GPU Snapshots in Containers

Stock vLLM reaches its first token in 4.6 to 5.2 s from an empty GPU, and four PyTorch servers in 1.13 to 3.94 s, with identical outputs

Pranav Varshney
Nodus Compute

Viswa Kotra
Nodus Compute

Abstract

A GPU that is kept warm only so that its model does not cold-start does no work while it waits, and a fleet gets more AI out of the GPUs it has if that GPU can be released when traffic stops and the server brought back in seconds when it returns. Our report 025 restored a stock vLLM engine from a GPU memory snapshot in a host Python environment, but the first slice of the GPU snapshot runtime in the Nodus node could not checkpoint a containerized vLLM engine at all. Nodus, the infrastructure execution layer for AI workloads, measured what it takes to restore real GPU servers in containers, through containerd, runc, CRIU and cuda-checkpoint as the Nodus node runtime drives them, from a GPU holding 0 MiB. On one host, a containerized stock vLLM 0.30.0 engine serving Qwen3-8B reached its first token in 4.6 to 5.2 s, against about 69 s for a stock cold start with every cache kept, 13 to 15× faster, with byte-identical outputs at temperature 0. A parked engine holds 21.8 GB of host RAM, 17.08 GB of weights and a 4.76 GB process image, and with the weights on local disk instead the restore took 21.6 s. On a second host, four PyTorch servers, an embedder, Whisper large-v3, Qwen3-8B in transformers and SDXL, restored into fresh containers to byte-identical first results in 1.13, 1.88, 2.99 and 3.94 s, 3.3 to 6.5× faster than stock starts with cached Python bytecode. Both pre-registered gates failed, independent verifiers rated both studies PARTIAL, a 70B result could not be reproduced, sharing one GPU through snapshots gained nothing over keeping the servers co-resident, and the method is already published in NVIDIA Dynamo Snapshot. What this report adds is the measurement in the Nodus stack and the container root causes that block a restore, with fixes merged into the Nodus platform's executor as PR #667 on 7 November 2026, where a containerized stock vLLM on one GPU restores into a new container with the same output. A dated follow-up found that the cuda-checkpoint step alone, run in place without CRIU, can page 12 sparsely used containerized servers of 7 small and medium models through one GPU with each inside its own SLO, 1.5× static co-location, but only at about 1.5% busy load, with 0.75× to 1.0× at the anchored load and 1.0× on a catalog with 8B LLMs, so under real load keeping models resident does as well. A second dated follow-up built the weight exclusion into the Nodus platform's Beta GPU snapshots as an opt-in for eager PyTorch code: on one GPU a Qwen3-8B worker in transformers restored in a median 5.1 s with its weights kept out of the snapshot, against 14.5 s with them in and 11.8 s for a cold start, with identical output. A third dated follow-up found that the engine process vLLM forks keeps NVIDIA device descriptors that CRIU cannot dump, so a vLLM engine created inside a worker class could not be snapshotted at all, and with the engine spawned, now the platform's

default, a Qwen3-0.6B worker on one GPU restored in a median 5.89 s with its weights and KV cache kept out through vLLM's sleep mode, against 14.84 s with the engine in the snapshot and 89.8 s for a cold start. A fourth dated follow-up restored a snapshot of a containerized stock vLLM 0.30.0 server, about 10.1 GB with its weights and KV cache inside, on a second host with the same GPU model and driver, onto a different physical GPU and in three of four cycles on a machine booted more recently, with the same greedy completion every time, in 37.2 to 44.2 s for the copy and the restore together against 88.8 to 96.9 s for a cold start, a feasibility result for a restore on another machine that the platform had not yet shipped. A fifth dated follow-up ran the restore on another machine through the platform's own fleet path, since shipped as PR #686 on 7 November 2026 behind a per-organization Beta flag, in which the node agent encrypts and uploads the snapshot after the take and another node's agent downloads it, checks it against the digest recorded at commit and restores it: on a second host whose monotonic clock was more than 6000 s behind the first's, 6 restores of a 5.39 GB Qwen3-0.6B vLLM snapshot with the weights and KV cache kept out were ready 16.3 to 17.0 s after the request, download included, against 91.5 to 95.5 s for a cold start, 5.4 to 5.8× faster, with identical greedy output, and all 3 substituted digests were refused as corrupt. A sixth dated follow-up ran Qwen3-8B through the same fleet path on a pair of hosts with an older inference GPU each and a pair with a previous-generation training GPU each: with the worth rule off, restores of 21.65 to 21.91 GB snapshots, most of each a weights file, were ready 71.2 to 71.5 s after the request on the older GPUs and 59.4 to 61.4 s on the previous-generation ones, download included, 1.8 to 2.3× faster than a cold start with identical greedy output, but with the rule enforced its assumed 250 MB/s started every worker cold, and on one host with two GPUs a snapshot taken on GPU 0 restored on GPU 1 with identical output, though a 53.9 GB snapshot failed to restore and left the GPU's capacity marked occupied. A seventh dated follow-up measured the fleet path after it shipped as a Beta feature, with PR #693 of the Nodus platform, which learns the worth rule's costs from an organization's own measurements and lets a node reuse a weights file it already holds: on two pairs of hosts, Qwen3-8B restores that reused the weights downloaded 5.07 to 5.08 GB instead of 21.63 to 21.65 GB and were ready 26.3 to 41.6 s after the request, against 116.1 to 200.8 s for a cold start, 3.2 to 7.1× faster with identical greedy output, and the learned rule fetched them, while full downloads were slower than before, 68.5 to 170.4 s, and one stalled removal is still being fixed.

1 Introduction

A GPU that is kept warm only so that its model does not have to cold-start does no work while it waits. A fleet gets more AI out of the GPUs it already has if that GPU can be released when traffic stops and the server brought back in seconds when traffic returns. Our report 025 [1] showed this for a stock vLLM [2] engine in a host Python environment on one GPU, with a restore to first token in 7.158 s. Customers' GPU servers run in containers, though, and the first slice of the GPU snapshot runtime in the Nodus node recorded that a containerized vLLM engine could not be checkpointed at all. This report measures what it takes to restore real GPU servers into containers from GPU memory snapshots, and how fast they come back.

Process and GPU checkpointing are established. CRIU [3] saves and restores a process tree, and cuda-checkpoint [4] moves a process's GPU state to host memory so that CRIU can save it. CRIUgpu [5] checkpoints GPU containers this way, and a serverless GPU platform snapshots GPU memory to cut container cold starts [6]. NVIDIA Dynamo Snapshot [7], published in May 2026, already checkpoints and restores vLLM in runc containers with CRIU and cuda-checkpoint and reports a 4.7 s restore for Qwen3-8B on a newer GPU, and its GPU Memory Service moves the weights out of the process image. llmux [8] combines

vLLM's sleep mode with CRIU, and torch_memory_saver [9] releases and restores PyTorch GPU memory inside a live process. The method here is therefore not new. What this report adds is a measurement through the Nodus node runtime's own path on two GPU types, for vLLM and for four other PyTorch servers, the root causes that stop a restore in a container and their fixes, and the host RAM a parked server costs.

The metric is restore to first token: seconds from the restore command, issued only when the GPU shows 0 MiB in use and no process, to the first streamed token of a vLLM engine or the first complete result of any other server. A stock cold start is timed from container create to the same response. Every comparison uses the same host, GPU, image and model files.

2 Experimental setup

Containerized vLLM. One host with one GPU runs stock vLLM 0.30.0 in its pinned container image, serving Qwen3-8B [10] in BF16 with sleep mode on, a worker extension that changes no vLLM code, and every other flag at its default, including a maximum length of 40960 and a memory fraction of 0.9. Containers are created, checkpointed and restored through containerd 2.4.1, runc 1.5.2, CRIU 4.3 and cuda-checkpoint, driven by the containerd client exactly as the Nodus runc executor drives them. At snapshot time the warmed engine copies its weights byte for byte from the GPU into a huge-page file in host memory, vLLM's own sleep releases the KV cache, cuda-checkpoint moves the remaining GPU state to the host, and CRIU writes the process tree to a memory file system without the weights. A restore starts the tree in a container created in advance, moves the GPU state back, copies the weights back from the host file and wakes the KV cache. The baseline is a stock cold start of the same image and flags, with the image already in containerd, the weights in the page cache and the compile caches kept from an earlier boot. Each restore is checked with 100 prompts of 16 tokens at temperature 0 against an engine that was never snapshotted.

Any PyTorch server. A second host with one GPU runs four servers in the same image: a Qwen3-Embedding-0.6B [11] embedder on 64 texts, Whisper large-v3 [12] transcribing an 11 s clip, Qwen3-8B generating 32 tokens at temperature 0 in transformers, and SDXL [13] drawing one 1024x1024 image in 20 steps. These servers have no sleep mode, so at the snapshot point every GPU tensor storage is written once to a weight file and freed, which leaves a process image of 1.44 to 1.72 GB, and after a restore 8 threads copy the weights back from the page cache at about 19 GB/s. Every restore goes into a fresh container. The stock baseline is the same server without the snapshot point, with the page cache warm and the Triton and model caches kept on node volumes. Outputs are compared byte for byte: pixels, transcript, token ids and embedding bytes.

Gates. Both studies registered their gates before the first measured run. The vLLM gate required a restore of at most 2.0 s that is also at least 3× faster than report 025's method run in the same container. The four-server gate required at least 10× over the stock start on 3 of 4 servers, and a handoff of the GPU between two parked servers in at most 5 s at p50. Both also required identical outputs and a GPU at 0 MiB with no process before every trial. Both gates failed, as reported below.

3 Results

From about a minute to about five seconds. On the verifier's fresh seeds, with restores and cold starts alternating, the restored engine served its first token in 5.209 s against 68.854 s for the stock cold start, 13.22× faster, with a bootstrap 95% interval of 12.15 to 13.83. The researcher's own trials gave 4.648 s against 65.420 s, 14.08×. The spread between 4.6 and 5.2 s comes from the host, not the method. With one more parked 8B model on the host, 21.8 GB more in memory file systems, the same restore took 5.05 to 5.21 s, and once the verifier freed that memory the researcher's snapshot restored in 4.606 s. Under that pressure the huge-page weight file fell back to 4 KB pages, the CRIU restore and GPU toggle rose from 2.80 s to 3.07 to 3.26 s, and the weight copy rose from 1.02 s to 1.47 s. A restore into a container and root file system created inside the timed window took 5.044 s.

Table 1: Restore to first token for stock vLLM 0.30.0 serving Qwen3-8B in a container on one host, with the GPU at 0 MiB and no process before every trial. Medians. The verifier used fresh prompts and alternated restores with cold starts. The last column divides the same person's stock cold start by the arm.

Arm	Run by	Trials	Median s	Stock cold start over arm
Restore, quiet host	Verifier	3	4.606	14.9×
Restore	Researcher	12	4.648	14.08×
Restore	Verifier	10	5.209	13.22×
Restore into a new container	Verifier	3	5.044	13.65×
Report 025's weight reload, same container	Researcher	9	7.054	9.27×
Weights inside the process image	Researcher	3	10.231	6.39×
Weights on local disk, not in host RAM	Verifier	2	21.616	3.19×
Stock cold start, every cache kept	Researcher	6	65.420	1.00×
Stock cold start, every cache kept	Verifier	10	68.854	1.00×

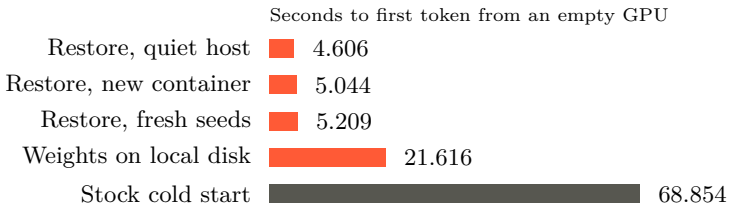


Figure 1: Seconds from the restore or create command to the first token for containerized vLLM with Qwen3-8B on one host, verifier's medians, lower is better.

Where the time goes. In the researcher's restores the CRIU restore of the process tree took 1.93 s, the GPU toggle 0.88 s, the weight copy 1.05 s for 17.08 GB, about 16.6 GB/s including mapping and page-locking, vLLM's KV cache wake 0.72 s and the first token 0.04 s. The 2.8 s of process restore and GPU toggle alone exceed the registered 2.0 s, so that gate cannot be met while the whole process is restored. Report 025's method, which reloads the weights from the model files instead of copying them from host memory, took 7.054 s in the same container, so the gain over it is 1.51×, short of the registered 3×. Putting the weights inside the process image took 10.231 s. The stock cold start spends roughly 30 to 40 s starting the container and the API server and 16 to 27 s initializing the engine, with the weights read from the page cache in 3.2 s, and keeping Triton's cache as well did not make it materially faster, 67.688 s.

Outputs and throughput are unchanged. Every guarded restore returned the same 100 completions as an engine that was never snapshotted, 100 of 100, on the researcher's prompts and on the verifier's fresh ones. Under a 60 s load of 4 requests per second after restore, the engine served 517 tokens per second with a TPOT of 9.97 ms, against 517 tokens per second and 10.20 ms for a normally started engine.

The 70B result is unverified. The researcher also reported a Llama-3.3-70B engine in FP8 restoring in 6.76 s against an 83.15 s cold start. That result fell short of its own pre-registration, with 6 trials against 10 required, 3 of 6 trials checked for identity, no throughput check and one cold start. It also could not be reproduced: the original container had been deleted, and restoring the snapshot into a new container failed 6 of 6 times because CRIU could not open a Triton-compiled cuda_utils file that existed only in the original container's writable layer. This report does not count it as a result.

Seconds for any PyTorch server. Restored into fresh containers from a GPU at 0 MiB, the four servers returned their first result in 1.13, 1.88, 2.99 and 3.94 s, byte-identical to the stock servers on fresh inputs the researcher never used: 48 of 48 timed trials, and 40 of 40 in the bytecode block. The researcher's own medians reproduced within about 2%. These times include each server's own work: SDXL's 20 denoising steps take 2.53 s of its 3.94 s, and Qwen3-8B's 32 tokens take 1.27 s. For Qwen3-8B, the runc and CRIU restore took 0.40 s, the GPU toggle 0.40 s and copying 16.38 GB of weights back 0.85 s.

Table 2: Restore to first result for four PyTorch servers in fresh containers on the second host, medians of the verifier's trials on fresh inputs. The first stock start recompiles Python bytecode on every boot, as the researcher's baseline did. The second keeps bytecode in a cache directory, and its ratio uses the restores of the same block.

Server	Restore s	Stock s, recompiling	Ratio	Stock s, bytecode kept	Ratio
Qwen3-Embedding-0.6B embedder	1.126	14.256	12.66×	7.314	6.50×
Whisper large-v3	1.877	16.038	8.54×	8.691	4.61×
Qwen3-8B, transformers	2.993	18.176	6.07×	11.159	3.74×
SDXL	3.938	22.366	5.68×	12.995	3.29×

The researcher's baseline recompiled Python. The researcher reported 5.4 to 12.8× over a stock start, and the verifier's fresh-input re-run gave 5.7 to 12.7×. That baseline is handicapped. The pinned image ships no Python bytecode for torch, 2,285 source files, or for transformers, 2,681, the harness turned off writing bytecode, and every fresh container starts with an empty writable layer, so every stock start compiled both libraries from source. Importing torch took 3.5 s instead of 1.7 s, and the framework imports finished at 11.5 to 16.8 s instead of 5.2 to 7.5 s. Any image build can ship bytecode, and none of that needs a GPU snapshot. With bytecode kept in the already persisted cache directory the stock start was 39% to 49% faster, and the restore's lead fell to 3.3 to 6.5×, a geometric mean of 4.4×. No server reached the registered 10×, against 1 of 4 on the handicapped baseline.

What makes it fast. Leaving the weights out of the process image matters most for the largest model: with them inside, the Qwen3-8B image was 17.89 GB and its restore took 5.752 s against 2.964 s in the researcher's runs. Transparent huge pages set to always on the node speed up CRIU's page restore, and with them set to madvise the restores were 20% to 63% slower. With the page cache dropped, so that the image and the weights come from the local disk, the researcher's restores took 2.20 to 7.20 s.

Handing one GPU between parked servers. The pre-registered handoff moved the GPU between a parked Qwen3-8B server and a parked SDXL server with a full CRIU dump of the outgoing one and a restore of the incoming one. Its p50 over 20 switches was 7.815 s in the verifier's run and 7.942 s in the researcher's, above the 5 s gate. Rotating over all four servers was faster: 2.918 s at p50 when the parked processes stay in host RAM and only their GPU state is moved out, 7.7 GB for the four, and 3.473 s when the outgoing dump overlaps the incoming restore and no parked process is left between switches. Outputs were identical at every switch and the GPU read 0 MiB between servers.

Snapshots did not raise density here. The researcher measured four customers' sparse servers sharing one GPU through these handoffs at 3.8 to 3.9× the SLO-met requests per GPU-second of four dedicated GPUs. The verifier found that all four stock servers fit on this GPU together, 36,661 MiB at peak, and that keeping them co-resident without any snapshot gave the same 4.00× on the same seeded arrivals, 34 of 34 requests within the 10 s SLO on seeds 0 and 1, at a p50 of 0.70 to 0.90 s against 3.1 to 4.7 s for the handoff. The gain came from consolidation, not from the snapshot, so this report claims no density gain from snapshots. They would add GPU work only where the servers do not fit together or must not share a GPU, and neither case was measured.

Table 3: What a parked server holds in host RAM. For vLLM the weights sit in a huge-page file, for the four servers in a weight file kept in the page cache. The last row is the restore with the weights read from local disk instead, for the four servers with the page cache dropped.

Item	vLLM	Four servers
Weights in host RAM	17.08 GB	1.19 to 16.38 GB
Process image	4.76 GB	1.44 to 1.72 GB
Host RAM per parked server	21.8 GB	2.6 to 17.9 GB
Restore from local disk	21.616 s	2.20 to 7.20 s

Host RAM is the price. A parked 8B vLLM engine holds 21.8 GB of host RAM, 17.08 GB of weights and a 4.76 GB process image, with the KV cache excluded. With the weights off host RAM and read from this host's local disk at 0.86 GB/s, the same restore took 21.616 s, still about 3.2× faster than the cold start. A snapshot also depends on its node: it restores onto the same GPU model and driver, the node must disable io_uring for the CRIU dump, and CRIU refuses a mapped weight file whose permission bits changed after the snapshot.

4 What it took to restore in a container

Each blocker below was a measured failure of a checkpoint or restore in a container, found by one or both studies, and each has a fix. The failure recorded by the first product slice, CRIU's parasite unable to get a /proc file descriptor, did not recur with containerd's default container spec in either study or in the product run.

Table 4: Container root causes found by the two studies, with the fix each needs.

Cause of the failure	Fix
A Triton file in the writable layer	Carry the layer with the snapshot
Unlinked /dev/shm semaphores in vLLM	link-remap through org.criu.config
CRIU loads its own cuda_plugin	An empty CRIU plugin directory
A clash on CUDA's cuda-uvmfd socket	A network namespace per container
A symlinked resolv.conf bind mount	Own resolv.conf and hosts files
Leftover CRIU network-lock rules	A namespace per attempt, then a flush
Page restore in 4 KB pages	Transparent huge pages set to always

Files from the writable layer. Triton compiles a cuda_utils library into the container's writable layer when the engine starts, and the restored process still maps it, so a restore into a new container cannot open the file. The fix is to carry the writable layer with the snapshot, or to keep compile caches on node volumes. Semaphores. vLLM's multiprocessing semaphores are open but unlinked files in /dev/shm, and CRIU stops the dump with Can't create link remap until it is given link-remap and a 1 GiB ghost limit, which runc passes to it from a per-container configuration named by the org.criu.config annotation. The same configuration points CRIU's plugin directory at an empty directory, because CRIU's cuda_plugin otherwise tries to drive cuda-checkpoint while the executor already toggles the GPU.

Network namespaces. CUDA binds an abstract socket named cuda-uvmfd after the process's pid namespace and pid. Every worker is pid 1 and namespace numbers are reused, so in a shared network namespace a live worker can hold the exact name a restore needs, and overlapping restores fail with Address already in use. An interrupted CRIU dump or restore also leaves its network-lock rules in a shared namespace, which silently drops loopback traffic, so every later engine listens but never answers. Both are fixed by a network namespace per container and attempt, with CRIU's rules flushed before reuse. Mounts and pages. A host resolv.conf bind mount whose source is a symlink into another memory file system cannot be remounted on restore, so the container needs its own resolv.conf and hosts files, and CRIU's page restore faults memory in 4 KB pages unless transparent huge pages are set to always on checkpoint nodes.

With the link-remap configuration, stock vLLM 0.30.0 serving Qwen3-0.6B, its API server and engine core processes included, checkpointed and restored into fresh containers on the second host and returned its first completion in a median 3.17 s against a 61.6 s cold start with a warm compile cache, with identical completions. With a network namespace per container, 60 overlapped switches had 0 failures.

In the Nodus platform. The platform change that carries the writable layer with the snapshot and passes this CRIU configuration through org.criu.config was merged into the Nodus platform as PR #667 on 7 November 2026. Its executor packs the container's writable-layer diff into the snapshot, refuses a layer above 8 GiB, and applies the layer to the new container's root file system before runc restores the tree. On another host through that executor, stock vLLM 0.30.0 serving Qwen3-0.6B in the product's container spec, in its own

network namespace, cold-started to its first completion in 94.8 s and 96.3 s. Checkpointed while left running, it produced a 17.19 GB snapshot in 42.4 s with a 59.7 MB compressed writable layer, and restored into a new container to its first completion in 12.3 s, and in 11.6 s on an earlier run, with the same text. The same test against the runtime before the change stops at the dump on the link-remap error. A CUDA workload that maps a file it wrote to its root file system, holds a multiprocessing lock and deletes a file from its image restored into a new container in 1.56 to 1.65 s with its GPU memory, the file and the deletion intact, and does not restore on the old runtime. This product snapshot keeps the GPU memory inside the image, so its 12.3 s is a correctness result, not the speed reported in section 3. Keeping the weights out of the image is now in the product as a Beta opt-in, merged on 7 November 2026: PR #673 for a worker's weights in eager PyTorch code, and PR #677 for a vLLM engine's weights and KV cache through its sleep mode, with GPU-snapshot workers spawning vLLM's engine by default.

5 Independent verification

A separate verifier reran the vLLM study on the same host with the researcher's code unchanged and fresh prompts, alternating restores and cold starts over 10 trials each. It reproduced the gain, 13.22× against the researcher's 14.08× on true medians, and identical outputs on every guarded trial. It ran arms that the researcher had registered but not run: the restore into a new container, 5.044 s, and the weights on local disk, 21.616 s, plus a cold start with Triton's cache kept, 67.688 s. It found that the absolute time depends on host memory, that the weight copy inside a restore runs at about 16.6 GB/s rather than the 55.4 GB/s ceiling the report implied, that one ablation was mislabeled, that the 70B point cannot be reproduced, and that the method is already published in Dynamo Snapshot [7], whose Qwen3-8B split into a 4.8 GiB process image and a 15 GiB weight artifact matches this one. It rated the study PARTIAL and recommended the claim this report makes.

A second verifier reran the four-server study on the same host with fresh inputs the researcher never used. Every re-measured number reproduced within about 2%, and outputs were byte-identical throughout. It found the stock baseline handicapped by Python bytecode compilation and the density gain matched by co-residency, and it rated the study PARTIAL and not publishable as claimed. This report uses its corrected numbers: restores of 1.13 to 3.94 s, 3.3 to 6.5× faster than a stock start with bytecode kept, a handoff that failed its gate, and no density claim from snapshots. It also noted that the researcher's second snapshot shared its weight file with the first, and that two identity counts in the researcher's report did not trace to the results file, although every output was identical.

6 Scope

Each study ran on one host with one GPU, restored only onto the node that took the snapshot and never ran two restores at once, so cross-node restore, concurrent restores and other GPU types are untested. The fast restores of the four servers keep the process image and the weight file in the page cache. Their weight release frees tensor storages, which is valid only for code that does not keep raw device addresses, so engines that capture graphs need their own sleep mode, as vLLM has. The vLLM headline restores run into a container created in advance, so those snapshots depend on that container's writable layer. The platform change that carries the writable layer and the CRIU configuration in GPU snapshots was merged into the Nodus platform as PR #667 on 7 November 2026, as Section 4 reports. The 70B point is unverified. The product run covers one small model on one GPU with the GPU memory inside the image.

7 Related work

Checkpointing GPU processes with CRIU and cuda-checkpoint [3] [4] is the basis of CRIUgpu [5], a serverless GPU platform's memory snapshots [6] and NVIDIA Dynamo Snapshot [7], whose GPU Memory Service keeps the weights out of the process image as both studies here do. llmux [8] combines vLLM's sleep mode with CRIU, and

torch_memory_saver [9] pauses and resumes tagged PyTorch GPU memory, a more general form of the storage release used for the four servers. ServerlessLLM [14] speeds up loading the weights rather than restoring a process. Switching one GPU between models is the subject of PipeSwitch [15], which keeps standby workers live and pipelines the model transfer, and of Aegaeon [16], which pools many models per GPU with switching at token granularity, a far stronger result on GPUs saved than four servers per GPU with a handoff of seconds. Our report 025 [1] restored vLLM in a host Python environment and is the direct predecessor of this work.

8 Follow-up, 7 November 2026: paging whole GPU servers through one GPU

Question and setup. Section 3 found that snapshots raised no density where four servers fit on the GPU together. A later study asked whether the same cuda-checkpoint step, run in place without CRIU or a disk, can page whole containerized GPU servers of different models through one GPU, so that many sparse tenants share it. Every tenant is an unmodified server in its own container: stock vLLM 0.30.0, diffusers, Whisper in transformers, or an embedding or reranking encoder. When a tenant is idle, a pager moves its whole CUDA state to host RAM, and when a request arrives it moves the state back while a router holds the request, so the page-in counts in the tenant's latency. Tenant popularity takes its shape from the public ServeGen per-client rates [17], at a mean of 2.37 requests per minute per tenant, the light load, and at 4 times that, the anchored load. Each tenant is judged against its own SLO over 300 s windows, at p95 with the page-in counted: a TTFT of at most 2.0 s and a TPOT of at most 50 ms for LLMs, its dedicated p95 plus 2 s for speech, twice its dedicated p95 plus 2 s for images, and 1.0 s for embedding and reranking. A window passes only if every tenant that sent a request passes. The baseline is static co-location, the same tenants packed onto GPUs by measured memory and all resident. The pre-registered gate asked for 3× the tenants per GPU of static co-location.

Table 5: Measured cost of paging a whole server process in place with cuda-checkpoint on one GPU, medians per server. Host RAM is what the parked process holds.

Server	GPU MiB	Page-in s	Page-out s	Host RAM parked, GiB
bge-reranker-v2-m3 encoder	1672	0.658	1.395	3.73
Whisper large-v3	4024	1.027	2.736	5.62
Qwen3-4B-FP8 in vLLM	6280	1.380	4.011	9.76
SDXL in diffusers	11676	2.238	7.083	12.95
Qwen3-8B BF16 in vLLM	17536	3.260	10.403	20.68

What a page costs. Across 14 servers from a 1672 MiB reranker to a 17660 MiB 8B LLM, a page-in took 0.336 s plus 0.172 s per GiB of GPU memory, about 6.2 GB/s, and a page-out 0.438 s plus 0.585 s per GiB, about 1.8 GB/s. Qwen3-8B in BF16 pages in in 3.26 s and out in 10.4 s, and holds 20.7 GiB of host RAM while parked. The three 7-8B BF16 LLMs page in in 3.26 to 3.41 s, longer than the whole 2.0 s TTFT budget, so they cannot be paged on demand and must stay resident. With that rule removed, the Qwen3-8B tenant reached a p95 TTFT of 7.28 s. A page-out is slower than a page-in, so a page-in that has to wait for one misses its SLO: without a free-memory reserve, an SD1.5 tenant reached a p95 of 5.96 s against its 3.56 s limit.

Table 6: Tenants per GPU with every tenant inside its own SLO, page-in counted. Paging counts only if every seed and repeat passes. Static co-location packs the same tenants by measured GPU memory, all resident.

Fleet	Load	Paging	Static co-location	Paging over static
7 small and medium models	Light, about 1.5% busy	12	8	1.5×
7 small and medium models	4 times light, anchored	6 to 8	8	0.75× to 1.0×
14 models with 8B LLMs	Light	6	6	1.0×

Twelve tenants on one GPU, at almost no load. On one GPU, 12 tenants of 7 small and medium models, Qwen3-4B-FP8 in vLLM, SD1.5, SDXL, Whisper large-v3 and medium, Qwen3-Embedding-0.6B and bge-reranker-v2-m3, stayed inside their own SLOs with page-in counted on 7 seeds, the researcher's 1 to 3 and the verifier's fresh 7 to 10, plus 2 same-seed repeats. Static co-location fits only the first 8 of these tenants on one GPU, so paging held 1.5× as many. The load was very light. A tenant alone on its own GPU is busy about 1.5% of the time at this rate, 1.1% to 1.7% across 3 seeds measured on the 14-model catalog's tenants, only 6 to 8 of the 12 tenants sent any request in a 5-minute window, and the whole GPU served 39 to 59 requests per window. Plain least-recently-used paging, with no tenant kept resident, no reserve and no prefetch, also passed with 12 tenants on seed 7, with 15 page-ins against 21 for the SLO-aware pager, so the gain comes from parking idle processes in host RAM, not from the pager's rules. With 14 tenants, 2 of 4 seeds failed, each on one tenant that sent 2 or 3 requests.

No gain at the anchored load or with 8B LLMs. At 4 times the light load, the level the study was anchored to, paging held 8 tenants on seeds 1, 2, 3, 7 and 8, but the same-seed repeat of seed 1 failed when the busiest Whisper tenant reached a p95 of 2.660 s against its 2.651 s limit. In the first run of seed 1, 3 of its 36 requests paid a page-in and its p95 was 2.64 s, against 1.49 s under static co-location, which held the same 8 tenants on all 3 seeds it ran. Paging therefore holds 6 to 8 tenants per GPU at this load against 8 for static co-location, 0.75× to 1.0×, never more. On the full 14-model catalog, which adds three 7-8B BF16 LLMs, an 8B FP8 LLM and an interactive LLM, paging held 6 tenants and static co-location fits the same 6 on one GPU, 1.0×. The gate failed: its two-GPU windows were never run, and no one-GPU result came near 3×.

Outputs and driver hazards. Paging changed no output. After every window each tenant answered 3 fixed probes, LLM token ids at temperature 0, image bytes, transcripts and embedding or score bytes, and every digest matched the reference recorded before any paging, in all 63 windows that ran probes. The vLLM probes need a random cache salt, because without one a prefix-cache hit changed the temperature-0 output of 4 of the 8 vLLM servers whether or not they were paged. The driver has two hazards that a pager must avoid. A restore that ran out of device memory left the process permanently unrestorable, failing every later attempt with OS call failed or operation not supported on this OS. Processes that ran on a nearly full GPU, or were killed during a checkpoint, later failed to checkpoint with unknown error. The pager therefore restores a tenant only when the GPU shows its size plus 1 GiB free, and waits for every page operation in flight before it exits.

Independent verification and prior art. A separate verifier ran fresh seeds 7, 8, 9 and 10 on the same two hosts with the researcher's code unchanged, re-judged every window from the per-request logs with its own code and matched every verdict. It rated the study PARTIAL and not publishable as a report of its own: the gate fails, the advantage disappears at the anchored load and on the full catalog, and the SLO-aware pager adds nothing at the headline point. The method is also published. SwapServeLLM [18] suspends and resumes containerized vLLM, SGLang and TensorRT-LLM engines with the driver's transparent checkpoint and restore, and routes requests with memory reservation. Torpor [19] keeps idle inference models in host RAM, swaps them onto GPUs on request and queues requests against their SLOs, and NVIDIA Run:ai GPU memory swap [20] moves idle inference workloads to CPU memory in a product. What is new is only the measurement: a paging cost model for diffusers, Whisper and encoder servers as well as LLMs, and a density judged against each tenant's own SLO and against memory-packed static co-location, including the result at the anchored load.

Reading. For getting more AI out of each GPU, paging whole servers helps only where tenants are nearly idle. It lets one GPU host a long tail of sparse servers that would otherwise each hold GPU memory, but it adds no GPU work: at the anchored load it never beat keeping the same tenants resident, and an 8B BF16 LLM cannot be paged inside a 2 s TTFT at all. Under real load, keeping models resident or consolidating them onto shared engines does at least as well, as co-residency did against the snapshot handoff in Section 3.

9 Follow-up, 7 November 2026: keeping weights out of the snapshot, in the Nodus platform

Question and setup. Section 3 found that leaving the weights out of the process image keeps it at 1.44 to 1.72 GB for the four servers, and Section 4 named keeping the weights out of the image in the product as the next step. Nodus has since built this into its platform, where GPU snapshots are a Beta feature. A customer opts in by calling nodus.exclude_from_gpu_snapshot(self.model) in the worker class's enter hook. When the snapshot is taken, the worker writes the model's GPU tensors once to a file on the machine and frees them, the snapshot holds the rest of the process, and every restored worker reads the tensors back into the same tensors before it serves. A worker that cannot read them back exits instead of serving without its weights. The opt-in is safe only for eager PyTorch code, because the tensors come back at new device addresses, which breaks CUDA graphs and torch.compile in its reduce-overhead mode, so vLLM and other engines that capture graphs keep their weights in the snapshot. The measurement ran on one host through the platform's own node agent and container runtime, with a real SDK worker class that loads a model in transformers from its image: Qwen3-8B, whose weights file is 16.4 GB, or Qwen2.5-0.5B-Instruct, whose file is 0.99 GB. Each of four runs made 3 cold starts and 3 restores with the weights in the snapshot, and the same with them kept out.

Table 7: Time to ready on one GPU through the Nodus platform, median with the range in brackets, over 12 restores per arm from four runs and over 24 cold starts per model. The process image is the snapshot without the weights file, as a median.

Measure	Qwen3-8B	Qwen2.5-0.5B
Weights file	16.4 GB	0.99 GB
Cold start, s	11.8 (11.2 to 16.3)	5.9 (5.5 to 7.1)
Restore, weights in, s	14.5 (14.2 to 17.4)	3.1 (3.0 to 3.3)
Restore, weights kept out, s	5.1 (5.0 to 6.6)	2.6 (2.5 to 2.8)
Process image, weights in	18.2 GB	2.7 GB
Process image, weights kept out	1.7 GB	1.7 GB

An 8B model restores in a median 5.1 s. With the weights kept out, Qwen3-8B restored to ready in a median 5.1 s, against 14.5 s with the weights in the snapshot and 11.8 s for a cold start, 2.9× and 2.3× faster at medians. With the weights inside, every 8B restore, 14.2 to 17.4 s, was slower than 23 of the 24 cold starts, so on this host a snapshot of the 8B worker pays off only with the weights kept out. The reported snapshot size includes the weights file, and without that file the process image is about 1.7 GB with the weights kept out, against about 18.2 GB with them in, in line with the 1.44 to 1.72 GB of the four servers in Section 3. For Qwen2.5-0.5B, whose weights are small, the gain is small too: 2.6 s against 3.1 s with the weights in, and 5.9 s for a cold start. Greedy decoding gave identical output in every arm of every run, and the generated text was the same with the weights in or kept out.

Scope. Each run used one host and models read from the image with a warm page cache, and restored only onto the machine that took the snapshot, because restoring onto another machine is not built yet. It covers eager code only, and vLLM and other engines that capture graphs keep their weights in the snapshot for now. The comparisons are medians over 12 restores per arm. One cold start took 16.3 s, on a freshly booted host in the fourth run, against 11.2 to 12.9 s for the other 23. The four runs used successive revisions of the change during its review, and the numbers come from the platform's own integration test, which no independent verifier has rerun.

10 Follow-up, 7 November 2026: vLLM in a worker class, in the Nodus platform

Question and setup. Section 4 snapshotted and restored a containerized vllm serve through the platform's executor, and Section 9 left vLLM's weights inside the snapshot. A GPU

Function's code usually creates its engine inside the worker's own process instead, with the vllm.LLM class. Nodus measured that case on one host through the platform's own node agent and container runtime, with Qwen3-0.6B on vLLM 0.30.0 at GPU memory utilization 0.6 in a worker class, the engine either kept in the snapshot or with its weights and KV cache kept out through vLLM's sleep mode. To keep them out, the worker writes the weights once to a file on the machine and the engine sleeps, which releases the weights and the KV cache without a copy. Sleep mode keeps each allocation's GPU addresses reserved, so a restored worker maps the weights back at the same addresses and the engine's CUDA graphs stay valid. Each of two runs made 3 cold starts and 3 restores per arm, the first on a previous revision of the change with the spawn default described below set in the image's environment, the second on the change's final code with the SDK setting it.

A forked engine cannot be snapshotted. Before this change, the engine could not be snapshotted at all. Both arms of the platform's test failed at the CRIU dump, which stopped on vLLM's engine-core process with Can't dump file 9 of that type (chr 195:255). By default, vLLM 0.30.0 forks the engine process of an in-process vllm.LLM, and the forked engine inherits its parent's NVIDIA device descriptors. Probes on the same host counted them after NVIDIA cuda-checkpoint had toggled CUDA off in each process. The front-end process held none, but the engine still held 4, /dev/nvidiactl as descriptor 9 and three /dev/nvidia0, which cuda-checkpoint does not release and CRIU cannot dump. NVML was not the cause, because shutting it down inside the engine did not release them, and neither was sleep mode, since the engine held the same 4 with sleep mode off, or two copies of the driver library, since it held the same 4 with one. vllm serve spawns its engine process instead of forking it, and that engine released every descriptor, which is why the containerized server in Section 4 could be snapshotted. So did vllm.LLM with VLLM_WORKER_MULTIPROC_METHOD=spawn. The Nodus platform now sets that variable to spawn in every GPU-snapshot worker, unless the customer has set it. The hazard is general: any library that forks a process after its parent has opened NVIDIA devices leaves descriptors that a CRIU snapshot cannot hold.

Table 8: A vllm.LLM engine in a worker class on one GPU through the Nodus platform, Qwen3-0.6B on vLLM 0.30.0 at GPU memory utilization 0.6. Times to ready are medians with the range in brackets, over 6 restores and 6 cold starts per arm from two runs. Snapshot sizes are medians and include the weights file.

Measure	Engine in the snapshot	Weights and KV cache kept out
Before the spawn default	Fails at the CRIU dump	Fails at the CRIU dump
Cold start, s	90.3 (89.5 to 92.8)	88.8 (87.6 to 91.5)
Restore, s	14.84 (14.66 to 17.92)	5.89 (5.86 to 6.07)
Snapshot	17.09 GB	5.39 GB
Weights file in the snapshot	None	1.45 GB
Greedy output identical	Yes	Yes

A restore in a median 5.89 s. With its weights and KV cache kept out, the worker restored to ready in a median 5.89 s, against 14.84 s with the engine in the snapshot and 89.8 s over all 12 cold starts, 2.5× and 15× faster at medians, and every restore with them kept out was faster than every restore with the engine in. With the engine inside, the snapshot was 17.09 GB, most of it the KV cache reservation: keeping the weights and KV cache out shrank it by 11.7 GB to 5.39 GB, a 1.45 GB weights file included. Greedy decoding gave identical output in every arm of both runs. Section 9 said that vLLM keeps its weights in the snapshot. In fact, an engine in a worker class could not be snapshotted at all before the spawn default, and with vLLM 0.30.x its weights and KV cache can now stay out.

Scope. The measurement covers one host, one small model and vLLM 0.30.x only. vLLM 0.32 and later tag more of their GPU memory and move some of it to host memory when they sleep, so the platform keeps such an engine inside the snapshot. It covers one GPU per engine and restores only onto the machine that took the snapshot. Only the vllm.LLM class was measured, and the platform's other accepted engine classes, LLMEngine and AsyncLLM, were not. The descriptor probes ran on the same host at GPU memory utilization 0.5. The two runs used successive revisions of the change, the numbers come from the platform's own

integration test, and no independent verifier has rerun them. GPU snapshots, and keeping a vLLM engine's weights out of them, are Beta in the Nodus platform.

11 Follow-up, 7 November 2026: restoring on another machine

Question and setup. Every restore in this report so far ran on the machine that took the snapshot, and Sections 6, 9 and 10 left a restore on another machine untested. Restoring on another machine would let the platform bring a worker back on whichever machine has a free GPU, instead of only on the one that kept its snapshot. That needs a snapshot taken on one machine to restore on another machine with the same GPU model and driver, onto a different physical GPU, and also when the second machine has booted more recently, so that its monotonic clock, which counts from boot, is behind the first machine's. Nodus measured both on two hosts, A and B, with the same GPU model and driver, through the Nodus platform's guarded runc executor with containerd 2.4.1, runc 1.5.2, CRIU 4.3 and NVIDIA cuda-checkpoint. The server was a containerized stock vLLM 0.30.0, as in Section 4, serving Qwen3-0.6B, here at GPU memory utilization 0.3 to keep the snapshot small, with its weights and KV cache inside the snapshot. Each cycle cold-started the server on one machine, recorded its greedy completion, took a snapshot while the server kept running, copied the snapshot directory to the other machine as a tar stream over ssh, and restored it there. The test passed only if the restored server gave the same completion on its first request and on a second. B was rebooted before the first restore, and three cycles ran from A to B and one from B to A.

Table 9: Restoring a GPU memory snapshot on another machine, two hosts A and B with the same GPU model and driver, through the Nodus platform's executor. Stock vLLM 0.30.0 serves Qwen3-0.6B at GPU memory utilization 0.3, with the weights and KV cache inside the snapshot. Cold start and restore run to the first completion. Uptime is the machine's time since boot when its take or restore finished. The copy in cycle 1 was not timed, and its bound comes from the log timestamps.

Measure	Cycle 1	Cycle 2	Cycle 3	Cycle 4
Direction	A to B	A to B	A to B	B to A
Cold start before the take, s	89.4	88.8	88.9	96.9
Take, s	23.0	22.9	23.0	56.3
Snapshot, GB	10.06	10.06	10.09	10.09
Copy, s	Under 40	35.2	35.4	29.4
Restore, s	8.47	9.06	8.07	7.79
Copy and restore, s	Not timed	44.2	43.4	37.2
Taking machine up, s	607	872	1042	631
Restoring machine up, s	100	305	473	1297
Same completion, twice	Yes	Yes	Yes	Yes

The same output on another GPU, with the clock behind. All 4 restores passed. Each restored server gave the same greedy completion as the server that was snapshotted, on its first request and on a second, on a different physical GPU from the one that took the snapshot, as the two GPUs' identifiers show. In the three cycles from A to B, B had been up for 100 to 473 s when each restore finished, against 607 to 1042 s for A when it took the snapshot, so the restoring machine's monotonic clock was behind the one the snapshot was taken under. In the cycle from B to A it was ahead. Each snapshot was about 10.1 GB. The three timed copies took 29.4 to 35.4 s, and the first copy, which was not timed, took under 40 s by the log timestamps. Restoring to the first completion took 7.79 to 9.06 s, against 88.8 to 96.9 s for a cold start, so the copy and the restore together, 37.2 to 44.2 s, took less than half as long as every cold start. On B, after its reboot, the one take took 56.3 s and the one cold start 96.9 s, against 22.9 to 23.0 s and 88.8 to 89.4 s on A, and the cause was not measured.

A projection, not a measurement. At the copy rates measured here, 286 to 343 MB/s of snapshot data, the 5.39 GB snapshot of Section 10, with the weights and KV cache kept out, would copy in about 16 to 19 s. That snapshot was never copied or restored on another machine, so this is an estimate only.

Scope. This is a feasibility result for the restore on another machine that the Nodus platform plans, not a shipped feature. It covers two hosts of one GPU model with the same driver, one small model at GPU memory utilization 0.3 with the weights and KV cache inside the snapshot, and four cycles, three in one direction and one in the other. The copy was a plain tar stream over ssh from one machine to the other, not through object storage, and no encryption or integrity check of the snapshot was measured. Other GPU models and drivers, larger models and snapshots with the weights kept out were not tested. The numbers come from a test harness added to the platform's executor test, and no independent verifier has rerun them.

12 Follow-up, 7 November 2026: restoring from the fleet store

Question and setup. Section 11 restored a snapshot on another machine after copying its directory as a tar stream over ssh, with no encryption or integrity check. The Nodus platform's own fleet path, built since and under review (since merged as PR #686), moves the snapshot through the node agent instead. The agent that took a worker's snapshot encrypts it and uploads it to an object store right after the take, and when a worker of the same key starts on another node, that node's agent downloads the stored snapshot, checks it against the digest recorded when the upload was committed, accepts only regular files and directories, restores it and serves. The question is whether the gain of Section 11 holds with the encryption, the integrity checks and the agent in the loop. Nodus measured it on two hosts, A and B, with the same GPU model, the same driver and identical checkpoint profiles, through the node agent and its guarded runc executor with containerd 2.4.1, runc 1.5.2, CRIU and NVIDIA cuda-checkpoint. The worker was the vLLM 0.30.0 worker of Section 10, serving Qwen3-0.6B at GPU memory utilization 0.6 with its weights and KV cache kept out of the snapshot. On A, 3 workers started cold, and the agent took and uploaded a snapshot of each to an S3-compatible MinIO store on A. B was then rebooted, and its agent restored each of the 3 stored snapshots twice, first with the server's worth rule off and then with the rule and its download deadline applied as the server applies them. A third run on B substituted each snapshot's recorded digest.

Table 10: Restoring from the fleet store on two hosts A and B with the same GPU model, driver and checkpoint profile, through the Nodus node agent. The vLLM 0.30.0 worker of Section 10 serves Qwen3-0.6B at GPU memory utilization 0.6 with its weights and KV cache kept out of the snapshot. Each snapshot was taken and uploaded on A and restored twice on B, with the worth rule off and then enforced, the two values in a cell. Cold start and end to end run from the request for the worker to the worker ready to generate. Upload runs from the worker's report of its take to the agent's report of the upload. Restore to ready is end to end minus the download.

Measure	Snapshot 1	Snapshot 2	Snapshot 3
Cold start on A, s	95.5	92.9	91.5
Snapshot, GB	5.39	5.39	5.39
Upload done after the take, s	10.3	10.3	10.5
Download on B, s	8.89 / 8.85	8.86 / 8.88	8.84 / 8.86
Download rate, MB/s	606 / 609	608 / 607	609 / 608
Restore to ready, s	7.69 / 8.11	8.15 / 7.47	8.13 / 7.45
End to end, s	16.59 / 16.96	17.01 / 16.34	16.97 / 16.31
Faster than the cold start	5.76× / 5.64×	5.46× / 5.68×	5.39× / 5.61×
Worth rule's prediction, s	28.96, fetch	28.96, fetch	28.93, fetch
Same greedy output	Yes / Yes	Yes / Yes	Yes / Yes
Substituted digest refused, s	0.24	0.23	0.23

The gain held through the fleet path. All 6 restores came from the fleet store onto B's GPU, a different physical GPU from A's by their identifiers, and every restored worker gave the same greedy output as the cold worker on A. B had been up 63 to 131 s when each of its runs began, against 6731 s for A after its last take, so B's monotonic clock during every restore was more than 6000 s behind A's at the take, against about 484 to 546 s in Section 11. Each snapshot was 5.39 GB, a 1.45 GB weights file included. The agent reported each

encrypted upload complete 10.3 to 10.5 s after the worker reported its take, and 10.8 to 10.9 s after the agent logged the take. On B the download took 8.84 to 8.89 s, at 606 to 609 MB/s, and the restore to ready 7.45 to 8.15 s more, so each worker was ready 16.31 to 17.01 s after it was requested, against 91.5 to 95.5 s for the cold starts of the same workers on A, 5.39 to 5.76× faster. In Section 11 the copy and the restore together took 37.2 to 44.2 s, but for a snapshot of about 10.1 GB with the weights and KV cache inside and over a different transfer, so the two sections do not compare the fleet path with a tar copy like for like.

The integrity check and the worth rule. With one bit of each recorded digest flipped, all 3 fetches were refused as corrupt 0.23 to 0.24 s after the worker was requested, on the digest of the stored manifest and before any of the snapshot's content was read, and the agent started each worker cold instead. The test did not wait for those cold starts. The worth rule, which assumes a conservative 250 MB/s for the download, predicted 28.93 to 28.96 s from the request to ready for each snapshot, and when it was enforced it chose to fetch every time, with download deadlines of 65.8 to 69.0 s. Each measured end to end was 11.9 to 12.6 s shorter than the prediction. The margin came from the download, which the rule put at 21.5 to 21.6 s against the 8.84 to 8.89 s measured, while its allowance for the restore, 7.39 s, was slightly below every measured restore to ready.

Scope. This measures the fleet path of the Nodus platform, which is built and under review, not a shipped feature (since merged as PR #686). It covers two hosts of one GPU model with the same driver and identical checkpoint profiles, one small model with its weights and KV cache kept out of the snapshot, 3 takes, 6 restores and 3 substituted digests. The store was an S3-compatible MinIO on A, which B reached over an SSH tunnel, not the production object store, so the upload and download speeds measured here are not representative of production. The fleet decisions, whether a stored snapshot is offered, the worth rule's verdict and the deadline, came from the agent's node-protocol test double, so the server side of the fleet path was not in this loop. Cold starts were measured on A only. Other GPU models and drivers, larger models and snapshots with the weights inside were not tested through this path. The numbers come from the platform's own integration test, and no independent verifier has rerun them.

13 Follow-up, 7 November 2026: an 8B model, two GPU types, and another GPU index

Question and setup. Section 12 restored a Qwen3-0.6B worker through the platform's fleet path on two hosts. Two questions remained: whether the gain holds for an 8B model, whose snapshot is mostly its weights file, on more than one GPU type, and whether a snapshot taken on one GPU of a host restores on another GPU of the same host. For the first, Nodus ran the test of Section 12 on two new pairs of hosts, A and B, a pair with an older inference GPU each and a pair with a previous-generation training GPU each, with the same driver and identical checkpoint profiles in each pair. The worker was the vLLM 0.30.0 worker of Section 10, serving Qwen3-8B at GPU memory utilization 0.9 with its weights and KV cache kept out of the snapshot and its weights file uploaded with it. On A, 2 workers started cold, and the agent took and uploaded a snapshot of each to an S3-compatible MinIO store on A. B was then rebooted, and its agent restored each stored snapshot with the worth rule off, then received the same requests again with the rule enforced. For the second, on one host with two GPUs, a cold worker pinned to GPU 0 took a snapshot, a worker of the same key on GPU 0 restored from it as a control, and then a worker of the same key pinned to GPU 1 was started.

With the rule off, the gain held at 8B on both GPU types. All 4 restores came from the fleet store onto B's GPU, a different physical GPU from A's by their identifiers, and every restored worker gave the same greedy output as the cold worker on A. Each B had been rebooted and was up 65 and 222 s in the pair with the older GPU, and 60 and 208 s in the pair with the previous-generation GPU, when its two runs began, against 1847 and 2043 s for A after its last take. On the older GPUs each 21.65 GB snapshot downloaded in 64.07 to 64.11 s at 338 MB/s and restored to ready in 7.08 to 7.35 s more, so each worker was ready 71.15 to 71.46 s after it was requested, against 127.4 to 130.7 s for the cold starts on

Table 11: Qwen3-8B through the fleet path on a pair of hosts with the older GPU and a pair with the previous-generation GPU, A and B in each, with the same driver and checkpoint profile in a pair. The vLLM 0.30.0 worker of Section 10 serves Qwen3-8B at GPU memory utilization 0.9 with its weights and KV cache kept out of the snapshot, its weights file uploaded with it. The columns are the two snapshots of each pair, Older for the older GPU and Prev. gen. for the previous-generation GPU. Each was taken and uploaded on A and restored on B with the worth rule off, then requested again with the rule enforced. Cold start and end to end run from the request for the worker to the worker ready to generate. Restore to ready is end to end minus the download.

Measure	Older, 1	Older, 2	Prev. gen., 1	Prev. gen., 2
Cold start on A, s	130.7	127.4	138.9	138.5
Snapshot, GB	21.65	21.65	21.88	21.91
Weights file in it, GB	16.57	16.57	16.57	16.57
Upload done after the take, s	38.8	38.6	42.1	45.5
Download on B, s	64.07	64.11	47.78	47.74
Download rate, MB/s	338	338	458	459
Restore to ready, s	7.08	7.35	13.59	11.64
End to end, s	71.15	71.46	61.37	59.37
Faster than the cold start	1.84×	1.78×	2.26×	2.33×
Same greedy output	Yes	Yes	Yes	Yes
Worth rule's prediction, s	110.25	110.25	111.39	111.53
0.6 of the cold start, s	78.4	76.4	83.3	83.1
With the rule enforced	Started cold	Started cold	Started cold	Started cold

A, 1.78 to 1.84× faster. On the previous-generation GPUs the 21.88 to 21.91 GB snapshots downloaded in 47.74 to 47.78 s at 458 to 459 MB/s and restored to ready in 11.64 to 13.59 s, for 59.37 to 61.37 s end to end against 138.5 to 138.9 s cold, 2.26 to 2.33× faster.

With the rule enforced, every worker started cold. The worth rule predicted 110.25 to 111.53 s from the request to ready, the download at its assumed 250 MB/s, 86.6 to 87.6 s, plus its restore allowance of 2 s and 1 s per GB, 23.65 to 23.91 s. That is more than 0.6 of every cold start, 76.4 to 83.3 s, so with the rule enforced the agent started all 4 workers cold, and the test did not time those cold starts. The conservative prior gave up a measured 1.78 to 2.33× gain. Every measured end to end was below 0.6 of its cold start, so the rule given the measured download and restore would have fetched every time. Given the measured download alone, with the restore allowance unchanged, it would have fetched on the previous-generation GPUs, at 71.6 to 71.7 s against 83.1 to 83.3 s, but not on the older GPUs, at 87.7 to 87.8 s against 76.4 to 78.4 s. The server's rule replaces the restore allowance with a restore time recorded for the snapshot, but only after a fleet restore of it, which a rule that never fetches does not produce, and it keeps the bandwidth fixed. The server also applies the rule before it accepts an upload, which the test double does not, so with cold starts like these it would not have stored these snapshots at all. Learning the bandwidth from measured downloads is the next product step, and it is not yet built.

The weights file is most of an 8B snapshot. Each 8B snapshot held a 16.57 GB weights file, 75.6 to 76.5% of it, so the rest of the snapshot was 5.08 GB on the older GPUs and 5.31 to 5.34 GB on the previous-generation GPUs, against a 1.45 GB weights file in the 5.39 GB snapshots of Section 12.

A projection, not a measurement. If B did not have to download the weights file, then at the download rates measured here the download would take about 15.0 s on the older GPUs and 11.6 s on the previous-generation GPUs instead of 64.1 s and 47.7 to 47.8 s, and with the restore to ready as measured each worker would be ready in about 22.1 to 22.4 s and 23.3 to 25.2 s, about 5.5 to 5.9× faster than its cold start. No restore without downloading the weights file was run, so this is an estimate only.

On two GPUs, a snapshot taken on GPU 0 restored on GPU 1. On one host with two GPUs and one driver, the vLLM worker served Qwen3-0.6B at GPU memory utilization 0.1 with its weights and KV cache kept out, each worker pinned to one GPU through the node agent, the guarded runc executor and CDI. The snapshot taken on GPU 0 was 6.16

Table 12: A snapshot taken on GPU 0 of one host with two GPUs and restored on GPU 0 as a control and on GPU 1, through the Nodus node agent. Run 1 had no worker on GPU 1. The vLLM worker serves Qwen3-0.6B with its weights and KV cache kept out of the snapshot. Ready runs from the request for the worker to the worker ready to generate.

Measure	Run 2, GPU 0	Run 2, GPU 1	Run 1, GPU 0
GPU memory utilization	0.1	0.1	0.6
Snapshot, GB	6.16	6.16	53.9
Weights file in it, GB	1.95	1.95	None
Outcome	Restored	Restored	Failed, about 72 s
Ready after the request, s	65.6	67.6	Never ready
Restored process on	GPU 0	GPU 1	None
Same greedy output	Yes	Yes	Not applicable

GB with a 1.95 GB weights file. The control on GPU 0 and the worker on GPU 1 both restored from it and gave the same greedy output as the cold worker, with no device map: the executor runs cuda-checkpoint with a process id only, and the run script leaves CRIU's CUDA plugin, which handles device mappings, off. The restored process ran on GPU 1: NVML inside it listed only GPU 1's identifier, with minor number 1, and the host's NVML put its 8076 MiB of GPU memory on GPU 1. CRIU's log of that restore reached Restore finished successfully 2.37 s after it began. The worker on GPU 1 was ready 67.6 s after it was requested, against 65.6 s for the control on GPU 0. On this host the agent logged each container restored at least 65.0 s after its request, so nearly all of the time to ready passed before the restore was reported, and where it went was not measured. These times are given as measured and are not comparable with the older GPUs' times above.

A restore that failed. At GPU memory utilization 0.6 on the same host, the snapshot taken on GPU 0 was 53.9 GB, with no weights file beside it, and restoring it even on the same GPU failed: about 72 s after the control was requested, the agent reported the restore failed with execution admission expired. The agent then could not confirm that the attempt's processes were cleaned up, with the error no such device, and kept the GPU's capacity marked occupied, logging so 9425 times in the first 0.15 s and still when the test gave up at its 10-minute limit, while both GPUs held 0 MiB. The test stopped there, so that run restored nothing on GPU 1. A fix is under way, and this report does not claim the failure is fixed.

Scope. This measures the fleet path of the Nodus platform, which is under review, not a shipped feature (since merged as PR #686), and a restore onto another GPU of one host. It covers one small model, Qwen3-0.6B on the host with two GPUs, and one 8B model through the fleet path, two hosts per GPU type for the fleet path, the older and the previous-generation GPU, with identical checkpoint profiles in each pair, 2 takes and 2 restores per arm, with the rule off and with it enforced, and one cross-index run on one host model with two GPUs. The store was MinIO on host A of each pair, which B reached over an SSH tunnel, not the production object store, so the download speeds measured here are not representative of production. The fleet decisions came from the agent's test double, so the server side was not in the loop. The numbers come from the platform's own integration tests, and no independent verifier has rerun them.

14 Follow-up, 8 November 2026: learned costs and reusing weights

Question and setup. In Section 13 the worth rule's fixed priors, a download at 250 MB/s and a restore of 2 s plus 1 s per GB, declined every 8B fetch, although most of each 8B snapshot was a 16.57 GB weights file that a node could keep. The fleet path has since shipped in the Nodus platform as a Beta feature behind a per-organization flag, in PR #686, merged on 7 November 2026, and PR #693, merged on 8 November 2026, changes it in four ways. The worth rule learns its download rate and its restore time from the organization's own measurements, starting from its uploads and from restores of snapshots a node took itself, counting a fetch abandoned at its deadline, and weighting older measurements less. A restore reads a sealed snapshot object as parallel HTTP ranges, by default 8 ranges of 64

MiB at once. The weights file is stored as an object of its own, and a node that already holds it downloads only the rest: the node checks its copy once, when it keeps it, and copies it into each later restore without hashing it again. The fleet key no longer names a GPU index, after the restore on another GPU in Section 13. Nodus ran the test of Section 13 with an agent built from the PR's commit aa2e8dad6 on two new pairs of hosts with the older GPU, A and B, one pair set up with GPU persistence mode on and the other with it off. On A, 2 workers serving Qwen3-8B at GPU memory utilization 0.9, with the weights and KV cache kept out, started cold, and the agent took and uploaded a snapshot of each and restored each once on A. B was then rebooted, and its agent restored each stored snapshot twice with the worth rule off and then, as a new agent whose cache starts empty, twice more with the learned rule and its deadline enforced. A restore downloads the whole snapshot unless B has already downloaded and kept a weights file of the same bytes.

Table 13: Qwen3-8B through the fleet path with PR #693 at its commit aa2e8dad6, on a pair of hosts with the older GPU, set up with GPU persistence mode on, and a pair set up with it off, A and B in each. The vLLM 0.30.0 worker of Section 10 keeps its weights and KV cache out of the snapshot. Each pair took 2 snapshots on A, and B restored them with the worth rule off and then enforced. Ranges are over the restores of each kind. Cold start and end to end run from the request for the worker to the worker ready to generate. Restore to ready is end to end minus the download.

Measure	Persistence on	Persistence off
Cold start on A, s	121.1 to 134.4	116.1 to 200.8
Snapshot, GB	21.63 to 21.65	21.65
Same weights file in both takes	No	Yes
Full downloads	3	2
Full: download, s	49.69 to 49.71	59.88 to 59.89
Full: rate, MB/s	435 to 436	362
Full: restore to ready, s	18.74 to 120.71	41.90 to 97.30
Full: end to end, s	68.46 to 170.40	101.79 to 157.18
Weights reused	2	6
Reused: downloaded, GB	5.07 to 5.08	5.08
Reused: download, s	8.28 to 8.33	8.45 to 8.53
Reused: rate, MB/s	610 to 612	596 to 601
Reused: restore to ready, s	28.39 to 33.30	17.86 to 24.86
Reused: end to end, s	36.67 to 41.64	26.32 to 33.37
Reused: faster than cold	3.23× to 3.30×	3.84× to 7.05×
Same greedy output	Every restore	Every restore
Started cold by the rule	1	None
Not run, removal stalled	2	None

Reusing the weights cut the download to about 5 GB. Every restore that reused B's kept weights file downloaded 5.07 to 5.08 GB of a 21.63 to 21.65 GB snapshot, in 8.28 to 8.53 s at 596 to 612 MB/s, and its worker was ready 26.32 to 41.64 s after it was requested, against 116.1 to 200.8 s for the cold starts of the same workers on A, 3.2 to 7.1× faster. One cold start on the pair set up with persistence off took 200.8 s, against 116.1 to 134.4 s for the other three, and the 6.02× and 7.05× restores are measured against it, so the 3.23 to 4.41× of the other six restores is the more typical gain. Section 13 projected about 22.1 to 22.4 s end to end on the older GPU for a restore that did not download the weights file, using the 7.08 to 7.35 s restore to ready measured there, and here the restore to ready took 17.86 to 33.30 s. All 13 restores, 5 that downloaded the whole snapshot and 8 that reused the weights, came onto a GPU different from A's by their identifiers after B's reboot, and every restored worker gave the same greedy output as its cold worker on A.

A full download was slower than in Section 13. The 5 restores that downloaded the whole snapshot, before B held its weights, spent 49.69 to 59.89 s on the download, at 362 to 436 MB/s, against 64.07 to 64.11 s at 338 MB/s in Section 13, but 18.74 to 120.71 s on the restore to ready, against 7.08 to 7.35 s there. They were ready 68.46 to 170.40 s after the request, 0.79 to 1.97 times as fast as the cold start, so the slowest was slower than its cold start. On the PR's first revision, f5063be98, on another pair with the older GPU, full downloads were ready 6.84 to 7.45 s after their download and 66.91 to 68.37 s after the

request. Since then a node that downloads a weights file also makes its own checked copy of it, reading and writing 16.57 GB while the restore runs, where the first revision kept a link to the downloaded file. Where the extra time went was not broken down.

The learned rule fetched the reused weights. Before any fleet restore, the rule had learned 264 and 269 MB/s on the two pairs, 0.6 of the slower of the two uploads on A, 440 and 448 MB/s, and a restore of 2 s plus 0.42 and 0.73 s per GB from the two restores on A. For every restore that would reuse the weights it predicted 26.38 to 62.92 s from the request to ready, under 0.6 of the cold start, 69.7 to 120.5 s, so it fetched each of them: with the rule enforced, the 3 such restores that ran were ready 27.86 to 30.25 s after the request, and a fourth was granted but did not run, as described below. In Section 13 the fixed priors declined every 8B fetch. They would now fetch these too, predicting 43.89 to 43.98 s, because the rule now counts only the bytes a node downloads, so the reuse of the weights, more than the learning, is what makes these fetches pay off. The server's check before an upload counts the same, so, unlike in Section 13, it would store these snapshots, which this test did not exercise. With the rule enforced, the learned rule also fetched one full download on each pair, predicting 54.10 and 68.45 s, and those were ready 157.18 and 170.40 s after the request, the second later than its 134.4 s cold start. Each raised the learned restore time, from 0.73 and 0.77 to 1.84 and 2.27 s per GB beyond 2 s, and on the pair set up with persistence on the rule then declined the next full download, predicting 100.86 s against 0.6 of its cold start, 72.7 s.

Where a reused restore's time goes. A restore that reuses the weights copies the node's kept 16.57 GB file into the snapshot's directory while the rest downloads, without hashing it. On these hosts' ext4 root that is a full copy in the kernel. By the agent's log, each such fetch finished 5.62 to 10.38 s after its download, against 0.32 s after a full download, the restore was reported 6.44 to 23.42 s after the fetch, and the worker was ready 1.27 to 1.34 s after that. The copy is therefore part of the 17.86 to 33.30 s restore to ready, not all of it. On the PR's first revision, which hashed every copy, the fetch finished 20.49 to 20.99 s after its download and the restore to ready took 33.74 to 35.03 s. A projection, not a measurement: on a filesystem with clones, such as XFS or btrfs, the agent clones the kept file instead of copying it, which takes almost no time, so the fetch would not wait for the copy. How much of the restore to ready that would save was not measured.

Ranged reads showed no gain here. On the PR's first revision, full downloads that read 8 ranges of 64 MiB at once ran at 354 to 357 MB/s, and with a single stream at 358 to 361 MB/s. The store sat behind a single SSH tunnel, one TCP connection that every range shared, so the gain of ranged reads against a real object store is unmeasured, not disproven.

Failures the GPU runs found. Two runs of intermediate revisions hung at the second take: once its snapshot was uploaded, the second take's worker could not be removed after its kill, with the agent reporting its cleanup unconfirmed and its capacity occupied until the test failed at its 30-minute limit, so nothing was restored. The cause was not established. Before the merge, the PR stopped a taking node from keeping its own copy of an upload's weights, and on every later pair both takes completed and their workers were removed. A second fault surfaced in one of report 034's runs, where a Qwen3-8B snapshot of 38.14 GB kept its weights inside with nothing logged to say why, and PR #706, merged before PR #693, now reports the reason whenever a worker's weights stay in its snapshot. On another intermediate revision, one restore that reused its weights failed to start 13.27 s after its fetch with no error logged. The merged code logs such a failure, and the same arm passed on the final code, but the cause is not known. One failure remains. On the pair set up with persistence on, after the enforced rule started a worker cold, the test moved on and the attempt was cancelled while its container was being created. The agent could not confirm its cleanup, although none of its processes, mounts or snapshot descriptors was left, and kept its capacity marked occupied for the 19.9 minutes until the run was stopped, so the last 2 enforced restores did not run. A fix is under way separately, and this report does not claim it is fixed.

Scope. This covers one GPU model, the older GPU, and one 8B model, with two hosts per pair: two pairs on the final code and five pairs on earlier revisions, 2 takes per pair and 13 restores on the final code. The store was MinIO on host A, which B reached over a single SSH tunnel, not the production object store, so the download rates are not representative

of production. The fleet decisions came from the agent's test double, which applies the server's estimator to the run's own measurements, so the server itself was not in the loop, the measurements carried no age, which left the weighting of older ones unexercised, and no fetch was abandoned. The two pairs differ in persistence mode and also in their hosts, and with one run of each the difference in their restore times is not attributed to persistence mode. The numbers come from the platform's own integration test, and no independent verifier has rerun them. PR #693 has shipped in the Nodus platform as a Beta feature behind the per-organization fleet flag.

15 Follow-up, 8 November 2026: where a snapshot's fixed time goes, and cutting it

Question. The first follow-up of report 035 found that a training snapshot spent about 5 s outside the move of its GPU state and the CRIU dump, and the model swaps of report 034 pay the restore side of the same path. A follow-up on one GPU with persistence mode on timed each step of a take and a restore in the Nodus node runtime, with research builds of the platform that log every step, every containerd call and every cuda-checkpoint call, and then measured a change that removes most of that time. Three workloads ran before and after on the same host in one session: vLLM 0.30 with Qwen2.5-0.5B and Qwen3-8B in transformers, each with its weights kept out of the snapshot and 5 cold takes and 5 restores, four vLLM models taking 8 turns, and the trainer of report 035 with a snapshot every 25 s.

Table 14: Where a take's time went before the change, in seconds, the median of each step over the takes. The steps are medians, so they do not add up to the whole take exactly. The dump and the diff ran inside containerd's checkpoint helper, with the worker frozen.

Step	vLLM 0.5B	Qwen3-8B	Trainer
Takes	5	5	14
Move GPU state out	0.97	0.44	1.21
runc and CRIU dump	2.20	1.05	1.90
containerd diff of the layer	40.72	13.16	4.01
Move GPU state back	1.34	0.70	0.80
Layer copy out and sync	0.31	0.00	0.00
Whole take	45.92	15.45	8.08

The fixed time is a diff of the writable layer, with the worker frozen. The runtime took each snapshot through containerd's client helper for checkpoints. After runc and CRIU dumped the process tree, that helper recorded the container's writable layer with containerd's walking differ, which walks the whole image and the whole root filesystem side by side and reads both copies of every file whose modification time has no fractional second. It then gzipped the layer into containerd's content store, and the runtime copied it out, synced it and deleted the checkpoint image. The diff took 40.72 s of a 45.92 s vLLM take and 13.16 s of a 15.45 s 8B take, so it scales with the image rather than with the workload, and the vLLM takes ranged from 19.9 to 48.4 s. For the trainer it took 4.01 s of 8.08 s. The first follow-up of report 035 left a remainder of 5.3 s at the median, 4.2 to 8.9 s per take, measured on other hosts with a snapshot every 60 s. The diff is about 4.0 s of it, 76% of that median, and the rest of this session's path outside the move of GPU state and the dump added 71 ms, so about 1.2 s of that median is not attributed. Each cuda-checkpoint state query was its own process of about 78 ms, run one after another, 12 per vLLM take for 0.93 s. On a swap, the node's restore spent 1.41 s of 4.92 s applying the writable layer through a temporary overlay mount.

The change. The change is now in Nodus's node runtime, as PR #736, merged on 9 November 2026. It does three things. The take asks containerd for the dump alone, and the runtime, with the worker still frozen, writes the writable layer itself, uncompressed, from the overlay upper directory. A restore unpacks the layer straight into the new snapshot's upper directory, so nothing is mounted. The state queries run together, one per process. The research builds below wrote an uncompressed layer of 104.7 to 127.4 MB, against 31.4 to 34.1 MB gzipped. Their writer, like the PR's earlier versions, walked the upper directory

against a read-only view of the image, as containerd's erofs differ does. An independent review of an earlier head found cases that walk got wrong, among them a directory whose only change was its owner, and the merged writer writes every entry of the upper directory instead.

Table 15: Before and after on the same host in one session, with an earlier version of the change. Research build is a build of it that logs each step, PR binary the PR's own build of it, which does not log steps. Takes and restores are medians unless the row says mean. The trainer's stall is per snapshot, 14 before and 11 after.

	Before	After	After from
vLLM 0.5B take	45.92 s	4.15 s	Research build
Qwen3-8B take	15.45 s	1.98 s	Research build
Trainer stall	8.07 s	4.00 s	Research build
vLLM first worker, offer to serving	21.95 to 50.46 s	6.09 to 6.25 s	PR binary
Swap first answer, mean of 8	6.10 s, max 6.75 s	4.58 s, max 5.05 s	PR binary
Node restore in swaps	4.92 s	3.45 s	Research build
vLLM restore to ready, mean of 5	5.29 s	4.09 s	PR binary
Qwen3-8B restore to ready	4.96 s	4.71 s	Research build

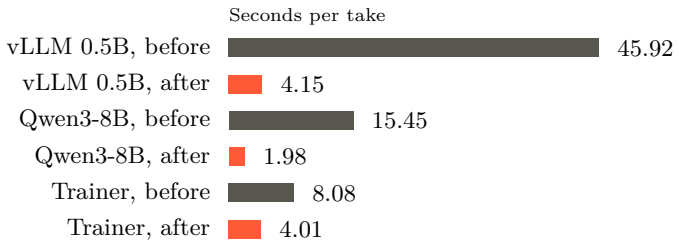


Figure 2: Seconds per take before and after the change, research builds, lower is better.

Takes fell by 87% to 91%. The vLLM take fell from 45.92 s to 4.15 s, 91% less, and the 8B take from 15.45 s to 1.98 s, 87% less, with the dump unchanged at 2.20 and 1.05 s and the layer written in 241.1 and 13.6 ms. The trainer's stall per snapshot halved, from 8.07 s to 4.00 s. With the PR binary, a vLLM worker's first snapshot, timed by the worker from its offer to serving again, took 6.09 to 6.25 s against 21.95 to 50.46 s before. Restores gained less. The four-model swap's first answer fell from 6.10 s to 4.58 s on average, 1.52 s or 25% less, and its slowest from 6.75 s to 5.05 s, and the node's restore during swaps fell from 4.92 s to 3.45 s at the median, with the layer applied in 536 ms instead of 1412 ms. A vLLM restore reached ready in 4.09 s against 5.29 s on average. The 8B restore barely moved, 4.71 s against 4.96 s, because reading back its 16.4 GB weights file dominates it. Every restore test reported the cold start's answer after each restore, every swap restored with its cold answer and a RESTORED event, and each cold worker logged one TAKEN event.

Two more savings, not in the change. Toggling the CUDA processes at once took another 0.16 s off a swap's move of GPU state back, 752 to 588 ms, and its first answer fell from 4.63 s to 4.41 s, but it changes how the driver sees a tree's toggles and was measured on one workload. Skipping processes that map no libcuda took about 0.06 s off a take's two moves of GPU state.

The merged head, measured again. The head that merged was measured on 9 November on one GPU in one session, against main with timing added, on the same host and with the same tests. A vLLM worker's first snapshot took 5.84 to 5.95 s from its offer to serving again, against 20.80 to 56.01 s on main. A vLLM restore reached ready in 3.97 s against 5.15 s on average, 23% less, and the four-model swap's first answer took 4.55 s against 6.09 s on average, 1.55 s or 25% less, and at most 5.00 s against 6.95 s. Every restore gave the cold start's answer, and every swap restored with its cold answer and a RESTORED event. The runtime logs a line whenever a take falls back to containerd's full diff, and no run on the merged head logged one.

Table 16: The merged head against main on one GPU in one session on 9 November 2026. A snapshot here is a worker's first, timed from its offer to serving again.

	Main	Merged head
vLLM 0.5B snapshot, 5 workers	20.80 to 56.01 s	5.84 to 5.95 s
Swap models' snapshots, 4 workers	21.03 to 52.75 s	5.98 to 8.57 s
vLLM restore to ready, mean of 5	5.15 s	3.97 s
Swap first answer, mean of 8	6.09 s, max 6.95 s	4.55 s, max 5.00 s

The layer test. A new CPU-only test of the platform, TestCheckpointLayerRoundTrip, takes and restores a workload and compares a record of every entry of its root filesystem before and after. On that host, on the merged head, it passed in all four of its variants, the upper-directory path, a take that leaves the task running, a forced fallback to the full diff, and the restore of a recorded gzip layer, with all 26,649 entries identical each time.

Scope of the follow-up. Each comparison is one host in one session. The shipped writer handles every directory in the upper directory, reads opaque directories only from trusted.overlay.opaque, and falls back to containerd's full diff whenever the overlay's options could hide a change from it. That host had those options off, so the fallback ran only in the layer test. The per-step numbers come from research builds of the earlier version, and the merged head was measured end to end only. The layer test leaves out the modification time of /tmp, which CRIU's dump changes, and, on the full diff, that of a directory with no other change, which that diff drops. The trainer's restores are not compared, because in 4 of 5 restores per arm the image was unpacked again first, for 32 s. The platform's GPU checkpoint test, extended to a deleted file, a copy-up, an opaque directory, a hard link and a symlink in the layer, passed on every head tested, the merged one included, and a separate test of lapsed restore grants failed the same way on every build.

16 Conclusion

Containerized GPU servers can be brought back in seconds. On one host, a stock vLLM engine serving Qwen3-8B returned to its first token in 4.6 to 5.2 s from a GPU at 0 MiB, 13 to 15× faster than a stock cold start with every cache kept, with identical outputs and throughput. On a second host, four PyTorch servers restored into fresh containers in 1.13 to 3.94 s, 3.3 to 6.5× faster than a stock start with bytecode kept. The price is host RAM, 21.8 GB per parked 8B vLLM engine, and without it the restore takes 21.616 s from local disk. Neither study met its pre-registered gate, the 70B point is unverified, snapshots did not beat co-residency on density, and the method is already published. For a fleet, the useful result is that a GPU held warm only to avoid a cold start can be released when its server goes idle, as long as the node keeps the snapshot, and that the container fixes this needs, the writable layer and the CRIU configuration, are in the Nodus platform, merged as PR #667 on 7 November 2026. The eighth follow-up found that most of a snapshot's fixed time was containerd's diff of the writable layer with the worker frozen, and that a change now in Nodus's node runtime, PR #736, cut takes by 87% to 91% in research builds and, as merged, a swap's first answer by 25%.

References

[1] Nodus Research. Cold Start in Seconds. Report 025, 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] R. Stoyanov et al. CRIUgpu: Transparent Checkpointing of GPU-Accelerated Workloads. arXiv preprint arXiv:2502.16631, 2025.

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

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

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

[9] SGLang project. torch_memory_saver: pause and resume of PyTorch GPU memory. github.com/sgl-project/torch_memory_saver.

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

[11] Y. Zhang et al. Qwen3 Embedding: Advancing Text Embedding and Reranking Through Foundation Models. arXiv preprint, 2025.

[12] A. Radford et al. Robust Speech Recognition via Large-Scale Weak Supervision. ICML 2023.

[13] D. Podell et al. SDXL: Improving Latent Diffusion Models for High-Resolution Image Synthesis. ICLR 2024.

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

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

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

[17] Y. Xiang et al. ServeGen: Workload Characterization and Generation of Large Language Model Serving in Production. NSDI 2026.

[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.