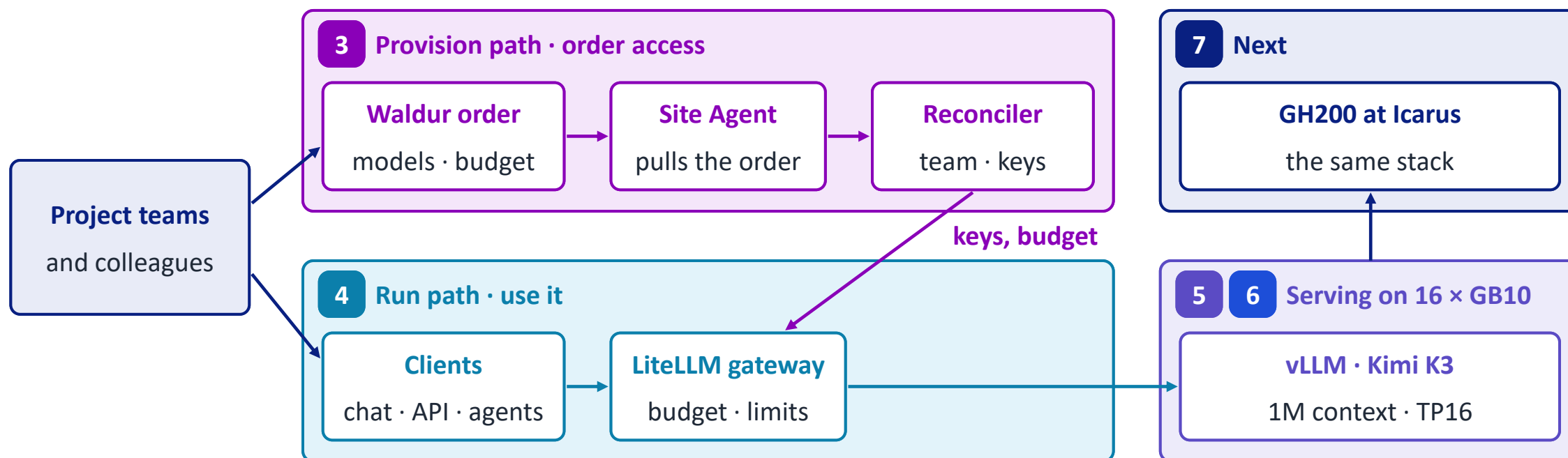


An Open Infrastructure Stack for Multitenant LLMs through the Pharos AI Factory

Ioannis Konstantinou

Research Associate, NTUA · Associate Professor, University of Thessaly
Open Local LLMs technical meeting · ΕΕΛΛΑΚ · NTUA, 9 October 2026



Contents

Click a topic to jump to its first slide

1	<u>Pharos AI Factory · where LLM access fits</u>	<u>slide 3</u>
2	<u>LLM access · the offering, the stack and the model catalog</u>	<u>slide 7</u>
3	<u>Provisioning path · ordering, Waldur, the Site Agent, the reconciler</u>	<u>slide 10</u>
4	<u>Run path · chat, API, agents and token metering</u>	<u>slide 12</u>
5	<u>Serving frontier models · prefill and decode, GB10 hardware, models, storage</u>	<u>slide 15</u>
6	<u>Making it fast · speculative decoding, lessons from the GB10</u>	<u>slide 19</u>
7	<u>Status and next steps · where we are, GH200 at Icarus, takeaways</u>	<u>slide 21</u>
+	<u>Backup · preliminaries, details of the main part, measurements</u>	<u>slide 25</u>

Pharos: the Greek AI Factory

One of the first EU AI Factories, built on national HPC

EU AI Factory

- One of the first seven EU AI Factories
- A trusted AI Factory for Southeast Europe

Infrastructure

- **DAEDALUS**: ≈89 PF, Lavrion
- New inference, cloud and storage systems

Sectors

- Health
- Culture & Language
- Sustainability

Two platforms

- **Technology** platform: this talk
- **Business** platform: the accelerator

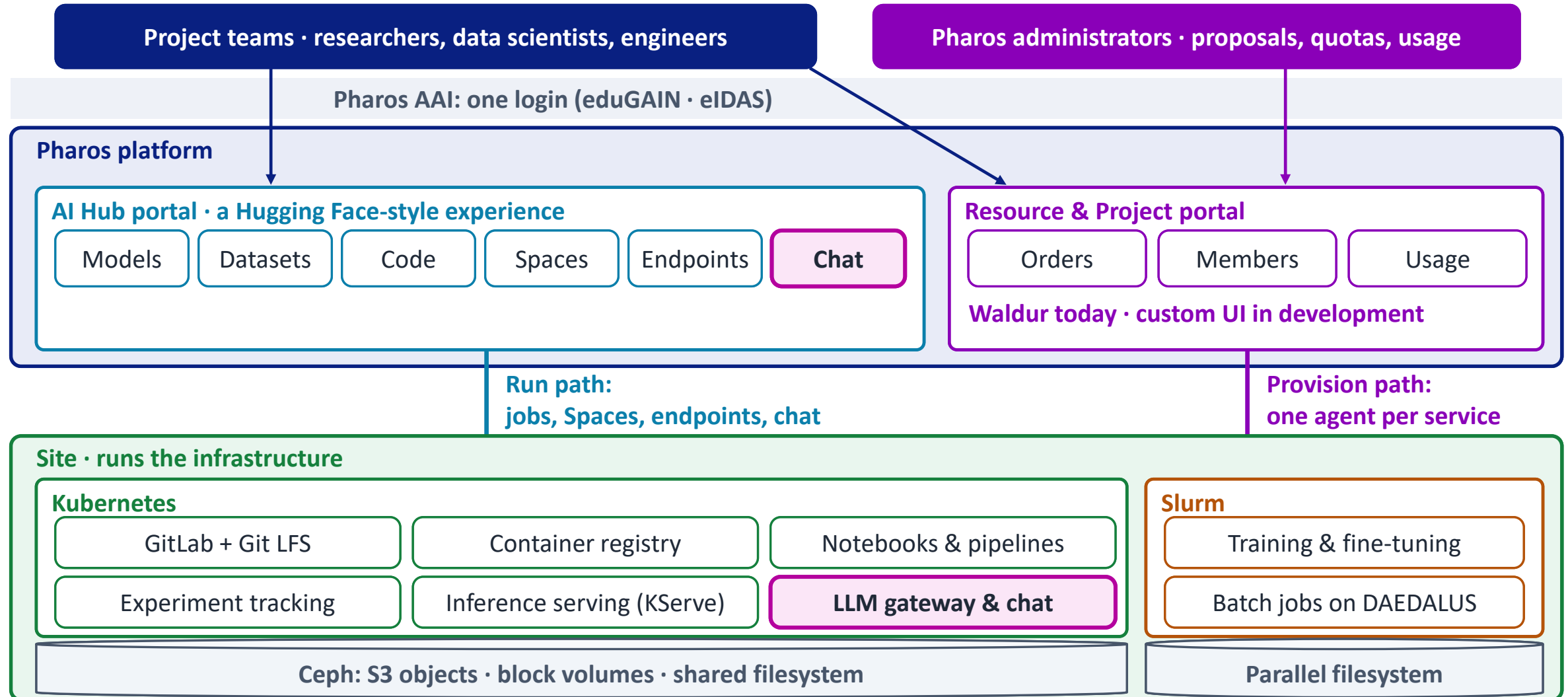


For

- Public sector
- Academia and research
- Startups and SMEs
- Industry

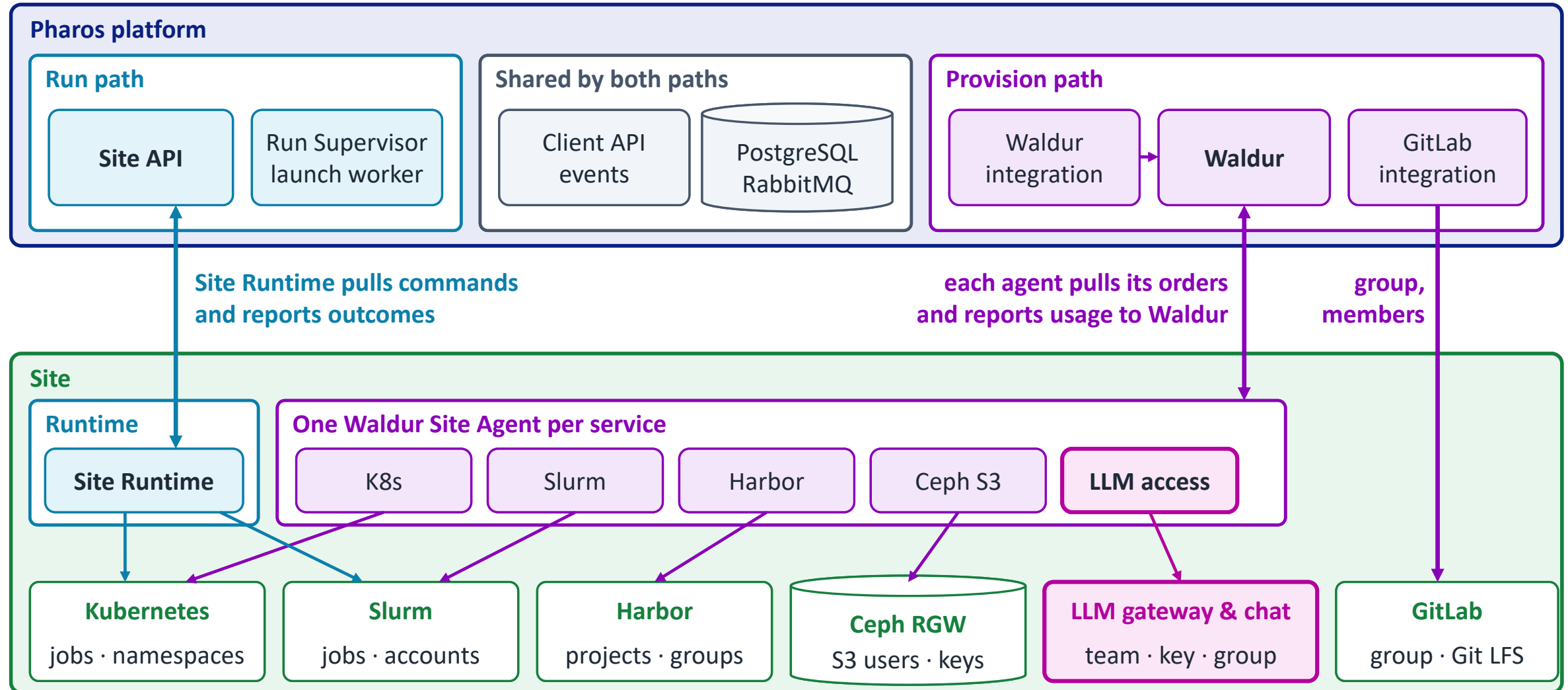
The platform at a glance

Two user groups, one login, two portals, and two paths down to the Site



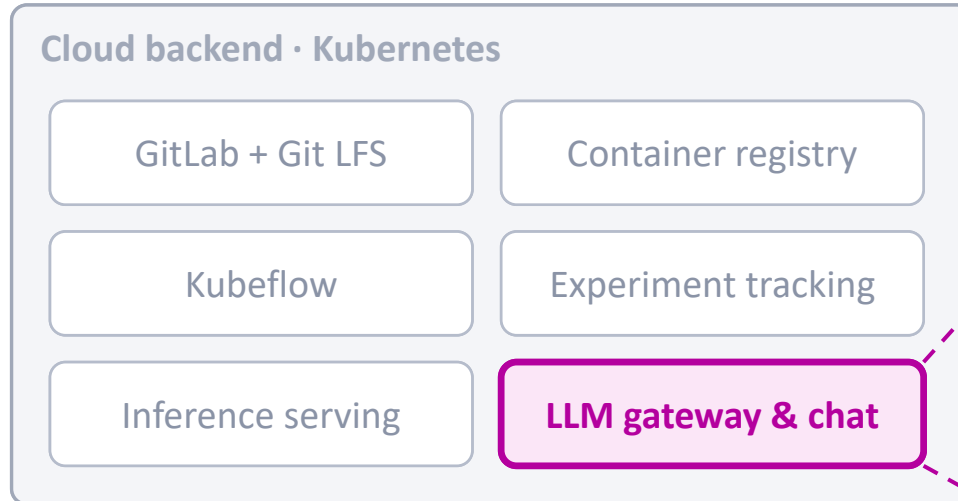
Two authorities, two paths

The platform decides; the Site runs the infrastructure and opens the connections



Zoom in: LLM access

The rest of this talk is one box of the platform



LLM access

Waldur decides **who** gets access and **how much**.
The Site serves the tokens.

The same pattern as every other offering: an order, a Site Agent, a resource

The LLM access offering

What a project gets when it orders LLM access

Chat

- In the browser, one Pharos login
- RAG over your own files

API

- OpenAI and Anthropic APIs
- A key per member

Agents

- Claude Code, Codex, SDKs
- Open models, up to 1M tokens

Fair share

- A token budget per project
- Limits per minute

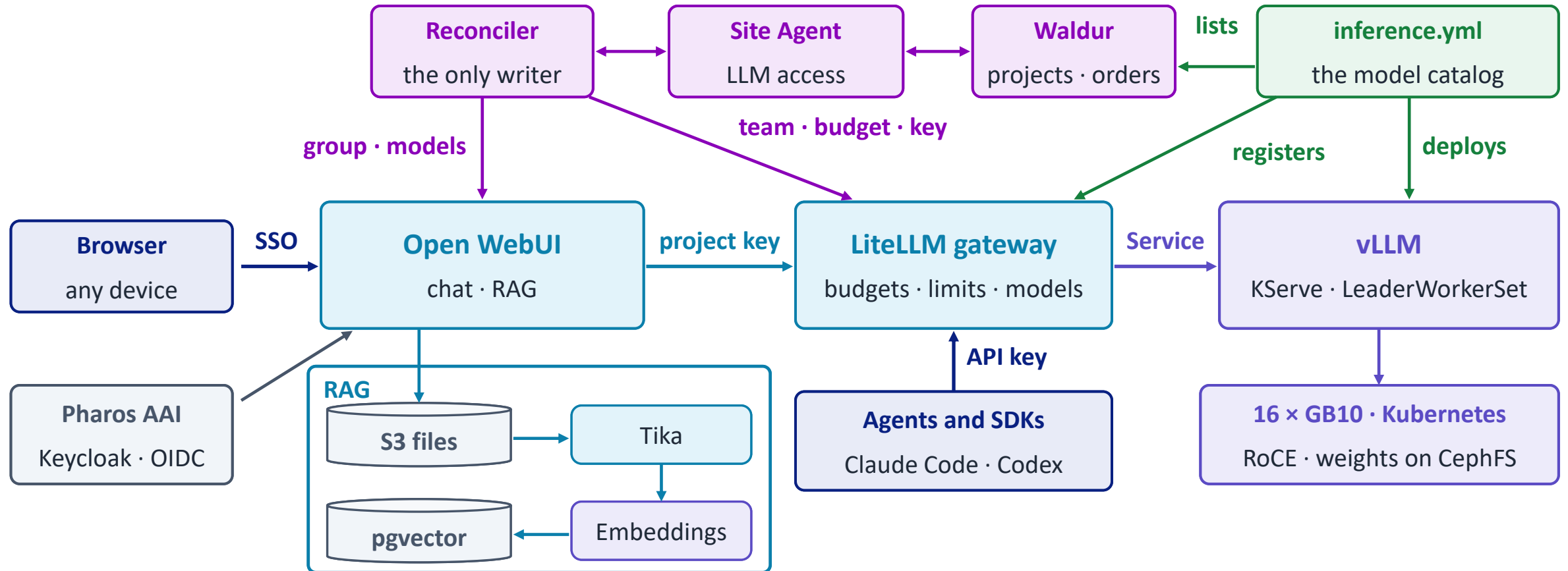
Every model runs on Pharos hardware: prompts and data stay on site

LIVE DEMO

ph-chat.cslab.ece.ntua.gr • ph-litellm.cslab.ece.ntua.gr

The stack, zoomed in

Every component is open source; arrows show who calls whom



Users reach only Open WebUI and LiteLLM; vLLM is never exposed

One catalog drives every deployment

inference.yml: one entry per model, with its image; read by three Ansible roles

group_vars/inference.yml · the Kimi K3 entry, abridged

```
- name: "kimi-k3"
  enabled: true
  served_name: "kimi-k3"    # the name clients use
  pvc_storage: "1600Gi"     # CephFS, shared
  vllm_image: "registry/kimi-k3@sha256:..." # our build
  node_count: 16
  tensor_parallel_size: 16
  max_model_len: 1048576
  litellm_model_info:
    # input_cost_per_token, output_cost_per_token:
    # the indicative prices on the order form
```

inference-deploy → vLLM

Runs it on 16 nodes, behind its own Service

litellm-deploy → LiteLLM

Registers it, with its output cap and prices

llm-access offering → Waldur

Lists it, with its prices, on the order form



Ordering LLM access for a project

The project lead picks the models and asks for one virtual budget for the whole team

1

Create a project, add colleagues

The team is the project's member list

2

Order LLM access

Models, a virtual budget, rate limits

3

An administrator approves

Never automatic for this offering

4

The team starts working

Chat and a personal API key for everyone

Order · LLM access · project "nlp-greek"

LLM models

(indicative cost per 1M tokens: input · output)

- ☒ kimi-k3 · 1M context, vision (1.00 · 4.00)
- ☒ qwen2.5-3b · small and fast (0.02 · 0.08)
- ☐ Qwen3-Embedding-0.6B · embeddings (0.01 · –)

Max budget

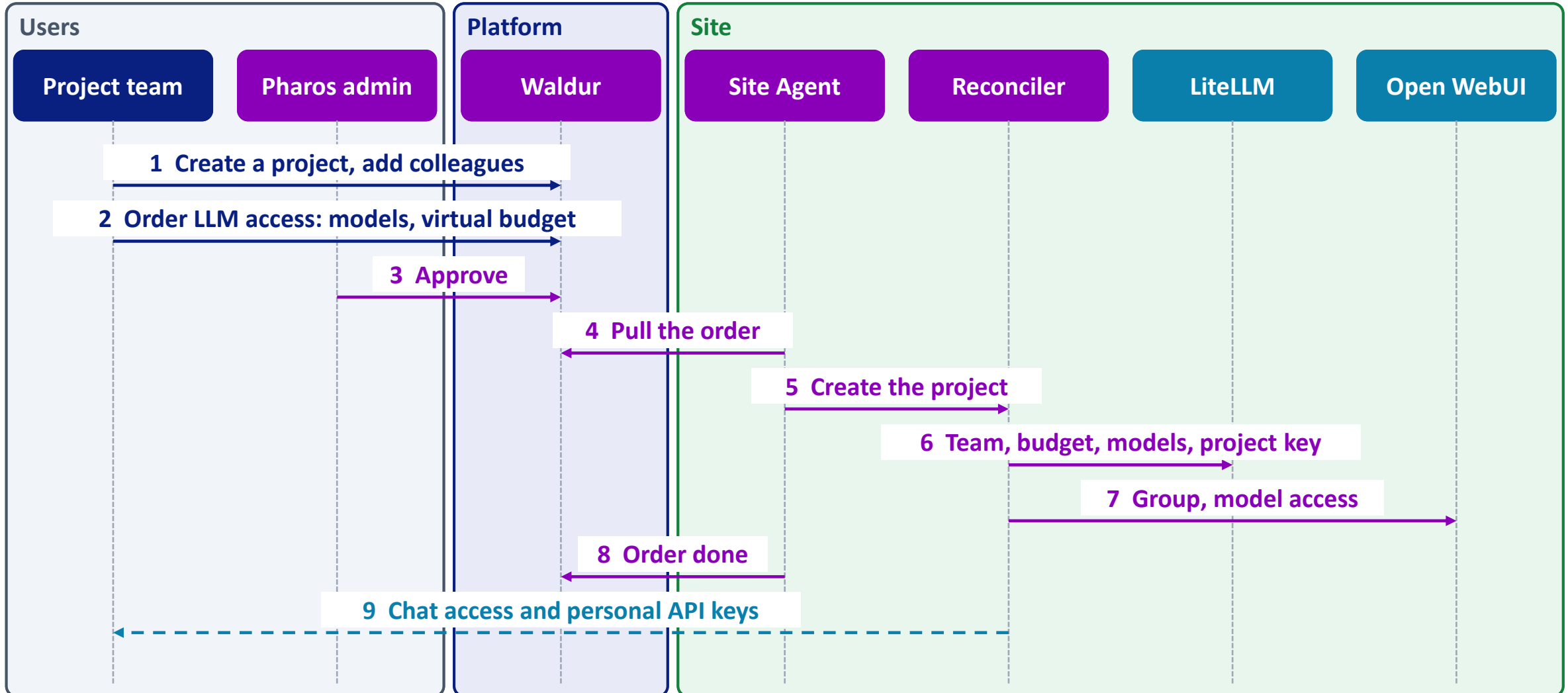
virtual units, per 30 days

Tokens per minute

Requests per minute

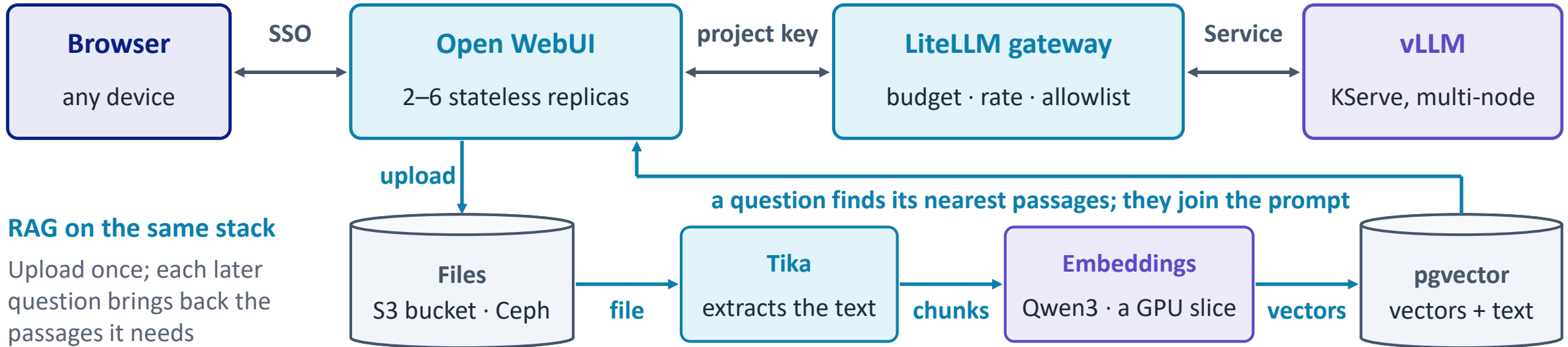
Provisioning path: from order to keys

Waldur → Site Agent → reconciler → LiteLLM and Open WebUI, in one idempotent call



Run path: chat

Every request carries the project key; the gateway checks it before routing



For the user

- Only their projects' models
- Chats and files stay private

Measured

- Stateless web tier: 2–6 replicas
- Ingestion 10–20× faster on a GPU slice

Run path: API and agents

The same gateway, the same key, the same budget as the chat

Claude Code on Kimi K3, through the Pharos gateway

```
export PHAROS_API_KEY=<REDACTED> # your personal key
export ANTHROPIC_BASE_URL=https://ph-litellm.cslab.ece.ntua.gr
export ANTHROPIC_AUTH_TOKEN=$PHAROS_API_KEY
export ANTHROPIC_MODEL=kimi-k3
claude
```

Codex: ~/.codex/config.toml (the key stays in PHAROS_API_KEY)

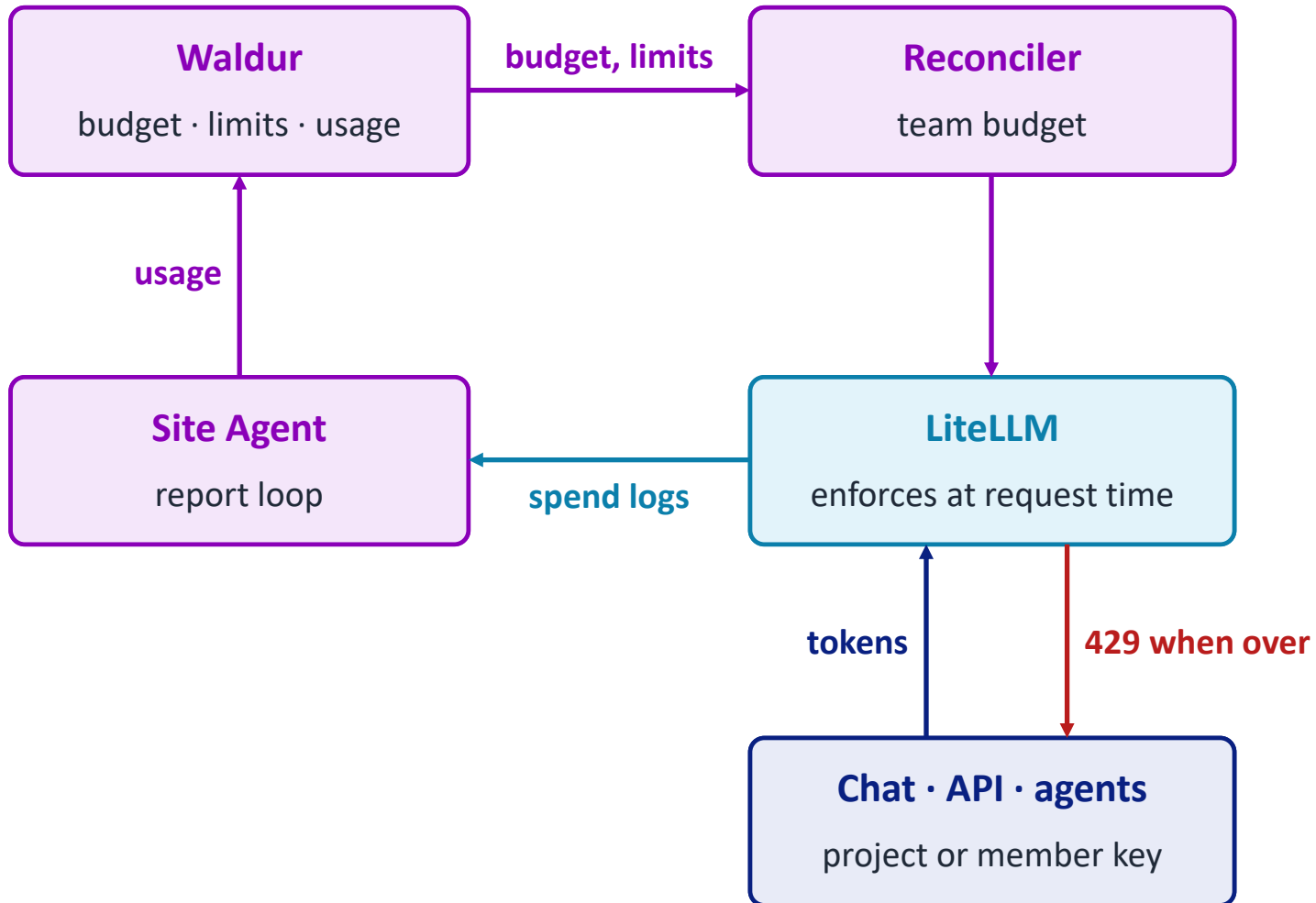
```
model = "kimi-k3"
model_provider = "pharos"
[model_providers.pharos]
base_url = "https://ph-litellm.cslab.ece.ntua.gr/v1"
env_key = "PHAROS_API_KEY"
wire_api = "responses"
```

One enforcement point

- OpenAI and Anthropic APIs: Codex and Claude Code
- Up to **1M tokens** of context, tools, vision
- No client reaches vLLM directly

Token metering: fair share, not billing

Budgets keep any one team from saturating the shared GPUs

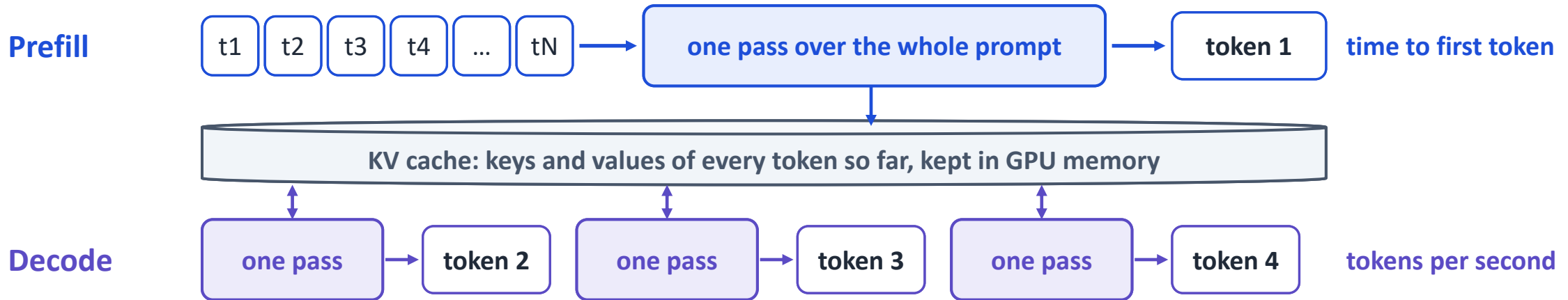


Why budgets

- One agent loop can saturate the shared GPUs
- Per project: a budget per 30 days, plus limits per minute
- Over budget: HTTP 429. Waldur sees totals, never prompts

How a model answers: prefill, then decode

Two phases with different bottlenecks; the rest of the talk follows from them



Prefill: compute-bound

- The whole prompt in one pass keeps the GPU busy

Decode: bandwidth-bound

- Every new token reads all active weights

Many users at once

- One pass serves them all: more tokens in total

Faster decode needs **more memory bandwidth per token** or **fewer passes per token**

Preliminaries: slide 25 →

The hardware: 20 × NVIDIA GB10

Plenty of memory per node, little bandwidth per node



The cluster at NTUA: 20 GB10 nodes, the switches on top

Per node

- 20-core Arm CPU + Blackwell GPU
- **128 GB** shared memory, ~122 GiB usable

In total

- **≈2.4 TiB** of memory
- Enough for 1.5 TB models

Measured

- **238 GB/s** memory bandwidth (273 nominal)
- **~97 TFLOP/s** BF16 GEMM

Fabric

- One 200 GbE link per node, 4 switches
- RoCE · MTU 9000

Decode is bandwidth-bound: **many nodes per token, fewer passes per token**

Models we have run on the cluster

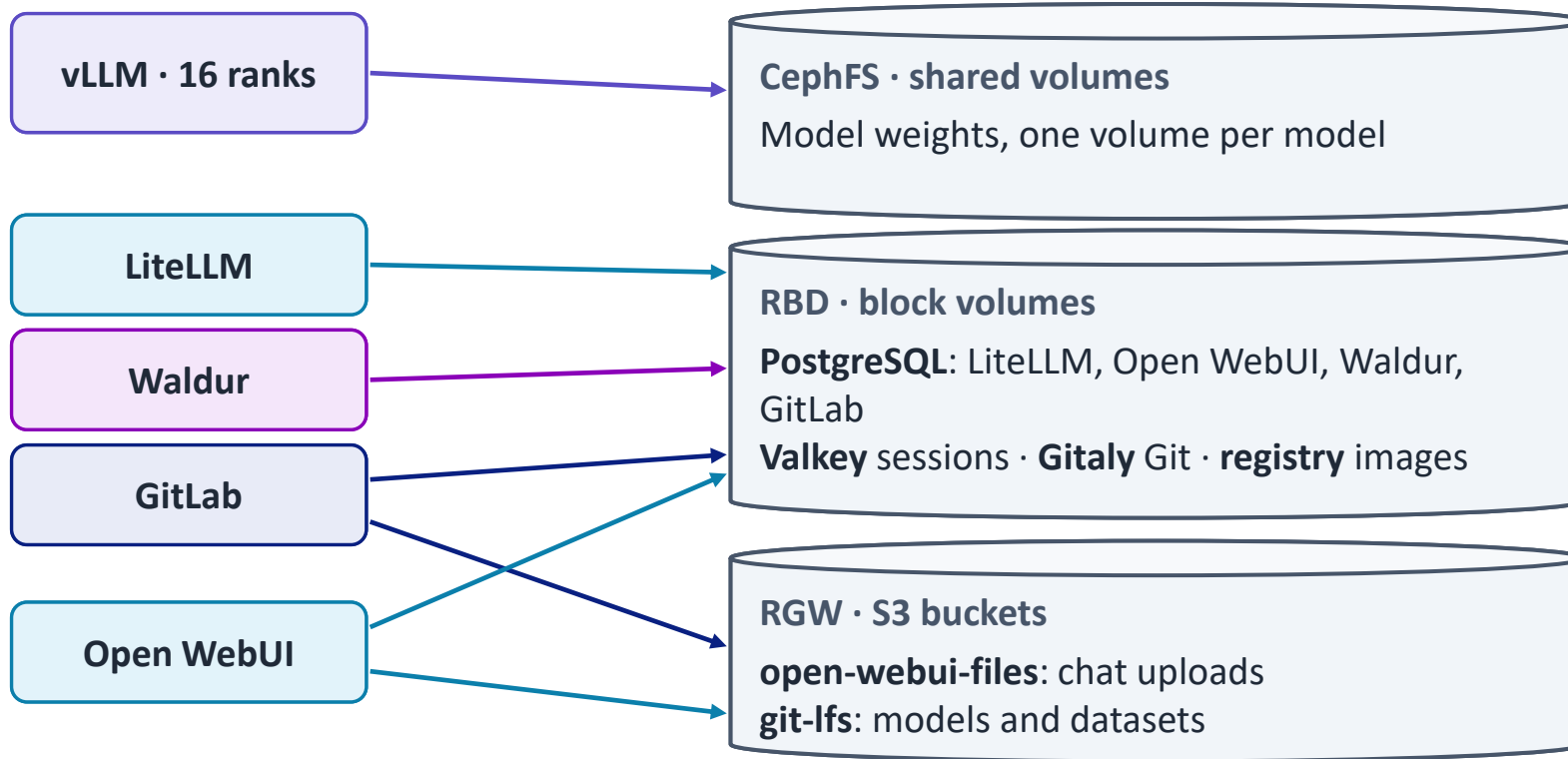
Live today, and tested before: switching is a catalog change

Model	Type	Nodes	Status
Kimi K3	MoE, 1M context	16	live
Qwen2.5-3B · Qwen3-Embedding	task model · RAG	1 each	live
GLM-5.3 and 5.2 NVFP4	MoE, 743B	8	tested
MiniMax-M3 NVFP4	MoE, multimodal	8	tested
Qwen3.8-2.4T-A95B NVFP4	MoE, 2.4T	16	tested
DeepSeek-V4-Pro and Flash	MoE, 1M context	16 · 2	tested
Qwen2.5 0.5B–32B, Qwen2-VL	dense ladder	1	benchmarked

Switching models: toggle **enabled** and redeploy; the weights stay on Ceph

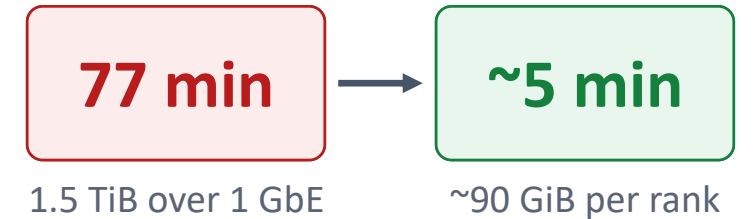
Storage: what lives where

One Rook-Ceph cluster on the GB10 nodes, three ways in



Rook-Ceph	Disks in Ceph	Raw	Usable at 3×	Free now
GB10 site	20 × 2.7 TiB NVMe	55 TiB	≈18 TiB	8.3 TiB
Icarus	75 × 7 TiB SSD	524 TiB	≈175 TiB	166 TiB

Loading the weights



What we learned

- Kimi K3: 1.45 TiB, downloaded once
- All traffic was on the 1 GbE NIC
- Per-rank shards cut reads 17×

Decode: speculative decoding

A small drafter guesses, the big model checks several tokens in one pass

One token per pass

target pass → t1

target pass → t2

target pass → t3

target pass → t4

A pass costs about the same for 1 or 4 tokens

Speculative

d1

d2

d3

one target pass verifies d1 d2 d3

✓ ✓ ✗ → 3 tokens

Kimi K3 · DSpark

- A small drafter, 3 tokens
- ≈2.2 tokens per pass

GLM-5.3 · MTP

- Draft heads in the model, 5 tokens
- ~86% accepted, ~50 tok/s

Result on Kimi K3

- 1.5 → 12–16 tok/s per user
- 2.2× aggregate at 8 users

Today on Kimi K3: **13 tok/s** for one user, **29 tok/s** aggregate at 8 users

GB10 lessons

Recipes written for one x86 server, run on 16 ARM nodes

No ready image

- 28 Sep 2026: no public Kimi K3 image for ARM64
- Built natively on a GB10 that day

Recipes assume one box

- Launchers expect 16 GPUs in one server
- Here: NCCL over RoCE, 16 nodes

Unified memory

- What the OS and NCCL hold, the model loses
- NCCL buffers capped, free RAM guarded

Silent slow paths

- Traffic on the 1 GbE NIC, RDMA fell back to TCP
- Nothing crashes: measure the path

[Details: slide 35 →](#)

Where we are

Serving and access are live; the Waldur offering moves to the GB10 site next

Kimi K3 on 16 GB10 · Qwen task model · embeddings

LIVE

LiteLLM gateway · Open WebUI chat with SSO and RAG

LIVE

Claude Code and Codex on Kimi K3

IN USE

Waldur offering · Site Agent · reconciler

VALIDATED · PORTING

Per-project chat connections

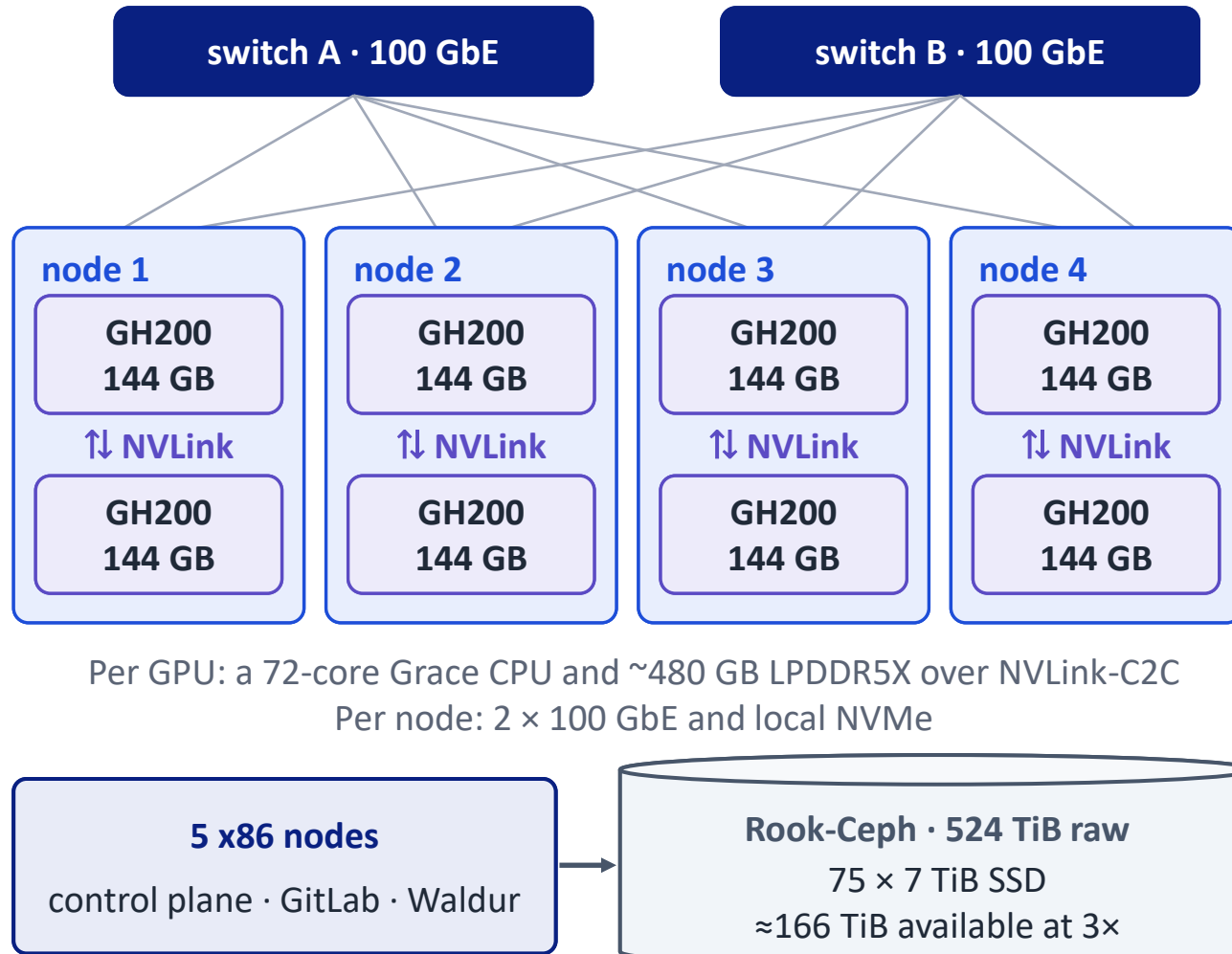
WITH THE OFFERING

GH200 inference at Icarus

NEXT

Next: GH200 inference at Icarus

A second Pharos site, the same stack, a different hierarchy



The plan

- Already there: Kubernetes, Ceph, GitLab, Waldur, KServe
- TP2 over NVLink inside a node; PP or EP across nodes
- Same catalog, gateway and offering

8 GPUs

GH200, over
4 nodes

1,152 GB

HBM3e in
total

≈5 TB/s

per GPU,
~18x a GB10

Takeaways

What we would tell a team that wants to do the same

1 **Open end to end:** Waldur, Kubernetes, vLLM, LiteLLM, Open WebUI, Ceph, GitLab

2 **One control plane** for budgets, **one gateway** for chat, API and agents

3 **Token budgets as fair share** on shared GPUs

4 **Frontier open models, 1M context**, on 20 GB10 nodes

5 **Measure before tuning:** our problems all looked like something else

Recipes, roles and catalog will be released as **open source**

Thank you

Questions?

Ioannis Konstantinou · ikons@cslab.ece.ntua.gr



Co-funded by
the European Union



PHAROS
GREECE



ΕΛΛΗΝΙΚΗ ΔΗΜΟΚΡΑΤΙΑ
Υπουργείο Ψηφιακής Διακυβέρνησης
και Τεχνητής Νοημοσύνης

Preliminaries: why prefill and decode differ

The same weight matrix W , multiplied by a whole prompt or by one new token

Prefill

$$\begin{array}{c}
 \boxed{} \\
 X \cdot T \times H
 \end{array}
 \times
 \begin{array}{c}
 \boxed{} \\
 W \cdot H \times D
 \end{array}
 =
 \begin{array}{c}
 \boxed{} \\
 Y \cdot T \times D
 \end{array}$$

Many tokens per weight

Each weight read from memory serves all T prompt tokens: the tensor cores stay busy, so prefill is compute-bound

Decode

$$\begin{array}{c}
 \boxed{} \\
 x \cdot 1 \times H
 \end{array}
 \times
 \begin{array}{c}
 \boxed{} \\
 W \cdot H \times D
 \end{array}
 =
 \begin{array}{c}
 \boxed{} \\
 y \cdot 1 \times D
 \end{array}$$

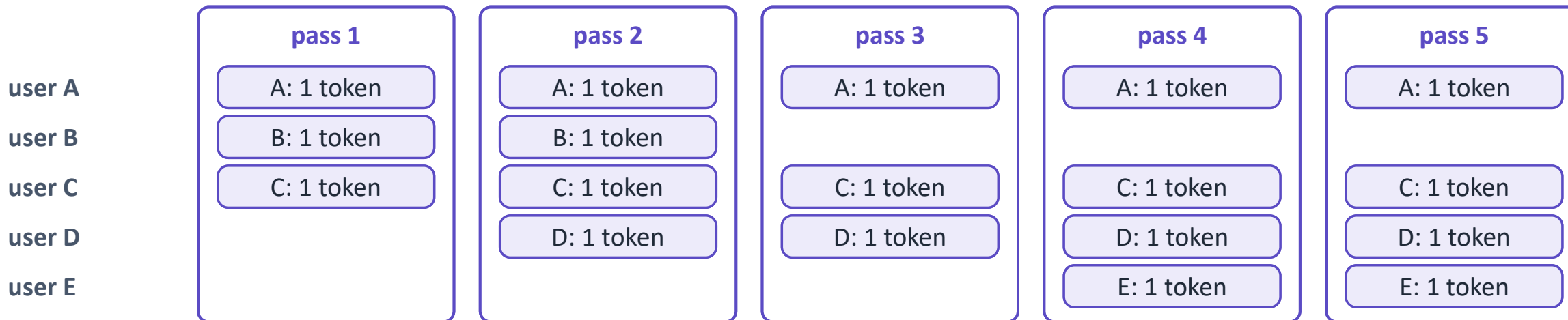
One token per weight

Each weight is read for a single token: the GPU waits on memory, so decode is bandwidth-bound. B users in one pass make x a $B \times H$ matrix again

	Prefill	Decode
Tokens per pass	the whole prompt	one per running request
Limited by	compute	memory bandwidth, network latency
KV cache	written for every prompt token	read, one entry added
Users feel it as	time to first token	tokens per second

Preliminaries: many users in one pass

Continuous batching: every decode pass carries one token of each running request



Batch of 1

- 100 ms per pass
- 10 tok/s for the user and in total

Batch of 32

- 150 ms per pass
- 6.7 tok/s per user, 213 tok/s in total

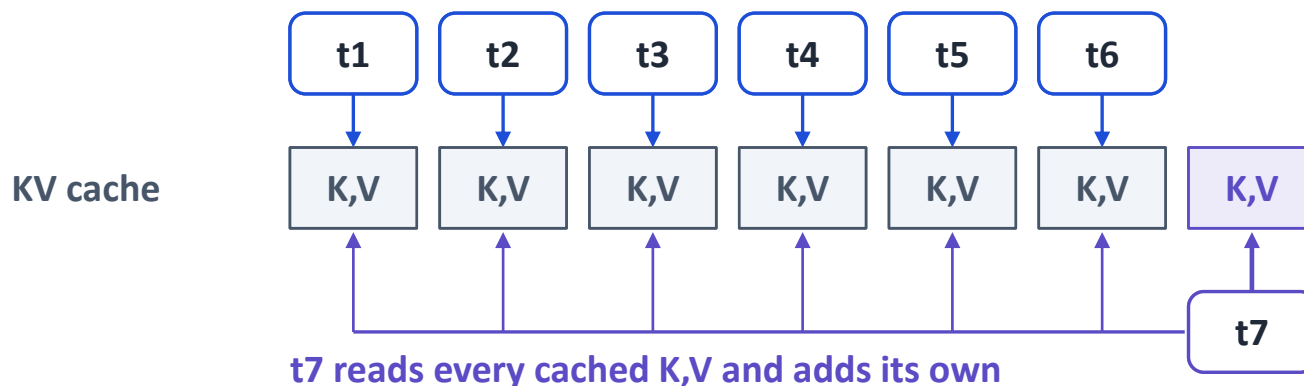
Kimi K3, measured

- max-num-seqs 1 → 8
- 8 users: 15.8 → 25.6 tok/s in total

A pass costs about the same for 1 or 32 tokens: batching multiplies the total, and each user gets a slightly slower stream

Preliminaries: KV cache and prefix caching

Keep the keys and values of past tokens; reuse them when a chat resends its history



Size

Per token: $\text{layers} \times 2 \text{ (K and V)} \times \text{KV heads} \times \text{head size} \times \text{bytes}$. It grows with every token of every running request; MLA keeps one compressed vector per layer instead

Turn 1

system prompt · history · question 1

Turn 2

the same prefix: taken from the cache

new: prefill

Kimi K3, measured

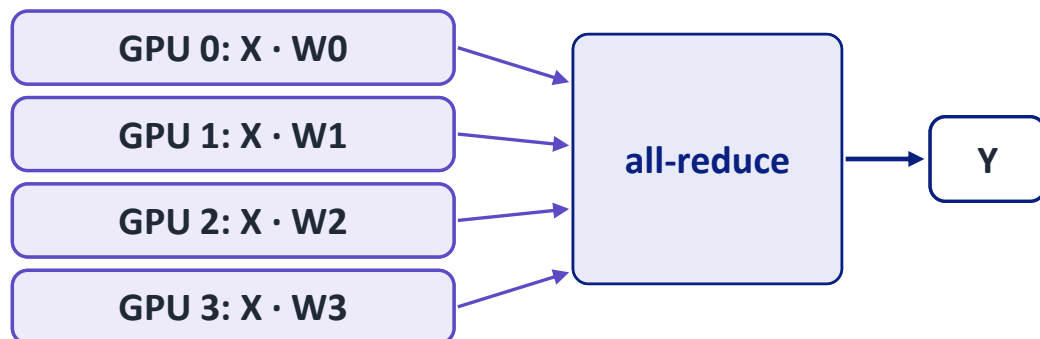
39,037-token chat, 36,864 tokens from the cache: first token in 6.1 s instead of 81.8 s

Prefix caching saves prefill, not decode: it pays off for long system prompts, RAG templates and agents

Preliminaries: tensor and pipeline parallelism

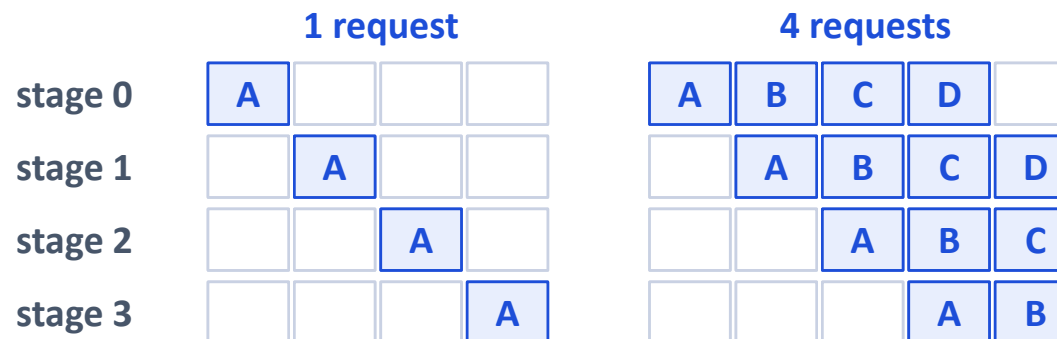
TP splits every layer across GPUs; PP splits the stack of layers into stages

TP · every layer split



In every layer, for every token: all GPUs work at once, then exchange partial results over the network

PP · the layers split into stages



With one user each stage waits for the one before it: pipeline bubbles

TP

- Splits the memory and adds bandwidth to every token
- Costs a collective in every layer, so network latency matters

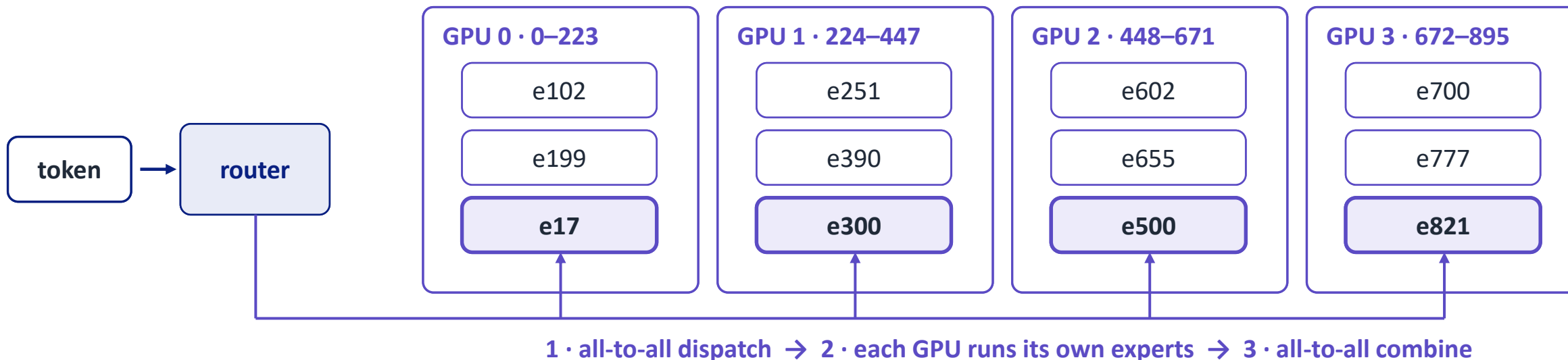
PP

- Activations cross each stage boundary once
- Idle stages with few users; busy with many

Nodes = TP × PP · Kimi K3 runs TP16 × PP1 · DeepSeek-V4-Pro was fastest at TP2 × PP8

Preliminaries: mixture of experts and expert parallelism

A router picks a few experts per token; the experts stay on their GPUs, the tokens travel



Kimi K3

- 896 routed experts, 16 per token
- Huge in total, small active part

GLM-5.3 on 8 GB10

- TP8 for attention, EP8 for the experts
- On the same 8 GPUs

The price

- Two all-to-all exchanges in every MoE layer
- Few tokens per expert at small batch

Batching helps MoE twice: bigger expert matrix products and bigger, cheaper all-to-all messages

Preliminaries: four resources that limit speed

Read each measurement against the resource that ran out

Memory capacity

- Weights, KV cache and buffers must fit
- Ours: Kimi K3's 1.45 TiB needs 16 nodes; NCCL buffers took ~16 GiB per node

Memory bandwidth

- Decode at small batch reads every weight for few tokens
- Ours: 238 GB/s per GB10 (273 nominal), hence TP16 and speculative decoding

Compute

- Long prompts keep the tensor cores busy
- Ours: prefill at 0.5–1.4K tok/s; 4- and 8-bit formats help

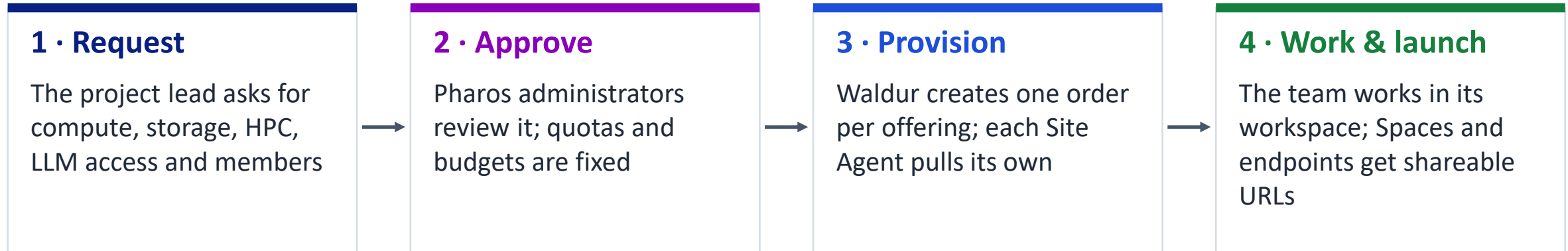
Network

- TP collectives in every layer, EP all-to-all; latency more than bandwidth at small batch
- Ours: ~210 ms per step on 16 nodes, mostly collectives; RDMA instead of TCP mattered

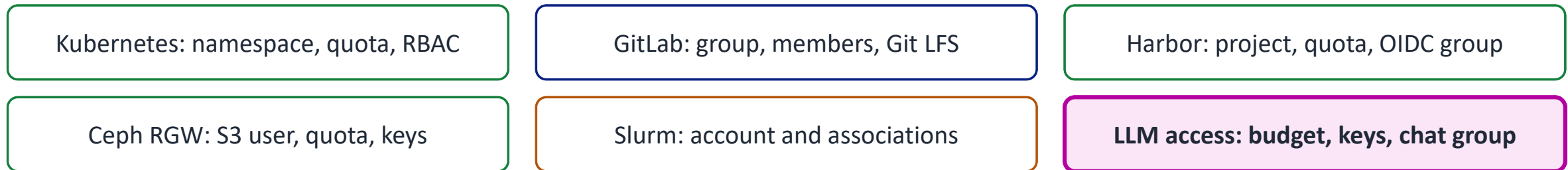
If decode speeds up a lot with more users, the limit was bandwidth or latency, not compute

From an approved order to a working project

One order per offering, one Site Agent per service



Each service gets its resource: five through their Site Agents, GitLab with the project



Members are synced on a timer · usage flows back to Waldur · quotas grow with a new order

Fitting the model: TP, PP, EP and DCP

One GPU per node, so nodes = $TP \times PP$

TP · tensor

- Every layer split across nodes
- All ranks work on each token

PP · pipeline

- Layers split into stages
- Fits, but each token crosses every stage

EP · expert

- MoE experts spread across ranks
- All-to-all in every layer

DCP · context

- KV cache sharded across ranks
- A 1M-token context fits

Model	Nodes	Layout	Context
Kimi K3 (≈1.45 TiB, multimodal)	16	TP16 · DCP16 · DSpark drafter	1M
GLM-5.3 NVFP4 (743B MoE, 39B active)	8	TP8 · EP8 · MTP5	512K
DeepSeek-V4-Pro	16	TP2 × PP8, best of 5 layouts	1M

TP is bounded by the attention heads (96 heads: TP16 at most on 20 nodes) · a deeper pipeline lost for single-user decode · some drafters need PP = 1

Prefill: getting to the first token

Long prompts, chat histories and agent sessions

What we use

- **Chunked prefill:** 4K-token chunks, interleaved with other users' decode
- **Low precision:** MXFP4 experts, MXFP8 projections, NVFP4 (GLM), FP8 KV cache
- **Kernels per phase:** FlashKDA and FlashAttention for prefill, B12X MLA for decode
- **Prefix caching:** a chat re-sends its whole history every turn
- Prefill rates: Kimi $\approx$ 490–610 tok/s, GLM-5.3 $\approx$ 1.2–1.4K tok/s

Time to first token, repeated 39K-token chat

without cache

81.8 s

6.1 s

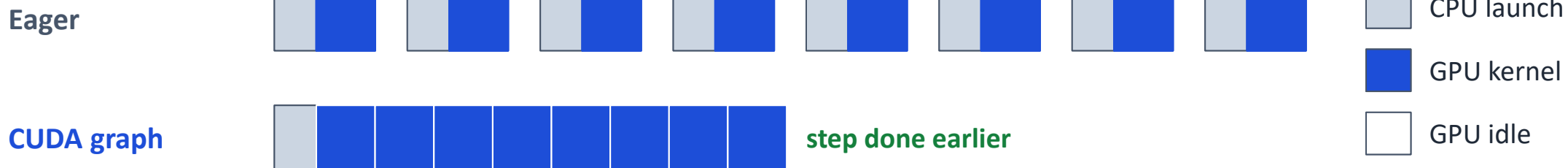
36,864 tokens reused from the prefix cache

Getting there

- 0 cache hits under DCP16
- Traced to block alignment: 12,288 vs 4,608 tokens
- Fixed upstream; image rebuilt

CUDA graphs: why they are fast

Record a whole decode step once, replay it with one launch



Why fast

- A decode step is thousands of small kernels
- One launch replays them: no Python, no gaps

Fixed shapes

- Captured sizes **4** and **8**
- = $(3 \text{ drafts} + 1) \times 1 \text{ or } 2$ sequences

Our bug

- cuBLAS first used on a side stream inside the capture
- A one-line guard fixed it

At 16 nodes

- ~210–220 ms per step, mostly collectives
- Graphs give back the launch time only

Graphs are on: capture takes 15 s and 0.21 GiB · most of the speed came from TP16, speculation and concurrency

GB10 lessons (1/2): ARM64 and one-box assumptions

Most problems came from a new architecture and recipes written for a single server

ARM64 and sm_121

- On 28 Sep 2026 the public Kimi K3 vLLM image (tag of 18 Sep) was amd64-only; the newest arm64 one (27 Sep) lacked the attention kernel Kimi needs
- Built natively on a GB10 that day: 408 compile steps, done 11:31 UTC

One box assumed

- Launchers expect 16 GPUs in one server
- CUDA IPC and PCIe shortcuts disabled; NCCL over RoCE instead

Valid is not loadable

- The first Kimi drafter had the wrong architecture for the attention backend
- Replaced with the native one

Silent failures

- The wrong tool parser turns tool calls into plain text
- Nothing crashes: check the output, not the logs

GB10 lessons (2/2): unified memory and the network

What the OS and NCCL hold, the model loses

NCCL buffers

- ~16 GiB per rank in 64 channels
- Capped at 8 channels, 1 MiB

An interlock, not a knob

- Lowering the GPU-memory fraction turned a clean error into a node OOM

Page cache

- Shares the same RAM as the GPU
- A watcher stops a boot below 3 GiB free

1 GbE by default

- Calico picked the management NIC
- NCCL needed host networking on 200 GbE

RDMA fell back to TCP

- A library mismatch in the image
- Rebuilt: NCCL now runs over RoCE

Method

- One change per boot, measured
- A 16-node boot takes ~15 min

Why a 1M-token context fits (Kimi K3)

Hybrid attention keeps the KV cache small

Linear attention

- 69 of 93 layers use KDA
- Their state does not grow with the context

MLA for the rest

- 24 layers $\times$ 576 values $\times$ 2 bytes
- $\approx$ **27 KiB per token**

1M tokens

- $\approx$ 27 GiB for the whole model
- Sharded by DCP16: a pool of **1.06M tokens**

A conventional multi-head model of this size would need about **4.5 TB** for the same context

DeepSeek-V4-Pro: five layouts on 16 nodes

Single-user decode and 26K-token prefill

Layout	Executor	Decode tok/s	26K prefill tok/s
TP8 × PP2	Ray	4.08	460–665
TP4 × PP4	mp	4.83	973–1,027
TP2 × PP8	mp	5.46	1,361
TP1 × PP16 (RDMA)	mp	4.22	~1,822
TP4 × PP4	Ray	engine dies	–

Narrower TP means a deeper pipeline; with one user there is nothing to fill the stages

Kimi K3: concurrency before and after

max-num-seqs 1 → 8; aggregate tok/s and sequences running

Users	Before: tok/s	Before: running	After: tok/s	After: running
1	16.1	1	10.5	1
2	15.4	1	15.4	2
4	15.6	1	23.1	4
8	15.8	1	25.6	4

By the engine's own counters one user did not get slower (209 vs 223 ms per step); the 1-user client figures differ by run-to-run spread

GLM-5.3 NVFP4 on 8 GB10

TP8 · EP8 · MTP5 · one request at a time, 64 output tokens, p50 of 10 runs

Context	Time to first token	Prefill tok/s	Decode tok/s
~0.5K	0.81 s	615	50.3
~7.4K	5.3 s	1,384	50.6
~29K	22.5 s	1,302	49.2
~117K	100.7 s	1,163	45.7

MTP acceptance 85–88% at every context length

Document ingestion: CPU vs one GPU slice

Same 22 KB Markdown file, distinct users, upload to vectors

Concurrent users	CPU p50	GPU slice p50	Files/s: CPU → GPU
1	9.2 s	1.0 s	0.10 → 1.02
2	16.0 s	1.5 s	0.12 → 1.40
4	22.5 s	1.6 s	0.12 → 2.32
8	40.0 s	2.8 s	0.12 → 2.67

CPU: two embedding replicas with 4 CPUs each · GPU: one replica on one time-sliced GB10 share