READ MODE · ANSWER FIRST

How to design a ChatGPT-scale GPU serving platform

Design an LLM GPU serving platform across streaming SLOs, model memory, parallelism, KV cache, scheduling, resilience, autoscaling, and cost.

CHEAT SHEET · 28 ESSENTIAL IDEAS

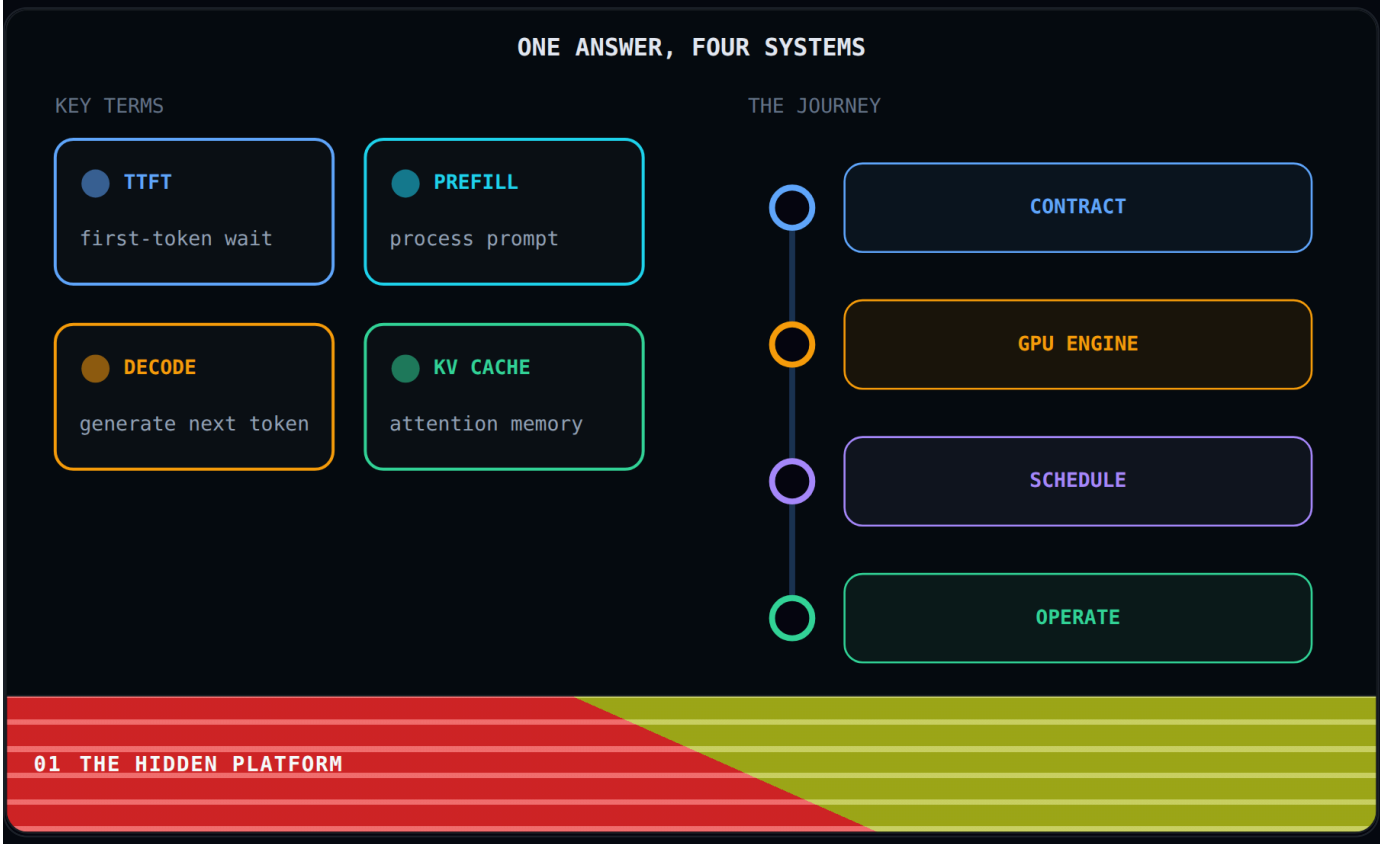
The whole story in 28 lines

A production LLM platform turns a streaming contract into safe GPU admission, phase-aware scheduling, resilient capacity, and measurable...

1. Streaming sends ordered partial events under one response identity and ends exactly once.
2. Time to first token and inter-token delay protect different parts of the user experience.
3. Tail lengths and burst distributions drive capacity more safely than one average request.
4. A stable control snapshot keeps the token data path independent from slow fleet changes.
5. Authentication, quotas, and safety reject unsuitable work before it consumes GPU capacity.
6. Routing checks compatibility first, then balances load and useful cache affinity.
7. Compute, HBM capacity, memory bandwidth, and interconnects constrain inference separately.
8. Weight precision changes device count and the HBM left for runtime state.
9. A loaded model is not ready until kernels, communication, and representative shapes are warm.
10. The runtime receives an ordered token sequence plus explicit generation limits.
11. Prefill processes known prompt positions in parallel and creates the initial KV state.
12. Decode accepts one token per iteration because each choice becomes the next input.
13. Prefill can fill compute while narrow decode often waits on memory movement.
14. Each token stores paired key and value state across every layer and KV head.
15. KV memory grows linearly with tokens and turns context limits into HBM obligations.
16. Tensor parallelism shards layers but introduces collectives inside model execution.
17. Pipeline parallelism assigns layer ranges and trades communication for fill and drain bubbles.
18. Data parallel replicas scale independent request throughput while duplicating weights and caches.
19. Topology-aware placement keeps frequent collectives on the fastest local links.
20. Safe admission reserves future KV growth before a request enters the active set.
21. Iteration-level scheduling replaces finished sequences instead of leaving batch holes.
22. Chunked prefill bounds decoder stalls while preserving the total prompt work.
23. Paged KV allocation reuses scattered blocks and avoids contiguous maximum reservations.
24. Prefix caching reuses only exact leading blocks and computes every unmatched suffix.
25. Separate prefill and decode pools trade lower interference for KV transfer and complexity.
26. Autoscaling must predict ready capacity because model loading and warm-up take time.
27. A partial stream ends explicitly, while a retry starts under a new response identity.
28. Correlated traces connect user SLOs to GPU pressure, failures, utilization, and cost.

STAGE 1 OF 30

The hidden platform



You press Send during a traffic spike, but the answer still needs to feel immediate. Behind that moment, a platform must protect latency, safety, memory, availability, and cost together.

TTFT means time to first token. Prefill reads the prompt, decode adds output tokens, and the KV cache remembers attention state needed by later iterations.

Our journey crosses four chapters: workload contracts, GPU mechanics, scheduling and memory, then resilient fleet operations. Every stage follows one illustrative request through those decisions.

The four chapters now connect around one streamed response. Let us begin with the public API contract that every hidden component must preserve.

STAGE 2 OF 30

API and stream contract

02 API AND STREAM CONTRACT

We start at the product boundary because every internal choice must preserve what clients receive. One request names a model, carries input, limits output, and chooses whether delivery streams.

With streaming enabled, the server sends ordered events under one response identity. Creation arrives first, text deltas follow, and a terminal event closes the response exactly once.

Streaming changes delivery timing while preserving one response identity and ordered events. This consequence follows the same illustrative request, so the resource tradeoff stays concrete across stages.

Generation and delivery can overlap, but clients still need a reliable ending. Which event should make the response final?

PAUSE AND PREDICT

Which event should make the response final?

- Any text delta
- The terminal completion event
- The creation event

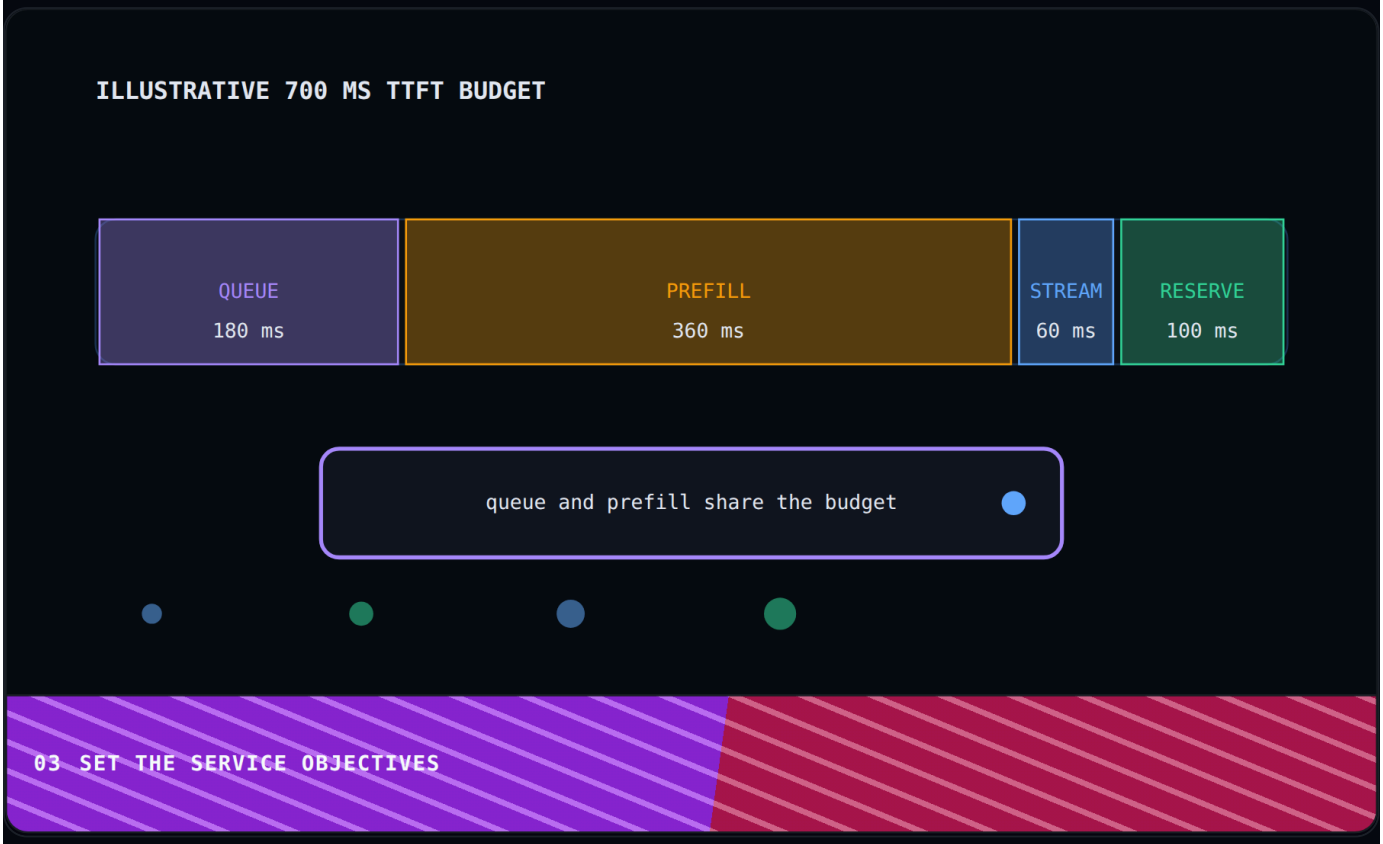
Skip this check

The event tape now opens before generation finishes, so useful text can arrive early. Its ordered cells still lead to one terminal state, which prevents ambiguous completion or hidden replay.

The contract is the platform invariant. Internals may batch, retry, or move work, but clients receive ordered deltas and one explicit ending. Next, we turn that promise into measurable objectives.

STAGE 3 OF 30

Set the service objectives



03 SET THE SERVICE OBJECTIVES

Now that the stream contract is clear, we can measure its experience. Time to first token includes queueing and prefill, while inter-token latency measures the gaps during decode. A single latency number cannot protect both the first token and the rest of the stream. This consequence follows the same illustrative request, so the resource tradeoff stays concrete across stages.

Throughput counts useful tokens across the fleet, availability tracks successful service, and cost prices the result. Improving one metric can quietly damage another, so each needs its own target.

A burst has consumed most queue headroom before GPU work begins. Which policy leaves more time for the first token?

PAUSE AND PREDICT

Which policy protects the first token during a burst?

- Balanced budget
- First token priority

Skip this check

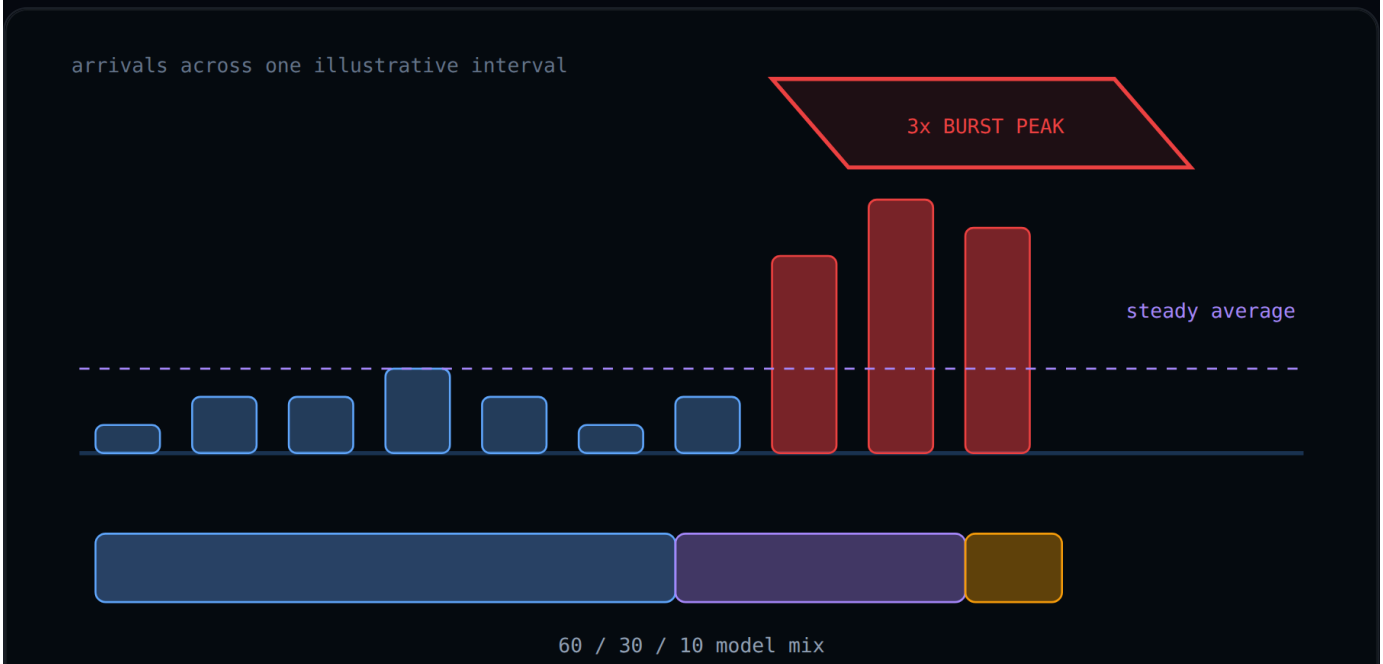
The same illustrative 700 milliseconds now splits into different proportional spans. First-token priority shortens queue allowance, while balanced policy shares more time between waiting and prefill.

Switch the Latency policy through both choices. Compare the downstream queue deadline and notice that the total budget stays fixed while its internal protection changes.

An SLO is a budget for design decisions, not a decorative dashboard number. Next, we model the distributions that spend those budgets under ordinary traffic and sudden bursts.

STAGE 4 OF 30

Model the workload



04 MODEL THE WORKLOAD

Our objectives need a workload shape. The fixture has short everyday prompts, a long p95 tail, varied output lengths, and three model classes sharing the same entry service.

Prompt length drives prefill work and initial KV allocation. Output length keeps decode slots alive, so two requests with equal total tokens can stress the platform in different phases.

Tail lengths and synchronized bursts determine memory pressure long before the average looks dangerous. This consequence follows the same illustrative request, so the resource tradeoff stays concrete across stages.

The steady rate looks safe, but a synchronized launch triples arrivals. Which capacity view exposes that risk before queues explode?

PAUSE AND PREDICT

Which view exposes burst risk?

- Only the average rate
- The full arrival distribution

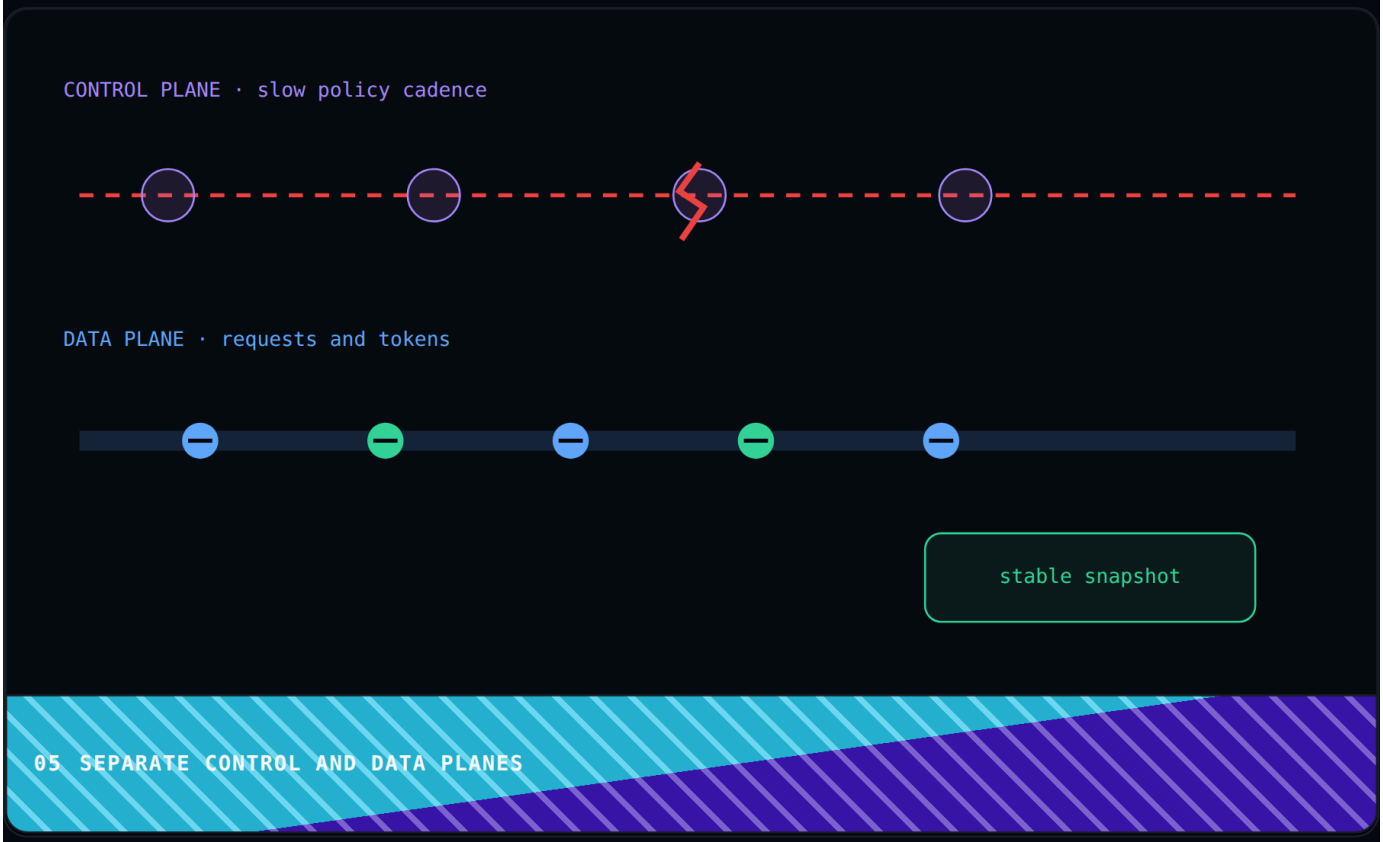
Skip this check

The arrival marks now gather into a three-times burst while the model mixture remains concurrent. Long prompts occupy prefill beside short chats, which is the interference our scheduler must manage.

Capacity planning begins with distributions because p95 memory and burst concurrency create the hard cases. Next, we separate slow fleet decisions from the fast path carrying these requests.

STAGE 5 OF 30

Separate control and data planes



05 SEPARATE CONTROL AND DATA PLANES

We just shaped the traffic. The next question is cadence, because fleet configuration changes slowly while every token event must travel through a short and predictable path.

The control plane publishes model versions, placement, quotas, and rollout policy. Workers consume a stable snapshot rather than consulting a central database during every decode iteration.

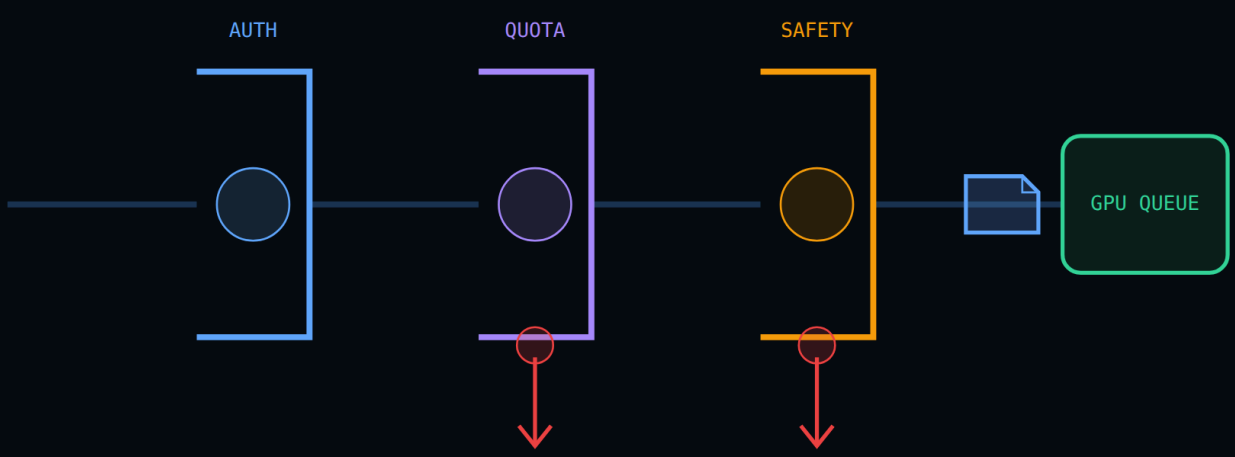
The data plane authenticates requests, routes them, schedules GPU work, and streams results. Its wide lane carries many concurrent events without waiting for a new placement decision.

The data plane can keep serving from a stable snapshot while control updates recover. This consequence follows the same illustrative request, so the resource tradeoff stays concrete across stages.

The control link now breaks while the existing data lane remains intact. New rollouts pause, but warm replicas continue serving from their last valid placement snapshot.

Plane separation contains failure and removes control chatter from the latency budget. Next, the request enters the first data-plane gate, where cheap checks protect expensive GPU capacity.

Authenticate, limit, and protect



06 AUTHENTICATE, LIMIT, AND PROTECT

With the planes separated, a blue request enters three cheap gates. Authentication establishes identity, which lets every later decision attach to one accountable tenant or session.

Quota checks requests and tokens because one small call and one enormous prompt consume different resources. The limiter reserves fairness before an abusive burst can occupy the shared queue.

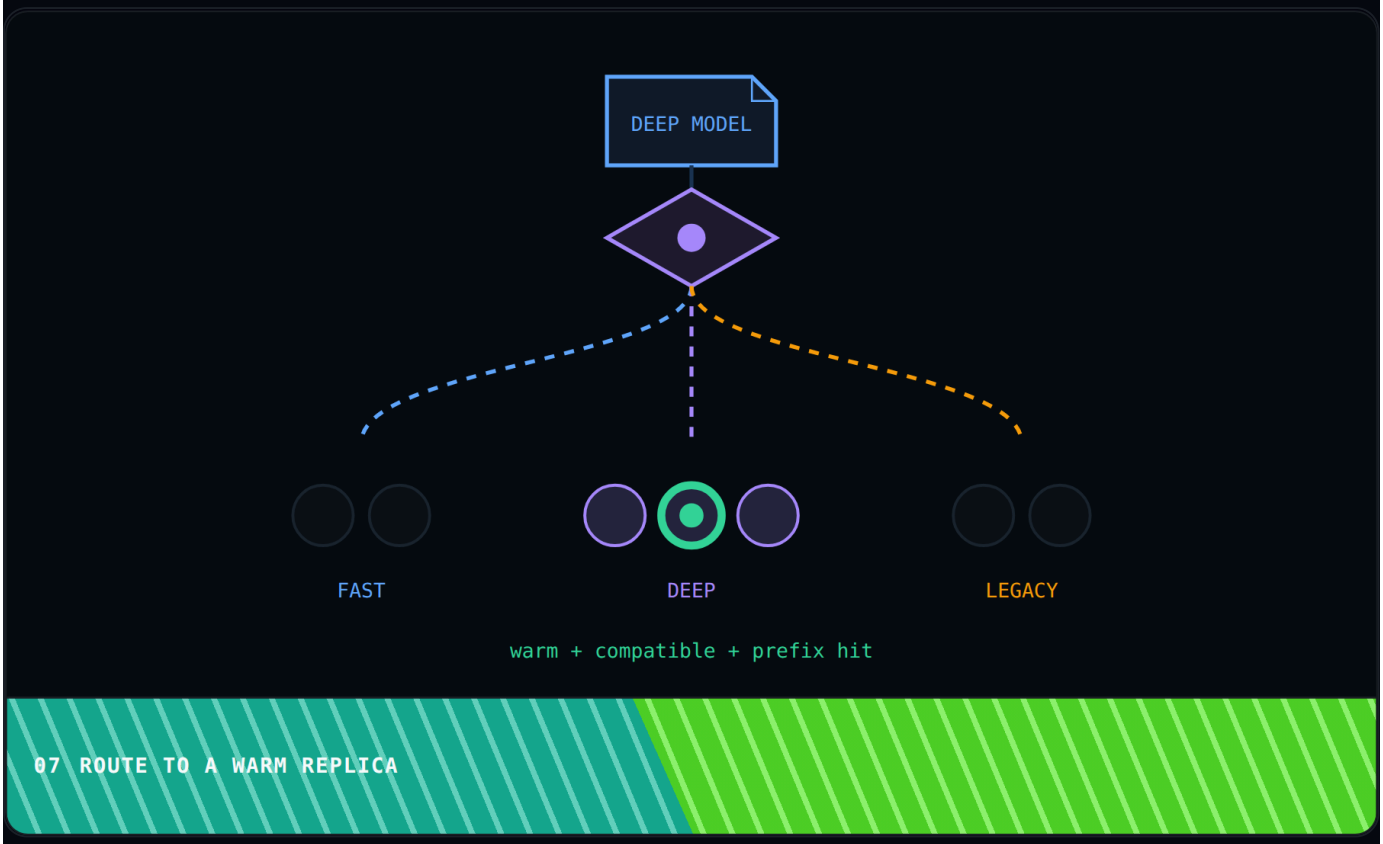
Safety policy can inspect input before inference and evaluate output as it streams. OpenAI documents moderation and privacy-preserving safety identifiers, but the exact internal path remains unknown.

Authentication, quotas, and safety are capacity controls as well as policy controls. This consequence follows the same illustrative request, so the resource tradeoff stays concrete across stages.

Two rejected cards now leave through red exits before the GPU queue. Only the accepted card advances, so scarce HBM and decode slots never carry obviously invalid work.

Early gates protect users, fairness, and capacity together. Next, the accepted request must find a compatible warm model replica without accidentally losing useful cache locality.

Route to a warm replica



07 ROUTE TO A WARM REPLICA

The accepted request carries a model name, capability needs, and generation limits. The router first removes replicas that cannot satisfy that contract, even when they look lightly loaded.

Three warm pools serve fast, deep, and legacy models. Each pool owns full replicas or compatible parallel groups, so routing never sends a request into mismatched weights.

Within the deep pool, load is not the only signal. A replica with the same cached prefix may avoid most prefill work, while session affinity can preserve active KV state.

Compatibility narrows the pool first, then load and prefix locality choose a warm replica. This consequence follows the same illustrative request, so the resource tradeoff stays concrete across stages.

The branch now lands on deep replica two because it is warm, compatible, and holds the prefix. Other branches remain available for requests with different model contracts.

Routing is a constrained choice rather than blind round robin. Next, we open the selected worker and inspect the compute, memory, and interconnect resources behind its capacity.

Read the GPU as a machine

We have reached one illustrative H100 worker. Tensor cores perform dense matrix operations, while streaming multiprocessors schedule many thread blocks across the chip.

Eighty GiB of HBM holds weights, runtime buffers, and KV pages. NVIDIA lists 3.35 terabytes per second of HBM bandwidth for the H100 SXM reference.

NVLink moves tensors between GPUs in a parallel group. Its published 900 gigabytes per second is fast, but collective communication still adds work and topology sensitivity.

A GPU can have idle math units while decode waits on weight and cache movement. This consequence follows the same illustrative request, so the resource tradeoff stays concrete across stages.

The cutaway now runs compute and memory together. Dense tiles fill the math region, while a narrow decode batch repeatedly sweeps the much larger weight ribbon from HBM.

Compute rate, memory capacity, bandwidth, and links are separate limits. Next, we measure the largest resident object first, which is the model weight set.

Size model weights

The GPU cutaway gives us an 80 GiB boundary. Our illustrative model has 70 billion parameters, so weight memory is parameter count multiplied by stored bytes per parameter. Precision changes both how many GPUs hold a model and how much HBM remains for KV state. This consequence follows the same illustrative request, so the resource tradeoff stays concrete across stages.

FP16 uses two bytes per parameter before metadata and runtime overhead. That creates 140 billion bytes of weights, which already exceeds one device boundary.

Lower precision shortens the same ribbon, but the parameter count never changes. Which format first fits the weights on one reference GPU?

PAUSE AND PREDICT

Which format first fits the weights on one 80 GiB GPU?

- FP16
- FP8
- INT4

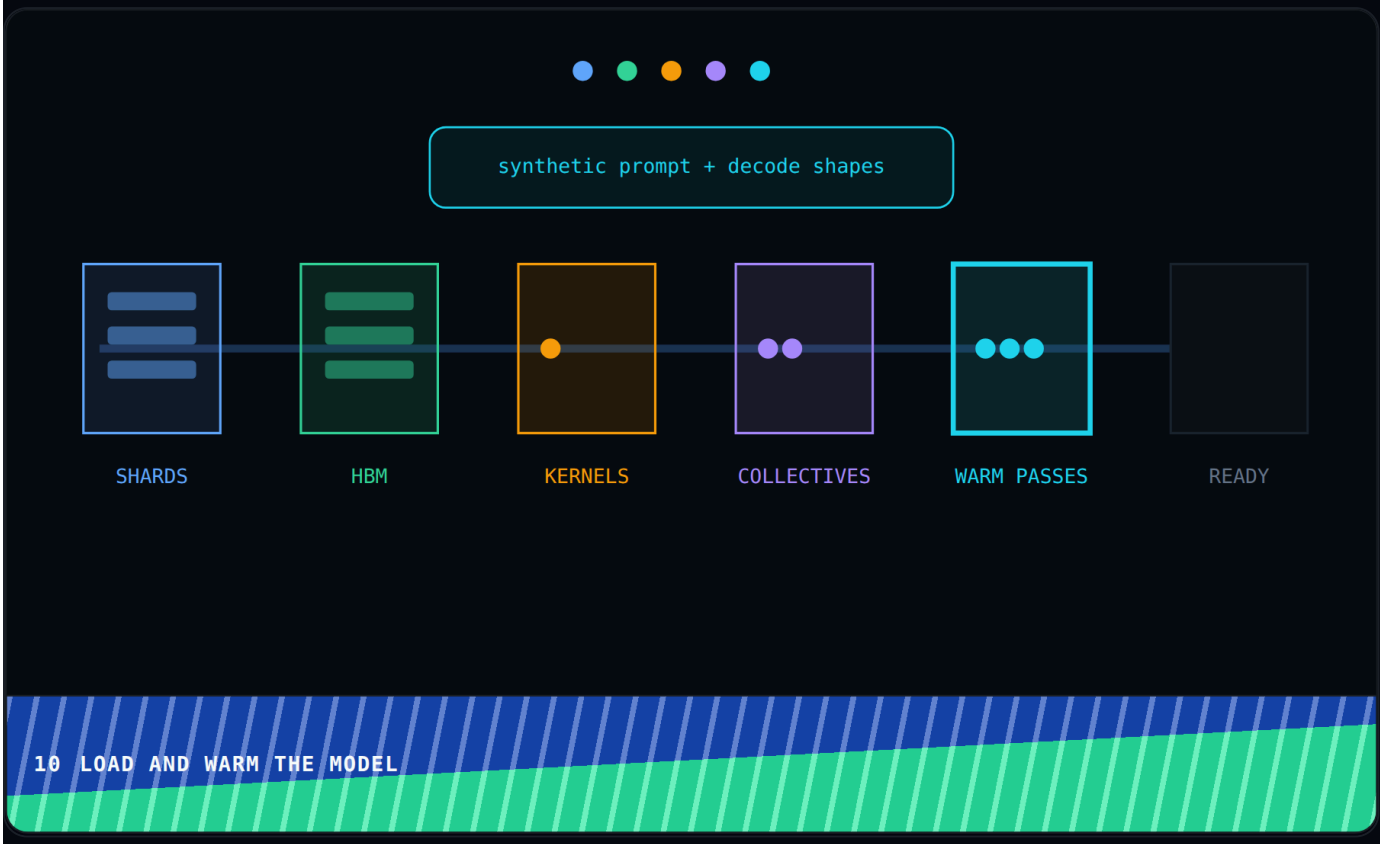
Skip this check

Equal-scale ribbons now cross the 80 GiB marker differently. FP16 spans two devices, FP8 fits with modest room, and INT4 leaves much more space for runtime state.

Try every Weight precision. Compare device count and HBM headroom, which later determine how many KV pages can stay resident.

Weight precision sets the floor for placement and the ceiling for cache concurrency. Next, we trace how those bytes become a genuinely ready model rather than a cold allocation.

Load and warm the model



Weight sizing told us where the model can fit. Startup begins by reading shards, validating them, and copying each shard into the HBM owned by its parallel worker.

The runtime then selects kernels, allocates workspaces, and prepares communication groups. Some shapes trigger compilation or autotuning, which makes a merely loaded process an unsafe serving target.

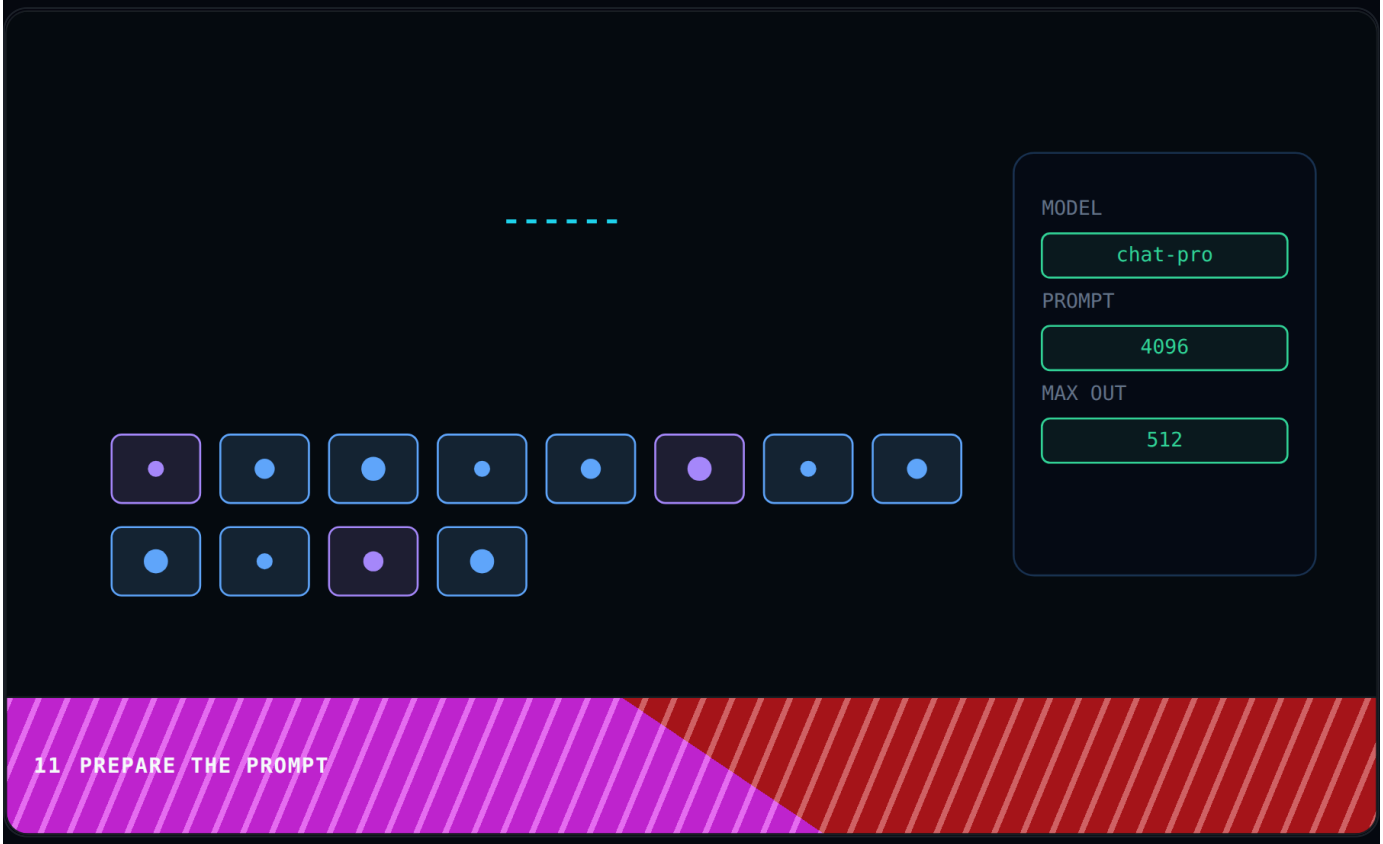
Representative warm-up passes exercise prompt and decode shapes before readiness. NVIDIA Triton documents model warmup specifically to avoid initial inference delays reaching live requests.

Readiness must wait for real warm-up work or the first user pays the cold-start cost. This consequence follows the same illustrative request, so the resource tradeoff stays concrete across stages.

The readiness gate opens only after HBM fills, kernel tiles lock in, and warm passes complete. The first real request now sees a prepared execution path.

A warm replica is an inventory item with a measurable startup delay. Next, our chosen request arrives and becomes the exact token sequence and limits consumed by admission.

Prepare the prompt



The replica is warm, so the request can become model input. Application messages still carry roles and content, but the causal model consumes one ordered sequence.

A model-specific template inserts control markers before tokenization maps text pieces to identifiers. Exact pieces vary by tokenizer, so this visual uses computed counts rather than invented token IDs.

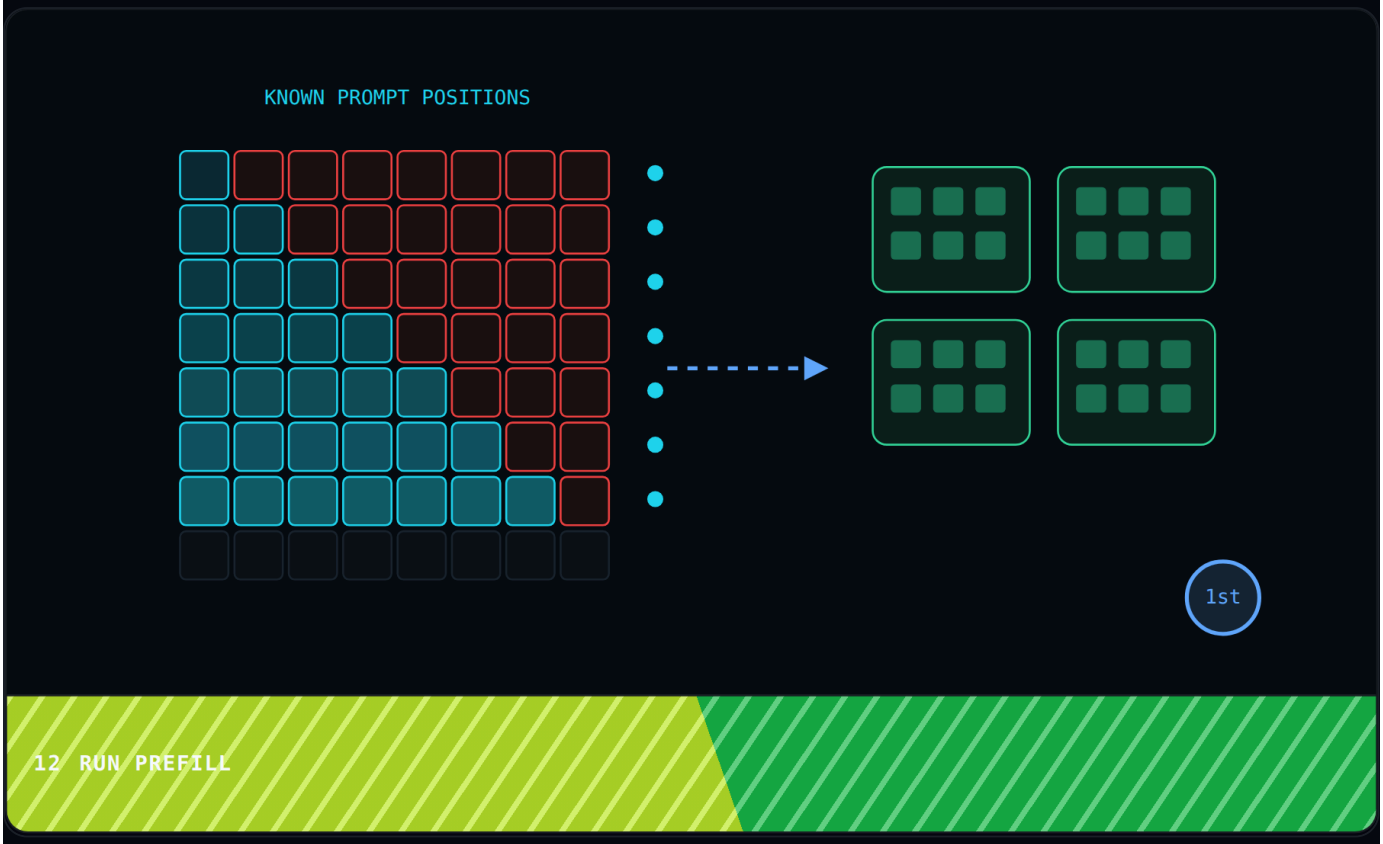
The sidecar preserves the model name, maximum output, tenant identity, and streaming choice. Those bounds let quota, admission, and scheduling reason about future work.

Admission receives one ordered token tape plus explicit generation limits, not chat bubbles. This consequence follows the same illustrative request, so the resource tradeoff stays concrete across stages.

The message cards now collapse into a 4096-token tape beside a 512-token output limit. This pair is the concrete request that every later memory calculation follows.

Prompt preparation creates deterministic input and declared limits. Next, prefill processes the entire known prompt and creates the first attention state for generation.

Run prefill



Our 4096-token tape enters prefill. Every prompt position is known before execution, so the runtime can process many positions together through dense matrix operations.

Causal attention still blocks future positions. The allowed lower triangle grows across the prompt, while matrix kernels expose enough parallel work to occupy the GPU compute tiles.

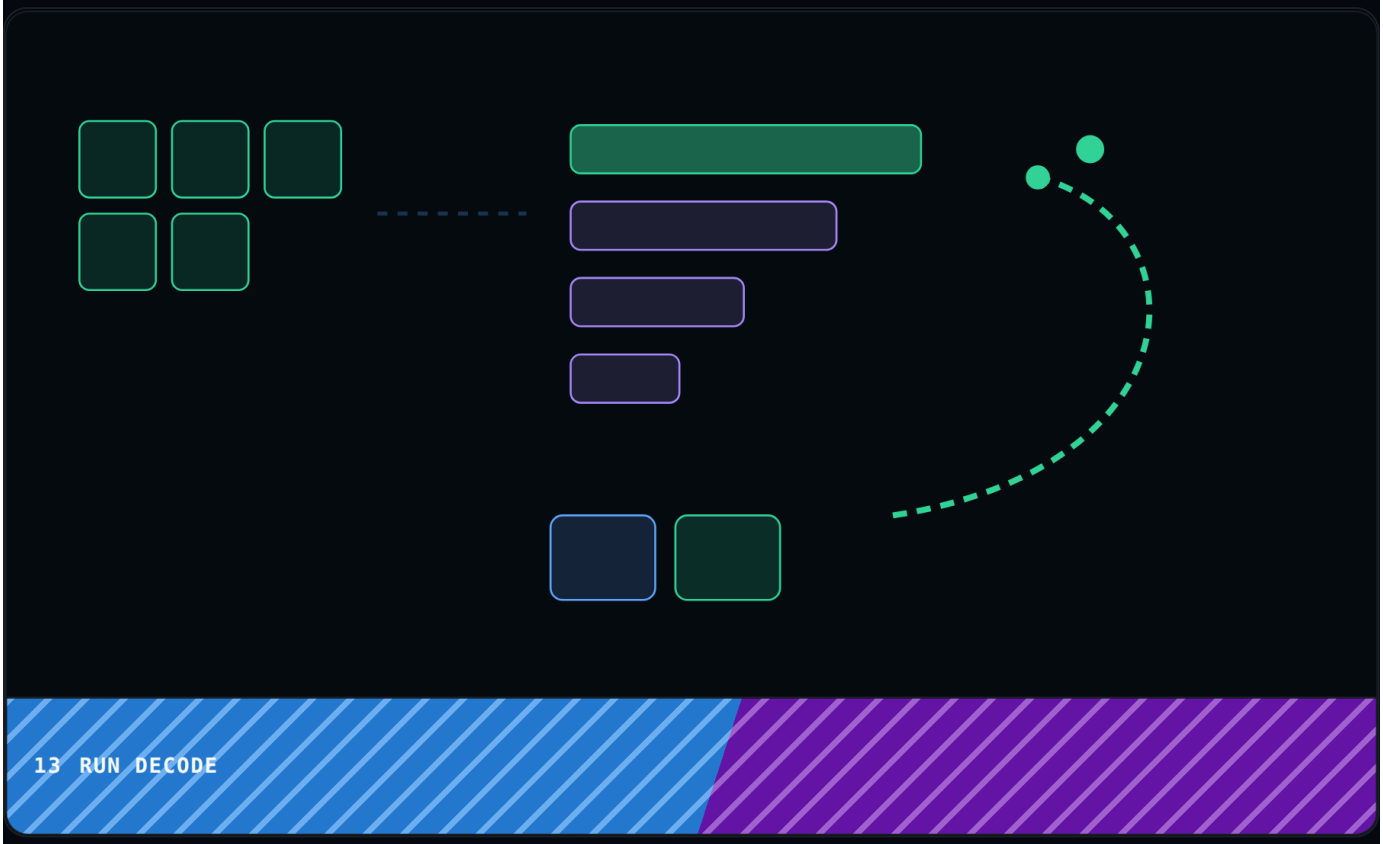
Each layer writes key and value state for every prompt position. The final position also produces logits used to choose the first generated token.

Prefill can use large matrix operations because every input position is already known. This consequence follows the same illustrative request, so the resource tradeoff stays concrete across stages.

The full triangle fills at once and releases an initial wall of KV pages. A first token can now be selected because the complete prompt state exists.

Prefill is parallel prompt processing with a large memory output. Next, decode consumes that state one iteration at a time and turns each accepted token into new input.

Run decode



Prefill left a complete prompt cache. Decode feeds the newest token through the model, reads earlier keys and values, and produces a score for each vocabulary candidate.

Sampling or another decoding rule selects one candidate. Many scores exist together, but only the accepted token becomes part of the authoritative sequence.

That token receives new key and value entries. The extended sequence becomes the input for the next iteration, so later tokens depend on choices already made.

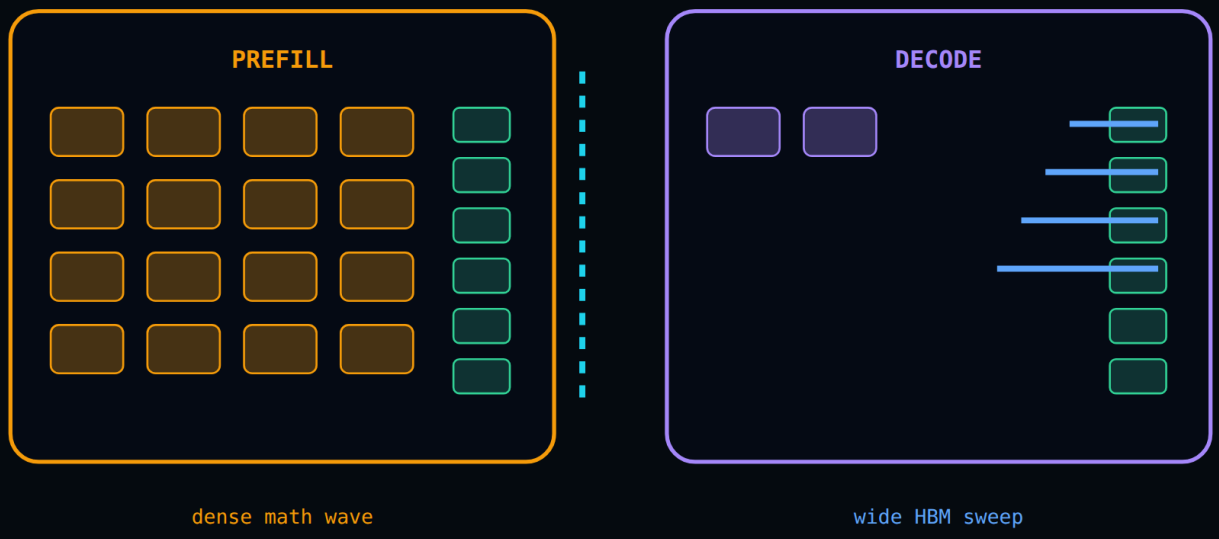
Autoregressive dependency prevents ordinary decode from accepting all future tokens in parallel. This consequence follows the same illustrative request, so the resource tradeoff stays concrete across stages.

The selected green token now lands on the output tape and curves back into the next input position. Its KV slot appears beside the preserved prompt pages.

Decode is a sequential acceptance loop wrapped around highly optimized kernels. Next, we compare why the same GPU behaves differently during parallel prefill and narrow decode.

Contrast the hardware stress

★ If you remember one thing · Prefill packs dense compute while narrow decode repeatedly streams weights and KV state through HBM.



We just saw two phases on the same model. NVIDIA defines performance as limited by math bandwidth, memory bandwidth, or latency, depending on work per byte moved.

Prefill forms large matrix operations from many known positions. More arithmetic reuses loaded weights, so the compute tiles can fill across a dense wave of work.

Prefill can pack compute while decode repeatedly streams weights and cache through memory. This consequence follows the same illustrative request, so the resource tradeoff stays concrete across stages.

Decode often has a much narrower batch and must revisit the same weights. Which resource becomes the stronger limiter?

PAUSE AND PREDICT

Which resource often limits narrow decode?

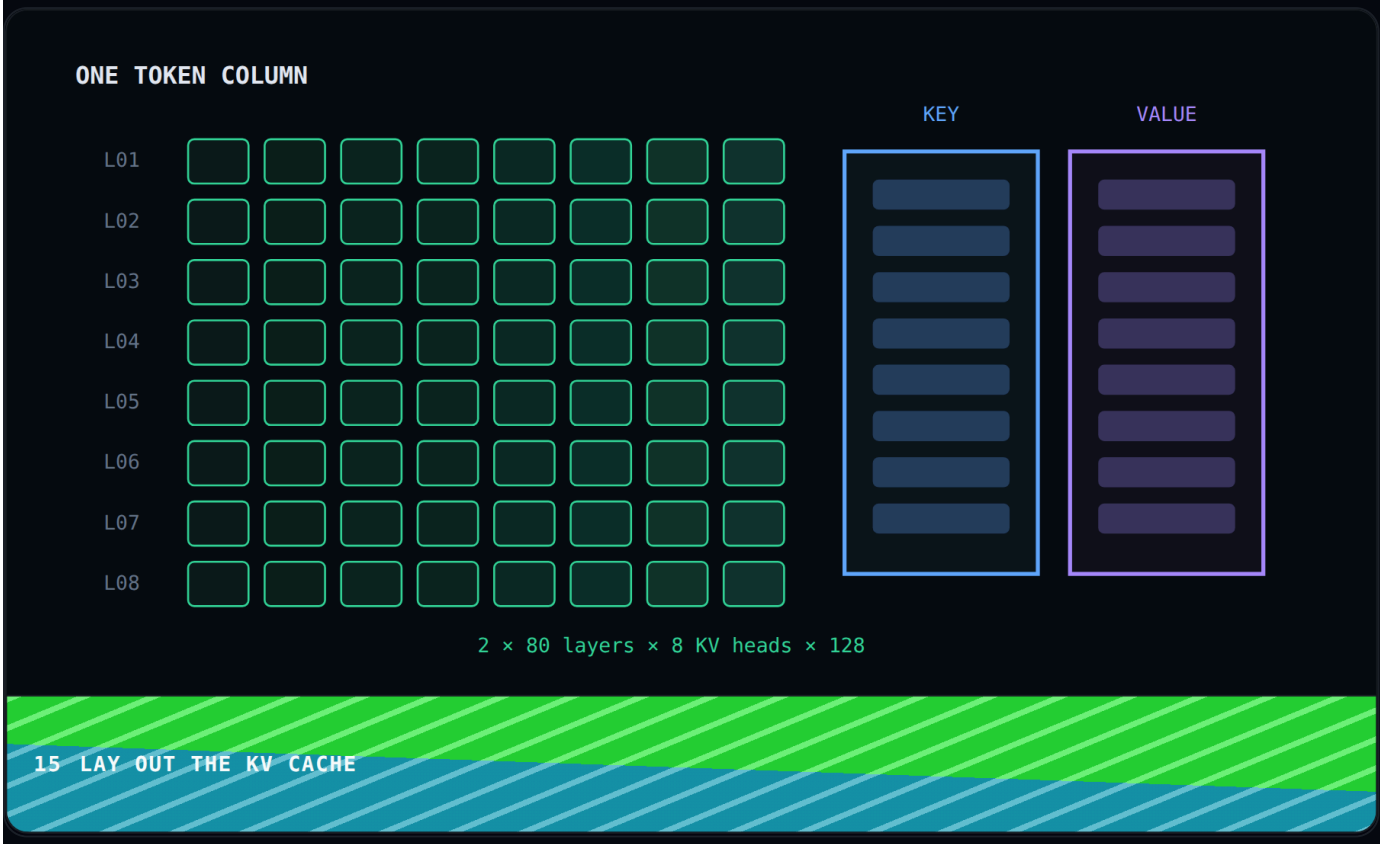
- Peak tensor math
- HBM bandwidth
- Model storage

```
Skip this check
```

The equal GPU silhouettes now diverge. Prefill fills many compute tiles, while decode shows a broad HBM sweep feeding only a narrow active column.

The picture explains why one batch policy cannot optimize every phase. Next, we open the green KV pages and derive the memory that decode must preserve.

Lay out the KV cache



The hardware contrast points directly to KV state. For each token, every transformer layer stores keys and values used by later attention calculations.

Grouped-query attention can use fewer KV heads than query heads. Our illustrative model has 80 layers, eight KV heads, and 128 values per head.

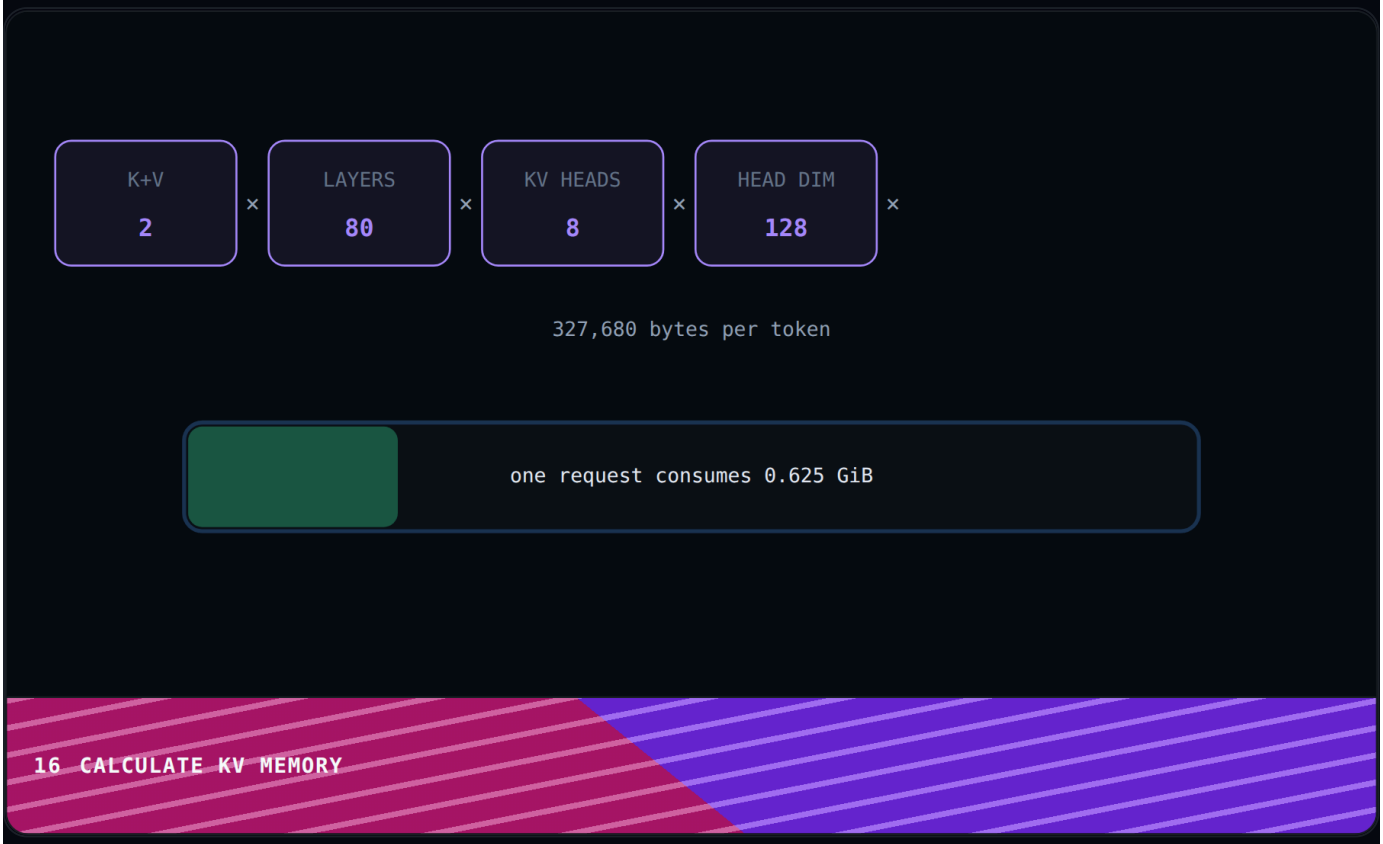
Two bytes store each illustrative KV element. Keys and values remain paired, while the token axis grows one column every decode iteration.

Every token adds paired state across every layer and KV head. This consequence follows the same illustrative request, so the resource tradeoff stays concrete across stages.

One green token column now expands through the layer stack and eight head lanes. A matching value page doubles the state carried for that position.

KV layout exposes every multiplicative factor instead of hiding memory in one cache box. That clarity makes capacity review much safer. Next, we multiply those factors and watch context length consume HBM.

Calculate KV memory



16 CALCULATE KV MEMORY

The layout becomes one formula: two for key and value, then layers, KV heads, head dimension, bytes per element, and total tokens. Context length scales KV memory linearly and turns one long request into a concurrency decision. This consequence follows the same illustrative request, so the resource tradeoff stays concrete across stages.

For this illustrative model, every token adds 327,680 bytes. The value comes from the published formula shape and our explicitly declared model fixture.

The request grows from 2048 to 8192 tokens while every model factor stays fixed. What happens to its KV footprint?

PAUSE AND PREDICT

What happens when context grows from 2K to 8K tokens?

- **Memory stays constant**
- **Memory doubles**
- **Memory quadruples**

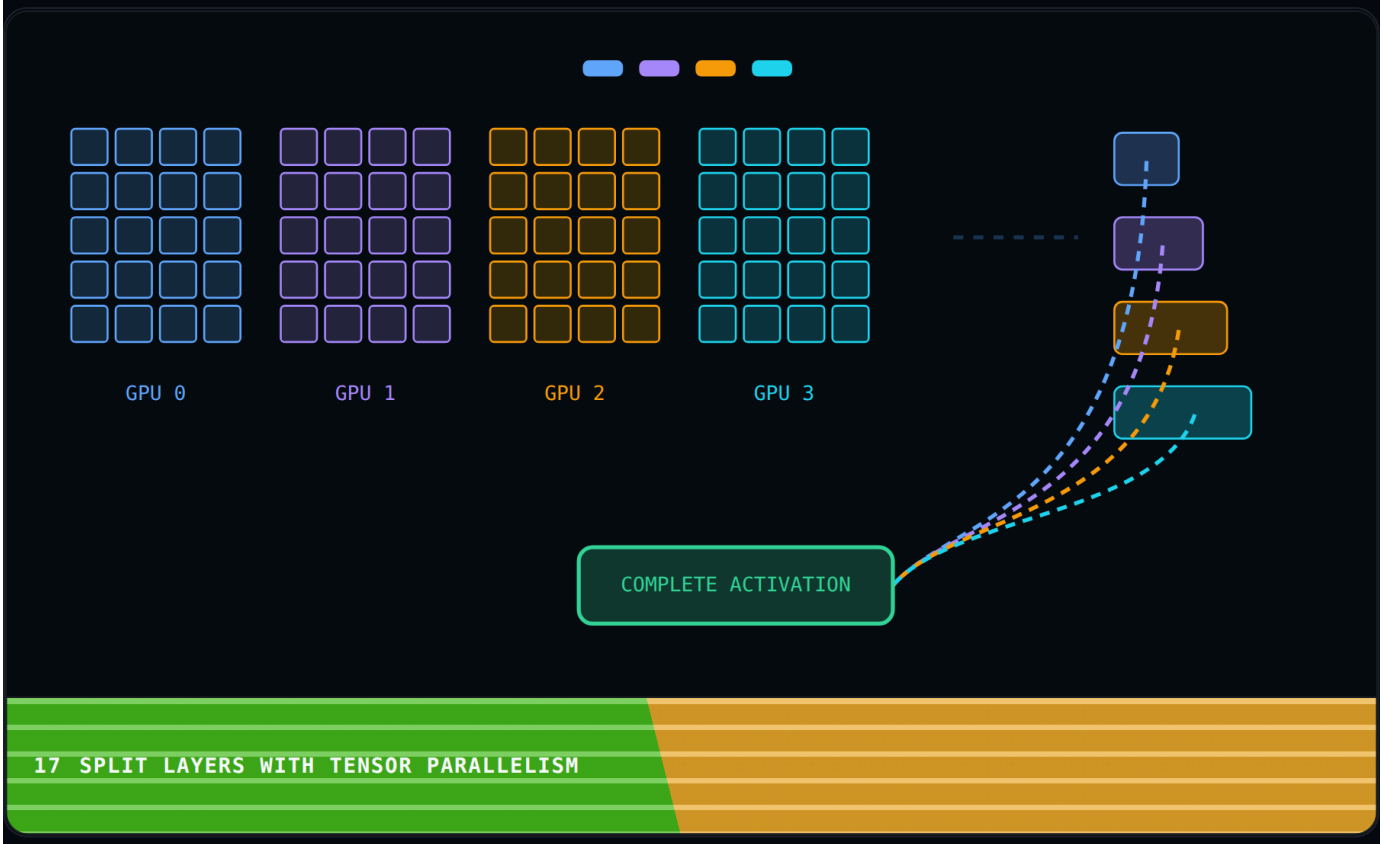
Skip this check

The multiplied strip now fills 0.625 GiB at 2048 tokens and 2.5 GiB at 8192 tokens. The HBM gauge grows by the same four-times factor.

Switch the Context tokens control through both limits. Compare the downstream per-request HBM reservation and remember that many admitted sequences grow at once.

KV memory turns a token limit into a capacity obligation. Next, we split model execution across GPUs and see where communication enters the latency budget.

Split layers with tensor parallelism



Our weight ribbon may need several GPUs or more cache headroom. Tensor parallelism divides one layer matrix into shards owned by different devices.

Four GPUs compute partial output vectors at the same time. The matrix geometry stays intact because each colored column range owns a distinct slice.

Later operations need a complete activation, so workers exchange or reduce partial results. Megatron-LM documents this composition and its communication tradeoffs.

Tensor parallelism reduces per-device weight memory but adds communication inside every layer. This consequence follows the same illustrative request, so the resource tradeoff stays concrete across stages.

The four partial vectors now converge through one all-reduce into a complete green vector. Every layer gains memory capacity, but every layer also pays this collective.

Tensor parallelism trades frequent communication for smaller per-device shards and shared compute. The collective remains part of every latency estimate. Next, pipeline parallelism makes a different trade by assigning whole layer ranges.

Split depth with pipeline parallelism

18 SPLIT DEPTH WITH PIPELINE PARALLELISM

Tensor parallelism split work inside every layer. Pipeline parallelism instead assigns consecutive layer groups to different GPUs, so activations move between depth segments.

The first microbatch enters stage zero before later stages have work. Empty cells form a pipeline bubble, which is idle capacity created by dependency.

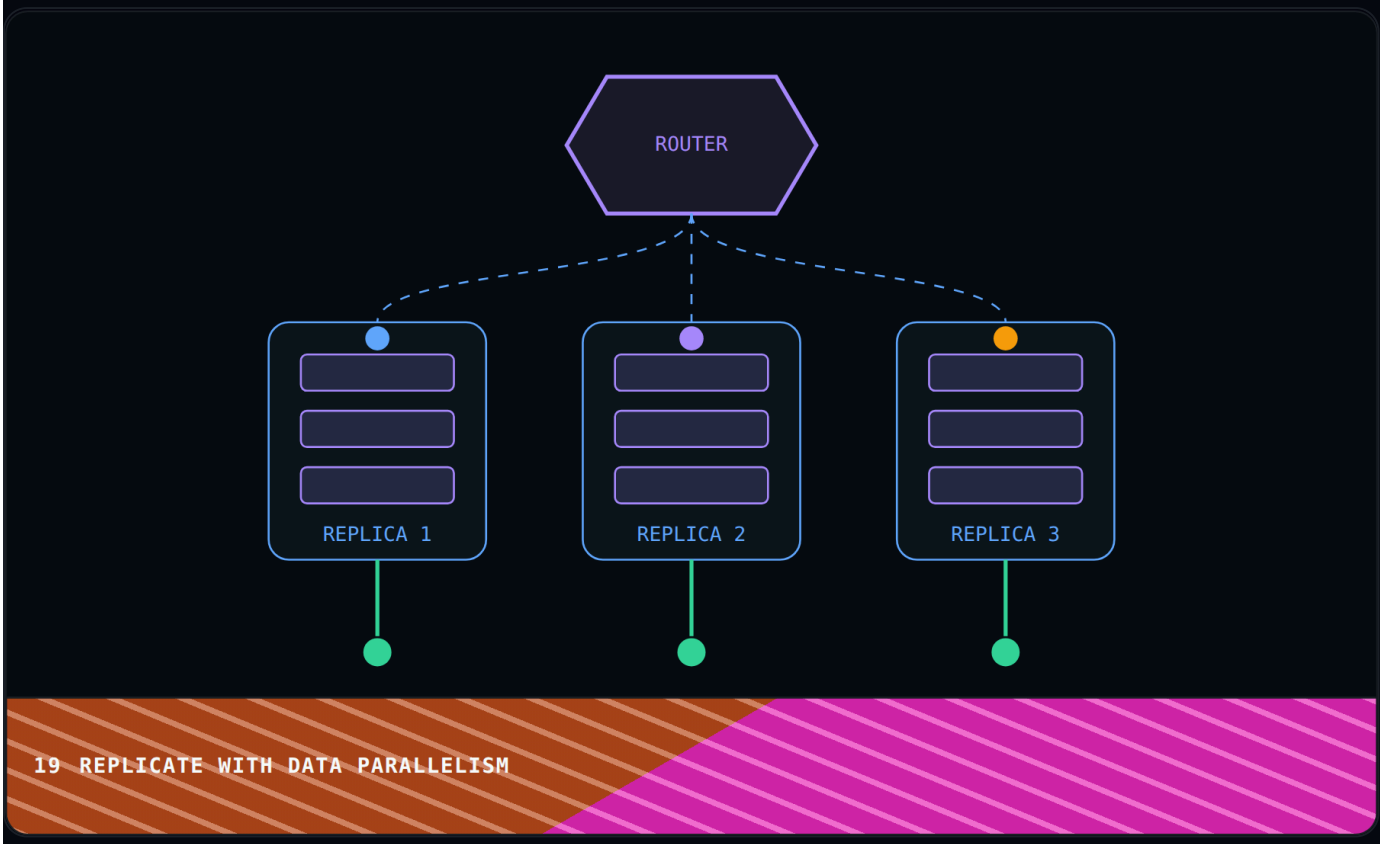
Additional microbatches fill the diagonal. Different requests or chunks can occupy different depth segments at the same time once the pipeline reaches steady flow.

Pipeline parallelism reduces shard communication frequency but creates fill and drain bubbles. This consequence follows the same illustrative request, so the resource tradeoff stays concrete across stages.

The space-time grid now fills across four devices and three microbatches. Only the leading and trailing corners remain empty, so useful overlap replaces most of the bubble.

Pipeline parallelism trades bubbles and activation transfers for whole-layer ownership. Next, data parallelism copies the complete model so independent requests avoid model collectives.

Replicate with data parallelism



Pipeline parallelism still describes one model instance. Data parallelism creates several complete instances, each with its own weights, scheduler, and KV memory pool.

The router sends independent requests to different replicas. No cross-replica collective is needed for ordinary inference because each copy can produce a complete result.

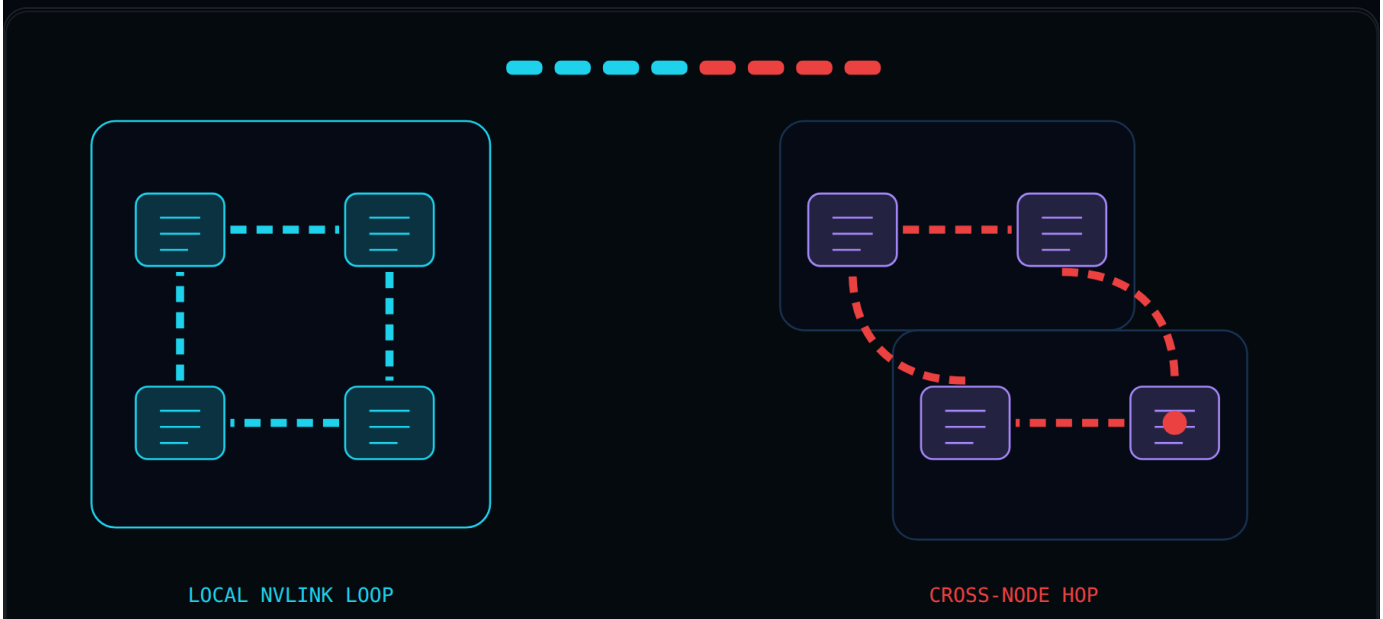
Replicas increase throughput and failure isolation, but they duplicate weight memory. A model version rollout must also keep every active copy compatible with its request contract.

Data parallel replicas scale request throughput without sharing per-request KV state. This consequence follows the same illustrative request, so the resource tradeoff stays concrete across stages.

Three request lanes now fan into three complete stacks and return separate streams. Each stack owns its cache, which makes replica choice important for prefix locality.

Data parallelism multiplies capacity by copying full execution groups. Next, we place the GPUs inside each group so their collectives stay on the fastest available links.

Place collectives on fast links



20 PLACE COLLECTIVES ON FAST LINKS

Our parallel strategies now meet physical topology. Tensor collectives repeat frequently, so their participants should share high-bandwidth links whenever the group fits inside one node.

The local placement keeps four GPUs behind the same NVSwitch fabric. Short edges form one contained collective loop without consuming cross-node bandwidth.

A scattered placement puts the same logical group across two nodes. It remains correct, but every collective now crosses the slower fabric and competes with KV transfers.

A mathematically valid placement can still be slow when collectives cross node boundaries. This consequence follows the same illustrative request, so the resource tradeoff stays concrete across stages.

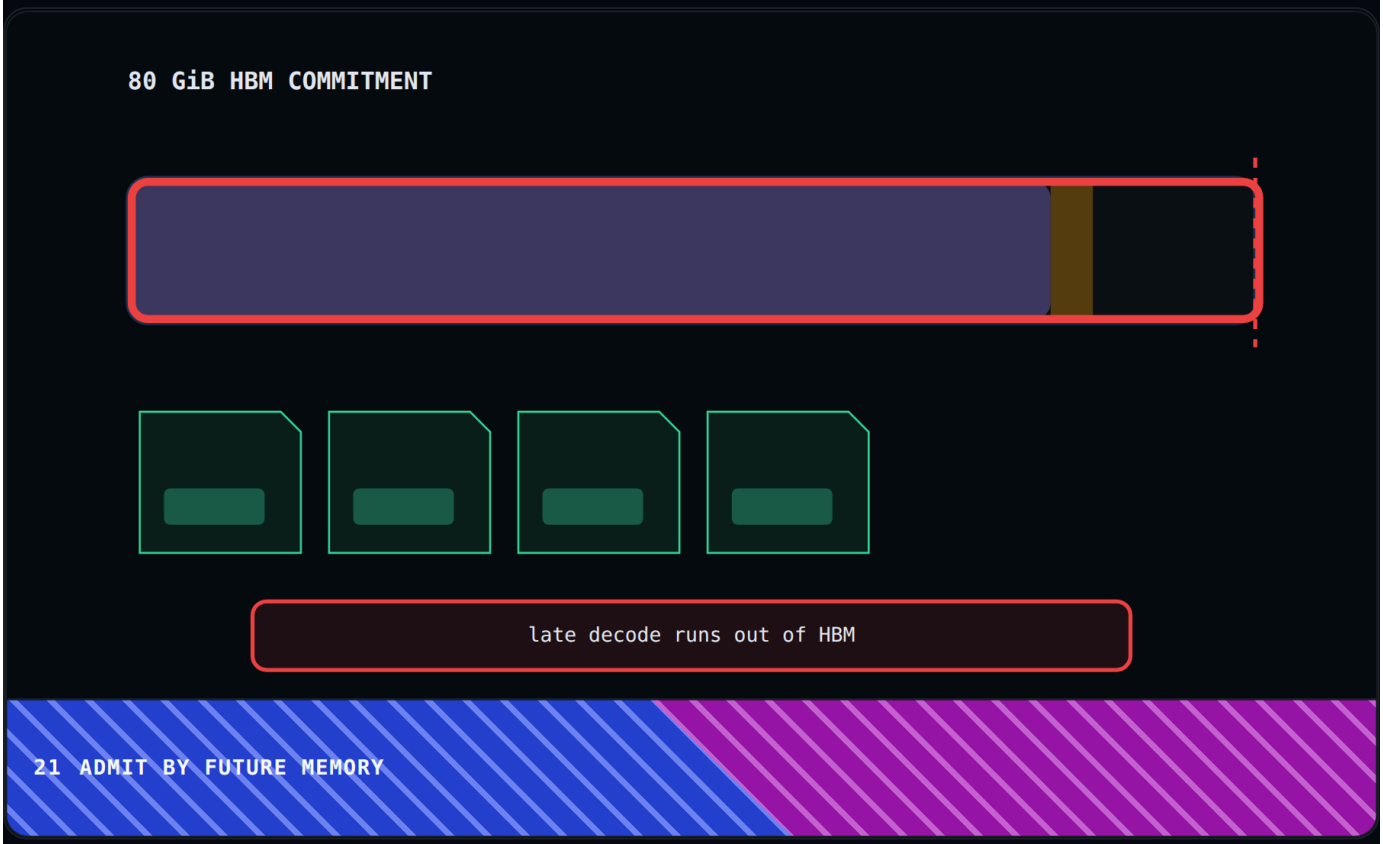
Equal all-reduce loops now reveal different physical distances. The local loop stays inside one outline, while the scattered loop stretches across a red cross-node hop.

Placement must understand both model parallelism and the network graph. Next, the scheduler uses those placed replicas while promising enough memory for every admitted request.

STAGE 21 OF 30

Admit by future memory

Try it in Watch mode. Keep every admitted request inside the 80 GiB HBM limit.



Topology gives us valid replicas, but HBM is still finite. FP8 weights and runtime buffers leave a measured region for all active KV pages. Admission must budget each request at its allowed future length, not only its current prompt. This consequence follows the same illustrative request, so the resource tradeoff stays concrete across stages.

An optimistic check counts only current prompt pages. Several requests appear to fit, even though each can grow toward its declared 8192-token limit during decode.

The fifth request enters before earlier sequences finish growing. Which admission rule prevents a late memory failure?

PAUSE AND PREDICT

Which rule prevents late HBM exhaustion?

- **Count only current pages**
- **Reserve future KV growth**

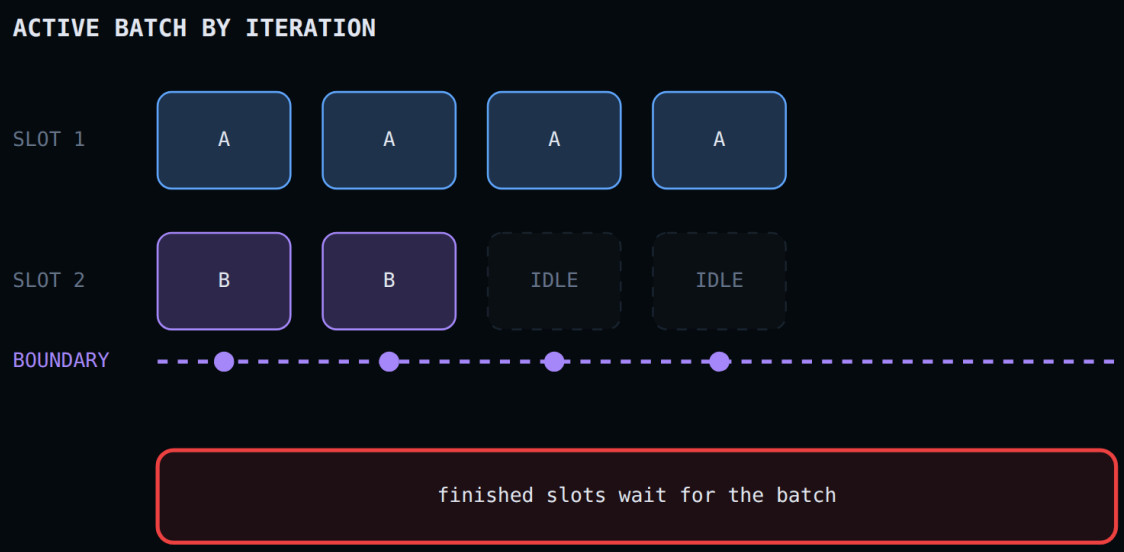
Skip this check

Future footprints now expand inside the gauge before admission. Optimistic mode crosses the 80 GiB edge later, while reserved mode keeps every accepted request below it.

Switch the Admission mode through both policies. Compare the downstream HBM verdict, then keep Future reserve selected to solve the capacity challenge.

Admission converts sequence limits into memory promises and queue decisions. Next, continuous batching decides which admitted requests share each decode iteration.

Schedule every iteration



22 SCHEDULE EVERY ITERATION

Admission produced a safe queue with varied output lengths. A static batch fixes membership until the longest original sequence finishes, which leaves holes behind shorter requests. Continuous batching turns early completions into immediate capacity instead of idle holes. This consequence follows the same illustrative request, so the resource tradeoff stays concrete across stages.

Orca schedules at iteration granularity. After each token step, the scheduler can remove finished sequences and select waiting work for the next batch.

Request B finishes while A still decodes and C waits. What should occupy B’s slot on the next iteration?

PAUSE AND PREDICT

What should occupy the finished slot?

- **Nothing until A finishes**
- **The next eligible request**

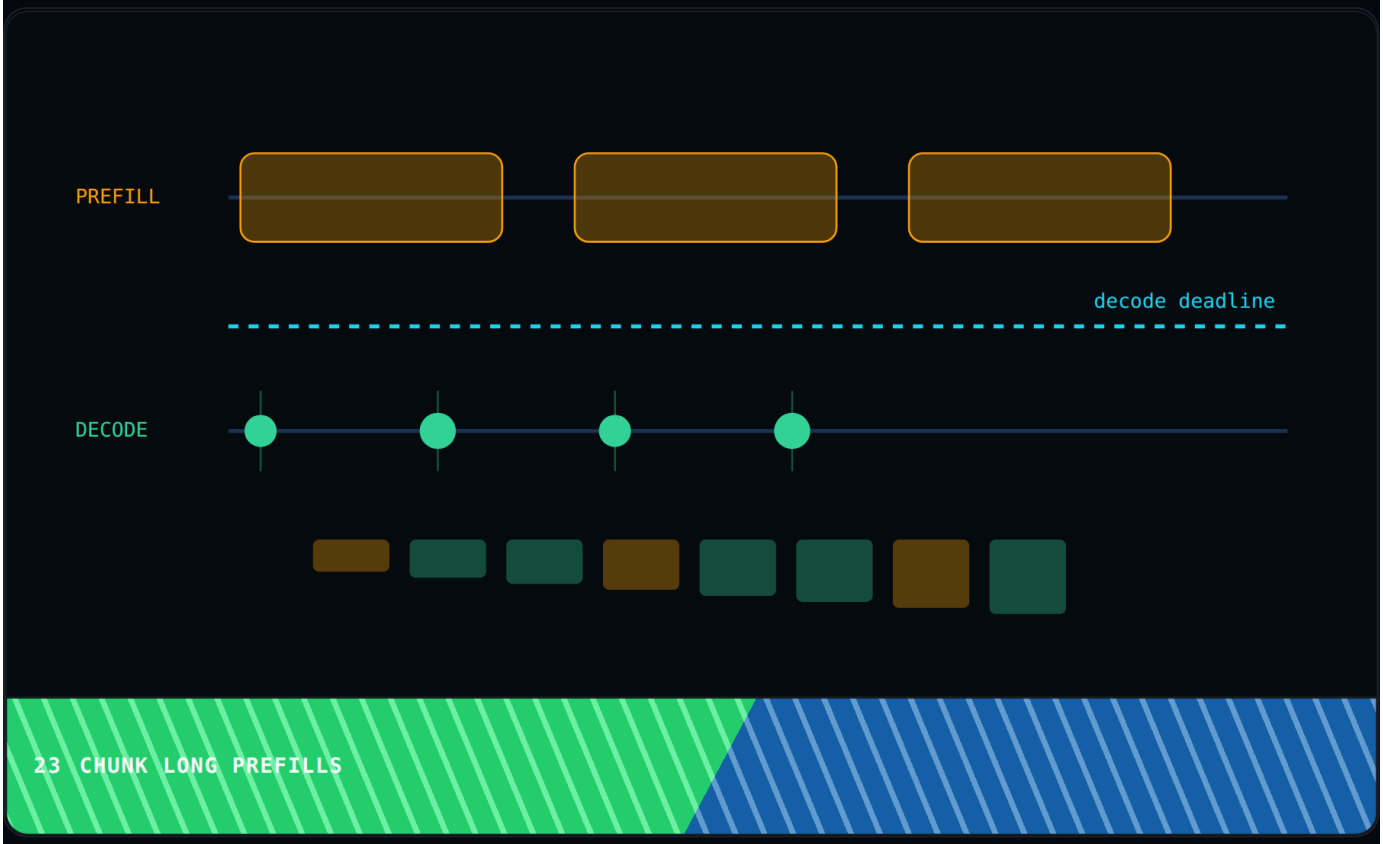
Skip this check

The equal time grids now diverge. Static membership leaves empty cells, while continuous membership moves C and D into freed slots at later iteration boundaries.

Switch the Batch policy through both choices. Compare the downstream occupancy pattern and finish after you expose both idle holes and newly admitted work.

Continuous batching improves useful occupancy, but one giant prefill can still delay every decoder. Next, we divide long prefills into bounded chunks that share time fairly.

Chunk long prefills



Continuous batching reuses slots, but phases still interfere. A 4096-token prefill can occupy a long compute interval while active decoders wait for their next token.

Chunked prefill splits that prompt into bounded segments. Each segment creates useful dense work without claiming the entire scheduling horizon.

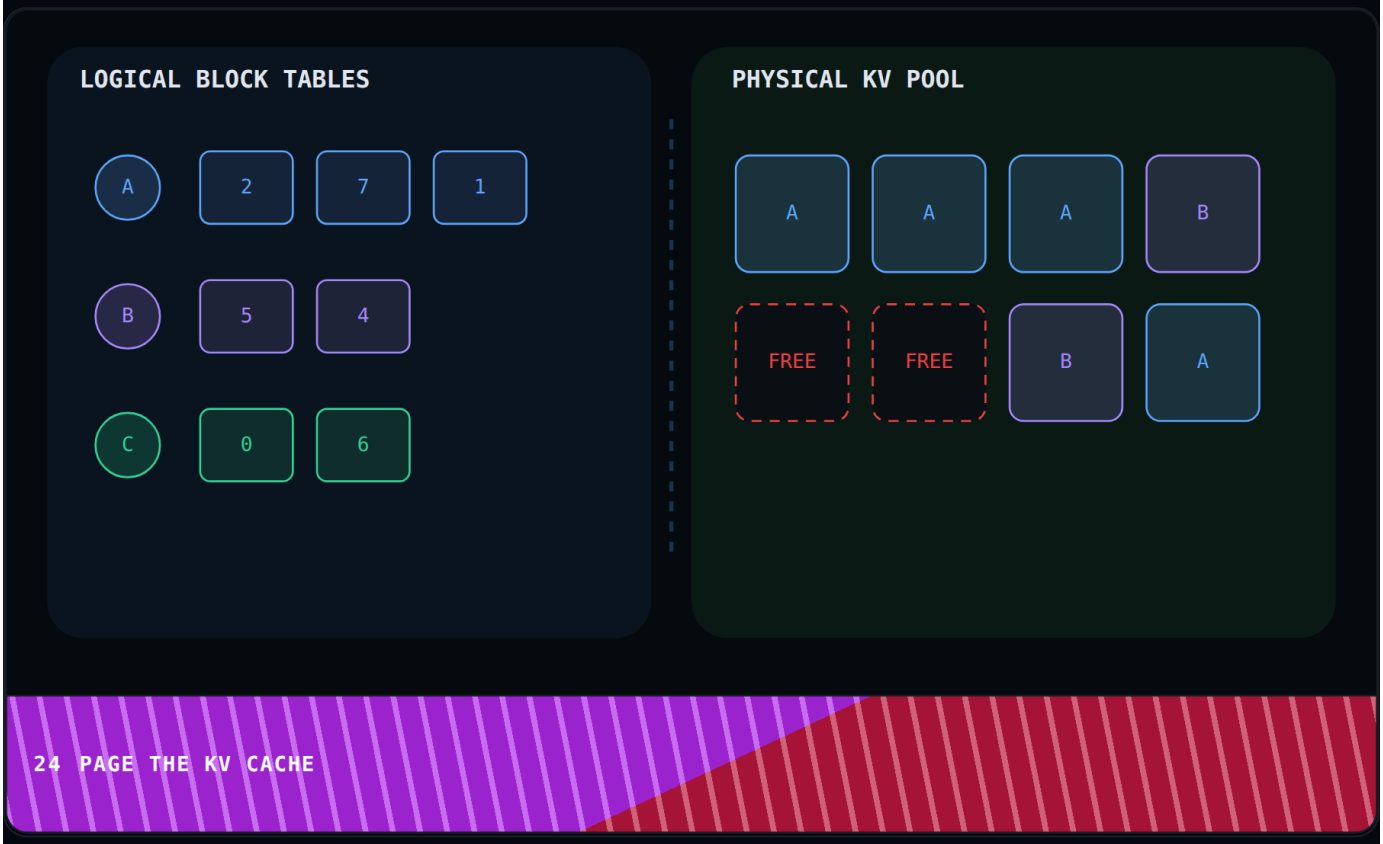
Decode ticks fit between chunks, so time per output token stays near its deadline. Sarathi-Serve studies this stall-free batching tradeoff for LLM serving.

Chunking trades some scheduling overhead for shorter decode stalls and steadier token gaps. This consequence follows the same illustrative request, so the resource tradeoff stays concrete across stages.

One long amber bar now becomes three separated chunks with green decode ticks between them. The total prefill remains, but its longest blocking interval shrinks.

Chunking controls interference instead of pretending prefill and decode cost the same. Next, the KV allocator uses blocks so varied sequence lengths do not fragment HBM.

Page the KV cache



Chunked scheduling still leaves sequences with different lifetimes and lengths. Reserving one contiguous maximum-sized region for each request wastes space and creates unusable gaps.

PagedAttention divides KV state into fixed token blocks. Each request owns a logical block table whose entries point to physical slots allocated on demand.

The physical slots need not be adjacent. Attention follows the table, so request A can own slots two, seven, and one without moving its existing contents.

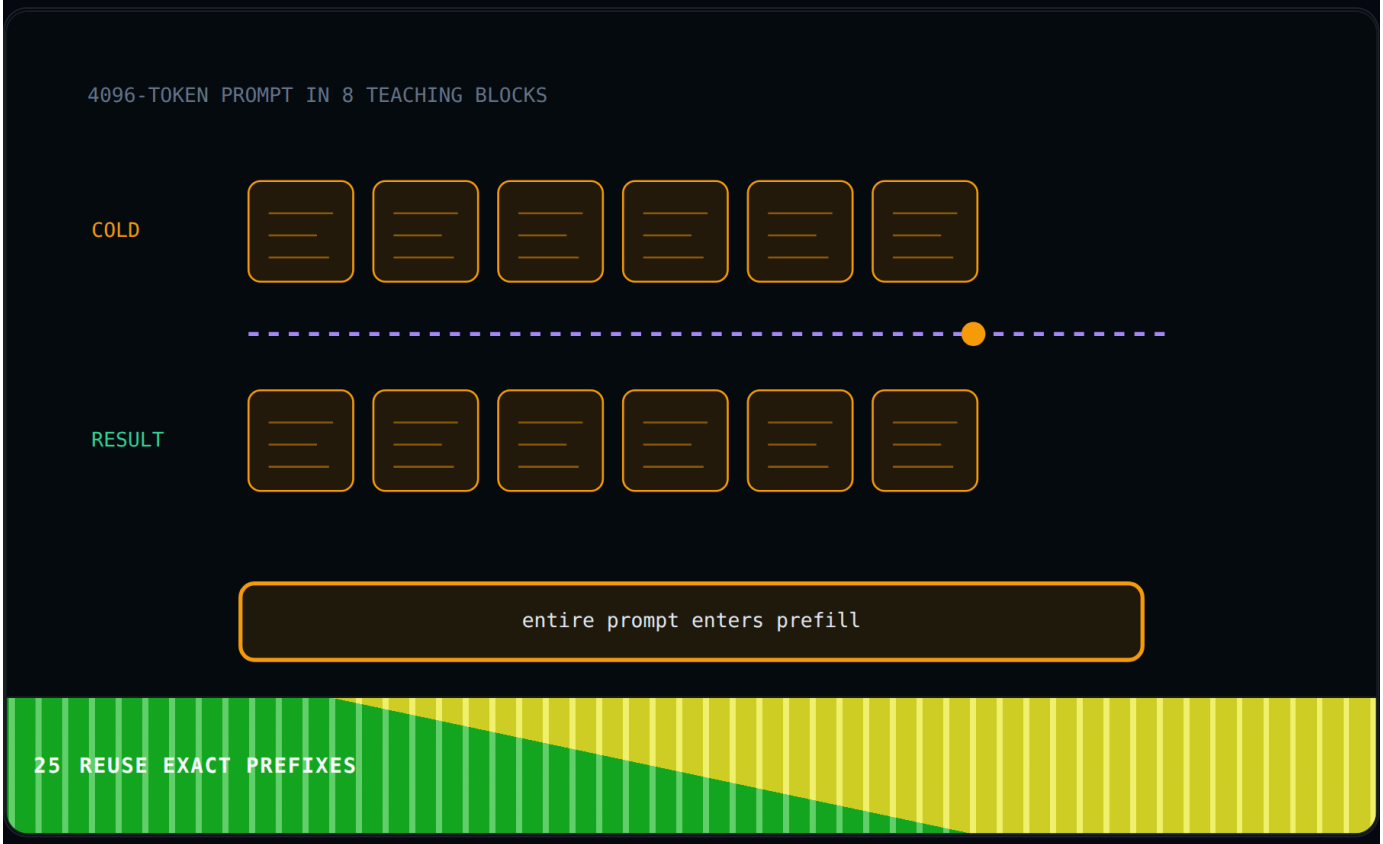
Paging removes the requirement for one large contiguous reservation per sequence. This consequence follows the same illustrative request, so the resource tradeoff stays concrete across stages.

Request B now finishes and releases two scattered slots. Request C immediately maps those slots into its own table without waiting for a large contiguous hole.

Paged KV allocation turns fragmented HBM into reusable blocks and enables flexible sharing. Next, exact matching prefixes reuse those blocks across requests before prefill begins.

STAGE 25 OF 30

Reuse exact prefixes



Paged blocks give the server a reusable unit. A long system prompt or shared document prefix can appear at the start of many requests. A prefix hit saves prefill only for the exact leading token blocks that match. This consequence follows the same illustrative request, so the resource tradeoff stays concrete across stages.

The cache hashes complete token blocks and their model context. Reuse is valid only when the leading tokens and relevant model settings match exactly.

A later request shares the first 3072 tokens but has a new suffix. Which region can skip prefill?

PAUSE AND PREDICT

Which region can skip prefill?

- The entire prompt
- Only the exact shared prefix
- Only the new suffix

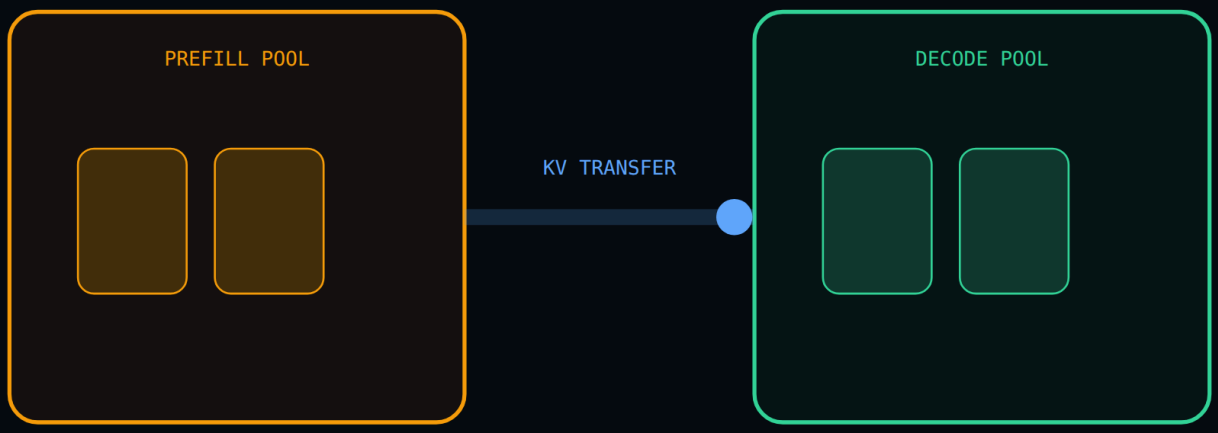
Skip this check

The hit lane now preserves three quarters of the token tape as green cached blocks. Only the new suffix enters amber prefill, while the cold lane computes everything.

Switch the Prefix result through both outcomes. Compare the downstream prompt work and notice that the unmatched suffix always remains real computation.

Prefix caching converts repeated leading context into saved compute and faster first tokens. Next, we consider separating prefill and decode onto different GPU pools altogether.

Disaggregate prefill and decode



26 DISAGGREGATE PREFILL AND DECODE

Prefix reuse reduces some prompt work, but misses still create dense prefills beside narrow decodes. A shared pool couples both latency objectives to one resource plan.

DistServe separates the phases. Prefill GPUs can favor compute-efficient parallelism, while decode GPUs can favor memory bandwidth and larger active batches.

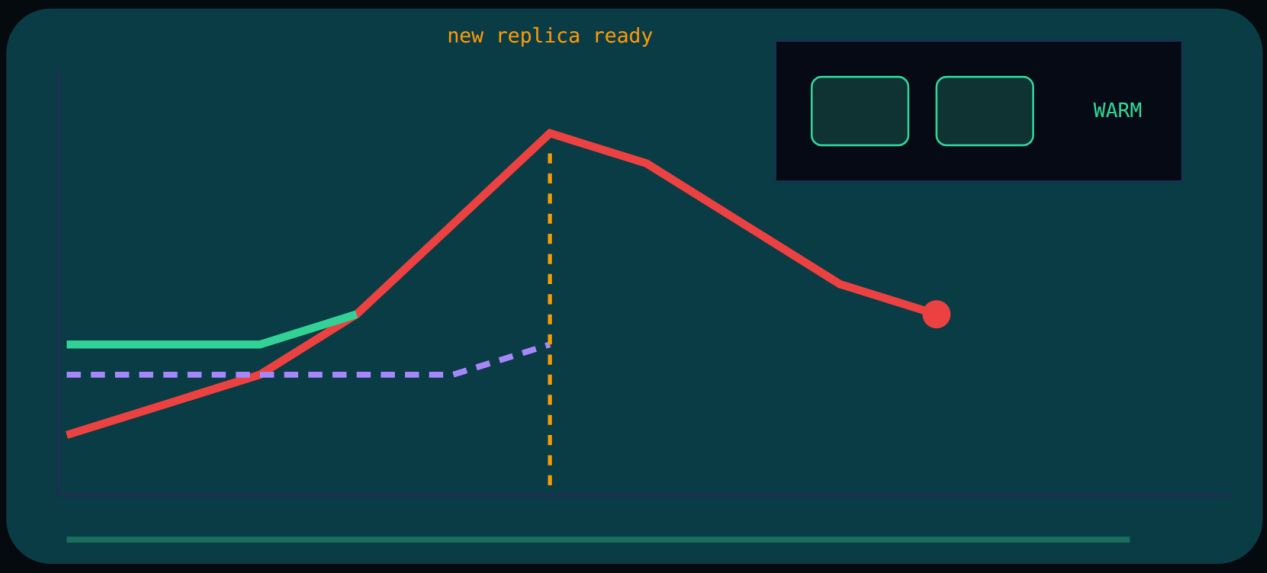
The boundary is not free because the completed KV state must reach the decode group. Placement must keep that transfer below the latency saved by removing interference.

Separate pools remove phase interference but add KV transfer and placement constraints. This consequence follows the same illustrative request, so the resource tradeoff stays concrete across stages.

The mixed queue now splits into amber prompt work and green token loops. A blue KV bridge connects the pools, while each queue tracks its own latency target.

Disaggregation is valuable when interference costs more than KV movement and operational complexity. Next, fleet scaling must account for the minutes or seconds required to make new replicas warm.

Scale warm capacity



27 SCALE WARM CAPACITY

Separated pools need independent capacity. Kubernetes can scale replicas from observed metrics, but a GPU process must still load weights and warm kernels before it serves.

Reactive scaling that waits for high GPU utilization starts too late. Queue age and admitted token load rise first, while the new replica remains unavailable during warm-up.

A warm pool keeps a small prepared reserve and combines it with burst forecasts. The reserve absorbs immediate traffic while background scaling restores headroom.

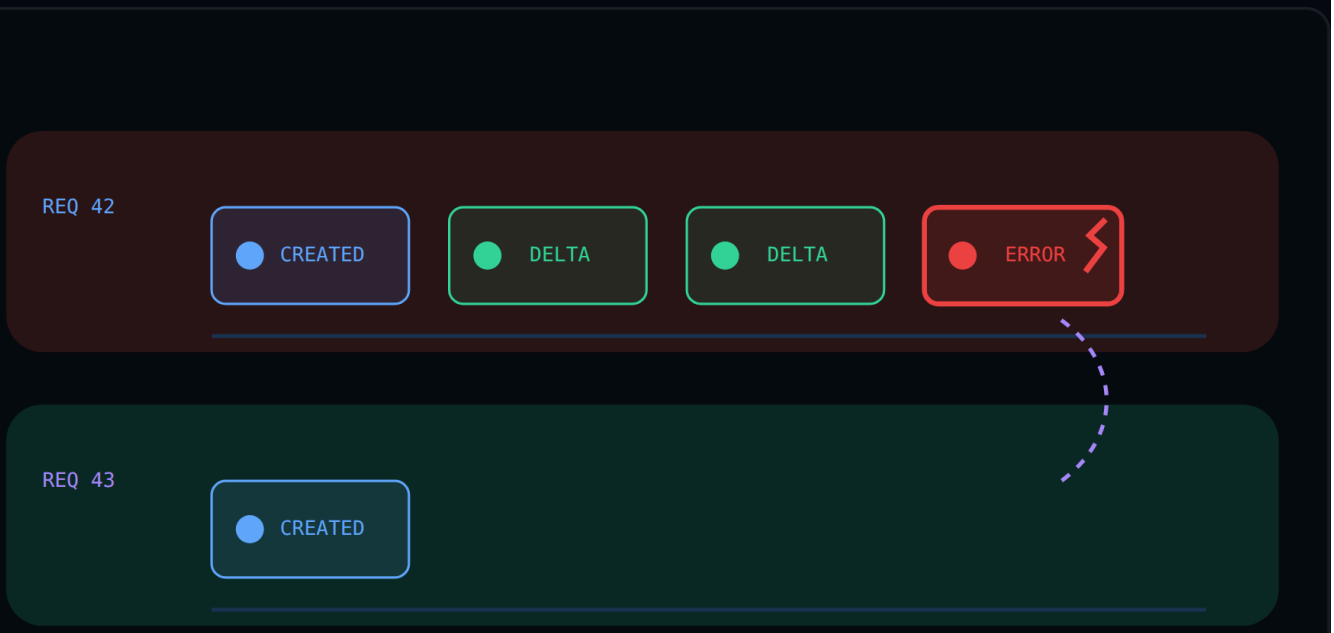
GPU utilization alone reacts after queues grow and cannot erase model warm-up delay. This consequence follows the same illustrative request, so the resource tradeoff stays concrete across stages.

The demand peak now meets two capacity curves. Reactive readiness arrives after the queue spike, while the warm reserve bridges the gap until new replicas open.

Autoscaling must target future ready capacity rather than current process count. Next, we contain replica and network failures without corrupting an already visible stream.

STAGE 28 OF 30

Contain streaming failures



28 CONTAIN STREAMING FAILURES

Warm capacity reduces overload, but replicas and links still fail. Our response has already emitted two ordered deltas when its decode worker disappears.

Other replicas continue because failure domains are small and routing health updates quickly. The broken request is different because its partial output is already client-visible state.

A failed stream cannot be invisibly replayed after the client has already received tokens. This consequence follows the same illustrative request, so the resource tradeoff stays concrete across stages.

Silently replaying the request could repeat or diverge from delivered tokens. What should the stream do instead?

PAUSE AND PREDICT

What should happen after a partial stream fails?

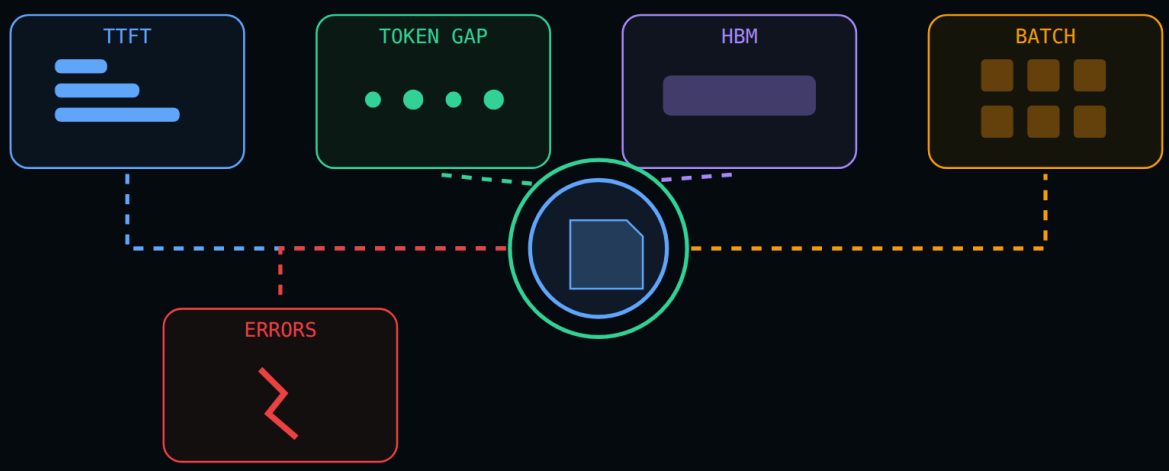
- Replay invisibly
- Close with an error and retry explicitly
- Append another replica without a boundary

Skip this check

The first event tape now ends with a red terminal error after its two deltas. A second request begins on a new identity instead of pretending the old stream continued.

Failure handling preserves exactly what the client has observed and contains unhealthy replicas. Next, correlated telemetry closes the loop between latency, memory, utilization, and cost.

Close the operations loop



29 CLOSE THE OPERATIONS LOOP

Failure containment gives every request a clear identity and terminal state. OpenTelemetry describes emitting correlated traces, metrics, and logs, which lets operations follow that identity across services.

Request spans separate gate time, queue time, prefill, decode gaps, and streaming delivery. GPU metrics add HBM pressure, batch occupancy, collective delay, and warm replica state.

Cost needs a useful denominator. GPU hours divided by successful useful tokens exposes idle replicas, wasted retries, and policies that buy latency with excessive overprovisioning.

The platform improves only when one request identity connects product SLOs to GPU causes and fleet cost. This consequence follows the same illustrative request, so the resource tradeoff stays concrete across stages.

One trace now fans into aligned latency, memory, occupancy, error, and cost instruments. Red thresholds point back to admission and scaling instead of producing disconnected alarms.

Observability makes the design a feedback system rather than a frozen diagram. Now let us step back and connect every contract, hardware limit, scheduler, and operating control.

STAGE 30 OF 30

The serving platform map

We started with streaming sends ordered partial events under one response identity and ends exactly once. This callback connects the decision to the same request and its promised service objectives.

We then continued with time to first token and inter-token delay protect different parts of the user experience. This callback connects the decision to the same request and its promised service objectives.

We then continued with tail lengths and burst distributions drive capacity more safely than one average request. This callback connects the decision to the same request and its promised service objectives.

We then continued with a stable control snapshot keeps the token data path independent from slow fleet changes. This callback connects the decision to the same request and its promised service objectives.

We then continued with authentication, quotas, and safety reject unsuitable work before it consumes gpu capacity. This callback connects the decision to the same request and its promised service objectives.

We then continued with routing checks compatibility first, then balances load and useful cache affinity. This callback connects the decision to the same request and its promised service objectives.

We then continued with compute, hbm capacity, memory bandwidth, and interconnects constrain inference separately. This callback connects the decision to the same request and its promised service objectives.

We then continued with weight precision changes device count and the hbm left for runtime state. This callback connects the decision to the same request and its promised service objectives.

We then continued with a loaded model is not ready until kernels, communication, and representative shapes are warm. This callback connects the decision to the same request and its promised service objectives.

We then continued with the runtime receives an ordered token sequence plus explicit generation limits. This callback connects the decision to the same request and its promised service objectives.

We then continued with prefill processes known prompt positions in parallel and creates the initial kv state. This callback connects the decision to the same request and its promised service objectives.

We then continued with decode accepts one token per iteration because each choice becomes the next input. This callback connects the decision to the same request and its promised service objectives.

We then continued with prefill can fill compute while narrow decode often waits on memory movement. This callback connects the decision to the same request and its promised service objectives.

We then continued with each token stores paired key and value state across every layer and kv head. This callback connects the decision to the same request and its promised service objectives.

We then continued with kv memory grows linearly with tokens and turns context limits into hbm obligations. This callback connects the decision to the same request and its promised service objectives.

We then continued with tensor parallelism shards layers but introduces collectives inside model execution. This callback connects the decision to the same request and its promised service objectives.

We then continued with pipeline parallelism assigns layer ranges and trades communication for fill and drain bubbles. This callback connects the decision to the same request and its promised service objectives.

We then continued with data parallel replicas scale independent request throughput while duplicating weights and caches. This callback connects the decision to the same request and its promised service objectives.

We then continued with topology-aware placement keeps frequent collectives on the fastest local links. This callback connects the decision to the same request and its promised service objectives.

We then continued with safe admission reserves future kv growth before a request enters the active set. This callback connects the decision to the same request and its promised service objectives.

We then continued with iteration-level scheduling replaces finished sequences instead of leaving batch holes. This callback connects the decision to the same request and its promised service objectives.

We then continued with chunked prefill bounds decoder stalls while preserving the total prompt work. This callback connects the decision to the same request and its promised service objectives.

We then continued with paged kv allocation reuses scattered blocks and avoids contiguous maximum reservations. This callback connects the decision to the same request and its promised service objectives.

We then continued with prefix caching reuses only exact leading blocks and computes every unmatched suffix. This callback connects the decision to the same request and its promised service objectives.

We then continued with separate prefill and decode pools trade lower interference for kv transfer and complexity. This callback connects the decision to the same request and its promised service objectives.

We then continued with autoscaling must predict ready capacity because model loading and warm-up take time. This callback connects the decision to the same request and its promised service objectives.

We then continued with a partial stream ends explicitly, while a retry starts under a new response identity. This callback connects the decision to the same request and its promised service objectives.

We then continued with correlated traces connect user slos to gpu pressure, failures, utilization, and cost. This callback connects the decision to the same request and its promised service objectives.

The complete platform now connects product contracts, GPU physics, scheduling, memory, resilience, and cost around one request. The design works when every layer protects the same measurable promises.