Elastic GPU Pools for Multi-Tenant LLM Fleets

32 tenants on 4 GPUs instead of 32 at 1× load across 3 seeds, with models activated in 0.161 to 0.766 s against 34.5 to 49.9 s cold

Pranav Varshney
Nodus Compute

Viswa Kotra
Nodus Compute

Abstract

Serving one model deployment per GPU gives each tenant a simple ownership boundary, but makes fleet cost scale with tenant count rather than useful work. We present a unified evaluation of a Nodus research prototype that runs an elastic GPU pool for heterogeneous LLM tenants. The prototype combines pooled serving, traffic-derived placement, sub-second model activation, idle release, and memory-budget enforcement behind one control plane, a configuration measured in research and not yet in the product. On one 8-GPU node, 32 synthetic tenants driven by identical replayed traffic use 4 GPUs at 1× load instead of 32 dedicated GPUs across 3 seeds. SLO-met requests per GPU-second improve by 7.6×, 8.0×, and 8.0×; cost per million SLO-met tokens falls by 7.8×, 8.1×, and 8.1×; lowest per-class SLO attainment is 99.7%, 99.0%, and 98.5%, with 0 errors. At 2× and 3× traffic, self-planned layouts use 5 and 6 GPUs. Planning from past traffic at 2× costs one GPU relative to perfect foresight and still uses 5.33× fewer GPUs than dedicated provisioning. In the prototype, model activation took 0.161–0.766 s at p50 versus 34.5–49.9 s from cold, and measured memory virtualization enforced caps across 8 CUDA allocation APIs while admitting 1.5–2× more active jobs than strict packing at useful throughput of at least 0.90. We report each component under its own accounting and do not multiply independent speedups.

1 Introduction

A tenant wants a model-specific response within a latency objective. Infrastructure is purchased as GPU capacity over time. When every tenant receives a dedicated GPU, cost therefore follows the number of deployments even when their aggregate traffic would fit on a much smaller fleet. The unused capacity is not an accounting detail: it is paid GPU time that produces no SLO-qualified output.

An elastic pool changes the allocation unit from a physical GPU to a schedulable tenant placement. That requires more than multiplexing CUDA processes. The pool must decide how many GPUs a workload needs, move model state when demand changes, reclaim memory from idle tenants, enforce memory budgets, and retain the service objective while tenants contend. A result on only one of these mechanisms cannot establish the fleet outcome.

This paper combines five previously separate measurement campaigns into one systems evaluation. We first measure the complete fleet against one dedicated GPU per tenant. We then vary offered load, compare planning with and without foresight, measure the transition operations that make layouts elastic, and test memory isolation and overcommit independently. The result is one bounded claim: for the measured replayed workload, an elastic

pool delivers substantially more SLO-met output per paid GPU while exposing the latency and evidence boundaries of sharing.

2 System model and useful-work accounting

A request is useful only when it completes without error and satisfies its tenant's service-level objective (SLO). We report SLO-met requests per GPU-second and dollars per million SLO-met tokens. A late response may consume GPU time and produce tokens, but it does not enter either useful-work numerator. This prevents raw utilization or token throughput from hiding queueing failure.

The prototype pool continuously maps tenant identities to active model instances. Model activation changes the model served by a running GPU group; reassignment changes the tenant-to-capacity mapping; idle release reduces the resident footprint of inactive tenants; memory caps bound each tenant's allocation surface. Placement selects a GPU count and layout from observed traffic. These operations share a control plane, but their measurements retain separate denominators.

The dedicated reference provisions one GPU per tenant. It is intentionally simple and strong on isolation and latency. The pooled arm is charged for every GPU it occupies for the full measurement window. Component ratios are never multiplied into the fleet ratio because the campaigns use different workloads and acceptance conditions.

3 Evaluation methodology

Fleet platform. The main study uses one 8-GPU node. The workload contains 32 synthetic tenants in four classes: full fine-tunes of an 8B model, separate-architecture models, LoRA adapters on a shared 8B base, and one 32B model served in FP8. Traffic is a paired Poisson/bursty replay. At 1× load, runs last 15 minutes and use 3 independent traffic seeds. Both arms receive identical arrivals, prompts, and output constraints.

Load ladder. We retain the same tenant set and trace shape at 1×, 2×, and 3× offered traffic. The 1× result spans 3 seeds. The 2× and 3× scaling results use seed 0. A configuration is accepted only when every reported tenant class passes its SLO gate and the run has no request errors. Two GPUs fail the 1× gate; four GPUs are the measured floor among tested layouts.

Planning regimes. Perfect-foresight planning sees the trace it will serve and provides an oracle bound. Deployable planning uses only past traffic. Comparing the two at 2× isolates the capacity cost of not knowing future arrivals. The dedicated arm remains fixed at 32 GPUs at every traffic level.

Component platforms. Activation and reassignment are measured on the fleet node. Active-memory overcommit is measured separately on one GPU in a separate host with synthetic CUDA job mixes. Allocation-cap coverage is a black-box suite over 8 CUDA allocation APIs. These component results explain feasibility; they are not relabeled as fleet throughput.

4 Fleet consolidation at matched traffic

Table 1: Paired 1× fleet results, 32 tenants, 15-minute runs. The dedicated arm uses 32 GPUs and has 100% SLO attainment with 0 errors in every seed.

Seed	Nodus GPUs	Lowest class SLO	Useful-work gain
0	4	99.7%	7.6× req/GPU-s; 7.8× lower cost
1	4	99.0%	8.0× req/GPU-s; 8.1× lower cost
2	4	98.5%	8.0× req/GPU-s; 8.1× lower cost

The pooled arm serves all 32 tenants on 4 GPUs in every seed, an 8× capacity reduction relative to dedicated provisioning. Nodus goodput is 2.05, 2.094, and 2.109 SLO-met requests

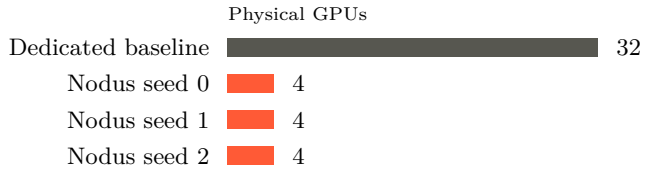


Figure 1: GPUs required to serve the same 32-tenant replay while meeting the stated SLO gate.

per GPU-second; the matched dedicated arms reach 0.271, 0.262, and 0.264. Nodus cost is $4.07–$4.10 per million SLO-met tokens, while the dedicated arms cost $31.92–$33.05.

The resource reduction is accompanied by a visible service tradeoff. Dedicated provisioning reports 100% SLO attainment. The pooled arm's lowest class attains 99.7%, 99.0%, and 98.5%. Its TTFT p95 is 0.21, 0.32, and 0.31 s; TPOT p95 is 31, 32, and 33 ms. Errors are 0/7,878, 0/7,929, and 0/7,983. Zero errors does not mean zero SLO misses, so both outcomes remain explicit.

During the same 15-minute windows the system performed 203 model-weight reassignments in seed 1 and 176 in seed 2, each with p95 latency of 0.39 s. Transition costs are therefore included in the fleet SLO result rather than removed as setup.

5 Scaling and traffic-derived planning

Table 2: Self-planned GPU demand as offered traffic increases. The dedicated reference remains at 32 GPUs.

Traffic	Nodus GPUs	Lowest class SLO	Capacity reduction
1×, 3 seeds	4	98.5–99.7%	8×
2×, seed 0	5	99.83%	6.4×
3×, seed 0	6	99.78%	5.333×

At 2× traffic, the self-planned layout uses 5 GPUs, completes 3.318667 SLO-met requests per GPU-second, costs $2.6016 per million SLO-met tokens, attains 99.98% over the full run, and records 0 errors. The dedicated arm uses 32 GPUs, completes 0.518854 SLO-met requests per GPU-second, and costs $16.6884 per million SLO-met tokens.

At 3× traffic, the prototype selected 6 GPUs and every class remained at or above 99.78% attainment with 0 errors. It reached 5.329× the SLO-met requests per GPU-second and 5.384× lower cost per SLO-met token than dedicated provisioning ($2.1088 versus $11.3528 per million). GPU demand grows from 4 to 5 to 6 as traffic triples, rather than tracking tenant count.

Table 3: The cost of deployable planning at 2× traffic, seed 0.

Planner	GPUs	SLO-met req/GPU-s	$/M SLO-met tokens
Past traffic only	6	2.765556	$3.1220
Perfect foresight	5	3.318667	$2.6016
Dedicated	32	0.518854	$16.6884

Planning from past traffic pays exactly one GPU relative to the perfect-foresight layout. It still uses 5.33× fewer GPUs and reduces cost by 5.34× relative to dedicated provisioning. This is the deployable comparison: the oracle identifies headroom, while the past-traffic row shows what the measured planner can choose without seeing the future.

6 Fast transitions make the pool elastic

A placement plan is operationally useful only if model state can follow it. Cold-starting a model takes 34.5–49.9 s in the measured environment, and the combined service reaches cold readiness in 170.0 s. A transition on that timescale cannot occur repeatedly inside an interactive SLO.

Table 4: Measured transition latency and fidelity.

Operation	Measured latency	Reference	Correctness
2-GPU model activation	0.161 s p50	34.5–49.9 s cold	Bit-exact tensors
1-GPU model activation	0.766 s p50	34.5–49.9 s cold	Deterministic output
Fleet reassignment	0.39 s p95	176–203 events/15 min	Included in SLO
Idle resume, 8B–120B	0.44–2.77 s	170.0 s cold service	25/25 outputs match

The measured 45–300× activation reduction turned reassignment in the prototype from an exceptional restart into a routine scheduling action. Idle suspension reduces a tenant to roughly 1 GB in the component study and 4,529 MiB in the combined end-to-end test. These operations explain how a 4-GPU layout can follow traffic from 32 tenants without keeping every model fully resident.

7 Memory budgets and active-job overcommit

Pooling model state is useful only if tenant memory budgets remain enforceable and extra admitted work continues to make progress. We therefore separate allocation-surface coverage from active-job density. 'Exact cap' means exact enforcement in the measured suite; it is not a confidentiality or side-channel claim.

Table 5: Memory virtualization results from the independent component campaign.

Experiment	Nodus	Comparator	Outcome
Allocation coverage	8 APIs capped	2 HAMi leaks	All paths tested
Active admission	1.5–2× strict packing	Strict packing	$\geq$0.90 throughput
Unmanaged overcommit	Controlled admission	0.03 useful throughput	0 valid-arm OOMs

On one GPU, active admission fits 1.5–2× as many jobs as strict packing while maintaining at least 0.90 of solo useful throughput and recording no out-of-memory failures in valid arms. Unmanaged overcommit collapses to 0.03 useful throughput. Admission count alone would hide that difference, so density, progress, and failures are reported together.

The HAMi-core comparison concerns the tested allocation paths only. It does not establish a general defect rate, exhaustive CUDA compatibility, or a throughput comparison. Likewise, the no-OOM result is evidence for the reported job mixes, not a universal safe oversubscription ratio.

8 Integrated interpretation

The fleet result is the primary outcome. Planning chooses a compact layout; fast activation and idle release let that layout change; memory budgets prevent one tenant from consuming the tested allocation surface; active admission increases density when jobs have reclaimable headroom. The components form an operational chain, but their ratios measure different denominators and are not multiplicative.

Dedicated GPUs remain the reference for strict ownership and the lowest observed latency. Pooled service is slower at every reported latency percentile, though it remains inside the measured SLO gate. Hardware partitioning remains appropriate where a workload requires a stronger fault, bandwidth, or security boundary than this evaluation establishes.

8.1 Evidence boundary

The fleet study covers one 8-GPU node, synthetic tenants, replayed traffic, three seeds at 1× load, and one seed at 2× and 3×. The dedicated reference is a per-tenant capacity model and is not claimed to fit physically on the single Nodus node. The memory-overcommit campaign uses a separate GPU host and must remain distinct from the fleet result. None of the measurements proves adversarial isolation, universal application compatibility, production demand, or cloud-billing savings when spare node capacity cannot be resold.

9 Reproducibility

Every headline comparison fixes tenant count, replayed arrivals, request contents, output limits, and SLO classification between arms. Per-seed fleet outcomes are reported separately. Transition events remain inside the measured windows. The load-scaling rows state when only seed 0 is available, and oracle planning is labeled separately from deployable planning.

The source evidence package retains the paired per-seed summaries, request denominators, latency percentiles, error counts, planner outputs, transition counts, allocation-suite outcomes, and active-admission controls. Reproducing the complete claim requires replaying each campaign under its own hardware and accounting; component outcomes should not be combined into a synthetic super-run.

10 Conclusion

An elastic GPU pool can make tenant count a logical property rather than a physical GPU count. On the measured workload, a Nodus research prototype served 32 tenants on 4 GPUs instead of 32 across 3 seeds, improving useful work per GPU by 7.6–8.0× while retaining 98.5–99.7% lowest per-class SLO attainment and 0 errors. At higher load it expanded to 5 and 6 GPUs, and planning without foresight cost one GPU at 2× traffic. Sub-second activation, idle release, memory-cap coverage, and controlled overcommit provided the measured mechanisms that made those layouts operational in the prototype, none of which is yet in the product. The result is a bounded systems claim on real hardware: fewer GPUs for the same replayed tenants and accepted SLO, with the latency and isolation tradeoffs stated explicitly.