

Building Distributed Inference at Scale: Lessons, Challenges and Opportunities

Senthilkumar Gopal
Sr. Engineering Manager, NVIDIA Dynamo

Slides adapted from
Vikram Sharma Mailthody

Disclaimer



The views expressed in the content belong to me and my collaborators, not NVIDIA, its affiliates, or employees

Senthilkumar Gopal



Where we are going towards

Shift towards Agentic AI



World view of inference

- Model trends
- Usage trends
- Challenges in serving

Agentic AI

- Runs LLM in a controlled loop
- Interaction with CPU to run execution environment
 - Bash commands
 - Web search
 - Database calls

CPU runs

- Tool calls in execution environment
- Environment runs in a sandbox
- Control loop called agent harness
- Other orchestration tasks

The laws behind every claim in this talk

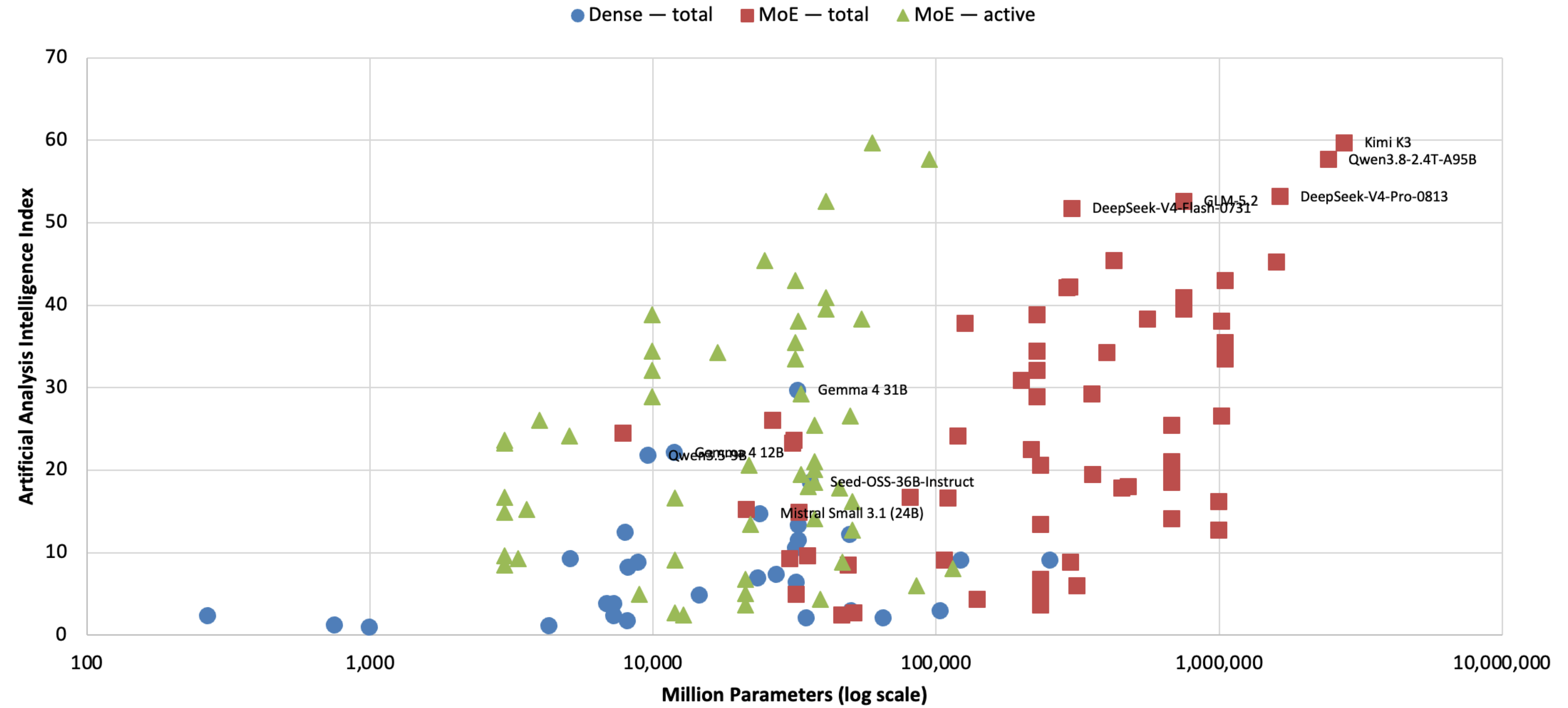
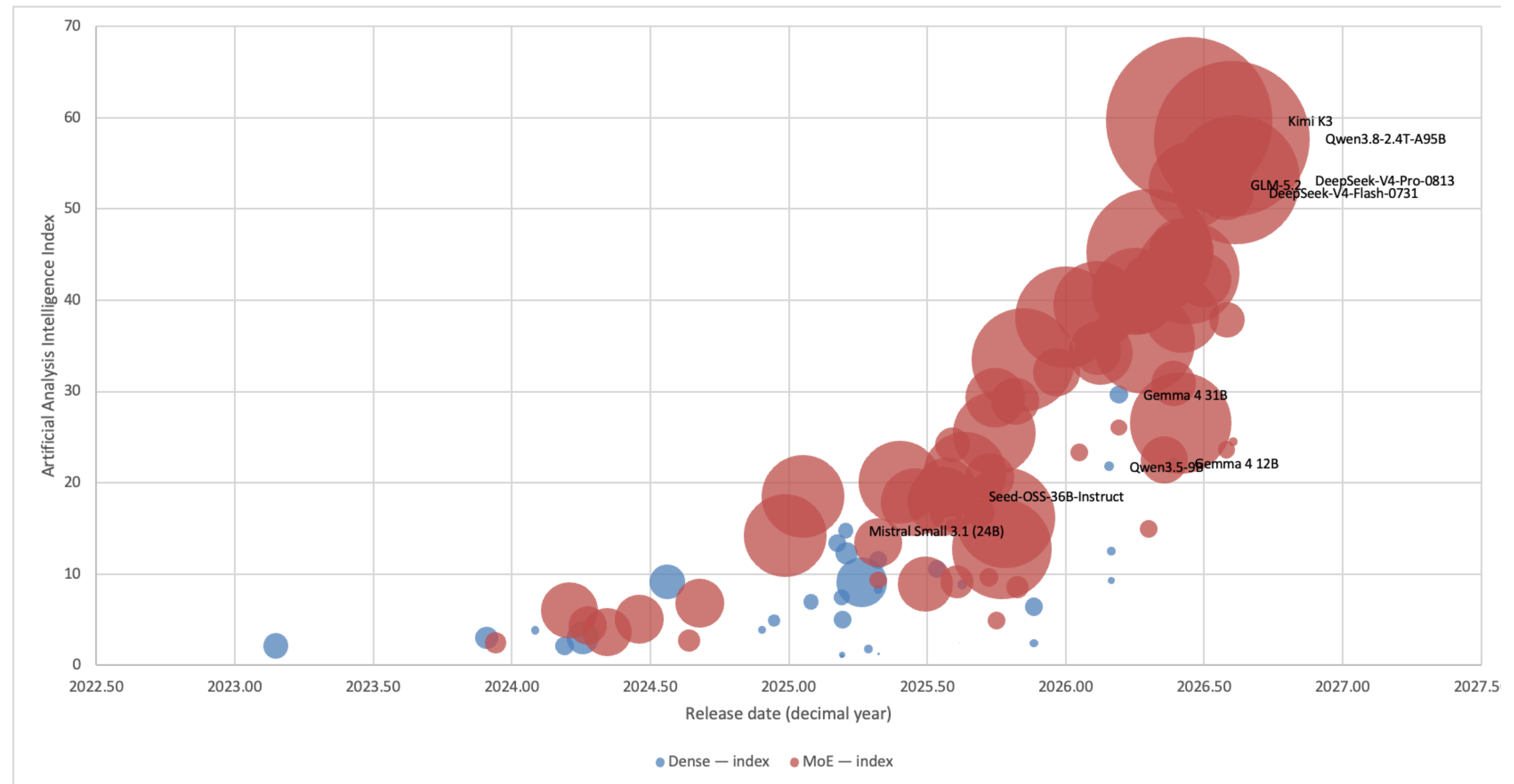
Inference at scale using NVIDIA Dynamo

- Aggregated and disaggregated serving
- Dynamo components
- Fault tolerance
- Unsolved problems and opportunities

Law	Statement	What it says about agents
Amdahl	$S = 1 / ((1 - p) + p/N)$	Speeding one phase has a ceiling set by all the others. The accelerator is p; the five host phases are the rest
Little	$L = \lambda \times W$	Agents in flight = arrival rate $\times$ time in system. Once W grows, admitting more λ buys queue, not work
Universal Scalability	$C(N) = N / (1 + \alpha(N-1) + \beta N(N-1))$	Contention bends the curve and coherency traffic turns it down. This is the shape of a density knee
Utilization and service demand	$U = X \times S$, and the bottleneck is the largest S	The phase with the largest demand per request sets capacity — the minimum in the capacity model
Roofline	$P \leq \min(\text{peak compute}, \text{intensity} \times \text{bandwidth})$	The roof you hit depends on the memory you actually touch. A cache-resident loop never reaches the DRAM roof
Gustafson	$S = N - (1 - p)(N - 1)$	Scale the work rather than the speed: more agents per node is a different question from faster agents

None of this is new. Rather applying it per phase or each part instead of the whole system is the key challenge

Unprecedented Model Growth



Artificial Analysis Intelligence Index combines performance across 10 evaluations:

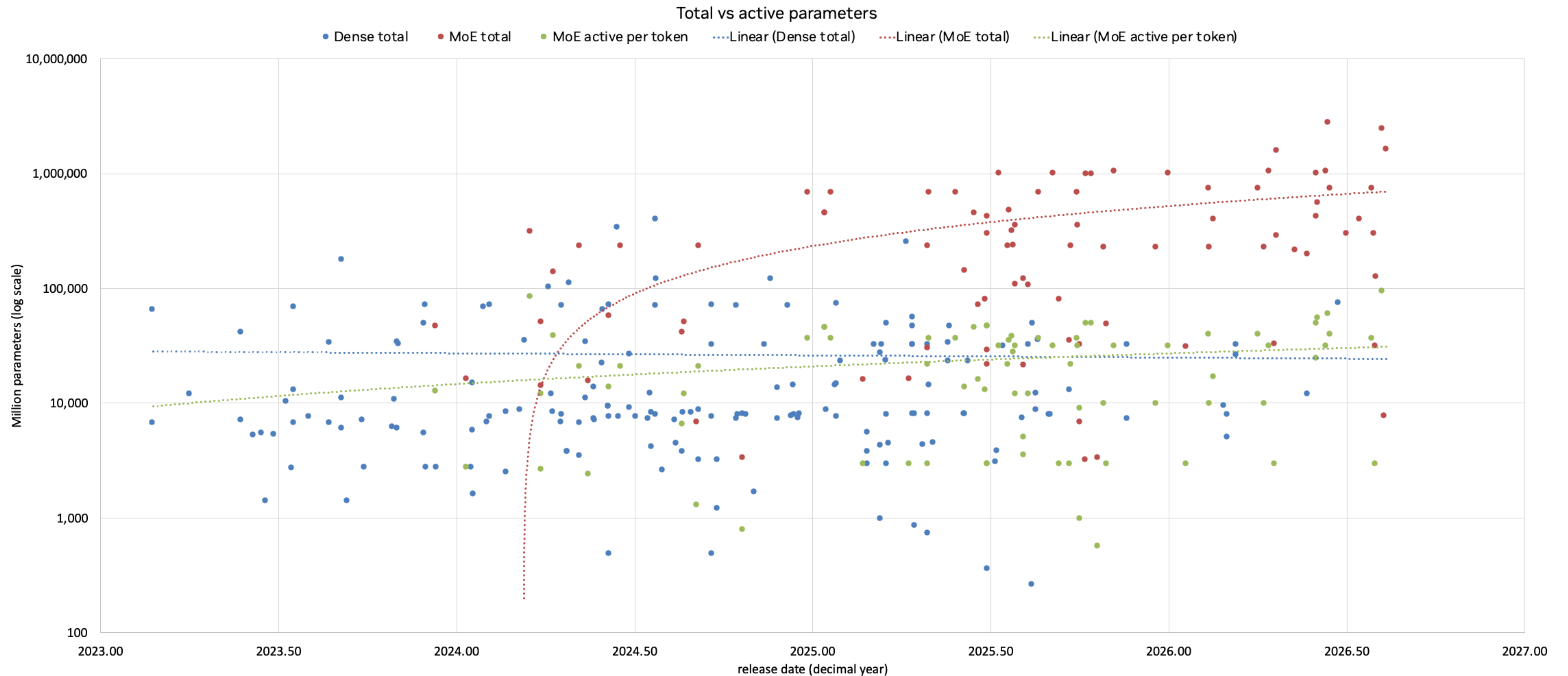
[AA-Briefcase v1.1](#), [GDPval-AA v2.1](#), [AutomationBench-AA](#), [Terminal-Bench 4.0](#), [SciCode](#), [Humanity's Last Exam](#), [GDP.pdf](#), [CritPt](#), [AA-Omniscience](#), and [AA-LCR v1.1](#).

This composite metric provides a single score for tracking progress toward artificial general intelligence across mathematics, science, coding, and reasoning.

All models from Huggingface.com
<https://artificialanalysis.ai/#intelligence>

Unprecedented Model Growth

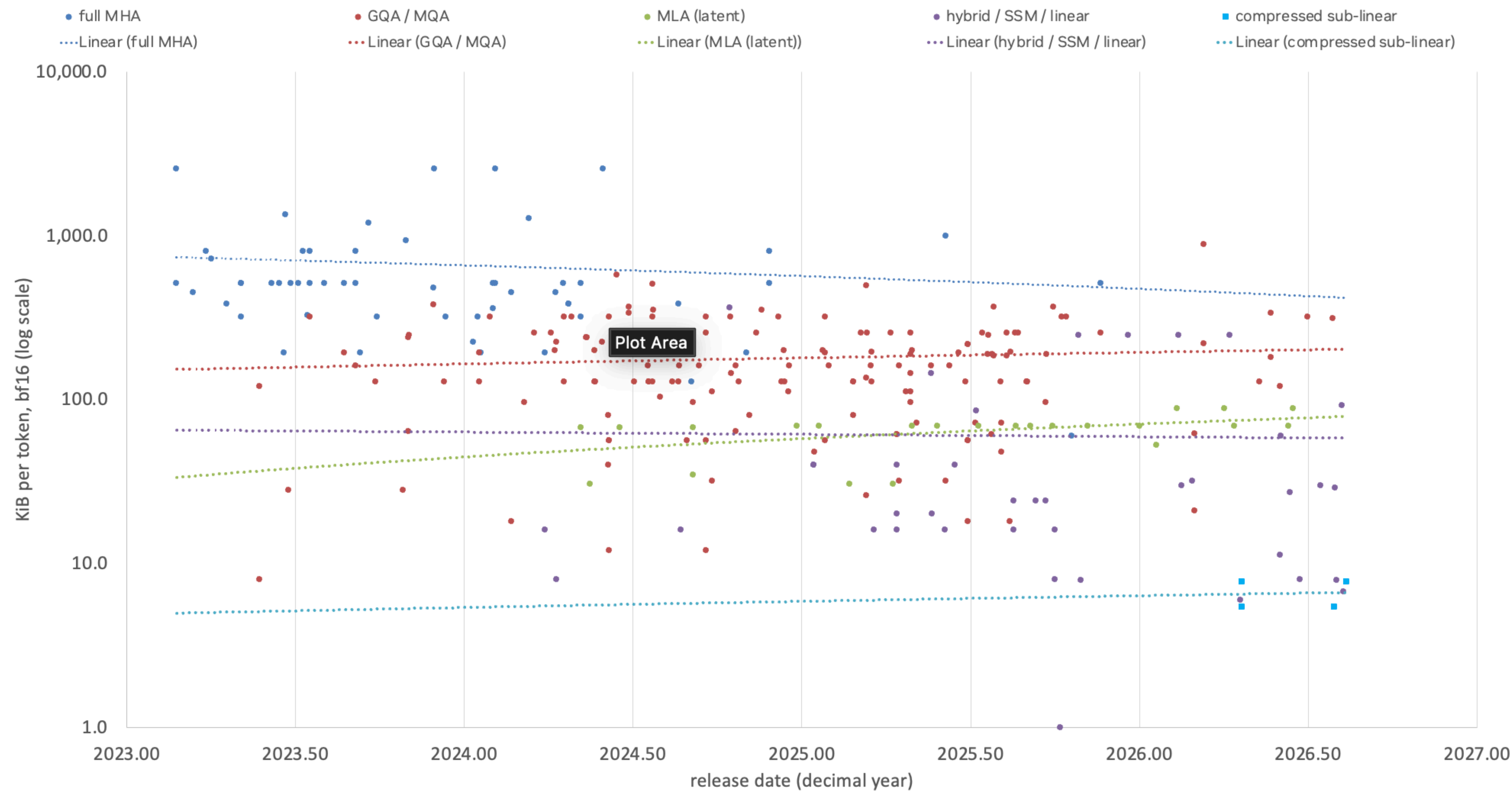
MoE total parameters: $\sim 2.2\times/\text{year}$ (frontier at $3.56\times/\text{year}$), doubling every ~ 10.6 months, Dense: growth has stopped
Sparsity: $(0.55\times)$ halving every ~ 12 months



In 2026 the median MoE release stores **68× more parameters than the median dense release while computing with only 3.3× more**. That ratio was 3.6:1 in 2023 and is 20.5:1 now — memory capacity and arithmetic throughput have become separate provisioning problems, which is the single most consequential trend in the dataset for anyone sizing inference hardware.

KV cache growth

0.50×/year, halving roughly every 12 months



KV cache KiB per token, by attention family

KV cache growth

0.50×/year — halving roughly every 12 months.

Share of releases by attention family

Year	MHA	GQA/MQA	MLA	hybrid	compressed	n
2023	70%	30%	0%	0%	0%	43
2024	23%	66%	6%	4%	0%	94
2025	3%	66%	10%	21%	0%	99
2026	0%	29%	20%	40%	11%	35

Step reduction, KiB per token

Step	Reduction
MHA → GQA/MQA	3.2×
GQA/MQA → MLA	2.3×
MLA → hybrid	2.9×
hybrid → compressed	3.7×
MHA → compressed	78×

GiB per sequence at 128k context

Attention family	GiB @128k
MHA	64.0
GQA/MQA	16.0
MLA	8.6
hybrid	3.1
compressed	0.8

Median values, bf16, batch size 1, 271 releases through 2026-08-13. Hybrid and compressed figures include their fixed per-sequence state. 2026 is a partial year.

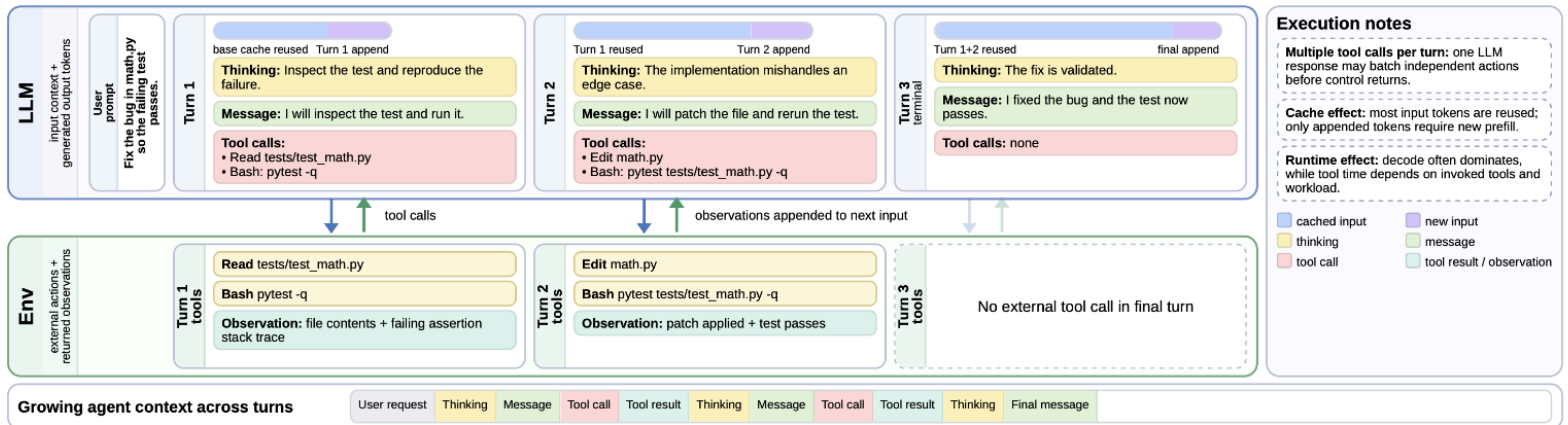
Usage trends

Agentic AI

- Runs LLM in a controlled loop
- Interaction with CPU to run execution environment
 - Bash commands
 - Web search
 - Database calls

CPU runs

- Tool calls in execution environment
- Environment runs in a sandbox
- Control loop called agent harness
- Other orchestration tasks



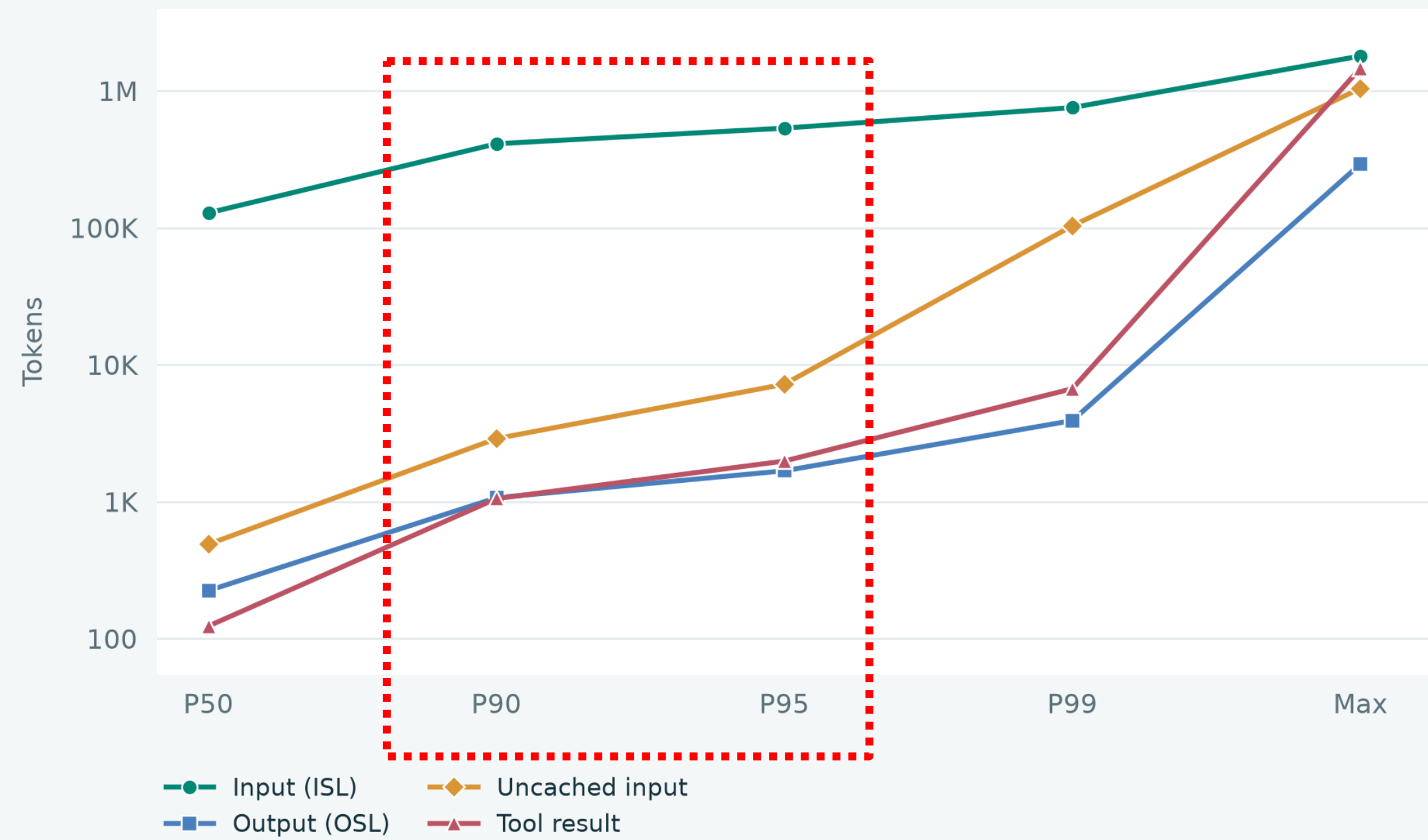
Usage trends

Agentic usage

Agentic workload at a glance

Human-driven top-level sessions · Summary distributions on logarithmic scales

Token lengths



Conversation & tool exposure

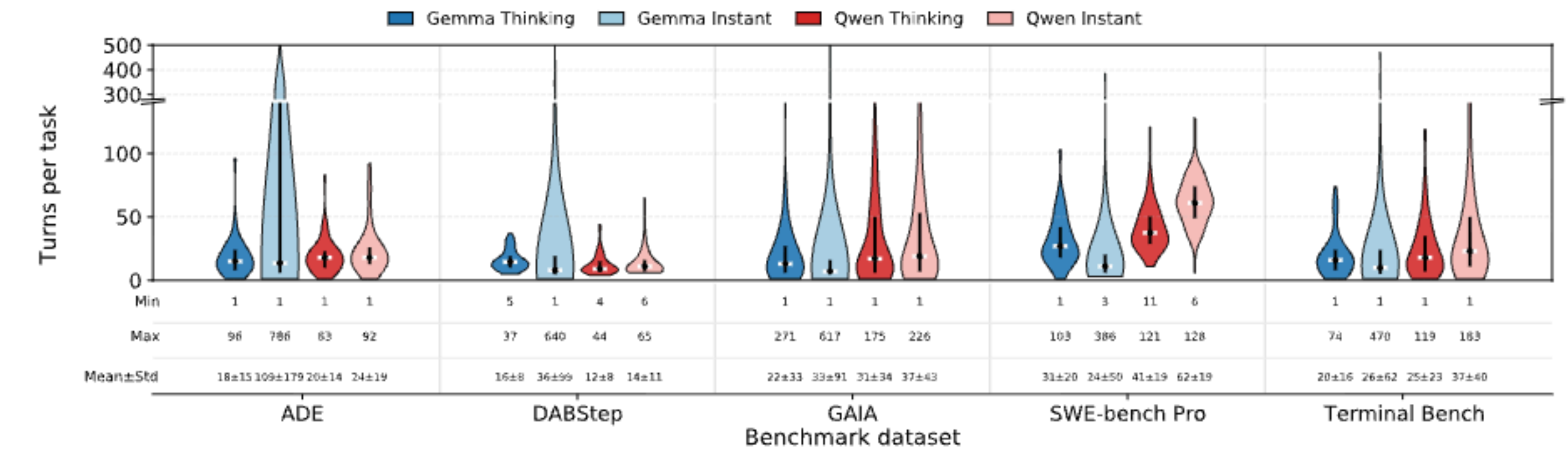
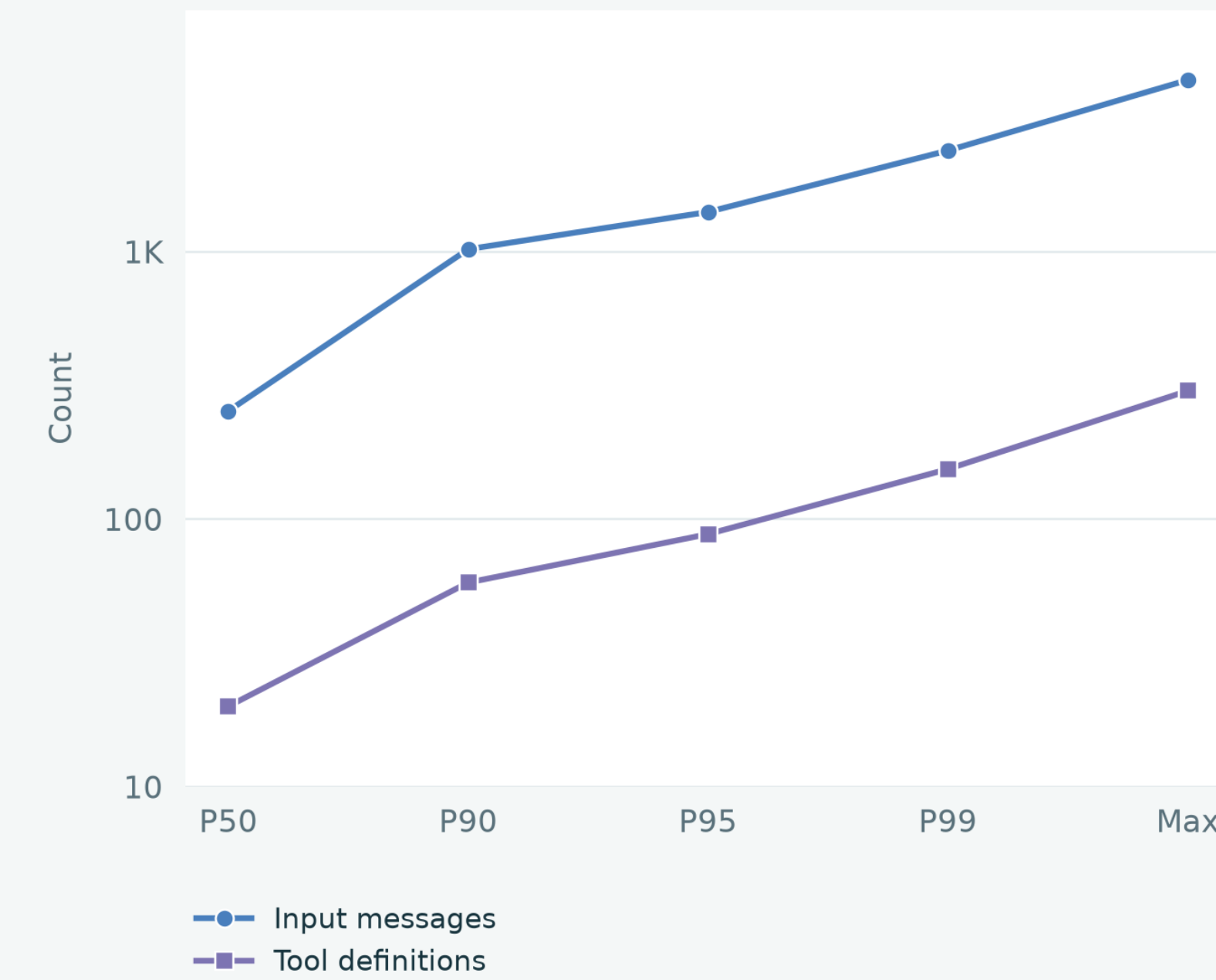


Fig. 3. Distribution of agent turns per task. Agent trajectories are highly variable: instant variants often require more turns and exhibit longer tails, while reasoning variants generally produce more compact interactions.

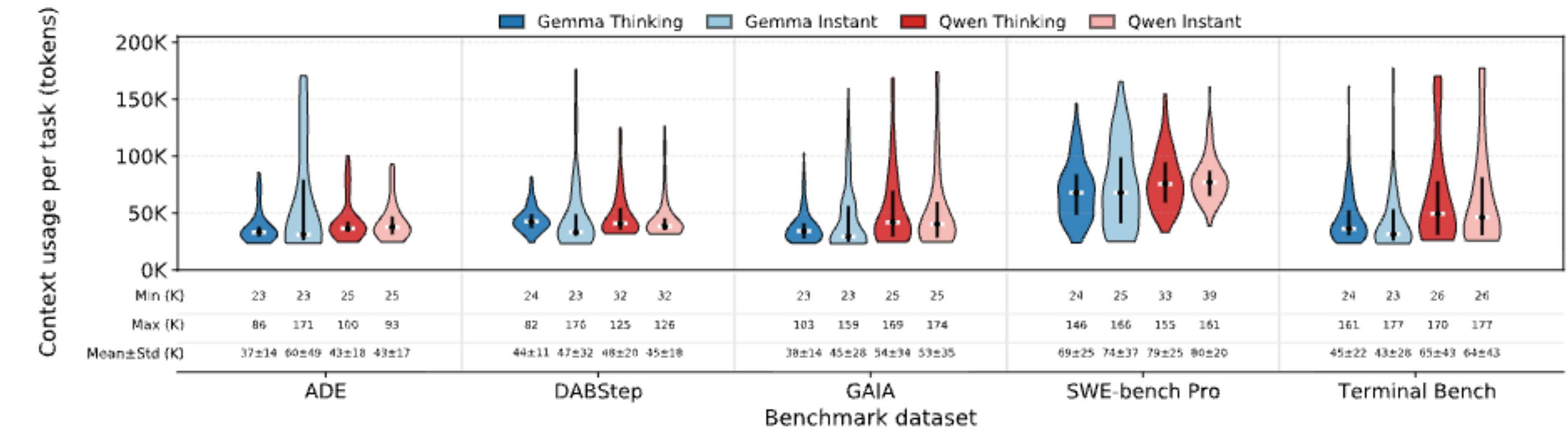


Fig. 4. Distribution of accumulated context usage across all turns for each task. Context usage varies substantially by workload, with SWE-bench Pro consistently producing the largest contexts, while instant variants often introduce longer tails even when their medians remain close to reasoning variants.

https://huggingface.co/datasets/harvardMadsys/freeinference_agentic_trace

<https://arxiv.org/pdf/2605.26297>

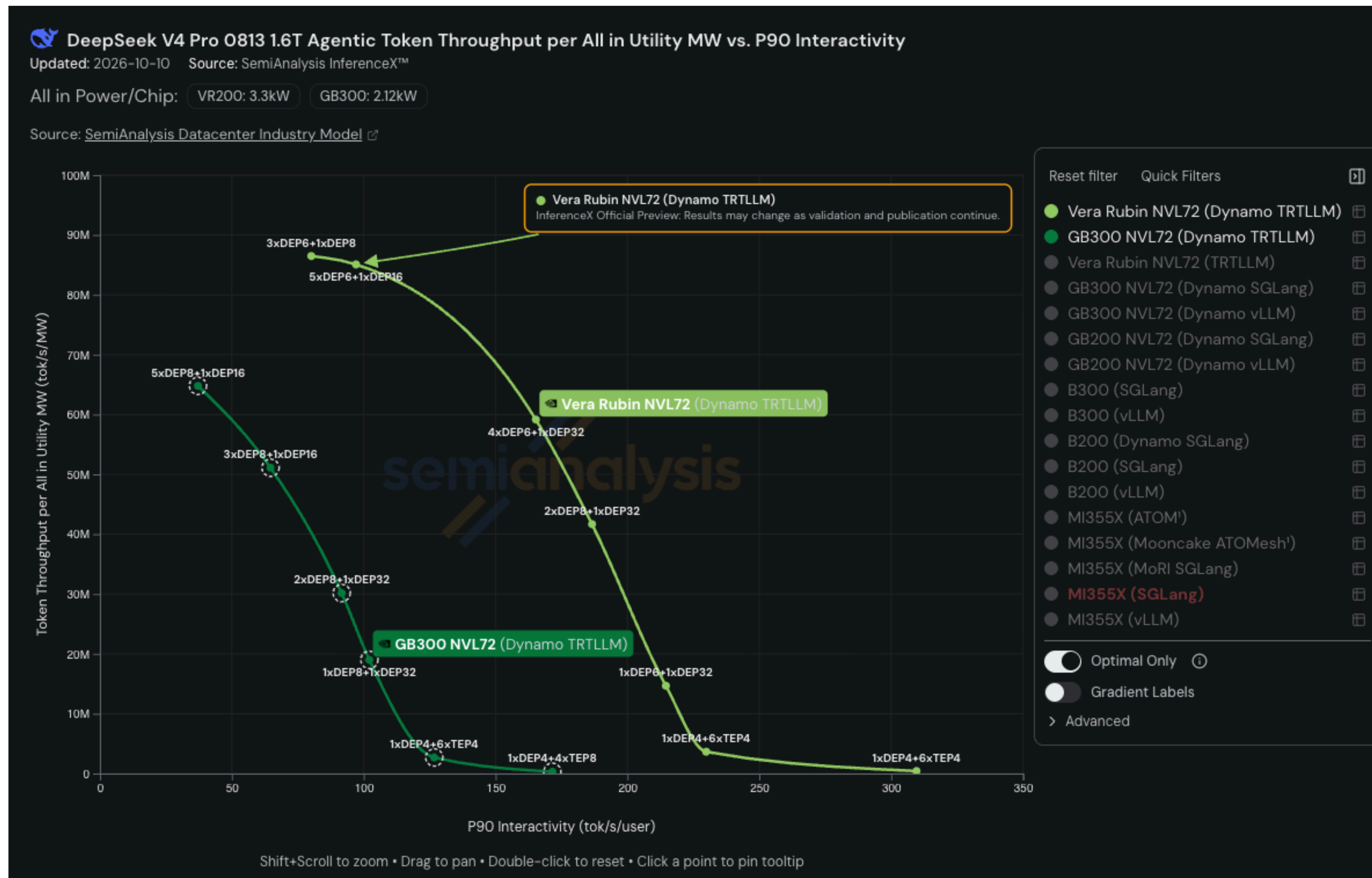
ISL is already hitting million token lengths
turns is growing upwards to thousands !!!

Task completion is the unit of cost

Instant models often need more turns, so cheaper per token can cost more per task.

Challenges in serving today

SLO Requirements vary depending on serving goal



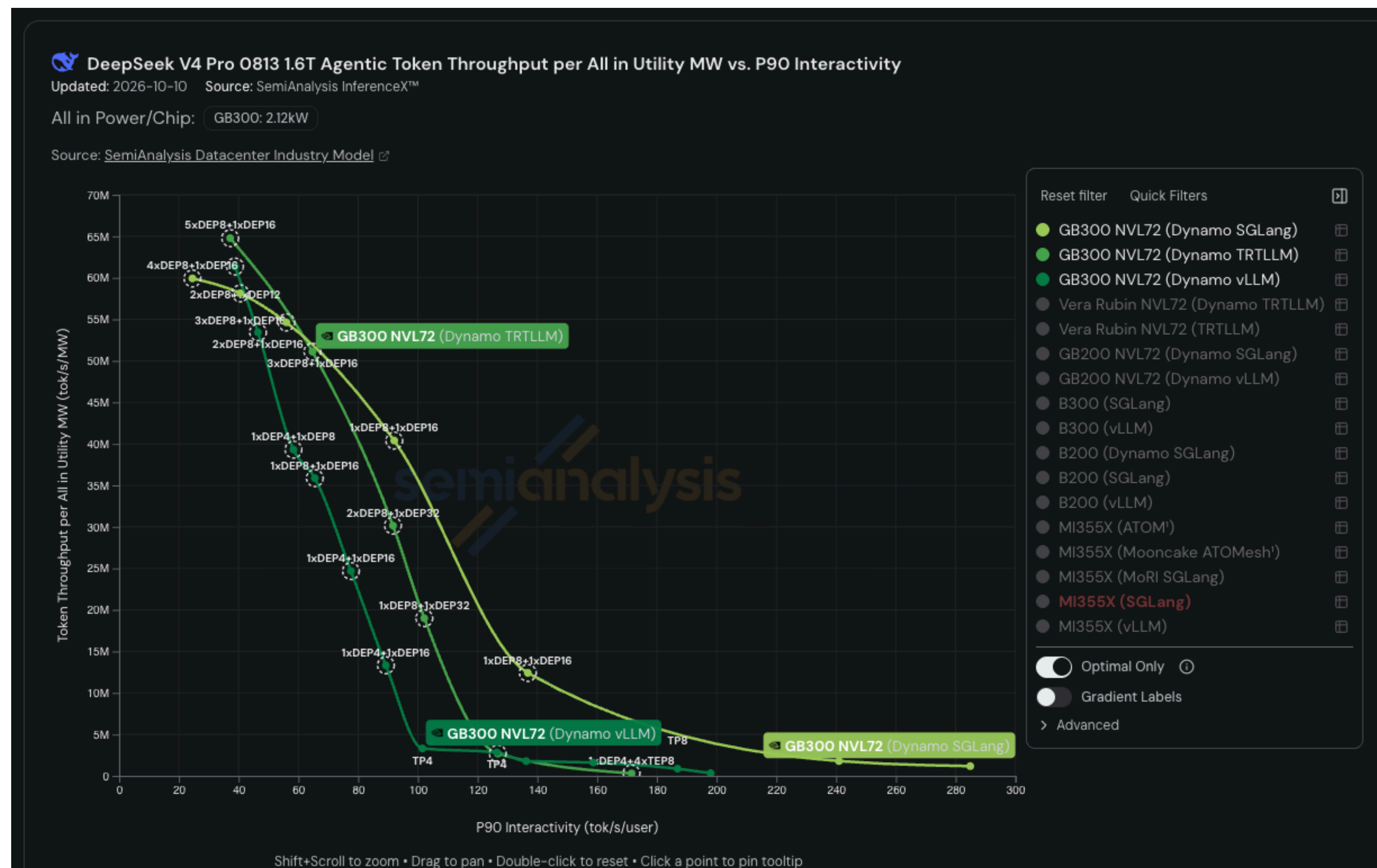
Challenges in serving today

SLO Requirements vary depending on serving goal



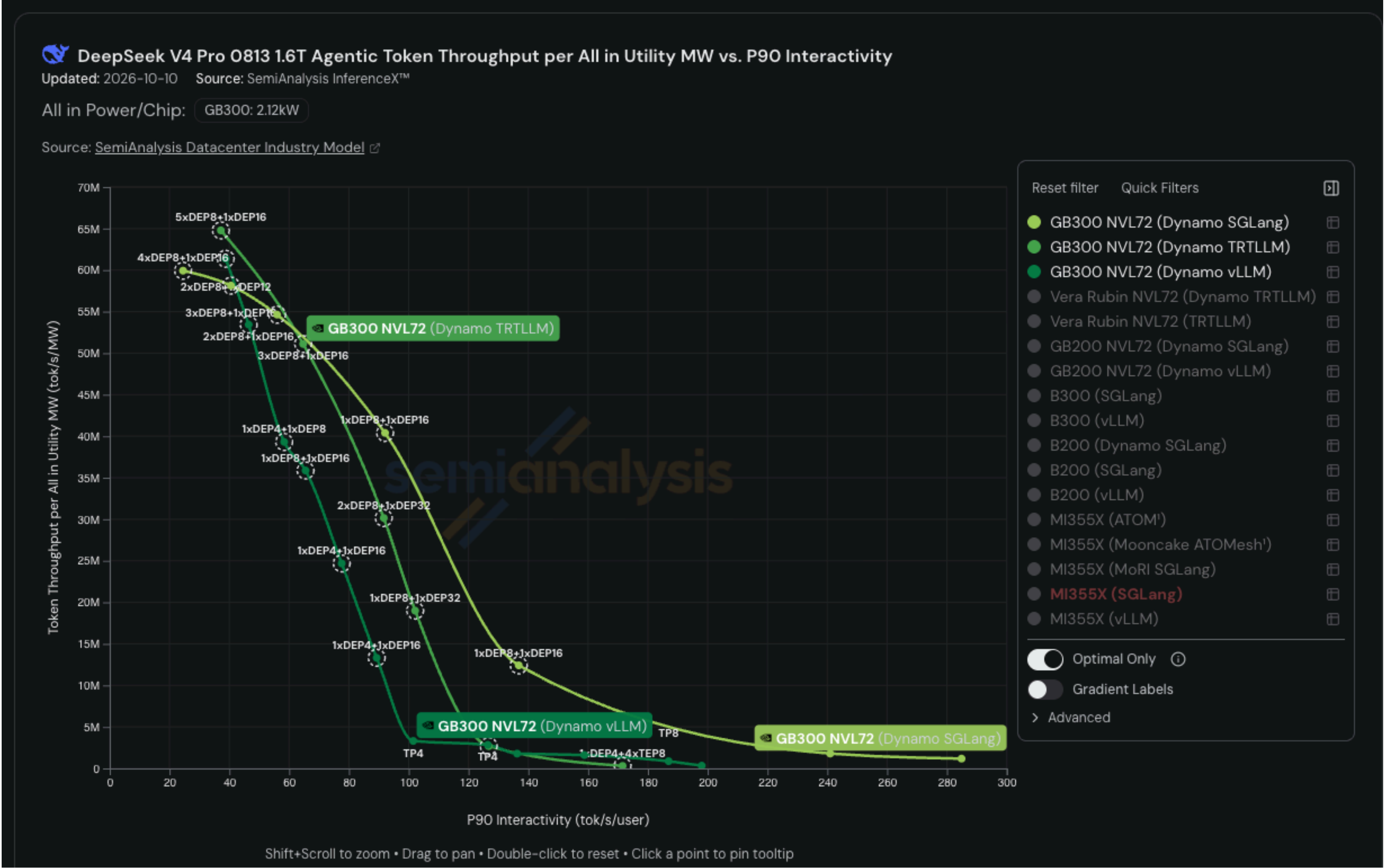
Economics of Inference Serving is an active area of research today!

Heterogeneity - Challenges in serving today



Inference engines like SGLang, vLLM and TRTLLM
provide different optimality given SLO targets
They change over time and over model choices

Heterogeneity - Challenges in serving today



Inference engines like SGLang, vLLM and TRTLLM provide different optimality given SLO targets
They change over time and over model choices

	Amazon Web Services • Cloud Infrastructure and Compute	Global
	Google Cloud • Cloud Infrastructure and Compute	Global
	Crusoe • Cloud Compute	US, Ireland
	Oracle • Cloud Compute	United States, United Kingdom, United Arab Emirates
	Slack • Communications	United States
	WorkOS • Single Sign On	United States
	Pylon • Customer Support	United States
	Orb • Invoicing	United States
	Coreweave • Cloud Compute	United States
	Nebius • Cloud Compute	United States, France, Finland
	Lambda • Cloud Compute	Global
	Hydra Host • Cloud Compute	United States, Iceland
	Vultr • Cloud Compute	United States
	Voltage Park • Cloud Compute	United States
	Radiant • Cloud Compute	United States
	Fluidstack • Cloud Compute	United States, Canada
	Cirrascale • Cloud Compute	United States

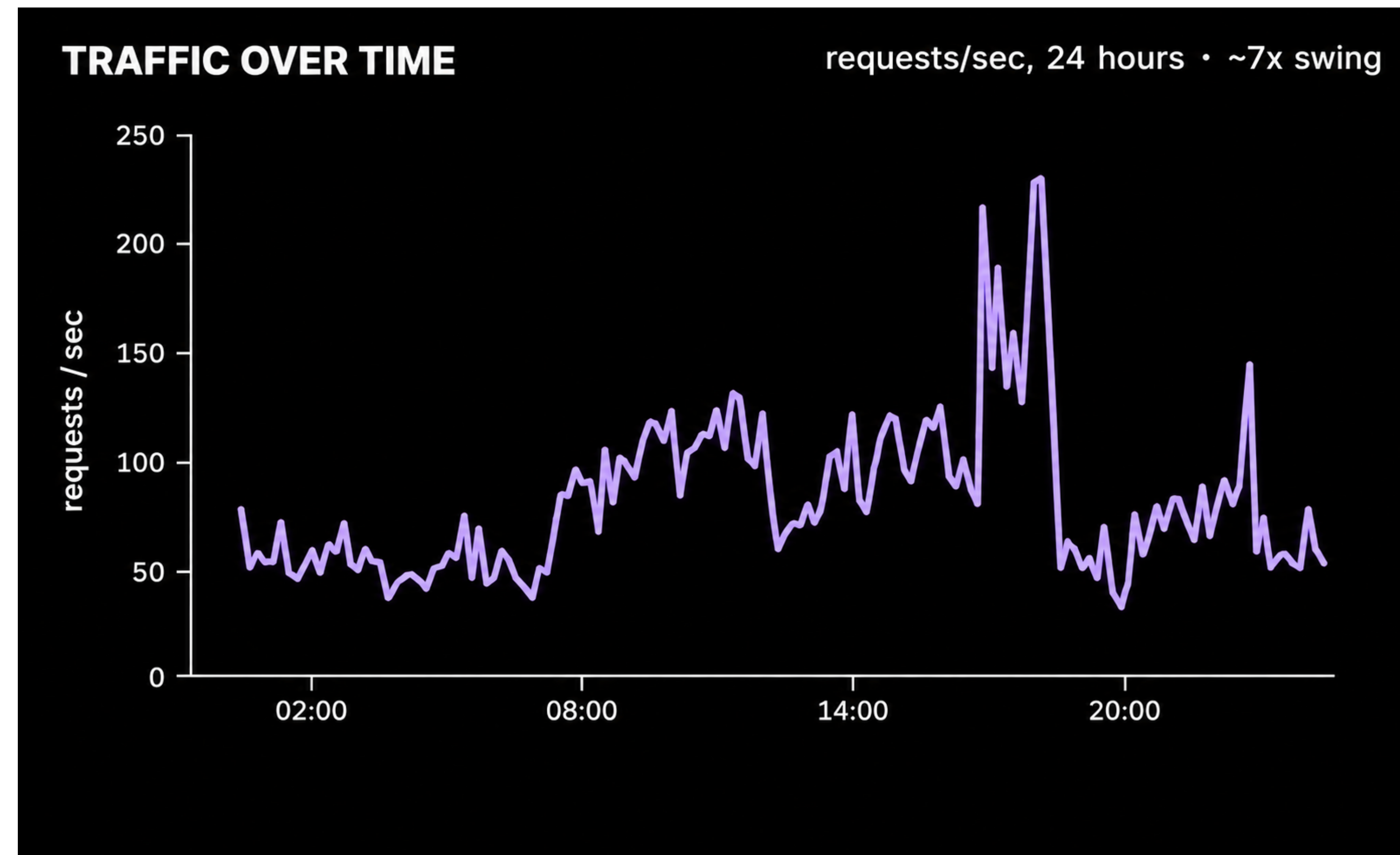
<https://trust.baseten.co/subprocessors>

	Amazon Web Services • Cloud infrastructure and service provider	United States, Japan, Singapore, Australia, Germany, United Kingdom
	Google Cloud Platform • Cloud infrastructure and service provider	United States, Japan, Australia, Sweden, United Kingdom, Germany, Netherlands, Switzerland, France
	Oracle Cloud Infrastructure • Cloud infrastructure and service provider	United States, Japan, United Kingdom, Germany, Australia
	Cloudflare • Content delivery network provider	Processing is performed at the data center that is closest to the End User
	Vercel • Frontend hosting	United States
	Pylon • Customer support platform	United States
	Linear • Internal support collaboration	United States
	Discord Inc. • Communication services	United States
	Slack • Collaboration	United States
	Lambda Labs • Infrastructure provider	United States
	Crusoe • Infrastructure Provider	United States, Iceland
	Vultr • Infrastructure Provider	United States
	CoreWeave • Infrastructure Provider	United States
	DigitalOcean • Infrastructure Provider	United States
	Fluidstack • Infrastructure Provider	United States
	TensorWave • Infrastructure Provider	United States
	IDFN • Infrastructure Provider	Canada

<https://trust.fireworks.ai/subprocessors>

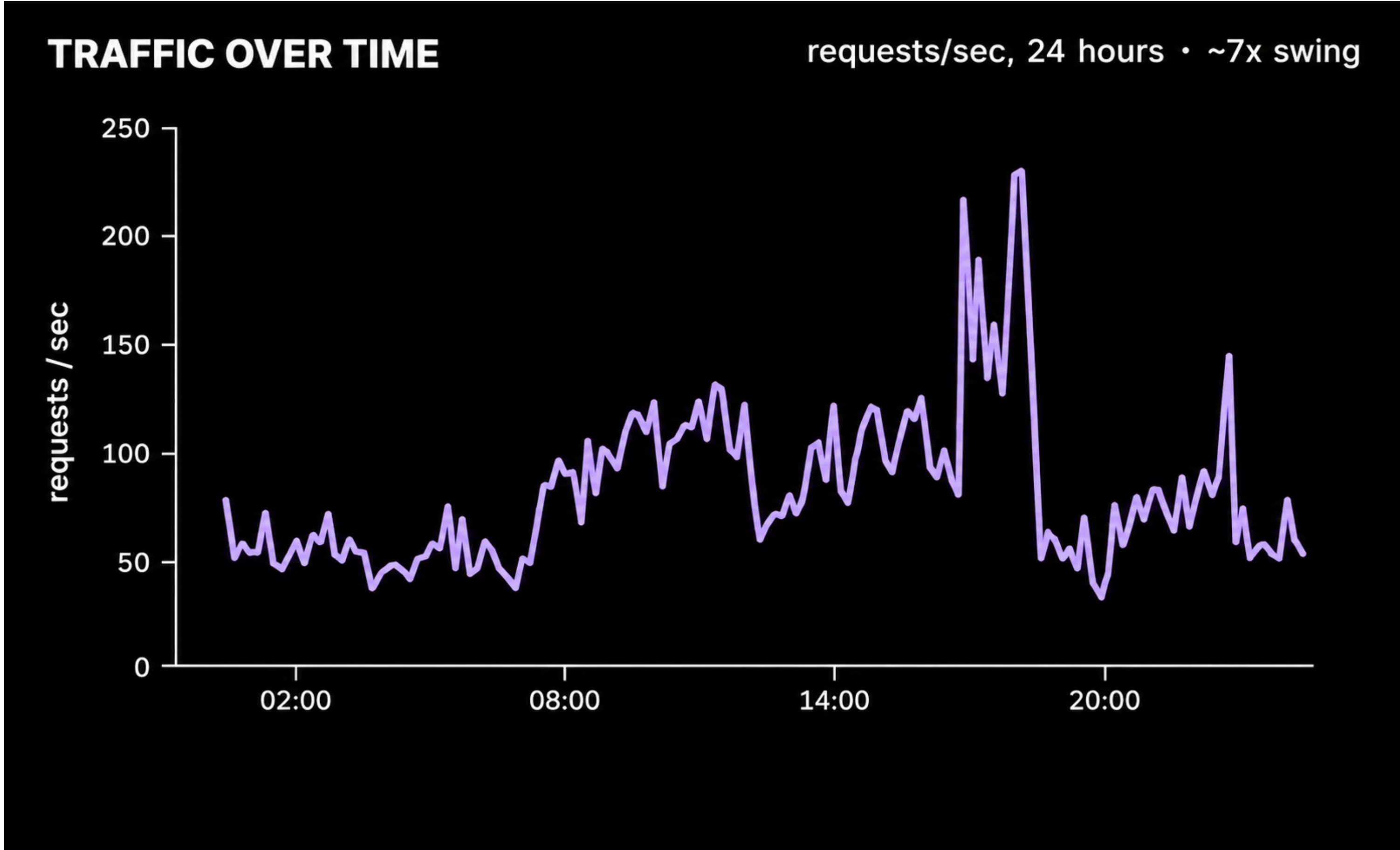
ISP providers need to use multiple hardware and DC providers to serve tokens
Each DC providers have unique HW and SW stack making serving very complex

Challenges in serving today



Source: DeepInfra, Dynamo Event, July'26

Challenges in serving today



Source: DeepInfra, Dynamo Event, July'26

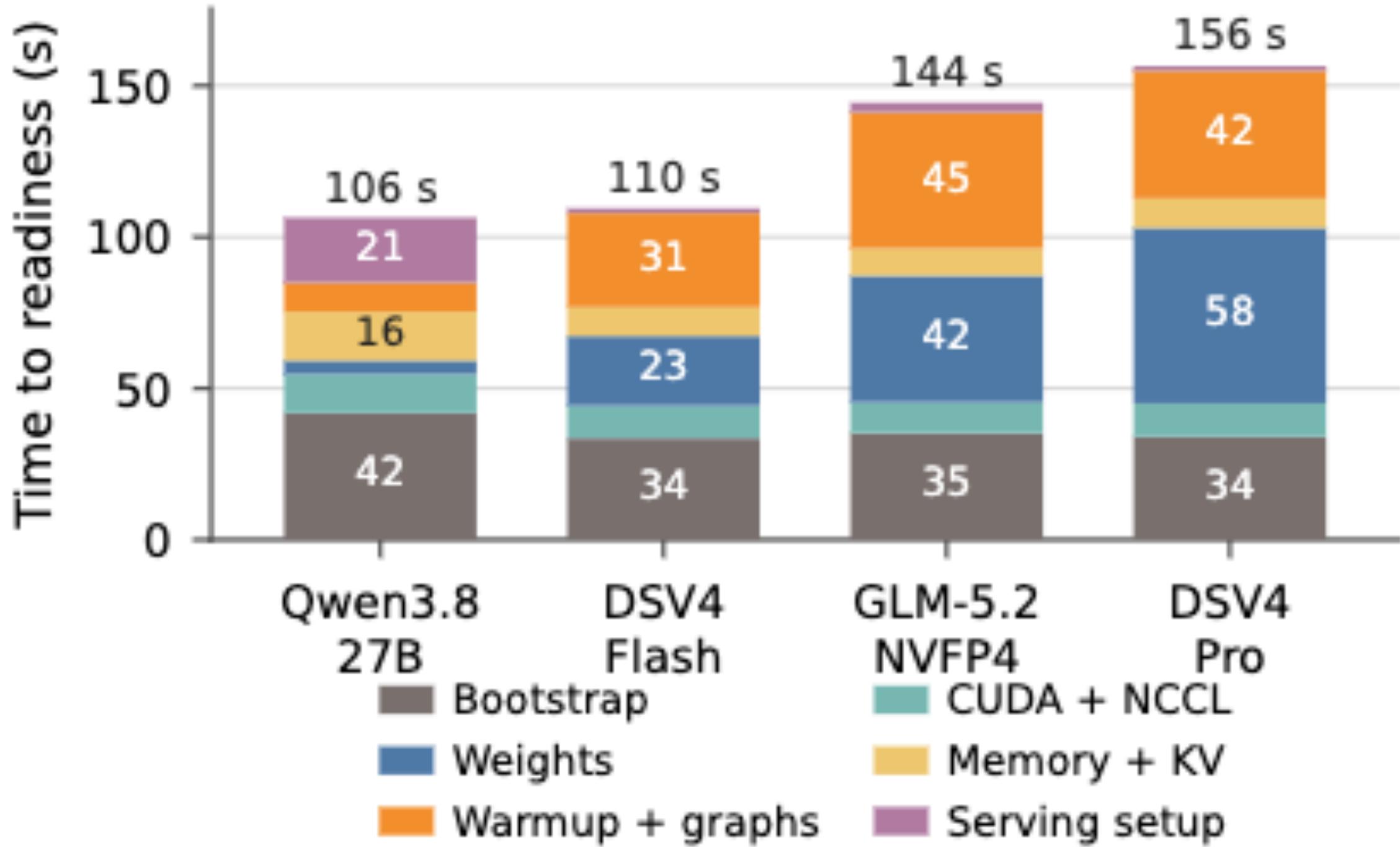
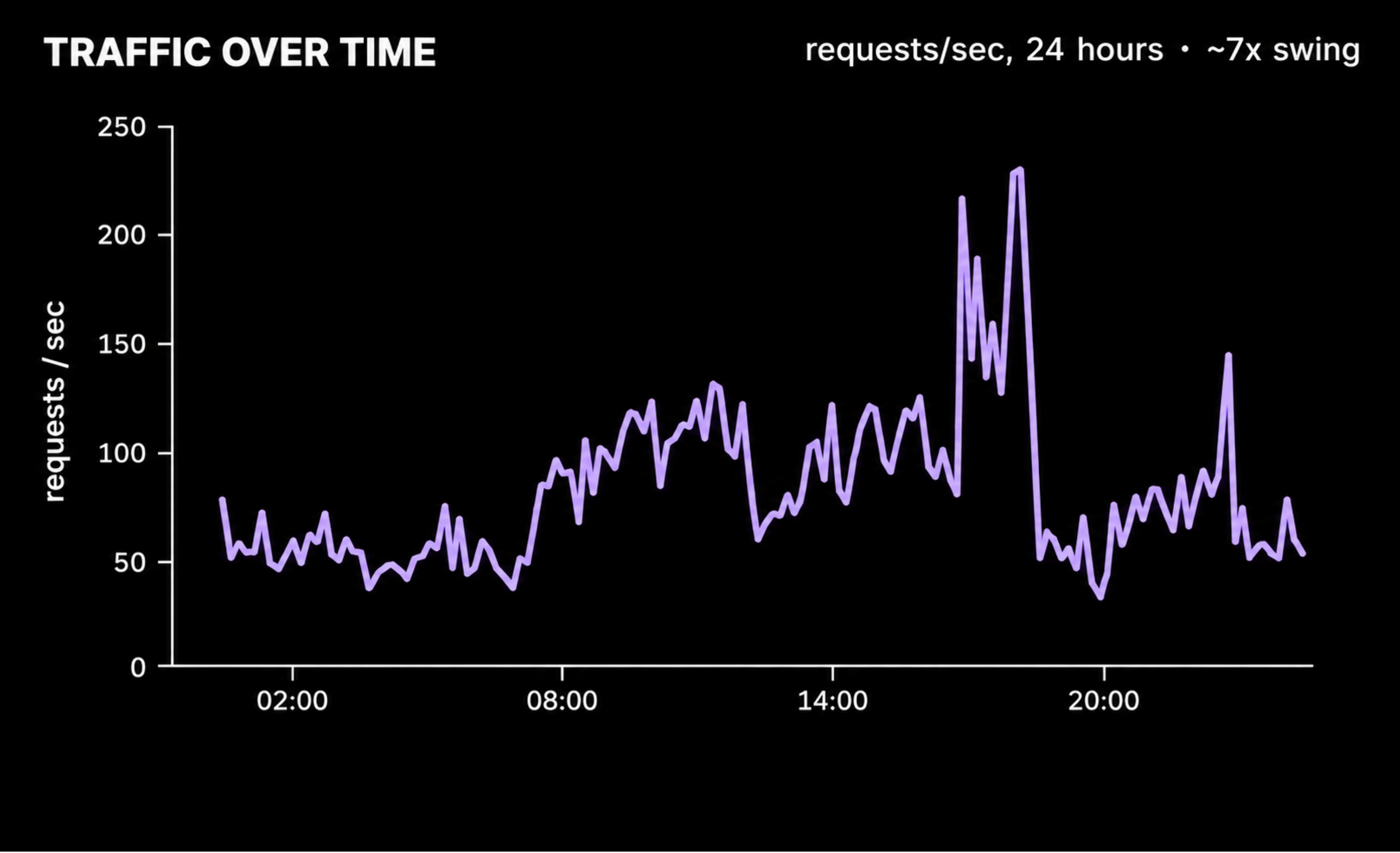


Figure 1. Restart time for vLLM 0.27.1 on 8xB200 GPUs (TP/TEP8). Compilation artifacts are pre-cached and weights pre-pulled into page cache.

<https://arxiv.org/pdf/2609.25451>

Challenges in serving today



Source: DeepInfra, Dynamo Event, July'26

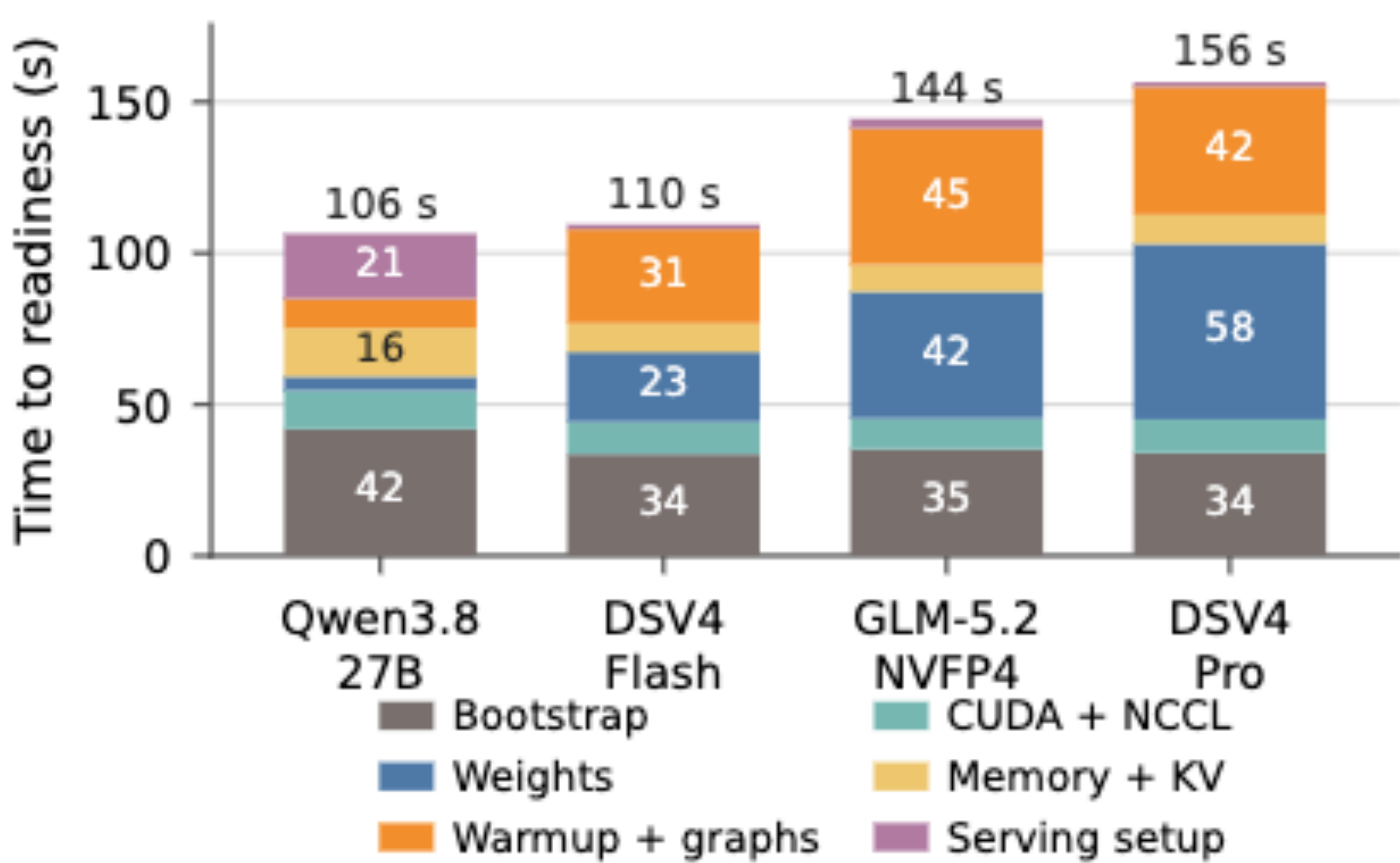


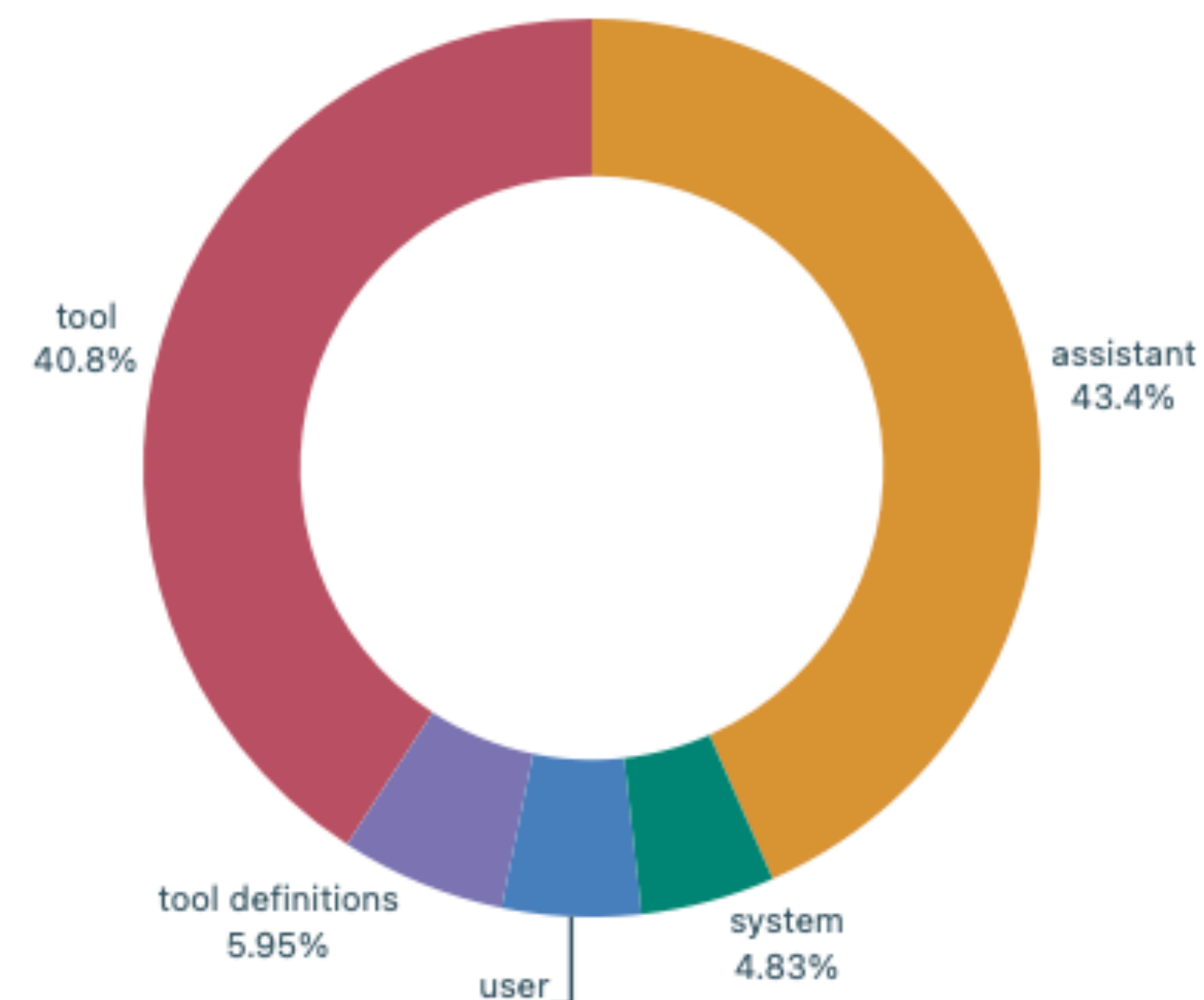
Figure 1. Restart time for vLLM 0.27.1 on 8xB200 GPUs (TP/TEP8). Compilation artifacts are pre-cached and weights pre-pulled into page cache.

<https://arxiv.org/pdf/2609.25451>

Providers expect availability of 99.99999% while maintaining SLO

Challenges in serving today

What fills the input?



Sum of o200k_base role-token counts across calls. These are repeated input exposures, not unique content; their tokenizer differs from provider usage.

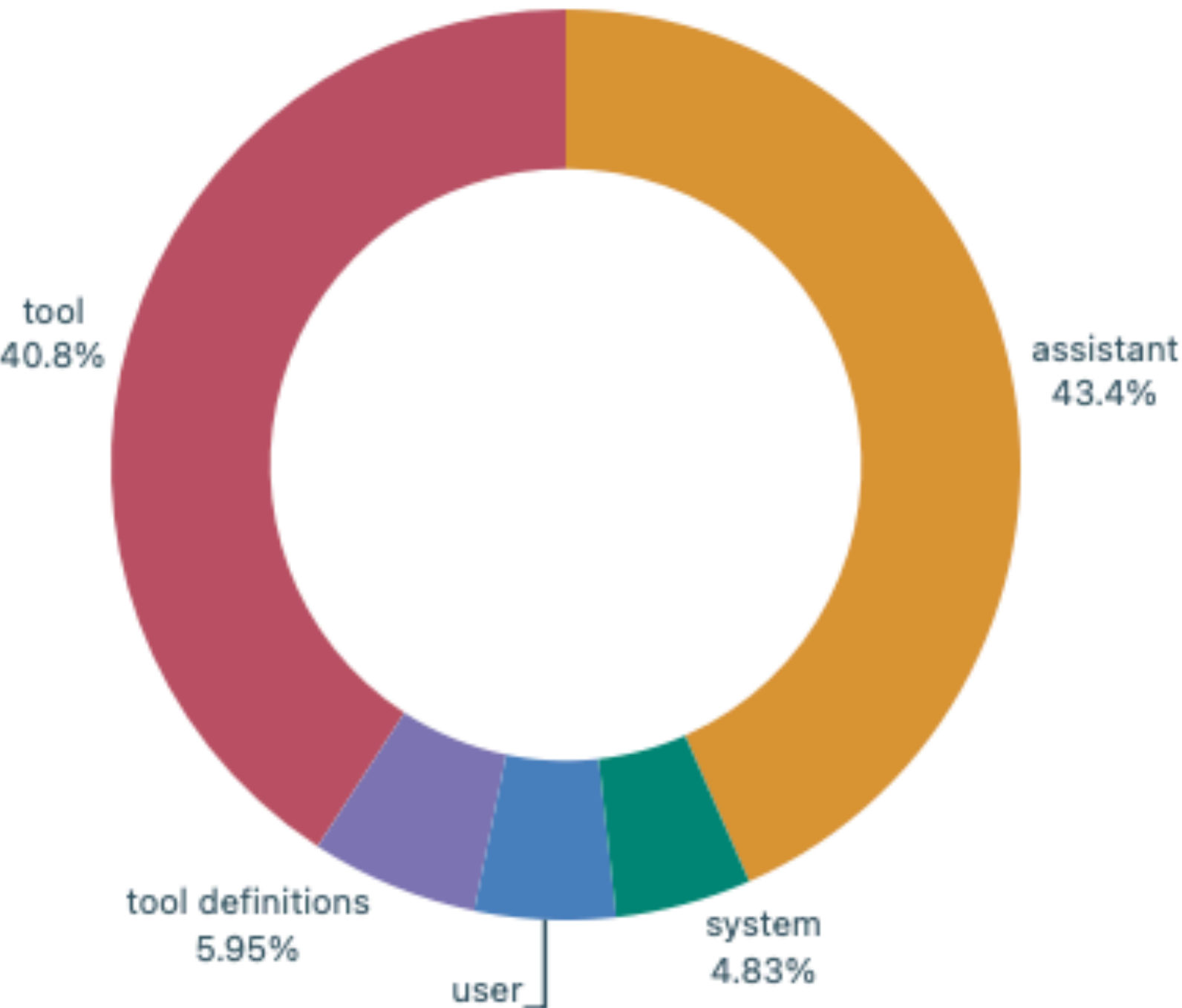
Input ctx length gets filled very quickly with tool calls output

“Compaction”

Compaction results in dropping KV cache hit rate to 0%

Challenges in serving today

What fills the input?



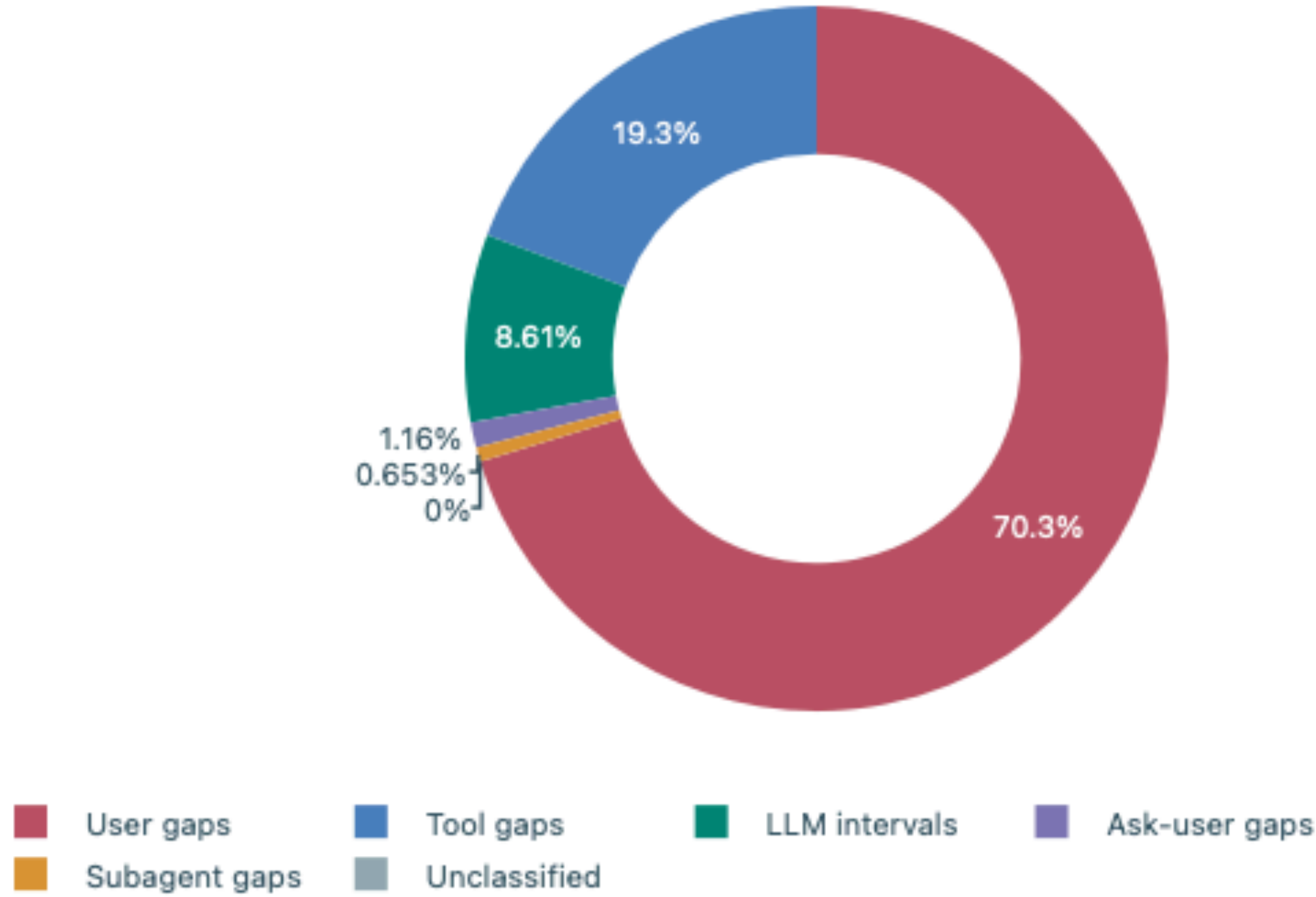
Sum of o200k_base role-token counts across calls. These are repeated input exposures, not unique content; their tokenizer differs from provider usage.

Input ctx length gets filled very quickly with tool calls output

“Compaction”

Compaction results in dropping KV cache hit rate to 0%

Complete session time budget



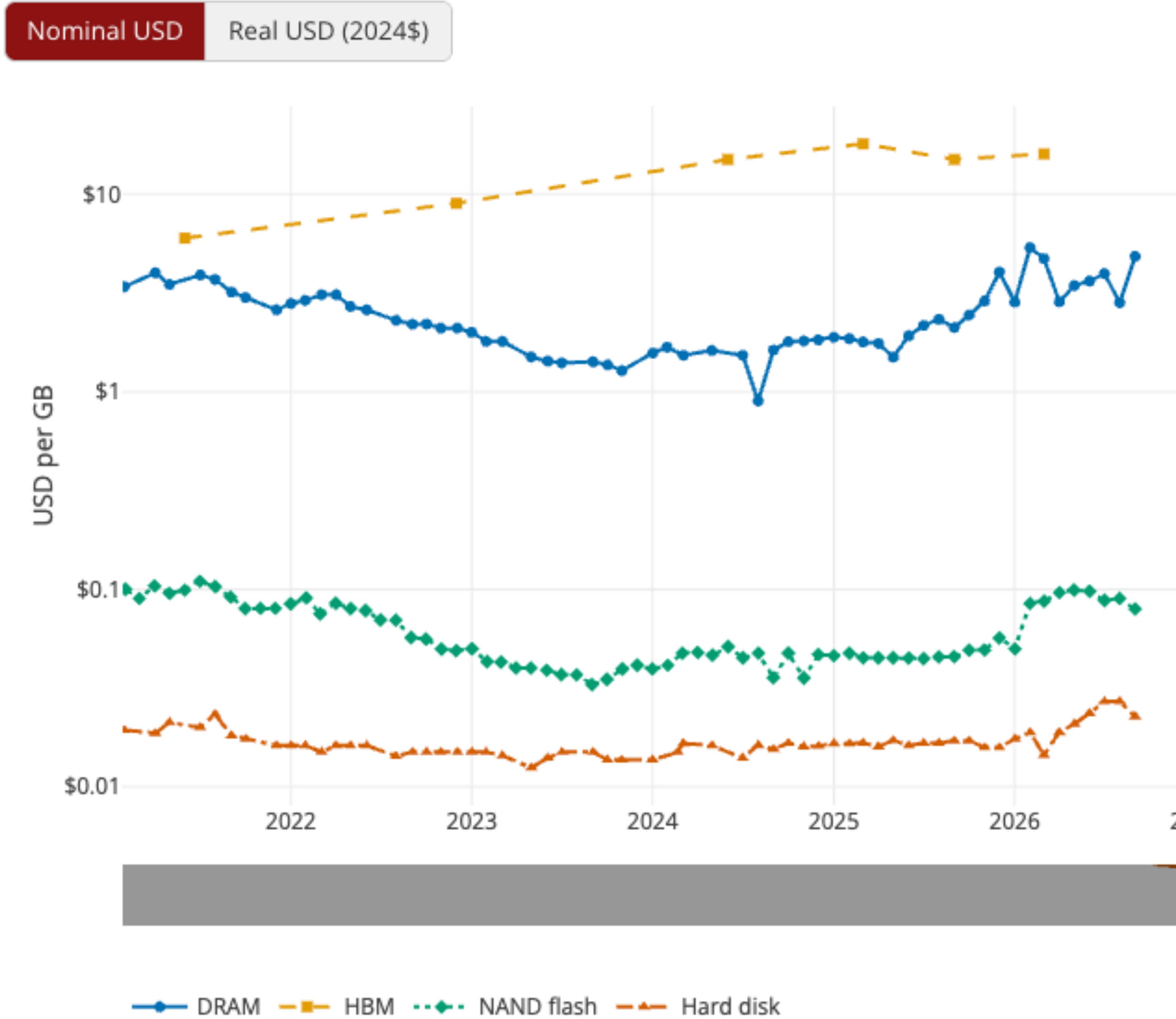
LLM intervals are unioned within each session. Gaps are assigned by the following request's released waited_for label. This reconciles exactly to observed session span.

Most sessions are sitting idle and waiting for user input

Keeping states resident is an expensive proposition

Price per gigabyte over time

Historical lowest \$/GB on a log scale — one line per memory type: **DRAM**, **NAND flash**, **HBM**, and **hard disks**. Toggle nominal vs **inflation-adjusted** dollars (constant 2024 \$, US CPI-U).



Challenges Summary

- **Models are getting larger:** MoE is becoming the dominant scaling architecture.
- **State is still growing:** KV cache per token is shrinking, but longer contexts continue to increase total state.
- **Task completion is the new metric:** Agents require longer ISL, OSL, and substantially more turns.
- **Performance/W/\$ is the optimization target:** The Pareto-optimal configuration varies significantly across applications SLOs and workloads.
- **Inference engines are heterogeneous:** No single engine consistently delivers the best Performance/W/\$ across workloads and over time.
- **Infrastructure is heterogeneous:** Capacity increasingly spans multiple datacenter providers with different hardware and infrastructure stacks.
- **Demand is highly dynamic:** Rapid arrival-rate changes require fast autoscaling, cold starts, and warm restarts.
- **Availability must approach 100%:** Failures and recovery should have minimal impact on user experience.
- **State cannot always remain in HBM:** Agents frequently wait on users and tools, making continuous state movement and management essential.
- **Recomputation can beat storage:** In some regimes, recomputing KV state is cheaper than preserving it.

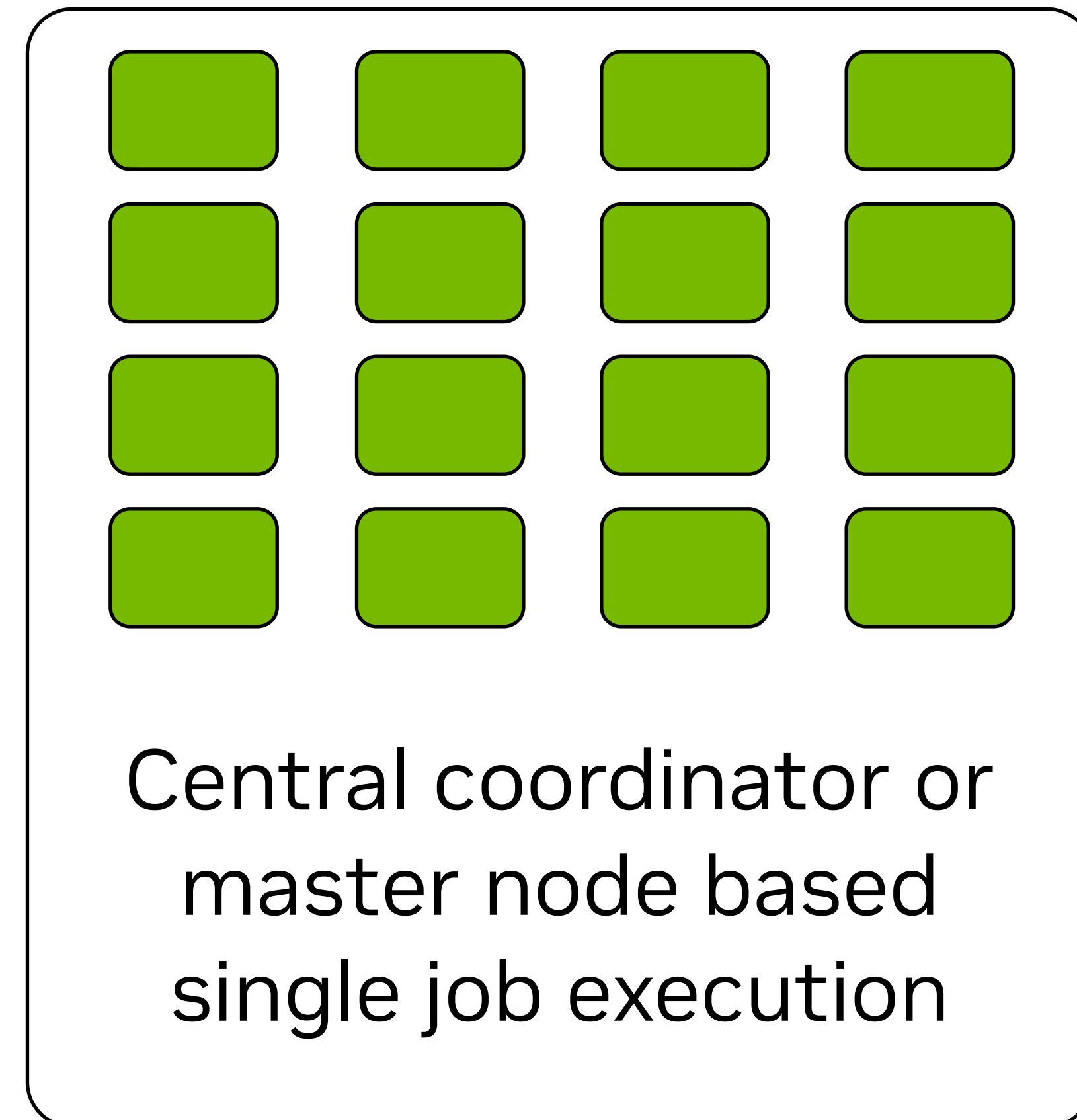


Inference

Inference

Different compared to Training

Parallelism strategies
Failure and resiliency
Weak and strong scaling
Topology awareness
Preprocessing data
Optimizers
....



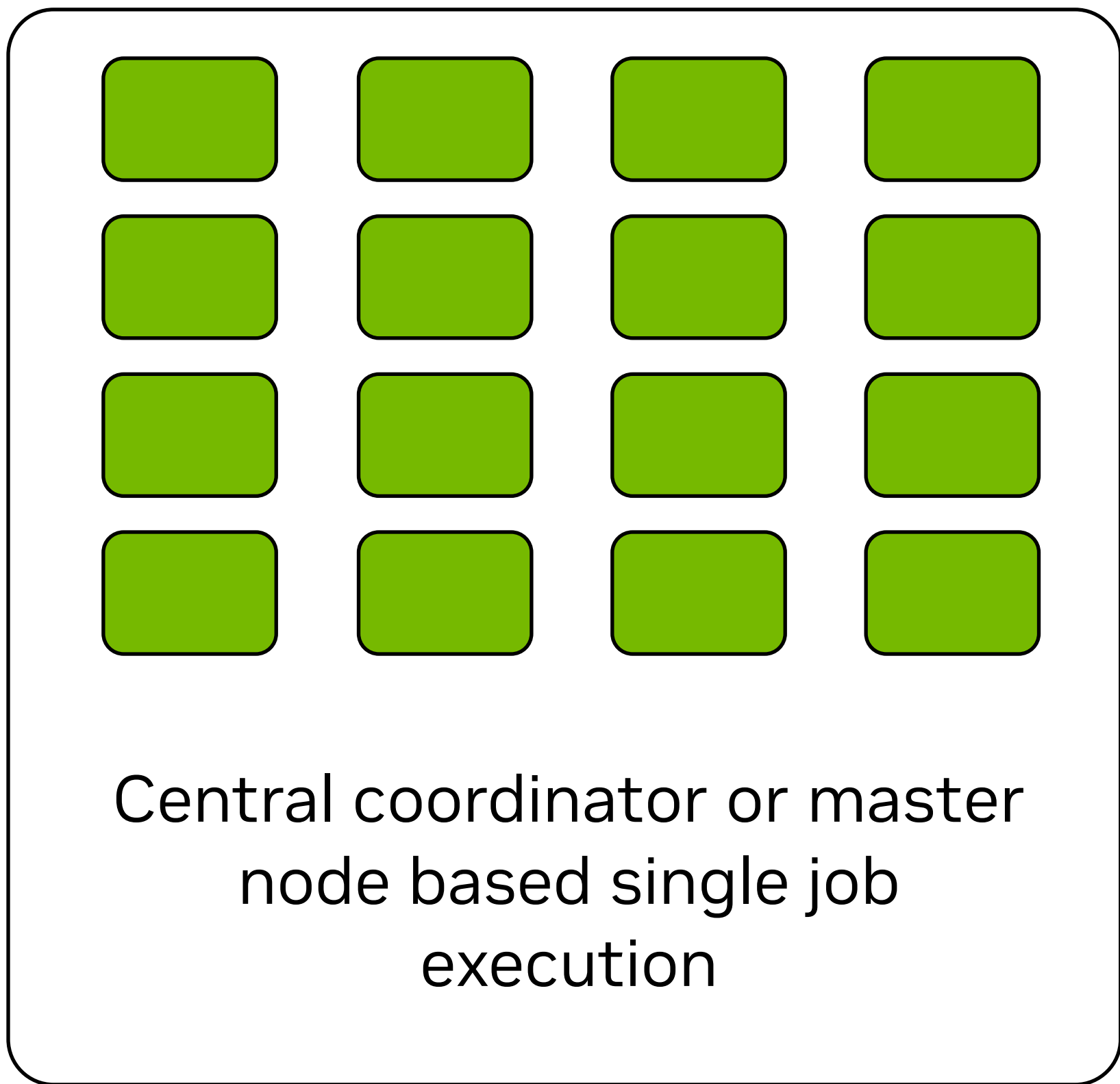
LLM Training

Inference

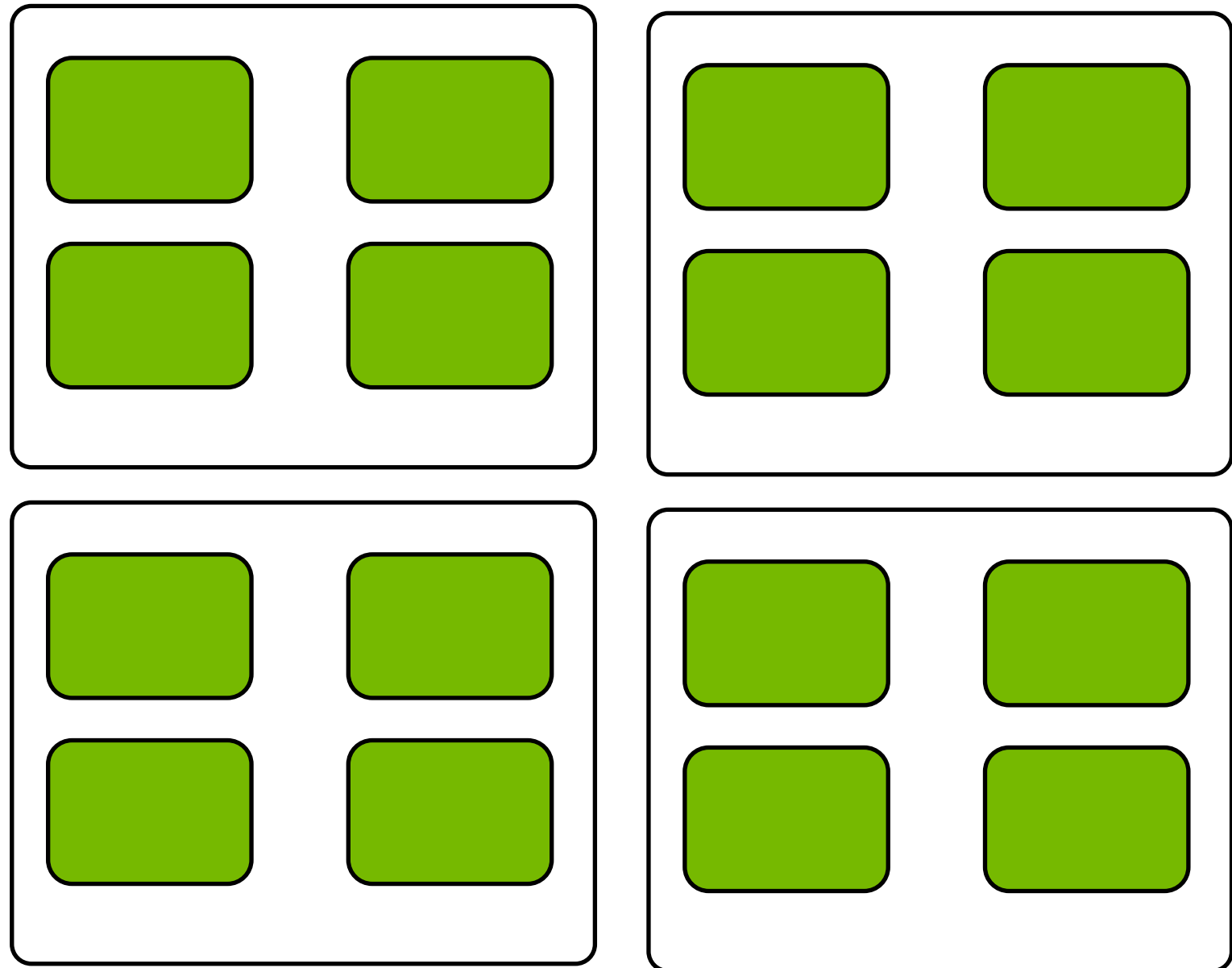
Different compared to Training

Unique challenges and lot of pending optimizations in both stacks

Parallelism strategies
Failure and resiliency
Weak and strong scaling
Topology awareness
Preprocessing data
Optimizers
....



LLM Training



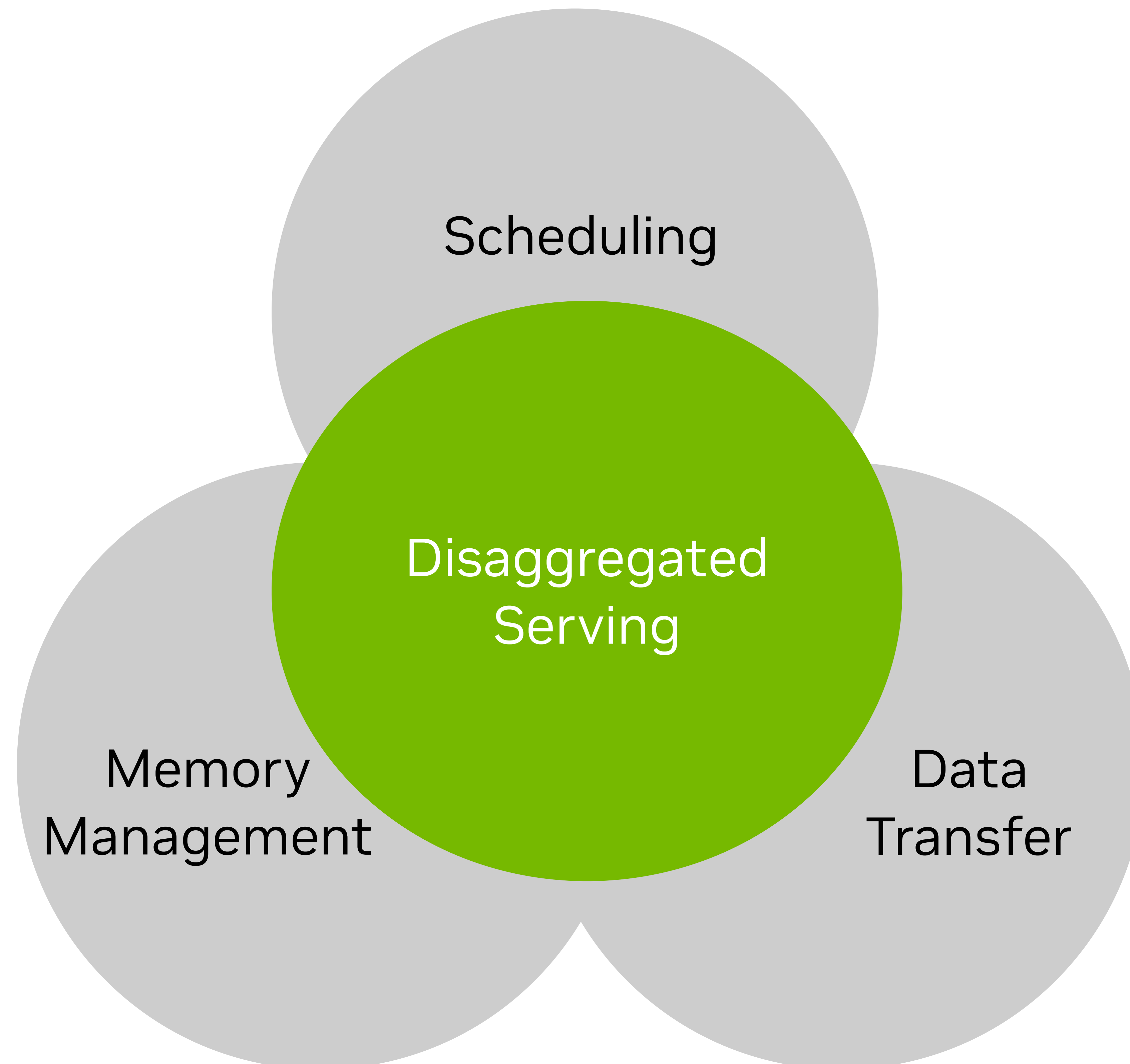
LLM Inference

Inflight batching
KV cache optimization
Quantization
Parallelism strategies
User-defined scheduling
Fault tolerance
Cold start
Hardware heterogeneity
...

Dimension	Training	Inference
Unit of work	One job, weeks	Millions of requests, ms to minutes, each with an SLO
Load	Steady, known batch	~7× daily swing (13); new-token count varies ~1000×
State	Weights + optimizer, fixed	Per-session KV that grows, idles, must move
Failure	Checkpoint, lose minutes	User sees it now; 99.99999% target
Scheduling	Placed once per run	Decided per request, in milliseconds
Success metric	Time to train, MFU	TTFT, ITL, tokens/W/\$ per tier

NVIDIA Dynamo: Distributed Inference At Scale

Systematic approach to AI inference at scale



System level optimization

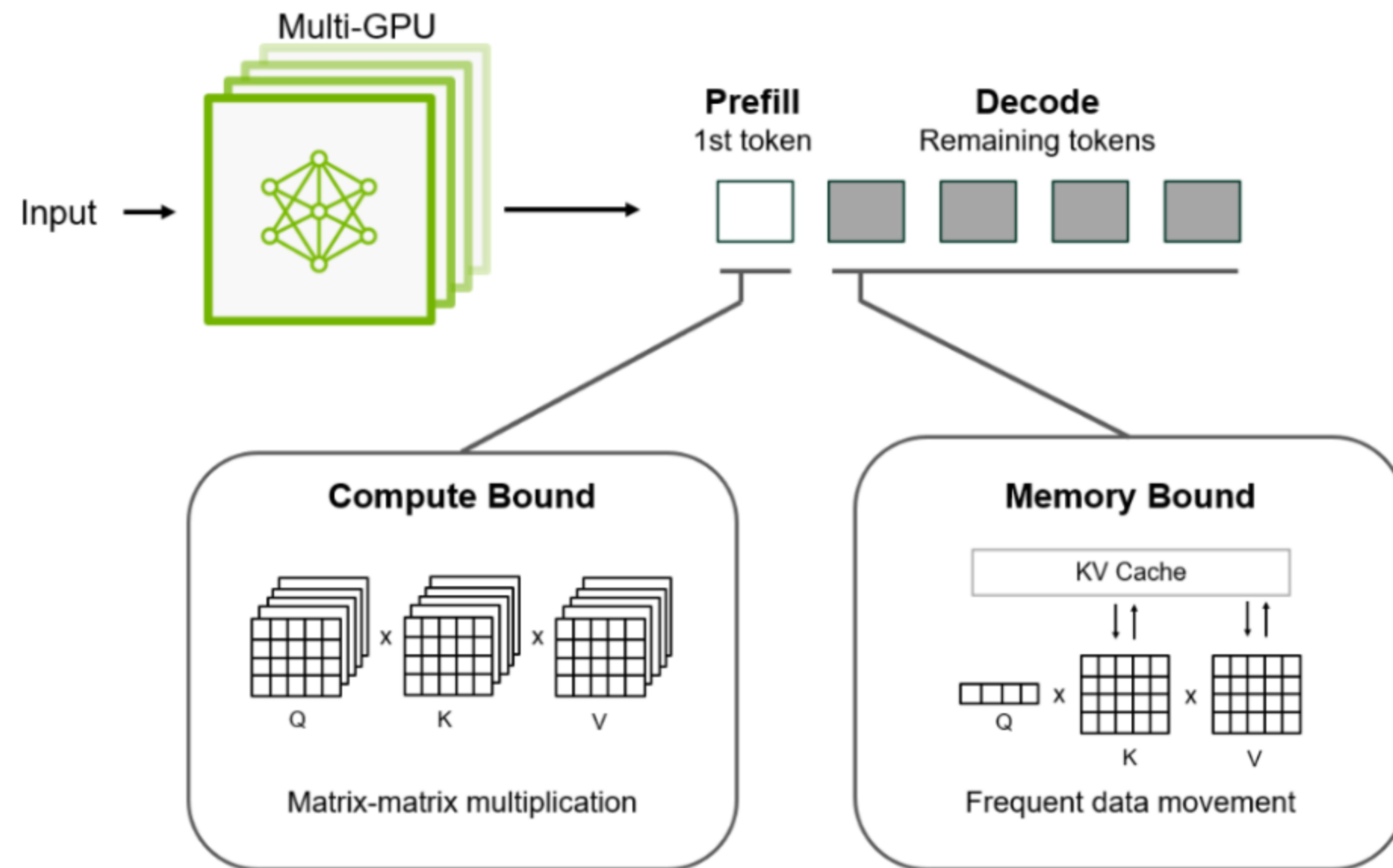
**Supports all
inference engines**

Modular Components

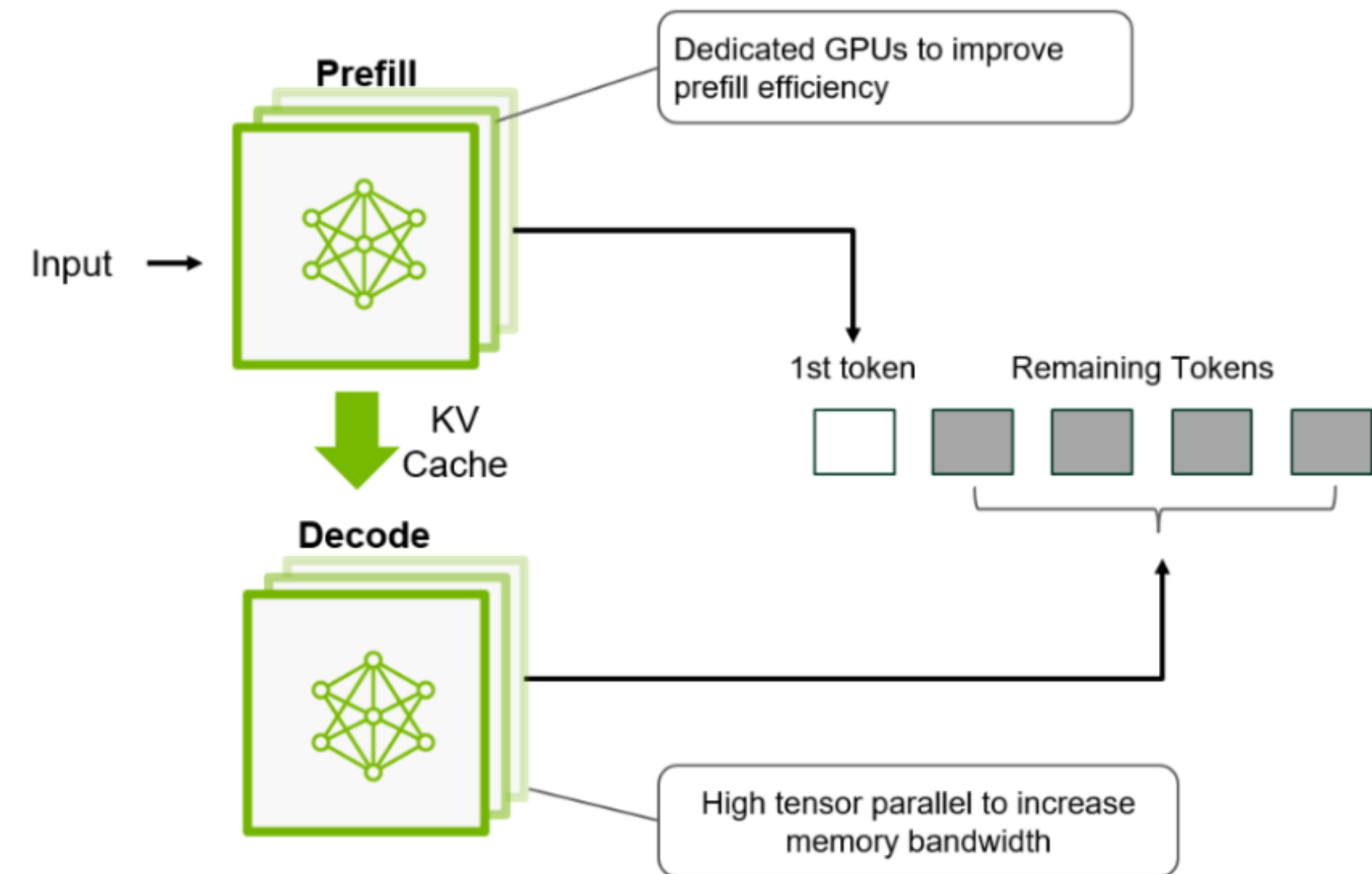
Production grade serving

Disaggregated Serving

Traditional Serving

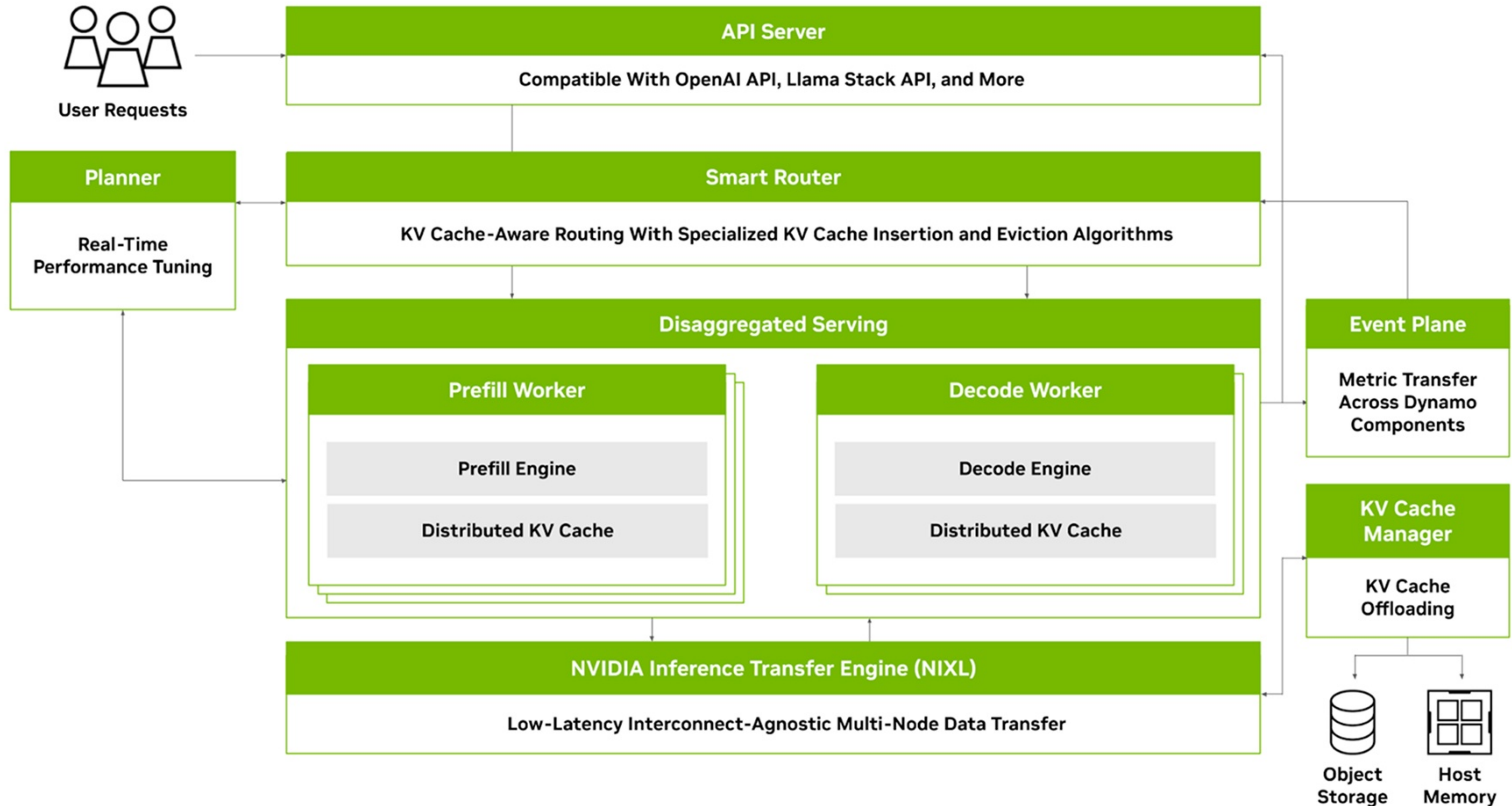


Disaggregated Serving



NVIDIA Dynamo: Distributed Inference At Scale

Modular design, independently scalable yet highly efficient



Inside Dynamo

Request plane · state plane · control plane

CONTROL PLANE

size · place · start

03

Planner

How many prefill and decode workers?

07

Grove

Gang-schedule and place pods on Kubernetes

08

Snapshot (checkpoint/restore)

Start new workers in seconds

REQUEST PLANE

one request's path

Clients

apps, agents

01

Frontend

OpenAI/Anthropic API ·
tokenize · stream

02

Router

Pick a worker: cache vs.
load

04 Aggregated vs. disaggregated serving

Prefill workers

Compute the prompt's KV
cache

Decode workers

Generate tokens one by one

STATE PLANE

KV cache & data
movement

05

KVCR

KV cache shared across nodes and tiers

06

NIXL

Move KV bytes over any fabric, to any memory

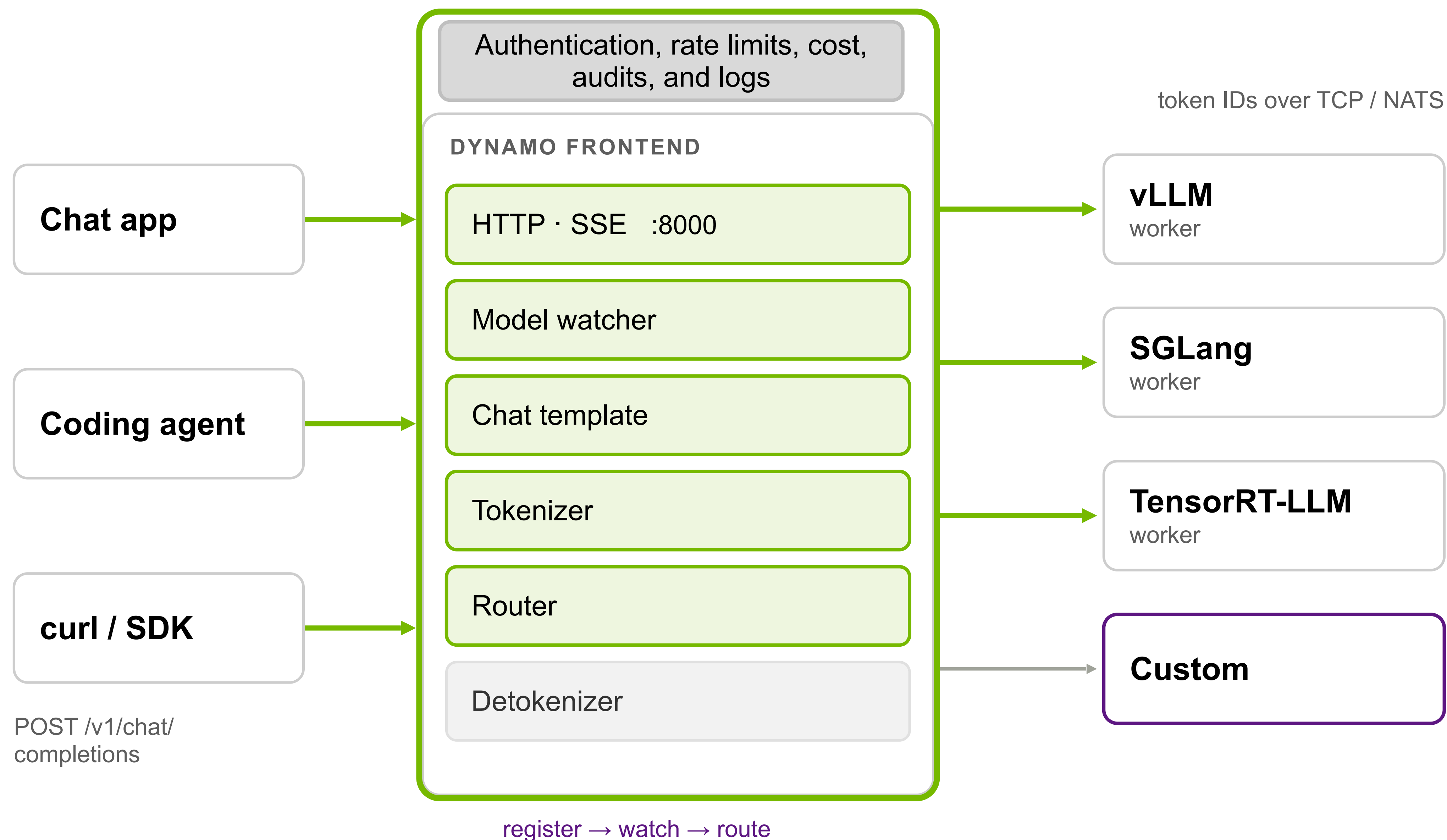
Frontend (Model Gateway)

The model's front door: one API for every engine, tokenized once, streamed back

An API gateway. It speaks OpenAI-compatible HTTP (and KServe gRPC), turns each request into tokens, hands them to the router, and streams the answer back.

WHY IT MATTERS

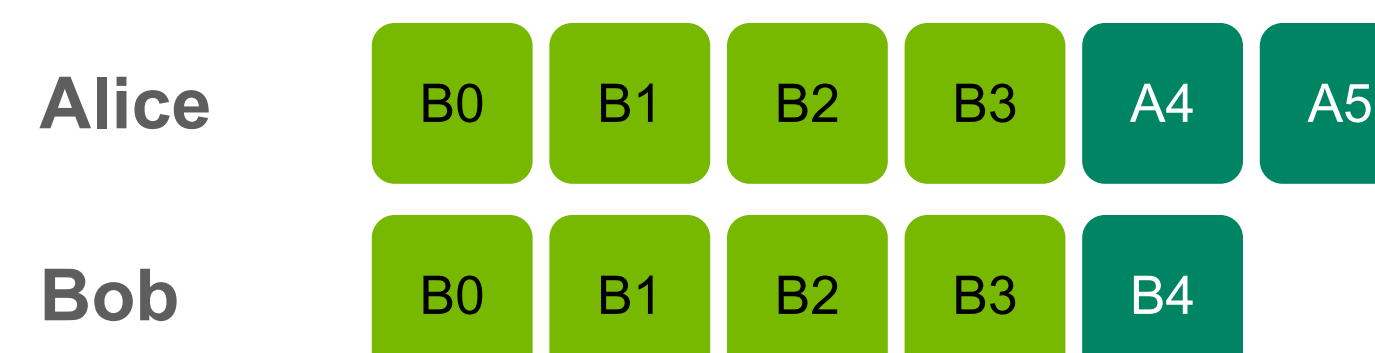
- **One API, any engine.** Clients speak OpenAI; vLLM, SGLang and TensorRT-LLM workers sit behind it.
- **Workers come and go.** The frontend discovers them through etcd or Kubernetes, so clients never have to know where the GPUs are.
- **Tokenize once, centrally.** The same prompt prefix always becomes the same token blocks. That is what makes KV-aware routing possible.



01

Frontend

OpenAI/Anthropic API ·
tokenize · stream



Same system prompt → identical token blocks **B0–B3** → one cached KV prefix serves both users.

Frontend: Following One Request

Discover → accept → template → tokenize → route → stream

- 1 **Discover.** Workers register their model and endpoint. The frontend watches discovery and loads the tokenizer and chat template, never the weights.
- 2 **Accept.** An OpenAI-style request arrives at /v1/chat/completions.
- 3 **Template.** The chat messages are rendered into one prompt string with the model's chat template.
- 4 **Tokenize.** The prompt becomes token IDs, which the router reads in fixed-size KV blocks.
- 5 **Route.** The router picks a worker; token IDs travel over the request plane (TCP or NATS).
- 6 **Stream back.** Output token IDs are detokenized incrementally and streamed as server-sent events.



WHAT THE FRONTEND SEES

Advance to follow one request from discovery to the streamed answer. The user message is “Explain the KV cache in one sentence.”

01

Frontend

OpenAI/Anthropic API
· tokenize · stream

Frontend: Following One Request

Discover → accept → template → tokenize → route → stream

- 1 Discover.** Workers register their model and endpoint. The frontend watches discovery and loads the tokenizer and chat template, never the weights.
- 2 Accept.** An OpenAI-style request arrives at /v1/chat/completions.
- 3 Template.** The chat messages are rendered into one prompt string with the model's chat template.
- 4 Tokenize.** The prompt becomes token IDs, which the router reads in fixed-size KV blocks.
- 5 Route.** The router picks a worker; token IDs travel over the request plane (TCP or NATS).
- 6 Stream back.** Output token IDs are detokenized incrementally and streamed as server-sent events.

Discover

→ Accept

→ Template

→ Tokenize

→ Route

→ Stream

BEFORE ANY REQUEST: DISCOVERY

```
worker-3 ▶ register model=Llama-3.1-70B-Instruct
           endpoint=dynamo.backend.generate
discovery ▶ store  model card + endpoint
           (etcd, or Kubernetes EndpointSlices)
frontend ▶ watch   new model appeared
           load tokenizer.json, tokenizer_config.json,
           config.json # the weights stay on the worker
```

01

Frontend

OpenAI/Anthropic API
· tokenize · stream

Frontend: Following One Request

Discover → accept → template → tokenize → route → stream



HTTP REQUEST, PORT 8000

```
POST /v1/chat/completions
{
  "model": "Llama-3.1-70B-Instruct",
  "stream": true,
  "messages": [
    {"role": "system", "content": "You are a helpful assistant."},
    {"role": "user", "content": "Explain the KV cache in one sentence."}
  ]
}
```

1 **Discover.** Workers register their model and endpoint. The frontend watches discovery and loads the tokenizer and chat template, never the weights.

2 **Accept.** An OpenAI-style request arrives at /v1/chat/completions.

3 **Template.** The chat messages are rendered into one prompt string with the model's chat template.

4 **Tokenize.** The prompt becomes token IDs, which the router reads in fixed-size KV blocks.

5 **Route.** The router picks a worker; token IDs travel over the request plane (TCP or NATS).

6 **Stream back.** Output token IDs are detokenized incrementally and streamed as server-sent events.

01

Frontend

OpenAI/Anthropic API
· tokenize · stream

Frontend: Following One Request

Discover → accept → template → tokenize → route → stream



AFTER THE CHAT TEMPLATE (LLAMA 3 FORMAT)

<|begin_of_text|><|start_header_id|>system<|end_header_id|>

You are a helpful assistant.<|eot_id|><|start_header_id|>user<|end_header_id|>

Explain the KV cache in one sentence.<|eot_id|><|start_header_id|>assistant<|end_header_id|>

Green: special tokens the template inserts. Blue: the user's words. The same messages render differently for other model families.

1 **Discover.** Workers register their model and endpoint. The frontend watches discovery and loads the tokenizer and chat template, never the weights.

2 **Accept.** An OpenAI-style request arrives at /v1/chat/completions.

3 **Template.** The chat messages are rendered into one prompt string with the model's chat template.

4 **Tokenize.** The prompt becomes token IDs, which the router reads in fixed-size KV blocks.

5 **Route.** The router picks a worker; token IDs travel over the request plane (TCP or NATS).

6 **Stream back.** Output token IDs are detokenized incrementally and streamed as server-sent events.

01

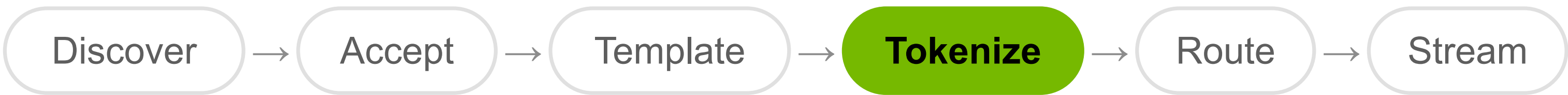
Frontend

OpenAI/Anthropic API
· tokenize · stream

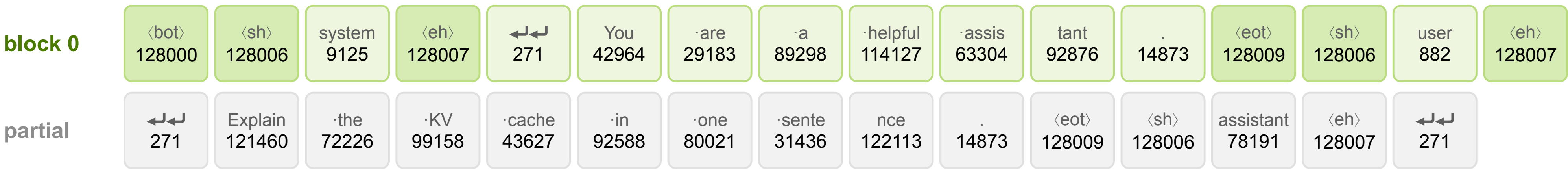
Frontend: Following One Request

Discover → accept → template → tokenize → route → stream

- 1 **Discover.** Workers register their model and endpoint. The frontend watches discovery and loads the tokenizer and chat template, never the weights.
- 2 **Accept.** An OpenAI-style request arrives at /v1/chat/completions.
- 3 **Template.** The chat messages are rendered into one prompt string with the model's chat template.
- 4 **Tokenize.** The prompt becomes token IDs, which the router reads in fixed-size KV blocks.
- 5 **Route.** The router picks a worker; token IDs travel over the request plane (TCP or NATS).
- 6 **Stream back.** Output token IDs are detokenized incrementally and streamed as server-sent events.



31 TOKEN IDS → 1 FULL KV BLOCK OF 16 + 15 IN A PARTIAL BLOCK



Special-token IDs are Llama 3's; other IDs are illustrative. Block size comes from the engine (16 and 64 are common). Only full blocks are reusable cache entries.

01

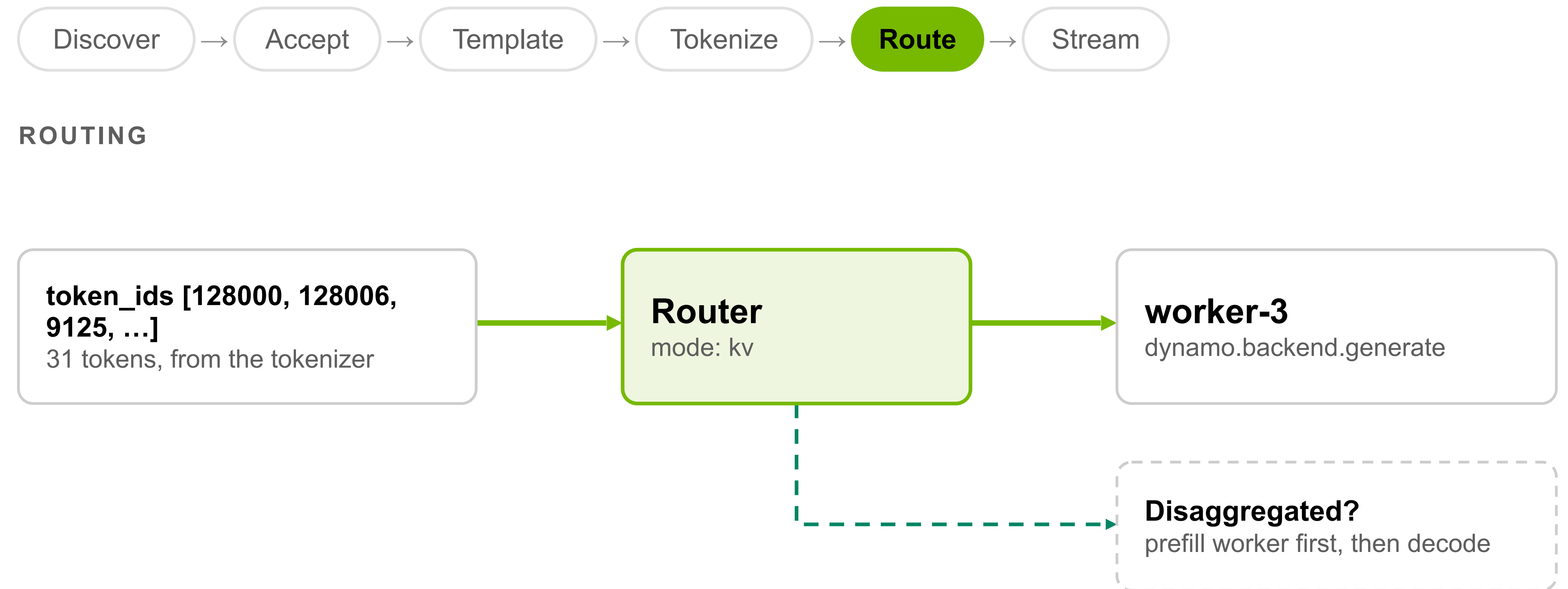
Frontend

OpenAI/Anthropic API
· tokenize · stream

Frontend: Following One Request

Discover → accept → template → tokenize → route → stream

- 1 Discover.** Workers register their model and endpoint. The frontend watches discovery and loads the tokenizer and chat template, never the weights.
- 2 Accept.** An OpenAI-style request arrives at /v1/chat/completions.
- 3 Template.** The chat messages are rendered into one prompt string with the model's chat template.
- 4 Tokenize.** The prompt becomes token IDs, which the router reads in fixed-size KV blocks.
- 5 Route.** The router picks a worker; token IDs travel over the request plane (TCP or NATS).
- 6 Stream back.** Output token IDs are detokenized incrementally and streamed as server-sent events.



Request plane: TCP by default (NATS optional) · payload encoded as msgpack or JSON

01

Frontend

OpenAI/Anthropic API
· tokenize · stream

Frontend: Following One Request

Discover → accept → template → tokenize → route → stream



SERVER-SENT EVENTS BACK TO THE CLIENT

```
data: {"choices":[{"delta":{"content":"reuses "}}]}
data: {"choices":[{"delta":{"content":"them "}}]}
data: {"choices":[{"delta":{"content":"instead "}}]}
data: {"choices":[{"delta":{"content":"of "}}]}
data: {"choices":[{"delta":{"content":"recomputing "}}]}
data: {"choices":[{"delta":{"content":"the "}}]}
data: {"choices":[{"delta":{"content":"whole "}}]}
data: {"choices":[{"delta":{"content":"prefix."}}]}
data: [DONE]
```

The KV cache stores the attention keys and values of tokens already processed, so each new token reuses them instead of recomputing the whole prefix.

01

Frontend

OpenAI/Anthropic API
· tokenize · stream

KV-Aware Router

Send each request to where its KV cache already lives

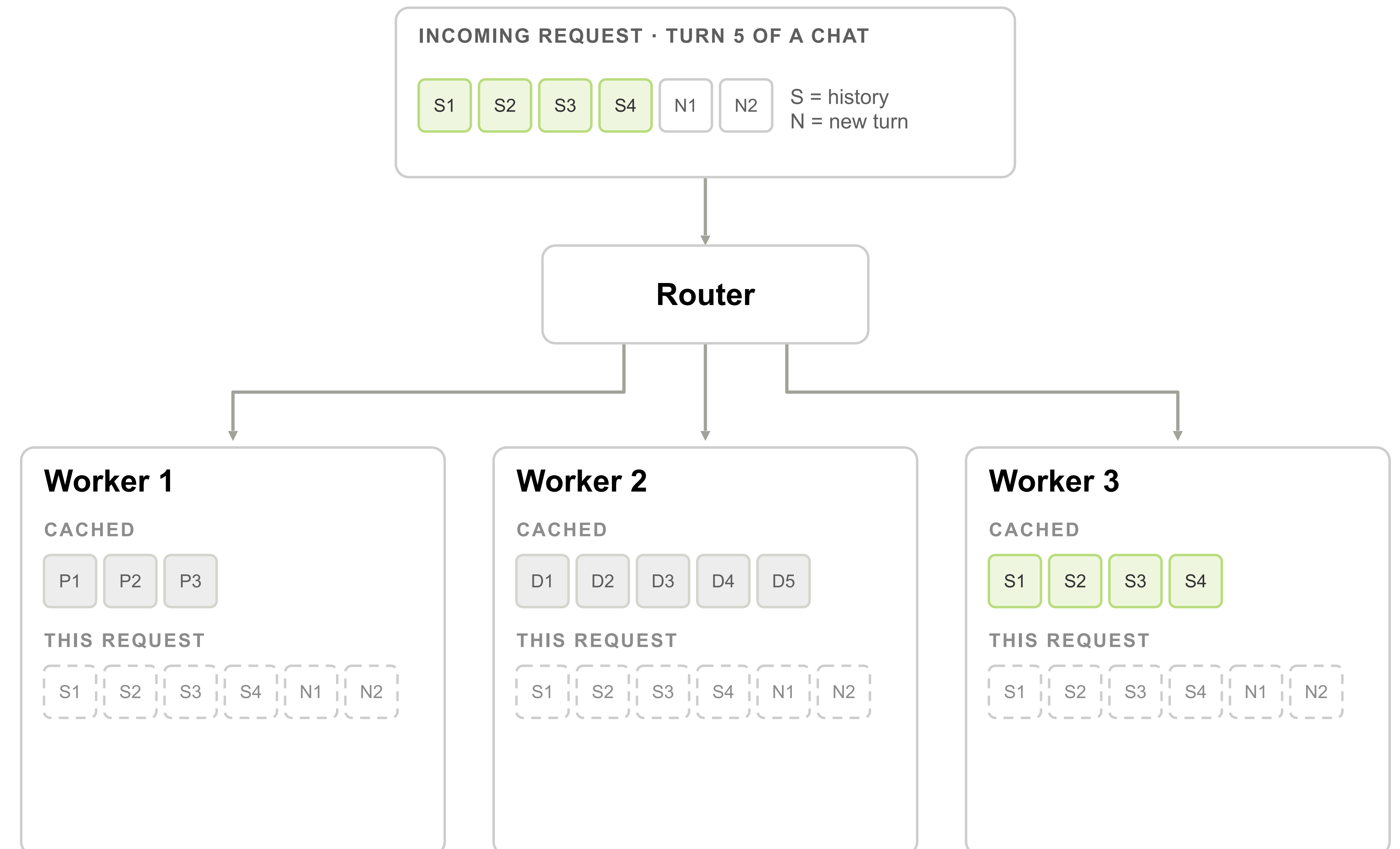
The KV-aware router picks a worker for each request by weighing how much of the prompt that worker **already has cached** against how **busy** it is.

WHY IT MATTERS

- **Prefill is the expensive part.** If the prompt's prefix is cached on worker 3 but the request lands on worker 1, worker 1 recomputes all of it: wasted GPU time and a slower first token.
- **Reuse is everywhere.** Multi-turn chat, agents that resend their whole history, shared system prompts, RAG over the same documents.

3× faster TTFT · 2× lower latency

KV-aware vs. random routing. R1-Distill-Llama-70B FP8 on 2 H100 nodes, 100K requests, ~4K input / 800 output tokens. Source: Dynamo architecture docs.



Advance to route the request: first round-robin, then KV-aware.

prefill work

02

Router

Pick a worker: cache vs. load

KV-Aware Router

Send each request to where its KV cache already lives

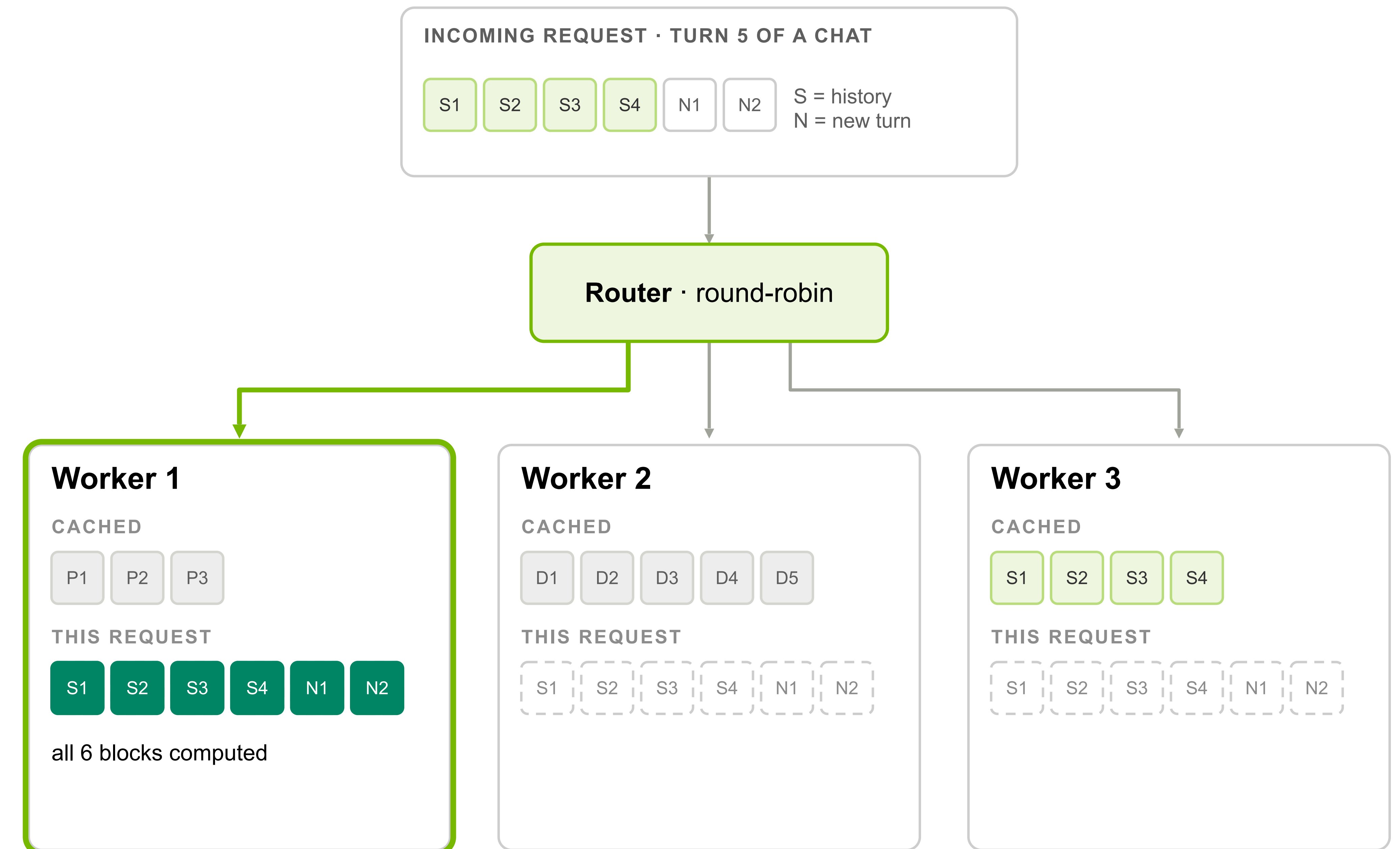
The KV-aware router picks a worker for each request by weighing how much of the prompt that worker **already has cached** against how **busy** it is.

WHY IT MATTERS

- **Prefill is the expensive part.** If the prompt's prefix is cached on worker 3 but the request lands on worker 1, worker 1 recomputes all of it: wasted GPU time and a slower first token.
- **Reuse is everywhere.** Multi-turn chat, agents that resend their whole history, shared system prompts, RAG over the same documents.

3× faster TTFT · 2× lower latency

KV-aware vs. random routing. R1-Distill-Llama-70B FP8 on 2 H100 nodes, 100K requests, ~4K input / 800 output tokens. Source: Dynamo architecture docs.



Round-robin → worker 1, next in rotation. Nothing useful cached: **6 blocks** of prefill.

prefill work

02

Router

Pick a worker: cache vs. load

KV-Aware Router

Send each request to where its KV cache already lives

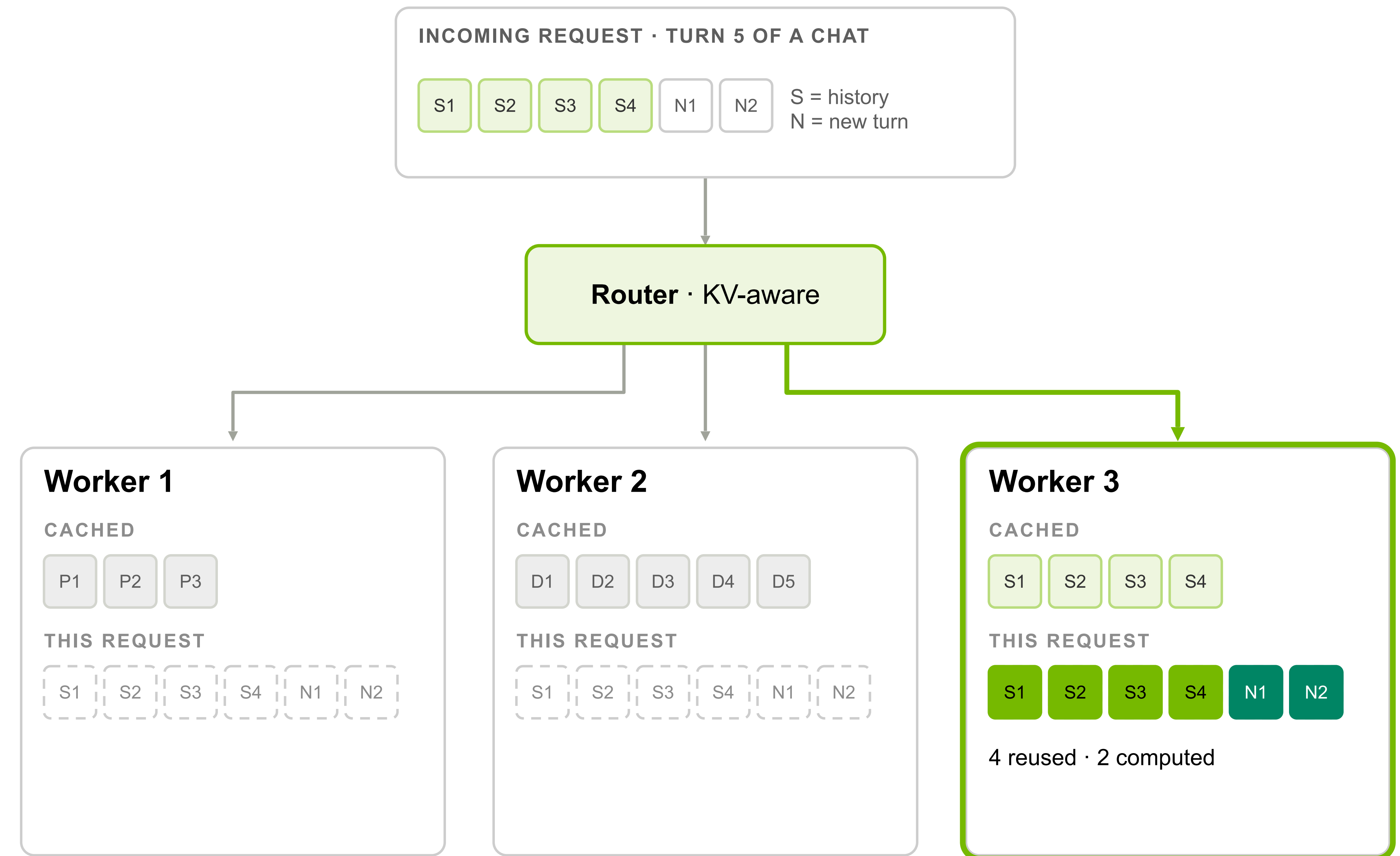
The KV-aware router picks a worker for each request by weighing how much of the prompt that worker **already has cached** against how **busy** it is.

WHY IT MATTERS

- **Prefill is the expensive part.** If the prompt's prefix is cached on worker 3 but the request lands on worker 1, worker 1 recomputes all of it: wasted GPU time and a slower first token.
- **Reuse is everywhere.** Multi-turn chat, agents that resend their whole history, shared system prompts, RAG over the same documents.

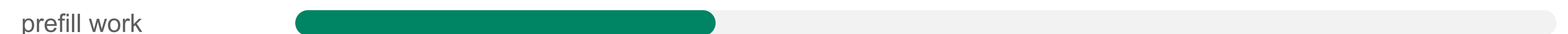
3× faster TTFT · 2× lower latency

KV-aware vs. random routing. R1-Distill-Llama-70B FP8 on 2 H100 nodes, 100K requests, ~4K input / 800 output tokens. Source: Dynamo architecture docs.



KV-aware → worker 3, which holds S1–S4. Only **2 blocks** of prefill.

prefill work



02

Router

Pick a worker: cache vs. load

Router: How It Scores Workers

cost $\approx$ blocks still to prefill + active decode blocks; lowest cost wins

- 1 **Split and hash.** Token IDs are cut into fixed-size blocks; each block's hash also covers everything before it.
- 2 **Look up.** A radix tree of block hashes, kept current by KV events from every worker, gives each worker's longest cached prefix.
- 3 **Check load.** The router tracks each worker's in-flight prefill and decode work.
- 4 **Score.** Cost $\approx$ blocks still to prefill + active decode blocks. The lowest cost wins.
- 5 **Learn.** The chosen worker publishes stored and removed KV events, so the next request sees the new cache.

Simplified. The documented cost also scales the prefill term, credits host- and disk-tier hits at lower weight, and can sample with a temperature instead of always taking the minimum.

cost = (10 - credit $\times$ cached) + decode load

W1 cached 2
 decode load 10

W2 cached 5
 decode load 5

W3 cached 8
 decode load 9

Overlap credit = 1.0 here. At 0 the router ignores the cache and balances load only. What if W2's decode load rises to 7? Its cost becomes 12 and **W3 wins**. (Worked example from the router design docs.)

02

Router

Pick a worker: cache
vs. load

Router: How It Scores Workers

cost ≈ blocks still to prefill + active decode blocks; lowest cost wins

- 1

Split and hash. Token IDs are cut into fixed-size blocks; each block's hash also covers everything before it.
- 2

Look up. A radix tree of block hashes, kept current by KV events from every worker, gives each worker's longest cached prefix.
- 3

Check load. The router tracks each worker's in-flight prefill and decode work.
- 4

Score. Cost ≈ blocks still to prefill + active decode blocks. The lowest cost wins.
- 5

Learn. The chosen worker publishes stored and removed KV events, so the next request sees the new cache.

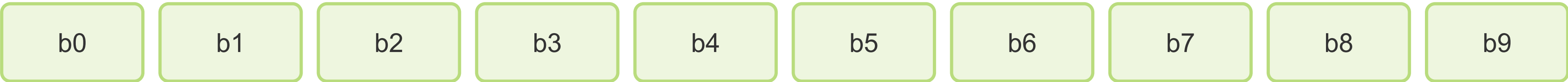
Simplified. The documented cost also scales the prefill term, credits host- and disk-tier hits at lower weight, and can sample with a temperature instead of always taking the minimum.

02

Router

Pick a worker: cache vs. load

PROMPT → 10 BLOCKS · $H_i = \text{HASH}(H_{i-1}, \text{TOKENS OF BLOCK } i)$



$$\text{cost} = (10 - \text{credit} \times \text{cached}) + \text{decode load}$$

W1	cached 2 decode load 10
W2	cached 5 decode load 5
W3	cached 8 decode load 9

Overlap credit = 1.0 here. At 0 the router ignores the cache and balances load only. What if W2's decode load rises to 7? Its cost becomes 12 and **W3 wins**. (Worked example from the router design docs.)

Router: How It Scores Workers

cost $\approx$ blocks still to prefill + active decode blocks; lowest cost wins

- 1

Split and hash. Token IDs are cut into fixed-size blocks; each block's hash also covers everything before it.
- 2

Look up. A radix tree of block hashes, kept current by KV events from every worker, gives each worker's longest cached prefix.
- 3

Check load. The router tracks each worker's in-flight prefill and decode work.
- 4

Score. Cost $\approx$ blocks still to prefill + active decode blocks. The lowest cost wins.
- 5

Learn. The chosen worker publishes stored and removed KV events, so the next request sees the new cache.

Simplified. The documented cost also scales the prefill term, credits host- and disk-tier hits at lower weight, and can sample with a temperature instead of always taking the minimum.

PROMPT $\rightarrow$ 10 BLOCKS $\cdot H_i = \text{HASH}(H_{i-1}, \text{TOKENS OF BLOCK } i)$



RADIX-TREE LOOKUP $\rightarrow$ LONGEST CACHED PREFIX ON EACH WORKER



$$\text{cost} = (10 - \text{credit} \times \text{cached}) + \text{decode load}$$

W1	cached 2 decode load 10
W2	cached 5 decode load 5
W3	cached 8 decode load 9

Overlap credit = 1.0 here. At 0 the router ignores the cache and balances load only. What if W2's decode load rises to 7? Its cost becomes 12 and **W3 wins**. (Worked example from the router design docs.)

02

Router

Pick a worker: cache
vs. load

Router: How It Scores Workers

cost $\approx$ blocks still to prefill + active decode blocks; lowest cost wins

- 1

Split and hash. Token IDs are cut into fixed-size blocks; each block's hash also covers everything before it.
- 2

Look up. A radix tree of block hashes, kept current by KV events from every worker, gives each worker's longest cached prefix.
- 3

Check load. The router tracks each worker's in-flight prefill and decode work.
- 4

Score. Cost $\approx$ blocks still to prefill + active decode blocks. The lowest cost wins.
- 5

Learn. The chosen worker publishes stored and removed KV events, so the next request sees the new cache.

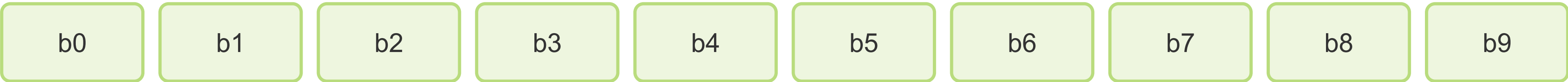
Simplified. The documented cost also scales the prefill term, credits host- and disk-tier hits at lower weight, and can sample with a temperature instead of always taking the minimum.

02

Router

Pick a worker: cache vs. load

PROMPT $\rightarrow$ 10 BLOCKS $\cdot H_i = \text{HASH}(H_{i-1}, \text{TOKENS OF BLOCK } i)$



RADIX-TREE LOOKUP $\rightarrow$ LONGEST CACHED PREFIX ON EACH WORKER



cost = (10 - credit $\times$ cached) + decode load



Overlap credit = 1.0 here. At 0 the router ignores the cache and balances load only. What if W2's decode load rises to 7? Its cost becomes 12 and **W3 wins**. (Worked example from the router design docs.)

Router: How It Scores Workers

cost $\approx$ blocks still to prefill + active decode blocks; lowest cost wins

- 1

Split and hash. Token IDs are cut into fixed-size blocks; each block's hash also covers everything before it.
- 2

Look up. A radix tree of block hashes, kept current by KV events from every worker, gives each worker's longest cached prefix.
- 3

Check load. The router tracks each worker's in-flight prefill and decode work.
- 4

Score. Cost $\approx$ blocks still to prefill + active decode blocks. The lowest cost wins.
- 5

Learn. The chosen worker publishes stored and removed KV events, so the next request sees the new cache.

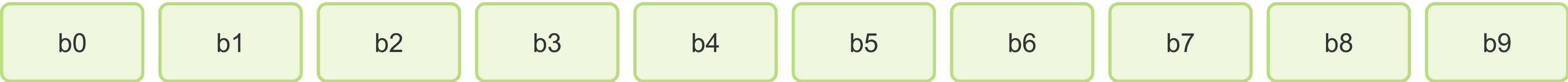
Simplified. The documented cost also scales the prefill term, credits host- and disk-tier hits at lower weight, and can sample with a temperature instead of always taking the minimum.

02

Router

Pick a worker: cache vs. load

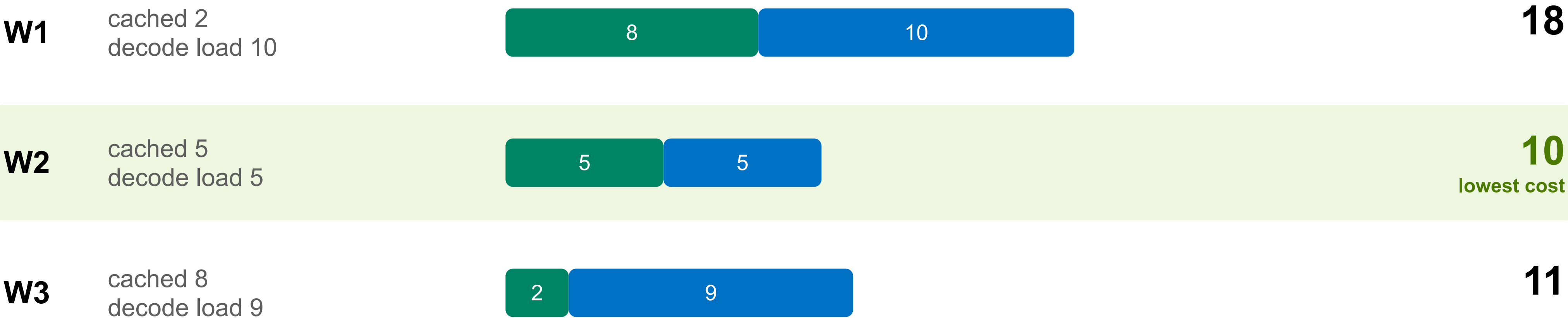
PROMPT $\rightarrow$ 10 BLOCKS $\cdot H_i = \text{HASH}(H_{i-1}, \text{TOKENS OF BLOCK } i)$



RADIX-TREE LOOKUP $\rightarrow$ LONGEST CACHED PREFIX ON EACH WORKER



cost = (10 - credit $\times$ cached) + decode load



Overlap credit = 1.0 here. At 0 the router ignores the cache and balances load only. What if W2's decode load rises to 7? Its cost becomes 12 and **W3 wins**. (Worked example from the router design docs.)

Router: How It Scores Workers

cost ≈ blocks still to prefill + active decode blocks; lowest cost wins

- 1

Split and hash. Token IDs are cut into fixed-size blocks; each block's hash also covers everything before it.
- 2

Look up. A radix tree of block hashes, kept current by KV events from every worker, gives each worker's longest cached prefix.
- 3

Check load. The router tracks each worker's in-flight prefill and decode work.
- 4

Score. Cost ≈ blocks still to prefill + active decode blocks. The lowest cost wins.
- 5

Learn. The chosen worker publishes stored and removed KV events, so the next request sees the new cache.

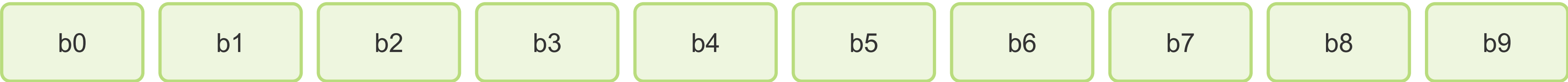
Simplified. The documented cost also scales the prefill term, credits host- and disk-tier hits at lower weight, and can sample with a temperature instead of always taking the minimum.

02

Router

Pick a worker: cache vs. load

PROMPT → 10 BLOCKS · $H_i = \text{HASH}(H_{i-1}, \text{TOKENS OF BLOCK } i)$

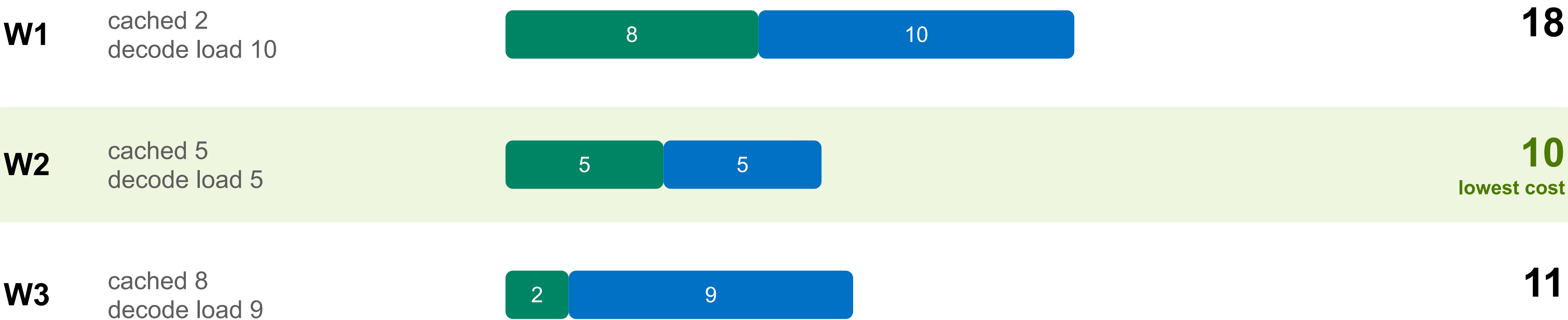


RADIX-TREE LOOKUP → LONGEST CACHED PREFIX ON EACH WORKER



KV event from W2: stored b0–b9 → the next identical prompt matches all 10 blocks there.

cost = (10 – credit × cached) + decode load



Overlap credit = 1.0 here. At 0 the router ignores the cache and balances load only. What if W2's decode load rises to 7? Its cost becomes 12 and **W3 wins**. (Worked example from the router design docs.)

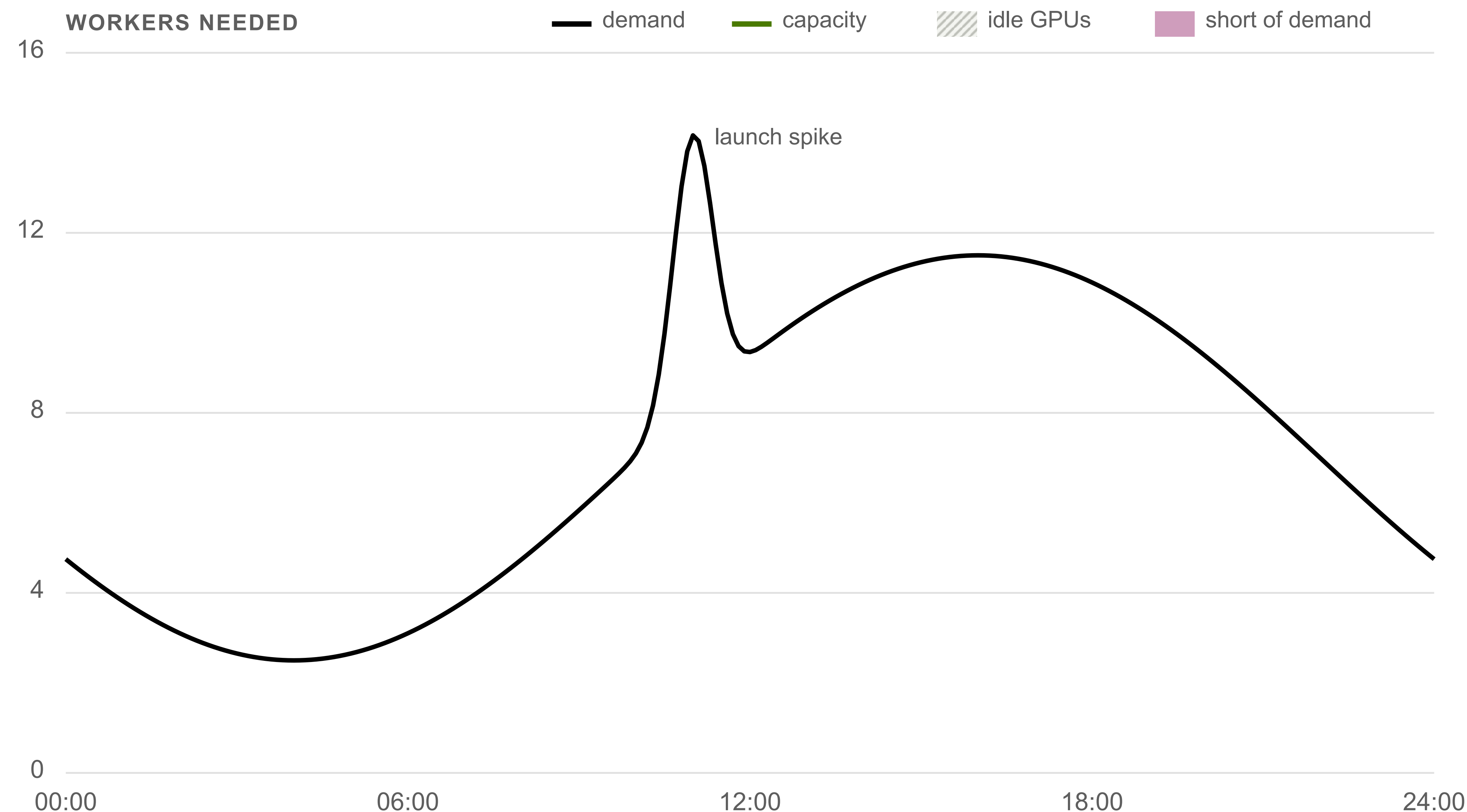
Planner

Traffic moves; GPU capacity has to follow

An SLA-driven autoscaler. It decides how many **prefill** and how many **decode** workers to run, aiming at time-to-first-token (TTFT) and inter-token latency (ITL) targets.

WHY GENERIC AUTOSCALERS FALL SHORT

- **Cost depends on content.** One 32K-token prompt is not one short chat.
- **Prefill and decode scale differently.** Prefill grows with input length; decode grows with concurrent sequences and KV memory.
- **The SLA is TTFT and ITL**, not CPU utilization or request count.
- **GPU workers take minutes to start**, so the planner has to predict demand, not only react to it.



Illustrative 24-hour demand with a launch spike at 11:00. Capacity in worker-units.

03

Planner

How many prefill and decode workers?

Planner

Traffic moves; GPU capacity has to follow

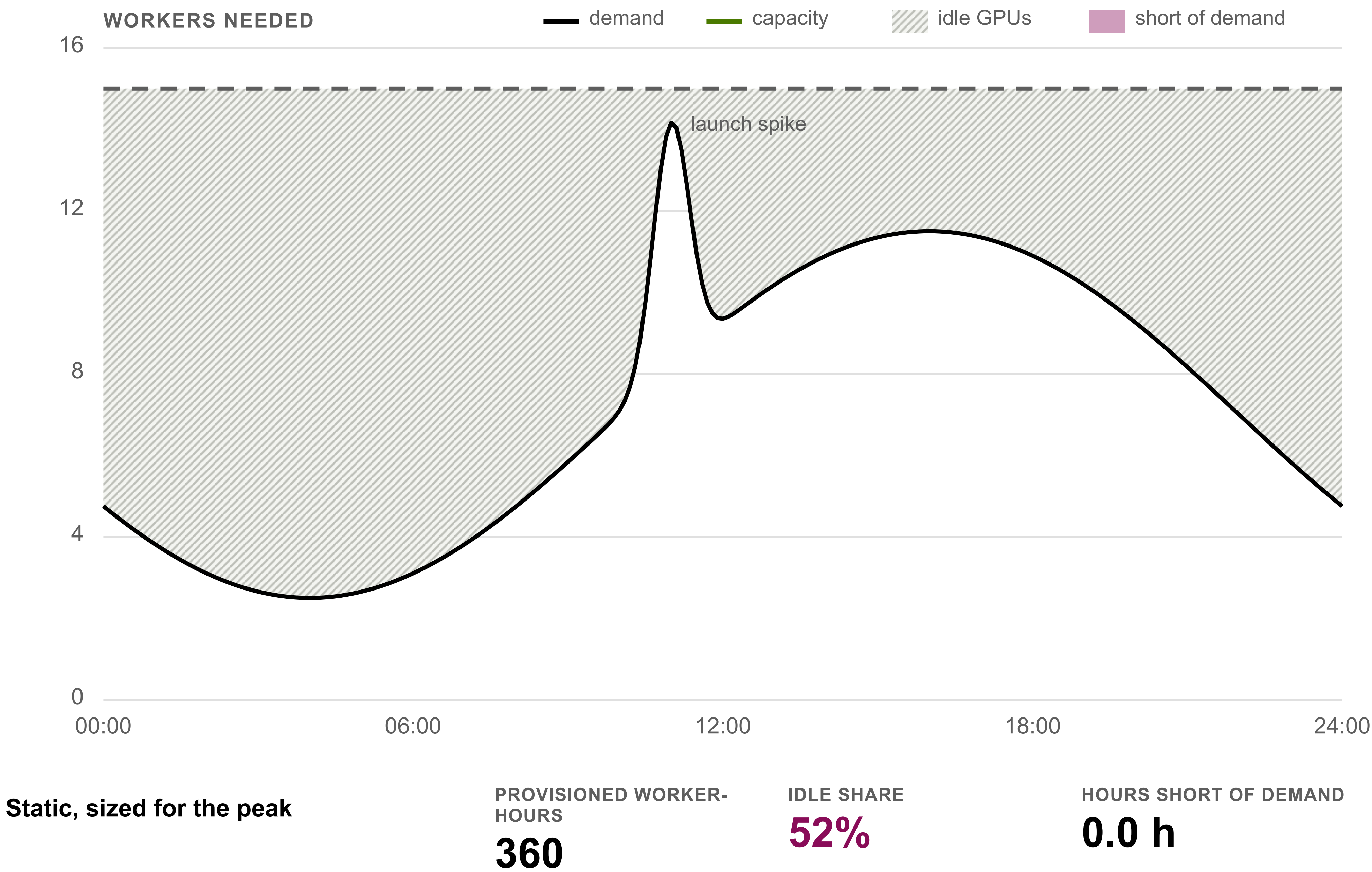
An SLA-driven autoscaler. It decides how many **prefill** and how many **decode** workers to run, aiming at time-to-first-token (TTFT) and inter-token latency (ITL) targets.

WHY GENERIC AUTOSCALERS FALL SHORT

- **Cost depends on content.** One 32K-token prompt is not one short chat.
- **Prefill and decode scale differently.** Prefill grows with input length; decode grows with concurrent sequences and KV memory.
- **The SLA is TTFT and ITL**, not CPU utilization or request count.
- **GPU workers take minutes to start**, so the planner has to predict demand, not only react to it.

03 Planner

How many prefill and decode workers?



Illustrative 24-hour demand with a launch spike at 11:00. Capacity in worker-units.

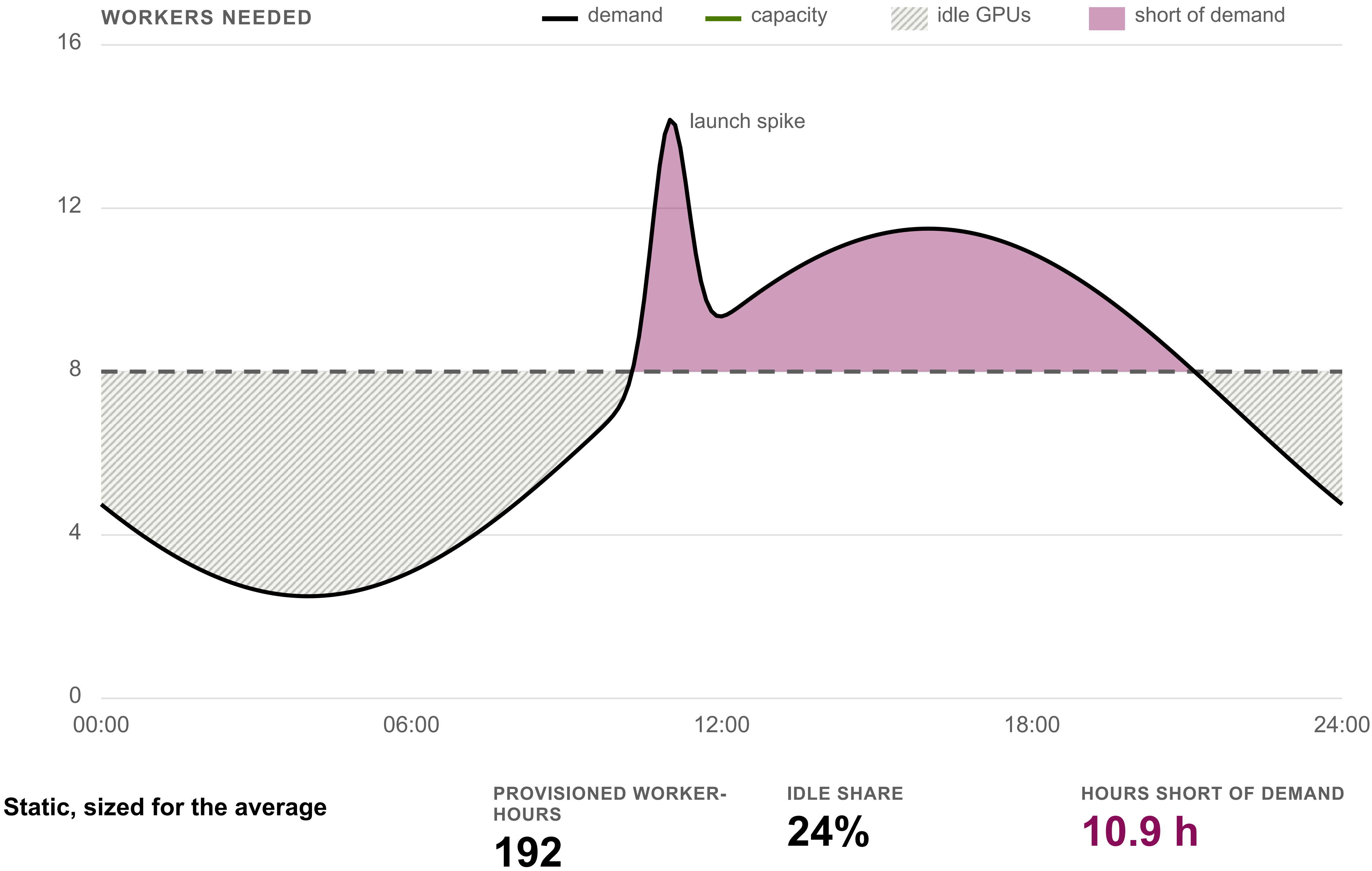
Planner

Traffic moves; GPU capacity has to follow

An SLA-driven autoscaler. It decides how many **prefill** and how many **decode** workers to run, aiming at time-to-first-token (TTFT) and inter-token latency (ITL) targets.

WHY GENERIC AUTOSCALERS FALL SHORT

- **Cost depends on content.** One 32K-token prompt is not one short chat.
- **Prefill and decode scale differently.** Prefill grows with input length; decode grows with concurrent sequences and KV memory.
- **The SLA is TTFT and ITL**, not CPU utilization or request count.
- **GPU workers take minutes to start**, so the planner has to predict demand, not only react to it.



Illustrative 24-hour demand with a launch spike at 11:00. Capacity in worker-units.

03

Planner

How many prefill and decode workers?

Planner

Traffic moves; GPU capacity has to follow

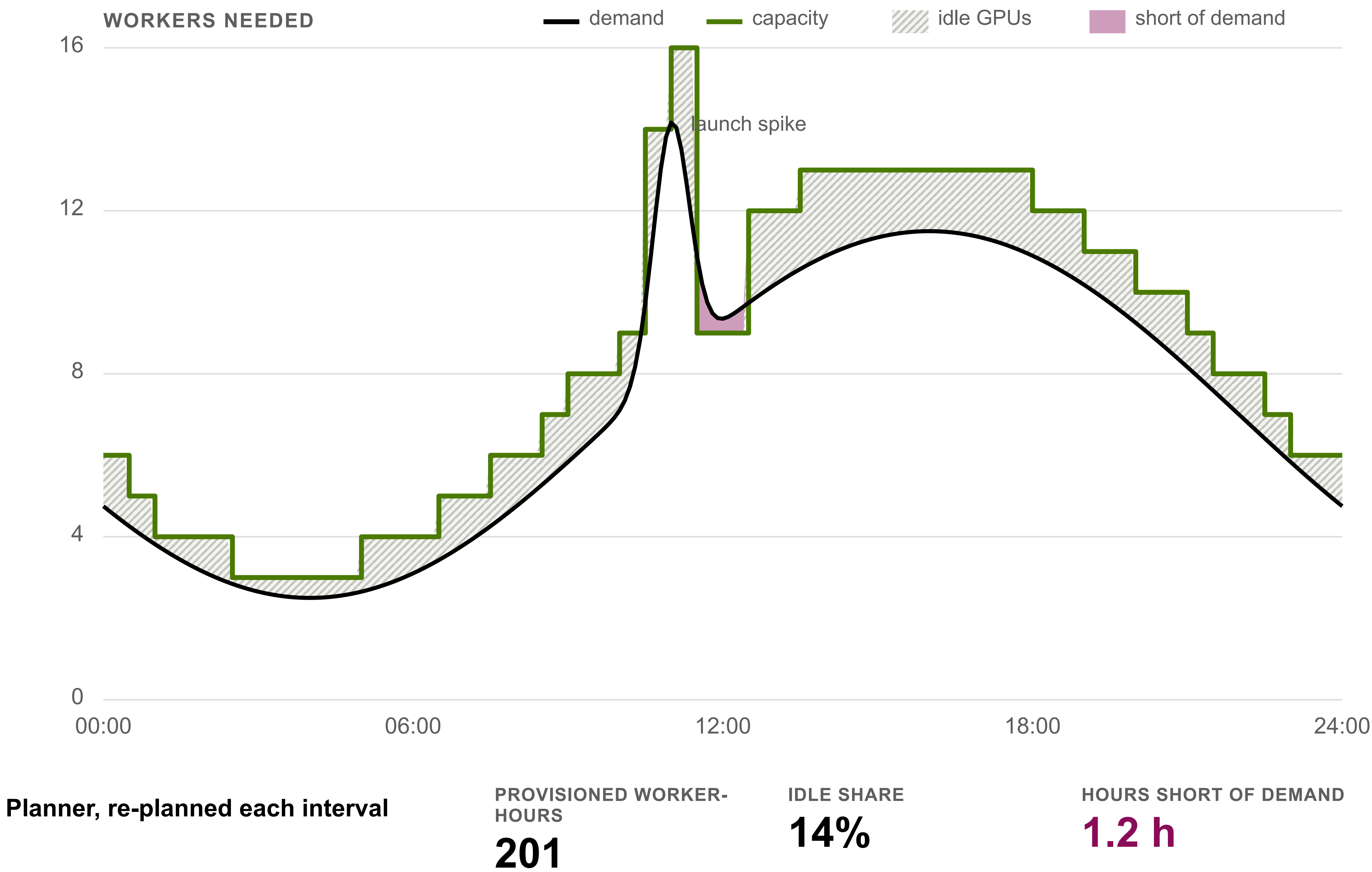
An SLA-driven autoscaler. It decides how many **prefill** and how many **decode** workers to run, aiming at time-to-first-token (TTFT) and inter-token latency (ITL) targets.

WHY GENERIC AUTOSCALERS FALL SHORT

- **Cost depends on content.** One 32K-token prompt is not one short chat.
- **Prefill and decode scale differently.** Prefill grows with input length; decode grows with concurrent sequences and KV memory.
- **The SLA is TTFT and ITL**, not CPU utilization or request count.
- **GPU workers take minutes to start**, so the planner has to predict demand, not only react to it.

03 Planner

How many prefill and decode workers?

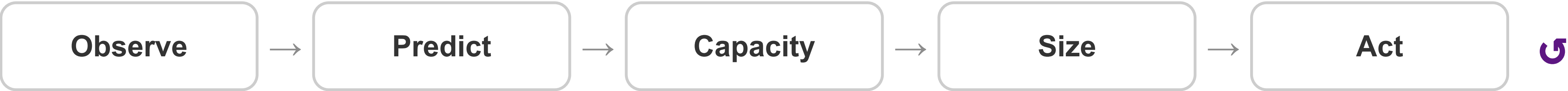


Illustrative 24-hour demand with a launch spike at 11:00. Capacity in worker-units.

Planner: The Control Loop

Observe → predict → look up capacity → size → act

- 1 **Observe.** Request rate, average input and output length (ISL, OSL) and KV hit rate from the frontend and router, plus per-iteration engine metrics.
- 2 **Predict** the next interval's load (constant, ARIMA, Kalman or Prophet predictors).
- 3 **Look up capacity.** A profiled performance model gives the tokens/s one prefill engine sustains within the TTFT target, and one decode engine within the ITL target.
- 4 **Size** each pool: predicted load ÷ per-engine capacity, rounded up.
- 5 **Act.** Clamp to the GPU budget and minimums; a connector patches the Kubernetes deployment or tells your own orchestrator.



SIZING EXAMPLES · 8 REQUESTS/S

A. Baseline: 4K-token prompts, 1K-token answers

prefill $\lceil 8 \times 4,096 \div 40,000 \rceil = 1$ decode $\lceil 8 \times 1,024 \div 4,000 \rceil = 3$

prefill  decode 

B. 4× longer prompts (16K in, 1K out) → prefill grows, decode unchanged

prefill $\lceil 8 \times 16,384 \div 40,000 \rceil = 4$ decode $\lceil 8 \times 1,024 \div 4,000 \rceil = 3$

prefill  decode 

C. 4× longer answers (4K in, 4K out) → decode grows, prefill unchanged

prefill $\lceil 8 \times 4,096 \div 40,000 \rceil = 1$ decode $\lceil 8 \times 4,096 \div 4,000 \rceil = 9$

prefill  decode 

Illustrative capacities: one prefill engine handles 40,000 input tok/s within TTFT; one decode engine handles 4,000 output tok/s within ITL. Real values come from profiling.

Two loops. Throughput-based scaling predicts (every 180 s by default) and sets the floor. Load-based scaling reacts (every 5 s), adding or removing one engine when estimated latency crosses the SLA.

03

Planner

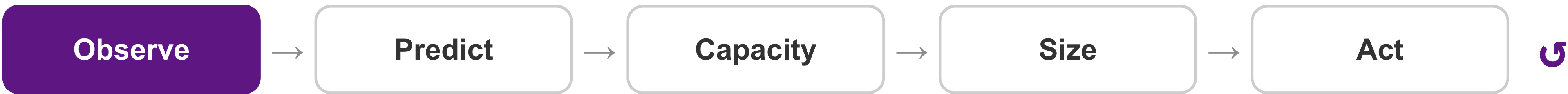
How many prefill and decode workers?

Planner: The Control Loop

Observe → predict → look up capacity → size → act

- 1 Observe.** Request rate, average input and output length (ISL, OSL) and KV hit rate from the frontend and router, plus per-iteration engine metrics.
- 2 Predict** the next interval's load (constant, ARIMA, Kalman or Prophet predictors).
- 3 Look up capacity.** A profiled performance model gives the tokens/s one prefill engine sustains within the TTFT target, and one decode engine within the ITL target.
- 4 Size** each pool: predicted load ÷ per-engine capacity, rounded up.
- 5 Act.** Clamp to the GPU budget and minimums; a connector patches the Kubernetes deployment or tells your own orchestrator.

Two loops. Throughput-based scaling predicts (every 180 s by default) and sets the floor. Load-based scaling reacts (every 5 s), adding or removing one engine when estimated latency crosses the SLA.



SIZING EXAMPLES · 8 REQUESTS/S

A. Baseline: 4K-token prompts, 1K-token answers

prefill $[8 \times 4,096 \div 40,000] = 1$ decode $[8 \times 1,024 \div 4,000] = 3$

prefill  decode 

B. 4× longer prompts (16K in, 1K out) → prefill grows, decode unchanged

prefill $[8 \times 16,384 \div 40,000] = 4$ decode $[8 \times 1,024 \div 4,000] = 3$

prefill  decode 

C. 4× longer answers (4K in, 4K out) → decode grows, prefill unchanged

prefill $[8 \times 4,096 \div 40,000] = 1$ decode $[8 \times 4,096 \div 4,000] = 9$

prefill  decode 

Illustrative capacities: one prefill engine handles 40,000 input tok/s within TTFT; one decode engine handles 4,000 output tok/s within ITL. Real values come from profiling.

03

Planner

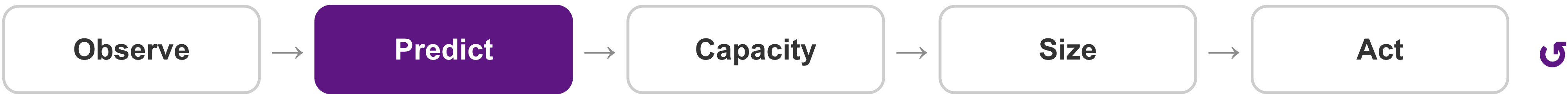
How many prefill and decode workers?

Planner: The Control Loop

Observe → predict → look up capacity → size → act

- 1 Observe.** Request rate, average input and output length (ISL, OSL) and KV hit rate from the frontend and router, plus per-iteration engine metrics.
- 2 Predict** the next interval's load (constant, ARIMA, Kalman or Prophet predictors).
- 3 Look up capacity.** A profiled performance model gives the tokens/s one prefill engine sustains within the TTFT target, and one decode engine within the ITL target.
- 4 Size** each pool: predicted load ÷ per-engine capacity, rounded up.
- 5 Act.** Clamp to the GPU budget and minimums; a connector patches the Kubernetes deployment or tells your own orchestrator.

Two loops. Throughput-based scaling predicts (every 180 s by default) and sets the floor. Load-based scaling reacts (every 5 s), adding or removing one engine when estimated latency crosses the SLA.



SIZING EXAMPLES · 8 REQUESTS/S

A. Baseline: 4K-token prompts, 1K-token answers

prefill $[8 \times 4,096 \div 40,000] = 1$ decode $[8 \times 1,024 \div 4,000] = 3$

prefill  decode 

B. 4× longer prompts (16K in, 1K out) → prefill grows, decode unchanged

prefill $[8 \times 16,384 \div 40,000] = 4$ decode $[8 \times 1,024 \div 4,000] = 3$

prefill  decode 

C. 4× longer answers (4K in, 4K out) → decode grows, prefill unchanged

prefill $[8 \times 4,096 \div 40,000] = 1$ decode $[8 \times 4,096 \div 4,000] = 9$

prefill  decode 

Illustrative capacities: one prefill engine handles 40,000 input tok/s within TTFT; one decode engine handles 4,000 output tok/s within ITL. Real values come from profiling.

03

Planner

How many prefill and decode workers?

Planner: The Control Loop

Observe → predict → look up capacity → size → act

- 1 **Observe.** Request rate, average input and output length (ISL, OSL) and KV hit rate from the frontend and router, plus per-iteration engine metrics.
- 2 **Predict** the next interval's load (constant, ARIMA, Kalman or Prophet predictors).
- 3 **Look up capacity.** A profiled performance model gives the tokens/s one prefill engine sustains within the TTFT target, and one decode engine within the ITL target.
- 4 **Size** each pool: predicted load ÷ per-engine capacity, rounded up.
- 5 **Act.** Clamp to the GPU budget and minimums; a connector patches the Kubernetes deployment or tells your own orchestrator.



SIZING EXAMPLES · 8 REQUESTS/S

A. Baseline: 4K-token prompts, 1K-token answers

$\text{prefill } [8 \times 4,096 \div 40,000] = 1$ $\text{decode } [8 \times 1,024 \div 4,000] = 3$

prefill  decode 

B. 4× longer prompts (16K in, 1K out) → prefill grows, decode unchanged

$\text{prefill } [8 \times 16,384 \div 40,000] = 4$ $\text{decode } [8 \times 1,024 \div 4,000] = 3$

prefill  decode 

C. 4× longer answers (4K in, 4K out) → decode grows, prefill unchanged

$\text{prefill } [8 \times 4,096 \div 40,000] = 1$ $\text{decode } [8 \times 4,096 \div 4,000] = 9$

prefill  decode 

Illustrative capacities: one prefill engine handles 40,000 input tok/s within TTFT; one decode engine handles 4,000 output tok/s within ITL. Real values come from profiling.

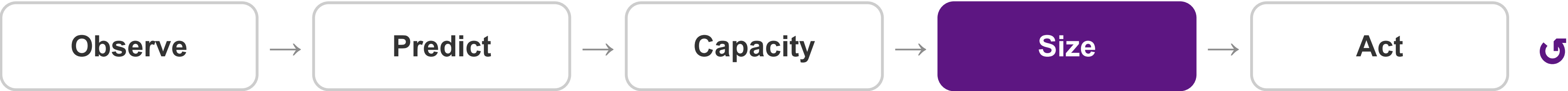
Two loops. Throughput-based scaling predicts (every 180 s by default) and sets the floor. Load-based scaling reacts (every 5 s), adding or removing one engine when estimated latency crosses the SLA.

03
Planner
How many prefill and
decode workers?

Planner: The Control Loop

Observe → predict → look up capacity → size → act

- 1 **Observe.** Request rate, average input and output length (ISL, OSL) and KV hit rate from the frontend and router, plus per-iteration engine metrics.
- 2 **Predict** the next interval's load (constant, ARIMA, Kalman or Prophet predictors).
- 3 **Look up capacity.** A profiled performance model gives the tokens/s one prefill engine sustains within the TTFT target, and one decode engine within the ITL target.
- 4 **Size** each pool: predicted load ÷ per-engine capacity, rounded up.
- 5 **Act.** Clamp to the GPU budget and minimums; a connector patches the Kubernetes deployment or tells your own orchestrator.



SIZING EXAMPLES · 8 REQUESTS/S

A. Baseline: 4K-token prompts, 1K-token answers

prefill $[8 \times 4,096 \div 40,000] = 1$ decode $[8 \times 1,024 \div 4,000] = 3$

prefill  decode 

B. 4× longer prompts (16K in, 1K out) → prefill grows, decode unchanged

prefill $[8 \times 16,384 \div 40,000] = 4$ decode $[8 \times 1,024 \div 4,000] = 3$

prefill  decode 

C. 4× longer answers (4K in, 4K out) → decode grows, prefill unchanged

prefill $[8 \times 4,096 \div 40,000] = 1$ decode $[8 \times 4,096 \div 4,000] = 9$

prefill  decode 

Illustrative capacities: one prefill engine handles 40,000 input tok/s within TTFT; one decode engine handles 4,000 output tok/s within ITL. Real values come from profiling.

Two loops. Throughput-based scaling predicts (every 180 s by default) and sets the floor. Load-based scaling reacts (every 5 s), adding or removing one engine when estimated latency crosses the SLA.

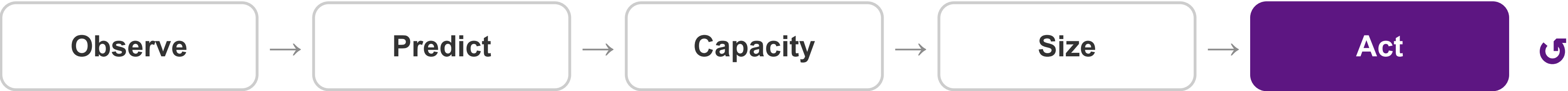
03
Planner
How many prefill and
decode workers?

Planner: The Control Loop

Observe → predict → look up capacity → size → act

- 1 Observe.** Request rate, average input and output length (ISL, OSL) and KV hit rate from the frontend and router, plus per-iteration engine metrics.
- 2 Predict** the next interval's load (constant, ARIMA, Kalman or Prophet predictors).
- 3 Look up capacity.** A profiled performance model gives the tokens/s one prefill engine sustains within the TTFT target, and one decode engine within the ITL target.
- 4 Size** each pool: predicted load ÷ per-engine capacity, rounded up.
- 5 Act.** Clamp to the GPU budget and minimums; a connector patches the Kubernetes deployment or tells your own orchestrator.

Two loops. Throughput-based scaling predicts (every 180 s by default) and sets the floor. Load-based scaling reacts (every 5 s), adding or removing one engine when estimated latency crosses the SLA.



SIZING EXAMPLES · 8 REQUESTS/S

A. Baseline: 4K-token prompts, 1K-token answers

prefill $[8 \times 4,096 \div 40,000] = 1$ decode $[8 \times 1,024 \div 4,000] = 3$

prefill decode

B. 4× longer prompts (16K in, 1K out) → prefill grows, decode unchanged

prefill $[8 \times 16,384 \div 40,000] = 4$ decode $[8 \times 1,024 \div 4,000] = 3$

prefill decode

C. 4× longer answers (4K in, 4K out) → decode grows, prefill unchanged

prefill $[8 \times 4,096 \div 40,000] = 1$ decode $[8 \times 4,096 \div 4,000] = 9$

prefill decode

Illustrative capacities: one prefill engine handles 40,000 input tok/s within TTFT; one decode engine handles 4,000 output tok/s within ITL. Real values come from profiling.

03

Planner

How many prefill and decode workers?

Prefill vs. Decode

Same GPU, opposite bottlenecks, interference avoidance

Prefill

once per request



The whole prompt goes through the model in one parallel pass.

BOTTLENECK Compute: large matrix multiplies

PRODUCES The prompt's KV cache and the first token

SETS Time to first token (TTFT)

PREFERS Less tensor parallelism, so less communication



utilization (illustrative)

Decode

once per output token



One new token per sequence per step; every step rereads the weights and the KV cache.

BOTTLENECK Memory bandwidth

PRODUCES One token per sequence per step

SETS Inter-token latency (ITL)

PREFERS More tensor & expert parallelism



utilization (illustrative)

Prefill vs. Decode

Same GPU, opposite bottlenecks, interference avoidance

Prefill

once per request



The whole prompt goes through the model in one parallel pass.

BOTTLENECK Compute: large matrix multiplies

PRODUCES The prompt's KV cache and the first token

SETS Time to first token (TTFT)

PREFERS Less tensor parallelism, so less communication

Compute 

Memory bandwidth 

utilization (illustrative)

Decode

once per output token



One new token per sequence per step; every step rereads the weights and the KV cache.

BOTTLENECK Memory bandwidth

PRODUCES One token per sequence per step

SETS Inter-token latency (ITL)

PREFERS More tensor & expert parallelism

Compute 

Memory bandwidth 

utilization (illustrative)

Prefill vs. Decode

Same GPU, opposite bottlenecks, interference avoidance

Prefill

once per request



The whole prompt goes through the model in one parallel pass.

BOTTLENECK Compute: large matrix multiplies

PRODUCES The prompt's KV cache and the first token

SETS Time to first token (TTFT)

PREFERS Less tensor parallelism, so less communication

Compute

Memory bandwidth

utilization (illustrative)

Decode

once per output token



One new token per sequence per step; every step rereads the weights and the KV cache.

BOTTLENECK Memory bandwidth

PRODUCES One token per sequence per step

SETS Inter-token latency (ITL)

PREFERS More tensor & expert parallelism

Compute

Memory bandwidth

utilization (illustrative)

Prefill vs. Decode

Same GPU, opposite bottlenecks, interference avoidance

Prefill

once per request



The whole prompt goes through the model in one parallel pass.

BOTTLENECK Compute: large matrix multiplies

PRODUCES The prompt's KV cache and the first token

SETS Time to first token (TTFT)

PREFERS Less tensor parallelism, so less communication

Compute 

Memory bandwidth 

utilization (illustrative)

Decode

once per output token



One new token per sequence per step; every step rereads the weights and the KV cache.

BOTTLENECK Memory bandwidth

PRODUCES One token per sequence per step

SETS Inter-token latency (ITL)

PREFERS More tensor & expert parallelism

Compute 

Memory bandwidth 

utilization (illustrative)

Same GPU, opposite bottlenecks. Run both phases together and a long prefill stalls everyone's decode, and no single parallelism suits both. Disaggregation gives each phase its own pool, sized and parallelized for its job.

Aggregated vs. Disaggregated

What a long prompt does to everyone else

Aggregated. One pool runs both phases. While it works through a long prefill, every other user's decode waits.

Disaggregated. Prefill runs on its own pool. Decode keeps its rhythm, and the new request joins once its KV cache arrives over NIXL.

LONGEST GAP BETWEEN TOKENS, USERS 1-3

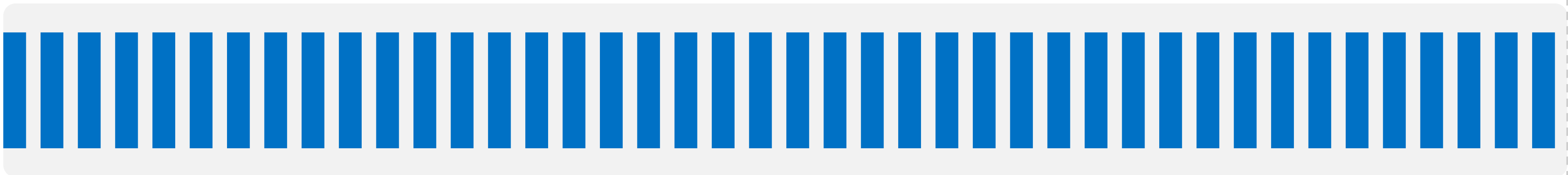
1× (steady)

USER 4'S TIME TO FIRST TOKEN

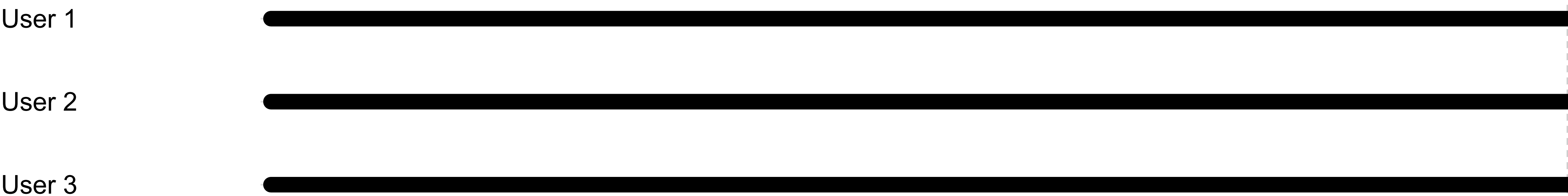
—

Times are in units of one decode step and illustrative. Chunked prefill splits a long prompt into pieces: pauses get shorter but more frequent. User 4 pays a small transfer cost in disaggregated mode.

GPU worker
prefill + decode



TOKENS DELIVERED TO EACH USER



User 4 (new)

time → now

prefill decode iteration KV transfer (NIXL) token delivered

04 Aggregated vs. disaggregated serving

Prefill workers

Compute the prompt's KV cache

Decode workers

Generate tokens one by one

Aggregated vs. Disaggregated

What a long prompt does to everyone else

Aggregated. One pool runs both phases. While it works through a long prefill, every other user's decode waits.

Disaggregated. Prefill runs on its own pool. Decode keeps its rhythm, and the new request joins once its KV cache arrives over NIXL.

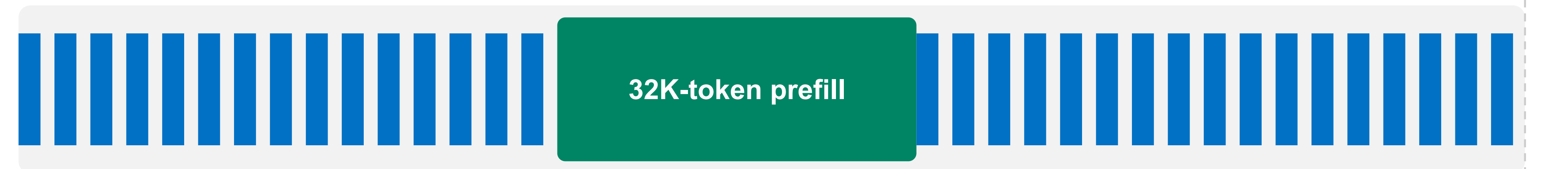
LONGEST GAP BETWEEN TOKENS,
USERS 1–3

11× normal

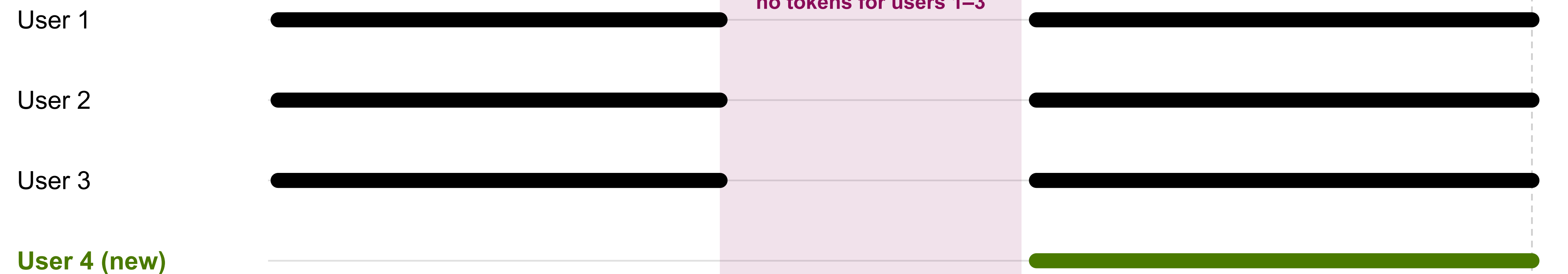
10 steps

Times are in units of one decode step and illustrative. Chunked prefill splits a long prompt into pieces: pauses get shorter but more frequent. User 4 pays a small transfer cost in disaggregated mode.

GPU worker
prefill + decode



TOKENS DELIVERED TO EACH USER



time →

now

■ prefill ■ decode iteration ■ KV transfer (NIXL) ● token delivered

04 Aggregated vs. disaggregated serving

Prefill workers

Compute the prompt's KV cache

Decode workers

Generate tokens one by one

Aggregated vs. Disaggregated

What a long prompt does to everyone else

Aggregated. One pool runs both phases. While it works through a long prefill, every other user's decode waits.

Disaggregated. Prefill runs on its own pool. Decode keeps its rhythm, and the new request joins once its KV cache arrives over NIXL.

LONGEST GAP BETWEEN TOKENS,
USERS 1-3

1× (steady)

USER 4'S TIME TO FIRST TOKEN

—

Times are in units of one decode step and illustrative. Chunked prefill splits a long prompt into pieces: pauses get shorter but more frequent. User 4 pays a small transfer cost in disaggregated mode.

Prefill worker
prefill only

Decode worker
decode only

TOKENS DELIVERED TO EACH USER

User 1

User 2

User 3

User 4 (new)

time →

now

prefill

decode iteration

KV transfer (NIXL)

token delivered

04 Aggregated vs. disaggregated serving

Prefill workers

Compute the prompt's KV cache

Decode workers

Generate tokens one by one

Aggregated vs. Disaggregated

What a long prompt does to everyone else

Aggregated. One pool runs both phases. While it works through a long prefill, every other user's decode waits.

Disaggregated. Prefill runs on its own pool. Decode keeps its rhythm, and the new request joins once its KV cache arrives over NIXL.

LONGEST GAP BETWEEN TOKENS,
USERS 1-3

USER 4'S TIME TO FIRST TOKEN

1× (steady)

12 steps

Times are in units of one decode step and illustrative. Chunked prefill splits a long prompt into pieces: pauses get shorter but more frequent. User 4 pays a small transfer cost in disaggregated mode.

Prefill worker
prefill only

Decode worker
decode only

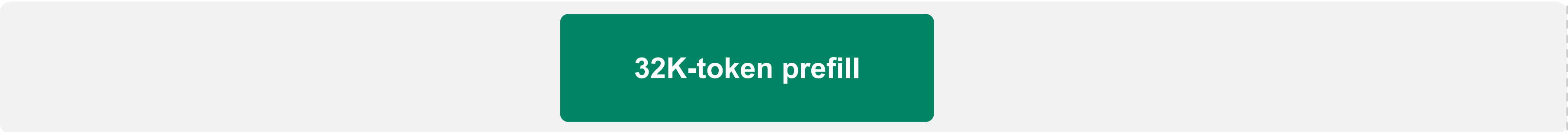
TOKENS DELIVERED TO EACH USER

User 1

User 2

User 3

User 4 (new)



time → now

■ prefill ■ decode iteration ■ KV transfer (NIXL) ● token delivered

04 Aggregated vs. disaggregated serving

Prefill workers

Compute the prompt's KV cache

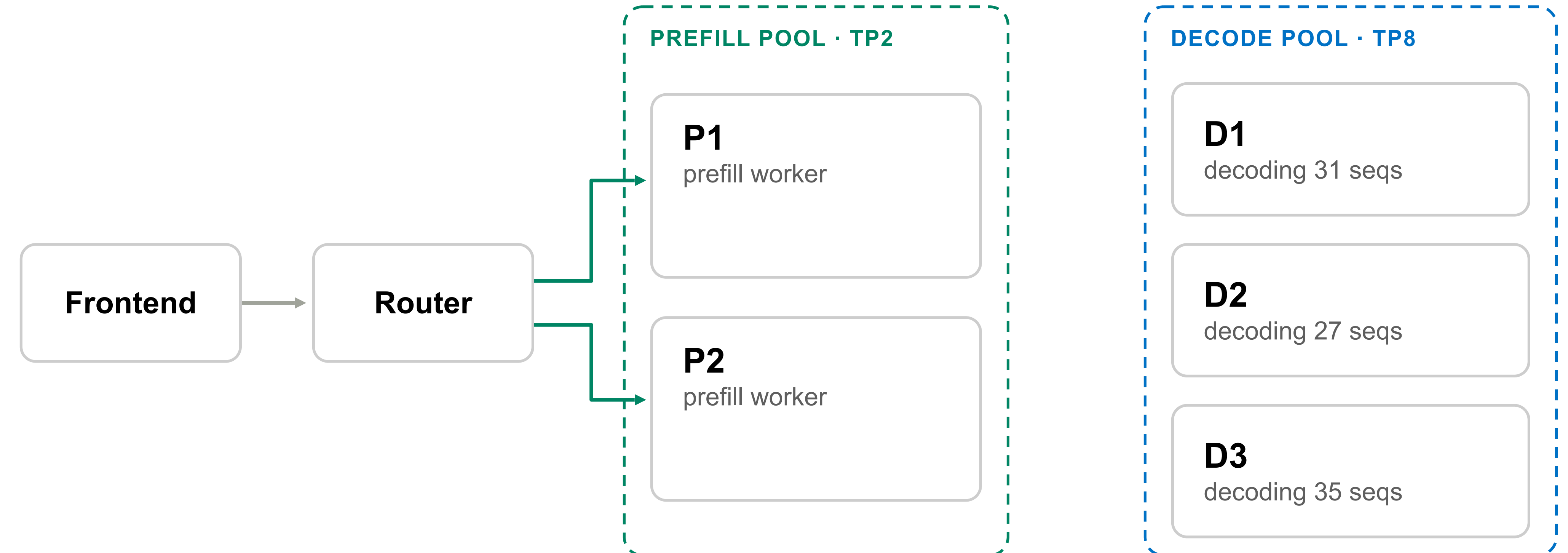
Decode workers

Generate tokens one by one

Disaggregated Serving in Dynamo

Separate pools; KV cache moves from prefill to decode over NIXL

- 1 **Preprocess.** The frontend tokenizes; the router sees prefill and decode workers registered for the same model.
- 2 **Pick a prefill worker** using KV overlap and load.
- 3 **Prefill** computes the prompt's KV cache and returns transfer metadata.
- 4 **Pick a decode worker.**
- 5 **Transfer.** The decode worker pulls the KV blocks GPU-to-GPU with NIXL (NVLink or RDMA), without stalling its other requests.
- 6 **Decode** streams tokens back through the frontend; KV events update the router's view.



DISAGGREGATE WHEN...

- prompts are long and load is sustained
- prefill and decode want different parallelism (large models)
- ITL targets are tight

STAY AGGREGATED WHEN...

- prompts are short or mostly cached: the transfer costs more than it saves
- concurrency is low
- the model is small

04 Aggregated vs. disaggregated serving

Prefill workers

Compute the prompt's KV cache

Decode workers

Generate tokens one by one

Disaggregated Serving in Dynamo

Separate pools; KV cache moves from prefill to decode over NIXL

1 **Preprocess.** The frontend tokenizes; the router sees prefill and decode workers registered for the same model.

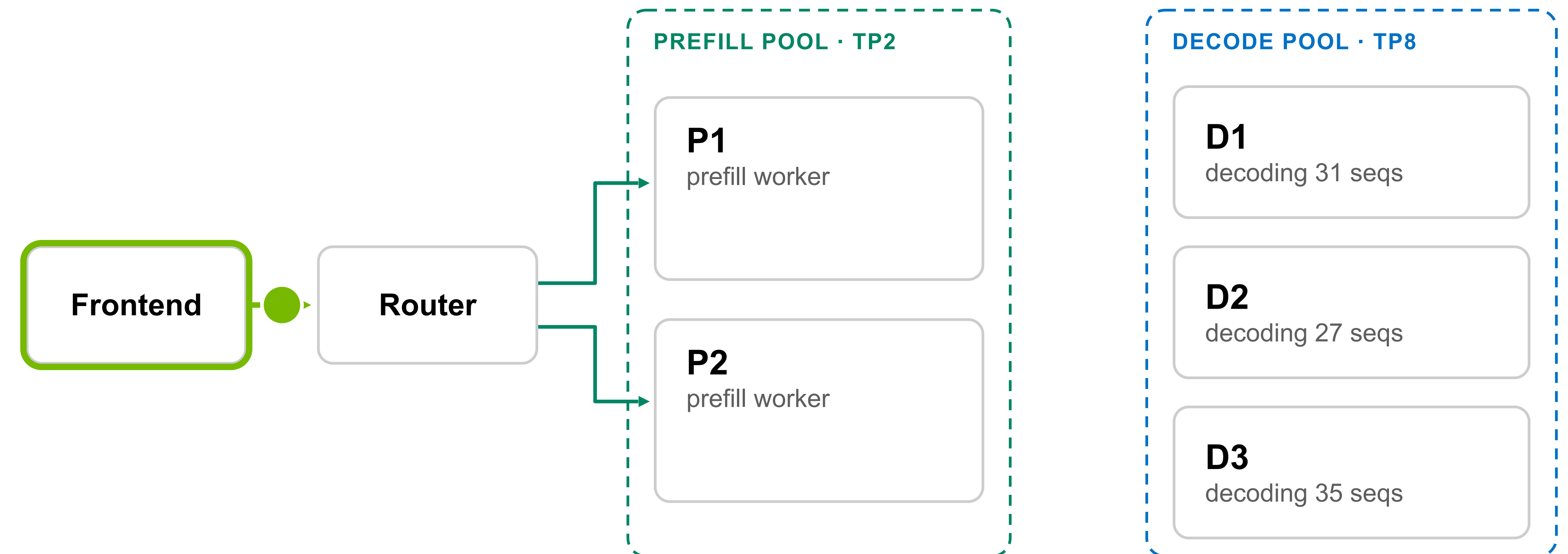
2 **Pick a prefill worker** using KV overlap and load.

3 **Prefill** computes the prompt's KV cache and returns transfer metadata.

4 **Pick a decode worker.**

5 **Transfer.** The decode worker pulls the KV blocks GPU-to-GPU with NIXL (NVLink or RDMA), without stalling its other requests.

6 **Decode** streams tokens back through the frontend; KV events update the router's view.



DISAGGREGATE WHEN...

- prompts are long and load is sustained
- prefill and decode want different parallelism (large models)
- ITL targets are tight

STAY AGGREGATED WHEN...

- prompts are short or mostly cached: the transfer costs more than it saves
- concurrency is low
- the model is small

04 Aggregated vs. disaggregated serving

Prefill workers

Compute the prompt's KV cache

Decode workers

Generate tokens one by one

Disaggregated Serving in Dynamo

Separate pools; KV cache moves from prefill to decode over NIXL

1 **Preprocess.** The frontend tokenizes; the router sees prefill and decode workers registered for the same model.

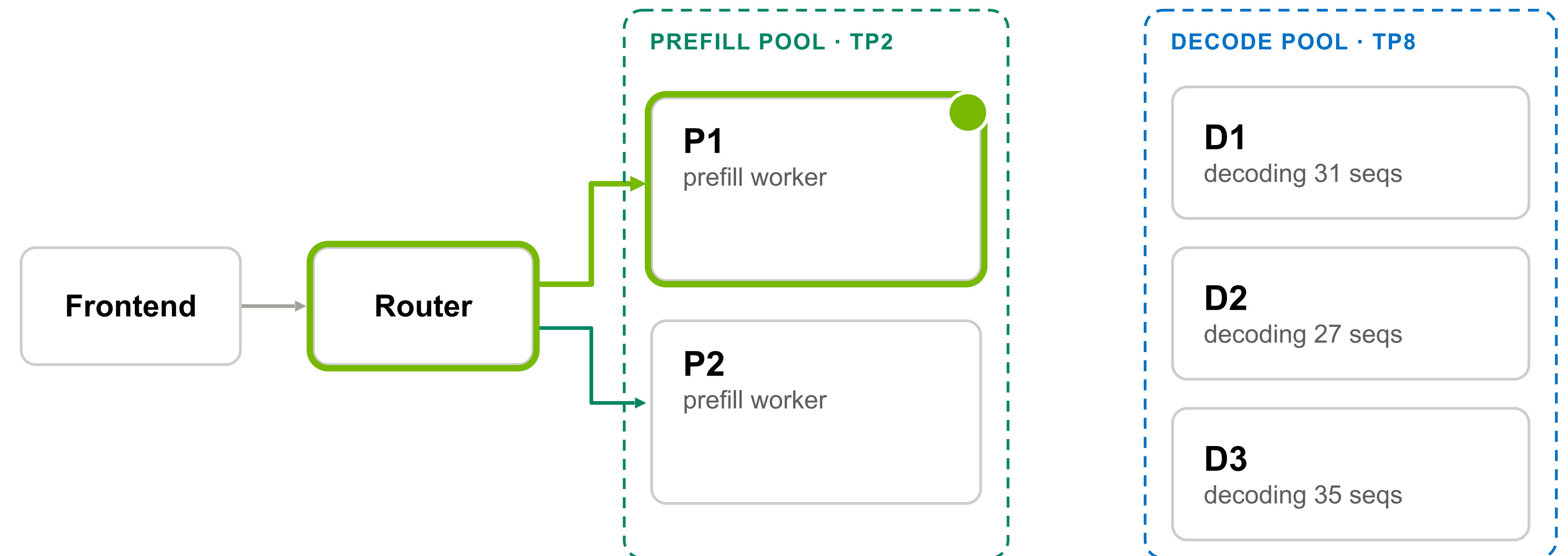
2 **Pick a prefill worker** using KV overlap and load.

3 **Prefill** computes the prompt's KV cache and returns transfer metadata.

4 **Pick a decode worker.**

5 **Transfer.** The decode worker pulls the KV blocks GPU-to-GPU with NIXL (NVLink or RDMA), without stalling its other requests.

6 **Decode** streams tokens back through the frontend; KV events update the router's view.



DISAGGREGATE WHEN...

- prompts are long and load is sustained
- prefill and decode want different parallelism (large models)
- ITL targets are tight

STAY AGGREGATED WHEN...

- prompts are short or mostly cached: the transfer costs more than it saves
- concurrency is low
- the model is small

04 Aggregated vs. disaggregated serving

Prefill workers

Compute the prompt's KV cache

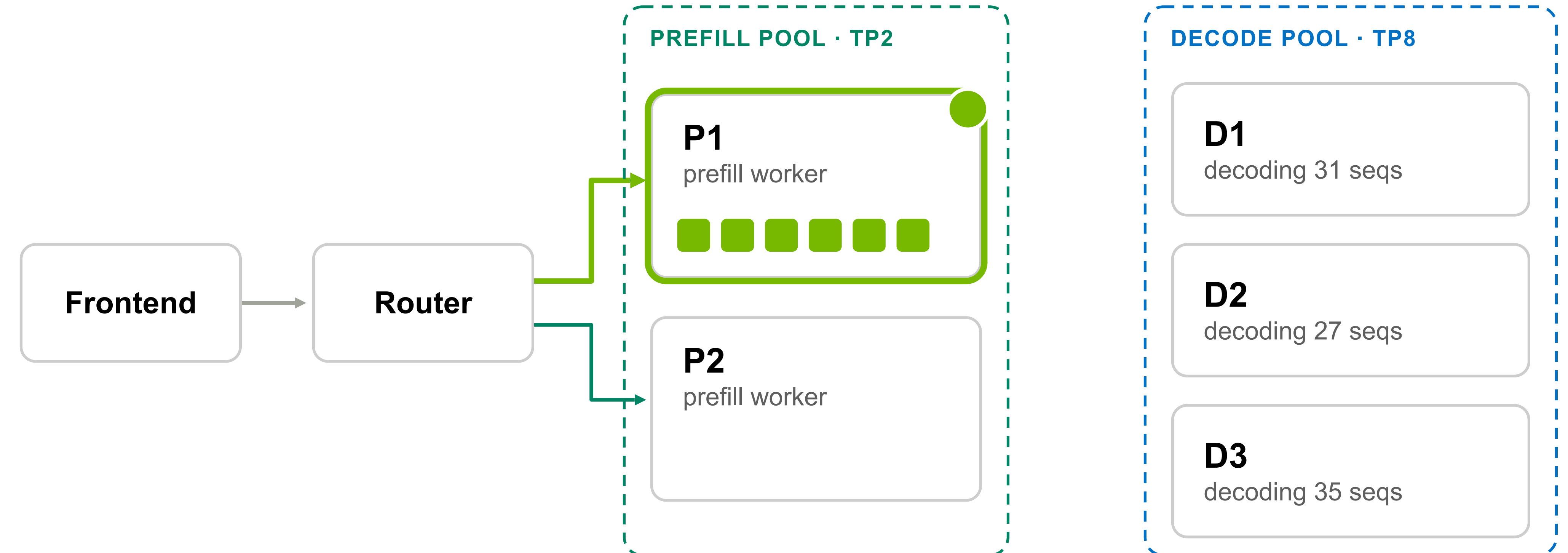
Decode workers

Generate tokens one by one

Disaggregated Serving in Dynamo

Separate pools; KV cache moves from prefill to decode over NIXL

- 1 **Preprocess.** The frontend tokenizes; the router sees prefill and decode workers registered for the same model.
- 2 **Pick a prefill worker** using KV overlap and load.
- 3 **Prefill** computes the prompt's KV cache and returns transfer metadata.
- 4 **Pick a decode worker.**
- 5 **Transfer.** The decode worker pulls the KV blocks GPU-to-GPU with NIXL (NVLink or RDMA), without stalling its other requests.
- 6 **Decode** streams tokens back through the frontend; KV events update the router's view.



DISAGGREGATE WHEN...

- prompts are long and load is sustained
- prefill and decode want different parallelism (large models)
- ITL targets are tight

STAY AGGREGATED WHEN...

- prompts are short or mostly cached: the transfer costs more than it saves
- concurrency is low
- the model is small

04 Aggregated vs. disaggregated serving

Prefill workers

Compute the prompt's KV cache

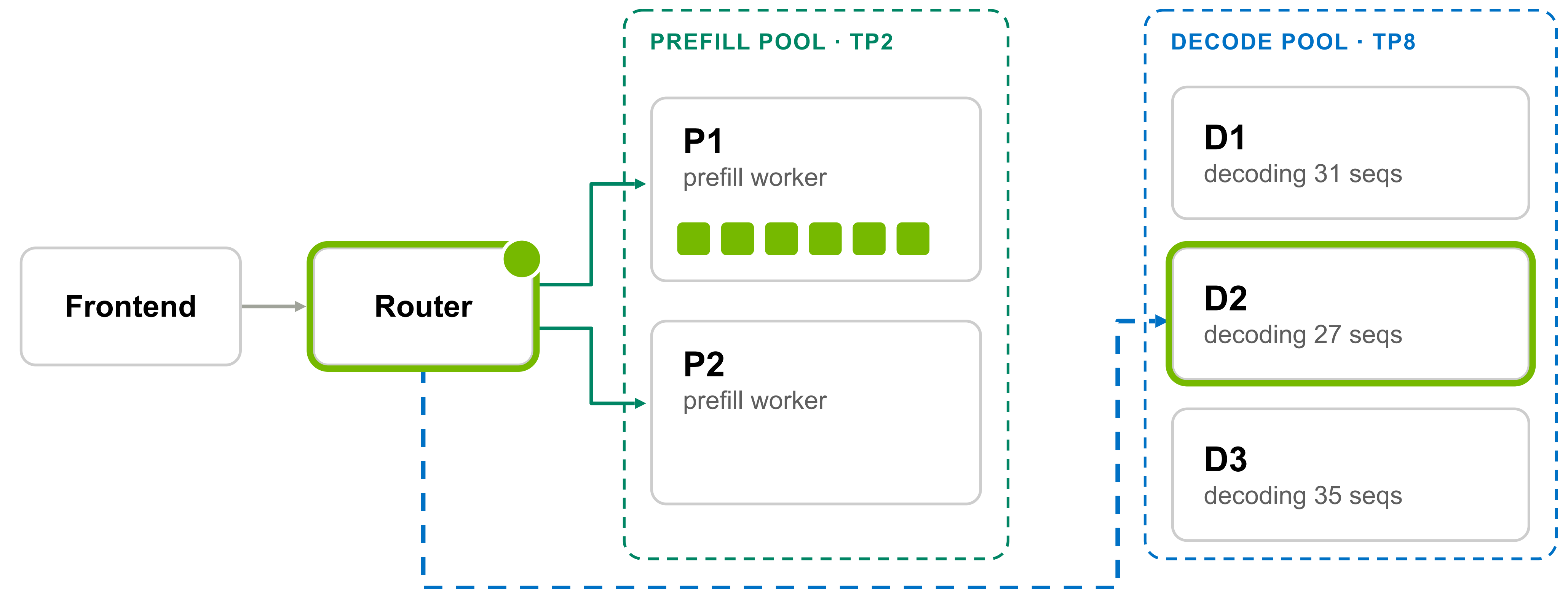
Decode workers

Generate tokens one by one

Disaggregated Serving in Dynamo

Separate pools; KV cache moves from prefill to decode over NIXL

- 1 **Preprocess.** The frontend tokenizes; the router sees prefill and decode workers registered for the same model.
- 2 **Pick a prefill worker** using KV overlap and load.
- 3 **Prefill** computes the prompt's KV cache and returns transfer metadata.
- 4 **Pick a decode worker.**
- 5 **Transfer.** The decode worker pulls the KV blocks GPU-to-GPU with NIXL (NVLink or RDMA), without stalling its other requests.
- 6 **Decode** streams tokens back through the frontend; KV events update the router's view.



DISAGGREGATE WHEN...

- prompts are long and load is sustained
- prefill and decode want different parallelism (large models)
- ITL targets are tight

STAY AGGREGATED WHEN...

- prompts are short or mostly cached: the transfer costs more than it saves
- concurrency is low
- the model is small

04 Aggregated vs. disaggregated serving

Prefill workers

Compute the prompt's KV cache

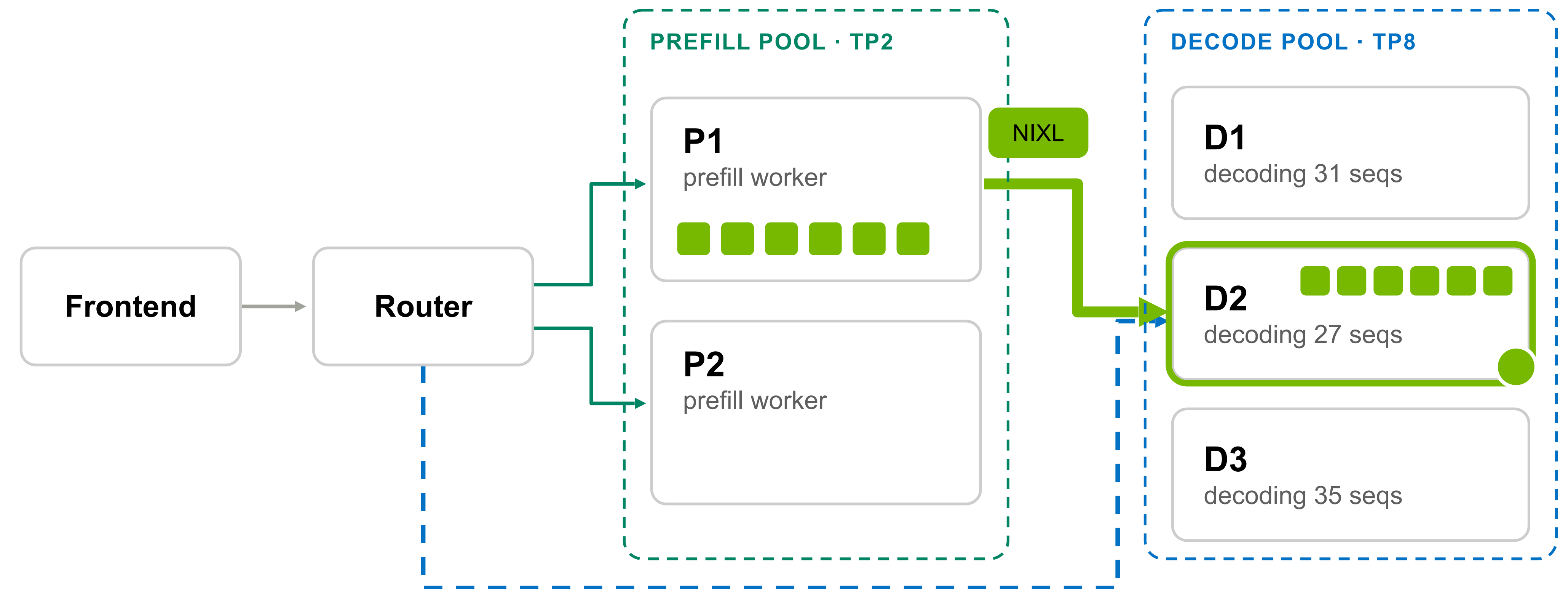
Decode workers

Generate tokens one by one

Disaggregated Serving in Dynamo

Separate pools; KV cache moves from prefill to decode over NIXL

- 1 **Preprocess.** The frontend tokenizes; the router sees prefill and decode workers registered for the same model.
- 2 **Pick a prefill worker** using KV overlap and load.
- 3 **Prefill** computes the prompt's KV cache and returns transfer metadata.
- 4 **Pick a decode worker.**
- 5 **Transfer.** The decode worker pulls the KV blocks GPU-to-GPU with NIXL (NVLink or RDMA), without stalling its other requests.
- 6 **Decode** streams tokens back through the frontend; KV events update the router's view.



DISAGGREGATE WHEN...

- prompts are long and load is sustained
- prefill and decode want different parallelism (large models)
- ITL targets are tight

STAY AGGREGATED WHEN...

- prompts are short or mostly cached: the transfer costs more than it saves
- concurrency is low
- the model is small

04 Aggregated vs. disaggregated serving

Prefill workers

Compute the prompt's KV cache

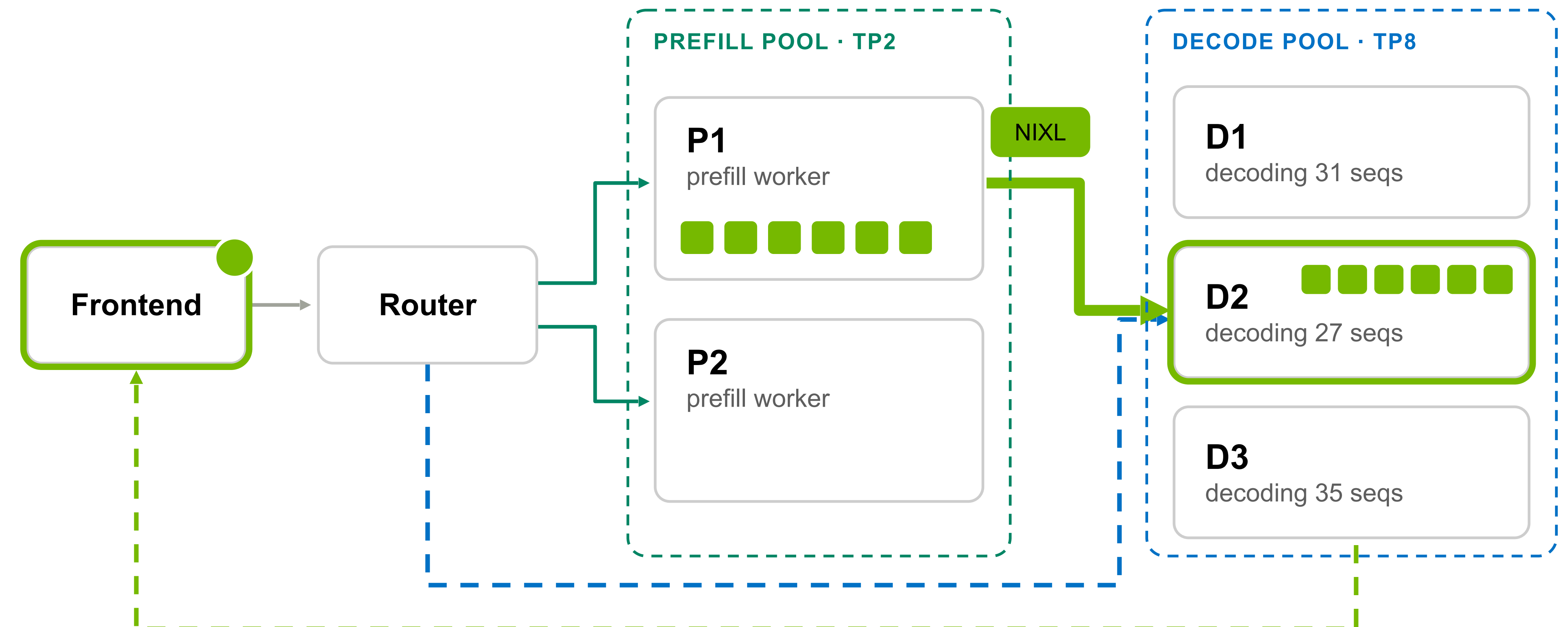
Decode workers

Generate tokens one by one

Disaggregated Serving in Dynamo

Separate pools; KV cache moves from prefill to decode over NIXL

- 1 Preprocess.** The frontend tokenizes; the router sees prefill and decode workers registered for the same model.
- 2 Pick a prefill worker** using KV overlap and load.
- 3 Prefill** computes the prompt's KV cache and returns transfer metadata.
- 4 Pick a decode worker.**
- 5 Transfer.** The decode worker pulls the KV blocks GPU-to-GPU with NIXL (NVLink or RDMA), without stalling its other requests.
- 6 Decode** streams tokens back through the frontend; KV events update the router's view.



DISAGGREGATE WHEN...

- prompts are long and load is sustained
- prefill and decode want different parallelism (large models)
- ITL targets are tight

STAY AGGREGATED WHEN...

- prompts are short or mostly cached: the transfer costs more than it saves
- concurrency is low
- the model is small

04 Aggregated vs. disaggregated serving

Prefill workers

Compute the prompt's KV cache

Decode workers

Generate tokens one by one

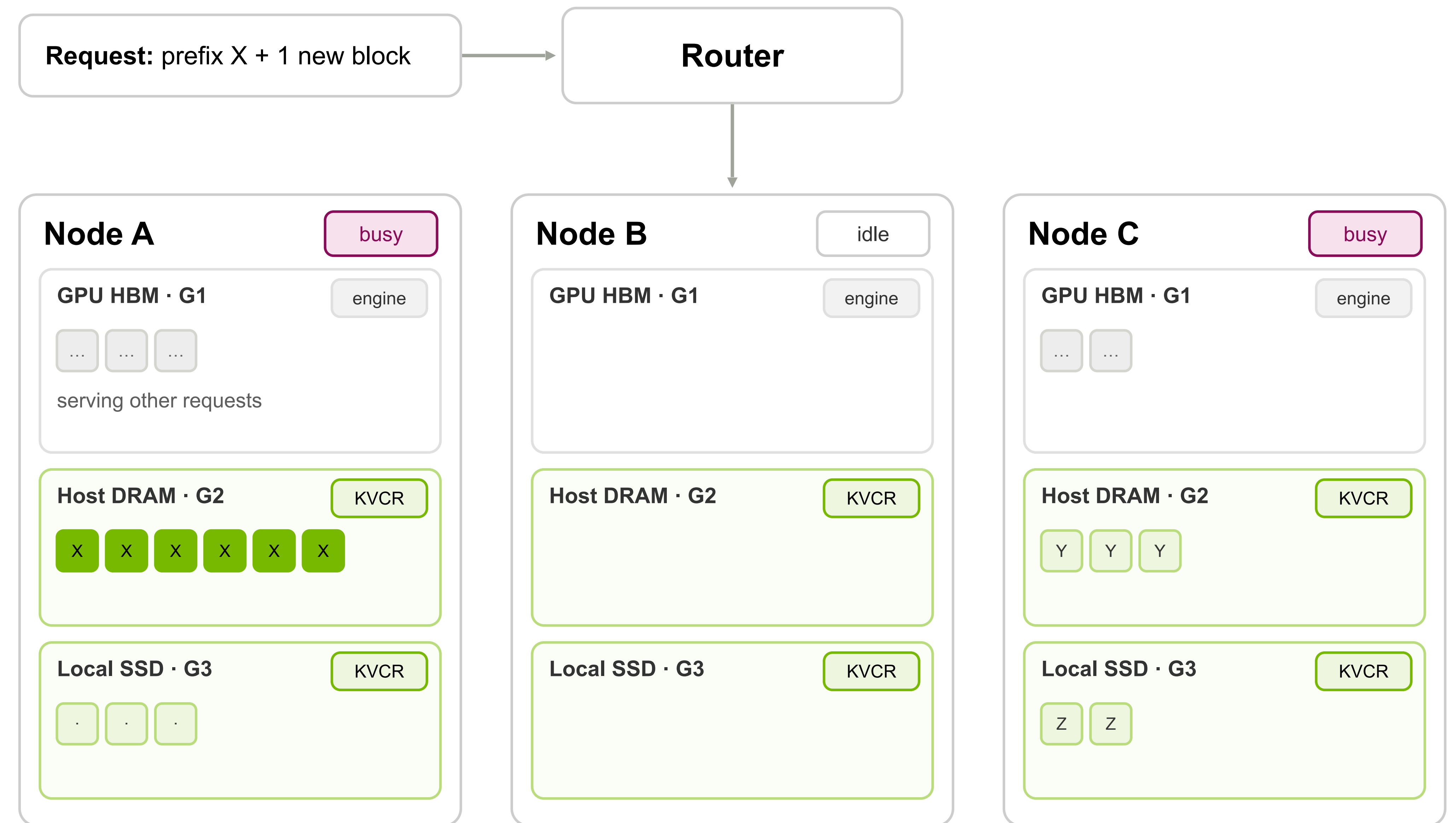
KVCR: KV Cache Runner

KV cache as a cluster-wide resource. Alternatives: LMCache, FlexKV, Mooncake store...

KVCR runs inside each worker. It manages KV tiers beyond the GPU (host memory, SSD, object storage), tells the router what it holds, and pulls blocks from peers over NIXL when the router says where they are.

WHY IT MATTERS

- **Today each node's cache is an island.** When a follow-up turn lands on a different worker because the original one is busy, that worker recomputes the whole prefix.
- **Popular prefixes get recomputed on many nodes.**
- **Crashes lose cache.** When an engine dies, its KV dies with it.
- **With KVCR, reuse crosses node boundaries:** the router hints where the blocks are, and the worker fetches instead of recomputing.



G1 · GPU HBM
Fastest and smallest. Owned by the engine.

G2 / G3 · host DRAM, local SSD
Larger and slower. Managed by KVCR, shareable with peers.

Remote / object storage
Largest. Optional tier. Engines keep their own local host offload too.

05

KVCR

KV cache shared across nodes and tiers

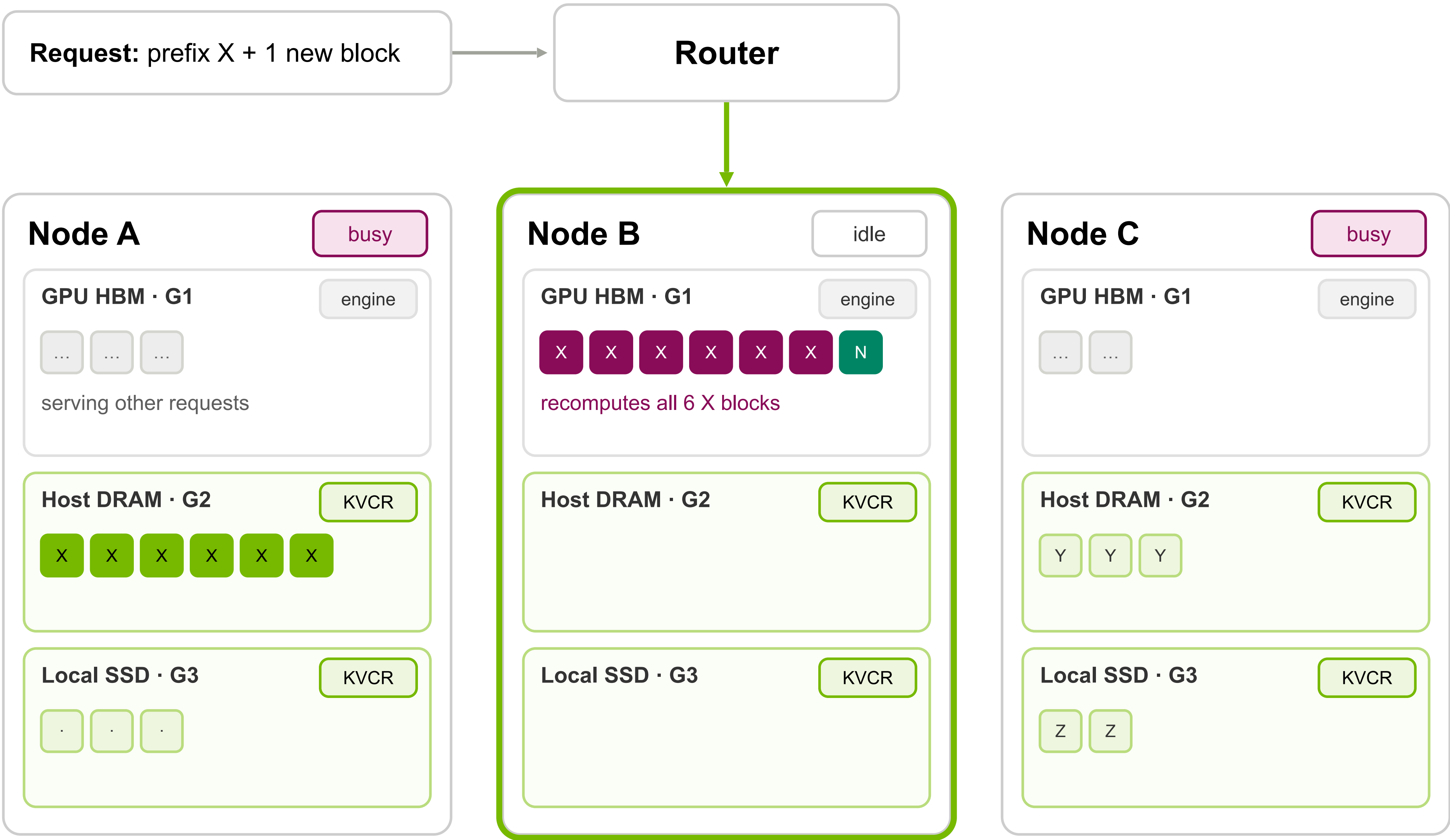
KVCR: KV Cache Runner

KV cache as a cluster-wide resource. Alternatives: LMCache, FlexKV, Mooncake store...

KVCR runs inside each worker. It manages KV tiers beyond the GPU (host memory, SSD, object storage), tells the router what it holds, and pulls blocks from peers over NIXL when the router says where they are.

WHY IT MATTERS

- **Today each node's cache is an island.** When a follow-up turn lands on a different worker because the original one is busy, that worker recomputes the whole prefix.
- **Popular prefixes get recomputed on many nodes.**
- **Crashes lose cache.** When an engine dies, its KV dies with it.
- **With KVCR, reuse crosses node boundaries:** the router hints where the blocks are, and the worker fetches instead of recomputing.



G1 · GPU HBM
Fastest and smallest. Owned by the engine.

G2 / G3 · host DRAM, local SSD
Larger and slower. Managed by KVCR, shareable with peers.

Remote / object storage
Largest. Optional tier. Engines keep their own local host offload too.

05

KVCR

KV cache shared across nodes and tiers

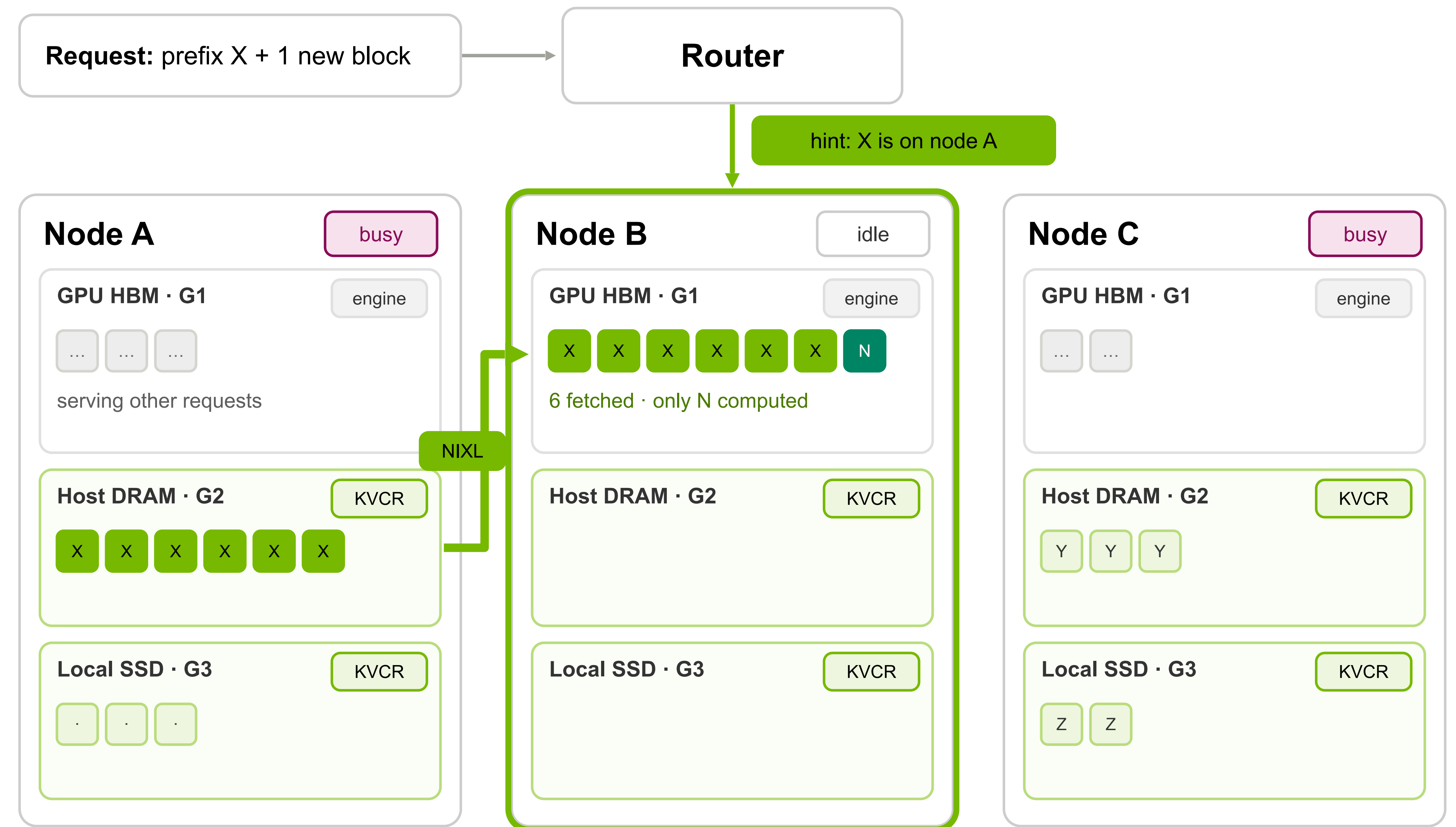
KVCR: KV Cache Runner

KV cache as a cluster-wide resource. Alternatives: LMCache, FlexKV, Mooncake store...

KVCR runs inside each worker. It manages KV tiers beyond the GPU (host memory, SSD, object storage), tells the router what it holds, and pulls blocks from peers over NIXL when the router says where they are.

WHY IT MATTERS

- **Today each node's cache is an island.** When a follow-up turn lands on a different worker because the original one is busy, that worker recomputes the whole prefix.
- **Popular prefixes get recomputed on many nodes.**
- **Crashes lose cache.** When an engine dies, its KV dies with it.
- **With KVCR, reuse crosses node boundaries:** the router hints where the blocks are, and the worker fetches instead of recomputing.



G1 · GPU HBM
Fastest and smallest. Owned by the engine.

G2 / G3 · host DRAM, local SSD
Larger and slower. Managed by KVCR, shareable with peers.

Remote / object storage
Largest. Optional tier. Engines keep their own local host offload too.

05

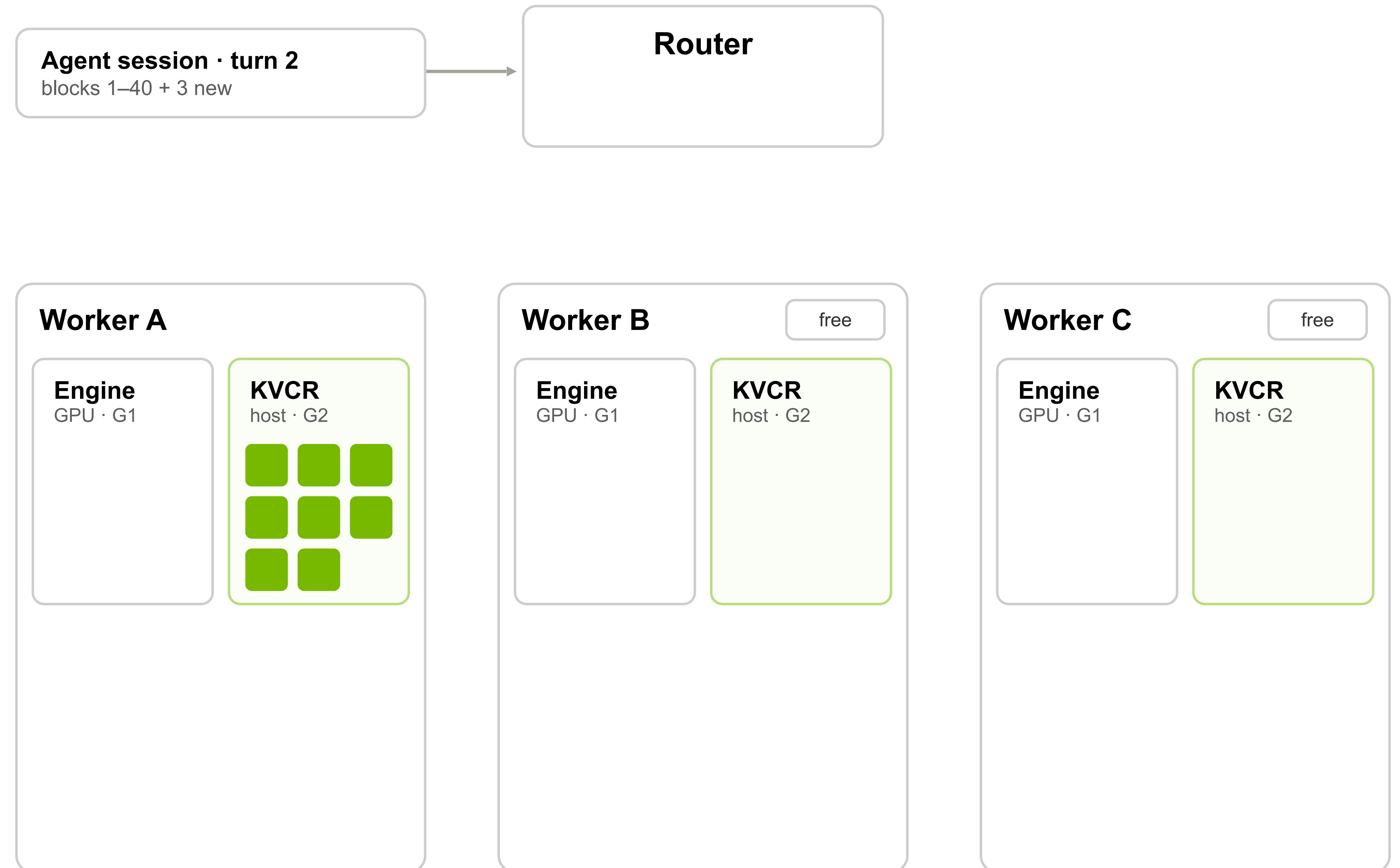
KVCR

KV cache shared across nodes and tiers

KVCR: A Cache Hit That Crosses Nodes

Advertise → hint → query → fetch → compute only what is new → survive a crash

- 1 **Advertise.** Each KVCR reports which blocks it holds, and in which tier; the router builds a cluster-wide map.
- 2 **Route with a hint.** A is busy, so turn 2 goes to B with a hint: "blocks 1–40 are on A".
- 3 **Query.** B's engine asks its KVCR about those blocks: hit, fetching, fetchable or miss.
- 4 **Fetch.** B pulls the blocks straight from A over NIXL; the bytes never pass through the router.
- 5 **Compute only what's new.** B uses the longest delivered prefix and prefills just the new tokens.
- 6 **Survive.** If A's engine dies, its KVCR-Guard sidecar keeps serving A's host-tier blocks to peers.



05

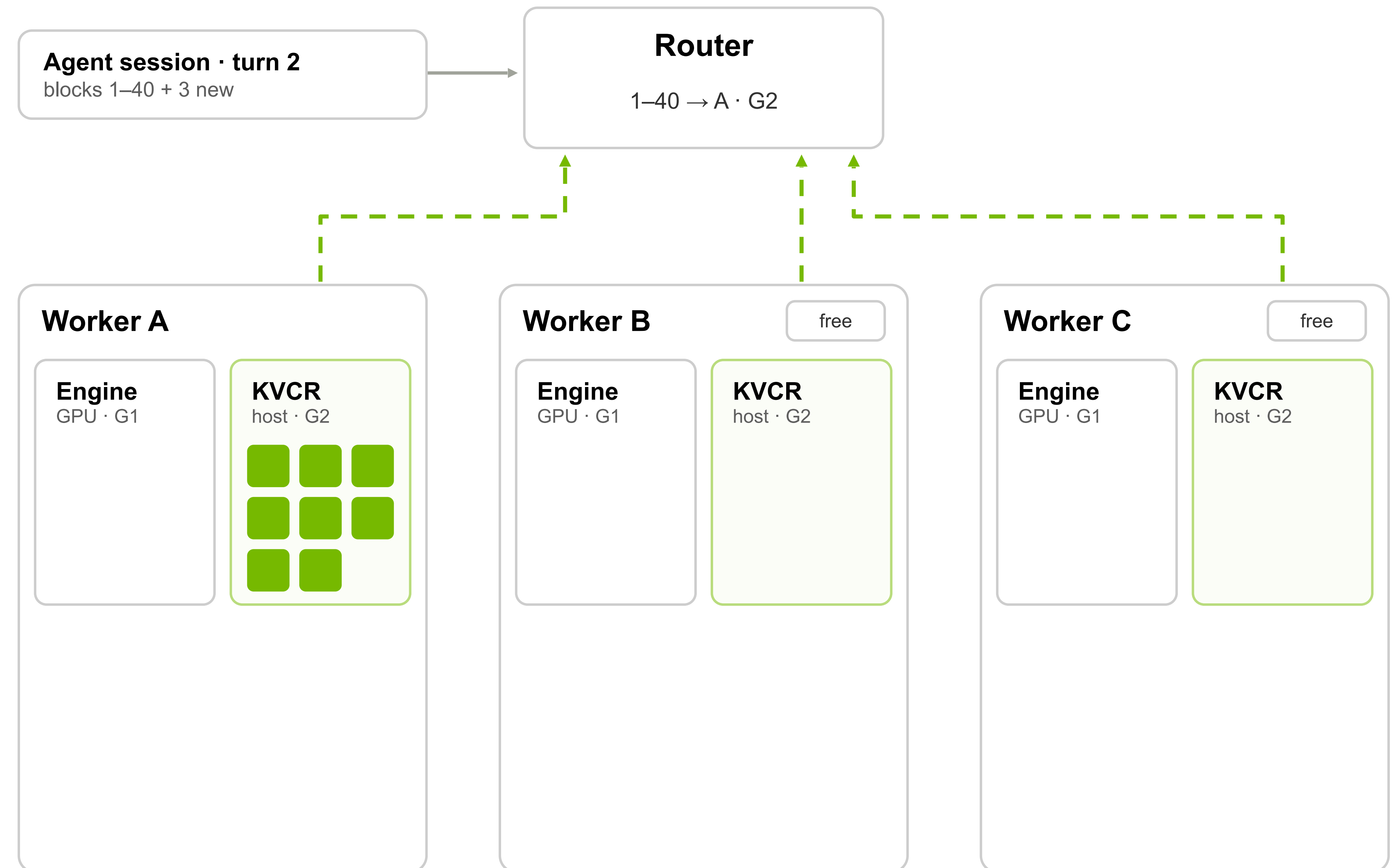
KVCR

KV cache shared across nodes and tiers

KVCR: A Cache Hit That Crosses Nodes

Advertise → hint → query → fetch → compute only what is new → survive a crash

- 1 Advertise.** Each KVCR reports which blocks it holds, and in which tier; the router builds a cluster-wide map.
- 2 Route with a hint.** A is busy, so turn 2 goes to B with a hint: "blocks 1–40 are on A".
- 3 Query.** B's engine asks its KVCR about those blocks: hit, fetching, fetchable or miss.
- 4 Fetch.** B pulls the blocks straight from A over NIXL; the bytes never pass through the router.
- 5 Compute only what's new.** B uses the longest delivered prefix and prefills just the new tokens.
- 6 Survive.** If A's engine dies, its KVCR-Guard sidecar keeps serving A's host-tier blocks to peers.



05

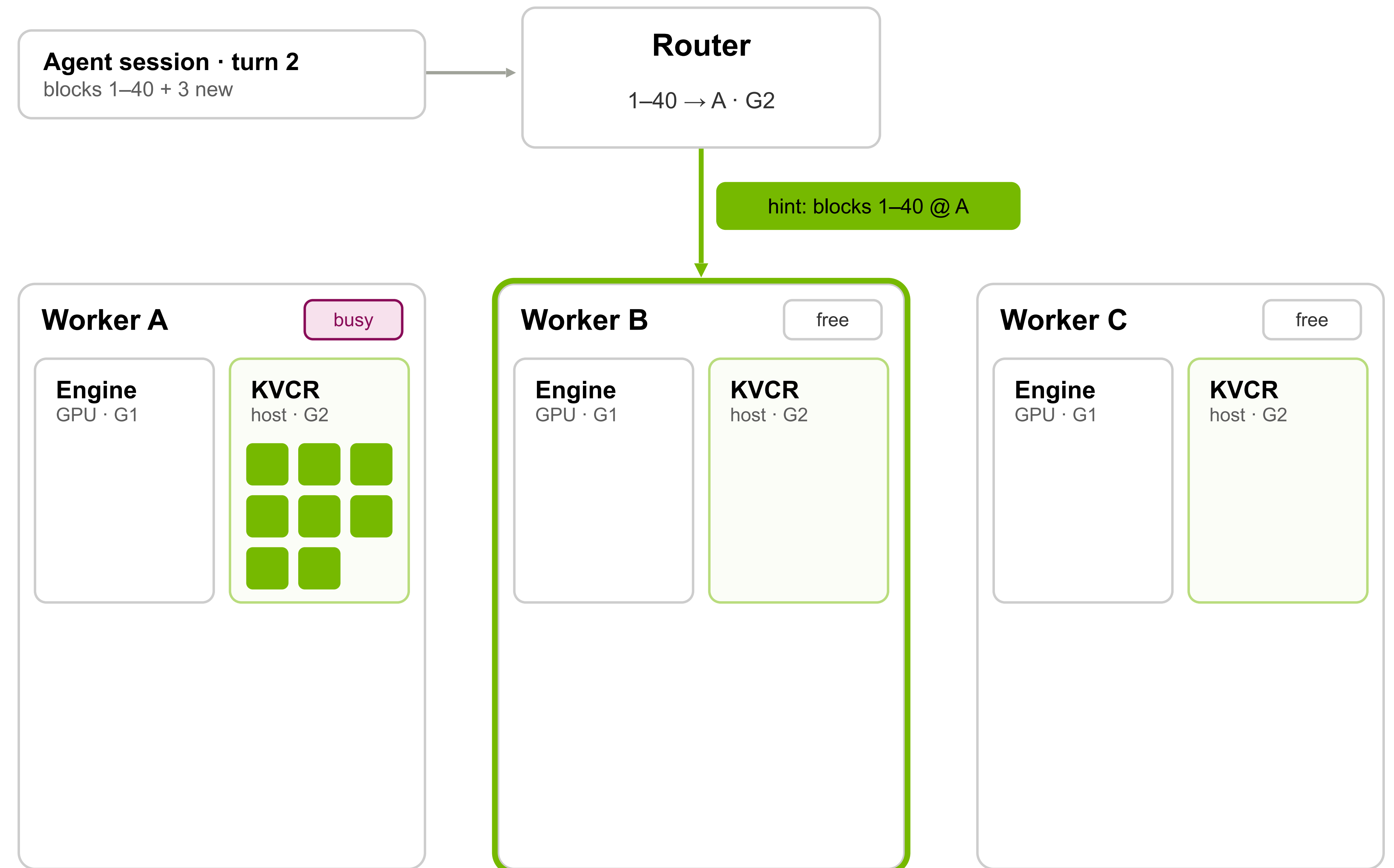
KVCR

KV cache shared across nodes and tiers

KVCR: A Cache Hit That Crosses Nodes

Advertise → hint → query → fetch → compute only what is new → survive a crash

- 1 Advertise.** Each KVCR reports which blocks it holds, and in which tier; the router builds a cluster-wide map.
- 2 Route with a hint.** A is busy, so turn 2 goes to B with a hint: "blocks 1–40 are on A".
- 3 Query.** B's engine asks its KVCR about those blocks: hit, fetching, fetchable or miss.
- 4 Fetch.** B pulls the blocks straight from A over NIXL; the bytes never pass through the router.
- 5 Compute only what's new.** B uses the longest delivered prefix and prefills just the new tokens.
- 6 Survive.** If A's engine dies, its KVCR-Guard sidecar keeps serving A's host-tier blocks to peers.



05

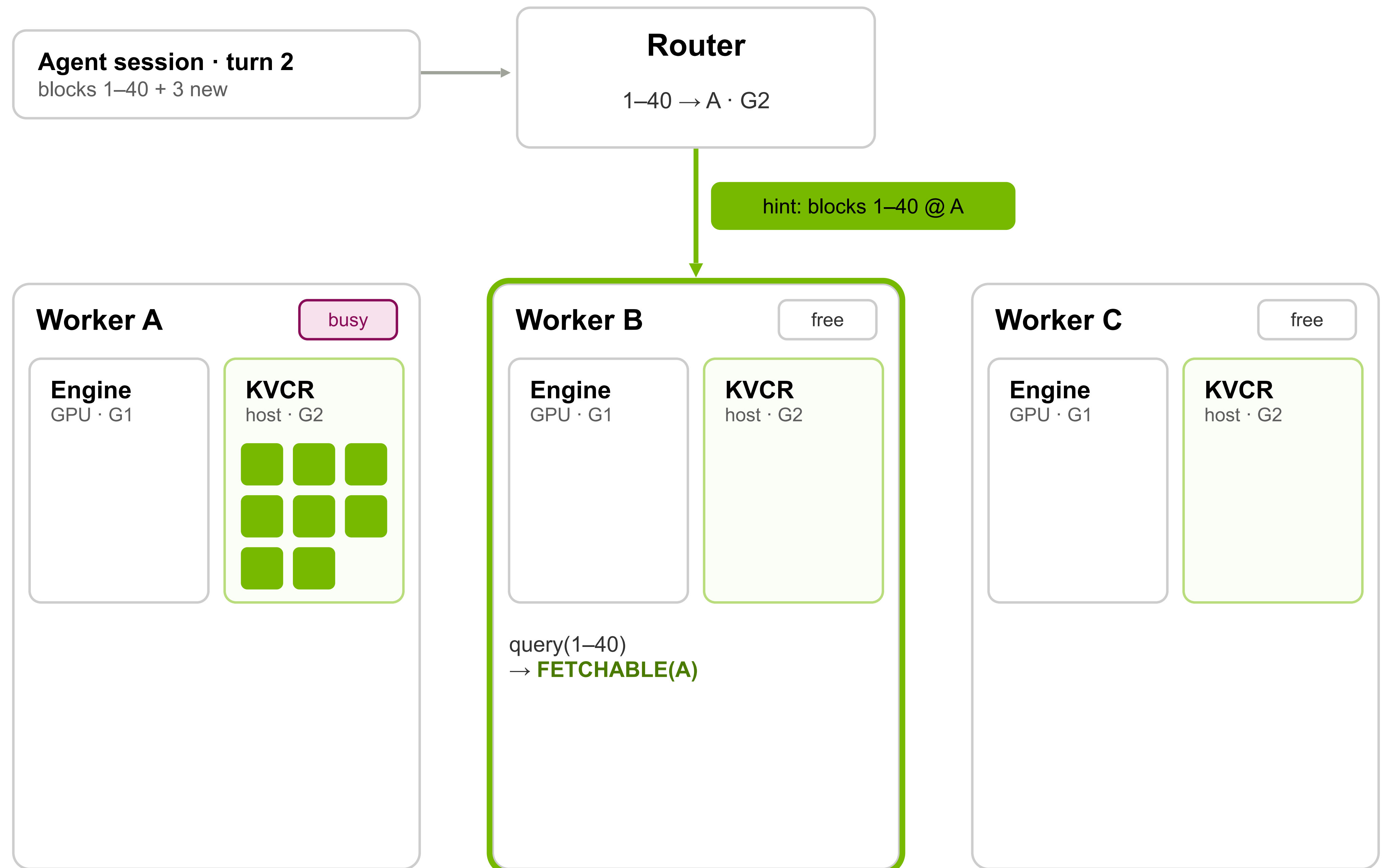
KVCR

KV cache shared across nodes and tiers

KVCR: A Cache Hit That Crosses Nodes

Advertise → hint → query → fetch → compute only what is new → survive a crash

- 1 Advertise.** Each KVCR reports which blocks it holds, and in which tier; the router builds a cluster-wide map.
- 2 Route with a hint.** A is busy, so turn 2 goes to B with a hint: "blocks 1–40 are on A".
- 3 Query.** B's engine asks its KVCR about those blocks: hit, fetching, fetchable or miss.
- 4 Fetch.** B pulls the blocks straight from A over NIXL; the bytes never pass through the router.
- 5 Compute only what's new.** B uses the longest delivered prefix and prefills just the new tokens.
- 6 Survive.** If A's engine dies, its KVCR-Guard sidecar keeps serving A's host-tier blocks to peers.



05

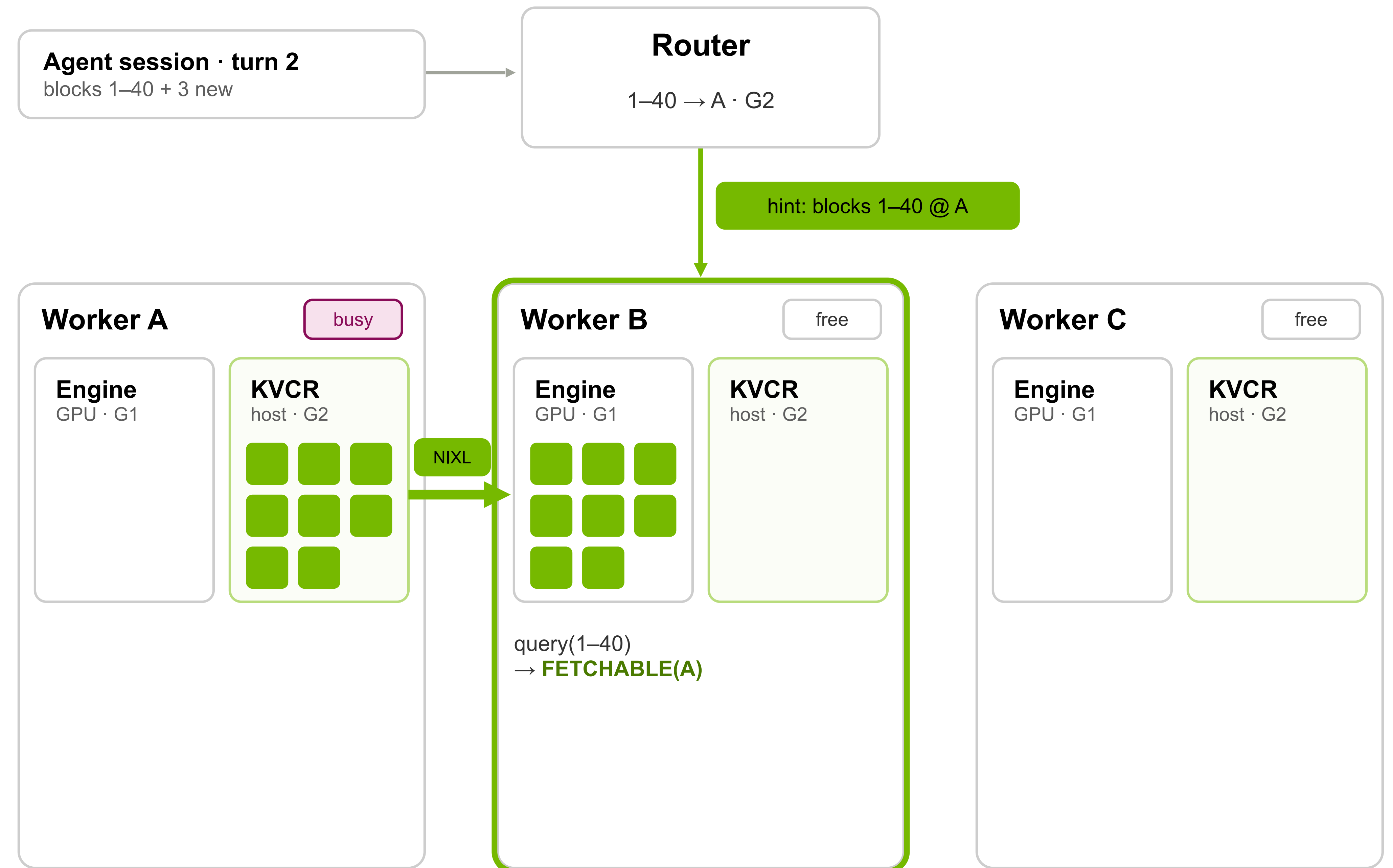
KVCR

KV cache shared across nodes and tiers

KVCR: A Cache Hit That Crosses Nodes

Advertise → hint → query → fetch → compute only what is new → survive a crash

- 1 Advertise.** Each KVCR reports which blocks it holds, and in which tier; the router builds a cluster-wide map.
- 2 Route with a hint.** A is busy, so turn 2 goes to B with a hint: "blocks 1–40 are on A".
- 3 Query.** B's engine asks its KVCR about those blocks: hit, fetching, fetchable or miss.
- 4 Fetch.** B pulls the blocks straight from A over NIXL; the bytes never pass through the router.
- 5 Compute only what's new.** B uses the longest delivered prefix and prefills just the new tokens.
- 6 Survive.** If A's engine dies, its KVCR-Guard sidecar keeps serving A's host-tier blocks to peers.



05

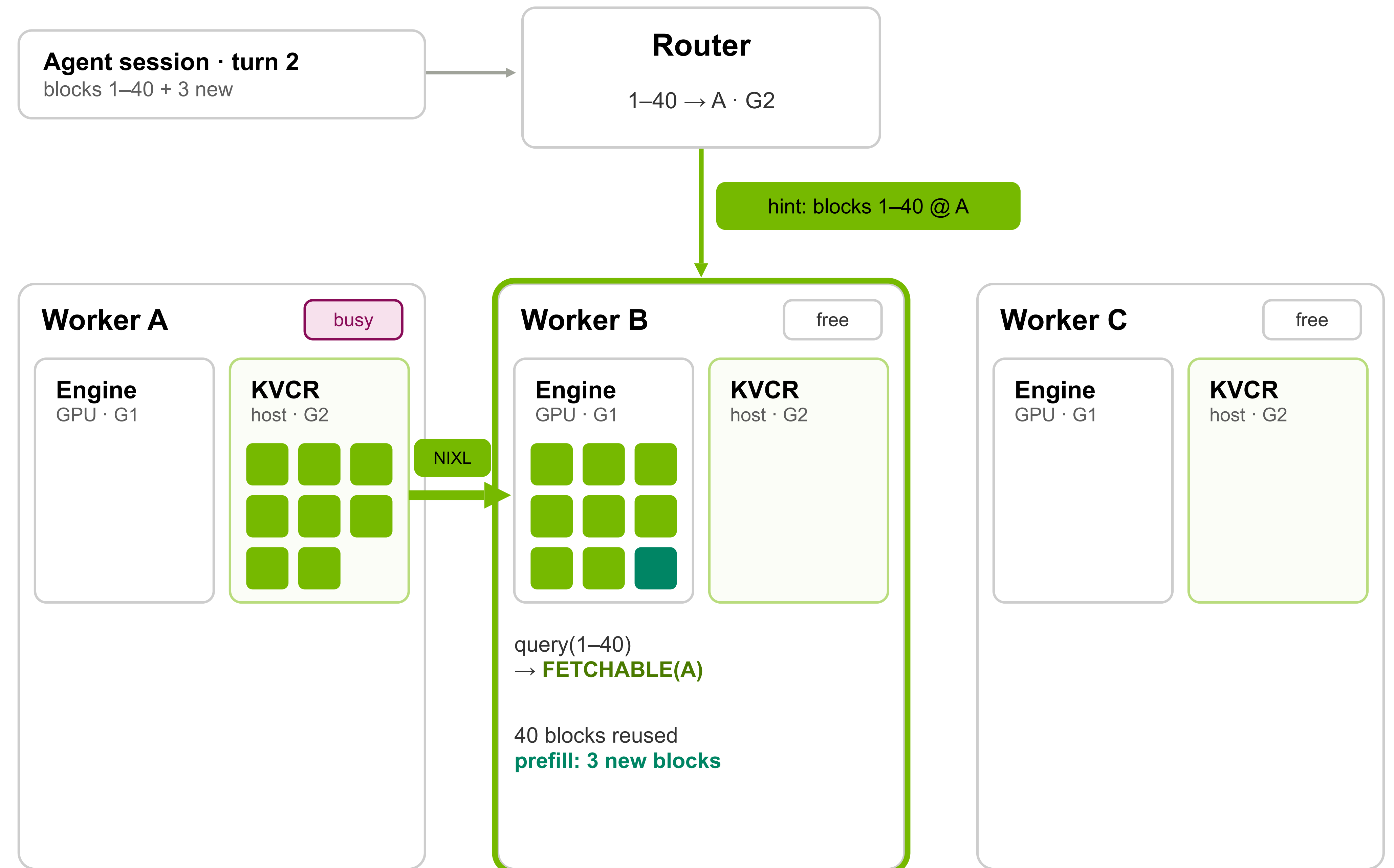
KVCR

KV cache shared across nodes and tiers

KVCR: A Cache Hit That Crosses Nodes

Advertise → hint → query → fetch → compute only what is new → survive a crash

- 1 Advertise.** Each KVCR reports which blocks it holds, and in which tier; the router builds a cluster-wide map.
- 2 Route with a hint.** A is busy, so turn 2 goes to B with a hint: "blocks 1–40 are on A".
- 3 Query.** B's engine asks its KVCR about those blocks: hit, fetching, fetchable or miss.
- 4 Fetch.** B pulls the blocks straight from A over NIXL; the bytes never pass through the router.
- 5 Compute only what's new.** B uses the longest delivered prefix and prefills just the new tokens.
- 6 Survive.** If A's engine dies, its KVCR-Guard sidecar keeps serving A's host-tier blocks to peers.



05

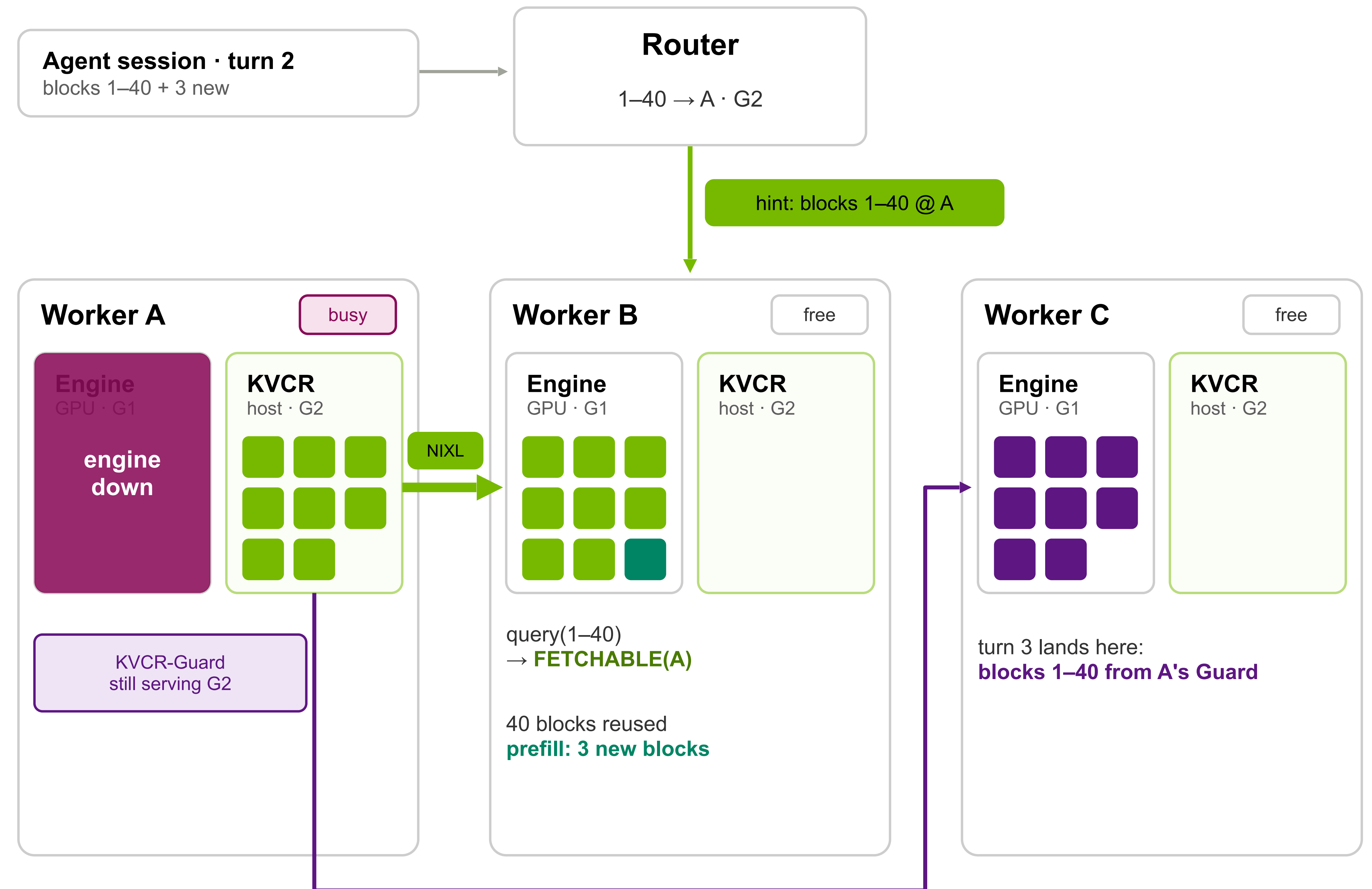
KVCR

KV cache shared across nodes and tiers

KVCR: A Cache Hit That Crosses Nodes

Advertise → hint → query → fetch → compute only what is new → survive a crash

- 1 Advertise.** Each KVCR reports which blocks it holds, and in which tier; the router builds a cluster-wide map.
- 2 Route with a hint.** A is busy, so turn 2 goes to B with a hint: "blocks 1–40 are on A".
- 3 Query.** B's engine asks its KVCR about those blocks: hit, fetching, fetchable or miss.
- 4 Fetch.** B pulls the blocks straight from A over NIXL; the bytes never pass through the router.
- 5 Compute only what's new.** B uses the longest delivered prefix and prefills just the new tokens.
- 6 Survive.** If A's engine dies, its KVCR-Guard sidecar keeps serving A's host-tier blocks to peers.



05

KVCR

KV cache shared across nodes and tiers

NIXL: NVIDIA Inference Xfer Library

One API to move data across any memory, any fabric

An open-source point-to-point transfer library. A framework asks to **READ** or **WRITE** a list of buffers; NIXL picks the transport and moves the bytes asynchronously, with no CPU copies.

WHY IT MATTERS

- **Distributed inference is mostly data movement:** KV from prefill to decode, KV to and from CPU, SSD and remote storage, weights during scale-up and RL updates.
- **Every memory and fabric has its own API:** NVLink, InfiniBand / RoCE, AWS EFA, GPUDirect Storage, S3.
- **Without a common layer,** every engine reimplements every path. With NIXL, swapping hardware swaps a plugin, not the engine.

USED BY

Dynamo

vLLM

SGLang

TensorRT-LLM

KVCR

NIXL agent one per process · north-bound API

register memory

exchange metadata

READ / WRITE (async)

notifications

PLUGINS

UCX

libfabric

GDS

POSIX

Mooncake

S3 / OBJ

Azure Blob

FABRIC

NVLink

InfiniBand / RoCE

AWS EFA

PCIe / C2C

NVMe

HTTP(S)

MEMORY

VRAM · GPU

DRAM · CPU

FILE

BLOCK

OBJECT

GPU → GPU, another node

GPU → host memory

GPU → local NVMe

GPU → object store

Click through four transfers to see which plugin, fabric and memory type NIXL uses. The call the framework makes is the same in every case.

06

NIXL

Move KV bytes over any fabric, to any memory

NIXL: NVIDIA Inference Xfer Library

One API to move data across any memory, any fabric

An open-source point-to-point transfer library. A framework asks to **READ** or **WRITE** a list of buffers; NIXL picks the transport and moves the bytes asynchronously, with no CPU copies.

WHY IT MATTERS

- **Distributed inference is mostly data movement:** KV from prefill to decode, KV to and from CPU, SSD and remote storage, weights during scale-up and RL updates.
- **Every memory and fabric has its own API:** NVLink, InfiniBand / RoCE, AWS EFA, GPUDirect Storage, S3.
- **Without a common layer,** every engine reimplements every path. With NIXL, swapping hardware swaps a plugin, not the engine.

USED BY

Dynamo

vLLM

SGLang

TensorRT-LLM

KVCR

NIXL agent one per process · north-bound API

register memory

exchange metadata

READ / WRITE (async)

notifications

PLUGINS

UCX

libfabric

GDS

POSIX

Mooncake

S3 / OBJ

Azure Blob

FABRIC

NVLink

InfiniBand / RoCE

AWS EFA

PCIe / C2C

NVMe

HTTP(S)

MEMORY

VRAM · GPU

DRAM · CPU

FILE

BLOCK

OBJECT

GPU → GPU, another node

GPU → host memory

GPU → local NVMe

GPU → object store

Prefill GPU → decode GPU on another node. UCX over NVLink or InfiniBand / RoCE, or libfabric on AWS EFA. GPU memory goes straight to GPU memory over RDMA, with no staging through host memory.

06

NIXL

Move KV bytes over any fabric, to any memory

NIXL: NVIDIA Inference Xfer Library

One API to move data across any memory, any fabric

An open-source point-to-point transfer library. A framework asks to **READ** or **WRITE** a list of buffers; NIXL picks the transport and moves the bytes asynchronously, with no CPU copies.

WHY IT MATTERS

- **Distributed inference is mostly data movement:** KV from prefill to decode, KV to and from CPU, SSD and remote storage, weights during scale-up and RL updates.
- **Every memory and fabric has its own API:** NVLink, InfiniBand / RoCE, AWS EFA, GPUDirect Storage, S3.
- **Without a common layer,** every engine reimplements every path. With NIXL, swapping hardware swaps a plugin, not the engine.

USED BY

Dynamo

vLLM

SGLang

TensorRT-LLM

KVCR

NIXL agent one per process · north-bound API

register memory

exchange metadata

READ / WRITE (async)

notifications

PLUGINS

UCX

libfabric

GDS

POSIX

Mooncake

S3 / OBJ

Azure Blob

FABRIC

NVLink

InfiniBand / RoCE

AWS EFA

PCIe / C2C

NVMe

HTTP(S)

MEMORY

VRAM · GPU

DRAM · CPU

FILE

BLOCK

OBJECT

GPU → GPU, another node

GPU → host memory

GPU → local NVMe

GPU → object store

Offload KV to host memory on the same node: a local transfer over PCIe (NVLink-C2C on Grace systems) into registered DRAM.

06

NIXL

Move KV bytes over any fabric, to any memory

NIXL: NVIDIA Inference Xfer Library

One API to move data across any memory, any fabric

An open-source point-to-point transfer library. A framework asks to **READ** or **WRITE** a list of buffers; NIXL picks the transport and moves the bytes asynchronously, with no CPU copies.

WHY IT MATTERS

- **Distributed inference is mostly data movement:** KV from prefill to decode, KV to and from CPU, SSD and remote storage, weights during scale-up and RL updates.
- **Every memory and fabric has its own API:** NVLink, InfiniBand / RoCE, AWS EFA, GPUDirect Storage, S3.
- **Without a common layer,** every engine reimplements every path. With NIXL, swapping hardware swaps a plugin, not the engine.

USED BY

Dynamo

vLLM

SGLang

TensorRT-LLM

KVCR

NIXL agent one per process · north-bound API

register memory

exchange metadata

READ / WRITE (async)

notifications

PLUGINS

UCX

libfabric

GDS

POSIX

Mooncake

S3 / OBJ

Azure Blob

FABRIC

NVLink

InfiniBand / RoCE

AWS EFA

PCIe / C2C

NVMe

HTTP(S)

MEMORY

VRAM · GPU

DRAM · CPU

FILE

BLOCK

OBJECT

GPU → GPU, another node

GPU → host memory

GPU → local NVMe

GPU → object store

Spill KV to local NVMe. GPUDirect Storage (GDS) moves file data directly between GPU memory and the SSD; POSIX is the portable fallback.

06

NIXL

Move KV bytes over any fabric, to any memory

NIXL: NVIDIA Inference Xfer Library

One API to move data across any memory, any fabric

An open-source point-to-point transfer library. A framework asks to **READ** or **WRITE** a list of buffers; NIXL picks the transport and moves the bytes asynchronously, with no CPU copies.

WHY IT MATTERS

- **Distributed inference is mostly data movement:** KV from prefill to decode, KV to and from CPU, SSD and remote storage, weights during scale-up and RL updates.
- **Every memory and fabric has its own API:** NVLink, InfiniBand / RoCE, AWS EFA, GPUDirect Storage, S3.
- **Without a common layer,** every engine reimplements every path. With NIXL, swapping hardware swaps a plugin, not the engine.

USED BY

Dynamo

vLLM

SGLang

TensorRT-LLM

KVCR

NIXL agent one per process · north-bound API

register memory

exchange metadata

READ / WRITE (async)

notifications

PLUGINS

UCX

libfabric

GDS

POSIX

Mooncake

S3 / OBJ

Azure Blob

FABRIC

NVLink

InfiniBand / RoCE

AWS EFA

PCIe / C2C

NVMe

HTTP(S)

MEMORY

VRAM · GPU

DRAM · CPU

FILE

BLOCK

OBJECT

GPU → GPU, another node

GPU → host memory

GPU → local NVMe

GPU → object store

Persist or share KV through object storage (S3-compatible or Azure Blob). Same READ / WRITE call, different plugin.

06

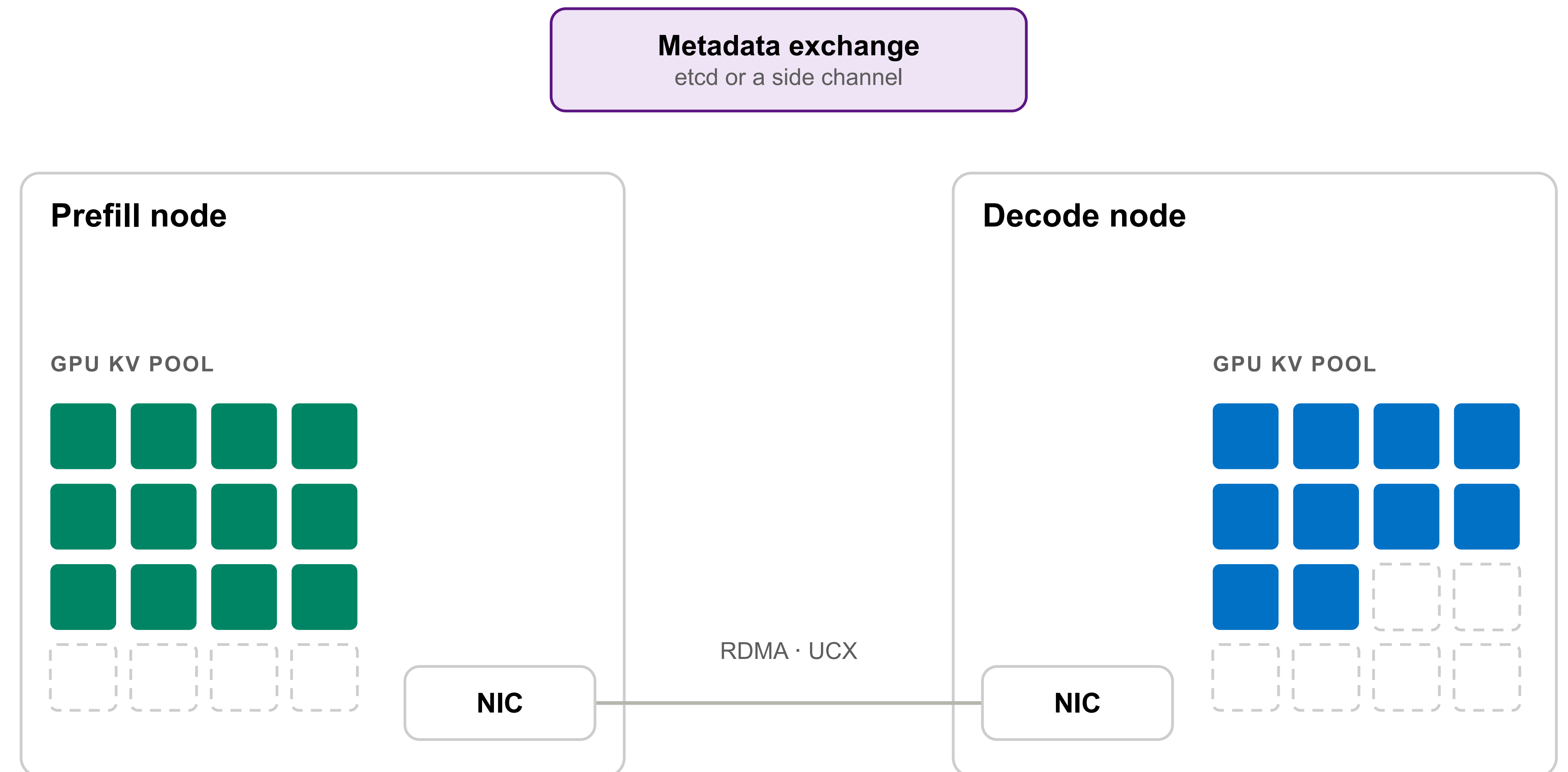
NIXL

Move KV bytes over any fabric, to any memory

NIXL: Anatomy of a KV Transfer

Register once, exchange metadata once, then non-blocking one-sided transfers

- 1 **Create agents**, one per process, each with its transport backends (UCX by default).
- 2 **Register memory** once: each agent registers its whole KV pool. Registration is expensive; transfers are cheap.
- 3 **Exchange metadata** once, on the control path: connection info and remote access keys.
- 4 **Describe the transfer**: READ these remote blocks into these local blocks, tagged with a notification.
- 5 **Post it**. Non-blocking, one-sided, zero-copy: GPU → NIC → NIC → GPU. Decode keeps serving meanwhile.
- 6 **Complete**. Decode sees DONE; prefill gets the notification and frees its blocks.



```
agent = nixl_agent("decode-0")           # transport backends: UCX
agent.register_memory(kv_pool)            # once, whole pool
agent.add_remote_agent(prefill_metadata)  # from etcd / side channel
xfer = agent.initialize_xfer("READ", my_free, their_kv, "prefill-0", b"req-42")
agent.transfer(xfer)                     # returns immediately
while agent.check_xfer_state(xfer) != "DONE":
    run_decode_step()                   # keep serving others
```

06

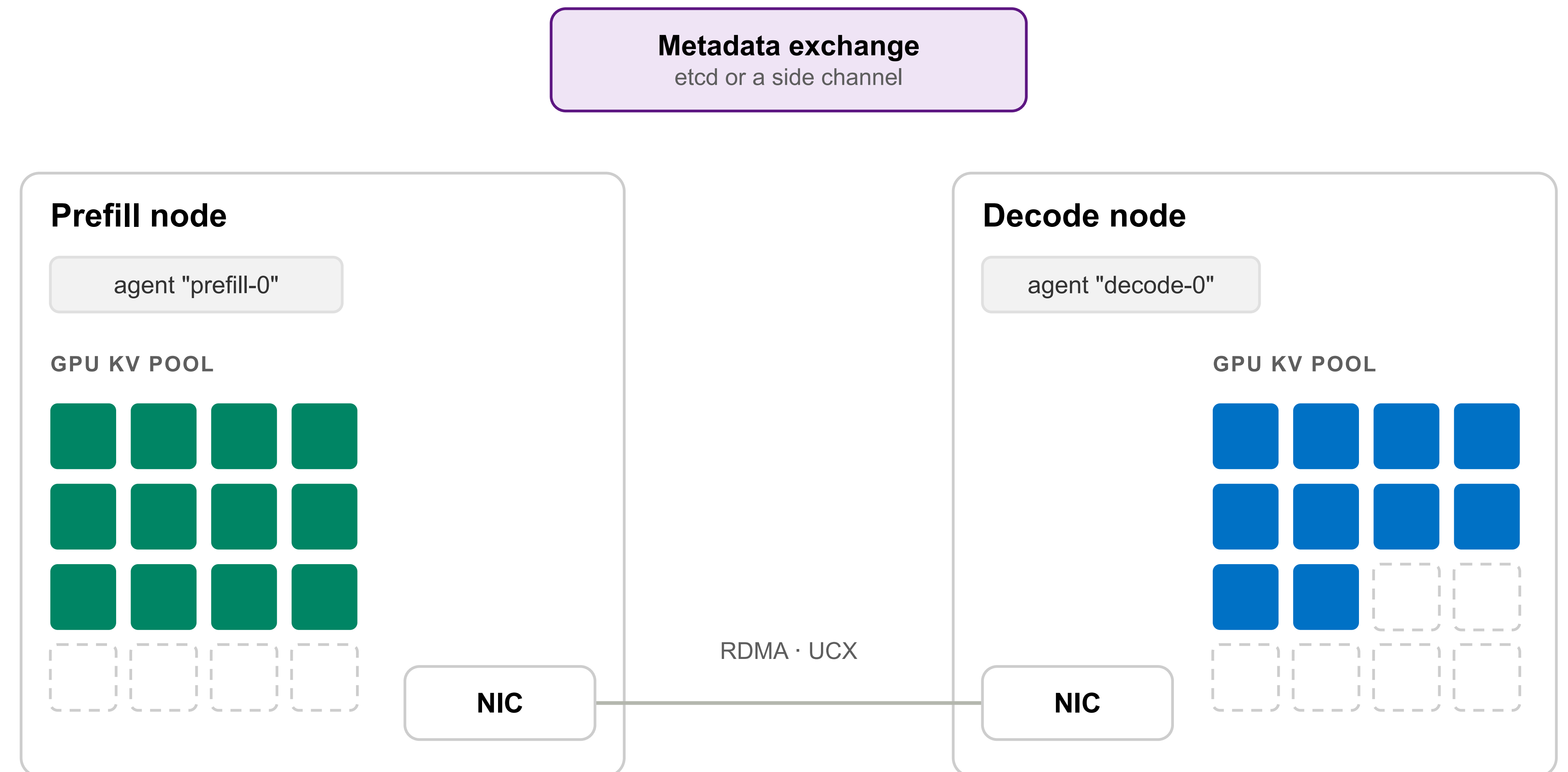
NIXL

Move KV bytes over any fabric, to any memory

NIXL: Anatomy of a KV Transfer

Register once, exchange metadata once, then non-blocking one-sided transfers

- 1 **Create agents**, one per process, each with its transport backends (UCX by default).
- 2 **Register memory** once: each agent registers its whole KV pool. Registration is expensive; transfers are cheap.
- 3 **Exchange metadata** once, on the control path: connection info and remote access keys.
- 4 **Describe the transfer**: READ these remote blocks into these local blocks, tagged with a notification.
- 5 **Post it**. Non-blocking, one-sided, zero-copy: GPU → NIC → NIC → GPU. Decode keeps serving meanwhile.
- 6 **Complete**. Decode sees DONE; prefill gets the notification and frees its blocks.



```
agent = nixl_agent("decode-0")          # transport backends: UCX
agent.register_memory(kv_pool)           # once, whole pool
agent.add_remote_agent(prefill_metadata)  # from etcd / side channel
xfer = agent.initialize_xfer("READ", my_free, their_kv, "prefill-0", b"req-42")
agent.transfer(xfer)                     # returns immediately
while agent.check_xfer_state(xfer) != "DONE":
    run_decode_step()                    # keep serving others
```

06

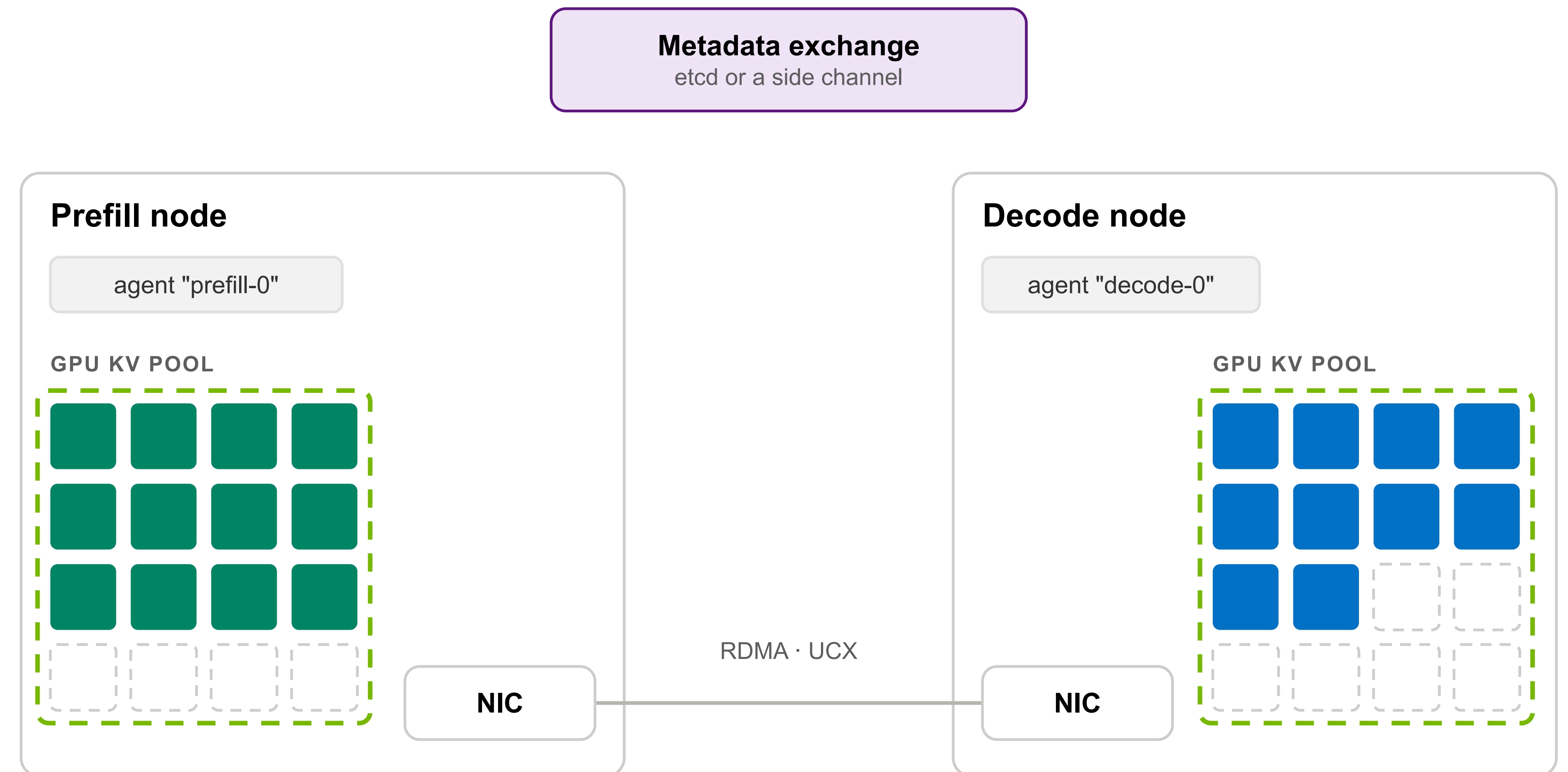
NIXL

Move KV bytes over any fabric, to any memory

NIXL: Anatomy of a KV Transfer

Register once, exchange metadata once, then non-blocking one-sided transfers

- 1 **Create agents**, one per process, each with its transport backends (UCX by default).
- 2 **Register memory** once: each agent registers its whole KV pool. Registration is expensive; transfers are cheap.
- 3 **Exchange metadata** once, on the control path: connection info and remote access keys.
- 4 **Describe the transfer**: READ these remote blocks into these local blocks, tagged with a notification.
- 5 **Post it**. Non-blocking, one-sided, zero-copy: GPU → NIC → NIC → GPU. Decode keeps serving meanwhile.
- 6 **Complete**. Decode sees DONE; prefill gets the notification and frees its blocks.



```
agent = nixl_agent("decode-0")           # transport backends: UCX
agent.register_memory(kv_pool)            # once, whole pool
agent.add_remote_agent(prefill_metadata)  # from etcd / side channel
xfer = agent.initialize_xfer("READ", my_free, their_kv, "prefill-0", b"req-42")
agent.transfer(xfer)                     # returns immediately
while agent.check_xfer_state(xfer) != "DONE":
    run_decode_step()                    # keep serving others
```

06

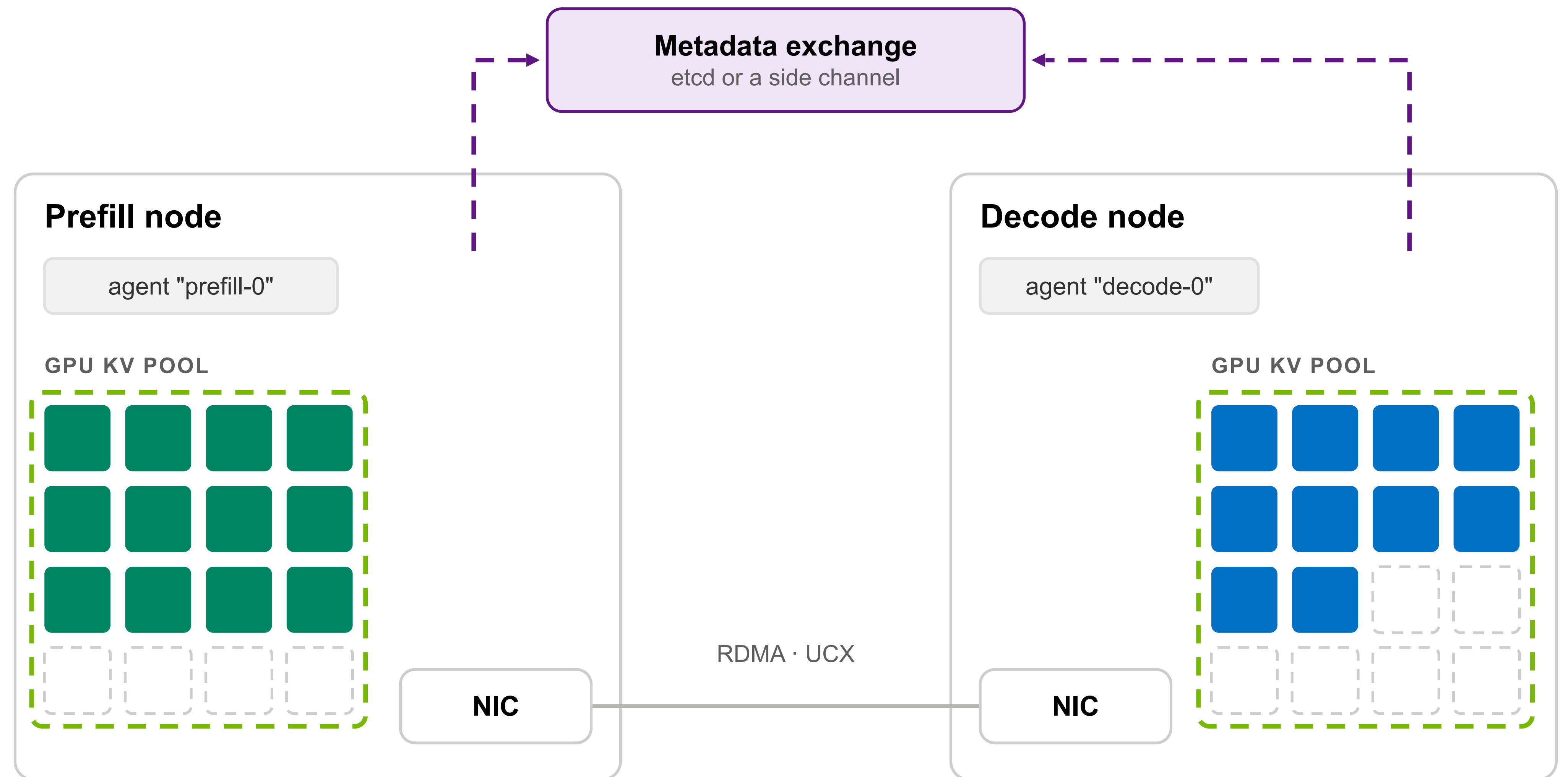
NIXL

Move KV bytes over any fabric, to any memory

NIXL: Anatomy of a KV Transfer

Register once, exchange metadata once, then non-blocking one-sided transfers

- 1 **Create agents**, one per process, each with its transport backends (UCX by default).
- 2 **Register memory** once: each agent registers its whole KV pool. Registration is expensive; transfers are cheap.
- 3 **Exchange metadata** once, on the control path: connection info and remote access keys.
- 4 **Describe the transfer**: READ these remote blocks into these local blocks, tagged with a notification.
- 5 **Post it**. Non-blocking, one-sided, zero-copy: GPU → NIC → NIC → GPU. Decode keeps serving meanwhile.
- 6 **Complete**. Decode sees DONE; prefill gets the notification and frees its blocks.



```
agent = nixl_agent("decode-0")           # transport backends: UCX
agent.register_memory(kv_pool)            # once, whole pool
agent.add_remote_agent(prefill_metadata)  # from etcd / side channel
xfer = agent.initialize_xfer("READ", my_free, their_kv, "prefill-0", b"req-42")
agent.transfer(xfer)                      # returns immediately
while agent.check_xfer_state(xfer) != "DONE":
    run_decode_step()                    # keep serving others
```

06

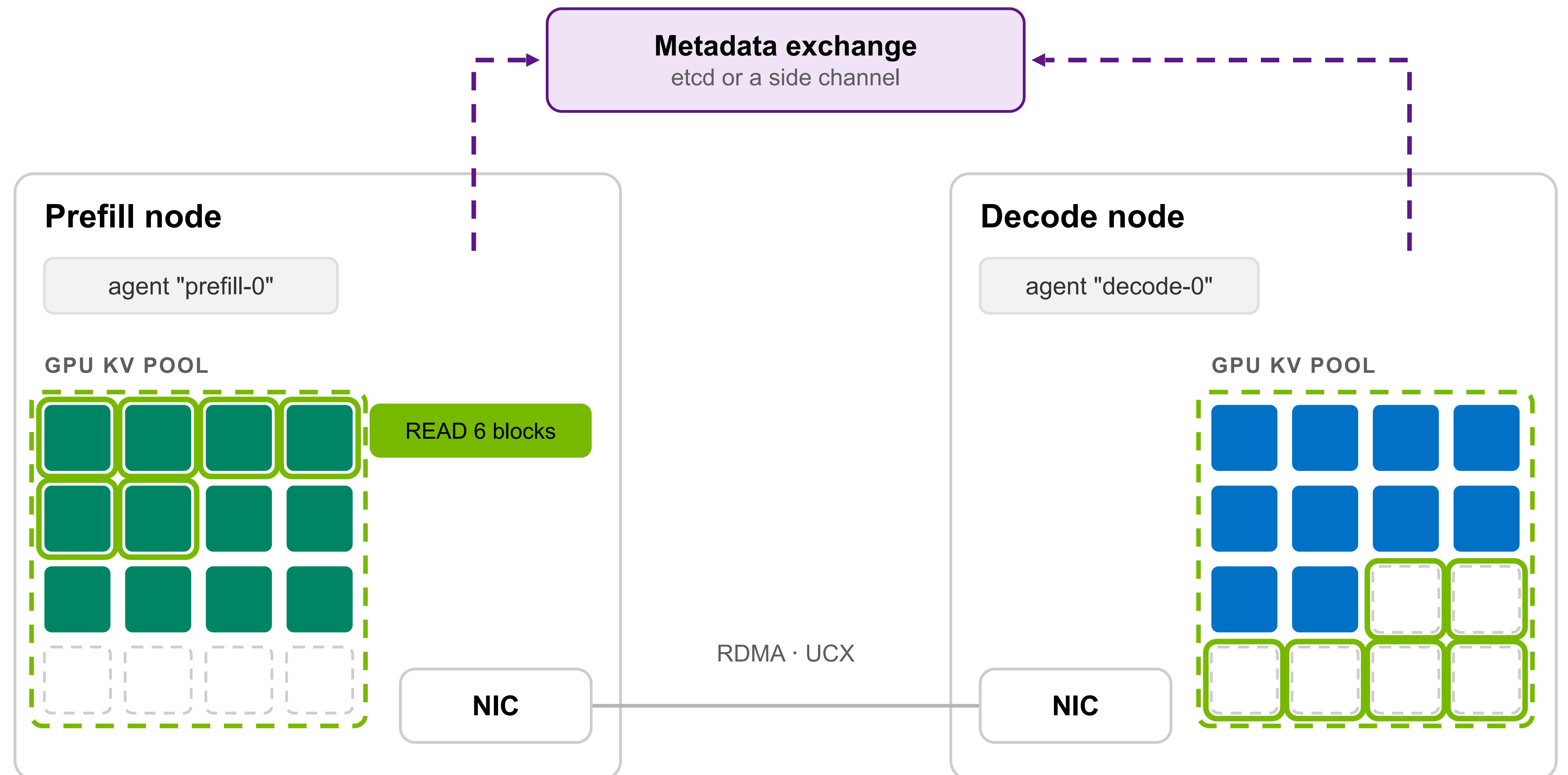
NIXL

Move KV bytes over any fabric, to any memory

NIXL: Anatomy of a KV Transfer

Register once, exchange metadata once, then non-blocking one-sided transfers

- 1 **Create agents**, one per process, each with its transport backends (UCX by default).
- 2 **Register memory** once: each agent registers its whole KV pool. Registration is expensive; transfers are cheap.
- 3 **Exchange metadata** once, on the control path: connection info and remote access keys.
- 4 **Describe the transfer**: READ these remote blocks into these local blocks, tagged with a notification.
- 5 **Post it**. Non-blocking, one-sided, zero-copy: GPU → NIC → NIC → GPU. Decode keeps serving meanwhile.
- 6 **Complete**. Decode sees DONE; prefill gets the notification and frees its blocks.



```
agent = nixl_agent("decode-0")           # transport backends: UCX
agent.register_memory(kv_pool)            # once, whole pool
agent.add_remote_agent(prefill_metadata)  # from etcd / side channel
xfer = agent.initialize_xfer("READ", my_free, their_kv, "prefill-0", b"req-42")
agent.transfer(xfer)                      # returns immediately
while agent.check_xfer_state(xfer) != "DONE":
    run_decode_step()                    # keep serving others
```

06

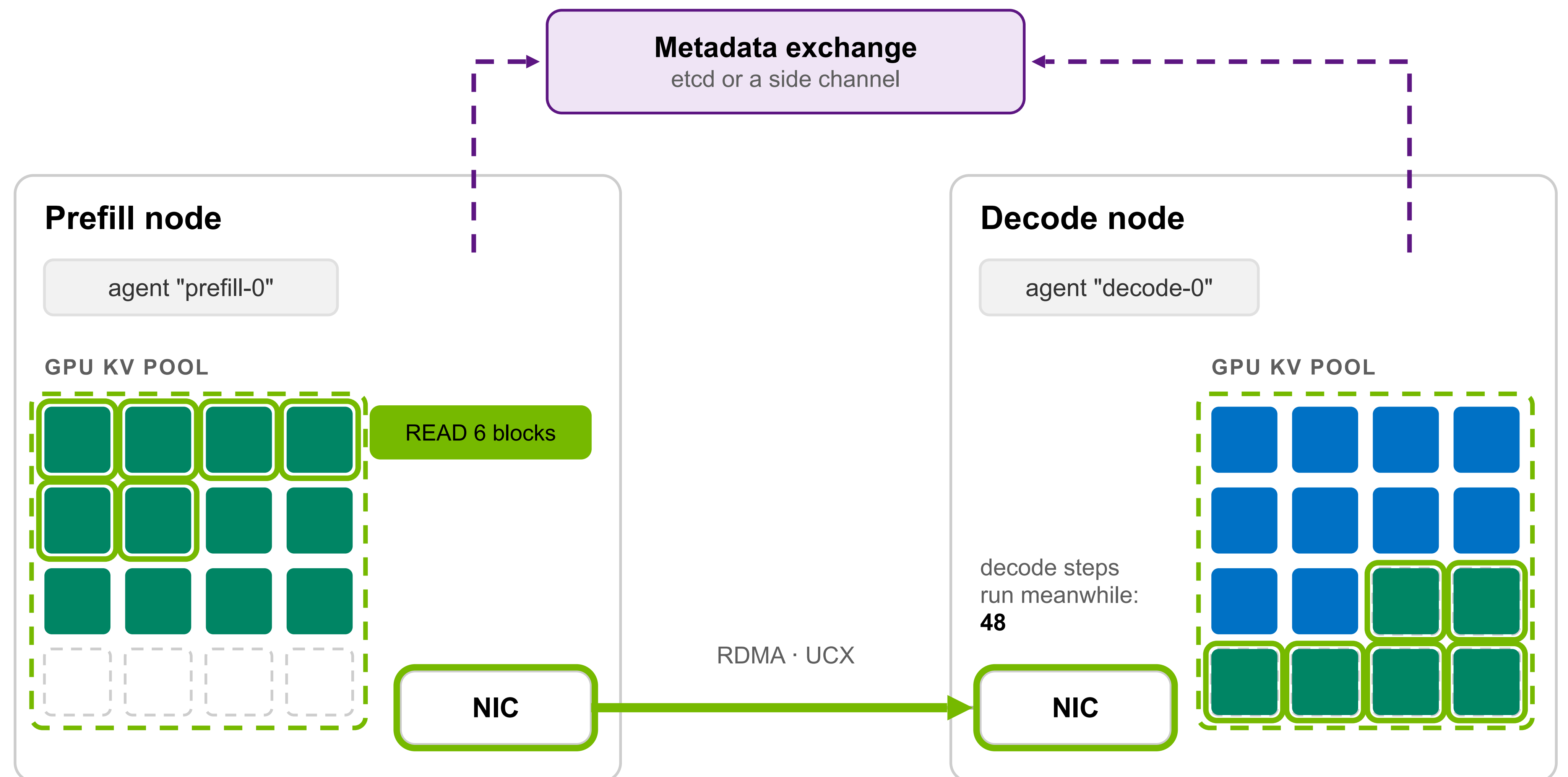
NIXL

Move KV bytes over any fabric, to any memory

NIXL: Anatomy of a KV Transfer

Register once, exchange metadata once, then non-blocking one-sided transfers

- 1 **Create agents**, one per process, each with its transport backends (UCX by default).
- 2 **Register memory** once: each agent registers its whole KV pool. Registration is expensive; transfers are cheap.
- 3 **Exchange metadata** once, on the control path: connection info and remote access keys.
- 4 **Describe the transfer**: READ these remote blocks into these local blocks, tagged with a notification.
- 5 **Post it.** Non-blocking, one-sided, zero-copy: GPU → NIC → NIC → GPU. Decode keeps serving meanwhile.
- 6 **Complete.** Decode sees DONE; prefill gets the notification and frees its blocks.



```
agent = nixl_agent("decode-0")           # transport backends: UCX
agent.register_memory(kv_pool)            # once, whole pool
agent.add_remote_agent(prefill_metadata)  # from etcd / side channel
xfer = agent.initialize_xfer("READ", my_free, their_kv, "prefill-0", b"req-42")
agent.transfer(xfer)                      # returns immediately
while agent.check_xfer_state(xfer) != "DONE":
    run_decode_step()                     # keep serving others
```

06

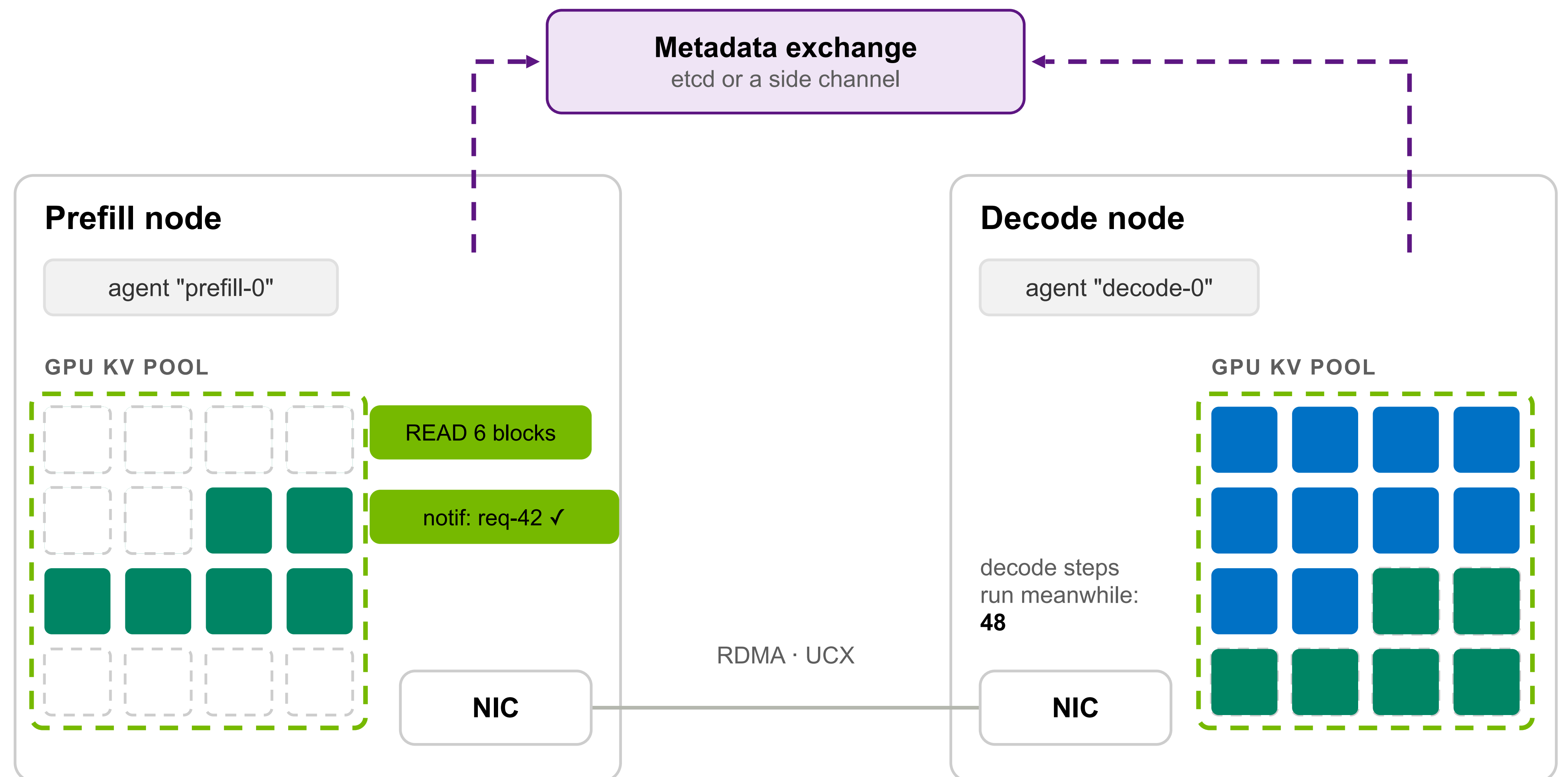
NIXL

Move KV bytes over any fabric, to any memory

NIXL: Anatomy of a KV Transfer

Register once, exchange metadata once, then non-blocking one-sided transfers

- 1 **Create agents**, one per process, each with its transport backends (UCX by default).
- 2 **Register memory** once: each agent registers its whole KV pool. Registration is expensive; transfers are cheap.
- 3 **Exchange metadata** once, on the control path: connection info and remote access keys.
- 4 **Describe the transfer**: READ these remote blocks into these local blocks, tagged with a notification.
- 5 **Post it**. Non-blocking, one-sided, zero-copy: GPU → NIC → NIC → GPU. Decode keeps serving meanwhile.
- 6 **Complete**. Decode sees DONE; prefill gets the notification and frees its blocks.



```
agent = nixl_agent("decode-0")           # transport backends: UCX
agent.register_memory(kv_pool)            # once, whole pool
agent.add_remote_agent(prefill_metadata)  # from etcd / side channel
xfer = agent.initialize_xfer("READ", my_free, their_kv, "prefill-0", b"req-42")
agent.transfer(xfer)                      # returns immediately

while agent.check_xfer_state(xfer) != "DONE":
    run_decode_step()                     # keep serving others
```

06

NIXL

Move KV bytes over any fabric, to any memory

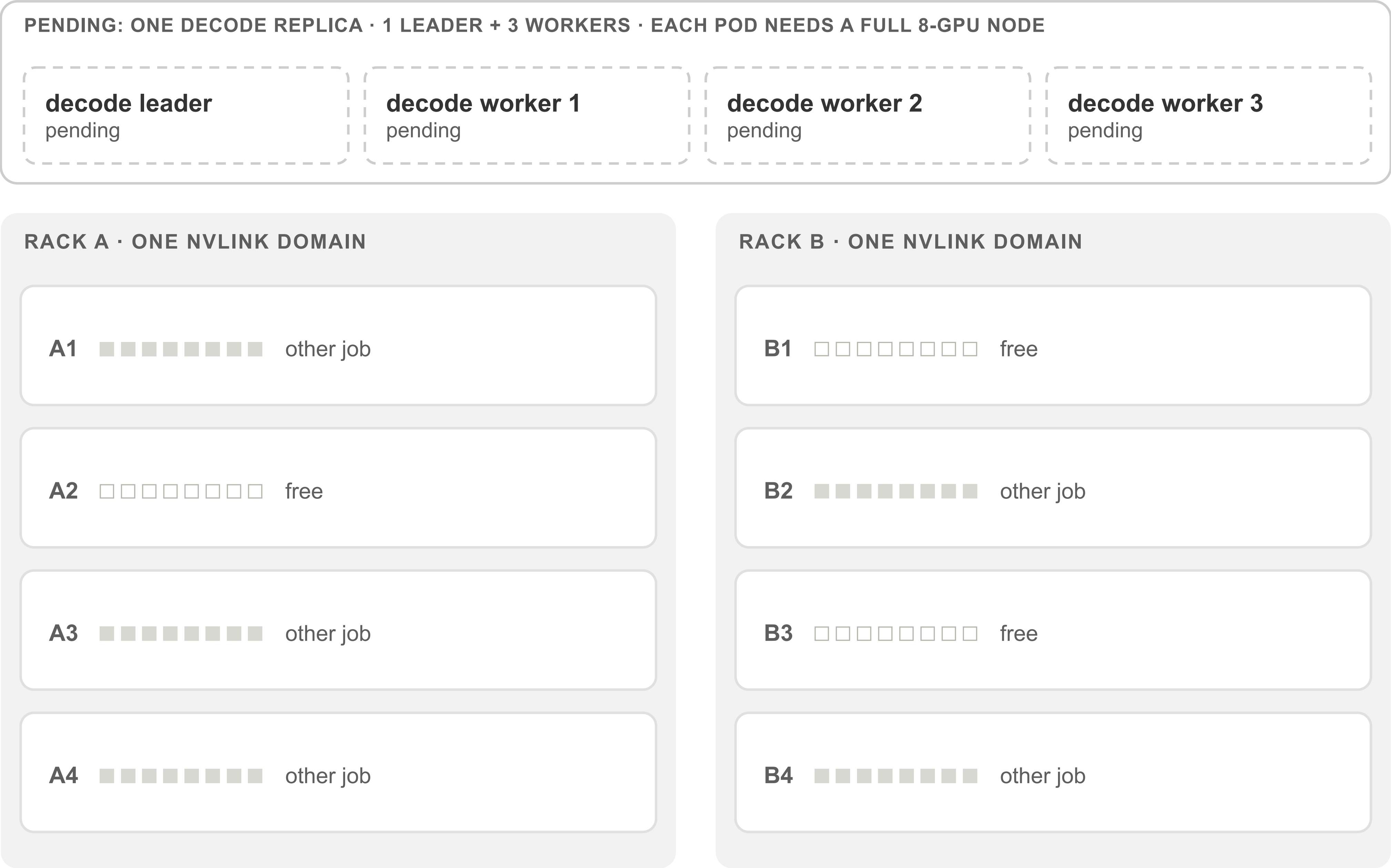
Grove

Scheduling an inference system, not a pod

A Kubernetes API (custom resources plus an operator) that describes a multi-role, multi-node inference deployment as one object, with **gang scheduling**, **topology-aware placement**, **startup ordering** and **multi-level autoscaling**.

WHY IT MATTERS

- **All or nothing.** A model replica sharded over 4 nodes is useless with 3; the 3 that started just hold GPUs.
- **Placement matters.** Pods of one replica, and prefill next to decode, belong in the same NVLink domain or rack.
- **Order matters.** Leaders and workers start in a defined sequence.
- **The scaling unit is a group:** add a whole leader-plus-workers unit, at different ratios for prefill and decode.



The replica needs 4 whole nodes. Only 3 are free, and they sit in different racks.

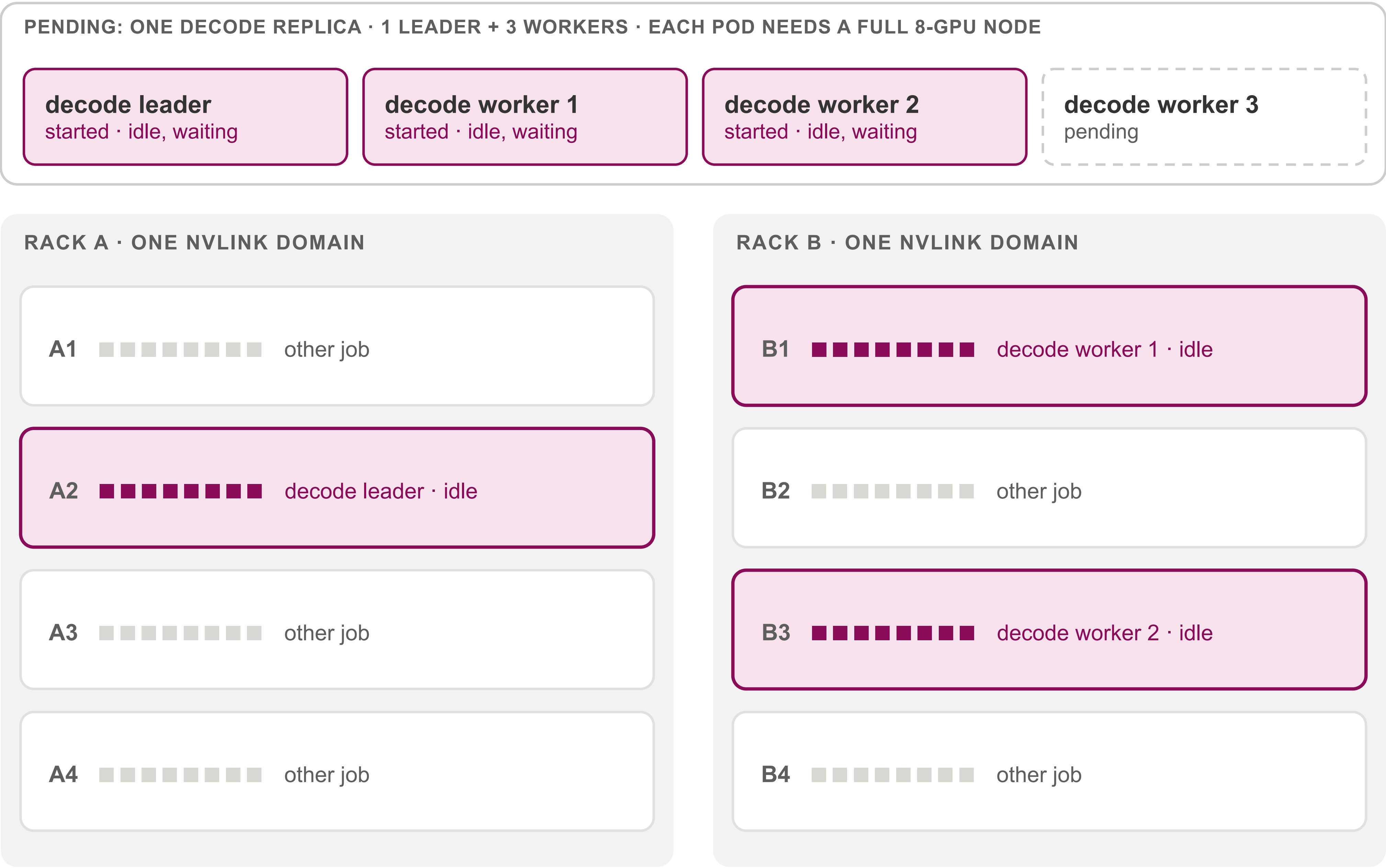
Grove

Scheduling an inference system, not a pod

A Kubernetes API (custom resources plus an operator) that describes a multi-role, multi-node inference deployment as one object, with **gang scheduling**, **topology-aware placement**, **startup ordering** and **multi-level autoscaling**.

WHY IT MATTERS

- **All or nothing.** A model replica sharded over 4 nodes is useless with 3; the 3 that started just hold GPUs.
- **Placement matters.** Pods of one replica, and prefill next to decode, belong in the same NVLink domain or rack.
- **Order matters.** Leaders and workers start in a defined sequence.
- **The scaling unit is a group:** add a whole leader-plus-workers unit, at different ratios for prefill and decode.



Pod-by-pod scheduling: 3 of 4 pods start and the 4th never fits. 24 GPUs sit allocated and idle, split across two racks.

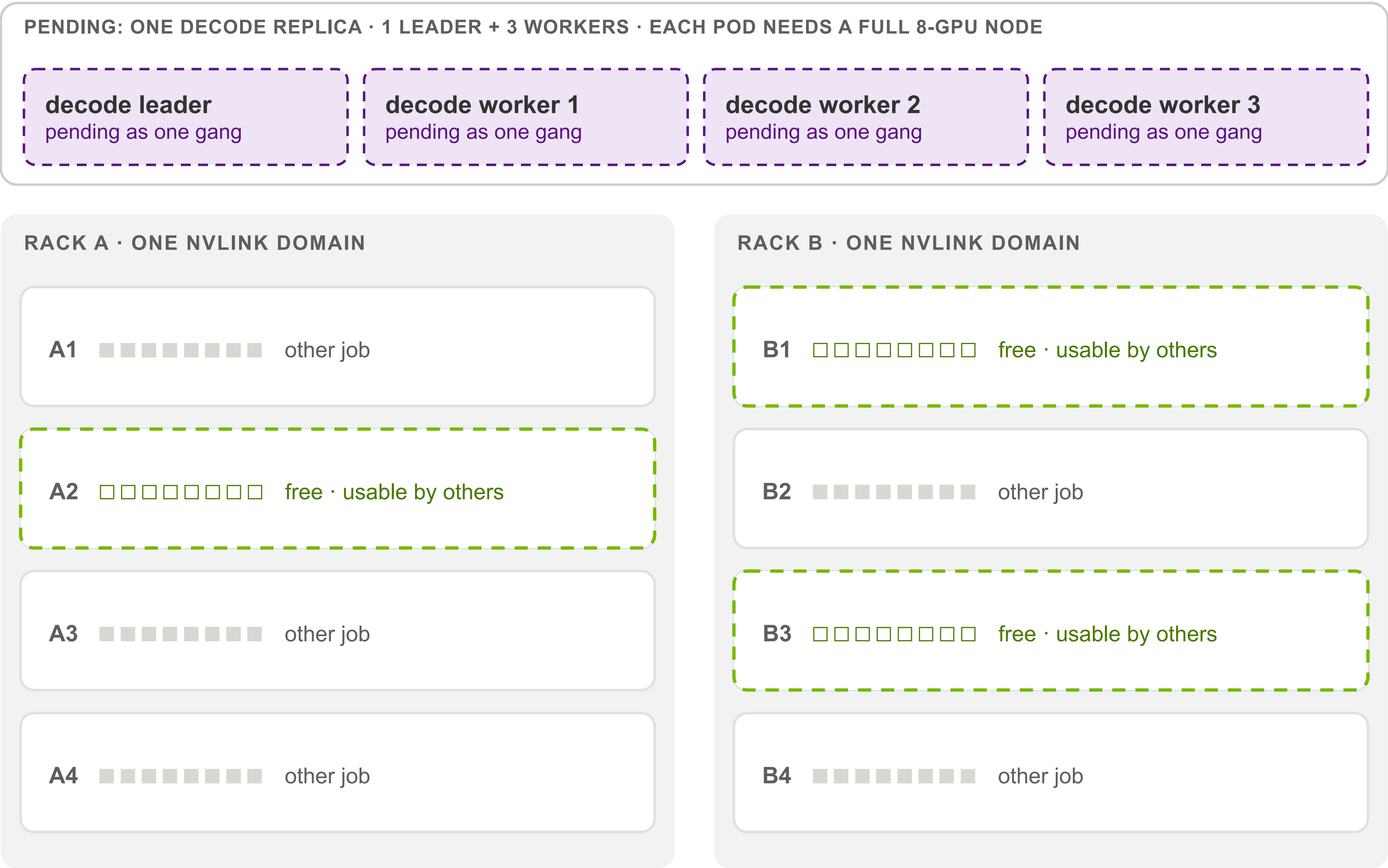
Grove

Scheduling an inference system, not a pod

A Kubernetes API (custom resources plus an operator) that describes a multi-role, multi-node inference deployment as one object, with **gang scheduling**, **topology-aware placement**, **startup ordering** and **multi-level autoscaling**.

WHY IT MATTERS

- **All or nothing.** A model replica sharded over 4 nodes is useless with 3; the 3 that started just hold GPUs.
- **Placement matters.** Pods of one replica, and prefill next to decode, belong in the same NVLink domain or rack.
- **Order matters.** Leaders and workers start in a defined sequence.
- **The scaling unit is a group:** add a whole leader-plus-workers unit, at different ratios for prefill and decode.



Gang scheduling: the replica waits as one unit. Nothing is stranded, and the 3 free nodes stay usable by other work.

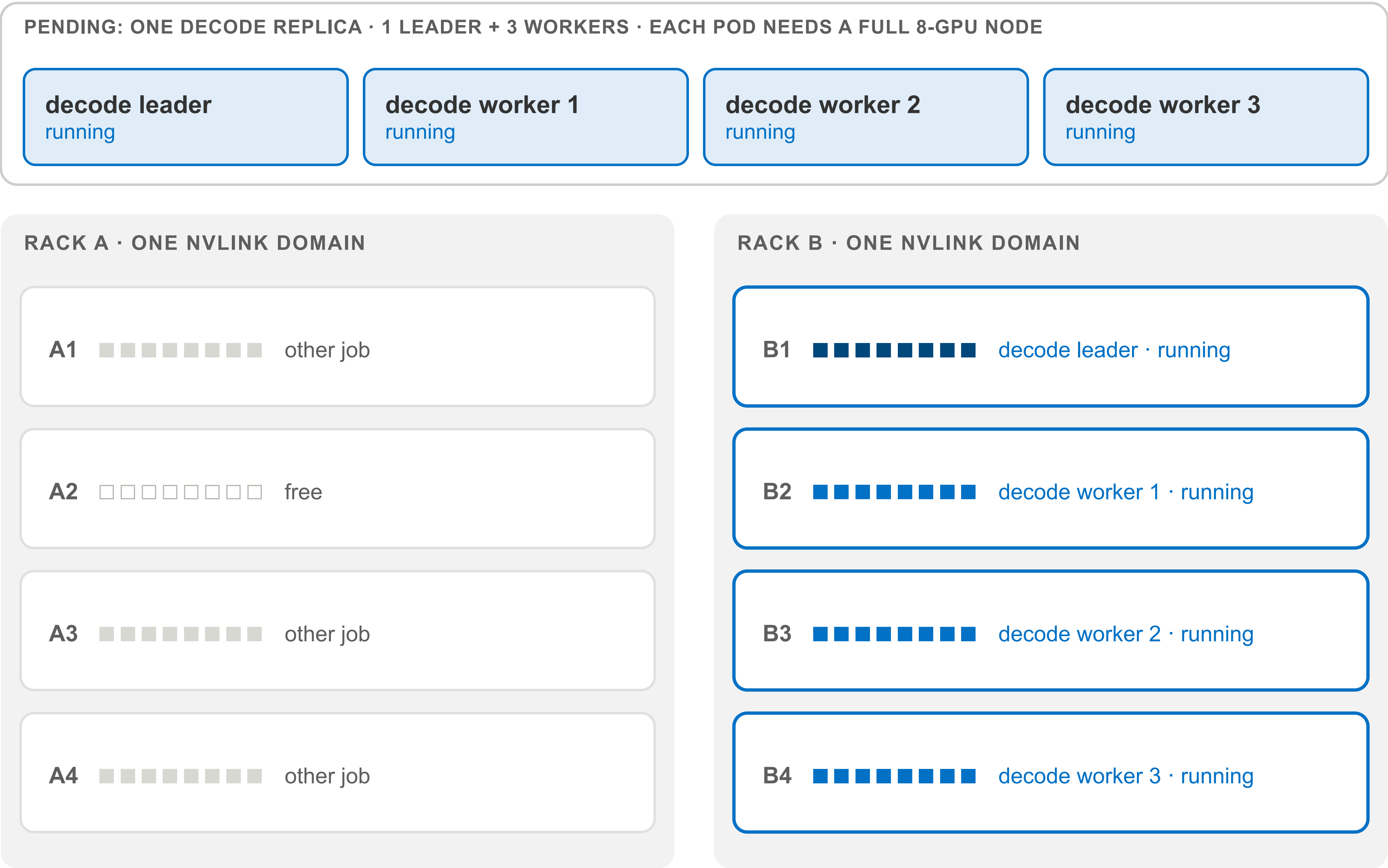
Grove

Scheduling an inference system, not a pod

A Kubernetes API (custom resources plus an operator) that describes a multi-role, multi-node inference deployment as one object, with **gang scheduling**, **topology-aware placement**, **startup ordering** and **multi-level autoscaling**.

WHY IT MATTERS

- **All or nothing.** A model replica sharded over 4 nodes is useless with 3; the 3 that started just hold GPUs.
- **Placement matters.** Pods of one replica, and prefill next to decode, belong in the same NVLink domain or rack.
- **Order matters.** Leaders and workers start in a defined sequence.
- **The scaling unit is a group:** add a whole leader-plus-workers unit, at different ratios for prefill and decode.



Topology-aware placement: when Rack B frees up, all 4 pods start together inside one NVLink domain.

Fault tolerance

Ensure detection, resilience, and recovery

High availability

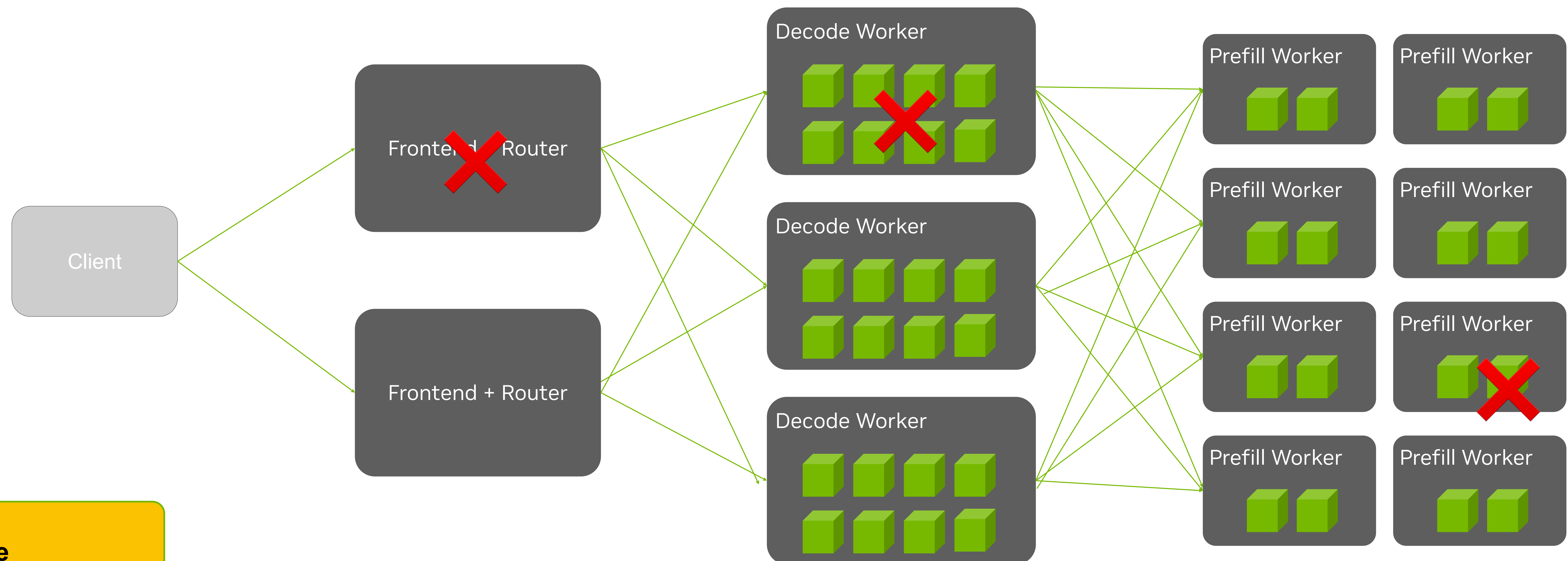
Automatic restart with shared router state

Request cancellation

Cancel request to avoid computation

Request migration

Migrate partial response at token level to avoid Decode



07

Grove

Gang-schedule and place pods on Kubernetes

Dynamo Snapshot

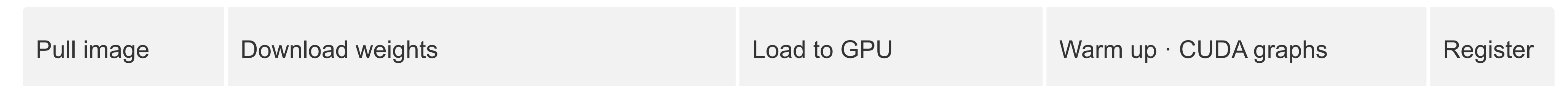
Skip the cold start: restore a warm worker instead of rebuilding it

Checkpoint and restore for GPU inference workers on Kubernetes. Start one worker the slow way, freeze its fully initialized process (CPU **and** GPU state), and restore new replicas from that image.

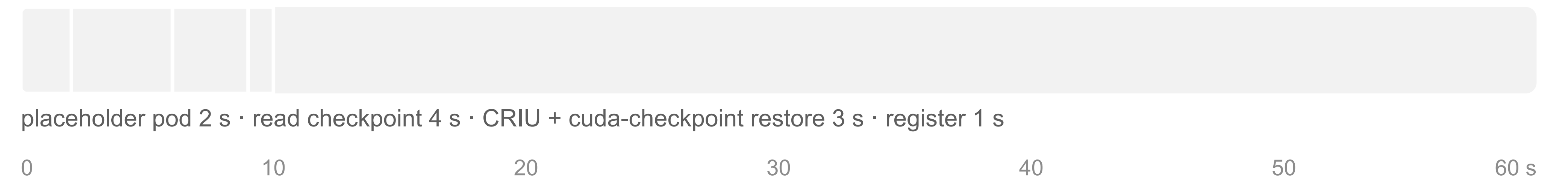
WHY IT MATTERS

- **Cold start takes minutes:** pull the image, download weights, load them to GPU, warm up kernels, capture CUDA graphs, join the cluster.
- **GPUs are allocated but idle** the whole time.
- **Autoscaling can't catch a spike** if each new worker needs minutes. The planner's decisions only help once the workers are up.
- **Agentic sessions** largely wait on user inputs and keeping state resident in memory is expensive.

Cold start



Restore from snapshot



Totals from the Dynamo docs (~1 min cold vs. ~10 s warm, depending on storage and GPU); the split between phases is illustrative.

Read the paper

<https://arxiv.org/pdf/2609.25451>

08

**Snapshot
(checkpoint/restore)**

Start new workers in seconds

Dynamo Snapshot

Skip the cold start: restore a warm worker instead of rebuilding it

Checkpoint and restore for GPU inference workers on Kubernetes. Start one worker the slow way, freeze its fully initialized process (CPU **and** GPU state), and restore new replicas from that image.

WHY IT MATTERS

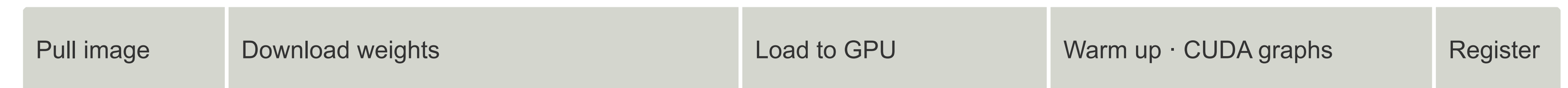
- **Cold start takes minutes:** pull the image, download weights, load them to GPU, warm up kernels, capture CUDA graphs, join the cluster.
- **GPUs are allocated but idle** the whole time.
- **Autoscaling can't catch a spike** if each new worker needs minutes. The planner's decisions only help once the workers are up.
- **Agentic sessions** largely wait on user inputs and keeping state resident in memory is expensive.

Read the paper

<https://arxiv.org/pdf/2609.25451>

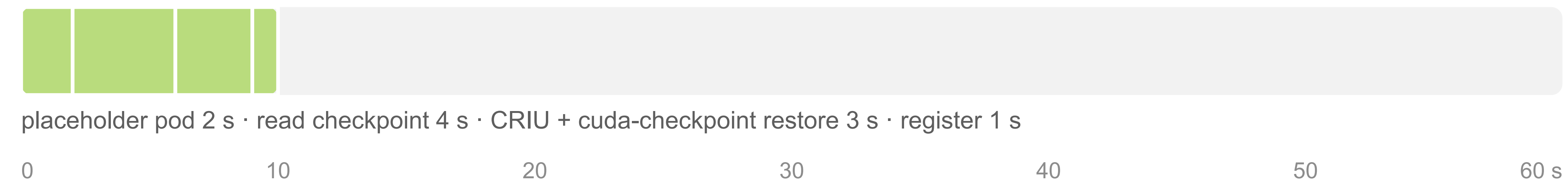
Cold start

≈ 60 s



Restore from snapshot

≈ 10 s



Totals from the Dynamo docs (~1 min cold vs. ~10 s warm, depending on storage and GPU); the split between phases is illustrative.

Dynamo Snapshot

Skip the cold start: restore a warm worker instead of rebuilding it

Checkpoint and restore for GPU inference workers on Kubernetes. Start one worker the slow way, freeze its fully initialized process (CPU **and** GPU state), and restore new replicas from that image.

WHY IT MATTERS

- **Cold start takes minutes:** pull the image, download weights, load them to GPU, warm up kernels, capture CUDA graphs, join the cluster.
- **GPUs are allocated but idle** the whole time.
- **Autoscaling can't catch a spike** if each new worker needs minutes. The planner's decisions only help once the workers are up.
- **Agentic sessions** largely wait on user inputs and keeping state resident in memory is expensive.

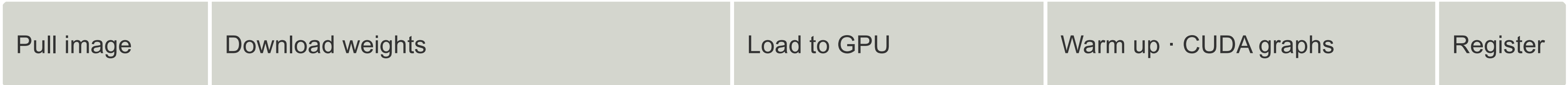
Read the paper
<https://arxiv.org/pdf/2609.25451>

08

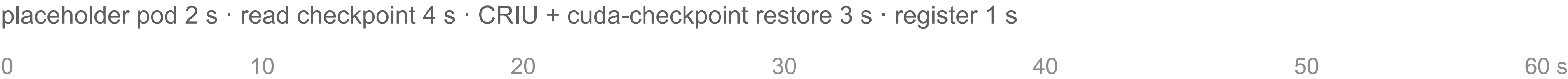
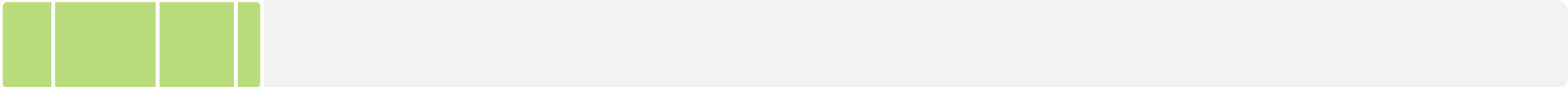
Snapshot
(checkpoint/restore)

Start new workers in seconds

Cold start ≈ 60 s



Restore from snapshot ≈ 10 s



Totals from the Dynamo docs (~1 min cold vs. ~10 s warm, depending on storage and GPU); the split between phases is illustrative.

RESTORE TIME, MEASURED ·

Table 3. Recovery latency (seconds) and speedup over warm restart for different models and frameworks.

Configuration	Qwen3.8-27B	DSV4-Flash	GLM-5.2	DSV4-Pro
vLLM				
Warm restart	106.4 (1.0×	109.5 (1.0×	144.4 (1.0×	156.5 (1.0×
GMS	80.0 (1.3×	85.3 (1.3×	82.0 (1.8×	91.9 (1.7×
Snapshot	40.5 (2.6×	60.0 (1.8×	131.7 (1.1×	189.7 (0.8×
Snapshot + cold GMS	21.9 (4.9×	26.8 (4.1×	26.8 (5.4×	50.0 (3.1×
Snapshot + warm GMS	16.6 (6.4×	24.1 (4.5×	24.9 (5.8×	25.7 (6.1×
GMS + Shadow Engine	4.6 (23.1×	5.6 (19.6×	5.6 (25.8×	5.4 (29.0×
SGLang				
Warm restart	79.6 (1.0×	95.7 (1.0×	144.6 (1.0×	151.2 (1.0×
Snapshot + cold GMS	37.1 (2.1×	31.1 (3.1×	38.0 (3.8×	55.2 (2.7×
Snapshot + warm GMS	33.3 (2.4×	31.1 (3.1×	35.3 (4.1×	29.8 (5.1×
GMS + Shadow Engine	6.2 (12.8×	6.2 (15.4×	5.7 (25.4×	5.3 (28.5×

Snapshot: Checkpoint Once, Restore Many

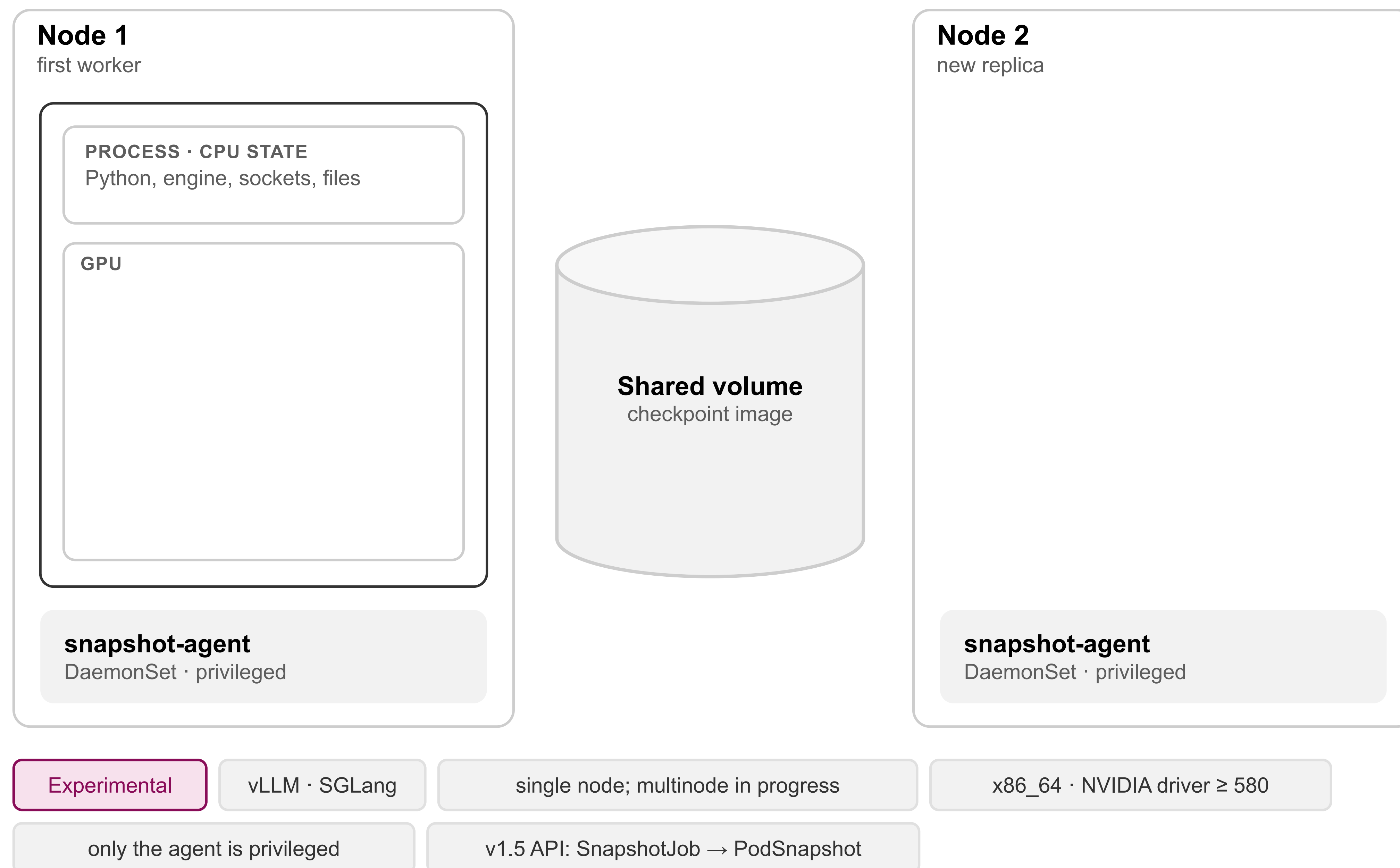
cuda-checkpoint + CRIU, driven by a node agent; restore on any node

CHECKPOINT · ONCE

- 1 **Initialize** one worker the slow way: load weights, warm up, capture CUDA graphs.
- 2 **Quiesce.** Release KV-cache memory (keeping the virtual addresses the CUDA graphs use) and signal "ready" before joining the cluster.
- 3 **Dump.** The node's snapshot agent runs cuda-checkpoint (GPU state → host) and CRIU (process tree and files), and writes the image to shared storage.

RESTORE · EVERY NEW REPLICA

- 4 **Place** a new pod from a lightweight placeholder image, on any node.
- 5 **Restore** the filesystem, the process (CRIU) and its GPU state (cuda-checkpoint).
- 6 **Resume** inside the waiting loop: re-allocate KV cache, register, take traffic.



Read the paper on advanced techniques

<https://arxiv.org/pdf/2609.25451>

Why the waiting loop: the worker writes "ready" before it registers, then polls for a restore signal. A restored copy wakes up at that exact instruction and registers as a fresh worker. Live TCP connections can't be checkpointed, which is why multi-node workers require special handling (read paper)

08

Snapshot
(checkpoint/restore)

Start new workers in seconds

Snapshot: Checkpoint Once, Restore Many

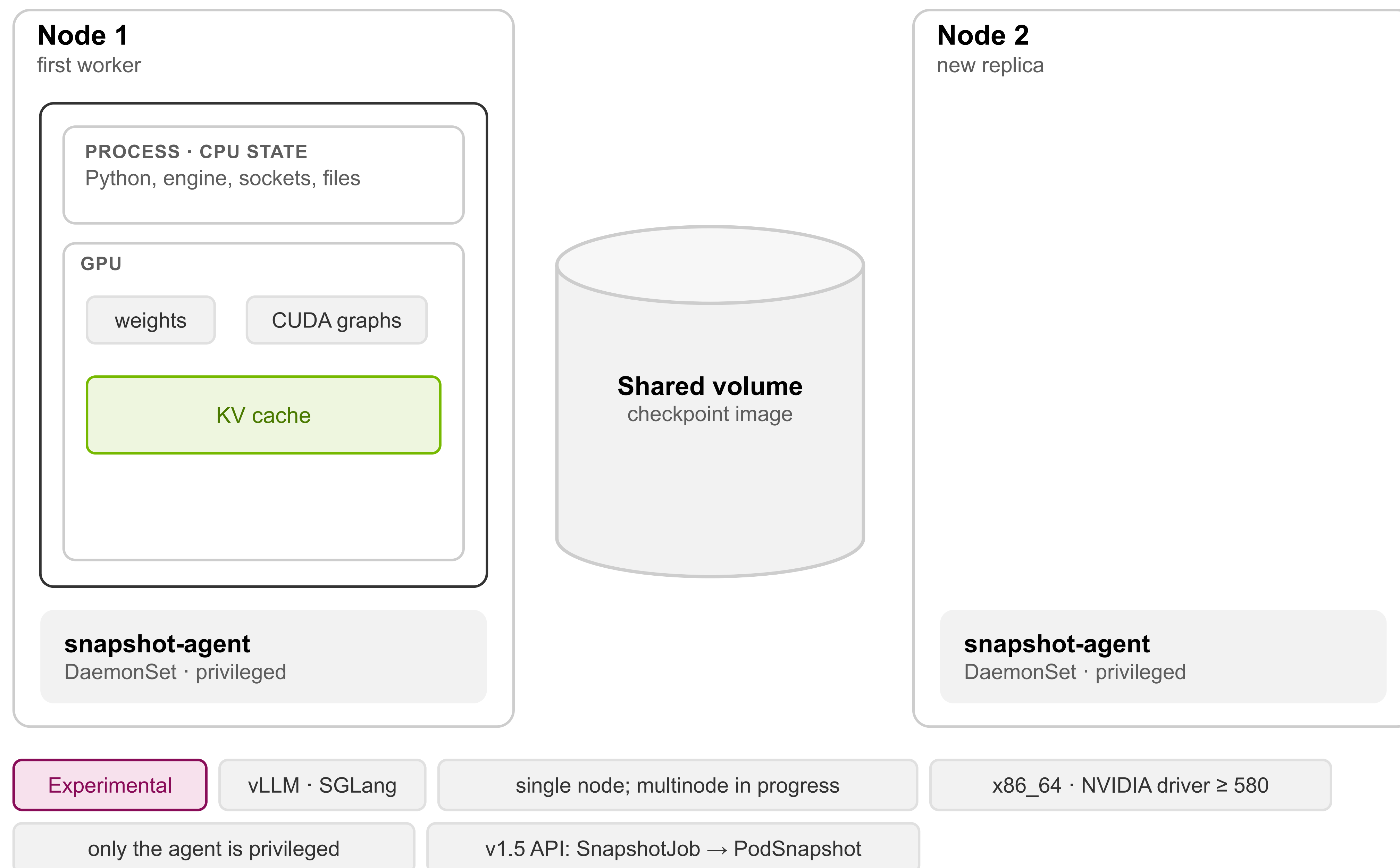
cuda-checkpoint + CRIU, driven by a node agent; restore on any node

CHECKPOINT · ONCE

- 1 **Initialize** one worker the slow way: load weights, warm up, capture CUDA graphs.
- 2 **Quiesce.** Release KV-cache memory (keeping the virtual addresses the CUDA graphs use) and signal "ready" before joining the cluster.
- 3 **Dump.** The node's snapshot agent runs cuda-checkpoint (GPU state → host) and CRIU (process tree and files), and writes the image to shared storage.

RESTORE · EVERY NEW REPLICA

- 4 **Place** a new pod from a lightweight placeholder image, on any node.
- 5 **Restore** the filesystem, the process (CRIU) and its GPU state (cuda-checkpoint).
- 6 **Resume** inside the waiting loop: re-allocate KV cache, register, take traffic.



Read the paper on advanced techniques

<https://arxiv.org/pdf/2609.25451>

08

Snapshot
(checkpoint/restore)

Start new workers in seconds

Snapshot: Checkpoint Once, Restore Many

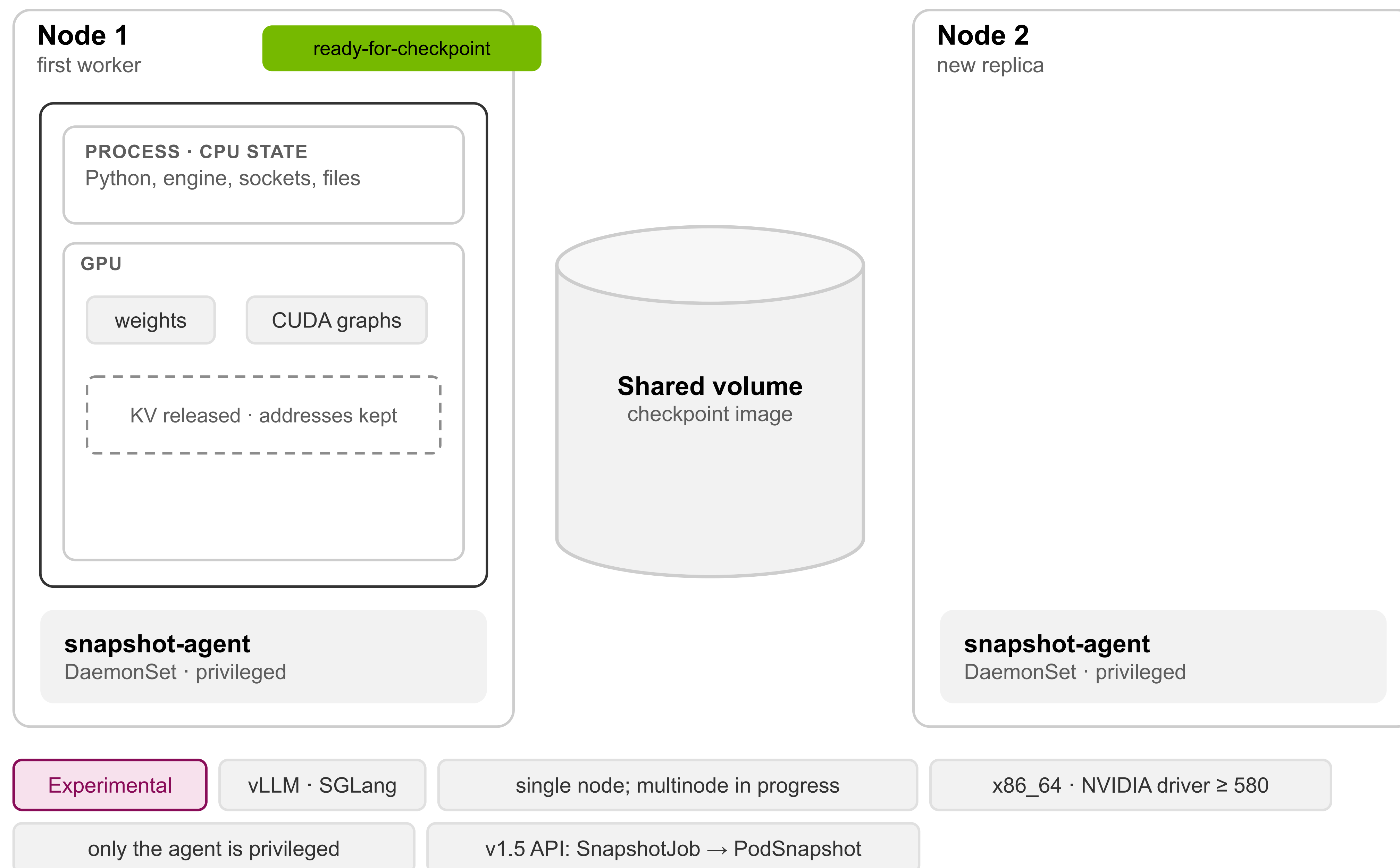
cuda-checkpoint + CRIU, driven by a node agent; restore on any node

CHECKPOINT · ONCE

- 1 **Initialize** one worker the slow way: load weights, warm up, capture CUDA graphs.
- 2 **Quiesce.** Release KV-cache memory (keeping the virtual addresses the CUDA graphs use) and signal "ready" before joining the cluster.
- 3 **Dump.** The node's snapshot agent runs cuda-checkpoint (GPU state → host) and CRIU (process tree and files), and writes the image to shared storage.

RESTORE · EVERY NEW REPLICA

- 4 **Place** a new pod from a lightweight placeholder image, on any node.
- 5 **Restore** the filesystem, the process (CRIU) and its GPU state (cuda-checkpoint).
- 6 **Resume** inside the waiting loop: re-allocate KV cache, register, take traffic.



Read the paper on advanced techniques

<https://arxiv.org/pdf/2609.25451>

08

**Snapshot
(checkpoint/restore)**

Start new workers in seconds

Snapshot: Checkpoint Once, Restore Many

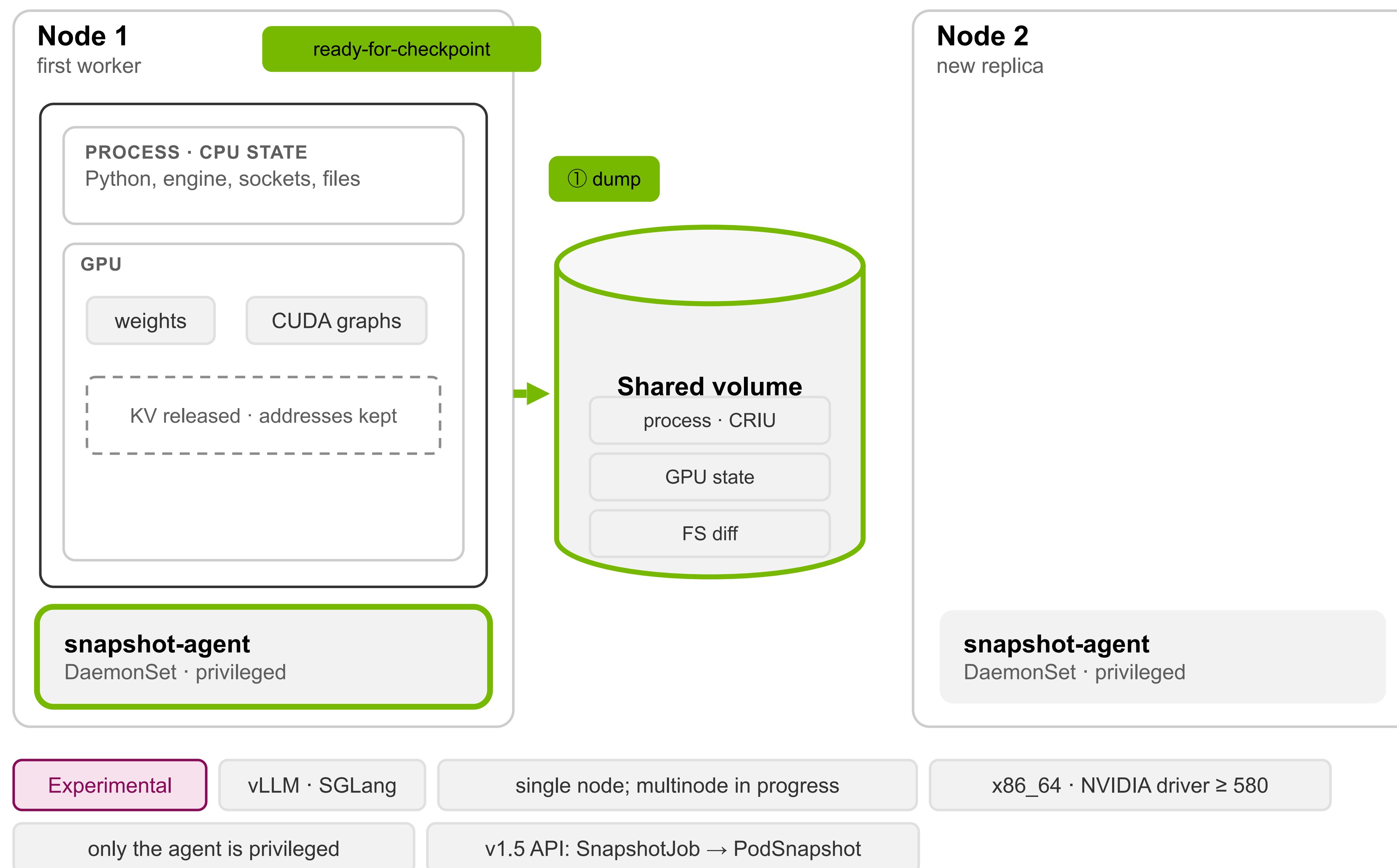
cuda-checkpoint + CRIU, driven by a node agent; restore on any node

CHECKPOINT · ONCE

- 1 **Initialize** one worker the slow way: load weights, warm up, capture CUDA graphs.
- 2 **Quiesce.** Release KV-cache memory (keeping the virtual addresses the CUDA graphs use) and signal "ready" before joining the cluster.
- 3 **Dump.** The node's snapshot agent runs cuda-checkpoint (GPU state → host) and CRIU (process tree and files), and writes the image to shared storage.

RESTORE · EVERY NEW REPLICA

- 4 **Place** a new pod from a lightweight placeholder image, on any node.
- 5 **Restore** the filesystem, the process (CRIU) and its GPU state (cuda-checkpoint).
- 6 **Resume** inside the waiting loop: re-allocate KV cache, register, take traffic.



Read the paper on advanced techniques

<https://arxiv.org/pdf/2609.25451>

08

**Snapshot
(checkpoint/restore)**

Start new workers in seconds

Snapshot: Checkpoint Once, Restore Many

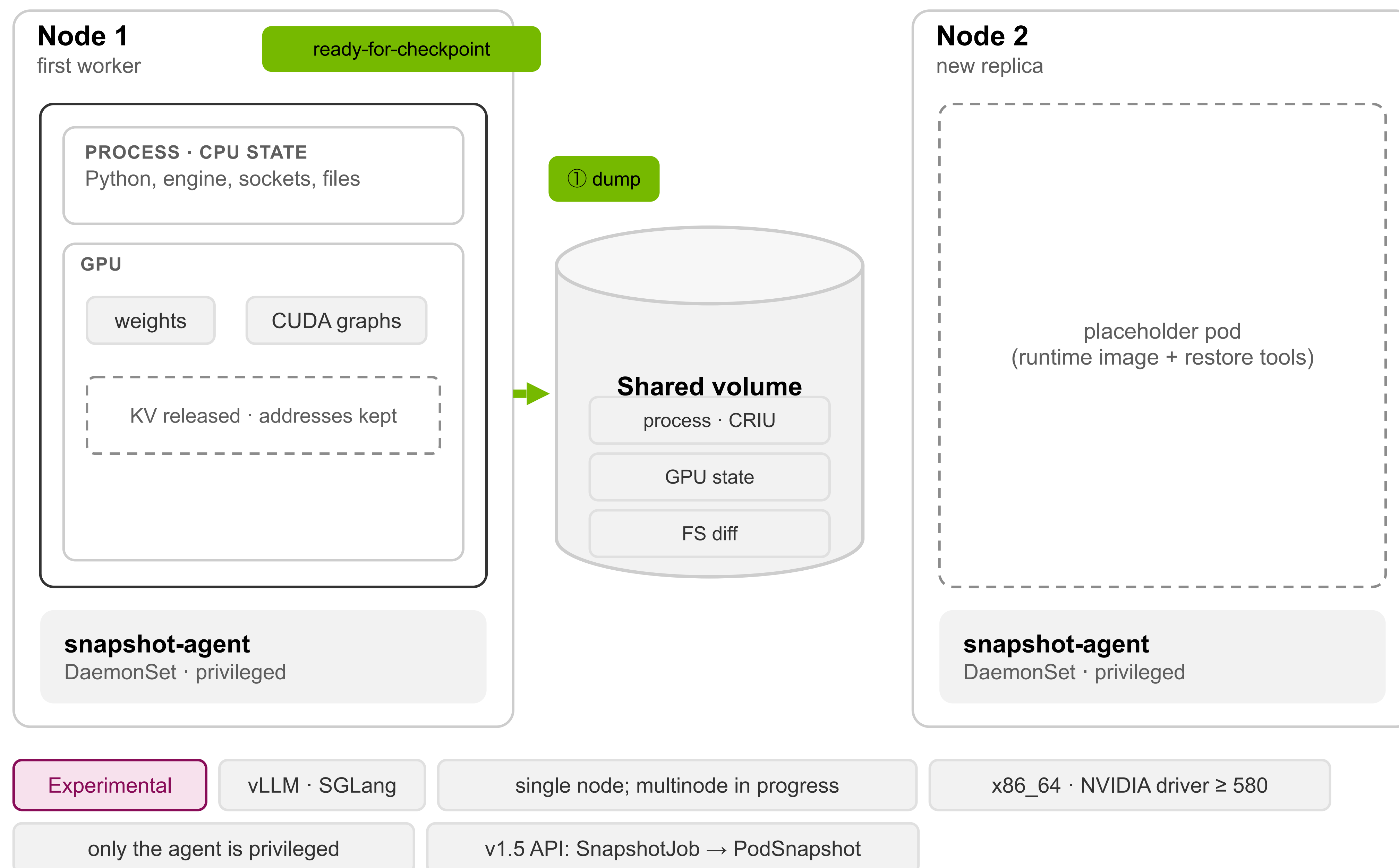
cuda-checkpoint + CRIU, driven by a node agent; restore on any node

CHECKPOINT · ONCE

- 1 **Initialize** one worker the slow way: load weights, warm up, capture CUDA graphs.
- 2 **Quiesce.** Release KV-cache memory (keeping the virtual addresses the CUDA graphs use) and signal "ready" before joining the cluster.
- 3 **Dump.** The node's snapshot agent runs cuda-checkpoint (GPU state → host) and CRIU (process tree and files), and writes the image to shared storage.

RESTORE · EVERY NEW REPLICA

- 4 **Place** a new pod from a lightweight placeholder image, on any node.
- 5 **Restore** the filesystem, the process (CRIU) and its GPU state (cuda-checkpoint).
- 6 **Resume** inside the waiting loop: re-allocate KV cache, register, take traffic.



Read the paper on advanced techniques

<https://arxiv.org/pdf/2609.25451>

08

**Snapshot
(checkpoint/restore)**

Start new workers in seconds

Why the waiting loop: the worker writes "ready" before it registers, then polls for a restore signal. A restored copy wakes up at that exact instruction and registers as a fresh worker. Live TCP connections can't be checkpointed, which is why multi-node workers require special handling (read paper)

Snapshot: Checkpoint Once, Restore Many

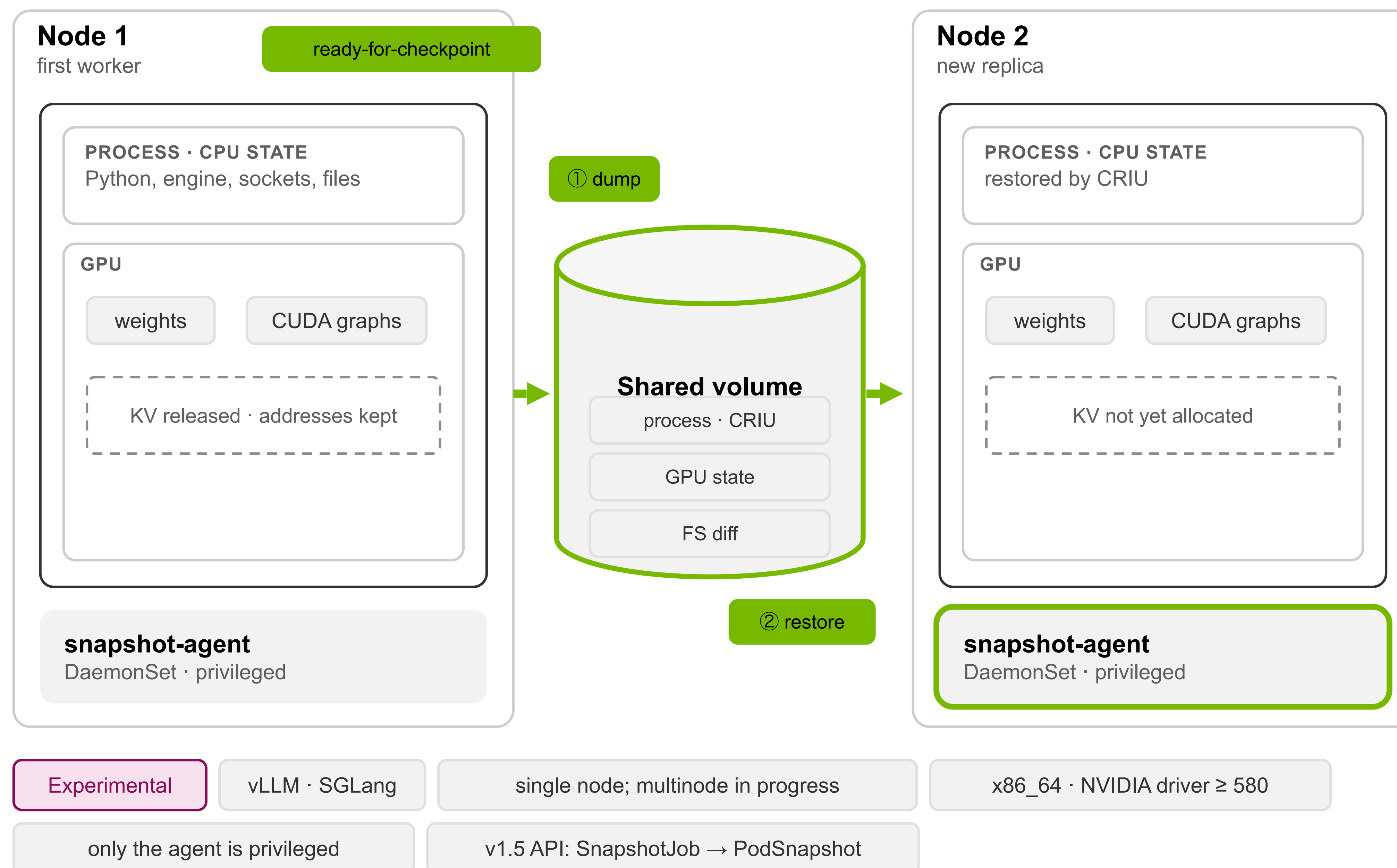
cuda-checkpoint + CRIU, driven by a node agent; restore on any node

CHECKPOINT · ONCE

- 1 **Initialize** one worker the slow way: load weights, warm up, capture CUDA graphs.
- 2 **Quiesce.** Release KV-cache memory (keeping the virtual addresses the CUDA graphs use) and signal "ready" before joining the cluster.
- 3 **Dump.** The node's snapshot agent runs cuda-checkpoint (GPU state → host) and CRIU (process tree and files), and writes the image to shared storage.

RESTORE · EVERY NEW REPLICA

- 4 **Place** a new pod from a lightweight placeholder image, on any node.
- 5 **Restore** the filesystem, the process (CRIU) and its GPU state (cuda-checkpoint).
- 6 **Resume** inside the waiting loop: re-allocate KV cache, register, take traffic.



Read the paper on advanced techniques

<https://arxiv.org/pdf/2609.25451>

Why the waiting loop: the worker writes "ready" before it registers, then polls for a restore signal. A restored copy wakes up at that exact instruction and registers as a fresh worker. Live TCP connections can't be checkpointed, which is why multi-node workers require special handling (read paper)

08

**Snapshot
(checkpoint/restore)**

Start new workers in seconds

Snapshot: Checkpoint Once, Restore Many

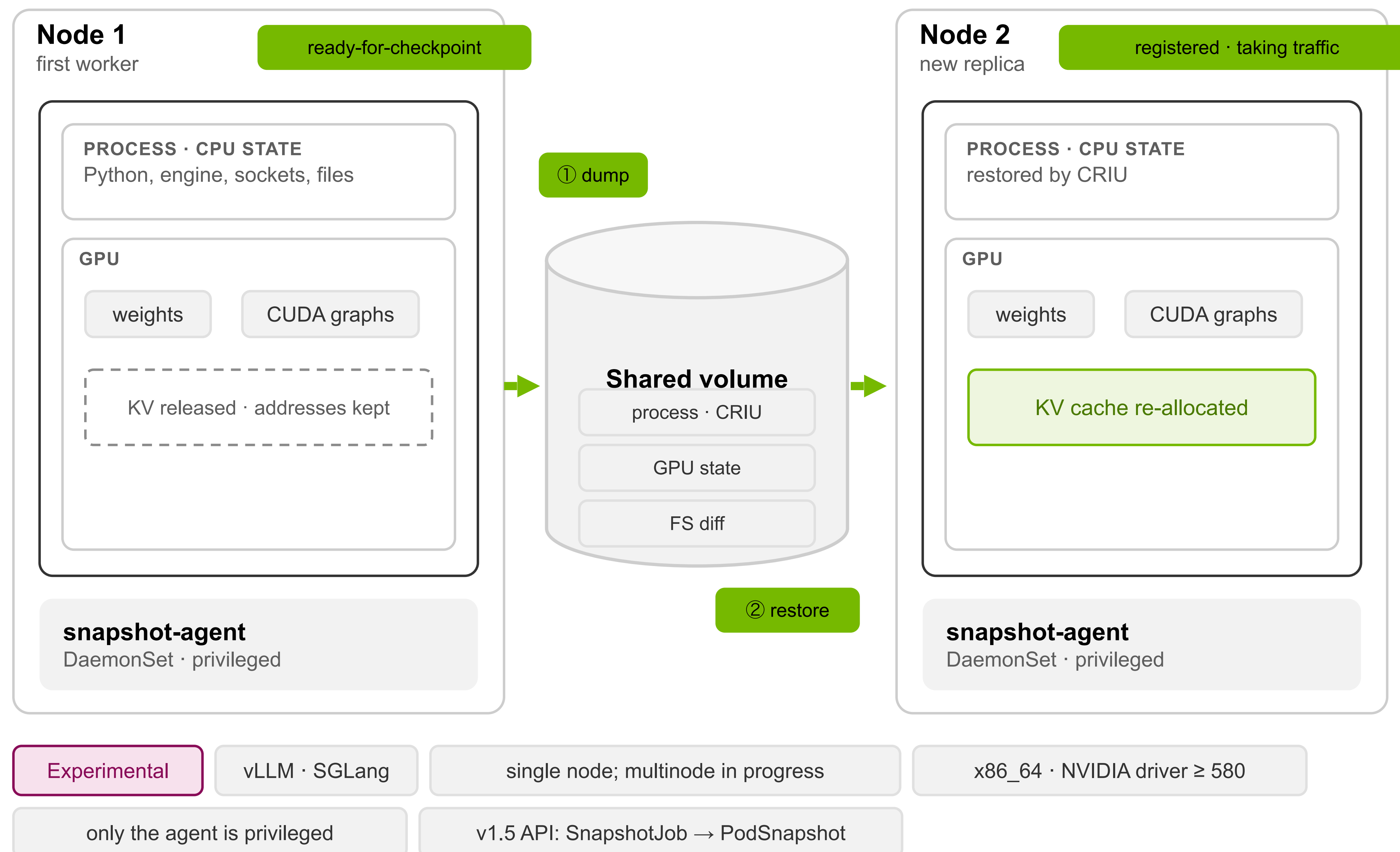
cuda-checkpoint + CRIU, driven by a node agent; restore on any node

CHECKPOINT · ONCE

- 1 **Initialize** one worker the slow way: load weights, warm up, capture CUDA graphs.
- 2 **Quiesce.** Release KV-cache memory (keeping the virtual addresses the CUDA graphs use) and signal "ready" before joining the cluster.
- 3 **Dump.** The node's snapshot agent runs cuda-checkpoint (GPU state → host) and CRIU (process tree and files), and writes the image to shared storage.

RESTORE · EVERY NEW REPLICA

- 4 **Place** a new pod from a lightweight placeholder image, on any node.
- 5 **Restore** the filesystem, the process (CRIU) and its GPU state (cuda-checkpoint).
- 6 **Resume** inside the waiting loop: re-allocate KV cache, register, take traffic.



Read the paper on advanced techniques

<https://arxiv.org/pdf/2609.25451>

Why the waiting loop: the worker writes "ready" before it registers, then polls for a restore signal. A restored copy wakes up at that exact instruction and registers as a fresh worker. Live TCP connections can't be checkpointed, which is why multi-node workers require special handling (read paper)

08

**Snapshot
(checkpoint/restore)**

Start new workers in seconds

Sizing and Scheduling Algorithm using AISimulate

Simulate the serving stack before you spend GPU time: predict, search, then validate

A GPU-free simulator for LLM serving. It predicts how a model, engine and deployment will perform on target GPUs, then searches thousands of configurations for the best one.

INPUTS

model + engine

GPU system

workload (trace / rate)

SLA (TTFT, TPOT)

GPU budget



WHY IT MATTERS

- **The design space is enormous.** Aggregated or disaggregated, how many prefill and decode workers, TP/PP/EP for each, router and planner knobs.
- **Real benchmarks are slow and expensive.** Each candidate ties up GPUs for hours; simulation ranks thousands in seconds to minutes.
- **Capacity questions get answered up front:** how many GPUs meet an SLA, which topology suits a workload, what to validate first.

```
aisimulate predict --stack dynamo --config prediction.yaml  
aisimulate recommend --stack dynamo --config recommendation.yaml
```

Example from the aisimulate README (AIConfigurator mode).

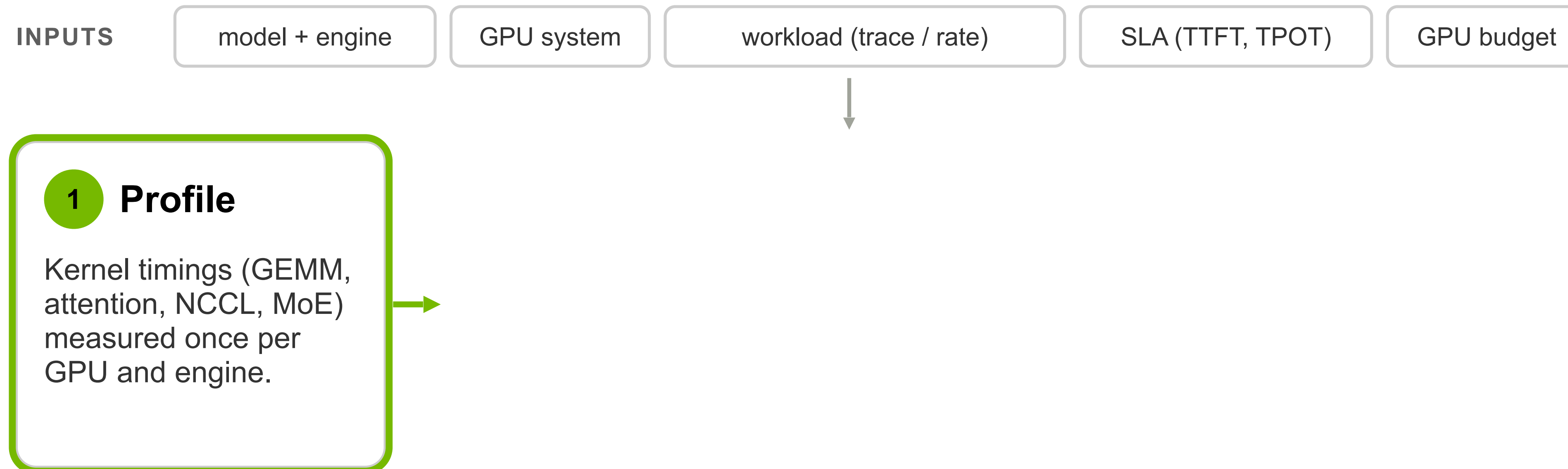
Sizing and Scheduling Algorithm using AISimulate

Simulate the serving stack before you spend GPU time: predict, search, then validate

A GPU-free simulator for LLM serving. It predicts how a model, engine and deployment will perform on target GPUs, then searches thousands of configurations for the best one.

WHY IT MATTERS

- **The design space is enormous.** Aggregated or disaggregated, how many prefill and decode workers, TP/PP/EP for each, router and planner knobs.
- **Real benchmarks are slow and expensive.** Each candidate ties up GPUs for hours; simulation ranks thousands in seconds to minutes.
- **Capacity questions get answered up front:** how many GPUs meet an SLA, which topology suits a workload, what to validate first.



```
aisimulate predict --stack dynamo --config prediction.yaml
aisimulate recommend --stack dynamo --config recommendation.yaml
```

Example from the aisimulate README (AIConfigurator mode).

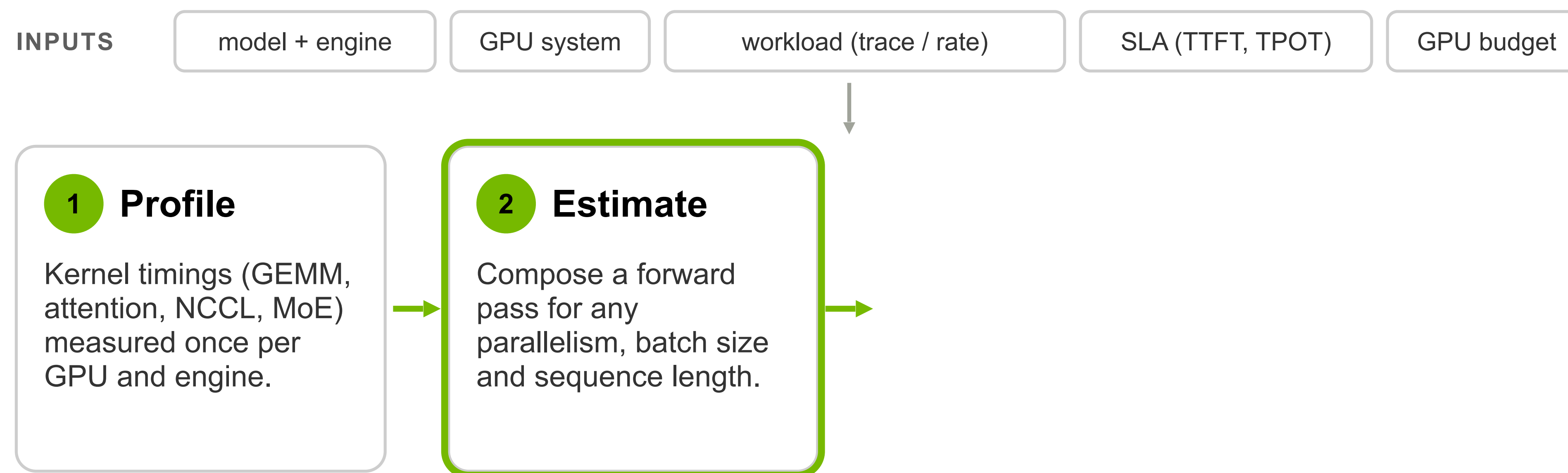
Sizing and Scheduling Algorithm using AISimulate

Simulate the serving stack before you spend GPU time: predict, search, then validate

A GPU-free simulator for LLM serving. It predicts how a model, engine and deployment will perform on target GPUs, then searches thousands of configurations for the best one.

WHY IT MATTERS

- **The design space is enormous.** Aggregated or disaggregated, how many prefill and decode workers, TP/PP/EP for each, router and planner knobs.
- **Real benchmarks are slow and expensive.** Each candidate ties up GPUs for hours; simulation ranks thousands in seconds to minutes.
- **Capacity questions get answered up front:** how many GPUs meet an SLA, which topology suits a workload, what to validate first.



```
aisimulate predict --stack dynamo --config prediction.yaml
aisimulate recommend --stack dynamo --config recommendation.yaml
```

Example from the aisimulate README (AIConfigurator mode).

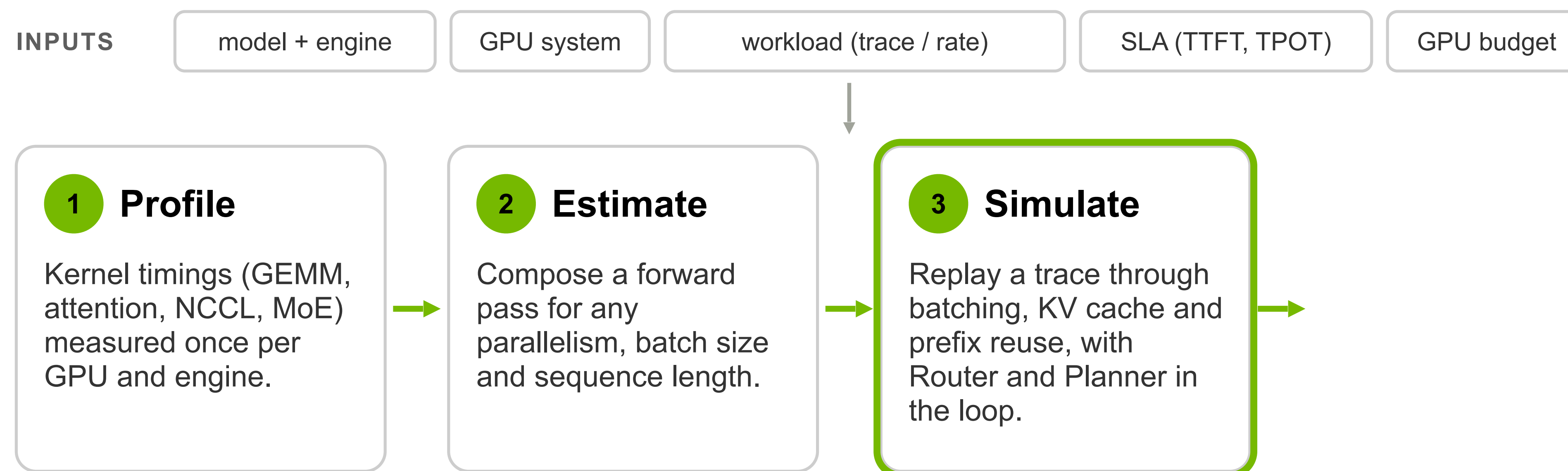
Sizing and Scheduling Algorithm using AISimulate

Simulate the serving stack before you spend GPU time: predict, search, then validate

A GPU-free simulator for LLM serving. It predicts how a model, engine and deployment will perform on target GPUs, then searches thousands of configurations for the best one.

WHY IT MATTERS

- **The design space is enormous.** Aggregated or disaggregated, how many prefill and decode workers, TP/PP/EP for each, router and planner knobs.
- **Real benchmarks are slow and expensive.** Each candidate ties up GPUs for hours; simulation ranks thousands in seconds to minutes.
- **Capacity questions get answered up front:** how many GPUs meet an SLA, which topology suits a workload, what to validate first.



```
aisimulate predict --stack dynamo --config prediction.yaml
aisimulate recommend --stack dynamo --config recommendation.yaml
```

Example from the aisimulate README (AIConfigurator mode).

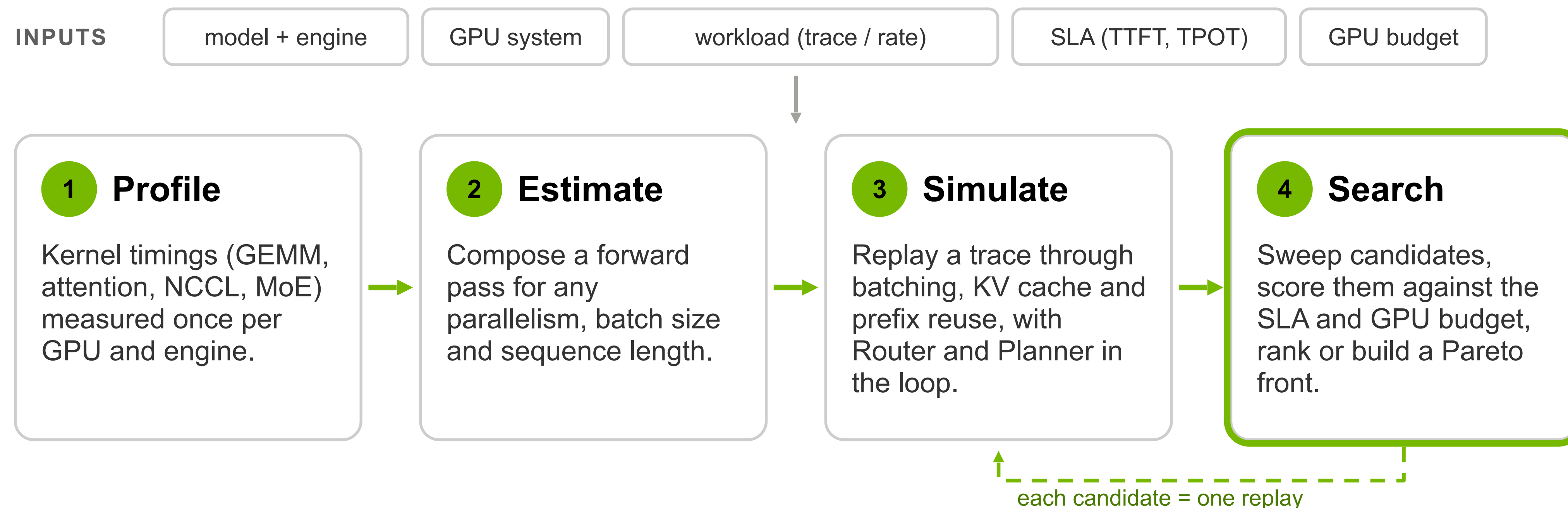
Sizing and Scheduling Algorithm using AISimulate

Simulate the serving stack before you spend GPU time: predict, search, then validate

A GPU-free simulator for LLM serving. It predicts how a model, engine and deployment will perform on target GPUs, then searches thousands of configurations for the best one.

WHY IT MATTERS

- **The design space is enormous.** Aggregated or disaggregated, how many prefill and decode workers, TP/PP/EP for each, router and planner knobs.
- **Real benchmarks are slow and expensive.** Each candidate ties up GPUs for hours; simulation ranks thousands in seconds to minutes.
- **Capacity questions get answered up front:** how many GPUs meet an SLA, which topology suits a workload, what to validate first.



```
aisimulate predict --stack dynamo --config prediction.yaml
aisimulate recommend --stack dynamo --config recommendation.yaml
```

Example from the aisimulate README (AIConfigurator mode).

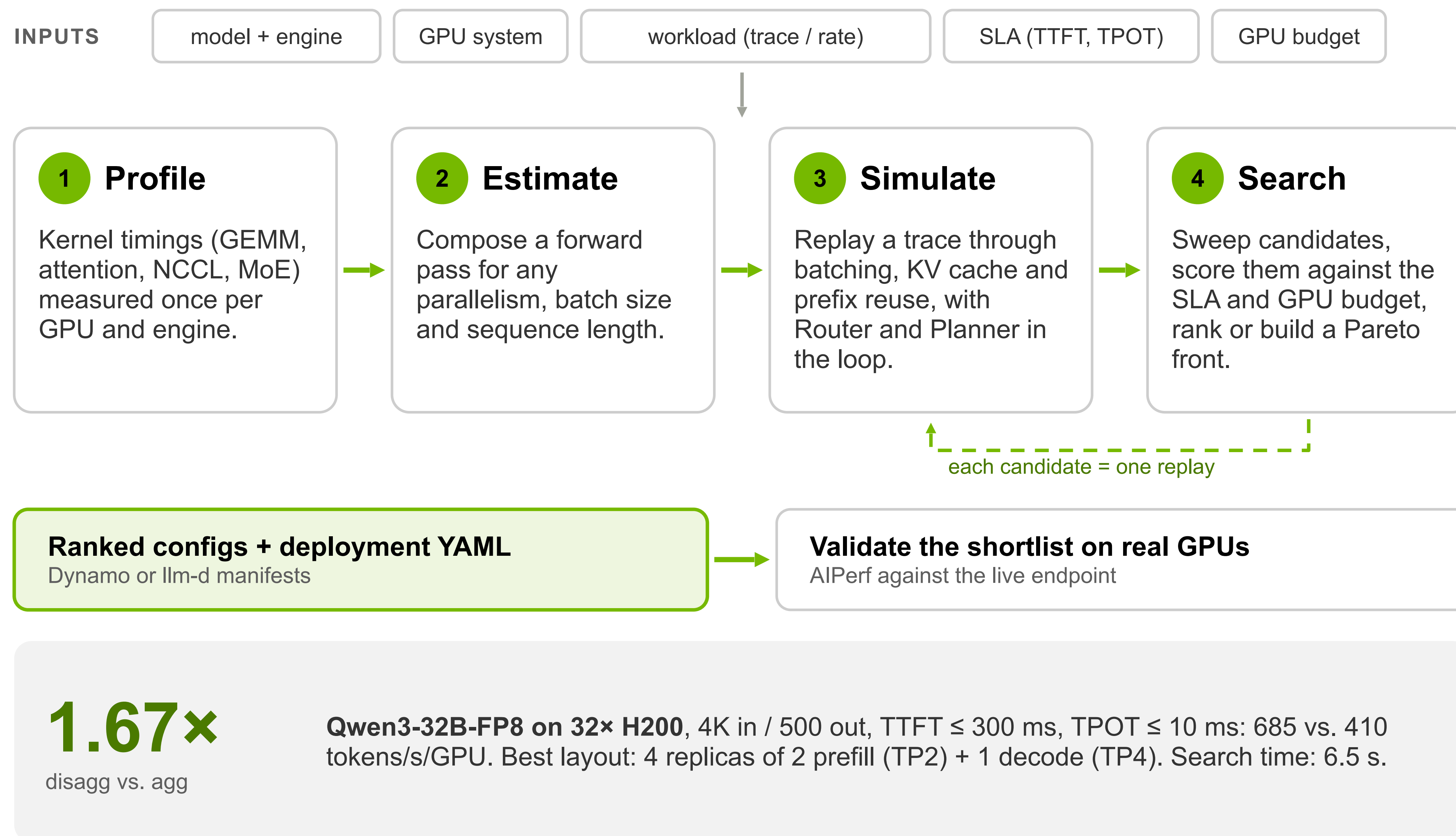
Sizing and Scheduling Algorithm using AISimulate

Simulate the serving stack before you spend GPU time: predict, search, then validate

A GPU-free simulator for LLM serving. It predicts how a model, engine and deployment will perform on target GPUs, then searches thousands of configurations for the best one.

WHY IT MATTERS

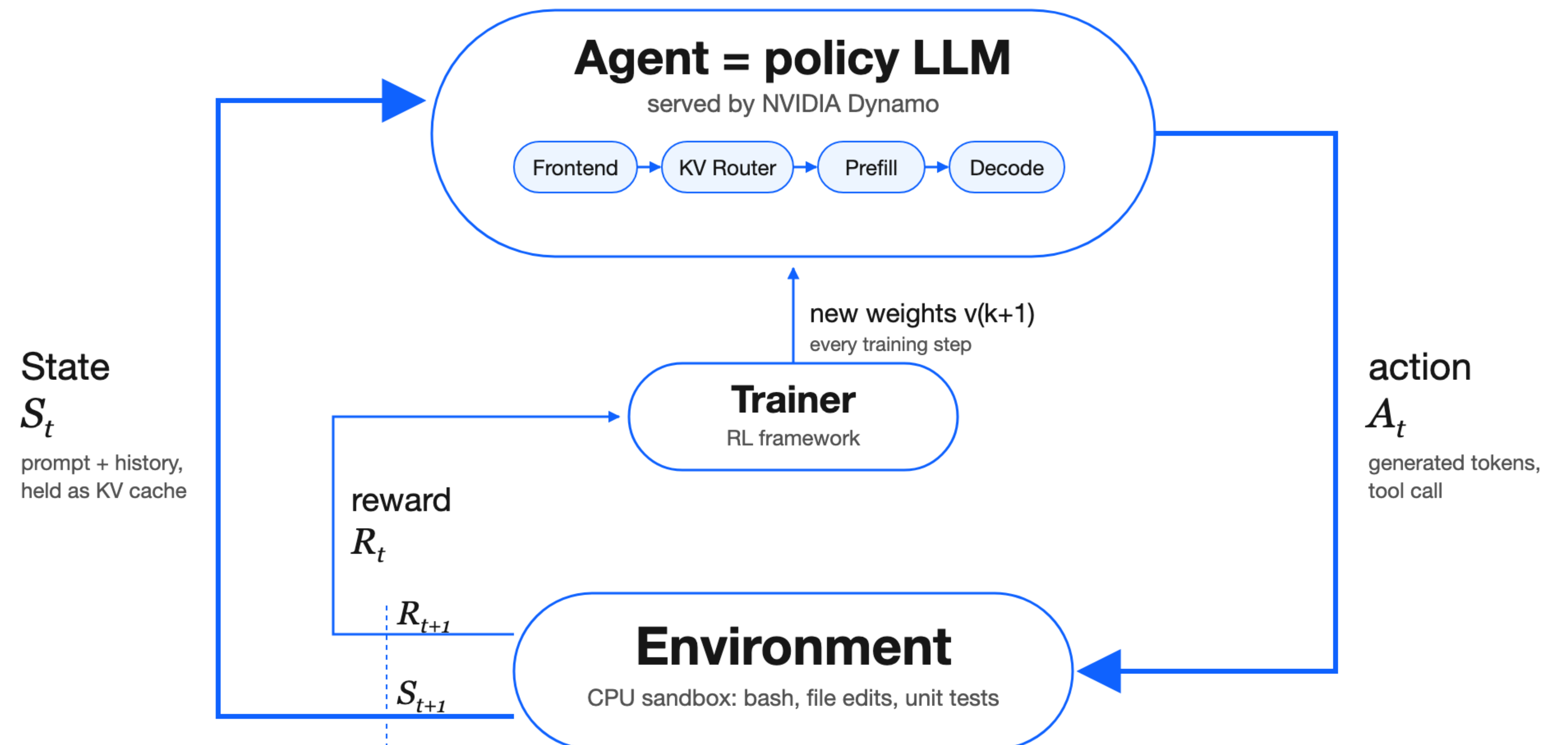
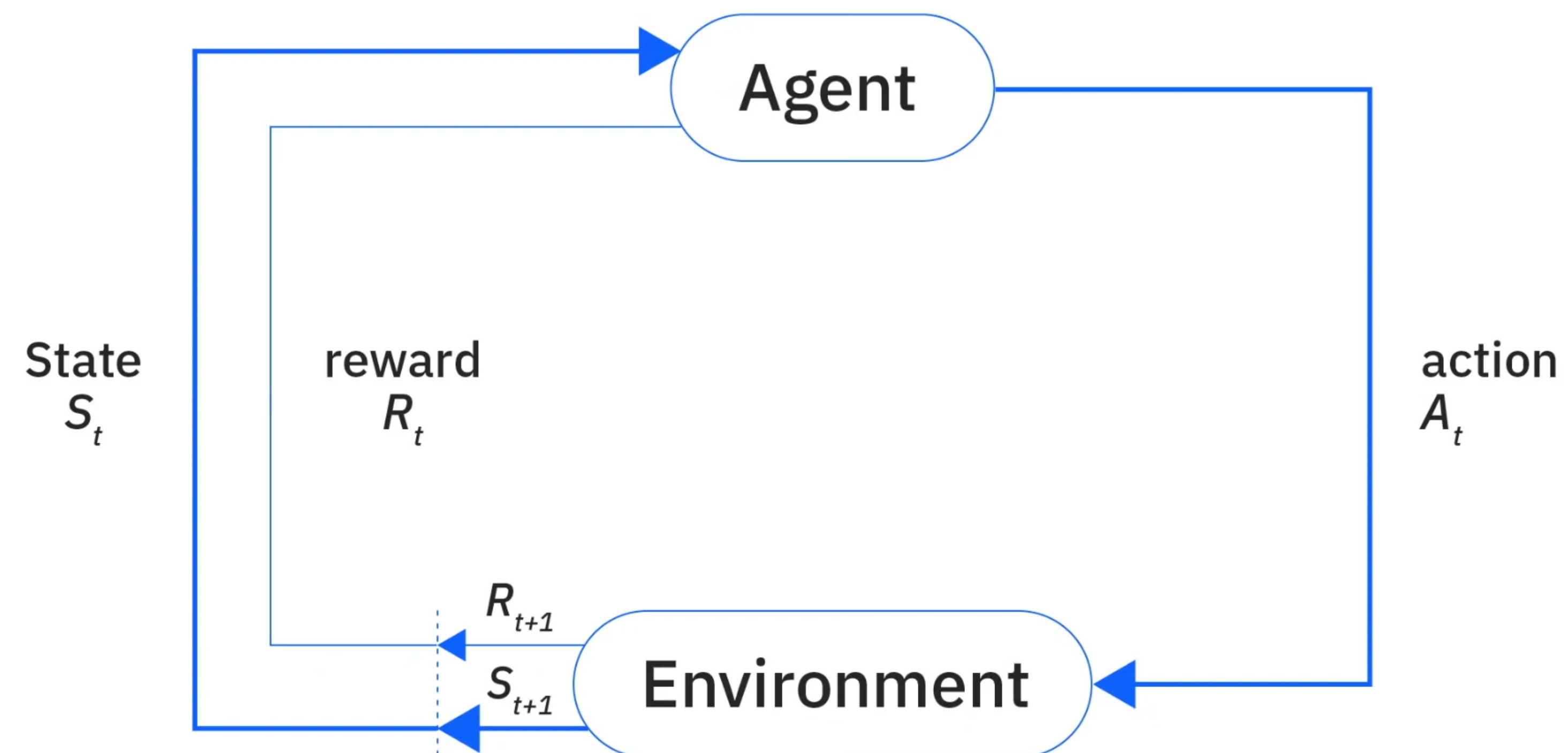
- **The design space is enormous.** Aggregated or disaggregated, how many prefill and decode workers, TP/PP/EP for each, router and planner knobs.
- **Real benchmarks are slow and expensive.** Each candidate ties up GPUs for hours; simulation ranks thousands in seconds to minutes.
- **Capacity questions get answered up front:** how many GPUs meet an SLA, which topology suits a workload, what to validate first.



```
aisimulate predict --stack dynamo --config prediction.yaml
aisimulate recommend --stack dynamo --config recommendation.yaml
```

Example from the aisimulate README (AIConfigurator mode).

Reinforcement Learning – Quick Recap



$S_{t+1} = S_t + A_t + \text{observation}$: almost all of the next state is a cached prefix

Reinforcement Learning is a serving problem

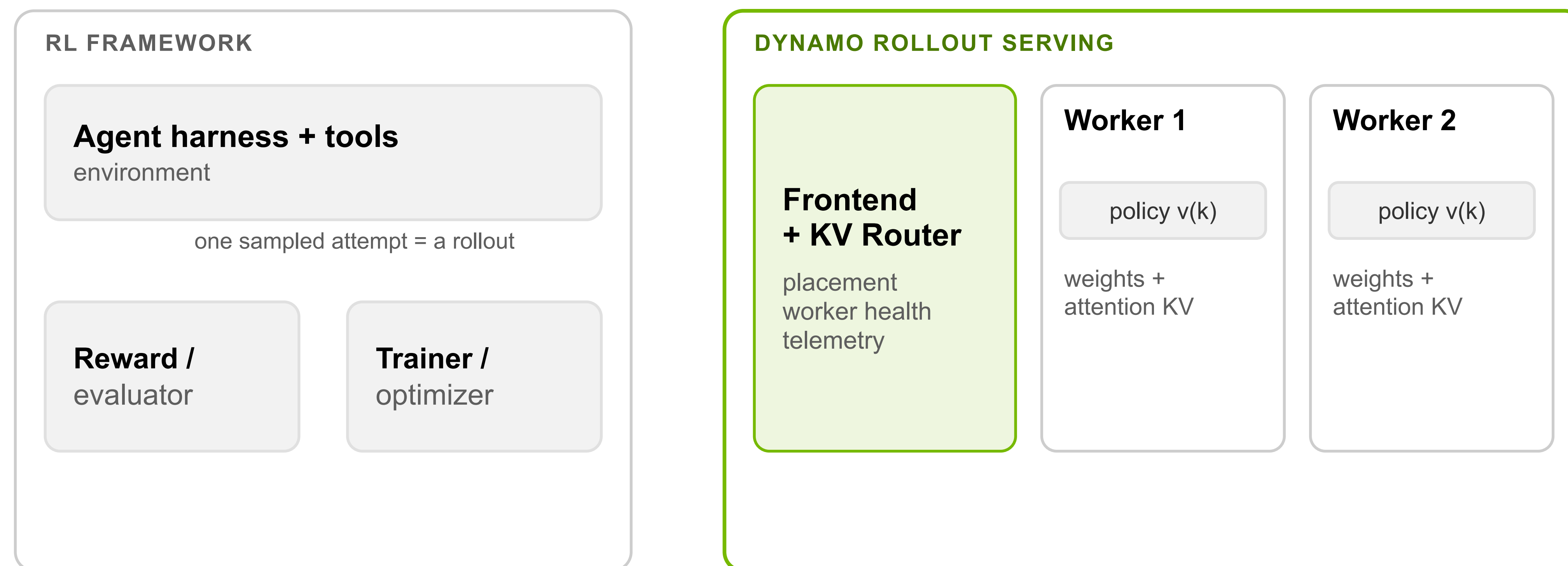
Dynamo serves and refreshes the rollout fleet; the RL framework owns learning

In RL post-training, the policy generates **rollouts** (sampled attempts, often multi-turn with tools), a reward scores them, and the trainer updates the policy. At scale, generating rollouts becomes a distributed serving problem.

WHY ROLLOUTS NEED A SERVING SYSTEM

- **Bursty, wide fan-out.** Each step launches many samples, then waits for the slowest ones to finish.
- **Shared prefixes.** A GRPO group samples several answers to one prompt; agent rollouts resend their history every turn.
- **The policy keeps changing.** Weights refresh every step without rebuilding the fleet, and old KV cache becomes invalid.
- **Failures are routine.** A lost worker must not stall the run.

Task: “Fix the checkout bug and pass the unit tests.”



In production: Cognition used Dynamo while training SWE-1.7 to manage engine lifecycles and route inference across a multi-cluster RL system.

Reinforcement Learning is a serving problem

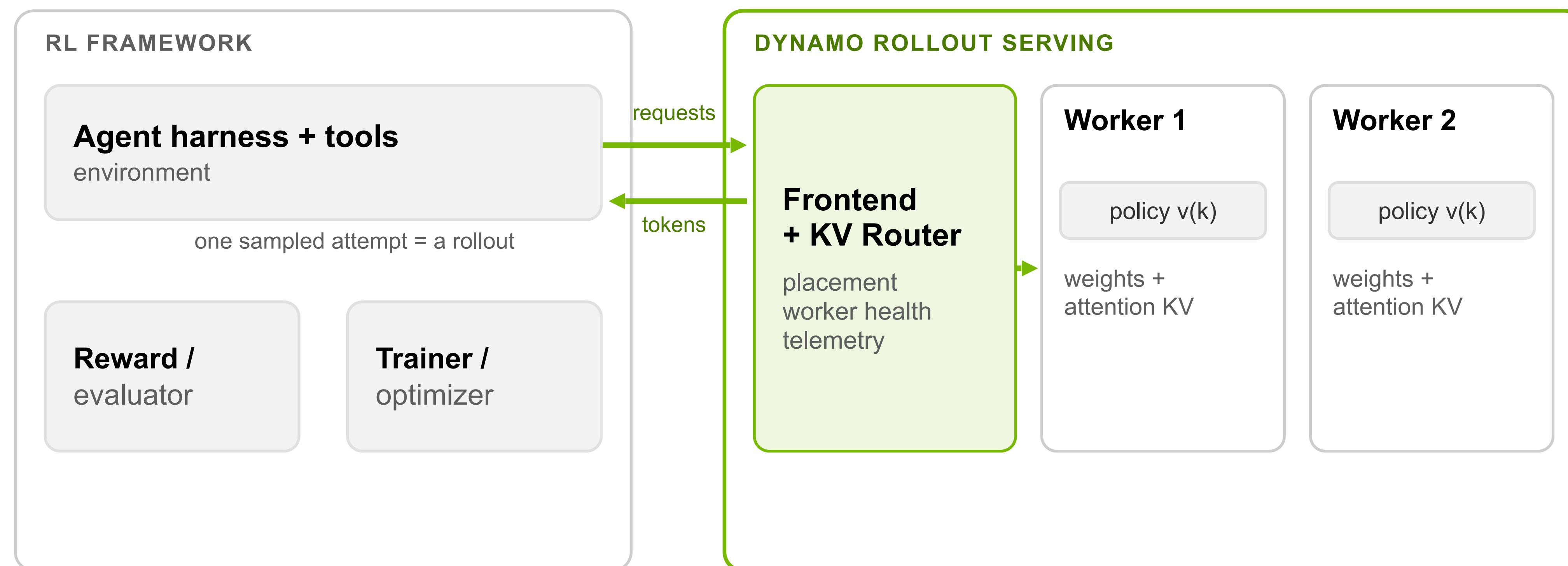
Dynamo serves and refreshes the rollout fleet; the RL framework owns learning

In RL post-training, the policy generates **rollouts** (sampled attempts, often multi-turn with tools), a reward scores them, and the trainer updates the policy. At scale, generating rollouts becomes a distributed serving problem.

WHY ROLLOUTS NEED A SERVING SYSTEM

- **Bursty, wide fan-out.** Each step launches many samples, then waits for the slowest ones to finish.
- **Shared prefixes.** A GRPO group samples several answers to one prompt; agent rollouts resend their history every turn.
- **The policy keeps changing.** Weights refresh every step without rebuilding the fleet, and old KV cache becomes invalid.
- **Failures are routine.** A lost worker must not stall the run.

Task: “Fix the checkout bug and pass the unit tests.”



In production: Cognition used Dynamo while training SWE-1.7 to manage engine lifecycles and route inference across a multi-cluster RL system.

Reinforcement Learning is a serving problem

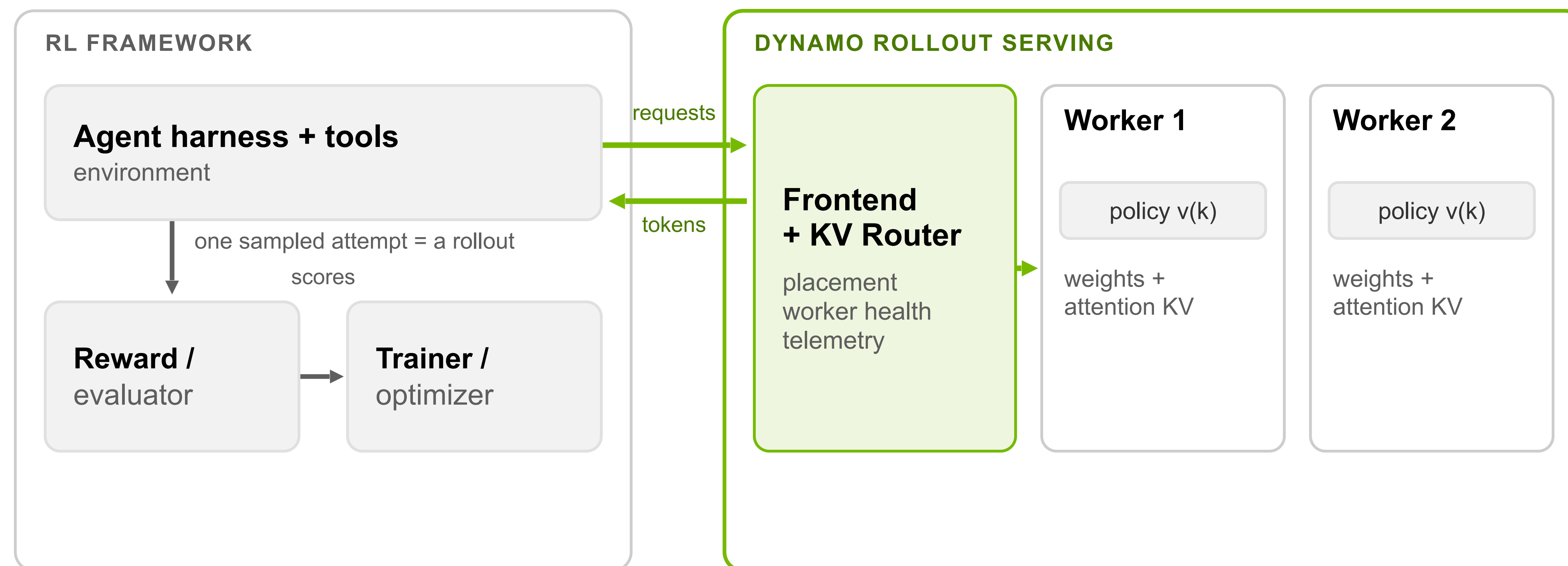
Dynamo serves and refreshes the rollout fleet; the RL framework owns learning

In RL post-training, the policy generates **rollouts** (sampled attempts, often multi-turn with tools), a reward scores them, and the trainer updates the policy. At scale, generating rollouts becomes a distributed serving problem.

WHY ROLLOUTS NEED A SERVING SYSTEM

- **Bursty, wide fan-out.** Each step launches many samples, then waits for the slowest ones to finish.
- **Shared prefixes.** A GRPO group samples several answers to one prompt; agent rollouts resend their history every turn.
- **The policy keeps changing.** Weights refresh every step without rebuilding the fleet, and old KV cache becomes invalid.
- **Failures are routine.** A lost worker must not stall the run.

Task: “Fix the checkout bug and pass the unit tests.”



In production: Cognition used Dynamo while training SWE-1.7 to manage engine lifecycles and route inference across a multi-cluster RL system.

Reinforcement Learning is a serving problem

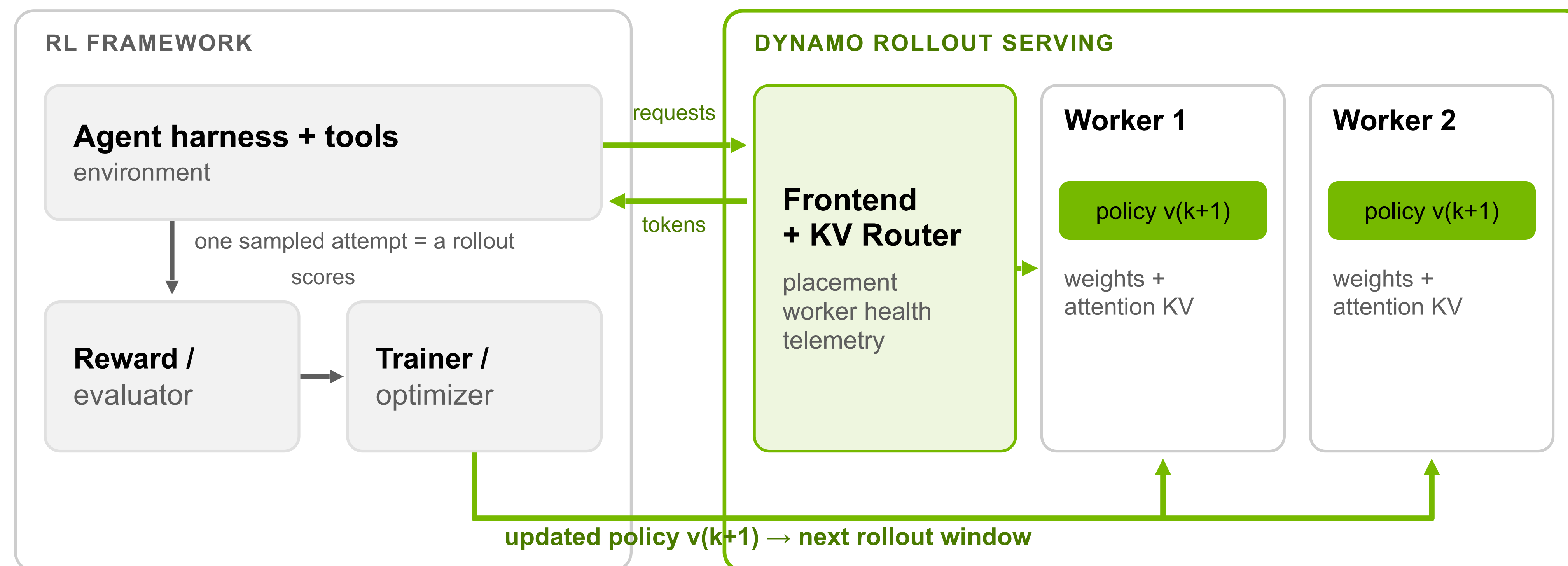
Dynamo serves and refreshes the rollout fleet; the RL framework owns learning

In RL post-training, the policy generates **rollouts** (sampled attempts, often multi-turn with tools), a reward scores them, and the trainer updates the policy. At scale, generating rollouts becomes a distributed serving problem.

WHY ROLLOUTS NEED A SERVING SYSTEM

- **Bursty, wide fan-out.** Each step launches many samples, then waits for the slowest ones to finish.
- **Shared prefixes.** A GRPO group samples several answers to one prompt; agent rollouts resend their history every turn.
- **The policy keeps changing.** Weights refresh every step without rebuilding the fleet, and old KV cache becomes invalid.
- **Failures are routine.** A lost worker must not stall the run.

Task: “Fix the checkout bug and pass the unit tests.”



In production: Cognition used Dynamo while training SWE-1.7 to manage engine lifecycles and route inference across a multi-cluster RL system.

Reinforcement Learning is a serving problem

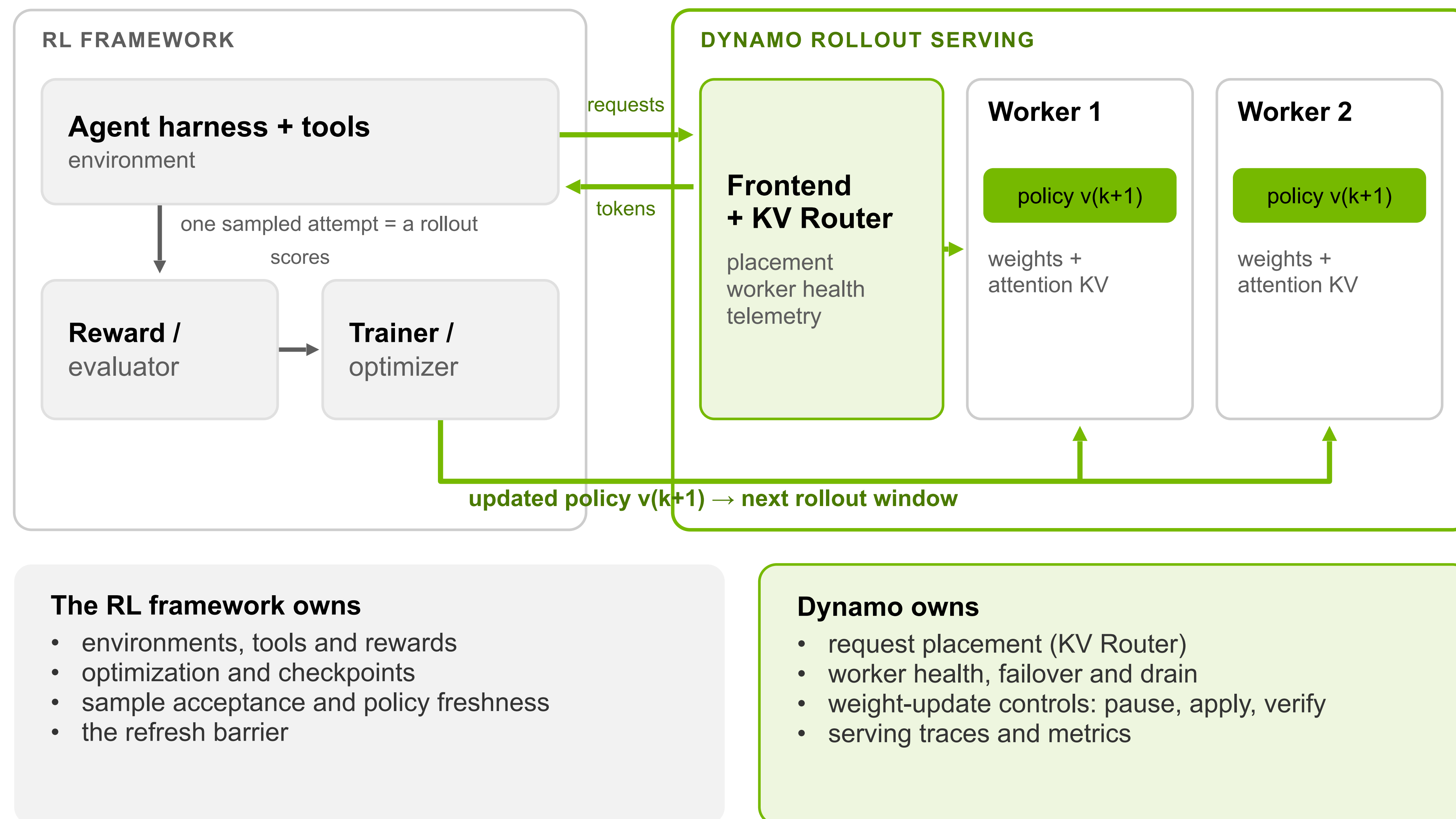
Dynamo serves and refreshes the rollout fleet; the RL framework owns learning

In RL post-training, the policy generates **rollouts** (sampled attempts, often multi-turn with tools), a reward scores them, and the trainer updates the policy. At scale, generating rollouts becomes a distributed serving problem.

WHY ROLLOUTS NEED A SERVING SYSTEM

- **Bursty, wide fan-out.** Each step launches many samples, then waits for the slowest ones to finish.
- **Shared prefixes.** A GRPO group samples several answers to one prompt; agent rollouts resend their history every turn.
- **The policy keeps changing.** Weights refresh every step without rebuilding the fleet, and old KV cache becomes invalid.
- **Failures are routine.** A lost worker must not stall the run.

Task: “Fix the checkout bug and pass the unit tests.”



Engines (vLLM, SGLang, TensorRT-LLM) generate tokens and apply the new weights.

In production: Cognition used Dynamo while training SWE-1.7 to manage engine lifecycles and route inference across a multi-cluster RL system.

RL Rollouts Exercise the Whole Stack

One RL step uses most of the components from this lecture

WHAT THE RL LOOP NEEDS

DYNAMO PIECE

WHAT IT DOES IN THE LOOP

Rollout groups share prefixes

GRPO samples several answers per prompt; agent turns resend history

KV Router

Routes repeated prefixes to cache-rich workers while accounting for live load and queue pressure.

Workers fail mid-rollout

long agentic rollouts on large fleets

Fault tolerance

Reroutes around lost workers, migrates supported in-flight requests, rejects overload explicitly and drains on shutdown. The framework decides retry or discard.

A new policy every step

weights must reach every rollout worker quickly

ModelExpress

NIXL

Stage weights from trainer ranks, an S3 artifact or an already-updated peer; the engine applies them. Next slide.

Fleet size and placement

when rollouts run on Kubernetes (optional)

Planner

Grove + KAI

Gang-schedule multi-node workers inside one topology domain and size pools to the rollout load.

Explain slow or wasted steps

useful samples vs. update pauses

Traces

AIPerf

DynoSim

Join framework records with Dynamo request traces; replay the serving traffic on GPUs or simulate alternatives. Tools and training are not rerun.

FRAMEWORK INTEGRATIONS · ALL EXPERIMENTAL

verl · rollout recipe

NeMo RL · managed backend

Prime-RL · router drop-in

Slime · SGLang + sidecars

custom trainer · external contract

RL Rollouts Exercise the Whole Stack

One RL step uses most of the components from this lecture

WHAT THE RL LOOP NEEDS

DYNAMO PIECE

WHAT IT DOES IN THE LOOP

Rollout groups share prefixes

GRPO samples several answers per prompt; agent turns resend history

KV Router

Routes repeated prefixes to cache-rich workers while accounting for live load and queue pressure.

Workers fail mid-rollout

long agentic rollouts on large fleets

Fault tolerance

Reroutes around lost workers, migrates supported in-flight requests, rejects overload explicitly and drains on shutdown. The framework decides retry or discard.

A new policy every step

weights must reach every rollout worker quickly

ModelExpress

NIXL

Stage weights from trainer ranks, an S3 artifact or an already-updated peer; the engine applies them. Next slide.

Fleet size and placement

when rollouts run on Kubernetes (optional)

Planner

Grove + KAI

Gang-schedule multi-node workers inside one topology domain and size pools to the rollout load.

Explain slow or wasted steps

useful samples vs. update pauses

Traces

AIPerf

DynoSim

Join framework records with Dynamo request traces; replay the serving traffic on GPUs or simulate alternatives. Tools and training are not rerun.

FRAMEWORK INTEGRATIONS · ALL EXPERIMENTAL

verl · rollout recipe

NeMo RL · managed backend

Prime-RL · router drop-in

Slime · SGLang + sidecars

custom trainer · external contract

RL Rollouts Exercise the Whole Stack

One RL step uses most of the components from this lecture

WHAT THE RL LOOP NEEDS

DYNAMO PIECE

WHAT IT DOES IN THE LOOP

Rollout groups share prefixes

GRPO samples several answers per prompt; agent turns resend history

KV Router

Routes repeated prefixes to cache-rich workers while accounting for live load and queue pressure.

Workers fail mid-rollout

long agentic rollouts on large fleets

Fault tolerance

Reroutes around lost workers, migrates supported in-flight requests, rejects overload explicitly and drains on shutdown. The framework decides retry or discard.

A new policy every step

weights must reach every rollout worker quickly

ModelExpress

NIXL

Stage weights from trainer ranks, an S3 artifact or an already-updated peer; the engine applies them. Next slide.

Fleet size and placement

when rollouts run on Kubernetes (optional)

Planner

Grove + KAI

Gang-schedule multi-node workers inside one topology domain and size pools to the rollout load.

Explain slow or wasted steps

useful samples vs. update pauses

Traces

AIPerf

DynoSim

Join framework records with Dynamo request traces; replay the serving traffic on GPUs or simulate alternatives. Tools and training are not rerun.

FRAMEWORK INTEGRATIONS · ALL EXPERIMENTAL

verl · rollout recipe

NeMo RL · managed backend

Prime-RL · router drop-in

Slime · SGLang + sidecars

custom trainer · external contract

RL Rollouts Exercise the Whole Stack

One RL step uses most of the components from this lecture

WHAT THE RL LOOP NEEDS

DYNAMO PIECE

WHAT IT DOES IN THE LOOP

Rollout groups share prefixes

GRPO samples several answers per prompt; agent turns resend history

KV Router

Routes repeated prefixes to cache-rich workers while accounting for live load and queue pressure.

Workers fail mid-rollout

long agentic rollouts on large fleets

Fault tolerance

Reroutes around lost workers, migrates supported in-flight requests, rejects overload explicitly and drains on shutdown. The framework decides retry or discard.

A new policy every step

weights must reach every rollout worker quickly

ModelExpress

NIXL

Stage weights from trainer ranks, an S3 artifact or an already-updated peer; the engine applies them. Next slide.

Fleet size and placement

when rollouts run on Kubernetes (optional)

Planner

Grove + KAI

Gang-schedule multi-node workers inside one topology domain and size pools to the rollout load.

Explain slow or wasted steps

useful samples vs. update pauses

Traces

AIPerf

DynoSim

Join framework records with Dynamo request traces; replay the serving traffic on GPUs or simulate alternatives. Tools and training are not rerun.

FRAMEWORK INTEGRATIONS · ALL EXPERIMENTAL

verl · rollout recipe

NeMo RL · managed backend

Prime-RL · router drop-in

Slime · SGLang + sidecars

custom trainer · external contract

RL Rollouts Exercise the Whole Stack

One RL step uses most of the components from this lecture

WHAT THE RL LOOP NEEDS

DYNAMO PIECE

WHAT IT DOES IN THE LOOP

Rollout groups share prefixes

GRPO samples several answers per prompt; agent turns resend history

KV Router

Routes repeated prefixes to cache-rich workers while accounting for live load and queue pressure.

Workers fail mid-rollout

long agentic rollouts on large fleets

Fault tolerance

Reroutes around lost workers, migrates supported in-flight requests, rejects overload explicitly and drains on shutdown. The framework decides retry or discard.

A new policy every step

weights must reach every rollout worker quickly

ModelExpress

NIXL

Stage weights from trainer ranks, an S3 artifact or an already-updated peer; the engine applies them. Next slide.

Fleet size and placement

when rollouts run on Kubernetes (optional)

Planner

Grove + KAI

Gang-schedule multi-node workers inside one topology domain and size pools to the rollout load.

Explain slow or wasted steps

useful samples vs. update pauses

Traces

AIPerf

DynoSim

Join framework records with Dynamo request traces; replay the serving traffic on GPUs or simulate alternatives. Tools and training are not rerun.

FRAMEWORK INTEGRATIONS · ALL EXPERIMENTAL

verl · rollout recipe

NeMo RL · managed backend

Prime-RL · router drop-in

Slime · SGLang + sidecars

custom trainer · external contract

RL Rollouts Exercise the Whole Stack

One RL step uses most of the components from this lecture

WHAT THE RL LOOP NEEDS

DYNAMO PIECE

WHAT IT DOES IN THE LOOP

Rollout groups share prefixes

GRPO samples several answers per prompt; agent turns resend history

KV Router

Routes repeated prefixes to cache-rich workers while accounting for live load and queue pressure.

Workers fail mid-rollout

long agentic rollouts on large fleets

Fault tolerance

Reroutes around lost workers, migrates supported in-flight requests, rejects overload explicitly and drains on shutdown. The framework decides retry or discard.

A new policy every step

weights must reach every rollout worker quickly

ModelExpress

NIXL

Stage weights from trainer ranks, an S3 artifact or an already-updated peer; the engine applies them. Next slide.

Fleet size and placement

when rollouts run on Kubernetes (optional)

Planner

Grove + KAI

Gang-schedule multi-node workers inside one topology domain and size pools to the rollout load.

Explain slow or wasted steps

useful samples vs. update pauses

Traces

AIPerf

DynoSim

Join framework records with Dynamo request traces; replay the serving traffic on GPUs or simulate alternatives. Tools and training are not rerun.

FRAMEWORK INTEGRATIONS · ALL EXPERIMENTAL

verl · rollout recipe

NeMo RL · managed backend

Prime-RL · router drop-in

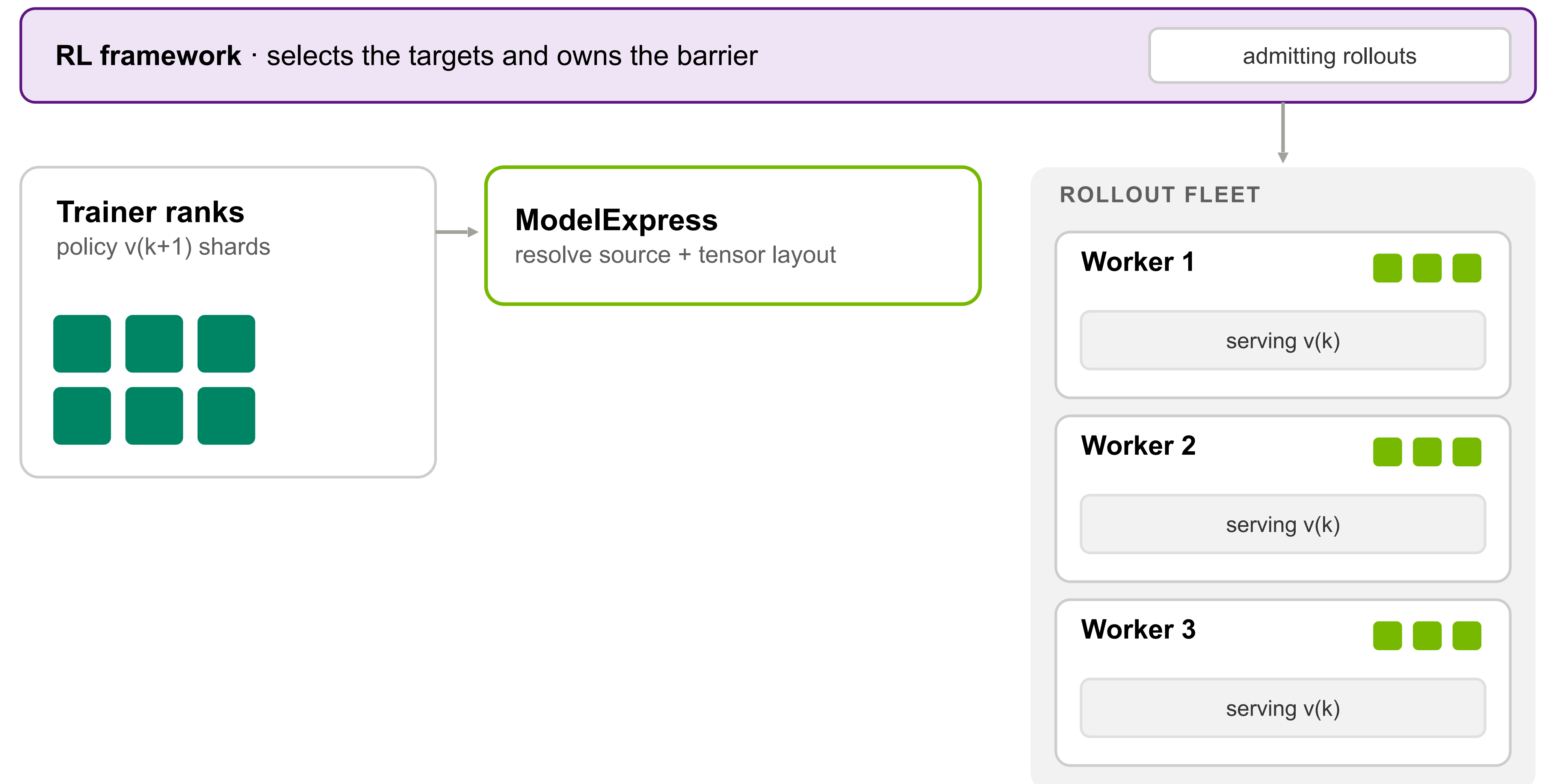
Slime · SGLang + sidecars

custom trainer · external contract

Policy Refresh: Gate, Stage, Apply, Verify, Resume

Dynamo and ModelExpress provide per-worker steps; the RL framework owns the fleet barrier

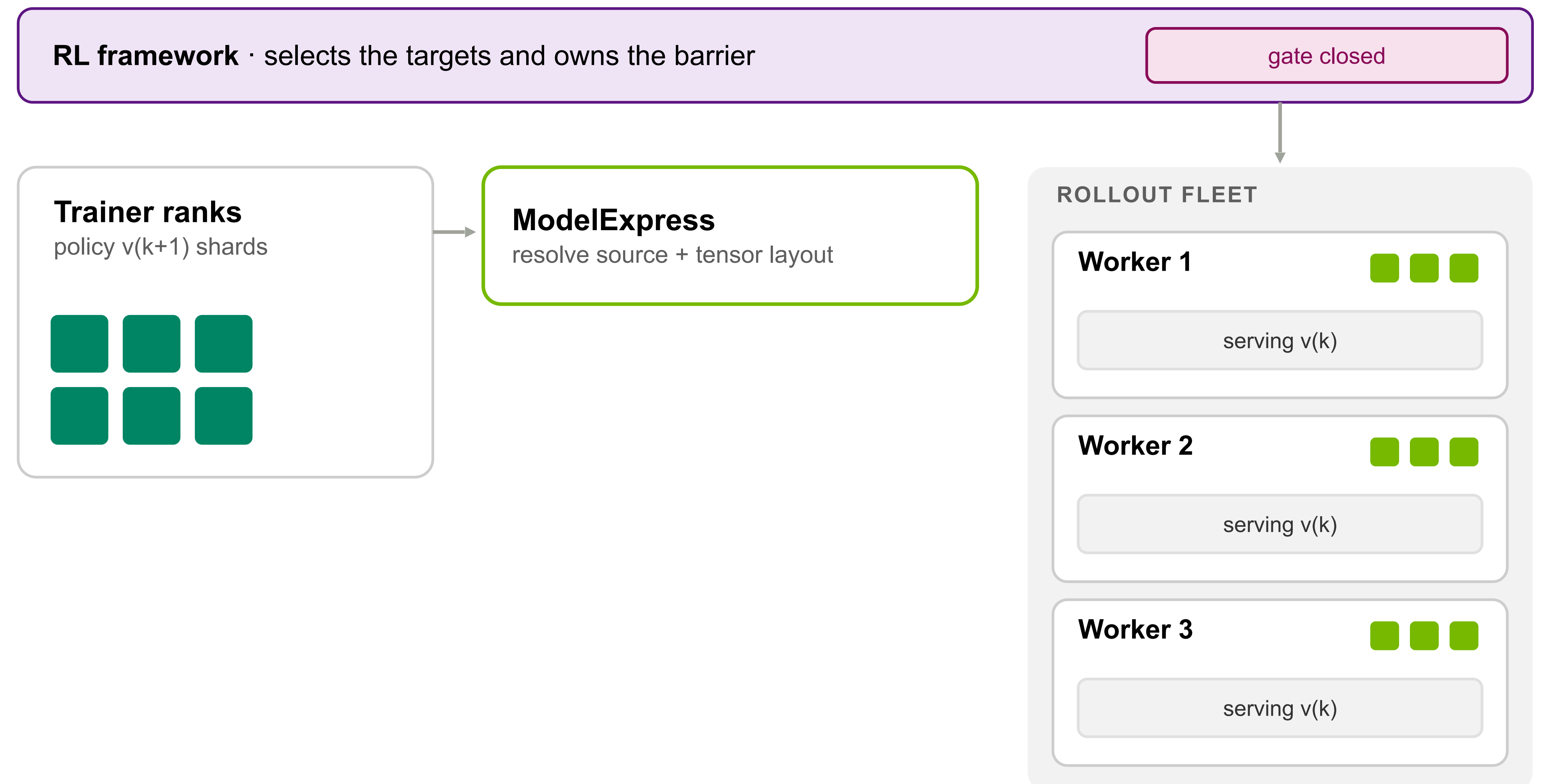
- 1 **Gate.** The framework stops admitting new rollouts to the target workers (or enforces its own bounded staleness).
- 2 **Pause** each target worker at a safe point.
- 3 **Stage.** ModelExpress resolves the source and tensor layout; each inference rank pulls the shards it needs over NIXL, from trainer ranks, an S3 artifact or an already-updated peer.
- 4 **Apply and reset.** The engine installs $v(k+1)$. KV cache built under $v(k)$ is invalid even for identical tokens, so every cache tier is cleared.
- 5 **Verify** the version on every target and run a post-update generation.
- 6 **Resume** only when the whole target set is verified.



Policy Refresh: Gate, Stage, Apply, Verify, Resume

Dynamo and ModelExpress provide per-worker steps; the RL framework owns the fleet barrier

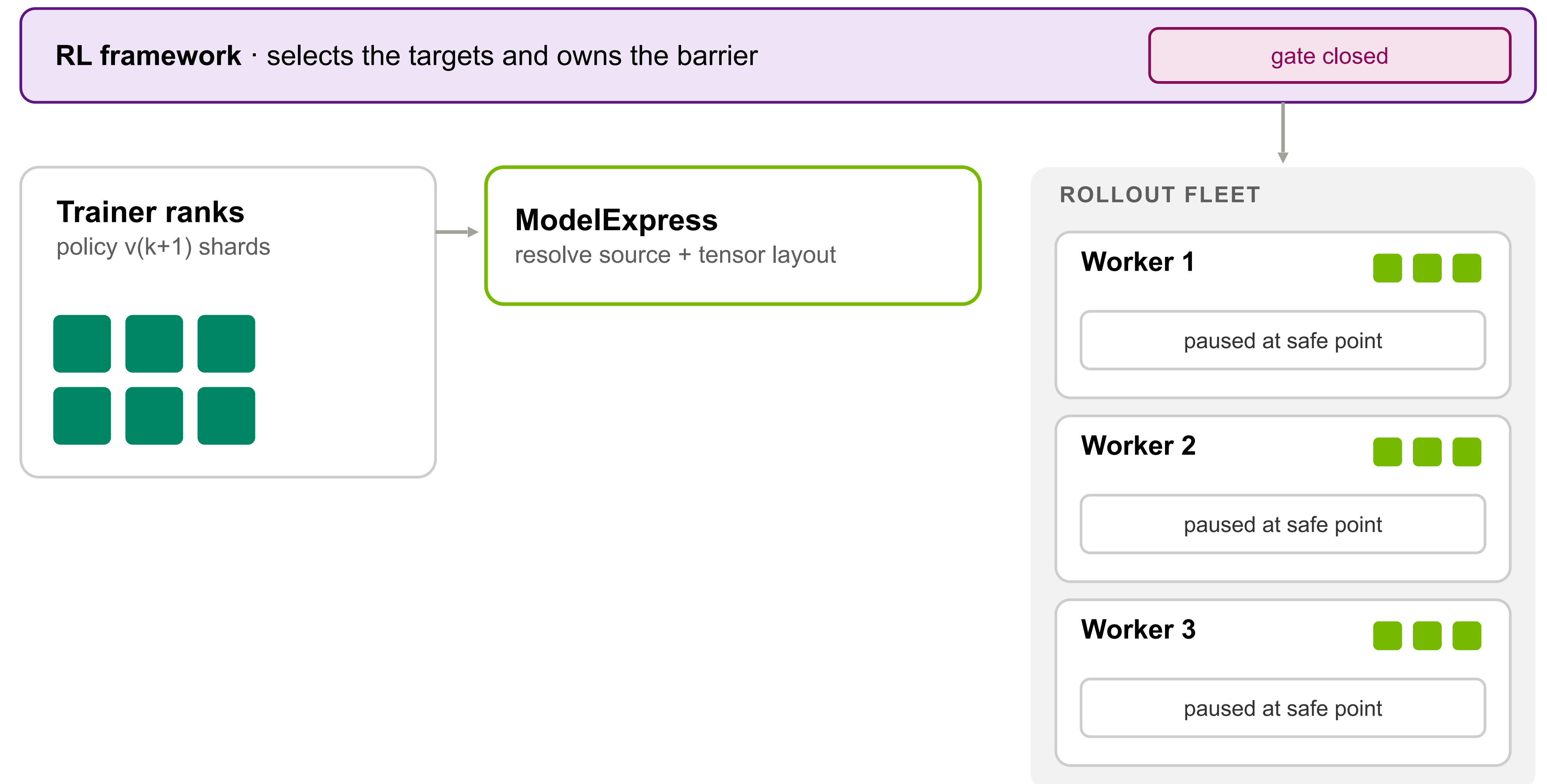
- 1 Gate.** The framework stops admitting new rollouts to the target workers (or enforces its own bounded staleness).
- 2 Pause** each target worker at a safe point.
- 3 Stage.** ModelExpress resolves the source and tensor layout; each inference rank pulls the shards it needs over NIXL, from trainer ranks, an S3 artifact or an already-updated peer.
- 4 Apply and reset.** The engine installs $v(k+1)$. KV cache built under $v(k)$ is invalid even for identical tokens, so every cache tier is cleared.
- 5 Verify** the version on every target and run a post-update generation.
- 6 Resume** only when the whole target set is verified.



Policy Refresh: Gate, Stage, Apply, Verify, Resume

Dynamo and ModelExpress provide per-worker steps; the RL framework owns the fleet barrier

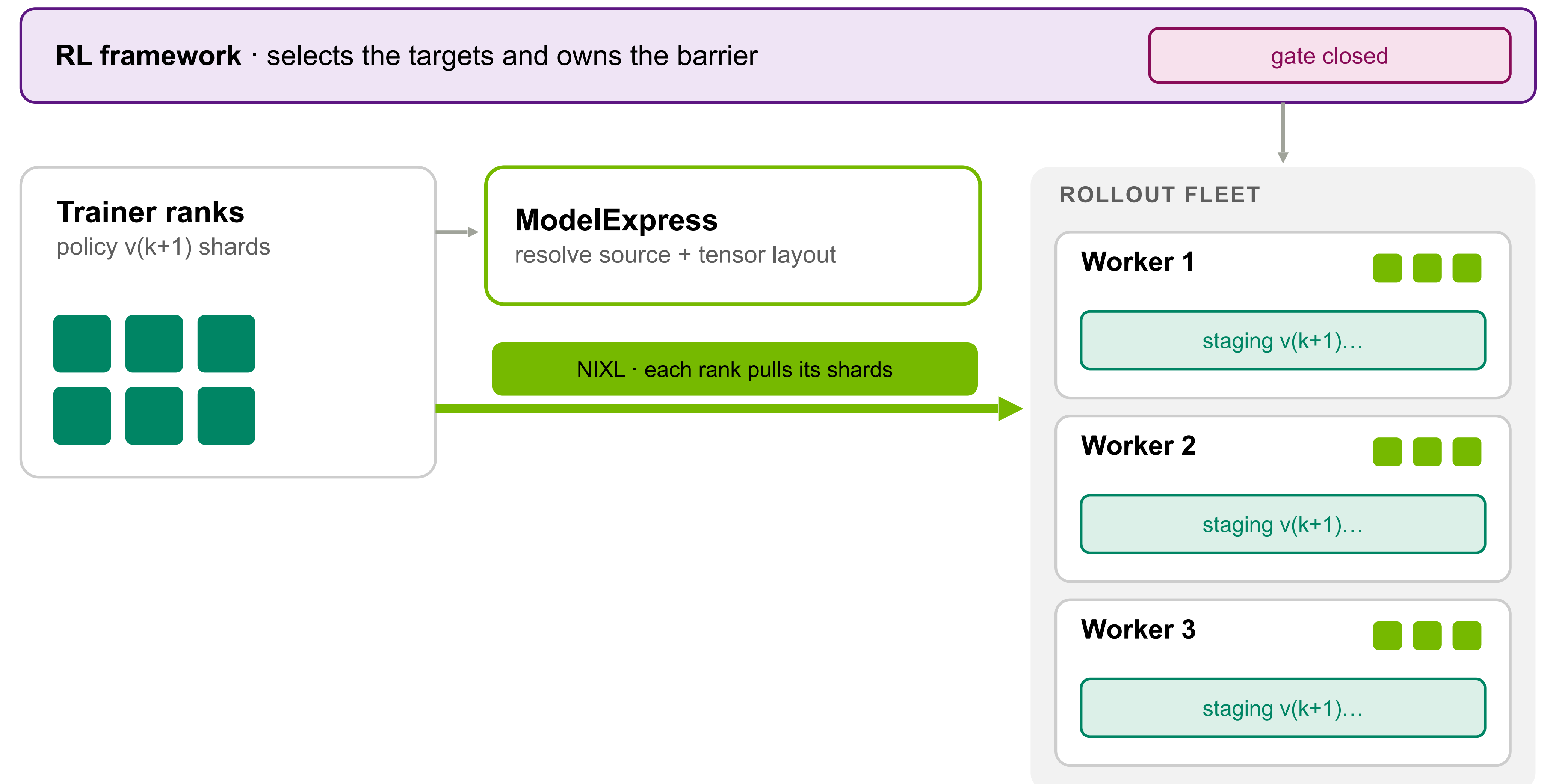
- 1 **Gate.** The framework stops admitting new rollouts to the target workers (or enforces its own bounded staleness).
- 2 **Pause** each target worker at a safe point.
- 3 **Stage.** ModelExpress resolves the source and tensor layout; each inference rank pulls the shards it needs over NIXL, from trainer ranks, an S3 artifact or an already-updated peer.
- 4 **Apply and reset.** The engine installs $v(k+1)$. KV cache built under $v(k)$ is invalid even for identical tokens, so every cache tier is cleared.
- 5 **Verify** the version on every target and run a post-update generation.
- 6 **Resume** only when the whole target set is verified.



Policy Refresh: Gate, Stage, Apply, Verify, Resume

Dynamo and ModelExpress provide per-worker steps; the RL framework owns the fleet barrier

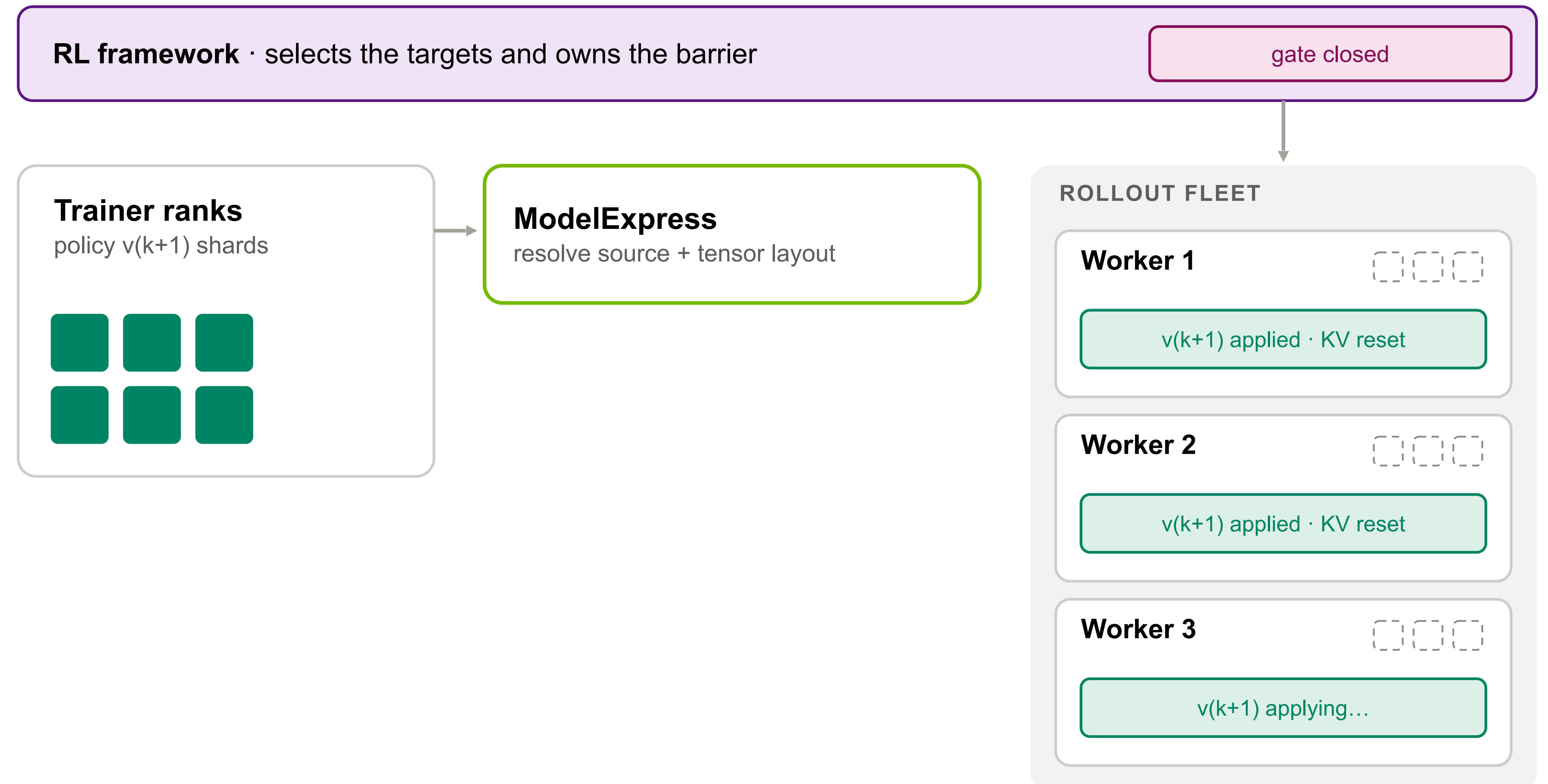
- 1 Gate.** The framework stops admitting new rollouts to the target workers (or enforces its own bounded staleness).
- 2 Pause** each target worker at a safe point.
- 3 Stage.** ModelExpress resolves the source and tensor layout; each inference rank pulls the shards it needs over NIXL, from trainer ranks, an S3 artifact or an already-updated peer.
- 4 Apply and reset.** The engine installs $v(k+1)$. KV cache built under $v(k)$ is invalid even for identical tokens, so every cache tier is cleared.
- 5 Verify** the version on every target and run a post-update generation.
- 6 Resume** only when the whole target set is verified.



Policy Refresh: Gate, Stage, Apply, Verify, Resume

Dynamo and ModelExpress provide per-worker steps; the RL framework owns the fleet barrier

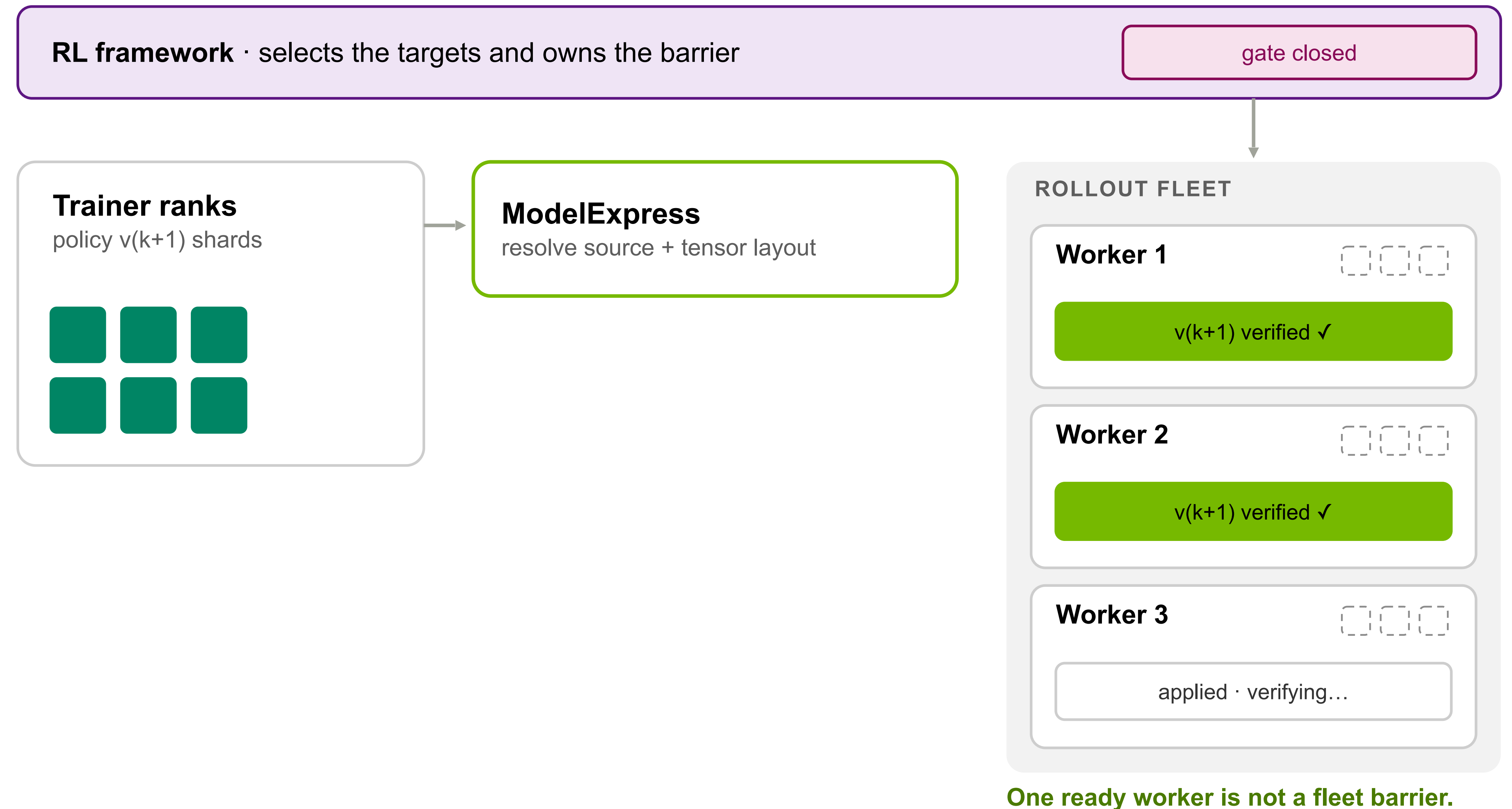
- 1 Gate.** The framework stops admitting new rollouts to the target workers (or enforces its own bounded staleness).
- 2 Pause** each target worker at a safe point.
- 3 Stage.** ModelExpress resolves the source and tensor layout; each inference rank pulls the shards it needs over NIXL, from trainer ranks, an S3 artifact or an already-updated peer.
- 4 Apply and reset.** The engine installs $v(k+1)$. KV cache built under $v(k)$ is invalid even for identical tokens, so every cache tier is cleared.
- 5 Verify** the version on every target and run a post-update generation.
- 6 Resume** only when the whole target set is verified.



Policy Refresh: Gate, Stage, Apply, Verify, Resume

Dynamo and ModelExpress provide per-worker steps; the RL framework owns the fleet barrier

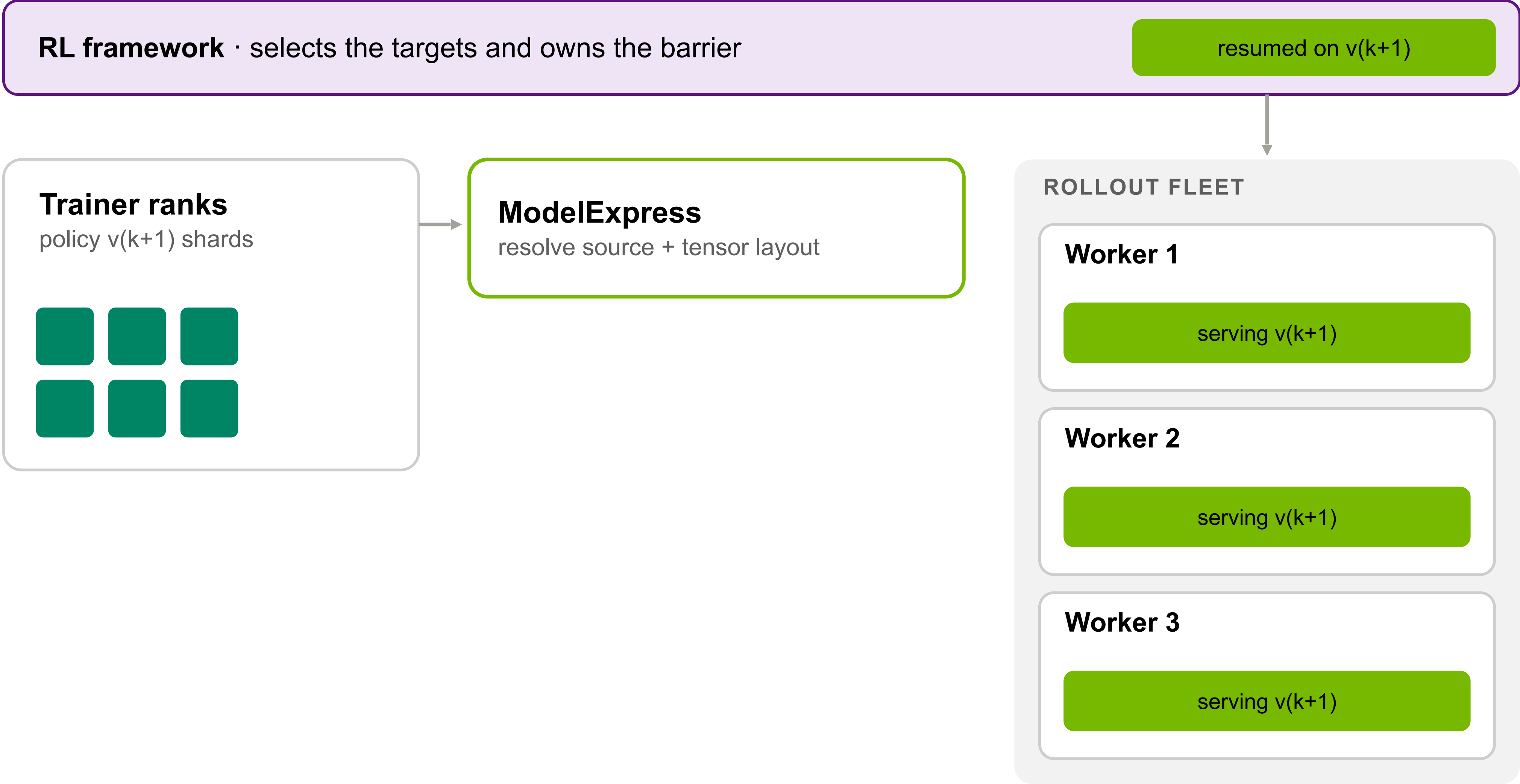
- 1 Gate.** The framework stops admitting new rollouts to the target workers (or enforces its own bounded staleness).
- 2 Pause** each target worker at a safe point.
- 3 Stage.** ModelExpress resolves the source and tensor layout; each inference rank pulls the shards it needs over NIXL, from trainer ranks, an S3 artifact or an already-updated peer.
- 4 Apply and reset.** The engine installs $v(k+1)$. KV cache built under $v(k)$ is invalid even for identical tokens, so every cache tier is cleared.
- 5 Verify** the version on every target and run a post-update generation.
- 6 Resume** only when the whole target set is verified.



Policy Refresh: Gate, Stage, Apply, Verify, Resume

Dynamo and ModelExpress provide per-worker steps; the RL framework owns the fleet barrier

- 1 Gate.** The framework stops admitting new rollouts to the target workers (or enforces its own bounded staleness).
- 2 Pause** each target worker at a safe point.
- 3 Stage.** ModelExpress resolves the source and tensor layout; each inference rank pulls the shards it needs over NIXL, from trainer ranks, an S3 artifact or an already-updated peer.
- 4 Apply and reset.** The engine installs $v(k+1)$. KV cache built under $v(k)$ is invalid even for identical tokens, so every cache tier is cleared.
- 5 Verify** the version on every target and run a post-update generation.
- 6 Resume** only when the whole target set is verified.



POLICY-UPDATE PAUSE · PRIME INTELLECT · GLM-5.2 (1.6 TB) ON 12 DGX H200 NODES



Median of equal 15-update windows; lower is better. sync/1 also changes consensus cadence, so the runs are not directly interchangeable. As cited in the Dynamo Ecosystem deck (Prime Intellect, 28 Aug 2026).

Open Opportunities

We are barely at the start of large-scale AI inference deployment

- **Agentic session aware management** needs propagation across the stack
- **Heterogenous model deployment** is least explored at scale
 - End to end agentic tasks need multiple models to closely exist together for performing the task
 - KV cache and other efficiency management
 - Fast data transfer mechanism
- **Heterogenous hardware management** is a new problem frontier
- **Fault tolerance and performance measurement** tools for inference at scale do not exist
 - AIPerf – a stab at performance measurement at scale from API standpoint of view
 - Agentic performance modelling?
 - Multimodal performance characterization?
 - Fault injection frameworks for inference?
 - Dynamo is working towards building something here but it is quite a task itself
- RL, Speculative decoding, diffusion-based reasoning model, ... all open for exploration
- ...

Recap

One question per component

#	COMPONENT	THE QUESTION IT ANSWERS	CORE IDEA
01	Frontend	<i>How do clients reach the fleet?</i>	One OpenAI API; template and tokenize once; discover workers.
02	Router	<i>Which worker runs this request?</i>	Cost = uncached prefill + current load.
03	Planner	<i>How many workers of each kind?</i>	Forecast load ÷ profiled per-engine capacity, separately for prefill and decode.
04	Agg vs. disagg	<i>Should prefill and decode share GPUs?</i>	Split them to stop interference and tune each phase; KV moves between pools.
05	KVCR	<i>Where else is this KV cache?</i>	Cluster-wide KV tiers with router hints, peer fetch and crash survival.
06	NIXL	<i>How do the bytes move?</i>	One async READ / WRITE API over every fabric and memory type.
07	Grove	<i>How do pods land on the cluster?</i>	Gangs, topology-aware placement, startup order, multi-level scaling.
08	Snapshot	<i>How fast can a new worker start?</i>	Checkpoint a warm worker once; restore CPU and GPU state everywhere.
09	AI Simulate	<i>How should I size my deployment given input traffic?</i>	Event driven Dynamo digital twin without using GPUs

Conclusions

Inference is an unsolved problem, and you can contribute and learn with us!

- <https://github.com/ai-dynamo/dynamo> (★)
- <https://docs.nvidia.com/dynamo/>
- Join the slack group: <https://ai-dynamo.slack.com>
- Invite Link - https://ai-dynamo.slack.com/join/shared_invite/zt-45l6z6ci8-PE97k6skMMaPsoygotKoVw#/shared-invite/email

References

Sources and further reading

DYNAMO DOCUMENTATION · V1.5.1

[Overall architecture](#)

[Frontend](#)

[Router design](#)

[Routing concepts](#)

[Planner design](#)

[Profiler](#)

[Disaggregated serving](#)

[Multinode orchestration \(Grove\)](#)

KVCR

github.com/ai-dynamo/kvcr (README, docs/design_overview.md)

AI Simulate

<https://github.com/ai-dynamo/aisimulate>

NIXL

github.com/ai-dynamo/nixl

[Enhancing distributed inference performance with NIXL \(NVIDIA blog\)](#)

GROVE

github.com/ai-dynamo/grove

[Streamline complex AI inference on Kubernetes with NVIDIA Grove](#)

SNAPSHOT

[NVIDIA Dynamo Snapshot: fast startup for inference workloads on Kubernetes \(May 2026\)](#)

[Snapshot digest \(Dynamo docs\)](#)

<https://arxiv.org/pdf/2609.25451>

BACKGROUND

[Introducing NVIDIA Dynamo \(disaggregation results, Mar 2025\)](#)

[DistServe, Zhong et al., OSDI 2024 \(arXiv:2401.09670\)](#)

