Several Models per GPU

Four models take turns on one GPU, each answering 6.84 s after its swap on average against 82.7 s cold, with identical outputs

Pranav Varshney
Nodus Compute

Viswa Kotra
Nodus Compute

Abstract

A GPU that is kept warm for one model only so that the model never cold-starts does no work while it waits. If several models take turns on one GPU instead, each restored from its GPU memory snapshot in seconds when its turn comes, the same answers need 1 GPU instead of one per model. Nodus, the infrastructure execution layer for AI workloads, measured this through the local GPU snapshot path that has shipped in the Nodus platform: the real node agent and its guarded runc executor, CRIU and NVIDIA cuda-checkpoint, with stock vLLM 0.30.0 and the weights and KV cache kept out of every snapshot. Each model is its own Function, one swap is one desired-state push that replaces the worker on the GPU, every answer is checked against the model's cold greedy answer, and the page cache is dropped before every start. On one GPU, two models took 6 turns over 3 rounds and every turn restored with the cold answer, 7.23 s on average from the push to the answer against 85.7 s cold. Four models from Qwen3-0.6B to Qwen2.5-3B took 12 turns. With GPU persistence mode on, all 12 answered with the cold answer, 11 from a snapshot, in 6.84 s on average and at most 8.22 s against 82.7 s cold, 12.1× faster, and the one GPU served the 12 turns in 233 s of wall time, where four warm dedicated GPUs would hold 933 GPU-seconds over the same time. With persistence mode off on another fresh host, the same test took 21.63 s per restored turn. The platform traced that to GPU queries on the node agent's start and restore path, each of which re-initialises an idle GPU without persistence mode for about 2 s, and has since shipped a fix, reported in the follow-up. The models are small, K is at most 4, the turns ran back to back rather than under request traffic, and no independent verifier has rerun the measurements. A dated follow-up found that seven models took 14 turns on one GPU in 6.28 s per turn on average against 73.8 s cold, after an eighth, Qwen3-4B, failed to start at the memory fraction the test gives every engine, and that on a larger GPU an 8B model took 49.63 s per turn on main, against 15.42 s for a 7B model with its weights kept out, because its take kept the weights inside after a GPU memory read failure that the agent caused itself. PR #699 of the Nodus platform, merged on 8 November 2026, fixes that read. In a rerun with it the 8B kept its weights out and swapped in 15.62 s, and in an earlier run of its first commit the turns without persistence mode fell from 21.63 s to 12.80 s. A follow-up served Poisson request traffic for 2, 4 and 6 models on one GPU: 862 restore swaps took 4.59 s at the median with none failed, and with p95 within 15 s one GPU served four models up to 8 requests per minute and six up to 6, at 457 and 396 requests per GPU-hour against 114 and 66 for warm dedicated GPUs, while the median latency rose from 0.51 s to about 5 s. A further follow-up kept two models

resident on one GPU, each at the memory fraction of 0.45 that left half the GPU's memory idle with one, and on the same arrivals and no more GPU-seconds it cut p95 by 29% to 38% under uniform traffic and doubled the highest offered rate within a p95 of 15 s, to at least 48 requests per minute for six models and 64 for four, while restoring the model a predictor expected next made no measurable difference. Two-resident placement has since shipped in the Nodus platform behind the per-organization Beta flag node.gpu_snapshot_two_resident, for an organization's own private Pools.

1 Introduction

A model that is kept warm on its own GPU only so that it never cold-starts holds that GPU while it waits for requests. Across a fleet of sparse models, most of those GPUs do no work most of the time. Our report 033 [1] showed that a containerized vLLM [2] engine comes back from a GPU memory snapshot in seconds, so a GPU held only to avoid a cold start can be released. This report asks the next question for more AI per GPU. Can several models take turns on one GPU, each restored from its own snapshot when its turn comes, so that K models need 1 GPU instead of K?

We measured it through the local GPU snapshot path that has shipped in the Nodus platform, with stock vLLM and up to four models on one GPU. Each turn is one swap: the platform replaces the model on the GPU with the next one, which restores from its snapshot and answers one greedy request, and the answer is compared with the same model's answer from a cold start. We report how long each turn took, whether it restored and answered correctly, what made the same test about three times slower on some hosts, and how many GPU-seconds the turns used against one warm GPU per model.

The metric is the first answer of a turn: seconds from the desired-state push that starts the swap to the incoming model's first answer, which includes stopping the outgoing model, restoring the snapshot from local disk and one greedy decode of 32 tokens. A cold start is timed the same way, from the push to the first answer of a worker that loads its model from the image. Every comparison is on the same host, GPU, image and models.

2 Experimental setup

Platform path. Every run used the local GPU snapshot path of the Nodus platform, slices 1 and 2 of its GPU snapshot design ADR-134, which have shipped and are in Beta: the real node agent and its guarded runc executor on containerd 2.4.1 and runc 1.5.2, with CRIU [3] and NVIDIA cuda-checkpoint [4], driven by the platform's own integration test of model swaps. Each worker runs stock vLLM 0.30.0 as an in-process vllm.LLM engine with sleep mode on, a maximum length of 2048 tokens and seed 0, and after its first answer it calls nodus.exclude_from_gpu_snapshot on the engine, so the weights and the KV cache stay out of the snapshot, as in report 033. Every snapshot sits in the node's cache on local disk with the model's weights file beside it.

One Function per model, one push per swap. Each model is its own Function, because a Function's worker prunes that Function's snapshots under other keys, so models sharing one Function would remove each other's snapshots. Each model first starts cold on the GPU, answers once and takes its snapshot. Then 3 rounds cycle through the models in a fixed order. One swap is one desired-state push that names only the next model's worker: the node stops the worker on the GPU, and the new worker is restored from its model's snapshot in the node's cache and answers one request. Nothing else ran on the GPU, and the turns followed each other back to back.

Checks. Every request is the same prompt, decoded greedily at temperature 0 for 32 tokens, and every answer is compared as text with the same model's answer from its cold start on that host. Each model's cold answer was also the same on every host where it started. The page cache is dropped before every cold start and every swap, so every restore reads its snapshot from local disk. The test records the GPU memory each model holds while

it serves, when the outgoing model's memory is released, and the snapshots in the node's cache.

Table 1: The four runs, each on its own fresh host with the same GPU type and driver. All four means the four models of the next table. Utilization is vLLM's GPU memory fraction per engine. Persistence mode is the GPU's, as each host's setup left it.

Run	Models	Utilization	Persistence mode	Turns
K=2	Qwen3-0.6B, Qwen2.5-1.5B	0.35	Not recorded	6
Run 1	All four	0.35	Off	12
K=4, persistence off	All four	0.45	Off	12
K=4, persistence on	All four	0.45	On	12

Table 2: The four models [5] [6]. The snapshot size includes the weights file. GPU memory is what each engine held while it served.

Model	Weights file, GB	Snapshot, GB	GPU MiB at 0.35	GPU MiB at 0.45
Qwen3-0.6B	1.45	5.38 to 5.39	7931	10191
Qwen2.5-1.5B-Instruct	3.41	7.49 to 7.51	7739	9999
Qwen3-1.7B	3.68	7.92 to 7.94	7581	9841
Qwen2.5-3B-Instruct	6.48	10.90 to 10.92	Did not start	9849

3 Results

Table 3: Every run as the test computed it. A turn's first answer is timed from the desired-state push to the incoming model's first answer. The ratio is the mean cold first answer over the mean restored one. The GPU time of each run is in the GPU time section below.

Run	Answered, restored, identical	Restored, mean s	Max s	Cold, mean s	Ratio
K=2	6, 6, 6 of 6	7.23	8.59	85.7	11.9×
Run 1, 3 of 4 models	9, 9, 9 of 12	23.28	24.79	95.9	4.1×
K=4, persistence off	12, 11, 12 of 12	21.63	22.67	92.2	4.3×
K=4, persistence on	12, 11, 12 of 12	6.84	8.22	82.7	12.1×

Two models on one GPU. All 6 turns restored and gave the cold answer, 7.23 s on average and at most 8.59 s from the push to the answer, against 85.7 s on average for the two cold starts, 11.9× faster. The outgoing model's GPU memory was back to 0 MiB 0.27 to 0.30 s after each push, and the restore took the remaining 5.71 to 8.04 s to ready. The first round was the slowest, 8.29 and 8.59 s, and the later rounds took 6.11 to 6.90 s. At these settings the two engines would also fit on the GPU together, so this run shows that swapping works, not that two models this small need it.

Run 1, three of four models. At utilization 0.35, Qwen2.5-3B failed to start in 4 of 4 attempts, its cold start and its 3 turns, at vLLM's engine-core startup, and the logs did not capture the root cause. The other three models took 9 turns, all restored with the cold answer, in 23.28 s on average and at most 24.79 s against 95.9 s cold, 4.1× faster. The test's GPU-time figure for this run counts four warm GPUs although only three models served, so this report does not use it. This host ran with persistence mode off, and its turns were as slow as those of the next run.

Four models, persistence mode off. At utilization 0.45 on another fresh host, all 12 turns answered with the cold answer and 11 restored, in 21.63 s on average and at most 22.67 s against 92.2 s cold, 4.3× faster. The node declined Qwen3-1.7B's first snapshot with reason MEMORY_LIMIT, so that model's first turn started cold, answered in 101.20 s and took its snapshot then, and its 2 later turns restored. The outgoing model was released 0.27 to 0.28 s after each push, and the restores took 20.51 to 21.91 s after that, about three times as long as in the K=2 run. The local disk read 3.0 GB/s on this host with the page cache dropped.

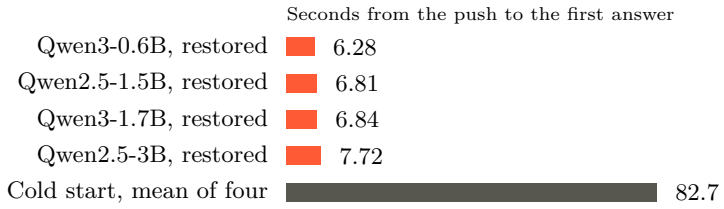


Figure 1: Seconds from the desired-state push to the first answer in the K=4 run with persistence mode on, mean of each model's restored turns against the mean cold start, lower is better.

Table 4: K=2, two models at utilization 0.35, 3 rounds, 6 turns. Per model, the cold start's first answer and the restored turns' first answers.

Model	Cold first answer, s	Turns restored	Restored, mean s	Max s
Qwen3-0.6B	89.7	3 of 3	7.06	8.29
Qwen2.5-1.5B	81.8	3 of 3	7.40	8.59

Four models, persistence mode on. On a host whose setup enabled GPU persistence mode, the same test with the same models and settings answered all 12 turns with the cold answer, 11 from a snapshot, in 6.84 s on average and at most 8.22 s against 82.7 s cold, 12.1× faster. This time the node declined Qwen2.5-3B's first snapshot with reason MEMORY_LIMIT, so its first turn started cold and answered in 79.49 s, and its 2 later turns restored in 8.22 and 7.22 s. The restores took 5.83 to 7.45 s after the release, and the largest model was the slowest, 7.72 s on average for Qwen2.5-3B against 6.28 s for Qwen3-0.6B. At the end the node held 31.8 GB of snapshots on local disk for the four models, and no snapshot was removed from its cache in any run. The MEMORY_LIMIT declines, one in each four-model run at 0.45 and a different model each time, are not diagnosed here.

4 What GPU persistence mode changed

The same test, image, models and settings took 21.63 s per restored turn with persistence mode off and 6.84 s with it on, 3.2× faster, and the restore after the outgoing model's release fell from 21.07 s to 6.29 s on average, a difference of 14.78 s per turn. The cold starts differed by less, 92.2 s against 82.7 s, on different hosts.

Without persistence mode, the NVIDIA driver tears a GPU's state down when the last program using it exits and sets it up again when the next one opens it [7]. In a swap the GPU is idle between the outgoing model's release and the restore, at 0 MiB in every turn. The platform's diagnosis is that the node agent queries the GPU with nvidia-smi on its start path and on its restore path, when it plans a snapshot and when it reads a restore's GPU information, and that on a GPU without persistence mode each of these calls re-initialises the idle device for about 2 s. With persistence mode on, nvidia-smi answered in 0.01 s on the persistence-on host.

The test's own GPU memory readings show the same signature. When a step began with no engine on the GPU, its first reading at idle came 1.69 to 1.84 s after the push on the two persistence-off hosts, against 0.028 s on the persistence-on host. When a step began with an engine serving, the release was seen 0.26 to 0.30 s after the push on every host. The K=2 host did not record its persistence setting. Its restores were about as fast as those with persistence mode on and its first reading at idle came after 0.027 s, which is consistent with a GPU that stayed initialised, but the inputs do not show it.

Two cautions apply. The persistence-off and persistence-on runs used different fresh hosts, and the persistence-on run used a different revision of the platform's main branch, while the persistence-off runs did not record theirs. The K=2 run, on the revision it recorded, restored about as fast as the persistence-on run, which argues against the revision as the cause, but this report does not exclude it. The platform fixed the cost in two places, by

Table 5: Run 1, the four models at utilization 0.35, 3 rounds. Qwen2.5-3B did not start at this utilization, so 3 models took turns.

Model	Cold first answer, s	Turns restored	Restored, mean s	Max s
Qwen3-0.6B	101.5	3 of 3	23.21	23.33
Qwen2.5-1.5B	83.7	3 of 3	23.96	24.79
Qwen3-1.7B	102.5	3 of 3	22.69	22.77
Qwen2.5-3B	Did not start	0 of 3	None	None

Table 6: K=4 with persistence mode off, the four models at utilization 0.45, 3 rounds, 12 turns.

Model	Cold first answer, s	Turns restored	Restored, mean s	Max s
Qwen3-0.6B	98.8	3 of 3	20.95	20.99
Qwen2.5-1.5B	80.6	3 of 3	21.57	21.66
Qwen3-1.7B	98.7	2 of 3	21.63	21.79
Qwen2.5-3B	90.6	3 of 3	22.38	22.67

taking the GPU inventory query off the restore path and by enabling persistence mode when a node is bootstrapped, in a change that shipped on 8 November 2026 and is reported in Section 8. The 6.84 s here comes from persistence mode enabled by hand in the test host's setup, not from that change, and the change has not been measured on production nodes.

5 GPU time

The test computes GPU time one way, and this report states it as computed. The wall time runs from the start of the first swap to the last answer, and the GPU-seconds used are that wall time on 1 GPU. The comparison is K warm dedicated GPUs, one per model, held over the same wall time, so its ratio is K by construction. What the measurement adds is that 1 GPU did serve every turn inside that wall time, at the turn times above.

Reading. With persistence mode on, one GPU served 12 turns of four models in 233 s, where four warm dedicated GPUs would have held 933 GPU-seconds over the same time, and where serving the same turns by cold starts on one GPU would have taken 992.5 s. The comparison is not free. A warm dedicated GPU answers in its decode time, 0.12 to 0.50 s for these models, while a model waiting for its turn answers 6.23 to 8.22 s after its swap begins, and a request that arrives for a model not on the GPU waits for the swap. The wall times include each four-model run's one cold turn and the snapshot it took. They exclude the first cold start and snapshot of every model, which happen once per model before any swap: 72.7 to 102.5 s to the first answer, then 42 to 75 s for the snapshot while the model kept serving.

6 Scope

Every run used one GPU type and small models from 0.6B to 3B, whose snapshots are 5.38 to 10.92 GB with the weights file. K was at most 4 and each run had 3 rounds, so each restored mean rests on 6 to 11 turns. The turns ran back to back, each started by the test rather than by requests, so queueing and latency under load are not measured. A request that arrives for a model not currently on the GPU waits for the swap, and this report does not design the scheduler that decides which model holds the GPU. The two engines of the K=2 run would fit on the GPU together, and the four engines at 0.45 would not, 39880 MiB together, more than the GPU's memory, but they could share the GPU at smaller memory fractions, and this report does not compare swapping with co-residency, which matched snapshot handoffs on density in report 033. The persistence-off and persistence-on runs used different fresh hosts, the disk was measured on one host only, and the MEMORY_LIMIT declines are not diagnosed. The numbers come from the platform's own integration test, and no independent verifier has rerun them. The honest next steps are larger models, which

Table 7: K=4 with persistence mode on, the same four models and settings on another fresh host, 3 rounds, 12 turns.

Model	Cold first answer, s	Turns restored	Restored, mean s	Max s
Qwen3-0.6B	88.6	3 of 3	6.28	6.34
Qwen2.5-1.5B	72.7	3 of 3	6.81	7.07
Qwen3-1.7B	90.8	3 of 3	6.84	7.23
Qwen2.5-3B	78.7	2 of 3	7.72	8.22

Table 8: GPU time as the test computed it: wall time from the first swap to the last answer on 1 GPU, against K warm dedicated GPUs over the same wall time. The last column is the wall time 1 GPU would need if every turn were a cold start, the number of turns times the mean cold first answer. Run 1 is left out because only 3 of its 4 models served.

Run	Turns	Wall on 1 GPU, s	K warm GPUs, GPU-s	Every turn cold, s
K=2	6	50	100	514.4
K=4, persistence off	12	426	1703	1106.1
K=4, persistence on	12	233	933	992.5

cannot share a GPU by co-residency and whose restores read more from disk, real request traffic with its queueing, and a scheduler that decides when to swap.

7 Related work

Switching one GPU between models is an established serving technique. Clockwork [8] loads models from host memory into GPU memory on demand and schedules every load and inference centrally for predictable latency. PipeSwitch [9] pipelines a model's transfer with its execution so that a GPU switches between applications in milliseconds. AlpaServe [10] uses model parallelism to multiplex several models statistically over a group of GPUs, MuxServe [11] multiplexes several LLMs on shared GPUs in space and time, and Aegaeon [12] pools many models per GPU with switching at token granularity, a far stronger result than four small models per GPU with a swap of seconds. ServerlessLLM [13] speeds up loading the weights rather than restoring a process.

Restoring a whole GPU process builds on CRIU and cuda-checkpoint, as in CRIUgpu [14], a serverless GPU platform's memory snapshots [15] and NVIDIA Dynamo Snapshot [16]. Closest to this report, llmux [17] multiplexes vLLM models on one GPU with vLLM's sleep mode and CRIU checkpoints, SwapServeLLM [18] suspends and resumes containerized engines with the driver's checkpoint and restore, Torpor [19] keeps idle inference models in host RAM and swaps them onto GPUs on request, and NVIDIA Run:ai GPU memory swap [20] moves idle inference workloads to CPU memory in a product. Our report 033 measured the restore itself in the Nodus stack. What this report adds is the measurement of several models taking turns through the platform's shipped snapshot path, and the cost of an idle GPU without persistence mode on that path.

8 Follow-up, 8 November 2026: eight models, and 7–8B models

Question and setup. The runs of Section 3 put at most four models of 0.6B to 3B on one GPU. Two questions remained: whether more models can take turns on one GPU, and whether the swap holds for 7B and 8B models, whose snapshots are larger. Nodus ran the test of Section 2 on the same platform revision as the persistence-on run of Section 3, with GPU persistence mode enabled in each host's setup, once on each of two fresh hosts. On one GPU of the same type as before, eight models from Qwen2.5-0.5B-Instruct to Qwen3-4B took turns at utilization 0.45 for 2 rounds, 16 turns. On one larger GPU, Qwen3-8B, Qwen2.5-7B-Instruct and Qwen3-4B took turns at utilization 0.85 for 3 rounds, 9 turns. Two more runs measured PR #699 of the Nodus platform, a change that was merged on 8 November 2026, each on its own fresh host: the four-model test of Section 3 on the

same GPU type with persistence mode off, and the 4B to 8B run repeated with persistence mode on. As before, every engine gets the same fraction of GPU memory, every answer is compared with the model's cold greedy answer, and the page cache is dropped before every start.

Table 9: Eight models on one GPU at utilization 0.45 with persistence mode on, 2 rounds. Per model, the snapshot with its weights file included, the cold start's first answer (cold) and the restored turns' first answers, each timed from the desired-state push. The Qwen2.5 models are the Instruct models. Qwen3-4B did not start at this utilization.

Model	Snapshot, GB	Cold, s	Turns restored	Restored, mean s	Max s
Qwen3-0.6B	5.38	85.2	2 of 2	5.64	5.67
Qwen2.5-0.5B	5.02	64.1	2 of 2	5.40	5.41
Qwen2.5-1.5B	7.50	68.7	2 of 2	6.26	6.32
Qwen3-1.7B	7.93	85.7	2 of 2	6.63	6.85
Qwen2.5-3B	10.89	76.1	2 of 2	7.54	7.78
Qwen3-4B	None	Did not start	0 of 2	None	None
Qwen2.5-Coder-1.5B	7.49	68.3	2 of 2	6.31	6.33
Qwen2.5-Math-1.5B	7.50	68.4	2 of 2	6.17	6.24
Seven that started		73.8	14 of 14	6.28	7.78

Seven of the eight models took turns on one GPU. Qwen3-4B failed to start in 3 of 3 attempts, its cold start and both of its turns, at vLLM's engine-core startup and before any snapshot, and its output was not captured, so the logs do not show the root cause. The other seven models, from Qwen2.5-0.5B-Instruct to Qwen2.5-3B-Instruct with a coder and a math model of 1.5B, took 14 turns, and all 14 restored from a snapshot and gave the cold answer, in 6.28 s on average and at most 7.78 s against 73.8 s cold, 11.7× faster. Where a turn replaced a serving model, the outgoing model was released 0.27 s after the push, and the restores took 5.00 to 7.02 s after the release. Per model the turns were no slower than with four models in Section 3, 5.64 s against 6.28 s for Qwen3-0.6B and 7.54 s against 7.72 s for Qwen2.5-3B-Instruct, on different hosts. At the end the node held 51.8 GB of snapshots on local disk for the seven models, and none was removed. The seven engines held 70109 MiB together while serving, about 3 times the GPU's memory, so at this utilization they could not all be resident at once.

Two startup failures are sizing limits of the test. The test gives every engine the same fraction of GPU memory. At 0.35 on the smaller GPU, Qwen2.5-3B-Instruct failed to start in run 1 of Section 3 and started at 0.45 in every other run that included it, and at 0.45 on the smaller GPU, Qwen3-4B failed to start here and started at 0.85 on the larger GPU. Both failed at vLLM's engine startup before any snapshot existed, so they are limits of how the test sizes its engines, not results about snapshots.

Table 10: Three models of 4B to 8B on one larger GPU at utilization 0.85 with persistence mode on, 3 rounds, 9 turns, all restored, on main. The snapshot includes the weights file. Qwen3-8B's take kept no weights file, so its weights are inside its snapshot. Cold is the cold start's first answer.

Model	Weights file, GB	Snapshot, GB	Cold, s	Restored, mean s	Max s
Qwen3-8B	None	38.14	132.1	49.63	49.97
Qwen2.5-7B	15.52	20.51	100.8	15.42	15.56
Qwen3-4B	8.29	13.15	122.8	13.76	14.03
All three			118.6	26.27	49.97

7B and 8B models on one larger GPU. All 9 turns restored from a snapshot and gave the cold answer. Qwen2.5-7B-Instruct, whose 20.51 GB snapshot keeps its weights out in a 15.52 GB weights file, answered 15.42 s after its push on average against 100.8 s cold, 6.5× faster, and Qwen3-4B answered in 13.76 s against 122.8 s, 8.9× faster. Qwen3-8B answered in 49.63 s against 132.1 s, 2.7× faster. Over all 9 turns the mean was 26.27 s against 118.6 s cold, 4.5×. The outgoing model was released 0.52 to 0.53 s after each push, against 0.27 s on the smaller GPU.

On main, the 8B take kept its weights inside. For Qwen3-8B the agent recorded no weights file. Its snapshot was 38.14 GB with the weights inside, more than the 35.21 GB the engine held on the GPU while serving, where every other take in these runs on main kept the weights out in a weights file. Every 8B turn restored from that snapshot with the cold answer, and Qwen2.5-7B-Instruct with its weights kept out swapped 3.2× faster than Qwen3-8B with its weights inside, 15.42 s against 49.63 s on the same GPU. The platform traced the fallback, from the code and the logs, to a GPU memory read failure that the agent caused itself. Before the take, the agent reads GPU memory through nvidia-smi to decide whether the weights can be kept out. On main that read fails at once whenever another nvidia-smi call of the agent holds its lock, such as a heartbeat or the startup inventory read, the agent answers the failed read as a MEMORY_LIMIT decline, and the worker then keeps the weights inside its snapshot without logging it. On main, the agent reported GPU usage reads failing with device maintenance in progress for 12 of the 12 attempts of this run and in both four-model runs on main in Section 3.

Table 11: The 4B to 8B run repeated with an agent built from the final commit of PR #699, merged on 8 November 2026, on another fresh GPU of the same type: the same models, image and settings, persistence mode on, 3 rounds, 9 turns, all restored. Every take kept a weights file, and the snapshot includes it. The last column is the run on main above.

Model	Weights file, GB	Snapshot, GB	Cold, s	Restored, mean s	On main, mean s
Qwen3-8B	16.57	21.87	131.0	15.62	49.63
Qwen2.5-7B	15.52	20.49	100.5	14.97	15.42
Qwen3-4B	8.29	13.15	122.4	13.10	13.76
All three			118.0	14.56	26.27

A rerun with the merged fix confirms the diagnosis. PR #699 lets the agent's GPU inventory and usage queries run together, and it makes the weights-out probe wait up to 2 s for a busy GPU memory read and, if the read still fails, leave the decision to the take instead of declining. With an agent built from its final commit, b8662cabb, every take of the 4B to 8B run kept the weights out. Qwen3-8B's snapshot was 21.87 GB with its weights kept out in a 16.57 GB weights file, against 38.14 GB with the weights inside on main, and the agent reported no failed GPU usage read, in 0 of 12 attempts. All 9 turns restored and gave the cold answer. The 8B answered 15.62 s after its push on average and at most 15.68 s, against 49.63 s on main, 3.2× faster, and 0.64 s more than Qwen2.5-7B-Instruct. Over all 9 turns the mean was 14.56 s against 118.0 s cold, 8.1× faster, where main was 4.5×. Qwen2.5-7B-Instruct and Qwen3-4B, whose takes kept their weights out on main as well, answered in 14.97 s and 13.10 s, 0.45 s and 0.66 s less than on main, on a different host. This is one run on one GPU.

Table 12: Four models on one GPU at utilization 0.45, 3 rounds, 12 turns, each run on its own fresh host. The first and last columns are the K=4 runs of Section 3 on main. The middle column is an agent built from the first commit of PR #699 of the Nodus platform, cb362c733, before the review fixes that the merged PR adds. Off and on are GPU persistence mode. The last row is when the first GPU memory reading at idle came after the push in the first cold start, which began with no engine on the GPU.

Measure	Main, off	PR #699, off	Main, on
Turns answered, restored, identical	12, 11, 12	12, 12, 12	12, 11, 12
Takes declined at the cold starts	1	0	1
Restored first answer, mean s	21.63	12.80	6.84
Restored first answer, max s	22.67	13.67	8.22
Restore after release, mean s	21.07	12.24	6.29
Cold first answer, mean s	92.2	83.3	82.7
Ratio	4.3×	6.5×	12.1×
First GPU reading at idle, s	1.69	1.65	0.028

The platform change has shipped. Section 4 gave the platform's diagnosis of the 21.6 s turns without persistence mode, GPU queries on the node agent's start and restore path,

each of which re-initialises an idle GPU, and the platform's fix. That fix shipped in PR #699 of the Nodus platform, merged on 8 November 2026, which takes the GPU inventory reads off the snapshot plan and restore path, reports an unreadable GPU memory as MEMORY_UNREADABLE instead of MEMORY_LIMIT, makes the weights-out probe wait for a busy read as described above, and turns persistence mode on in the node bootstrap of a GPU host. Nodus ran the four-model test of Section 3 with an agent built from the PR's first commit, cb362c733, on another fresh host of the same GPU type, with persistence mode off. All 12 turns restored and gave the cold answer, in 12.80 s on average and at most 13.67 s against 83.3 s cold, 6.5× faster. That is 8.83 s less than main without persistence mode and 5.96 s more than main with it, so the change removed 8.83 s of the 14.80 s difference and left the rest. Every cold take was taken, where each four-model run on main had one declined with MEMORY_LIMIT, which the platform traces to the same failed GPU memory read. nvidia-smi still took 1.65 to 1.69 s per query on this host, and the first GPU memory reading at idle came 1.65 s after the push. This is one run of the PR's first commit, before the review fixes that the merged PR adds, and PR #699 with persistence mode on was measured only in the 4B to 8B rerun above, not in the four-model test.

Table 13: GPU time as the test computes it, for the follow-up runs: wall time from the first swap to the last answer on 1 GPU, against one warm dedicated GPU per model over the same wall time. The test counts all eight listed models in the eight-model run, so that comparison is recomputed here for the 7 that served. The last column is the turns of the models that served times their mean cold first answer.

Run	Turns	Wall on 1 GPU, s	Warm GPUs, GPU-s	Every turn cold, s
Seven of eight	14	279	1955 for 7	1032.8
4B to 8B	9	293	880 for 3	1067.2
4B to 8B, PR #699	9	179	536 for 3	1061.7
PR #699, four models	12	165	660 for 4	999.9

GPU time. One GPU served the 14 turns of seven models in 279 s of wall time, where seven warm dedicated GPUs would hold 1955 GPU-seconds over the same time and serving the same turns by cold starts on one GPU would take 1032.8 s. That wall time includes the two failed Qwen3-4B turns, 176.7 s together from each push to the agent's last line for that turn, twice the 87.9 s from push to answer of the 14 turns that served. The test's own figure for this run, 2234 GPU-seconds for eight warm GPUs, counts Qwen3-4B, which never served, and this report does not use it. One larger GPU served the 9 turns of three 4B to 8B models in 293 s, where three warm GPUs would hold 880 GPU-seconds. With PR #699 the same test took 179 s, where three warm GPUs would hold 536 GPU-seconds, mostly because the 8B turns were shorter. As in Section 5, the ratio is K by construction, and a model waiting for its turn answers seconds after its swap begins, not in its decode time.

Scope. These are two GPU types, a smaller and a larger one, with one run each on main, one run of the first commit of PR #699 on a third fresh host, with the smaller GPU, and one run of its final commit on a fourth, with the larger. Each model's restored mean rests on 2 turns in the eight-model run and 3 in the others. The turns ran back to back, started by the test, with no request traffic, and no scheduler decided which model held the GPU. The rerun that confirms the diagnosis of the 8B's weights-in fallback, and its 15.62 s with the weights kept out, is one run on one GPU. PR #699 also enables persistence mode in the node bootstrap, but every host here had persistence mode set on or off by its own test setup, and no gain on production nodes is reported. The disk was not measured on these hosts, and co-residency was not compared. The numbers come from the platform's own integration test, and no independent verifier has rerun them.

9 Follow-up, 8 November 2026: several models under request traffic

Question and setup. The turns above ran back to back, started by the test, with no request traffic and no scheduler. In a follow-up on one GPU with GPU persistence mode on, Nodus served seeded Poisson request traffic for 2, 4 and 6 models of 0.5B to 3B through the same local snapshot path, the real node agent and its guarded runc executor, from a later research

commit of the platform. A scheduler in the test process kept one model resident and a queue per model. It sent the resident model's whole queue to it as one batch, and when that model had nothing queued or in flight it swapped to the model whose oldest request had waited longest, by one desired-state push that restored the incoming model from its snapshot. Each request was one of 8 short prompts, decoded greedily to 64 tokens without streaming, and its latency runs from its arrival to the last token of its answer. Two baselines used the same GPU and traffic generator: each model alone and warm, which is what a model on its own GPU sees, and the same scheduler with every swap a cold start.

Table 14: Restore swaps under traffic, by model, from the push that replaces the resident worker to the incoming model serving. The Qwen2.5 models are the Instruct models. 862 restores in all, none failed and none fell back to a cold start.

Model	Restores	p50, s	p95, s
Qwen2.5-0.5B	67	4.36	4.60
Qwen3-0.6B	230	4.47	4.59
Qwen2.5-1.5B	214	4.59	4.68
Qwen2.5-Coder-1.5B	62	4.60	4.77
Qwen3-1.7B	131	4.72	4.86
Qwen2.5-3B	158	4.86	5.03

Restores held up. All 862 restore swaps under traffic succeeded, with no failure and no cold fallback, and took 4.59 s at the median, 5.03 s at p99 and at most 5.66 s, barely depending on model size. A cold swap took 64.2 to 79.9 s, and the median cold swap, 64.5 s, is 14.0 times the median restore. The 717 answers decoded alone, in a batch of one, had exactly one text per model and prompt across all restores, and it matched the 3 such answers from cold-loaded workers. Answers decoded in larger batches can differ, as vLLM's batch kernels do. The 4.59 s is not directly comparable with the 6.84 s mean of the turns above: here the page cache stayed warm, and the time runs to the incoming model serving rather than to its first answer.

Table 15: Every run of the follow-up. Rate is the total in requests per minute, over the K models or, for the warm arm, per model. Latency runs from a request's arrival to the last token of its answer. Waited is the share of requests that arrived while their model was not on the GPU, swap-busy the share of the wall time spent swapping. Per GPU-hour gives the run's requests per GPU-hour on its one GPU, then on K warm dedicated GPUs held over the same wall time.

K	Rate	Requests	p50, s	p95, s	Waited	Swap-busy	Per GPU-hour
6	3	114	5.15	9.25	95%	19%	171 / 29
6	6	133	5.34	11.31	92%	44%	396 / 66
6	12	132	6.01	16.23	93%	62%	660 / 110
6	24	191	13.98	24.09	99%	89%	1417 / 236
4	4	98	5.12	10.33	76%	18%	196 / 49
4	8	115	5.26	11.47	79%	41%	457 / 114
4	16	144	8.41	17.25	97%	79%	1071 / 268
4	32	212	10.36	19.92	96%	86%	2071 / 518
2	4	96	4.72	6.92	71%	20%	288 / 144
2	16	136	4.74	8.71	84%	57%	1020 / 510
2	60	245	4.39	9.20	93%	87%	3599 / 1799
2, cold	2	29	64.92	128.46	93%	89%	129 / 64
1, warm	12	157	0.51	0.99	0%	0%	Not compared

One GPU for several models, at a latency cost. With p95 within 15 s, one GPU served two models at every load offered, up to 60 requests per minute in total, four models up to 8 per minute and six models up to 6 per minute. At those points the one GPU served 457 requests per GPU-hour with four models and 396 with six, where four and six warm dedicated GPUs over the same time would serve 114 and 66. The cost is latency. A warm dedicated GPU answered at a median of 0.51 s and p95 0.99 s, while with swapping the median rose to about 5 s, because 71% to 99% of requests arrived while another model held

Table 16: The highest measured total rate at which each K kept p95 within 15 s, and the first measured rate over it. The true boundary lies between the measured points.

K	Within 15 s	p95, s	First over	p95, s	GPU time
2	60 req/min, highest offered	9.2	None offered	None	1/2 of 2 GPUs
4	8 req/min	11.5	16 req/min	17.2	1/4 of 4 GPUs
6	6 req/min	11.3	12 req/min	16.2	1/6 of 6 GPUs

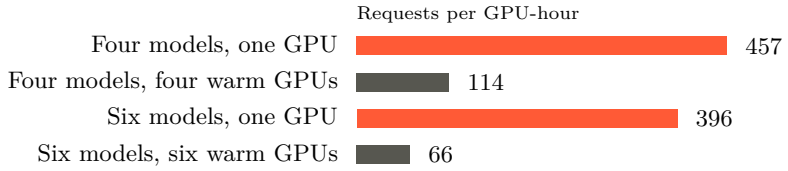


Figure 2: Requests per GPU-hour at the highest measured rate within a p95 of 15 s, one GPU shared by swapping against warm dedicated GPUs, higher is better.

the GPU, and one resident model at a time gives few hits. Cold swaps on the same scheduler were unusable even at 2 requests per minute, p50 64.92 s and p95 128.46 s, 18.6 times the p95 of snapshot swaps for two models at twice that load.

Two regimes. Below saturation most requests pay about one swap, and for four and six models p95 stayed within 15 s while the GPU spent up to 44% of its time swapping and broke it from 62%. At saturation, per-model batching bounds a request's wait to about one cycle of K swaps: at the highest loads p95 was 9.2 s for two models, 19.9 s for four and 24.1 s for six, against K times the run's median swap of 9.0, 18.6 and 27.7 s. Batches grew to 9 requests and decoding took at most 12% of the GPU's time, so the GPU never ran short of decode capacity, and swapping set both regimes. Every one of the 1802 requests was answered, the slowest under snapshot swaps after 30.13 s.

What we recommend. We recommend, and it was not built when this follow-up ran, that a scheduler admit swapped models onto a GPU by a swap budget rather than by memory alone. It has since been merged into the Nodus platform behind a Beta flag, as Section 10 reports. Up to the SLO divided by the restore p95 models can share a GPU at any load, 3 on this GPU at 15 s with 4.91 s restores, because per-model batching bounds their tail by one cycle. More models fit only while the GPU's swap-busy share stays at or below about 40%. The node can compute both numbers from the RESTORED events it already reports, and the scheduling tested here, per-model queues with the resident model's whole queue in one batch and the longest-waiting model next, is the one to keep on the GPU.

Scope of the follow-up. This is one GPU, one host and one seed, with 96 to 245 requests per point, so p99 is noisy. The models are small, 0.5B to 3B, answers were 64 tokens without streaming, and latency is to the last token. All six snapshots, 44.3 GB, stayed in the node's cache and largely in the page cache with persistence mode on, which is a best case: larger models, more models than memory holds or cold caches raise the swap time and shrink both rules. One model held the GPU at a time. The scheduler is research code in the test process driving the real node agent, the product has no swap scheduler yet, and requests reached the worker by a file in its state directory rather than the product gateway. The cold baseline is one point, and no independent verifier has rerun these measurements.

10 Follow-up, 9 November 2026: two resident models, and predicting the next model

Question and setup. In the previous follow-up one model was resident at a time, and each engine ran at a GPU memory fraction of 0.45, so the one resident engine left about half of the GPU's memory idle. This follow-up asks whether that memory can serve more requests on the same GPU, and whether restoring the model a predictor expects next helps. On one GPU with GPU persistence mode on, through the same local snapshot path and a

later main of the Nodus platform that includes PR #736, which cut the fixed overhead of GPU snapshot takes and restores, Nodus served four and six vLLM models of 0.5B to 3B, each at a memory fraction of 0.45. One engine alone held 9841 to 10231 MiB, under half the GPU's memory, so two fit side by side. Four arms ran under the previous follow-up's scheduler, with a queue per model, a resident model's whole queue sent as one batch and the longest-waiting model swapped in next: the base, with one resident model as before, (a) two resident models, replacing the least recently used one, (b) one resident model plus predictive pre-restore, and (a)+(b), both. The predictor scores each model by its arrivals, each weighted by exp(-age / 5 min) over a 10-minute history. When the GPU is idle it restores the best-scored model that is not resident, a guess, into a free slot, or in place of the lowest-scored resident model when the newcomer's score is at least 1.5 times that model's and at least 1 higher.

Paired traffic. Requests arrive by seeded Poisson processes per model, with seed 1, and each is one of 8 short prompts decoded greedily to 64 tokens without streaming, and latency runs from a request's arrival to the last token of its answer. Uniform traces split the total rate evenly, 6, 12, 24 and 48 requests per minute for six models and 8, 16, 32 and 64 for four, the previous follow-up's rates plus one higher rate each. Zipf traces with s = 1 give model i a share proportional to 1/(i+1), 48, 24, 16 and 12% for four models and 41% down to 7% for six, at 6 and 12 requests per minute for six models and 8 and 16 for four. A trace is generated from the seed and its name, so every arm of a trace served exactly the same arrivals, models and prompts, and each comparison below is paired per request, with 95% intervals from 2000 paired bootstrap resamples. Each point is one trace of 66 to 143 requests over 2 to 12 minutes, and three test processes on the one host ran the 50 runs.

Table 17: p95 latency of the base, one resident model, against (a), two resident models, on the same arrivals. Rate is the total in requests per minute. The change is (a) minus the base, with its 95% paired bootstrap interval in brackets.

Traffic	K	Rate	Base p95, s	(a) p95, s	Change, s
Uniform	6	6	11.12	7.04	-4.08 [-6.00, -0.75]
Uniform	6	12	10.10	7.21	-2.88 [-5.63, -1.29]
Uniform	6	24	13.72	9.26	-4.46 [-4.71, -1.75]
Uniform	6	48	20.45	13.88	-6.57 [-9.88, -5.63]
Uniform	4	8	8.81	5.85	-2.96 [-4.52, -1.76]
Uniform	4	16	10.93	7.37	-3.55 [-4.36, -1.14]
Uniform	4	32	13.09	8.46	-4.63 [-5.48, -2.21]
Uniform	4	64	15.36	9.59	-5.76 [-7.80, -4.14]
Zipf	6	6	7.71	5.45	-2.26 [-3.10, -0.82]
Zipf	6	12	11.03	8.72	-2.31 [-5.91, -1.39]
Zipf	4	8	8.67	5.61	-3.05 [-7.24, -1.59]
Zipf	4	16	10.73	6.01	-4.71 [-6.24, -1.48]

A second resident model cuts the tail. Against the base on the same arrivals, (a) cut p95 by 29% to 38% under uniform traffic, 2.9 to 6.6 s, and by 21% to 44% under Zipf traffic, 2.26 to 4.71 s, and all 12 intervals lie below zero. The share of requests that waited for their model fell by 10 to 34 points under uniform traffic and by 30 to 42 points under Zipf, where the median for four models fell to 0.57 s and 0.98 s. (a) also needed fewer restores, 378 against 420 over the uniform traces, 10% fewer, and 134 against 209 under Zipf, 36% fewer, because the second slot keeps the model used before the latest one. Restores beside a serving engine took 3.46 s at the median against 3.58 s alone, so two engines on one GPU did not slow restores, and the other engine kept serving through each swap. At the highest rates, p95 fell from 20.45 s to 13.88 s for six models at 48 requests per minute and from 15.36 s to 9.59 s for four at 64.

Twice the rate within the SLO, on no more GPU-seconds. With p95 within 15 s, the highest offered rate doubled, from 24 to at least 48 requests per minute for six models and from 32 to at least 64 for four. Because (a) stayed within 15 s at the highest rates offered, these are lower bounds. At those points the one GPU served 2624 and 3062 requests per GPU-hour with two resident models, against 1248 and 1598 with one, 2.1 and 1.9 times as many, since each trace's realized Poisson arrivals and the time to answer its last requests differ. (a) used

Table 18: The highest offered total rate at which the base and (a) kept p95 within 15 s under uniform traffic, and the first rate over it. The rates double from one point to the next, so each boundary lies between measured points, and (a) was within 15 s at the highest rate offered.

K	Arm	Within 15 s	p95, s	First over	p95, s
6	Base	24 req/min	13.72	48 req/min	20.45
6	(a)	At least 48 req/min	13.88	None offered	None
4	Base	32 req/min	13.09	64 req/min	15.36
4	(a)	At least 64 req/min	9.59	None offered	None



Figure 3: Requests per GPU-hour on one GPU at the highest offered rate within a p95 of 15 s, with two resident models against one, higher is better.

no more GPU-seconds than the base on any of the 12 traces. Each arm held the one GPU for the trace's duration plus the time to answer its last requests, 2792 GPU-seconds for (a) against 2809 for the base over the uniform traces and 2040 against 2044 under Zipf, and the arms of a trace differed by at most 4%. What (a) costs is GPU memory. It kept 1.49 engines resident on average under uniform traffic and 1.74 under Zipf, 15.5 and 17.6 GiB of the GPU's memory in use on average, against 0.45 and 0.62 engines and 5.3 and 6.8 GiB for the base. That is memory the one-resident layout at 0.45 left idle, but while (a) holds it no other workload can use it. The previous follow-up's highest rates within 15 s, 6 requests per minute for six models and 8 for four, came from other traces on an older main with slower restores. They are not a paired comparison with the rates here, and this report does not attribute the difference to one change. The paired result is (a) against the base on the same traces.

Table 19: (b), one resident model with predictive pre-restore, minus the base on the same arrivals: the change in the median and in p95, with 95% paired bootstrap intervals in brackets. The last column counts (b)'s guesses that served at least one request and those that served none.

Traffic	K	Rate	Median change, s	p95 change, s	Used / wasted
Uniform	6	6	-0.02 [-0.13, +0.13]	-0.11 [-0.19, +2.01]	3 / 8
Uniform	6	12	+0.18 [-0.15, +0.69]	+0.35 [-1.93, +2.05]	0 / 6
Uniform	6	24	-0.07 [-1.14, +1.60]	-0.12 [-0.22, +1.35]	0 / 0
Uniform	6	48	-0.35 [-0.89, +0.40]	+0.57 [-2.40, +1.22]	0 / 0
Uniform	4	8	-0.01 [-0.33, +0.19]	+0.02 [-1.64, +0.83]	0 / 3
Uniform	4	16	-0.16 [-0.31, +0.16]	-0.23 [-0.43, +0.23]	0 / 0
Uniform	4	32	-0.02 [-0.09, +0.47]	+0.21 [-0.10, +0.66]	0 / 0
Uniform	4	64	-0.19 [-0.43, +0.59]	+0.51 [-0.35, +0.68]	0 / 0
Zipf	6	6	+0.01 [-0.08, +0.09]	+0.12 [-0.03, +0.59]	7 / 16
Zipf	6	12	+0.27 [-0.44, +0.90]	-0.31 [-3.05, +0.54]	5 / 5
Zipf	4	8	+0.06 [-0.05, +0.39]	-0.07 [-0.24, +0.04]	11 / 11
Zipf	4	16	-0.16 [-0.56, +0.11]	-0.08 [-0.15, +0.50]	4 / 5

Predicting the next model does not help. On its own, (b) changed neither the median nor p95 measurably at any of the 12 traces. The median moved by -0.35 to +0.27 s and p95 by -0.31 to +0.57 s, and every interval spans zero. It made 4% more restores than the base under uniform traffic and 22% more under Zipf, and 54 of its 84 guesses were wasted. Under uniform traffic there is nothing to predict, because Poisson arrivals at even rates say nothing about which model comes next, and 17 of its 20 guesses there served no request. Under

Zipf traffic with one slot, every request for another model replaces the predicted one, and the guess that brings it back takes a full restore, so the share of requests that found their model serving was 9% to 25% with (b) against 8% to 20% for the base. From 16 requests per minute up under uniform traffic the GPU was rarely idle, and (b) and (a)+(b) made at most 1 guess per run.

Two slots and prediction together. (a)+(b) was no better than (a) alone. Under uniform traffic its p95 moved by -0.30 to +1.64 s against (a), and for four models at 64 requests per minute its median was 0.96 s worse, with an interval of +0.17 to +1.93 s. Under Zipf traffic it found the requested model serving more often, with 5 to 19 points fewer waits than (a) at three of four points, and its median fell to 0.51 to 0.53 s at those three. Its p95, set by the misses that still pay a full restore, moved by -1.70 to +1.14 s with every interval spanning zero. Its largest mean gain was 0.55 s per request, and it made 22% more restores than (a), 164 against 134, with 25 of its 58 guesses wasted.

Never stop a worker mid-restore. In the first test process, a request for a model other than the one being guessed killed the guess. That happened 7 times in two runs, for six models at 6 requests per minute. In 5 of the 7, the guess was 1.1 to 2.4 s into its restore. The agent's log then shows the cleanup of the half-restored worker ending after 10.0 s with its runtime delete killed, the attempt reported as still holding capacity at 20.0 s, and the capacity released 20.5 to 20.9 s after the stop, so the next worker took 23.4 to 24.1 s to be ready, against 3.54 s for the median restore on demand. The 2 stops that came 2.8 and 3.0 s into a restore ended within 0.34 s, and the next worker took 3.3 and 3.4 s. These stops raised p95 by 13.06 s with one slot, interval +5.29 to +17.33 s, and by 9.46 s with two, interval +3.97 to +14.03 s, to 24.18 and 20.58 s, over 15 s at the lowest rate. The other two processes therefore let a contested guess finish its restore and swapped after it, and every (b) and (a)+(b) result above uses that default. The two 10 s bounds are read from the agent's log, and the code path behind them was not traced further.

Restores and outputs. All 2,498 counted restores, in every arm and process, succeeded and restored from a snapshot, in 3.53 s at the median and 3.88 s at p95, against 4.59 s and 4.91 s in the previous follow-up on an older main, which again is not a paired comparison. A worker's first decode of a prompt, decoded alone, gave the same text in every arm, process and restore, 2,362 of 2,362, and the cold-loaded answer matched the restored one for 6 of 6 models. Of 199 repeat decodes, where a worker answered a prompt it had already decoded, 6 differed, all from workers in the two-slot arms, where workers live longer, 3 in (a) and 3 in (a)+(b). That is consistent with vLLM's prefix cache, which is on by default, but turning the cache off was not tested. Answers decoded in larger batches can differ, 1,182 of 1,831 equal to the text decoded alone, as vLLM's batch kernels do.

What is in the product, and what we recommend. The swap budget recommended in the previous follow-up is now in the Nodus platform behind the per-organization Beta flag node.gpu_snapshot_swap_budget, off by default, through PR #735, merged on 9 November 2026. It counts the snapshots a node reports, not its resident slots, and it has not been measured on a GPU with the flag on. Two-resident placement is now in the product too, behind the per-organization Beta flag node.gpu_snapshot_two_resident, off by default, through PR #754, merged on 9 November 2026, for an organization's own private Pools only. When a GPU holding one snapshot-swapped Function worker can fit a second at their measured GPU memory, it places the new worker beside the first rather than replacing it, releases the least recently used idle one only when a third needs the GPU, and with the flag on counts resident slots in the swap budget, so that a GPU with R resident slots takes models $\leq$ R + floor(SLO / restore p95). On this GPU at 15 s with 3.88 s restores that is 2 + 3 = 5 models, against 3 now, and six models with two resident, one more than the rule allows, held p95 at 13.88 s even at 48 requests per minute. We recommend not shipping predictive pre-restore. Holding a stop that arrives during a restore until the restore finishes, then stopping the restored worker at once, is now in the platform through PR #749, merged on 9 November 2026.

Related work. Keeping the most recently used models resident and loading the others on demand is the cache that the multi-model serving and model swapping systems of Section 7 manage, and restoring a model before it is asked for is the prefetching side of the same

problem. This follow-up does not compare with those systems. It measures both choices on the Nodus snapshot path, where a restore takes seconds.

Scope of the follow-up. This is one GPU, one host and seed 1, with one trace per point of 66 to 143 requests, so p99 is close to the maximum and the paired intervals are wide. The models are small, 0.5B to 3B, answers were 64 tokens without streaming, and all six snapshots, 41.7 GiB, stayed in the node's cache with a warm page cache and persistence mode on, which is a best case. Both layouts ran every engine at 0.45, so (a) used memory the base left idle, and neither a one-resident engine at a larger fraction nor larger models that cannot share a GPU were tested. The predictor ran at one setting, a 5-minute decay, a ratio of 1.5 and a margin of 1, and no bursty or session-shaped traffic was tested, where recent arrivals may say more about the next model. Short Zipf traces drift from their nominal shares, and in the six-model trace at 6 requests per minute the most popular model drew 33% of the requests, not 41%. The scheduler is research code in the test process driving the real node agent, and requests reached the worker by a file in its state directory rather than the product gateway. Some pairs cross test processes, which share arrivals but took their own snapshots. For six models at 6 requests per minute, the base and (a) ran in the first process and (b) and (a)+(b) in the second, and for four models at 8, (a) ran in the first and the other arms in the second. The cleanup bounds behind a killed guess come from the agent's log and were not traced in the code, and no independent verifier has rerun these measurements.

MIG slices are not swap targets yet. A separate probe on one larger GPU asked whether a slice of NVIDIA's Multi-Instance GPU could take turns by snapshots in the same way. It cannot yet. On every MIG instance tried, the smallest slice, a mid-sized one and the whole GPU as one instance, with two drivers, the host's stock one and a newer one, and both a minimal PyTorch process and vLLM 0.11.0, the checkpoint step of cuda-checkpoint failed with out of memory after 0.19 to 0.86 s in 10 of 10 probes, and the process kept running. With MIG off on the same GPU and the same tooling, vLLM restored 5 of 5 times with 8 of 8 identical answers each, decoded greedily, ready in 4.14 to 4.23 s. The probe ran plain host processes, without containers or Nodus code, and its result matches NVIDIA/cuda-checkpoint issue #42, which is open. Until the driver supports it, a MIG slice can hold a warm model but cannot be a snapshot-swap target.

11 Conclusion

Several models can take turns on one GPU through GPU memory snapshots. On one GPU, four vLLM models from 0.6B to 3B took 12 turns over 3 rounds through the local snapshot path that has shipped in the Nodus platform, every turn gave the model's cold greedy answer, and with persistence mode on a restored turn answered 6.84 s after its swap began on average and at most 8.22 s, against 82.7 s cold. One GPU served the 12 turns in 233 s, where four warm dedicated GPUs would hold 933 GPU-seconds over the same time. Without persistence mode the same turns took about 21.6 s, a cost the platform traced to GPU queries on the restore path. A platform change that shipped on 8 November 2026 takes those queries off the restore path, cut those turns to 12.80 s in one run of its first commit, and turns persistence mode on when a node is bootstrapped, which this report has not measured on production nodes. For a fleet, the result is that a model which waits for sparse requests need not hold a GPU of its own, as long as its users can wait seconds for its turn. Whether that holds for larger models, under real traffic and with a scheduler deciding the turns, is the next measurement. Under real request traffic, the follow-up found that one GPU serves two models at any load offered and four or six within a p95 of 15 s at moderate rates, at about 5 s of median latency instead of 0.51 s, and that a swap budget, not memory alone, decides how many models a GPU can hold. A further follow-up found that a second model kept resident in the memory a one-resident layout leaves idle doubles the highest offered rate one GPU serves within that p95, on no more GPU-seconds, and that predicting the next model does not help.

References

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

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

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

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

[5] A. Yang et al. Qwen3 Technical Report. arXiv preprint, 2025.

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

[7] NVIDIA. Driver Persistence. NVIDIA GPU Deployment and Management Documentation.

[8] A. Gujarati et al. Serving DNNs like Clockwork: Performance Predictability from the Bottom Up. OSDI 2020.

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

[10] Z. Li et al. AlpaServe: Statistical Multiplexing with Model Parallelism for Deep Learning Serving. OSDI 2023.

[11] J. Duan et al. MuxServe: Flexible Spatial-Temporal Multiplexing for Multiple LLM Serving. ICML 2024.

[12] Y. Xiang et al. Aegaeon: Effective GPU Pooling for Concurrent LLM Serving on the Market. SOSP 2025.

[13] Y. Fu et al. ServerlessLLM: Low-Latency Serverless Inference for Large Language Models. OSDI 2024.

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

[15] A serverless GPU platform's engineering blog. GPU memory snapshots for faster cold starts. 2025.

[16] NVIDIA. NVIDIA Dynamo Snapshot: Fast Startup for Inference Workloads on Kubernetes, with GPU Memory Service. NVIDIA technical blog and Dynamo documentation, 2026.

[17] llmux. Multiplexing vLLM models on one GPU with sleep mode and CRIU checkpoints. Crate documentation, docs.rs.

[18] R. Stoyanov et al. SwapServeLLM. SC25 Workshops, CANOPIE-HPC, 2025.

[19] M. Yu et al. Torpor: GPU-Enabled Serverless Computing for Low-Latency, Resource-Efficient Inference, earlier titled FaaSwap. USENIX ATC 2025, and arXiv preprint arXiv:2306.03622.

[20] NVIDIA. GPU memory swap. NVIDIA Run:ai documentation and NVIDIA technical blog.