

Beyond One Model per GPU

Agent-Driven Placement for Maximum GPU Utilization on Kubernetes

KubeCon + CloudNativeCon North America · Salt Lake City · 10 November 2026

Presenters

NVIDIA and AWS built and measured this together.



Weiqi Sun

Solutions Architect

NVIDIA



Ratnopam Chakrabarti

Sr. Solutions Architect

AWS



Ajay Vohra

Generative AI Applied Engineer

AWS



Vivek Kolasani

Solutions Architect

NVIDIA



Felipe Garcia

Solutions Architect

NVIDIA

Beyond One Model per GPU. Tutorial · 10 November 2026 · 2:50 – 4:10 PM

Tutorial outline

1. Problem statement (12 min)
2. GPU sharing and why MPS (13 min)
3. Agent driven pipeline (17 min)
4. Deploying using KAI scheduler (8 min)
5. Experiment results (22 min)
6. Running it yourself (3 min)

Everything rests on one example. Part 1 introduces a real fleet of fifteen services. Part 5 measures it.

PART 1

Problem statement

A platform team needs to deploy M models on N GPUs. How to do it using the least number of GPUs, while meeting your service level objectives (SLOs)?

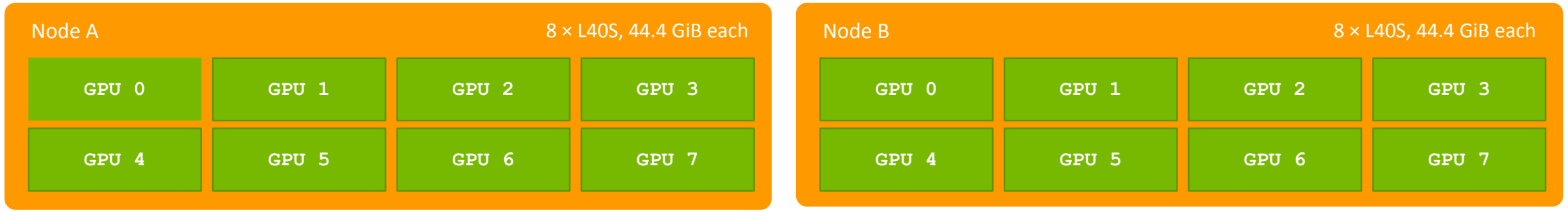
Example problem

FIFTEEN MODELS

ITL target · TTFT p95 target · Input sequence · Output sequence · Request rate

q-reason	Qwen3-30B-A3B	60 ms	2.0 s	1024	1024	1.0/s	**q-chat**	Qwen3-14B	55 ms	1.2 s	1024	256	1.0/s
q-route	Qwen3-0.6B	20 ms	0.3 s	512	8	8.0/s	**q-guard**	Qwen3-0.6B	80 ms	0.5 s	1024	16	6.0/s
q-title	Qwen3-0.6B	150 ms	3.0 s	2048	24	1.0/s	**g-reason**	GLM-4.5-Air-FP8	80 ms	4.0 s	1024	1024	0.5/s
g-route	Qwen3-0.6B	20 ms	0.3 s	512	8	8.0/s	**g-guard**	Qwen3-0.6B	80 ms	0.5 s	1024	16	6.0/s
g-title	Qwen3-0.6B	150 ms	3.0 s	2048	24	1.0/s	**g-followup**	Qwen3-0.6B	50 ms	0.8 s	512	256	2.0/s
m-docqa	Qwen3.5-9B	60 ms	12 s	8448	512	0.5/s	**m-caption**	Qwen3.5-4B	60 ms	2.0 s	1088	96	6.0/s
m-tag	Qwen3.5-2B	30 ms	0.4 s	304	16	8.0/s	**m-alt**	Qwen3.5-0.8B	25 ms	0.3 s	320	48	6.0/s
m-thumb	Qwen3.5-0.8B	20 ms	0.25 s	320	8	12.0/s							

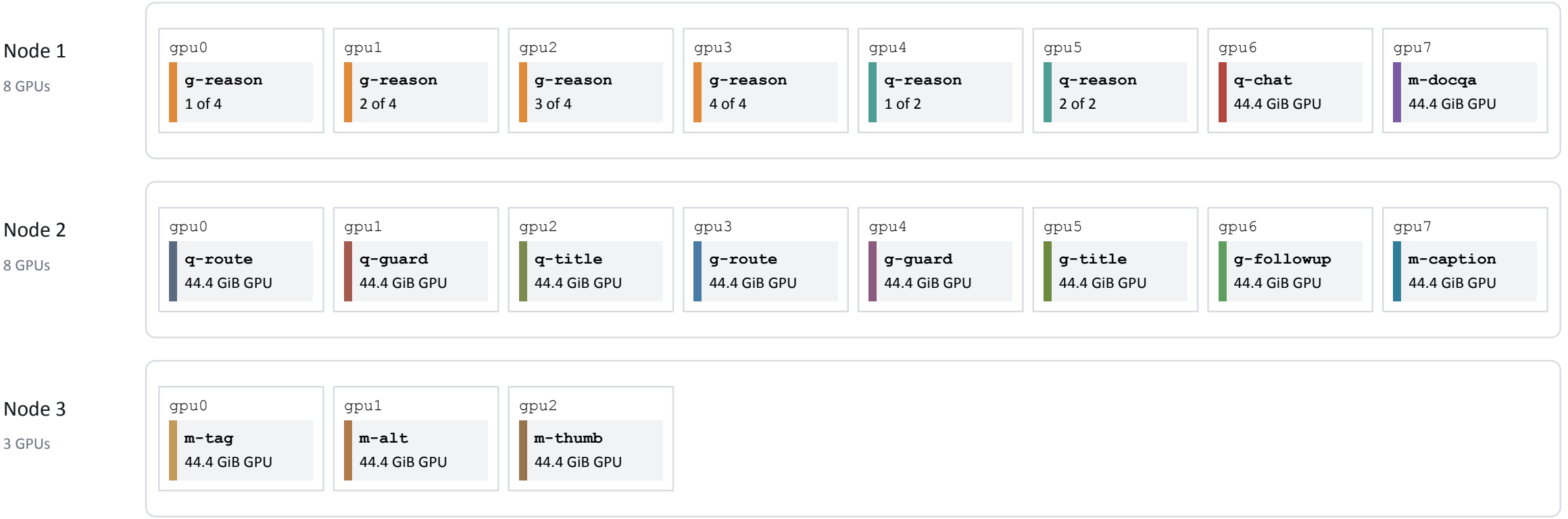
TWO MACHINES, EIGHT GPUs EACH



Problem statement: Can all fifteen models run on these two L40S GPU nodes, and still meet every SLO?

Baseline solution

One model shard per GPU. Larger models use Tensor Parallelism.



Assumption: no model needs pipeline parallelism across nodes.

Why not just use KAI?

Kubernetes schedulers that split a GPU already exist. KAI is one. So why is there anything left to build?

HERE IS WHAT KAI ASKS YOU FOR

```
containers:
  - name: m-docqa
    gpu-memory: ?????
  - name: q-chat
    gpu-memory: ?????
```

And a second blank it never shows you: which services end up on the same GPU.

> **You cannot guess gpu-memory for each model shard**
>
> The gpu-memory for each shard depends on the model's SLO and on which models share its GPU.

WHAT KAI GIVES YOU

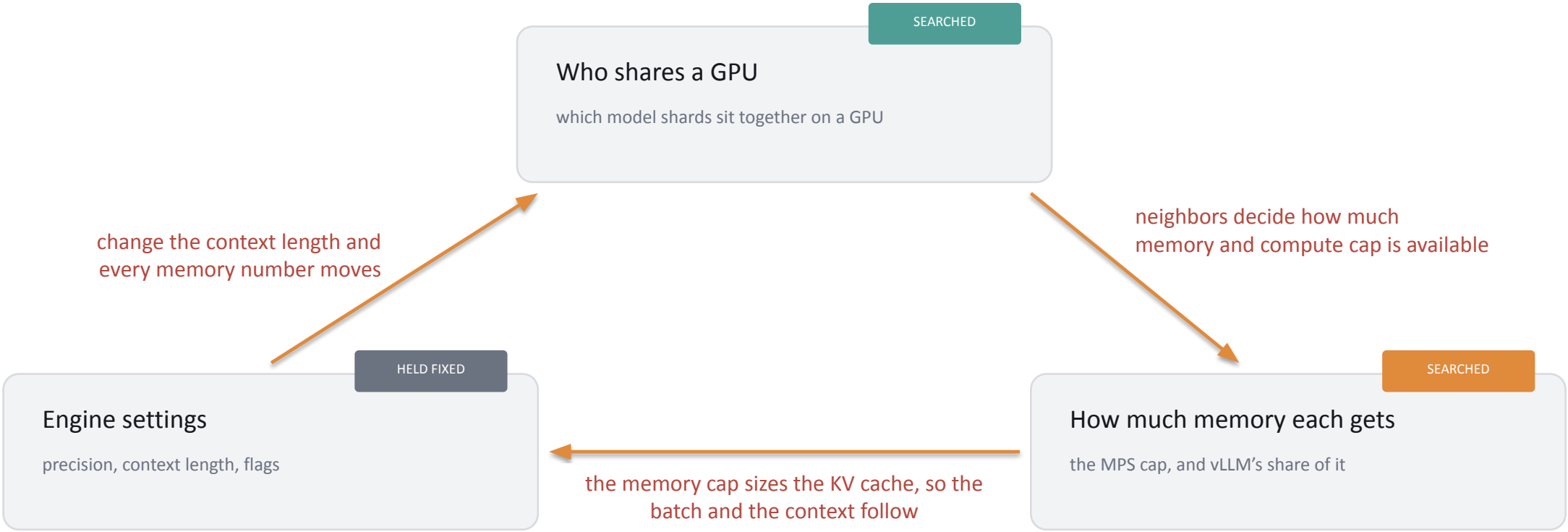
- A GPU split by memory, enforced by the driver
- Several containers on one GPU
- Deterministic placement, so a plan can be reproduced

WHAT IT DOES NOT

- Work out how much memory each service needs
- Work out which services can share a GPU
- Know that any service has a speed target at all

KAI starts where we finish. The puzzle we are trying to solve is its input.

What makes the problem hard: three-decision loop



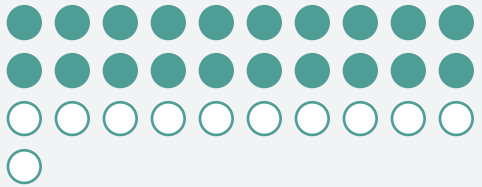
Move one and the other two move: One search loop, not a sequence of stages.

How ITL SLO affects KV Cache requirement

```
requests in flight  =  rate  ×  output tokens  ×  ITL target
```

IF THE SLO WERE TIGHTER

40 ms **between tokens (ITL)**



20 in flight, 31 at a burst

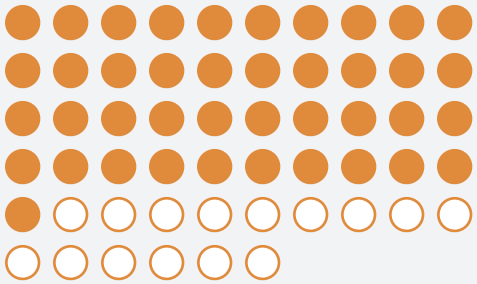
63,500 tokens of KV cache

fewer requests alive at once

→

g-reason's ACTUAL SLO

80 ms **between tokens (ITL)**



41 in flight, 56 at a burst

114,378 tokens of KV cache

twice as many, so twice the cache

A tighter ITL SLO needs LESS cache: As tokens output faster, tokens in flight in the KV-cache go down.

Why profiling model solo is not enough

g-reason (reasoning model) on the same four GPUs, measured twice. The only difference is the neighbors.

PROFILE: ALONE ON WHOLE GPUs



44 GiB each, nothing else resident

47.50 ms

ITL measured alone · ITL SLO 80 ms

SEARCH: SHARING WITH SEVEN SMALL TENANTS



34 GiB each, the rest to routers, guards, titlers

66.68 ms

+40%

ITL measured together · ITL SLO 80 ms

PERFORMANCE DEGRADATION INCREASES WITH THE CROWD

m-thumb, ITL SLO 20 ms

alone	+1 neighbor	+5 neighbors
3.0 ms	11.1 ms	40.2 ms

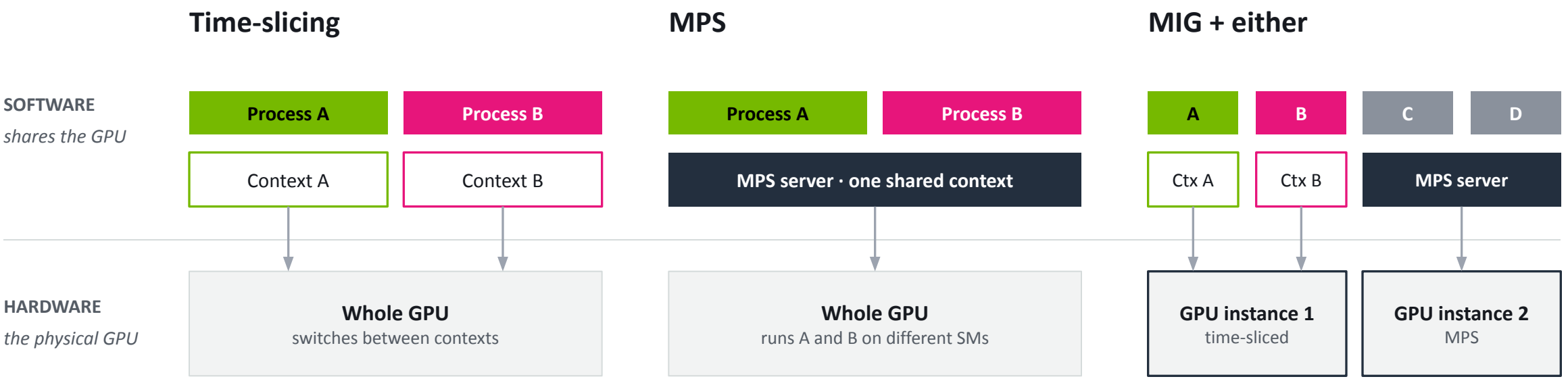
A solo model profile gives a floor for memory cap: Only measuring the combination of models gives real memory and compute cap.

PART 2

GPU sharing and why MPS

NVIDIA Multi-Process Service (MPS) is what lets several model servers share one physical GPU concurrently. Everything above depends on its memory ceiling behaving exactly as the following slides describe.

MIG splits the hardware; time-slicing or MPS shares it



Each process has its own context. The GPU switches between them, so kernels from different processes never overlap.

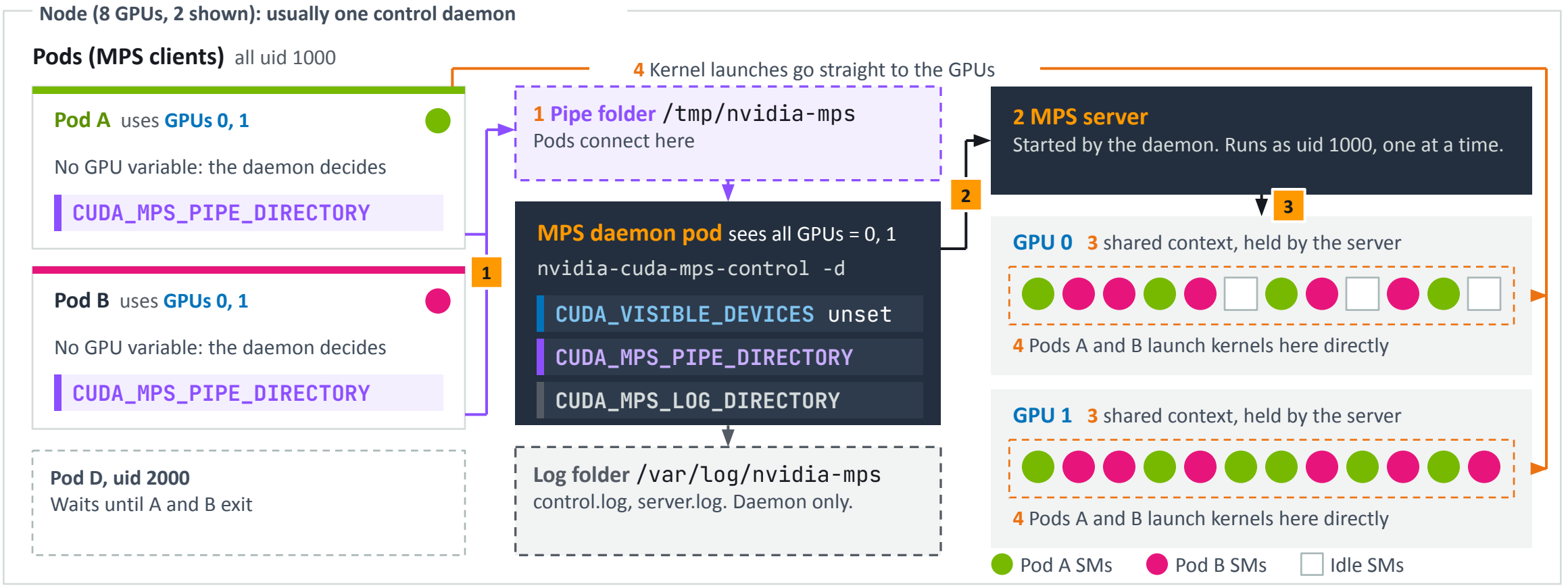
One MPS server holds a shared context. Its clients' kernels run at the same time on different SMs.

Each instance has its own SMs, L2, memory bandwidth and copy engines, and acts as its own GPU for time-slicing or an MPS server.

MIG decides which SMs and memory a workload gets. Time-slicing or MPS decides how processes share them.

Since Volta, MPS clients get separate GPU address spaces but share the server's scheduling resources. Without MIG, only one user's MPS server runs on a GPU at a time.

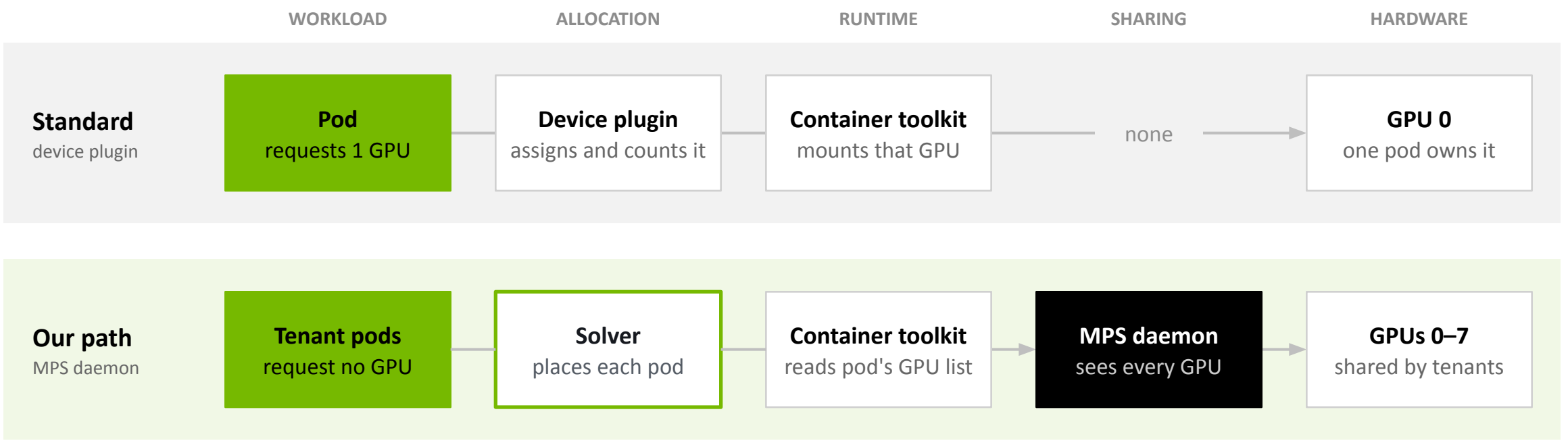
Pods connect to the MPS daemon over a shared pipe folder



With a host-run daemon, Kubernetes does not schedule the GPUs. The daemon takes the node's GPUs directly; pods only connect to it.
One server runs at a time, and it outlives its clients. A pod running as another user waits until all current clients exit.

Each process reads these variables at start. Paths are defaults; GPU numbers and dots are illustrative. The server inherits both folders from the daemon. Direct launches (step 4) need Volta or later GPUs.

Our solver places each pod, so we bypass the device plugin



Fully automated: the solver sets every placement and passes each pod its GPU list.

Caps per pod: the solver also sets each pod's GPU memory and SM limits through MPS[1]

1. Memory cap: CUDA_MPS_PINNED_DEVICE_MEM_LIMIT (per device, keyed by GPU UUID, e.g. GPU-a1f3…=20G). SM cap: CUDA_MPS_ACTIVE_THREAD_PERCENTAGE (share of SMs a client may use). Both are set in each pod's environment. Source: NVIDIA MPS documentation,

Use UUIDs. MPS renumbers GPU indices

HOST GPUS, AS NUMBERED BY NVIDIA-SMI

GPU 0
GPU-a1f3…

GPU 1
hidden by 0,2

GPU 2
GPU-9c07…

```
daemon: CUDA_VISIBLE_DEVICES=0,2
```

WHAT THE CLIENT SEES AFTER REMAPPING

Device 0
= host GPU 0

Device 1
= host GPU 2

Device 2
does not exist

Index 1 now means host GPU 2, and index 2 means nothing. The same number names different GPUs on each side.

INDEX KEY: 0,2

Fails in the client

Reusing 0,2 where the client already sees 0,1 leads to an error. Memory caps keyed by index inherit the same ambiguity.

UUID KEY: GPU-a1f3…

Same card everywhere

A UUID does not change when devices are remapped. A prefix starting with GPU- also works.

Key every device limit by UUID. List UUIDs with nvidia-smi -q and use them in CUDA_VISIBLE_DEVICES and CUDA_MPS_PINNED_DEVICE_MEM_LIMIT.

Source: NVIDIA MPS documentation, Appendix: Environment Variables (CUDA_VISIBLE_DEVICES; CUDA_MPS_PINNED_DEVICE_MEM_LIMIT), docs.nvidia.com/deploy/mps/latest

Size vLLM correctly

vLLM requests fraction × total memory (79.6 GiB) and needs that much free. Under the 20 GiB cap, only 20 GiB is free.

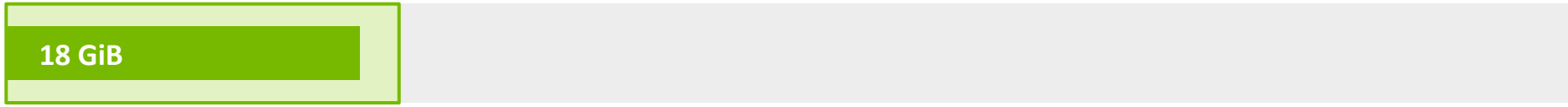
Default fraction
≈ 0.9 × 79.6 GiB

Requests 71.6 GiB, only 20 GiB free: startup fails



Adjusted fraction
18 ÷ 79.6 ≈ 0.2261

Requests 18 GiB, 20 GiB free: fits



Cap: 20 GiB free

Card total as reported: 79.6 GiB

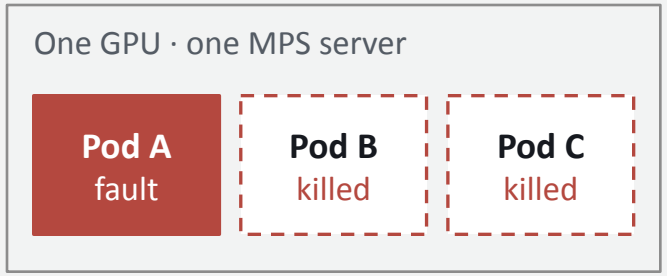
```
vllm serve <model> --gpu-memory-utilization #
# = envelope × headroom / physical_total
```

Fan-out computes this per tenant and passes it to vLLM.

MPS enforces the ceiling. The adjusted vLLM fraction keeps its planned budget within that ceiling. Configure both.

MPS trades isolation for density, so we benchmark

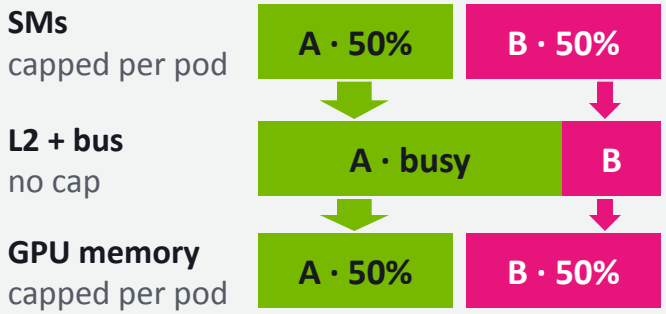
No fault isolation



One pod's fatal fault kills every pod on its MPS server. **If one is a TP shard, the whole model goes down.**

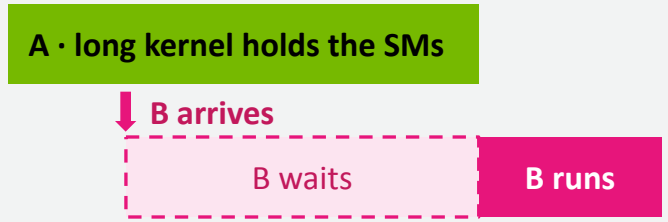
No performance isolation

illustrative



The thread cap limits SMs only. **Decode is bandwidth-bound, so a busy neighbour still slows you.**

No preemption



MPS can't pause A. **With no cap, urgent work waits until A's SMs free up.**

How we manage it: One v3 daemon can run several independent servers, each with its own pipe. The solver proposes a placement, the benchmark measures it, and we repeat.

PART 3

Agent driven pipeline

Claude Code runs the pipeline through the run-fleet skill

The agent-driven pipeline



plan	profile	plan	search	export	KAI render
IN fleet file ← The agent	IN fleet file ← The agent	IN solo rows ← measurement store	IN fleet, records ← measurement store	IN the best record ← measurement store	IN the export ← measurement store
OUT a report → The agent	OUT solo rows → measurement store	OUT a report → The agent	OUT records, export → measurement store	OUT certified result → measurement store	OUT Helm releases → k8s cluster

1 a measured TP degree can change what fits, so plan runs again 2 a group that missed is re-partitioned and searched again 3 a target nothing can meet within SLO: edit the fleet file and start over

All data passes through the measurement store. Agent checks the report and decides the next stage.

What you write: the fleet file

From three-teams.yaml: two of its 15 tenants.

```yaml
hardware:
  instance_type: g6e.48xlarge                                  # 1
  max_nodes: 2

tenants:
  - name: q-chat
    model: qwen3-14b
    rate_hz: 1.0                                               # 2
    isl_tokens: 1024
    osl_tokens: 256
    extra_args: ["--no-enable-prefix-caching"]                 # 3
    slo:                                                       # 4
      # 55, not 50: q-chat measured 49.72 ms solo …
      itl_mean_ms: 55.0
      ttft_p95_ms: 1200.0

  - name: m-docqa
    model: qwen35-9b
    rate_hz: 0.5                                               # 2
    isl_tokens: 8448
    osl_tokens: 512
    images:                                                    # 5
      count: 8
      pixels: 1048576

  # there is no tp: field; profile measures it                 # 6
```

1 Machine type and node count

Usable memory per GPU is read off a real node: 44.39 GiB on an L40S. max_nodes is 2 because these 15 need 19 GPUs alone.

2 Traffic as the team states it

Rate and request shape per tenant. If a target is out of reach, lower the rate rather than the SLO.

3 Prefix caching off

The load generator repeats prompts. With caching on we saw 96 to 99% hit rates, and prefill was not being measured.

4 SLOs come from the product

Changed once: q-chat ran 49.72 ms alone against 50, less room than run-to-run noise. The comment says why it is 55.

5 Images

m-docqa takes 8 images per request. Its 8,448 prompt tokens include the image tokens.

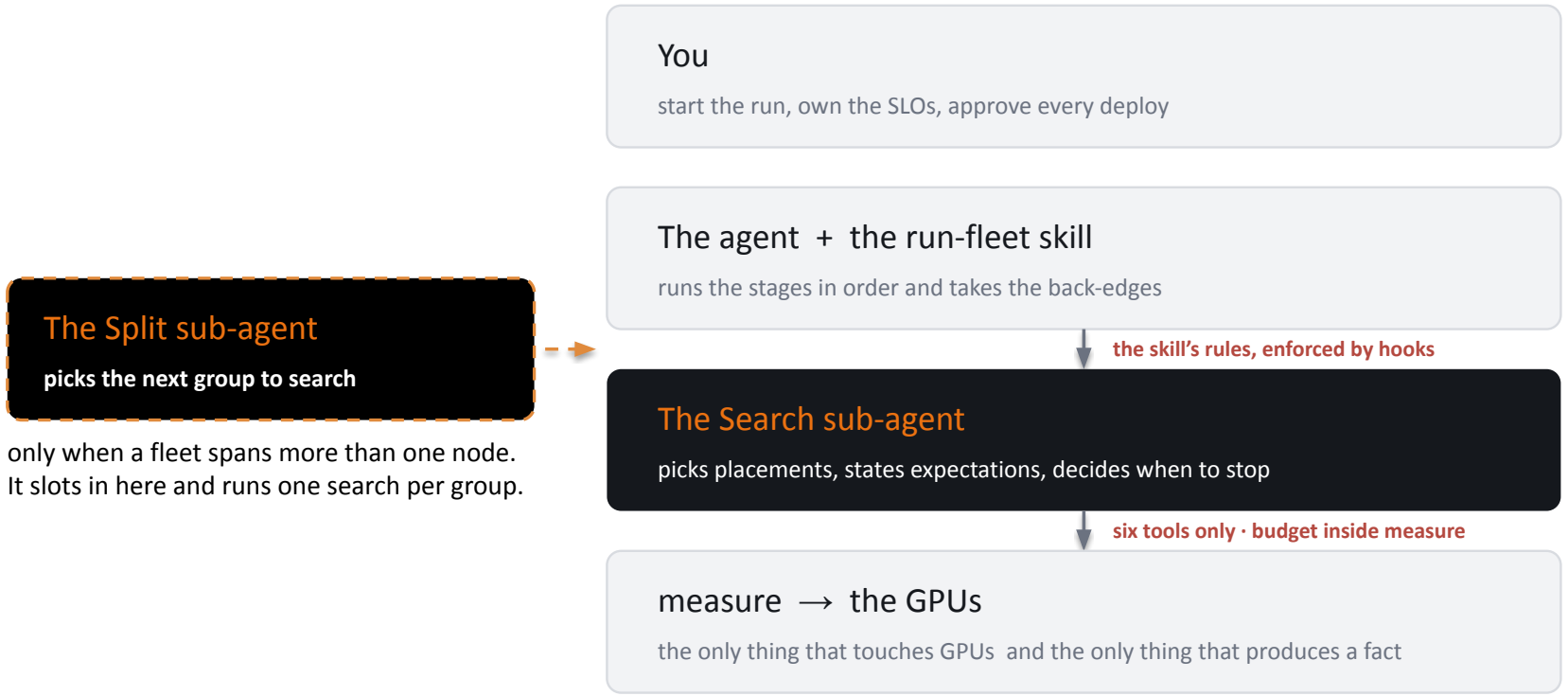
6 No TP field

Profile measures each model's smallest TP. The loader refuses a file that states one, so the baseline can't be padded.

An example model fleet file.

Search and Split sub-agents

A Split sub-agent is used only when a fleet is too big for one node.



only when a fleet spans more than one node.
It slots in here and runs one search per group.

The Main Agent

Cannot change an SLO, or deploy to KAI, without asking you

The Split sub-agent

Cannot declare a split is successful

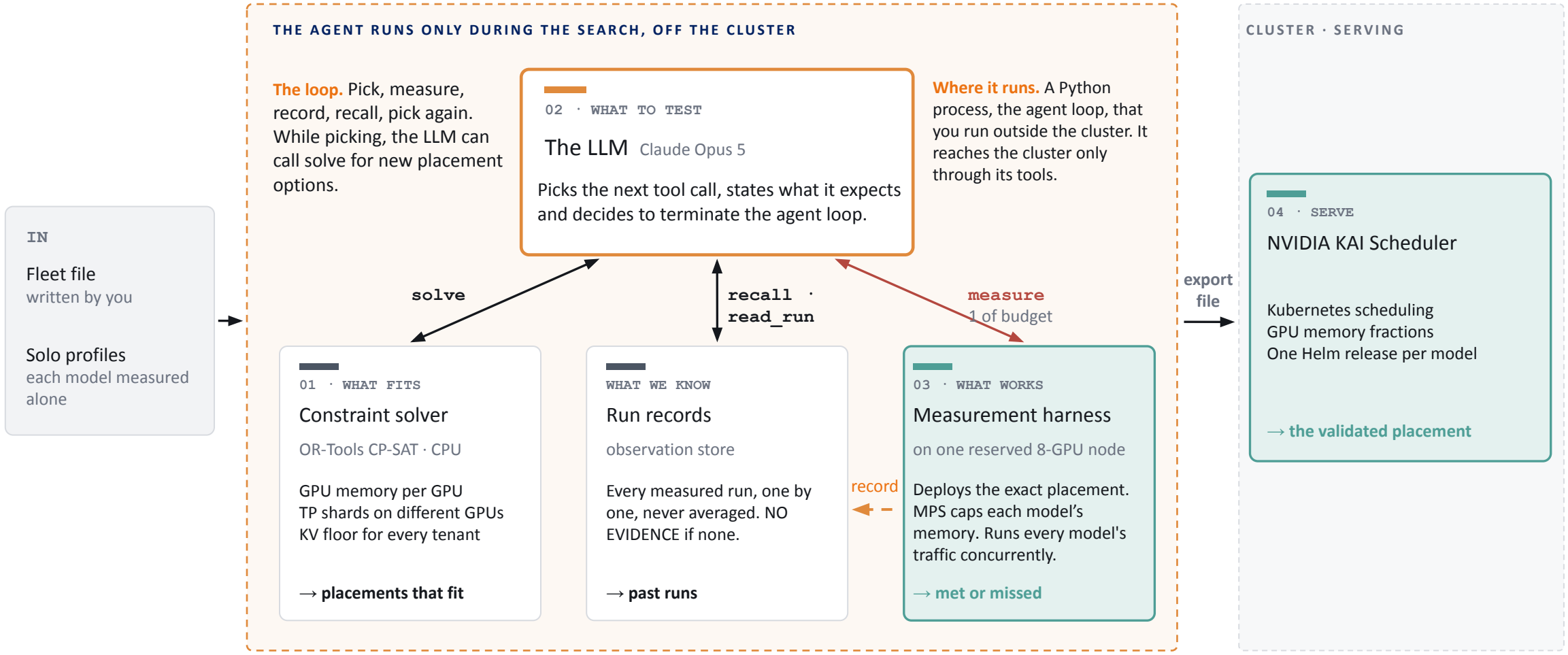
The Search sub-agent

Cannot certify a result, or touch the cluster except through measure

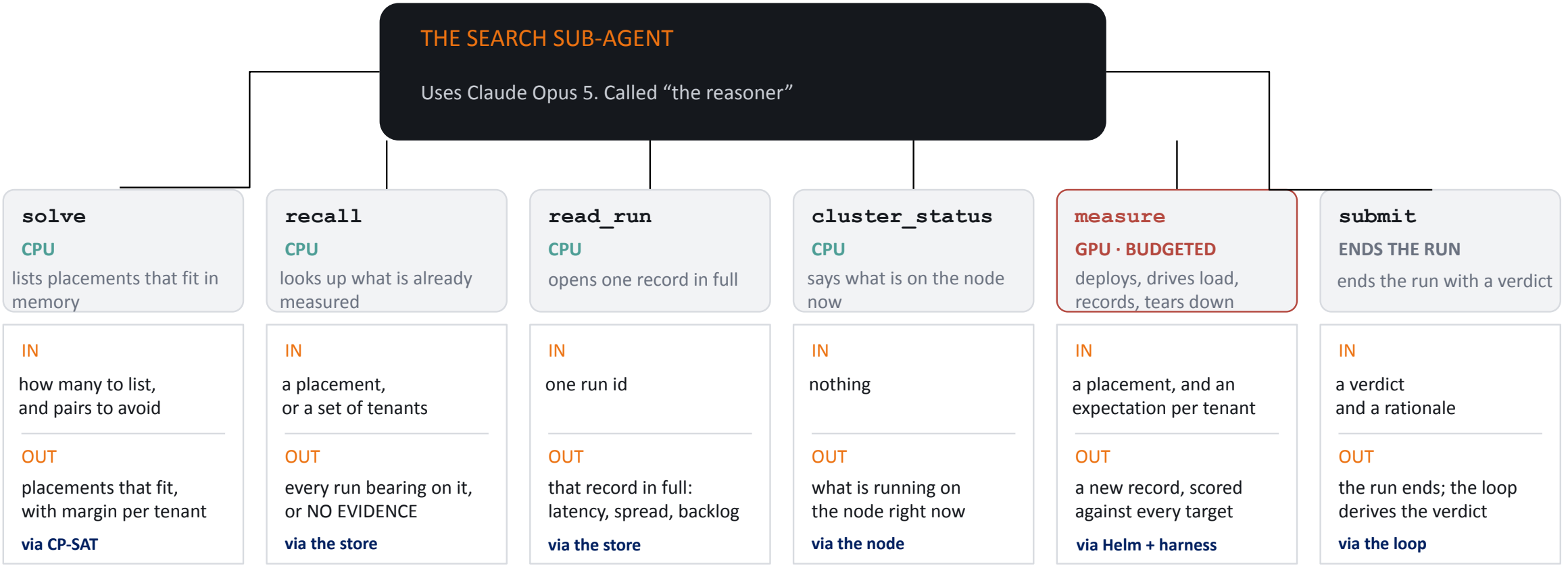
The sub-agents choose what to measure. The loop decides from the measured records whether it worked.

How the Search sub-agent works

Memory fit is the first filter. Whether a placement meets its SLOs is only known after the exact placement runs, with every model serving traffic at once.



The Search sub-agent and its tools



Five tools run on CPU. Only *measure* uses GPUs; each call counts against the budget.

Search sub-agent input

A fixed system prompt, one user turn: the problem, the evidence, the budget.

THE FIRST USER TURN · NODE B

```
== THE PROBLEM ==
  gpu node under test  any g6e.48xlarge
  GPUs gpu0 … gpu7  (44.39 GiB each, as the driver reports it
                  -- measured, not a nameplate)

  tenant   model                    tp   rate    in  out  target
  q-route  Qwen/Qwen3-0.6B           1  8.0Hz   512    8  itl 20, ttft 300
  g-reason zai-org/GLM-4.5-Air-FP8   4  0.5Hz  1024 1024  itl 80, ttft
4000
  thumb  Qwen/Qwen3.5-0.8B           1 12.0Hz   320    8  itl 20, ttft 250
  … (8 tenants)

  KV cache each tenant's own target requires, at its own rate:
    g-reason 114378 tokens  (0.5 Hz x 1024 osl x 80 ms = 40.96
                          mean + 2.33*sqrt(40.96) = 55.85 …)

== WHAT IS ALREADY MEASURED ==
  the store holds 2 record(s) that may be cited as evidence about
  latency. `recall` is how you see them …
== MEASURED RESOLUTION ==
  UNCALIBRATED. No run-to-run spread has been measured …
== BUDGET ==
  6 measurement(s).
```

THE SYSTEM PROMPT · FOUR FIXED PARTS

Preamble

The memory arithmetic says what a model needs. The tightest placement can be the slowest.

Transport

The agent can only call its six tools. It gets no shell, no files, and no other route to the cluster.

Tool catalogue

What each tool takes, and what it costs. Only measure spends GPU time.

Rules

Give an expectation for every model. Do not change the fleet, the targets or the traffic. Only the pipeline decides that a placement met its targets.

The Search sub-agent gets the model fleet, and measurement budget. It reads past runs only by calling *recall*.

Fleets that span several nodes

If a fleet doesn't fit on one node, it is split into groups by the Split sub agent, one node per group. The Search sub-agent picks which group to search next, and each group is then searched.

SPLIT AGENT

`partition` CPU `recall` CPU `search_group` WHOLE SEARCH `submit` ENDS

GROUP A · node 1

search agent

six tools · its own budget

its own fleet file, reserved node, budget, records and export

GROUP B · node 2

search agent

six tools · its own budget

its own fleet file, reserved node, budget, records and export

Groups are searched one at a time, and each releases its node before the next one starts.

LIMITS AT BOTH LEVELS

Turn cap: stops unnecessary tool calls

Fallback: if the agent is unavailable, use the ranked order and log it

Verdict: met only if every group in a split met its SLOs

THE REPORT SHOWS WHO PICKED EACH GROUP

reasoner | fallback | drive | store

store = reused from an earlier run with an identical group file

Out of scope: running one model across nodes. No pipeline parallelism or cross-node collectives.

Why a split sub-agent: we only learn which tenants interfere by running them together.

Search sub-agent trajectory

Node B, the second of the fleet's two nodes: eight tenants, a budget of six measurements, 33 turns.

1

```
▸ solve {"cap": 4}
  4 candidate(s) on 4 GPUs · tightest margin 1.078× (g-reason)
```

Four placements fit in memory, all on four GPUs. The solver reports memory margins and no latency.

2

```
▸ recall {"candidate": 1}
  search-rzyvij-001 · g-reason 66.40 ms ≤ 80 · met
```

One record, from an interrupted run. It passed, but g-reason varied ±7.77 ms within that run against 13.6 ms of headroom.

3

```
▸ measure {"choice": {"candidate": 1}, "label": "cand1-repeat"}
  run …-000 · all 8 met · g-reason 66.68 ms ≤ 80
```

Run-to-run spread was unmeasured, so it repeats the same placement. The result is within 0.3 ms of the earlier run.

4

```
▸ measure {"choice": {"assignment": {g-reason: gpu0–3, 7 small: gpu4}}, "expectation": {…}}
  run …-001 · g-reason 47.53 ms · m-tag 42.8 > 30 ✗ · m-thumb 40.2 > 20 ✗ · q-route 20.2 > 20 ✗
```

Its own placement: g-reason alone on four GPUs, the small models on a fifth. g-reason gains 19 ms and three small models miss.

5

```
▸ measure {"choice": {"candidate": 1}, "sm_caps": {q-route: 25, m-thumb: 35, …}}   × 3
  runs …-002, …-003, …-004 · g-reason 58.47, 56.86, 56.86 ms · all 8 met
```

Back to four GPUs, with SM caps on the small models. g-reason runs at its slowest shard, and on gpu0 it had only 35% of the SMs.

6

```
▸ submit {"verdict": "met", "choice": {"candidate": 1}, "sm_caps": {…}}
  accepted · 5 of 6 measurements used · 4 GPUs, 7 freed
```

Settled: four GPUs is the minimum, and isolating g-reason is worse. Not settled: candidates 2, 3 and 4 were never measured.

solve is deterministic: the same call returns the same list. A placement already measured is labelled ALREADY MEASURED.

Why the search sub-agent bets before it measures

A pass or a failure is a number. The bet made beforehand is what tells the agent whether to change the placement.

	MEASURED MET	MEASURED MISSED
IT EXPECTED HOLD	**Its assumption holds** Push. Try a denser placement. turn 3 · g-reason expected hold, met at 66.68 ms	**Its assumption was wrong** Stop and diagnose before trying another placement. Preemptions point at memory; none point at compute. turn 4 · q-route expected hold, measured 20.2 against 20
IT EXPECTED MISS	**It was too cautious** A pairing it expected to fail works. Ruling that out on a guess would have cost a GPU.	**Confirmed** Rule the pairing out for the rest of this run, citing the run id.

UNKNOWN Not scored. The agent says it when it has no evidence to go on - turn 4, m-tag and m-thumb at seven models on one GPU.

measure runs with a bet on every tenant: The bet helps the search sub-agent build a working hypothesis

One Helm release per tenant, on top of MPS

The placement becomes chart values. The chart writes each pod's MPS limits into its environment.

CHART VALUES, FROM THE PLACEMENT

```
# q-route — one shard on gpu0, beside g-reason
placement: {gpu_indices: [0], gpu_uuids: [GPU-adba0a16-…]}
mps:       {memory_limit_gib: 5, active_thread_percentage: 25}
vllm:      {tensor_parallel_size: 1, gpu_memory_utilization: 0.0931}

# g-reason — four shards, one per GPU, uncapped
placement: {gpu_indices: [0, 1, 2, 3], gpu_uuids: [… × 4]}
mps:       {memory_limit_gib: 34, active_thread_percentage: null}
vllm:      {tensor_parallel_size: 4, gpu_memory_utilization: 0.7123}
```

WHAT LANDS IN THE POD'S ENVIRONMENT

```
q-route's engine container:
  CUDA_VISIBLE_DEVICES=0
  CUDA_MPS_PIPE_DIRECTORY=/mps
  CUDA_MPS_PINNED_DEVICE_MEM_LIMIT=GPU-adba0a16-…=5G
  CUDA_MPS_ACTIVE_THREAD_PERCENTAGE=25

g-reason's engine container:
  CUDA_VISIBLE_DEVICES=0,1,2,3
  CUDA_MPS_PINNED_DEVICE_MEM_LIMIT=…=34G  × 4 UUIDs
  (no ACTIVE_THREAD_PERCENTAGE: uncapped)
```

PINNED_DEVICE_MEM_LIMIT
the memory envelope, keyed by GPU UUID

ACTIVE_THREAD_PERCENTAGE
the SM cap, when the agent asked for one

an init container
reads free memory back before the engine starts. A pod that fails never serves.

`gpu_memory_utilization = 0.95 × (envelope − measured reserve) ÷ GPU total` g-reason: `0.95 × (34 − 0.7178) ÷ 44.39 = 0.7123`

One MPS daemon per node, holding every GPU. Each Pod carries its own limits.

The certified export to KAI

Written by search itself when it ends, with every machine identifier removed.

```
strict:
  certified: true
  devices: 4   devices_freed: 7   scored_against_devices: 11
  run_id: search-r26ljz-000-cand1-repeat
  slots:                      # one GPU each, by position
    - residents: [g-reason/0, g-title, q-route]
    - residents: [g-reason/1, q-guard, q-title]
    - residents: [g-reason/2, m-tag]
    - residents: [g-reason/3, g-guard, m-thumb]
  per_tenant:
    g-reason: {shards: 4, required_gib: 33.591, granted_gib: 34.0,
        itl_mean_ms: 66.68, target_itl_ms: 80.0, ttft_p95_ms: 656.66,
        solo_itl_ms: 47.50, ratio_vs_solo: 1.40, targets_met: true}
    q-route:  {shards: 1, required_gib: 3.102, granted_gib: 5.0,
        itl_mean_ms: 5.43, target_itl_ms: 20.0, solo_itl_ms: 2.39,
        targets_met: true}
```

Slots, not whole GPUs

A slot is one GPU by position. No UUIDs, no ordinals: the machine gets replaced, and anything naming a GPU would expire with it.

Every tenant carries its proof

What it needed, what it was granted, what it measured, its target, what it managed alone, and the ratio between them. g-reason: 1.40×.

certified is derived, not claimed

The loop sets it from a run of this exact configuration in which every tenant held every target. The agent cannot write it.

ACROSS BOTH NODES

```
split: {verdict: met, nodes: 2, devices: 9, baseline_devices: 19}
```

Deploying exported configuration using KAI is a separate step

PART 4

Deploying using KAI scheduler

The exported certified config is deployed using KAI scheduler

PART 4 · OVERVIEW

Deploying the certified export in KAI

The search is over. A person starts the deployment, and the export is the only input it takes.

WHAT COMES IN

The certified export: slots, residents, and the memory each model was granted.

WHAT WE RENDER

One Helm release per model, with a GPU memory fraction and an MPS mode.

WHAT KAI DOES

Picks the physical GPUs, splits each one by memory, and lets the driver enforce the split.

THE NEXT FOUR SLIDES

1	Ask KAI for part of a GPU	The memory budget becomes a request and a limit. The compute mode turns on MPS sharing.
2	Turn the export into Helm values	Each slot resident becomes one release: its fraction, its GPU count, and a recomputed engine budget.
3	Cap the SM share	MPS can cap compute as well as memory. KAI has no setting for it, so we pass the variable through.
4	Check what KAI actually did	The groups that formed, the limits that took effect, and the latency under concurrent load.

Export defines KAI configuration: Every number in these files was measured during the search.

How you ask KAI for part of a GPU

The memory budget becomes a request and a limit. The compute mode turns on MPS sharing.

THE WORKLOAD SETTINGS

1 **Request GPU memory**

KAI counts this against the GPU's capacity.

2 **Set the memory limit**

GPU fractioning sets the boundary.
The NVIDIA driver enforces it.

3 **Choose SM sharing**

MPS lets the models compute at the same time.
We ask for no fixed share of the SMs.

POD EXCERPT · ILLUSTRATIVE 20 GiB

```
metadata:
  annotations:
    nvidia.com/container.vllm.gpu-memory.request: "20Gi"
    nvidia.com/container.vllm.gpu-memory.limit: "20Gi"
    nvidia.com/container.vllm.gpu-compute.mode: "sm-sharing"
spec:
  schedulerName: kai-scheduler
  containers:
    - name: vllm
```

Memory is not compute: A GPU-memory fraction reserves no matching share of compute.

From certified export to Helm values

One release per model. Every value comes from the export or is recomputed from it.

ONE TENANT OF NODE B · TRIMMED

```
# helm upgrade --install kai-three-teams-n2-g-guard
tenant:
  name: kai-three-teams-n2-g-guard
  podLabels:
    colocation.kai/deployment: three-teams-n2
    colocation.kai/tenant:     g-guard
    kai.scheduler/queue:       colocation-serving

  memory:  6108Mi     # the fraction KAI reserves
  devices: 1          # GPUs, the TP degree

  nodeSelector:
    node.kubernetes.io/instance-type: g6e.48xlarge

  image: nvcr.io/nvidia/vllm:26.07-py3
  args: [--model=.../Qwen3-0.6B,
         --tensor-parallel-size=1,
         --gpu-memory-utilization=0.826568,
         --max-num-seqs=64, --no-enable-prefix-caching]
```

memory comes from the export

g-guard was granted 6108 MiB during the search. KAI reserves exactly that much of one GPU.

The engine budget is recomputed

vLLM sizes itself from the whole GPU, so its fraction is the granted memory divided by the GPU total.

Every GPU is filled exactly

Fractions are rendered so one legal placement remains: the measured one. Across node B, 0 MiB left unused on every GPU.

The search sub-agent exports these placements: KAI deploys them.

Capping the SM share under KAI

MPS can cap compute as well as memory. KAI has no setting for it, so the variable goes in the pod.

POD EXCERPT · A 73% SM CAP

```
containers:
  - name: vllm
    env:
      - name:  CUDA_MPS_ACTIVE_THREAD_PERCENTAGE
        value: "73"
      - name:  CUDA_MPS_PINNED_DEVICE_MEM_LIMIT
        value: "GPU-<uuid>=18816M"

# KAI injects the MPS client environment;
# these variables ride along with it.
```

A cap is a ceiling, not a reservation

It hands no SMs to anybody. It stops one model from taking the SMs its neighborr needs. The agent tunes it per model, one measurement at a time, exactly as it tunes memory.

WHAT KAI DOES AND DOES NOT OFFER

Memory fraction — KAI SETTING

gpu-memory.request and .limit, enforced by the driver.

MPS sharing on or off — KAI SETTING

gpu-compute.mode: sm-sharing turns MPS on for the GPU.

Share of the SMs — NO KAI SETTING

Set as a pod environment variable and passed through KAI's MPS injection.

MEASURED ON KAI · EXPERIMENTAL

Remove only this variable and Qwen3.5-9B goes from 15.4 to 18.3 ms, over its ceiling. It takes effect, but KAI does not own it.

Memory is a KAI feature; compute is not. The cap works today as an MPS pass-through, so we treat it as experimental.

Why you check what KAI did

The plan names logical groups. KAI picks the physical devices. The adapter then checks that the groups came out the way the plan asked.

THE ADAPTER GUIDES AND VERIFIES

Sizing and submission order steer the packing. Only the assignments KAI returns say whether the grouping matched.

1. **Size each GPU request**
   Translate each memory budget, add the runtime's own reserve, and pad where padding steers the packing.
2. **Submit models in sequence**
   Wait for each GPU assignment before submitting the next. An assignment is not a ready endpoint.
3. **Read back the devices**
   Compare the groups that actually formed against the export, and report any mismatch for retry or review.

Illustrative target grouping

Logical slot 1: Model A + Model B
Logical slot 2: Model C

These are planned groups, not physical GPU identifiers.

KAI selects the physical GPUs

Fractional requests alone do not guarantee the pairing you asked for, so the adapter checks it.

We check that the placement reproduced. Even a matching allocation still has to pass a concurrent performance test.

Three checks after deployment

A passing search result is where this starts, not where it ends. Check placement, runtime controls and performance again once it is deployed.

Placement

allocation check

Read GPU assignments and memory requests. Confirm the actual model groups match the export.

Runtime

serving check

Confirm the endpoints are ready, and that the memory boundaries and MPS sharing really took effect.

Performance

concurrent load test

Deliver the stated traffic to all models together. Compare mean ITL and p95 TTFT with their targets.

Runtime changes need revalidation

The serving path uses vLLM endpoints. If the runtime differs from the one the search ran on, Dynamo for instance, test again.

Memory that fits and endpoints that answer still prove nothing about latency.

Drift tracking records traffic and latency changes. A person decides when to search and deploy again.

PART 5

Experiment results: Running example

This Is the running example model fleet on two L40S nodes.

Plan before solo model profile

plan is offline arithmetic: no reasoner, no tools, no GPU. It runs before profile so that nothing is leased until the fleet is known to be worth measuring.

SERVICE	KV CACHE NEEDED	SMALLEST TP	MEMORY PER GPU
q-reason	163,174 tokens	2	39.77 GiB
g-reason	114,378 tokens	4	33.59 GiB
m-docqa	219,318 tokens	1	27.86 GiB
q-chat	29,196 tokens	1	35.70 GiB
m-caption	57,112 tokens	1	13.60 GiB
the 10 others	1,688 – 28,701	1	3.10 – 6.10 GiB

2 + 4 + 13 = 19 GPUs = 3 machines. We are allowed 2.

Where the numbers come from: weights from the checkpoint on the Hub, KV geometry from the model's own config.json, the engine reserve measured in profile. Activation starts at an arbitrary 1.0 GiB, set above every value measured so far, so it can only over-reserve.

Why plan runs first

It answers three questions for free, in seconds. Does the fleet load and fit? Is sharing worth searching — here 90 of the 105 possible pairs fit on one GPU together. And which TP degrees are still only memory floors?

Those floors are exactly what profile has to measure.

What it cannot tell you

Memory says which arrangements are possible. It says nothing about speed. The baseline here is a lower bound built on floors, and even it already needs three machines.

Fifteen KV Cache figures derived by the solver: avoids wasteful profiling for infeasible configs on GPUs

Profile before search

The search sub-agent cannot start without this. profile settles each model's smallest GPU count, measures the engine reserve that sizes vLLM inside its envelope, and fixes the baseline the result is scored against.

SERVICE	ALONE	ITS SLO	HEADROOM	
m-docqa	56.20 ms	60 ms	6%	on file
q-chat	49.72 ms	55 ms	9.6%	measured new
g-reason	47.50 ms	80 ms	41%	measured new
q-reason	28.74 ms	60 ms	52%	measured new
m-caption	27.40 ms	60 ms	54%	on file
the 10 others	2.4 – 8.7 ms	20 – 150 ms	3–30× inside	on file

Twelve of fifteen cost nothing

These teams had each run the pipeline before. A service measured alone depends only on itself and the machine, so those rows were still good. Only three were new.

Two run close to their targets

m-docqa sits 6% under its SLO with the machine to itself. q-chat, 9.6%. Everything else runs 3 to 30 times inside.

They tell you where sharing will hurt first, before a single shared measurement is taken.

By profiling measurement, not solver output: Every model meets its SLO solo at its smallest GPU count.

Fifteen models on nine GPUs

Each chip: time per token alone → sharing, against its SLO. All fifteen fit one node in memory; latency needed two.

Node A

5 GPUs

gpu0 — 44.1 GiB	gpu1 — 44.1 GiB	gpu2 — 44.1 GiB	gpu3 — 44.1 GiB	gpu4 — 44.1 GiB
g-followup 13.8 GiB 2.8 → 13.2 ms · SLO 50	q-reason 39.9 GiB 28.7 → 42.5 ms · SLO 60	q-reason 39.9 GiB 28.7 → 42.5 ms · SLO 60	m-docqa 44.1 GiB 56.2 → 58.0 ms · SLO 60	q-chat 44.1 GiB 49.7 → 50.3 ms · SLO 55
m-caption 30.4 GiB 27.4 → 44.0 ms · SLO 60	g-route 4.2 GiB 2.4 → 5.4 ms · SLO 20	m-alt 4.2 GiB 3.9 → 9.3 ms · SLO 25		

Node B

4 GPUs

gpu0 — 44.1 GiB	gpu1 — 44.1 GiB	gpu2 — 44.1 GiB	gpu3 — 44.1 GiB
g-reason 33.7 GiB 47.5 → 66.7 ms · SLO 80	g-reason 33.7 GiB 47.5 → 66.7 ms · SLO 80	g-reason 33.7 GiB 47.5 → 66.7 ms · SLO 80	g-reason 33.7 GiB 47.5 → 66.7 ms · SLO 80
g-title 6.3 GiB met · SLO 150	q-guard 5.1 GiB met · SLO 80	m-tag 10.4 GiB met · SLO 30	g-guard 6.0 GiB met · SLO 80
q-route 4.2 GiB 2.4 → 5.4 ms · SLO 20	q-title 5.3 GiB met · SLO 150		m-thumb 4.5 GiB 2.8 → 11.1 ms · SLO 20

19 GPUs on 3 nodes becomes 9 on 2. Every model inside its SLO, measured twice on each node. Every GPU filled to the last megabyte.

Nine of them, by selective routing

19

GPUs

if every service had GPUs of its own

3 machines. More than allowed

9

GPUs

measured, every SLO met

2 machines: 5 + 4

53%

fewer

10 GPUs and one whole machine freed

for 15 services across 3 teams

Arithmetic picks candidates, measurement picks the answer

All 15 fit one machine’s memory. Sixteen measurements said no.

Results accumulate

12 of 15 solo profiles, and whole machine results from earlier runs, were reused at no cost.

The agent decides, the system verifies

It chose what to measure and explained why. It could not declare success, the verdict is computed from the measurements.

One service met with almost no room

m-docqa finished at 58–60 ms against 60. Valid, and the first thing a rollout should watch.

PART 5

Experiment results: Two new fleets

That was the running example on two L40S nodes. Now two new model fleets on one H100 node, starting from an empty measurement store, where every number is measured from a cold start.

1 Testbeds

Two new model fleets that do not fit by whole GPUs in one H100 node.

2 Results

Both fleets on 8 H100 GPUs, and how the agent got there

3 Serving on KAI

What carries over from the search

4 Performance analysis

What sharing costs, SM caps, MPS vs time-slicing

Two platforms, and fleets that do not fit by whole GPUs

H100

p5.48xlarge: 8× H100 80 GB, NVSwitch

EKS · KAI Scheduler v0.18 · vLLM 0.24

Search: our own MPS daemon

Serving: KAI + GPU fractioning, MPS SM sharing

Text-only fleet (H100)

Model	TP	req/s	in → out	ITL ≤
Llama-3.3-70B	4	0.12	8192 → 1024	41 ms
gemma-3-27b	2	0.6	4096 → 512	30 ms
phi-4	1	4	1024 → 64	22 ms
gemma-3-12b	1	0.8	2048 → 256	23 ms
Qwen3-8B	1	2	2048 → 256	27 ms
Qwen3-4B	1	8	512 → 16	16 ms

DISJOINT: 10 GPU SLOTS, THE NODE HAS 8



Memory is not what runs out: the GPUs do.

Multimodal fleet (H100)

Model	TP	req/s	in → out	ITL ≤
Qwen3.5-122B MoE	4	0.25	16 img → 512	28 ms
Qwen3.5-27B	2	0.6	12 img → 384	85 ms
Qwen3.5-9B	1	2	2 img → 128	16 ms
Qwen3.5-4B	1	8	1 img → 64	20 ms
phi-4	1	4	1024 → 64	22 ms
Qwen3-8B	1	2	2048 → 256	20 ms
Qwen3-4B	1	8	512 → 16	8 ms

DISJOINT: 11 GPU SLOTS, THE NODE HAS 8



A 16-image prompt is 16,640 tokens of KV per request.

Every number is stated per tenant. Rate, shape and the ITL / TTFT ceilings come from the fleet file; nothing is defaulted. Prefix caching is off for every tenant.

PART 5 · RESULTS

Both H100 fleets fit on 8 GPUs

TEXT-ONLY FLEET

10 → 8

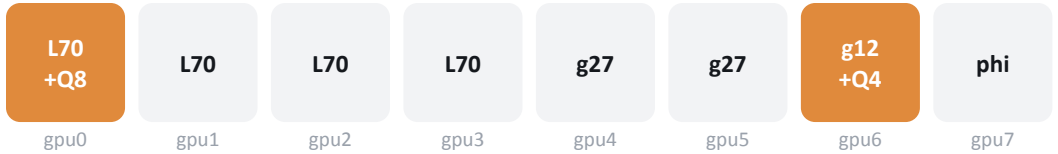
2 GPUs freed (20%) · no SM cap · deploy on KAI

MULTIMODAL FLEET

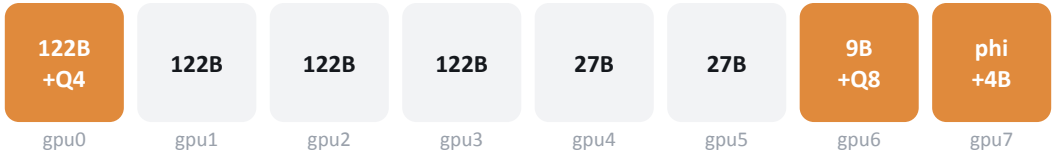
11 → 8

3 GPUs freed (27%) · explicit SM caps · deploy on KAI

THE AGENT'S ANSWER — TEXT-ONLY



MULTIMODAL — SM CAPS Q8 73%, PHI-4 55%



L70 Llama-3.3-70B · g27 gemma-3-27b · g12 gemma-3-12b · 122B Qwen3.5-122B-A10B · 27B / 9B / 4B Qwen3.5 · Q8 / Q4 Qwen3-8B / 4B · orange = shared GPU

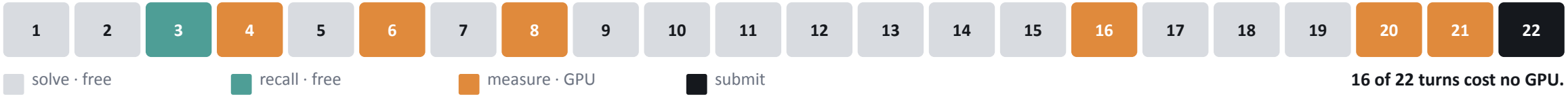
Text-only · 5 cold runs	L70	g27	g12	phi	Q8	Q4
ITL, ms	33.5	16.6	17.2	13.6	13.6	9.9
ITL, % of ceiling	82%	55%	75%	62%	50%	62%
TTFT p95, % of ceiling	33%	11%	13%	14%	16%	27%

Multimodal · 7 cold runs	122B	27B	9B	Q8	phi	4B	Q4
ITL, ms	24.4	64.7	14.8	19.0	20.9	18.9	6.5
ITL, % of ceiling	87%	76%	92%	95%	95%	95%	82%
TTFT p95, % of ceiling	64%	66%	9%	16%	20%	18%	25%

From an empty store, measured cold. Every model inside every ceiling in every cold redeployment. 7 GPUs was ruled out by separate measurements.

Text-only search: think for free, measure to decide

Text-only fleet, from an empty store. Profile ran 8 solo measurements; then the agent took 22 turns with a budget of 6.



16 of 22 turns cost no GPU.

solve × 14

What fits in memory?

Turn 5: can phi-4 have a GPU alone if Qwen3-8B moves onto a Llama GPU? It fits, so turn 6 measures it. Turns 9–19: ten more shapes, nine of them 7-GPU ideas; only two were worth a measurement.

recall × 1

What do we already know?

Turn 3: nothing, the store is empty. So the first measurement is a loose 8-GPU baseline, not a guess at 7.

measure × 6

One question per measurement.

Is phi-4's miss its neighbour's fault (turn 6: yes)? Does a 7-GPU shape hold (8, 16: no)? Is the answer repeatable (21: yes)? It predicts every model first: 26 of 27 'will meet' right.

submit × 1

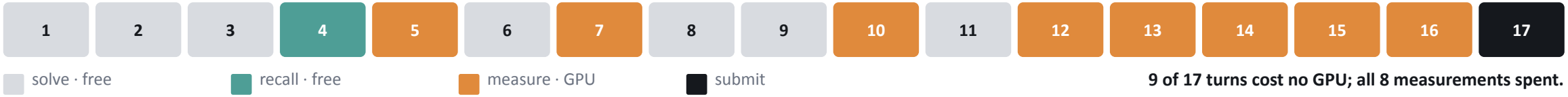
When to stop, and what to claim.

Turn 22, budget spent: 8 GPUs. Recommends the placement met twice, declines one with 2.4% margin, leaves 7 open.

The solver answers what fits; the agent decides what to ask next. Most ideas are ruled out for free, before any GPU time is spent.

Multimodal search: find the pairing, then tune the cap

Three mid-size models and two free GPUs: the question is who shares. 17 turns, a budget of 8 measurements.



9 of 17 turns cost no GPU; all 8 measurements spent.

Turns 1–4 · map it

What fits, and what is known?

The 122B takes four GPUs, the 27B two. Two 4B models can ride the 122B's GPUs; Qwen3.5-9B, Qwen3-8B and phi-4 cannot. The store is empty.

Turns 5–10 · pairing

Who shares with whom?

Baseline: the 27B cannot take a neighbour. phi-4 + Qwen3-8B: phi-4 misses. Qwen3.5-9B + Qwen3-8B: only the 9B misses, by 17%. The closest.

Turn 11 · last idea

Can Qwen3-8B ride the 122B?

No: it does not fit beside a 122B shard. Every structure is now tried, so the pairing stays and something else has to give.

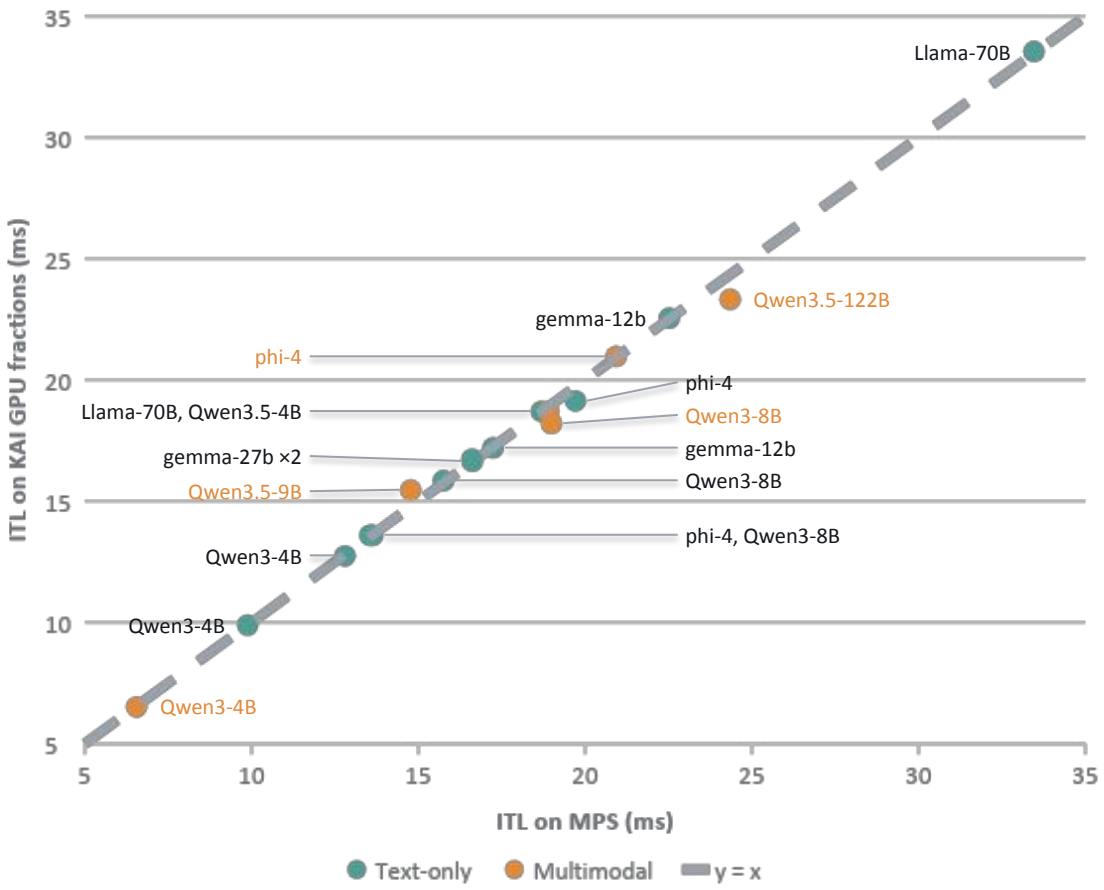
Turns 12–16 · the cap

How much of the GPU may Qwen3-8B use?

60% → 75% → 70% → 73%, then 73% again. All met from 70% up (slide on SM caps).

Turn 17 · submit: 8 GPUs, Qwen3-8B at 73%. 8 GPUs was already hard to meet and took the whole budget; the agent left 7 untried, and says so.

Search on MPS, serve on KAI: the latency carries over



On the supported path: the text-only fleet

10 → 8 GPUs with no SM cap.

Served on KAI with sm-sharing, all six models within 1% of their time per token under our own MPS, every ceiling met.

Kept separate: explicit SM caps

MPS can also cap one process's share of the SMs. Without a cap, the multimodal fleet does not fit one 8-GPU node: every forced pairing misses. With one, it fits on 8.

The current KAI Scheduler does not expose a per-workload SM setting: treat those results as MPS-layer experiments.

Deployable today: placement, memory fractions and MPS SM sharing through KAI. Static per-workload SM allocation is an MPS capability, not a KAI setting today.

PART 5 · MPS SM Cap

The agent tunes MPS SM caps: 'no placement' becomes 8 GPUs

When two busy models must share a GPU, time per token breaks. The agent tunes each model's SM cap, the way it sets memory.

MULTIMODAL FLEET: QWEN3.5-9B AND QWEN3-8B SHARE ONE GPU · THE AGENT TUNES QWEN3-8B'S CAP, ONE MEASUREMENT PER STEP

No cap	60%	75%	70%	73%	73% again
Qwen3.5-9B misses	Qwen3-8B misses	Met, too tight	Met, too tight	Met, both clear	Met again
17% over its ceiling	7% over its ceiling	9B only 2.6% below	Q8 only 0.7% below	9B 8% · Q8 5% below	9B 9% · Q8 5% below

Below = margin under the ceiling (Qwen3.5-9B 16 ms, Qwen3-8B 20 ms). The rest of the fleet met in every step. Result: 11 GPUs → 8.

Why these two share a GPU

On 8 GPUs the 122B and the 27B take six. Qwen3.5-9B, Qwen3-8B and phi-4 get the other two, so two must share. Uncapped, every pairing misses.

Why a cap helps

Uncapped, Qwen3-8B takes the SMs Qwen3.5-9B needs. A cap is a ceiling, not a reservation: it moves slack to the model that needs it.

On KAI: a pass-through

Set as a pod env var, A/B on KAI: without it, Qwen3.5 9B 15.4 → 18.3 ms.

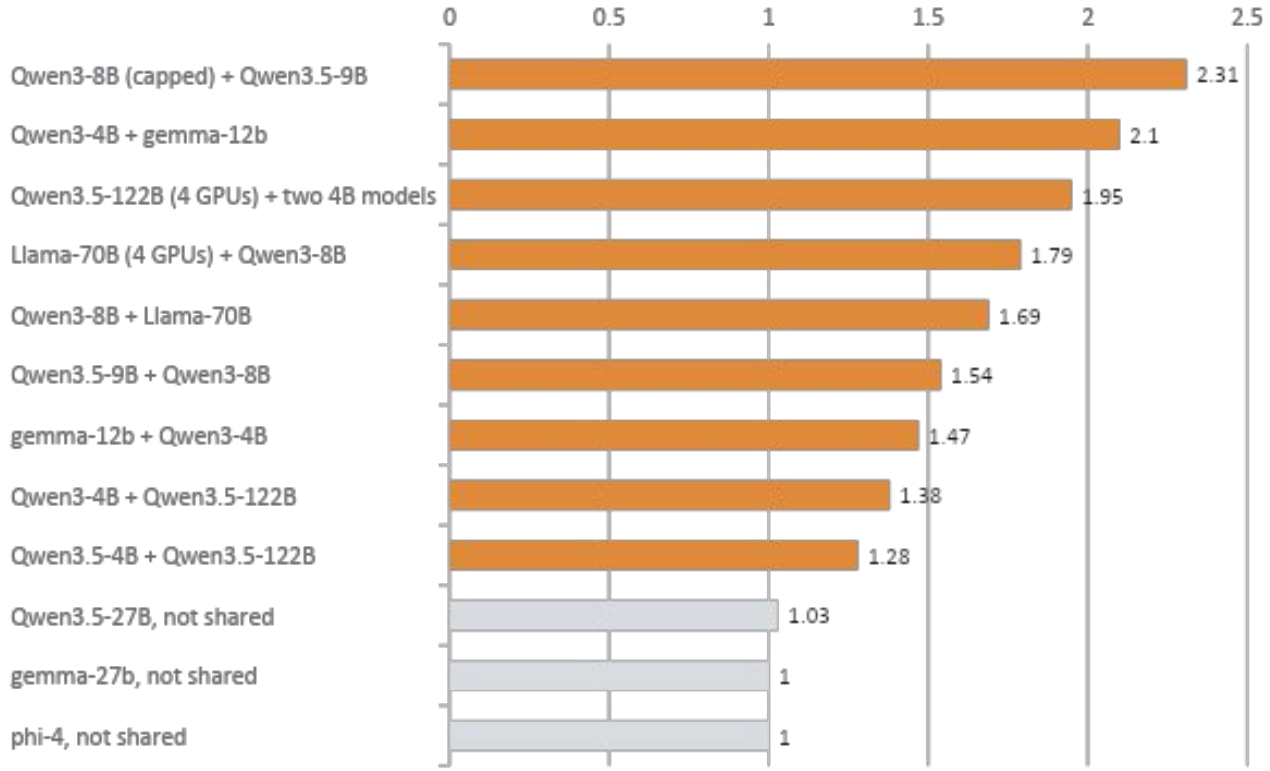
One more knob per model, set by measurement. The agent searches SM caps the way it searches memory. Experimental at the MPS layer today.

Same GPU, same memory — only the neighbours' work removed

1. **Deploy the final placement.** Every model up, with the GPU memory it gets when sharing.
2. **Load one model at a time.** The others stay up but idle.
3. **Compare with all loaded.** The only difference is the neighbours' work.

Less memory is not the cause

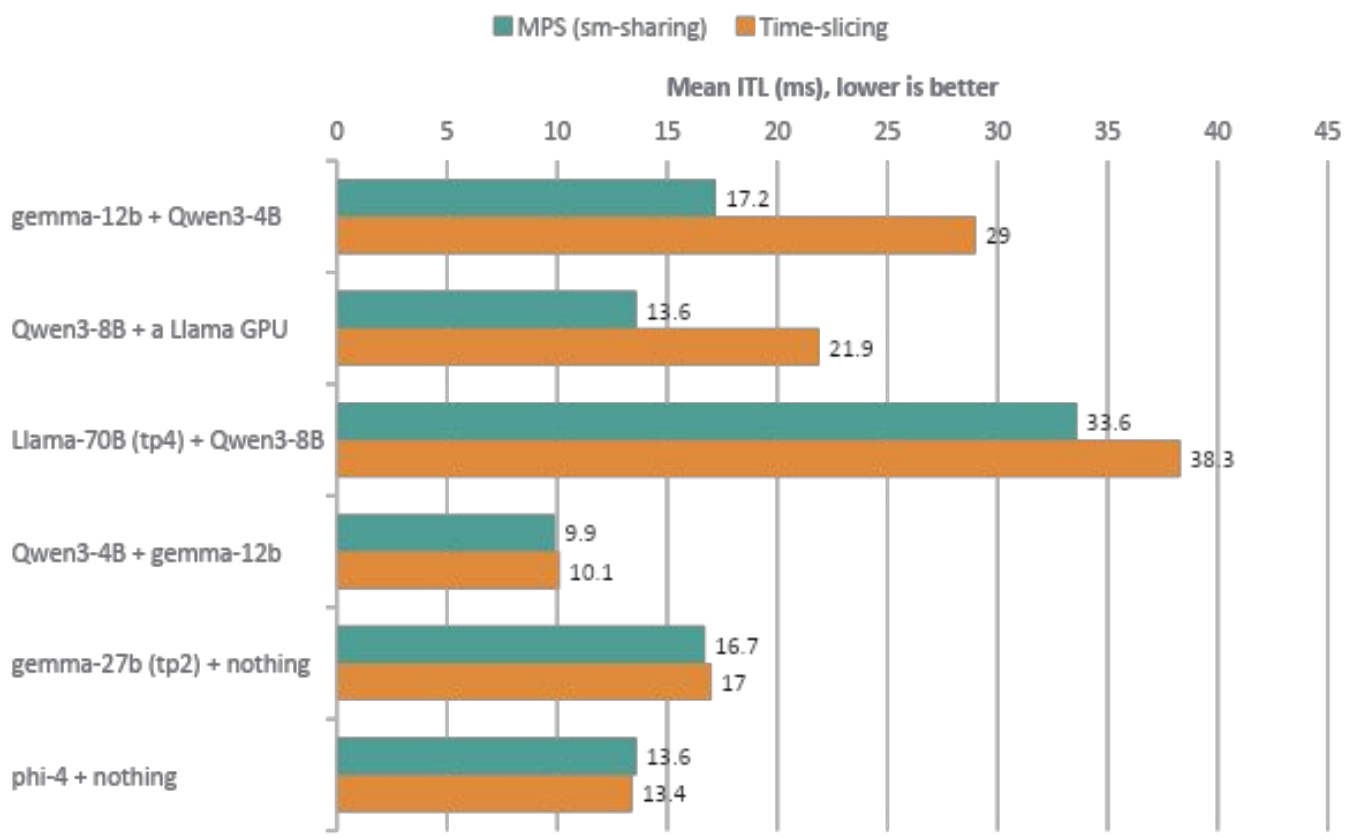
Llama-3.3-70B alone: 18.72 ms per token with the memory it gets when sharing, 18.66 ms with 3.7× more.



Each bar: a model's time per token with its GPU neighbours busy, divided by the same with them idle. **Grey** bars share no GPU (the control).

The price of sharing is 1.3–2.3× time per token and all of it is the neighbours' work.

Same placement, same memory: switch MPS to time-slicing



Text-only placement on KAI, both modes back to back, two cold deployments each. gemma-12b's ceiling: 23 ms.

MPS wins: two busy models on one GPU

gemma-12b +69%, over its ceiling. Qwen3-8B +61%. Llama, waiting on its shared GPU, +14%.

No difference: alone, or a light neighbour

gemma-27b and phi-4 alone: within 2%. Qwen3-4B's short steps fit in its own slices: +2%.

Multimodal fleet: time-slicing collapses

Qwen3.5-122B TTFT p95 85 s against 6 s; four more models miss their ITL. Same placement on MPS: all ITL met.

Time-slicing never won by more than 2%. MPS lets models on one GPU compute at the same time; time-slicing makes them take turns.

PART 5 · WHAT THE MEASUREMENTS SAY

Five things to take home

1. **Co-location saves GPUs, measured.** Text 10 → 8, multimodal 11 → 8: every model inside its ceilings in every cold redeployment, and on KAI.

2. **The agent thinks for free and measures to decide.** Most ideas are ruled out by the solver; each GPU measurement answers one question.

3. **Sharing costs 1.3–2.3× time per token.** All of it is the neighbours' work, not a smaller KV pool. The solver counts only memory, so measure.

4. **SM caps are a second knob, tuned for margin.** The agent sets them like memory. Explicit caps are an MPS-layer experiment; KAI deploys SM sharing.

5. **MPS, not time-slicing; search on MPS, serve on KAI.** Busy pairs ran 61–69% slower time-sliced. Latency carries over when every engine setting is in the fleet file.

PART 6

Running it yourself

Claude Code plugin with a skill, and six commands, four of them don’t use any GPUs. What your cluster needs, how the fleet file describes a workload, how to read a result that may certify nothing, and the agent that runs it all for you.