get reused)
    for source in cuda_sources:
        source.record_stream(copy_stream)           # tell the caching allocator: don't reuse this memory until the copy is done
    ready_event.record(copy_stream)                 # completion signal
```

- **Clone first**: CUDA graph or model buffers are overwritten on the next step's replay.
- **Pinned host memory**: pageable memory degrades D2H into a synchronous copy. A comment records the lesson of ~240 ms/step (`worker/gpu_ar_model_runner.py:1730-1739`).
- `record_stream`: the PyTorch caching allocator only tracks the stream an allocation was made on; cross-stream use must be registered, or the memory may be reclaimed and reused before the copy completes.
- **Background builder thread** `omni-async-output-builder` waits on `ready_event` and then assembles the payload; `get_output()` joins it and re-raises any builder exception (`worker/gpu_ar_model_runner.py:200-297`). Scheduler output, token lists, etc. are snapshotted first (`worker/gpu_ar_model_runner.py:2200-2207`).
- **Exception**: the connector-output drain stays on the main thread, because it may run TP collectives (collectives must happen in the same order on all ranks, `worker/gpu_ar_model_runner.py:2228-2231`).
- Enabled when: async scheduling is on, no spec decode, async_chunk is on, and the model declares `use_async_omni_output` (`worker/gpu_ar_model_runner.py:1670-1692`).

Sampled tokens are handed to the input batch together with a ready event (an upstream mechanism). omni synchronizes on this event before patching the `-1` placeholders (`worker/gpu_model_runner.py:357-361`). The Generation runner (code2wav) has a similar async snapshot; the clone is skipped only when the model owns its storage and runs in eager mode (`worker/gpu_generation_model_runner.py:74-140`).

> **Three CUDA streams in one stage process**
> The compute stream (forward + sampling), upstream's `async_output_copy_stream` (D2H of sampled tokens), and omni's `_omni_payload_copy_stream` (D2H of cross-stage payloads). Events and `wait_stream` establish only minimal dependencies among them, so the next step's forward can overlap with both kinds of copy from the previous step.

© 2026 kzhou003

10. Correctness: abort / failure / shutdown / RPC

10.1 Abort propagation (client disconnect)

1. `generate` catches `CancelledError` / `GeneratorExit` and runs `_abort_internal_requests` under `asyncio.shield`, bounded by `ABORT_TIMEOUT_S` (⟨/⟩ entrypoints/async_omni.py:325–332). The shield ensures the abort itself is not cut short by cancellation of the outer task.
2. `abort_async` generates an `rpc_id` and, on an executor thread, runs `CorrelatedRpcClient.execute(("abort", rpc_id))` (⟨/⟩ engine/omni_engine_base.py:1037–1054).
3. The Orchestrator's `_handle_abort` calls `_cleanup_request_ids(abort=True)`: expands CFG companions, fails orphaned parents, then runs `abort_requests` on each pool (⟨/⟩ engine/orchestrator.py:645–731).
4. Within each pool, requests are grouped by bound replica: **first** partial outputs are collected from the output processor (`commit_state=False`), **then** EngineCore's abort is called, and **only on success** is the processor's abort state committed (⟨/⟩ engine/stage_pool.py:1279–1339). This two-phase commit avoids the inconsistency where "the processor thinks it aborted but the engine never got the message".
5. With multiple final stages, only the last abort output carries `finished` (⟨/⟩ engine/orchestrator.py:726–730).
6. The frontend receives the abort output; if nothing arrives, it synthesizes a terminal message with `finish_reason="abort"` so `generate` never hangs (⟨/⟩ entrypoints/async_omni_base.py:460–493).

10.2 Fault isolation

- **A stage process dies**: the upstream monitor thread marks it dead, `get_output_async` raises `EngineDeadError`, the reader puts `("error", …)` into `ready_q`, and the dispatcher runs `_handle_dead_replica` to evict that replica (⟨/⟩ engine/orchestrator.py:1464–1515, ⟨/⟩ engine/stage_pool.py:1479–1496).
- **Only affected requests fail**: those bound to the dead replica; if the stage has no live replica left, all requests that pass through that stage. Each request gets a request-level `ErrorMessage(fatal=True, request_id=…)`, and the service itself keeps running.

```
# orchestrator.py:1485–1497
for req_id, req_state in list(self.request_states.items()):
    if stage_id not in req_state.stage_submit_ts: continue
    bound = pool.get_bound_replica_id(req_id)
    if bound == replica_id or not stage_has_live:
        await self.output_async_queue.put(ErrorMessage(
            error=str(error), fatal=True, request_id=req_id, stage_id=stage_id))
```

- **The orchestrator thread itself crashes**: a fatal `ErrorMessage` with no request_id is put on both the output sink and the RPC queue (⟨/⟩ engine/omni_engine_base.py:486–491). The frontend broadcasts the error to all request queues, and the router broadcasts it to all RPC waiters and latches the error (⟨/⟩ engine/rpc_result_router.py:75–110).

10.3 Shutdown order

1. Frontend: cancel `final_output_task`, then call `engine.shutdown()` (⟨/⟩ entrypoints/async_omni_base.py:570–573).
2. `OmniEngineBase.shutdown` (⟨/⟩ engine/omni_engine_base.py:1132–1220): send `ShutdownRequestMessage` (1 s timeout) → close the RPC router → join the orchestrator thread (up to 30 s) → close the janus queues → `runtime.shutdown()` → release the CuMem pool.
3. Inside the Orchestrator: first set `_shutdown_event` and pre-mark clients as `_shutting_down`, so the monitor does not misreport a normal exit as dead (⟨/⟩ engine/orchestrator.py:576–588); the finally block of `run()` cancels tasks,

© 2026 kzhou003

then calls `shutdown_replica` one by one (`engine/orchestrator.py:530-553`).

Ordering principle: **stop the orchestrator first, then the runtime**, so that during shutdown nobody is still sending messages to an already-closed client.

10.4 RPC correlation

```
# rpc_result_router.py:136-150 — synchronous caller (executor thread)
waiter = self._router.register(key)          # one queue.Queue(maxsize=1) per key
try:
    if block_on_submit: self._request_queue.put(message)
    else:               self._request_queue.put_nowait(message)
    result = waiter.get(timeout=timeout)
    if isinstance(result, ErrorMessage): raise RuntimeError(result.error)
    return result
finally:
    self._router.unregister(key, waiter)     # late results are dropped
```

The key is `("collective", rpc_id)` or `("abort", rpc_id)` (`engine/messages.py:125-148`). **Register before sending**, so a result can never arrive before registration. `pause_generation`, `sleep`, and `wake_up` first gate admission of new requests with an `asyncio.Condition` (`_pause_cond`) and wait for `_admitting == 0`, then send the RPC (`entrypoints/async_omni.py:972-995`). Because request_queue is FIFO, any add enqueued earlier is guaranteed to be processed before the sleep.

---

11. Startup-time concurrency

When several stages share a GPU, the biggest risk of concurrent initialization is **memory profiling interference**: vLLM sizes the KV cache by "measuring free memory", and if another stage happens to be loading weights at the same time, the measurement is wrong, leading to OOM or an undersized KV cache.

- **Group by device**: `_init_group_keys` groups stages into "connected components of overlapping devices" (`engine/stage_runtime.py:1093-1134`, `engine/stage_init_utils.py:1738-1762`). The default Qwen3-Omni deployment yields two groups: `device-group:0` (thinker) and `device-group:1` (talker then code2wav, serial within the group). The two groups start in parallel on the `stage-init` thread pool (`engine/stage_runtime.py:879-956`). The first failure is recorded under `init_state_lock` and makes subsequent groups short-circuit.
- **Serial mode**: the parent process takes an exclusive `flock` per device (`engine/stage_init_utils.py:1765-1900`) and releases it after the handshake; environment variable changes at spawn time (`CUDA_VISIBLE_DEVICES`) are serialized with `_replica_launch_lock`, because environment variables are process-global state.
- **Parallel mode** (`parallel_stage_init`): first `check_admission` does a per-device memory-ledger feasibility check (`engine/stage_admission.py:205`). Inside the child process, `DevicePhaseLock` (shared/exclusive flocks, sorted by device, fail-closed) is used: loading weights, allocating KV, and capturing graphs hold the shared lock; `determine_available_memory` holds the exclusive lock (`engine/stage_phase_lock.py:106-238`). This is essentially a cross-process reader-writer lock: multiple stages can modify memory concurrently, but measurement must be exclusive.

© 2026 kzhou003

12. MRv2 native data plane (brief)

When the deploy sets `model_runner: v2` and a stage declares `supports_native_mrv2_data_plane` (all three Qwen3-Omni stages declare it, but the default yaml does not enable it), the scheduler no longer creates a ChunkTransferAdapter; transport is handled instead by `OmniRunnerDataPlane` inside the runner (`⟨/⟩ core/sched/omni_scheduler_mixin.py:204-222`):

- A bounded `Queue(maxsize=8)` plus a single ordered output worker (`⟨/⟩ worker_v2/omni_data_plane.py:106-206`).
- `_native_send_lock` is acquired **before** releasing the state lock (hand-over-hand), so an abort cannot slip in ahead of an output (`⟨/⟩ worker_v2/omni_data_plane.py:362-382`).
- The terminal marker is emitted only after the in-flight count drops to zero; `DeliveryTicket` uses an event to provide an awaitable delivery acknowledgment (`⟨/⟩ worker_v2/delivery.py:34-74`).
- `FirstAudioSender`: the first audio frame gets a pinned D2H copy and a recorded event, and a background thread `put_nowait`s it directly into EngineCore's `output_queue`, bypassing both the scheduler and the busy loop to squeeze TTFP further (`⟨/⟩ worker_v2/first_audio_sender.py:79-154`).

13. Design trade-offs and known risks

Point	Explanation
Blocking put on the frontend loop	`add_request` does a blocking put on `request_queue.sync_q` (`⟨/⟩ engine/async_omni_engine.py:584`). When the queue is full at 256, it **blocks the entire frontend event loop**, stalling both HTTP and streaming output.
RPCs block admission	`_handle_collective_rpc` runs inline in the request-handler task (`⟨/⟩ engine/orchestrator.py:753-817`). A slow sleep/wake RPC makes every add/abort queued behind it wait.
Shared state relies on cooperative scheduling	`request_states` is shared by two tasks, the request-handler and the dispatcher; safety depends entirely on "same loop and no await inside critical sections". Be extra careful when inserting awaits on these paths.
Condition timeout as backstop	The adapter's lock-free append plus notify relies, in the worst case, on a 0.1 s or 1 ms timeout to be woken. This is a deliberate latency-vs-throughput trade-off.
Linear forwarding in the orchestrator	The forwarding target is always `stage_id + 1` (`⟨/⟩ engine/orchestrator.py:2341`, `⟨/⟩ engine/orchestrator.py:2357`). `input_sources` is used only for connector edges and default input processing. The stage graph looks like a DAG in config, but at runtime it is a chain.
Docs vs. code discrepancies	async_chunk.md's description of the payload-building thread, and the output materialization doc's description of where the drain happens, do not match the HEAD code (see §8, §9). Also, the load-based dynamic initial chunk for Qwen3-Omni (`compute_dynamic_initial_chunk_size`) is actually used only in qwen3_tts; Qwen3-Omni uses a fixed 4 frames or an optional ramp.

14. Recommended source reading path

1. **Topology and config**: `⟨/⟩ model_executor/models/qwen3_omni/pipeline.py:22-84` → `⟨/⟩ deploy/qwen3_omni_moe.yaml:15-77` → `⟨/⟩ config/stage_config.py:1256-1320` (`merge_pipeline_deploy`)

© 2026 kzhou003

2. **Threads and queues**: `⟨/⟩ engine/omni_engine_base.py:300-340` → `⟨/⟩ engine/async_engine_utils.py:40-97` → `⟨/⟩ engine/omni_engine_base.py:973-991`
3. **Orchestrator main loop**: `⟨/⟩ engine/orchestrator.py:1099-1320` → `⟨/⟩ engine/orchestrator.py:1912-2046` (`_route_output`) → `⟨/⟩ engine/orchestrator.py:2711-2870` (prewarm) → `⟨/⟩ engine/orchestrator.py:3062-3162` (add request)
4. **Stage process**: `⟨/⟩ engine/stage_engine_core_proc.py:133-345` → `⟨/⟩ core/sched/omni_ar_scheduler.py:408-451` → `⟨/⟩ core/sched/omni_ar_scheduler.py:880-901`
5. **Chunk data plane**: `⟨/⟩ distributed/omni_connectors/transfer_adapter/base.py:20-140` → `⟨/⟩ distributed/omni_connectors/transfer_adapter/chunk_transfer_adapter.py:381-503` (save_async) → `⟨/⟩ distributed/omni_connectors/transfer_adapter/chunk_transfer_adapter.py:1120-1186` (process_pending_chunks) → `⟨/⟩ distributed/omni_connectors/connectors/shm_connector.py:195-310`
6. **Model-side processors**: `⟨/⟩ model_executor/stage_input_processors/qwen3_omni.py:409-542` (thinker→talker) → `⟨/⟩ model_executor/stage_input_processors/qwen3_omni.py:765-842` (talker→code2wav)
7. **GPU overlap**: `⟨/⟩ worker/gpu_ar_model_runner.py:160-297` → `⟨/⟩ worker/gpu_ar_model_runner.py:1670-1796`
8. **Correctness**: `⟨/⟩ engine/orchestrator.py:645-731` (abort) → `⟨/⟩ engine/stage_pool.py:1279-1339` → `⟨/⟩ engine/orchestrator.py:1464-1515` (dead replica) → `⟨/⟩ engine/omni_engine_base.py:1132-1220` (shutdown)
9. **Upstream counterparts**: `vLLM ↗ core.py (EngineCoreProc)` `vLLM ↗ core_client.py (AsyncMPClient)` `vLLM ↗ async_scheduler.py` `vLLM ↗ output_processor.py`

Related design docs: `⟨/⟩ docs/design/feature/async_chunk.md` `⟨/⟩ docs/design/module/engine_orchestration.md` `⟨/⟩ docs/design/module/stage_runtime.md` `⟨/⟩ docs/design/feature/omni_async_output_materialization.md`

© 2026 kzhou003