Batch Work in Serving Gaps

One serving GPU gives back 91% of a batch GPU at 1 request per minute and 34% at 8, at a serving p95 of 7.6 to 8.8 s

Pranav Varshney
Nodus Compute

Viswa Kotra
Nodus Compute

Abstract

A GPU kept for one latency-sensitive model does no work between requests, and at low load that is most of the time. AI work per GPU and per fleet grows if those gaps run batch work without a second GPU. Nodus, the infrastructure execution layer for AI workloads, measured filling a serving GPU's gaps with preemptible training on one GPU, with GPU memory snapshots as the preemption. The served model, Qwen2.5-1.5B on vLLM with 90% of GPU memory, is a scale-to-zero Function restored from its snapshot through the Nodus node agent. The batch work, an 85.35 million parameter deterministic PyTorch trainer, is stopped by one snapshot through a second runc executor when a request arrives and restored from that snapshot when the queue drains. When the server's restore started as soon as the trainer's CUDA state left the GPU, 1.40 s after the stop was asked, the trainer ran at 91% of a dedicated GPU's step rate at 1 request per minute (88% in a 30-minute replicate), 76% at 2, 60% at 4, 34% at 8 and 11% at 16, and serving p95 was 7.6 to 8.8 s. Each preemption cost a nearly fixed 9.3 to 9.7 s of batch time, so the batch share falls in a straight line with the preemption rate. The price is latency: p95 was 3.3 to 4.6 s higher than with snapshot swaps alone and 7.6 to 7.9 s higher than on a warm dedicated GPU, which suits scale-to-zero endpoints that already accept a restore, not hot endpoints. The 271 trainer restores over 21 jobs continued bit for bit, with 0 of 32,442 loss values, 0 of 3,243 parameter bit-sums and 0 of 192 SHA-256 digests different from an uninterrupted run, and all 358 server restores and 278 trainer stops succeeded. A stop on a job's last step made 2 restores fail because the process finished first, and a trainer build with an added CUDA synchronize broke 3 restores, so its run is excluded. We recommended a flagged node-local gap-fill mode gated on the measured preemption rate, and it has since shipped as a Beta feature behind the per-organization flag node.gpu_snapshot_gap_fill through PR #747, for an organization's own private Pool hosts and Jobs that opt in with placement.interruptible: Prefer. The results come from one host, one arrival seed per rate and 12 to 23 requests at 1 request per minute, and no independent verifier has rerun them. A follow-up beside a warm, always-on endpoint recovered much less: with TTFT p95 and TPOT p95 held to at most 1.2× serving alone, the same trainer ran at 23.1% of a dedicated GPU at 10% of the endpoint's saturation, 9.7% at 30% and 6.4% at 60%, bit-identical, because a request was in flight 80.5% to 99.9% of the time. Capping the trainer's SMs under MPS changed all of its loss values, and an MPS client could not take a GPU memory snapshot.

1 Introduction

More AI per GPU and per fleet means that a GPU does useful work for more of its hours. A GPU kept for one latency-sensitive model is a common exception. It holds the model so that a request is answered in seconds, and between requests it does nothing. At a few requests per minute that is almost all the time. A fleet that wants that time back has two usual choices: keep the model warm and accept the idle GPU, or release the GPU and pay a cold start. Our reports 033 [1] and 034 [2] showed a third: a containerized vLLM [3] server comes back from a GPU memory snapshot in seconds, so it can leave the GPU when idle. Report 035 [4] showed that a snapshot also moves a live PyTorch [5] training process off a GPU and back exactly.

This report puts the two together. While the server is out, a preemptible training job runs on the GPU. A request stops the job with one GPU memory snapshot and restores the server from its own snapshot. When the queue drains, the server leaves and the job is restored from its snapshot. We asked how much of a dedicated batch GPU this gives back at Poisson rates of 1 to 16 requests per minute, what it costs serving latency against snapshot swaps alone and against a warm dedicated GPU, whether the batch result stays bit-identical, and up to what load it pays.

The answer has a clear price. At low load most of a batch GPU comes back, 91% at 1 request per minute, and the share falls steadily as requests come more often, to 11% at 16 per minute. Serving p95 stays far under 15 s at every rate, but every request that finds the trainer on the GPU waits for one preemption and one restore. That fits a scale-to-zero endpoint that already accepts a restore, and not an endpoint that is kept warm for sub-second answers.

2 Experimental setup

Platform path. One host with one GPU (GPU persistence mode on) ran the real Nodus node agent with its guarded runc executor on containerd 2.4.1 and runc 1.5.2, with CRIU 4.3 [6] and NVIDIA cuda-checkpoint [7], which used the CUDA 13.0 forward-compatible libcuda. A research test of the agent package, which is not merged, drove the runs with binaries built by GitHub Actions. A second build, used for the last 4 runs, differs only in starting the next job when a restored job turns out to have finished.

The served model, a Function. Qwen2.5-1.5B-Instruct [8] on vLLM 0.30.0 with gpu_memory_utilization 0.90 and a maximum model length of 2048 answers each request with up to 64 tokens at temperature 0, for one of 8 short prompts. It is one Function with GPU snapshots on. It starts cold once per test process, answers after 121 to 123 s and takes its snapshot, 7.57 to 7.58 GB, of which 3.41 GB is the weights file kept out of the process image. Every later server is restored from that snapshot through the node agent, and removing it is a desired-state push without it. At 0.90 the server's KV cache fills the GPU, 20,169 MiB, most of the GPU's memory, so a trainer that reserves 7.1 GB cannot run beside it, and the GPU has to be handed over.

The batch work, a Job. The trainer is report 035's: a deterministic character-level GPT with 12 layers, width 768 and 85.35 million parameters, trained with AdamW [9] and bf16 autocast on blocks of 256 in batches of 32, with PyTorch in deterministic mode, on PyTorch 2.13.0 in the server's image. Each job is the same 1,500-step run with the same seed, so one job run uninterrupted is the reference for every other job's bits. It runs through a second runc executor in its own containerd namespace, with the calls the node's Job reclaim snapshot makes: start, stop with one snapshot that ends the process, and restore with the snapshot as the container's checkpoint. Snapshots stay on the host's local disk, with no upload. Before each stop, the harness asks the trainer to stop at its next step boundary and release PyTorch's free cached memory, which takes 0.22 s and brings its reserved memory from 7.1 GB to 2.2 GB, with 1.46 GB live.

Traffic. Requests arrive as a seeded Poisson process at 1, 2, 4, 8 and 16 requests per minute, and every arm at one rate sees the same arrival times. Latency runs from the scheduled arrival to the answer's last token. The scheduler serves every queued request as one batch,

removes the server as soon as nothing is queued or in flight, and puts the trainer back in every gap. A request that finds the trainer on the GPU starts a preemption.

Table 1: The arms. All ran on the same GPU in one test process, and a second process repeated three points. In both gap-fill arms a request stops the trainer with one snapshot and the server is restored from its own. The sequential arm starts that restore after the stop has returned, and the overlapped arm as soon as the trainer's CUDA state is off the GPU, while its process is still being dumped. The batch share of a run is its executed training steps per second over the dedicated batch arm's 4.248 steps per second.

Arm	What is on the GPU
Dedicated batch GPU	The trainer alone, jobs back to back
Dedicated warm GPU	The server alone, always resident
Snapshot swaps alone	The server, restored per request, nothing else
Gap-fill, sequential	The trainer in the gaps, the server after its stop
Gap-fill, overlapped	The trainer in the gaps, the server once the GPU is free

3 How much of a batch GPU comes back

Table 2: Batch share and serving p95 at each rate. Batch is the trainer's executed steps per second as a share of the dedicated batch GPU's 4.248 steps per second. In parentheses, the replicate of a point in the second process, on a new 30-minute arrival stream at 1 request per minute and on the same arrivals at 2. The warm GPU was measured at 8 and 16 requests per minute.

Requests per minute	Overlapped: batch, p95	Sequential: batch, p95	Swaps alone: p95	Warm GPU: p95
1	91%, 7.6 s (88%, 8.0 s)	89%, 10.6 s	4.3 s	Not measured
2	76%, 8.4 s	71%, 10.7 s (72%, 10.5 s)	4.2 s	Not measured
4	60%, 8.8 s	56%, 11.2 s	4.3 s	Not measured
8	34%, 8.1 s	25%, 11.9 s	4.5 s	0.52 s
16	11%, 8.8 s	7%, 12.0 s	4.2 s	0.93 s

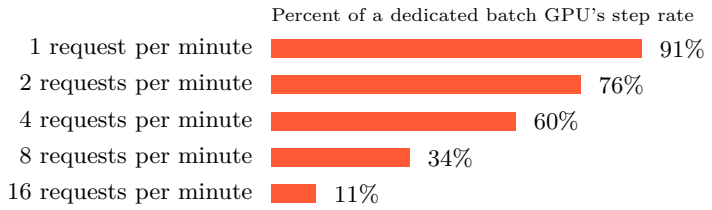


Figure 1: Batch throughput of the overlapped gap-fill arm as a share of a dedicated batch GPU, higher is better.

Most of a batch GPU at low load. At 1 request per minute the overlapped arm trained at 91% of a dedicated GPU's rate, and 88% in its 30-minute replicate. The share falls as requests come more often: 76% at 2, 60% at 4, 34% at 8 and 11% at 16 requests per minute. The sequential arm gave back a little less at every rate: 89%, 71% and 72% in its replicate, 56%, 25% and 7%. The shares are against the main process's 8-minute batch arm. Against the second process's own 3-minute batch arm, which ran 1.0% faster, its three points come out 1 point lower, at 75%, 71% and 87%.

Serving stays within 15 s. Every point kept p95 within 15 s, the worst at 12.0 s, so the batch share decides how far gap-fill pays. With the overlapped order, the GPU returned at least 50% of a batch GPU up to 4 requests per minute and at least 25% up to 8. The sequential order returned at least 50% up to 4 requests per minute and missed 25% at 8, with 24.8%. Overlapped p95 hardly moves with the rate, 7.6 to 8.8 s, because the tail is a request that finds the trainer running and waits for one preemption.

4 What a request pays

Table 3: Latency by what the GPU was doing when the scheduler saw the request, all rates pooled, with the excluded run left out. A request that finds the server being restored shares that restore with the request that started it.

Arm	Request found	Requests	p50	p95	p99
Overlapped	The trainer running	137	7.2 s	8.9 s	9.5 s
Overlapped	The server being restored	68	3.1 s	5.4 s	5.8 s
Overlapped	The trainer being stopped	10	6.6 s	7.2 s	7.2 s
Overlapped	The server resident	6	0.7 s	3.0 s	3.0 s
Sequential	The trainer running	118	10.0 s	12.0 s	12.4 s
Sequential	The server being restored	95	4.1 s	9.3 s	9.8 s
Sequential	The server resident	12	0.7 s	1.0 s	1.0 s

One preemption and one restore. A request that finds the trainer running in the overlapped arm waits for the trainer to stop at its step boundary and for its CUDA state to leave the GPU, 1.40 s after the stop was asked, then for the server's restore beside the trainer's dump, 5.09 s, and then for its own decode, 0.54 s. Its p50 was 7.2 s and its p95 8.9 s. Requests that arrive while the server is being restored or is resident share its stay, which is why the p50 falls to 4.3 s at 16 requests per minute while the p95 stays at the trainer case.

The price against the baselines. Snapshot swaps alone, with nothing else on the GPU, pay one restore, and their p95 was 4.2 to 4.5 s. A warm dedicated GPU answered at a p50 of 0.52 s and a p95 of 0.52 and 0.93 s. Gap-fill's overlapped p95 was 3.3 to 4.6 s above swaps alone at each rate, and 7.6 and 7.9 s above the warm GPU at the two rates where both ran. Gap-fill therefore belongs on an endpoint that already releases its GPU between requests and accepts a restore, where it adds a few seconds to the tail, and not on an endpoint that is kept warm for sub-second answers.

5 Each preemption costs a fixed amount of batch time

Table 4: Batch time lost per preemption, C, computed from each run as one minus its batch share, times 3600 s, over its preemptions per hour. A preemption is one stop of the trainer. At 1 and 2 requests per minute the second value is the replicate.

Arm	1	2	4	8	16	Median
Overlapped, C	9.4 s, 9.7 s	9.6 s	9.3 s	9.5 s	9.3 s	9.47 s
Overlapped, preemptions per hour	36, 46	92	156	250	343	
Sequential, C	12.5 s	12.2 s, 12.1 s	12.6 s	12.3 s	13.1 s	12.40 s
Sequential, preemptions per hour	33	84, 84	126	220	255	

A straight line. In the overlapped arm, every run lost 9.3 to 9.7 s of batch time per preemption, whatever the rate. That time is the trainer's stop, the server's restore and stay, the server's removal and the trainer's restore. So the batch share is 1 minus the preemptions per hour times C over 3600 s, and with one C of 9.47 s that line comes within 1.7 points of every overlapped point. The sequential arm lost 12.1 to 13.1 s per preemption, and its line comes within 5.0 points. A request causes a preemption only if it finds the trainer on the GPU. Requests that arrive while the server is being restored or is serving share its stay, so the arms saw 0.23 to 1.00 preemptions per request, fewer at the higher rates. The cost of one preemption, not the request rate, is what a node can measure and gate on.

6 Starting the restore before the dump ends

The two orders. The stop snapshot ends the trainer's process. cuda-checkpoint first moves its CUDA state to host memory, which frees the GPU, and CRIU then dumps the process. The 3.71 GB snapshot took 3.60 s to the end of the dump. The overlapped order started

Table 5: What each transition took, from its request to its end, counted over every gap-fill and swap run, with the excluded run left out. p50 is the median.

Transition	Count	p50	p95
Trainer stops at its step boundary and releases its cache	243	0.22 s	
GPU free of the trainer, overlapped, from the stop's request	132	1.40 s	1.55 s
Trainer's stop snapshot, to the end of the dump	243	3.60 s	3.78 to 3.85 s
Server restore, swaps alone	83	3.59 s	3.77 s
Server restore after the dump, sequential	115	3.61 s	5.56 s
Server restore beside the dump, overlapped	133	5.09 s	5.66 s
Server removed, until the GPU is free	329	0.22 s	0.30 s
Trainer restore from its snapshot	240	2.41 s	2.52 to 2.65 s

the server's restore when the GPU's used memory fell to idle during that stop, 1.40 s after the stop was asked, and the restore then ran 1.48 s slower beside the dump, 5.09 s against 3.61 s. The server was ready 6.55 s after the stop was asked. In the sequential order, the restore started 5.72 s after the stop was asked and was ready after 9.38 s.

What the overlap is worth. That 2.8 s difference is not all overlap. In the sequential arm the restore started 2.14 s after the stop returned, because the test harness settles a stop, removing the stopped container and the previous snapshot, before it starts the server, while the overlapped arm settles it during the server's restore. Without that gap, a sequential order would be ready after 3.59 plus 3.61 s, 7.20 s, so starting the restore before the dump ends saves about 0.7 s on this 3.7 GB stop, by our computation, and keeping the clean-up off the request's path saves the rest. Measured between the two arms on the same arrivals, the overlapped order lowered p95 by 2.1 to 3.8 s and p50 by 1.7 to 2.8 s, and gave back 2 to 9 points more of a batch GPU. We expect the overlap to be worth more for a trainer with more memory, because the GPU is free once the CUDA state has moved, while the dump also writes the process's host memory, but we did not measure that.

7 The batch result is bit-identical

Table 6: Every job compared with the reference job, which ran from its cold start to step 1,500 with no stop. Loss bits are compared at every step, parameter bit-sums every 10 steps, and SHA-256 digests of the parameters, the AdamW state and the random number state at steps 500, 1000 and 1500.

Process	Jobs	Trainer restores	Loss values compared	Bit-sums compared	Digests compared	Different
Main	12	203	19,329	1,932	38 × 3	0
Second	9	68	13,113	1,311	26 × 3	0
All	21	271	32,442	3,243	64 × 3	0

Exact through every preemption. Each of the 271 trainer restores continued the reference run bit for bit, in two test processes. None of 32,442 loss values, 3,243 parameter bit-sums and 192 digests differed. The most-preempted job was restored 60 times over three runs at 8 and 16 requests per minute, and all 556 of its steps match. The trainer needed no checkpoint code, only its step-boundary handshake and determinism. All 358 server restores of both processes succeeded, and so did all 278 stop snapshots of the trainer. The 324 answers decoded alone had exactly one text per prompt across all server restores and both processes.

8 Two problems the run found

A stop on a job's last step. 276 trainer restores were attempted, and 271 succeeded. Two of the 5 failures came from a stop that landed on a job's last step. The trainer holds after its final digest, and the restored process runs to its end and exits as soon as runc restores it, before its CUDA state is moved back, so the move fails. That happened once in each process.

The first build restarted the finished job from step 0, which is why its sequential run at 2 requests per minute shows 32% progress against 71% executed steps, and the second build starts the next job instead, which its replicate shows at 72% for both. All jobs are the same run, so executed steps count as batch work. For the product, a Job that finished during its restore must be reported as finished, by reading the process's exit before reporting a failed restore.

An added synchronize broke restores. For part of the first process, the trainer had an extra CUDA synchronize right after its hold, meant to make a restored process wait for its CUDA state. All 3 restores of that build failed, the job restarted cold, and the trainer was reverted to report 035's code. None of the reverted trainer's 273 restores failed for that reason, and the other 2 failures are the stops above. The affected run, overlapped at 2 requests per minute, is excluded from every table. The second process ran it again on the same arrivals, at 76% and a p95 of 8.4 s, where the excluded run had shown 76% executed and 8.2 s. That a synchronize in that window breaks the restore is an observation from 3 of 3 restores, not a verified driver behavior.

9 What we recommend

The measurements point to one design, which we recommended and which has since shipped as a Beta feature behind the per-organization flag node.gpu_snapshot_gap_fill, through PR #747, for an organization's own private Pool hosts and Jobs that opt in with placement.interruptible: Prefer. It is a flagged, node-local gap-fill mode in nodusd: a GPU that serves one scale-to-zero snapshot Function could run one preemptible Job attempt that opted in, in the Function's gaps. A request for the Function while the Job holds the GPU would stop the Job with a local stop snapshot, kept on the node's disk with no upload, because the same node restores it seconds later. The Function's restore would start as soon as the Job's CUDA state is off the GPU, signaled from inside the executor's checkpoint call rather than polled as here. The SDK would offer the step-boundary handshake this trainer used as a preemption point the Job can declare, and the node would admit a Job only if it passes the existing snapshot checks and its snapshot is small enough to leave the GPU quickly.

When to fill. The node would know C, the batch time one preemption costs, and count preemptions. For a bar of X, the share of a dedicated batch GPU that filling must return, it would fill while the preemptions per hour stay at or below 1 minus X, times 3600 s, over C, and above that keep the Function warm and place the Job elsewhere. With the overlapped C of 9.47 s, that is 190 preemptions per hour for X of 50% and 285 for X of 25%. Applied to the measured points, this gate keeps exactly the points that met each bar, but C comes from the same points, so that is a consistency check and not a test. The rule is a rate bound and not a hold timer, because with Poisson arrivals the expected wait for the next request is the same at every moment. The Function's latency budget must allow one preemption, as its p95 rose from 4.2 to 4.5 s with swaps alone to 7.6 to 8.8 s. A Job that finishes during its restore would be reported as finished.

10 Scope

All runs come from one GPU and one host, with one arrival seed per rate. The low rates have few requests: 12 at 1 request per minute in 20 minutes, 23 in the 30-minute replicate and 4 for swaps alone at 1 request per minute, so p99 there is the largest of a small sample. Two points were replicated and agree with the main run. Both snapshots stayed in the page cache of a host with plenty of memory, on local disk, with no upload, which is a best case. There was one served model, a small one, and one small trainer whose stop snapshot was 3.71 GB, and a trainer with more GPU memory moves more at each preemption. Several Functions per GPU with a filler, several fillers, streaming requests and a gateway were not measured. The scheduler is research code in the test process, it polled the GPU's memory for the overlap signal, and its sequential order includes the harness's clean-up. The rate gate is derived from these measurements, not measured as a running rule. No independent verifier has rerun the experiment.

11 Related work

Sharing a GPU between deep learning work is well studied. Gandiva [10] time-slices GPUs among training jobs by suspending and resuming them at iteration boundaries. Salus [11] gives deep learning applications fine-grained sharing primitives, including fast job switching. PipeSwitch [12] lets training and inference share a GPU by switching between them quickly, pipelining the model's transfer, so that idle serving capacity can train. AntMan [13] runs opportunistic jobs on the GPUs of resource-guaranteed jobs and scales them inside the framework. REEF [14] preempts best-effort GPU kernels in favor of real-time inference, and Orion [15] schedules best-effort work at kernel level around latency-critical inference. These systems keep both workloads in GPU memory or move model state through the framework. Here the server holds 90% of GPU memory, so the batch process leaves the GPU entirely, and it does so without framework support.

Transparent checkpointing saves a whole process without the program's help. CRIU [6] checkpoints Linux processes and containers, NVIDIA's cuda-checkpoint [7] moves a process's GPU state into host memory so that CRIU can save it, and CRIUgpu [16] combines the two for GPU workloads. Singularity [17] uses transparent checkpointing to preempt and migrate deep learning jobs across a fleet. Our reports 033 and 034 measured this stack restoring inference servers, and report 035 measured it for training through a reclaim. What this report adds is the measured frontier of batch work in a serving GPU's gaps through the Nodus node agent, with a snapshot as the preemption: the batch share at each request rate, the serving latency it costs, the fixed cost of one preemption and bit-exact training through 271 restores.

12 Follow-up, 9 November 2026: the warm-endpoint case

The question. The gaps above exist because the endpoint scales to zero. Many endpoints instead stay warm at low average load, so their model keeps its GPU memory and the GPU never empties. We asked how much of a dedicated GPU the same deterministic trainer recovers beside such a warm endpoint on one GPU, at 10%, 30% and 60% of the endpoint's saturation rate, while the endpoint's TTFT p95 and TPOT p95 each stay at most 1.2× what it shows serving alone at the same load. We also asked whether the trainer's results stay bit-identical, whether either process can still take a GPU memory snapshot, and whether the sharing works across separate containers. This is a research measurement, and none of it is built into Nodus.

Setup. One host with one GPU served Qwen2.5-1.5B-Instruct [8] on vLLM 0.30.0 [3] with gpu_memory_utilization 0.5, prefix caching off and streamed greedy answers of fixed length. Requests arrived as a seeded Poisson process, with prompt and answer lengths drawn from a public production conversation trace of 2023 and capped at 2,048 and 512 tokens. A closed loop of 128 concurrent requests saturated the endpoint at 10.193 requests per second, so the three loads were 1.019, 3.058 and 6.116 requests per second, with 120 s of arrivals at 10% and 90 s at 30% and 60%. Every arm at one load and seed sent the same requests at the same times. Each load ran three seeds, with 353, 807 and 1,659 answered requests, and the 4 requests that streamed no text, the same ones in every arm, are left out. The trainer is the 85.35 million parameter one above, started fresh in each window. Its share is the steps it completed during a window's arrivals over its dedicated rate on the same GPU, 4.106 steps per second. The two processes shared the GPU through NVIDIA's Multi-Process Service (MPS) [18], with the endpoint as an ordinary client.

Arms. Serving alone ran as a plain process and as an MPS client, and the MPS client is the baseline of every co-located arm. In three static arms the trainer ran as a second MPS client, capped at 25% or 50% of the GPU's SMs, or uncapped at below-normal priority. In three controlled arms it was uncapped at below-normal priority, launched one kernel at a time, and was frozen and thawed with SIGSTOP and SIGCONT by a controller that sees each request's send, first token and streamed tokens, as a node's request proxy would. The prefill shield freezes the trainer from each request's send to its first token. The governor adds a decode duty cycle: from the gaps between tokens with the trainer running and frozen, it measures how much a running trainer stretches a decode step, and it lets the trainer run

only so much of every 200 ms that the stretch stays within a target, 0.2 or 0.4. Launching one kernel at a time, so that a freeze takes the trainer off the GPU within one kernel, cost it 13% of its dedicated rate by itself, 3.567 against 4.106 steps per second, and that cost is inside the controlled arms' shares.

Budget and statistics. An arm is within the budget when its TTFT p95 and TPOT p95 are both at most 1.2× those of serving alone as an MPS client, over the same requests. Latency changes are paired by request, with 95% intervals from a seed-stratified paired bootstrap of 2,000 resamples. The trainer's interval is a bootstrap over the three seeds' windows, which with three windows equals their range.

Table 7: The governor at a target of 0.2, three seeds per load. Requests are the answered requests. The trainer column is its share of a dedicated GPU. The latency columns are the change against serving alone as an MPS client, in milliseconds and in percent, with 95% intervals in milliseconds.

Load	Requests	Trainer [95% CI]	TTFT p95 [95% CI]	TPOT p95 [95% CI]
10%	353	23.1% [19.9, 26.2]	+2.3 (+2%) [-2.3, +4.5]	+1.1 (+10%) [+0.8, +1.4]
30%	807	9.7% [9.5, 10.0]	+1.6 (+1%) [-3.3, +6.6]	+1.0 (+8%) [+0.8, +1.3]
60%	1,659	6.4% [5.7, 7.0]	+6.0 (+3%) [-0.8, +11.4]	+1.8 (+8%) [+1.3, +2.3]

Within the budget the trainer got 23.1% of a dedicated GPU at 10% load, 9.7% at 30% and 6.4% at 60%. TTFT p95 moved by +1% to +3%, with intervals that include zero, and TPOT p95 rose 8% to 10%. MPS by itself cost the endpoint some TPOT: serving alone as an MPS client had a TPOT p95 5%, 4% and 5% above serving alone as a plain process, each interval above zero. Against the plain process, which is what the endpoint runs when nothing is harvested, the governor's TPOT p95 was 12% to 16% higher, still within 1.2×.

Table 8: Every co-located arm at each load: the trainer's share of a dedicated GPU, then the change in TPOT p95 against serving alone as an MPS client. Within budget gives the loads at which TTFT p95 and TPOT p95 both stayed within 1.2×. Bit-exact says whether every trainer value equalled the uninterrupted dedicated run. The uncapped trainer ran at MPS's below-normal client priority, and the governor rows give its target.

Arm	10% load	30% load	60% load	Within budget	Bit-exact
25% SM cap	33.6%, +31%	29.6%, +44%	27.2%, +114%	None	No
50% SM cap	53.7%, +75%	49.1%, +106%	38.8%, +335%	None	No
Uncapped	64.1%, +127%	58.1%, +165%	40.2%, +271%	None	Yes
Prefill shield	52.8%, +80%	44.7%, +75%	27.3%, +53%	None	Yes
Governor, 0.2	23.1%, +10%	9.7%, +8%	6.4%, +8%	All	Yes
Governor, 0.4	31.3%, +26%	20.4%, +20%	13.0%, +18%	60% only	Yes

Running the trainer freely costs decode. Uncapped at below-normal priority, the trainer reached 64.1%, 58.1% and 40.2% of a dedicated GPU, but TPOT p95 rose 127% to 271% and TTFT p95 50% to 131%. The SM caps returned less trainer throughput and still missed the budget at every load. The prefill shield alone held TTFT p95 to +3% at 10% and 30% load, but TPOT p95 rose 53% to 80%. The governor's target sets the harvest: at 0.4 it returned 31.3%, 20.4% and 13.0% at TPOT p95 +26%, +20% and +18%, and only the last is within the budget, because the 30% load figure is 20.5% before rounding.

Why the harvest is small. A warm endpoint at modest load has almost no time between requests. With the endpoint alone as an MPS client, at least one request was in flight 80.5% of the time at 10% load, 99.5% at 30% and 99.9% at 60%. A trainer that waits for idle time therefore gets little, and a co-runner has to share the endpoint's decode, where every step it runs slows the endpoint's tokens. So the share falls as load rises, and a tighter budget returns less.

An MPS SM cap changed the trainer's results. Under a cap of 25% or 50% of the SMs, all 3,439 loss values the trainer logged differed from the uninterrupted dedicated run, from the first step, while all 3,439 equalled an uninterrupted run under the same cap. Capped runs were deterministic, but not the same computation as the dedicated run. Uncapped MPS,

the prefill shield and the governor left the results bit-identical: 0 of 6,085 loss values, 0 of 594 parameter bit-sums and 0 of 111 SHA-256 digests differed from the dedicated run. A harvester that must not change a job's results can choose when the job runs, but not cap its SMs.

An MPS client could not take a GPU memory snapshot. cuda-checkpoint [7] failed to lock the trainer as an MPS client, uncapped or capped at 25%, and a vLLM engine core as an MPS client, each with the error 'operation not supported'. The same calls on the same programs outside MPS locked, checkpointed with the process's GPU memory at 0 MiB, restored and unlocked, and the restored trainer's 300 loss values equalled the uninterrupted run. A newer driver gave the same results for the trainer. While MPS is on, neither process on the GPU can be moved by the snapshots that the scale-to-zero case above relies on, so as measured the two cases cannot share a GPU at the same time.

MPS worked across separate containers. With one MPS control daemon and its pipe directory bind-mounted, a client in a second container joined the same MPS server beside a client in the daemon's container, ran at the same time and saw a quarter of the GPU's SMs under its 25% cap, in all 7 configurations tested: 4 with Docker and 3 with containerd and runc, including containerd's default private IPC and PID namespaces. No host IPC or PID namespace was needed.

Table 9: Nodus results with work beside a serving engine, each scored by its own study's definition. The results are dedicated-GPU equivalents, the serving GPU's work plus the side work's share of a dedicated GPU, except priority backfill's useful GPU-hours per paid GPU-hour. Encoder sidecars' target is TTFT at most 500 ms and TPOT at most 30 ms for each request, and servetrain's is a TTFT p95 and TPOT p95 class per tenant, from 1 s and 50 ms to 10 s and 150 ms. servetrain's result is a mean over three seeds. This section's governor results are at 10%, 30% and 60% load, and the shield's at 10% and 30% load under encoder sidecars' limits applied at p95, which it missed at 60%.

Study	Side work	Latency target	Result
Encoder sidecars, report 021	Encoder batches	Absolute	1.5635
servetrain, internal	8 RL jobs	Absolute	1.268
Priority backfill, internal	One training job	Not in its record	1.3048 to 1.3102
Governor, this section	This trainer	1.2× serving alone	1.23, 1.10, 1.06
Prefill shield, this section	This trainer	Absolute, at p95	1.53, 1.45

Note on the priority-backfill row. It is an internal result. Its record is the frozen file on an unmerged draft branch of the older lab repository, at commit 5c74a185c1b, whose draft report was not deployed. That record keeps neither its SLO's values nor the raw data behind it, so this section cites its three values as recorded and could not recompute them.

Against earlier Nodus results. Earlier Nodus measurements of work beside serving reported more. Encoder sidecars and servetrain were scored against absolute latency targets, and the priority-backfill record does not give its target, while this section holds a budget relative to serving alone. Encoder sidecars (report 021 [19]) recovered 1.5635 dedicated-GPU equivalents at a live SLO attainment of 1.0000, a request meeting the SLO when its TTFT is at most 500 ms and its TPOT at most 30 ms. Its guarded arm's TTFT p95 was 0.0933 to 0.1007 s against 0.0499 to 0.0513 s for the server alone, 1.86 to 1.99 times as high on the same seed, inside that SLO but outside a 1.2× budget. The servetrain study, internal and not published, reached a mean of 1.268 with every tenant within its own TTFT p95 and TPOT p95 class, after its serving fleet was cut from 80 to 56 tenants. An internal priority-backfill record, also not published, gives 1.3048 to 1.3102 useful GPU-hours per paid GPU-hour at 100% serving-token SLO attainment, without the SLO's values or the raw data.

This section on the same scale. Counted the same way, with serving as one GPU plus the trainer's share, the governor here gave 1.23, 1.10 and 1.06. With absolute limits of encoder sidecars' values applied at p95 rather than to each request, the prefill shield kept 52.8% and 44.7% of a dedicated GPU at 10% and 30% load, 1.53 and 1.45, and missed at 60% with a TPOT p95 of 36.4 ms. Under limits like encoder sidecars', the result here comes close

to theirs, which points to the target as the main difference. The side work differs too: an encoder batch can be resized at will, and a training step cannot.

Scope. One GPU, one host, one small served model at half the GPU's memory and one 85 million parameter trainer, with three seeds per load and three windows behind each trainer interval. The freeze was SIGSTOP on a plain process, so the cgroup freezer a container would use was not measured, and neither was the governor without one-kernel launches. A trainer that needs less memory bandwidth, or a larger served model, changes how much a running trainer slows decode and so the harvest. The snapshot result covers the two drivers tried and the container result one Linux user, so MPS's one server per user, user namespaces and other container runtimes were not tested. No independent verifier has rerun these measurements.

13 Conclusion

A serving GPU's idle gaps can train. On one GPU serving a scale-to-zero model at 90% of GPU memory, a trainer stopped and restored by GPU memory snapshots kept 91% of a dedicated GPU's rate at 1 request per minute, 60% at 4 and 34% at 8, with serving p95 at 7.6 to 8.8 s and every one of 271 restores bit-exact. Each preemption cost a nearly fixed 9.5 s of batch time, so the share a node gets back follows from its preemption rate, and a node can gate filling on that rate. The price is 3.3 to 4.6 s of p95 over snapshot swaps alone, so gap-fill suits endpoints that already scale to zero, not hot ones. For a fleet, that turns the idle hours of sparse endpoints into batch work without another GPU. Whether it holds with larger trainers, several Functions per GPU and snapshots that leave the page cache is the next measurement. Beside a warm endpoint the room is much smaller: within 1.2× of serving alone at p95, the trainer kept 23.1% of a dedicated GPU at 10% load and 6.4% at 60%, bit-identical, because a request was almost always in flight, below earlier Nodus results that were scored against absolute SLOs.

References

[1] Nodus Research. GPU Memory Snapshots in Containers. Report 033, 2026.

[2] Nodus Research. Several Models per GPU by Swapping GPU Memory Snapshots. Report 034, 2026.

[3] W. Kwon et al. Efficient Memory Management for Large Language Model Serving with PagedAttention. SOSP 2023.

[4] Nodus Research. GPU Memory Snapshots as Reclaim Recovery for Training. Report 035, 2026.

[5] A. Paszke et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library. NeurIPS 2019.

[6] CRIU project. Checkpoint/Restore In Userspace. criu.org.

[7] NVIDIA. cuda-checkpoint: checkpoint and restore of CUDA process state. github.com/NVIDIA/cuda-checkpoint, 2024.

[8] Qwen Team. Qwen2.5 Technical Report. arXiv preprint arXiv:2412.15115, 2024.

[9] I. Loshchilov, F. Hutter. Decoupled Weight Decay Regularization. ICLR 2019.

[10] W. Xiao et al. Gandiva: Introspective Cluster Scheduling for Deep Learning. OSDI 2018.

[11] P. Yu and M. Chowdhury. Salus: Fine-Grained GPU Sharing Primitives for Deep Learning Applications. MLSys 2020.

[12] Z. Bai, Z. Zhang, Y. Zhu, X. Jin. PipeSwitch: Fast Pipelined Context Switching for Deep Learning Applications. OSDI 2020.

[13] W. Xiao et al. AntMan: Dynamic Scaling on GPU Clusters for Deep Learning. OSDI 2020.

[14] M. Han, H. Zhang, R. Chen, H. Chen. Microsecond-scale Preemption for Concurrent GPU-accelerated DNN Inferences. OSDI 2022.

[15] F. Strati, X. Ma and A. Klimovic. Orion: Interference-aware, Fine-grained GPU Sharing for ML Applications. EuroSys 2024.

[16] R. Stoyanov et al. CRIUgpu: Transparent Checkpointing of GPU-Accelerated Workloads. arXiv preprint arXiv:2502.16631, 2025.

[17] D. Shukla et al. Singularity: Planet-Scale, Preemptive and Elastic Scheduling of AI Workloads. arXiv preprint arXiv:2202.07848, 2022.

[18] NVIDIA. Multi-Process Service. docs.nvidia.com/deploy/mps.

[19] Nodus Research. Encoder Sidecars on LLM Serving GPUs. Report 021, 2026.