Transparent and Unified CPU-GPU Snapshots for LLM Inference Platforms

Radostin Stoyanov*
University of Oxford
Oxford, United Kingdom
radostin.stoyanov@eng.ox.ac.uk

Viktória Jurič Spišaková
Masaryk University
Brno, Czech Republic
spisakova@ics.muni.cz

Fergus Finn
Doubleword
London, United Kingdom
fergus.finn@doubleword.ai

Dan Feigin
NVIDIA
Tel Aviv-Yafo, Israel
dfeigin@nvidia.com

Andrei Vagin
CRIU
Seattle, Washington, USA
avagin@gmail.com

Rodrigo Bruno
INESC-ID, Instituto Superior Técnico,
University of Lisbon
Lisbon, Portugal
rodrigo.bruno@tecnico.ulisboa.pt

Abstract

Inference platforms often need to free GPUs held by inactive model servers. Scaling them to zero, however, makes each new replica repeat costly model loading and initialization. Our production measurements show that engine initialization can take several minutes, creating a tradeoff between making GPUs available to other workloads and limiting queueing delays when demand increases. Checkpointing can avoid repeating this work by preserving a consistent snapshot of the initialized model server's CPU, GPU, container, and filesystem state. In this paper, we present TUCS, an infrastructure-level checkpoint and restore system for AI platforms that combines CPU and GPU checkpointing with container-runtime state to reduce model-server startup latency through reusable snapshots. In our evaluation, TUCS reduces mean time to first token by up to 85%, from approximately nine minutes to under 90 seconds for GPT-OSS-120B. For requests queued while a production model scaled up from zero, we estimate an 82% reduction in mean queueing delay.

CCS Concepts

• **Software and its engineering** → **Checkpoint / restart**; • **Computer systems organization** → *Cloud computing*.

Keywords

GPU Checkpoint/Restore, Kubernetes, LLM Inference, CRIU

1 Introduction

Platform engineering teams increasingly deploy large language models (LLMs) to support production applications. However, demand for these models fluctuates over time, while static allocations continue to reserve CPU and GPU resources during low-activity periods. This increases operational costs and limits resource availability for other workloads in multi-tenant clusters [2]. Our production inference trace illustrates how model replicas retain GPU memory despite fluctuating compute utilization (Figure 1a). GPU compute utilization varies between 0% and 92% over the week, while GPU memory occupancy stays above 81%. Figure 1b highlights periods when queueing inside running replicas accounts for 94–100% of p95 time to first token (TTFT) for Gemma 4 31B and GLM 5.2.

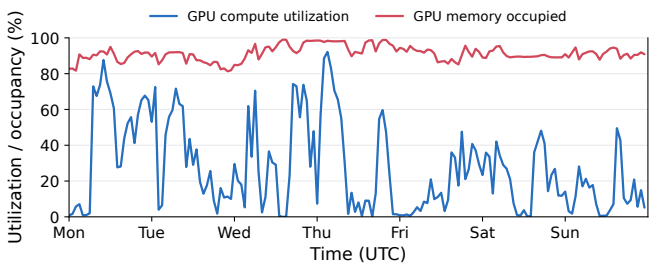
TUCS targets Kubernetes inference platforms where a gateway routes requests to model-server Pods and an autoscaler adjusts the number of replicas to demand. When a model has no ready replica, requests queue until one becomes available. Scaling down releases GPU capacity, but recreating replicas requires an initialization phase that can take several minutes [2, 5, 13, 19]. Reusable snapshots preserve initialized model servers in persistent storage, allowing the platform to release their GPUs and restore replicas on demand.

Transparent checkpointing can reduce the cost of recreating GPU workloads [3, 6, 12, 16–18]. It preserves loaded model weights and GPU contexts without requiring application-specific serialization [12, 17]. GPU checkpointing in production environments such as Kubernetes clusters requires more than capturing and restoring GPU state [7, 12, 17]. To create a consistent snapshot, the checkpoint operation must coordinate the pausing of GPU threads and saving of GPU state with CPU process checkpointing, while capturing changes to the container's writable filesystem layer. Similarly, the restore operation must recreate the Kubernetes Pod and container resources, restore the CPU and GPU state in the required order, and ensure that the workload resumes correctly. These cross-layer operations require interactions across the GPU driver, checkpointing engine (CRIU), container runtime (runc, containerd), and cluster-level orchestration system.

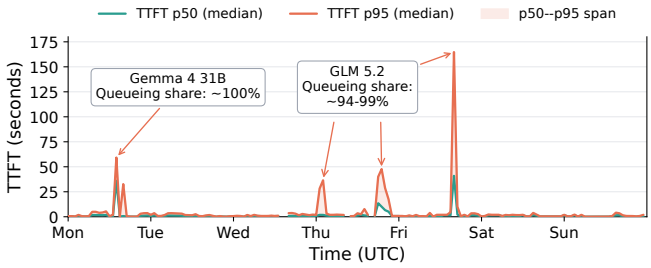
To address these challenges, we present TUCS, a system for transparent and unified CPU-GPU container checkpointing on AI platforms. Building on recently introduced driver capabilities [9], it coordinates GPU checkpoint and restore operations with CPU process checkpointing, container-runtime state, and the Kubernetes lifecycle to create and restore consistent snapshots. After the snapshot is stored, the platform can remove the source replica and release its GPU. When demand returns, the same snapshot can restore one or more replicas without repeating model loading and initialization. Reusing the snapshot amortizes its creation cost across repeated model activations.

Our evaluation results show that restoring from a TUCS snapshot reduces mean time to first token by 55–85% across four B200 configurations. For two production Qwen3.5-4B scale-up events containing 28 requests, we estimate an 82% reduction in mean queueing delay using the measured container restore time and holding the remaining request path fixed. Our compression experiments show that LZ4 reduces snapshot sizes by 12–46%. The restore

*Also with Doubleword.



(a) GPU compute utilization and memory occupancy.



(b) Time-to-first-token (TTFT).

Figure 1. Analysis of a production inference trace collected over one week from two Kubernetes nodes, each with 8×B200 GPUs. The trace covers 26 language and multimodal model configurations, including Qwen3.6 35B, GLM 5.2, and Nemotron 3 Ultra. (a) Hourly mean compute utilization and memory occupancy across Pods, weighted by GPU count. Memory occupancy is the percentage of available GPU memory reported as used by NVIDIA's GPU monitoring system. (b) Hourly medians of five-minute 50th- and 95th-percentile TTFT estimates, computed for each Pod/model pair. The annotations highlight periods when queueing for Gemma 4 31B and GLM 5.2 represents approximately 94–100% of p95 TTFT.

breakdown also shows that work outside CRIU, including archive processing, can outweigh the time saved on model initialization.

In summary, we make the following contributions:

- We quantify the tradeoff between GPU capacity and inference readiness in a production environment by analyzing a seven-day trace of GPU compute utilization, GPU memory occupancy, and model time-to-first-token (TTFT) estimates across 26 model configurations, together with engine initialization measurements collected over one week for eight production models.
- We design a Pod-level checkpoint/restore method that coordinates snapshot creation, Pod scheduling, GPU allocation, and node-local runtime execution. The proposed method is being integrated into the core Kubernetes APIs.
- We implement node-local checkpoint and restore through a container-level protocol that uses CRIUgpu to preserve consistent CPU, GPU, container, and filesystem state.
- We evaluate TUCS across dense and mixture-of-experts models, measuring checkpoint latency, snapshot size, and time to first token.

2 Background and Motivation

Cold-start latency is a major challenge for scaling LLM inference workloads to meet changing demand. Existing work studies initialization, control-path overhead, and resource contention in production serverless systems [1], and improves process snapshot restoration through changes to storage layout and kernel support [4]. For LLM serving, Llumnix reschedules requests across model instances [15], while DistServe separates prefill and decoding [20].

CRIUgpu coordinates CPU checkpointing with driver-assisted GPU checkpointing [12]. Previous work also examines Kubernetes integration [8], model hot-swapping [13], container snapshots for forensic analysis [11], and snapshot compression [10].

PhoenixOS overlaps checkpoint/restore with GPU execution through validated speculation and evaluates serverless startup with checkpoints held in host DRAM [16]. GCR combines driver-assisted capture of GPU control state with selective interception of memory allocations to speed up data transfers, and supports incremental checkpoints [18]. Its evaluation measures GPU restoration followed by inference. Here, we focus on the container-runtime and Kubernetes operations required to restore initialized model servers. Our measurements also include processing the checkpoint archive and reconstructing the container before the first token.

2.1 LLM Inference Cold Starts

Starting a new model instance requires more than allocating compute resources. The startup path includes creating the container and serving processes, initializing the model and tokenizer, loading model weights into GPU memory, compiling kernels, capturing CUDA graphs, and warming up the server before it is ready to serve requests. Caching model files and compilation artifacts can reduce startup latency, but each new instance must still reconstruct its runtime state and repopulate GPU memory. Keeping initialized instances running avoids these costs but retains GPU resources during idle periods, while scaling them to zero releases these resources at the cost of repeated initialization [2, 17, 19].

Figure 2 summarizes 3,945 reported engine-initialization observations collected over one week. The figure includes the seven SGLang models with the most reporting worker observations and Kimi-K3, served by vLLM. Mean initialization ranges from 1.0 to 14.7 minutes, while empirical 95th-percentile durations range from 2.3 to 40.4 minutes. SGLang reports its end-to-end initialization duration; Kimi-K3 uses paired log timestamps from vLLM engine-initialization begin to completion. These timings exclude scheduling, infrastructure provisioning, image pulls, and gateway readiness. The measurements cover only completed initializations reported by the engines and do not distinguish fresh launches from checkpoint restores. Hardware and parallelism configurations are pooled within each model, so the different timing definitions and deployment conditions preclude a controlled comparison between engines. The measurements show that engine initialization alone can impose substantial delays when activating model replicas.

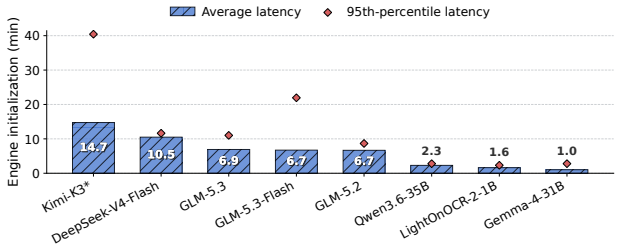


Figure 2. Cold-start latency over one week. Bars show means and diamonds empirical p95 values, pooled across configurations. *Initialization times are measured from vLLM logs for Kimi-K3 and reported by SGLang for other models.

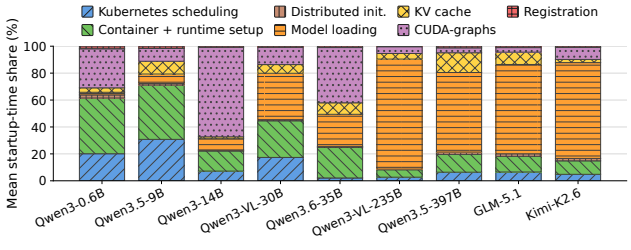


Figure 3. Breakdown of startup time from Pod creation to model registration. Each segment shows the mean percentage of time spent in one stage.

Figure 3 breaks down the path from Pod creation to model registration across representative model configurations, using production logs collected over four weeks. The breakdown shows that startup latency is distributed across multiple stages, with model loading, CUDA-graph capture, and KV-cache initialization generally accounting for a greater share of initialization in larger models.

2.2 Checkpoint/Restore State

A snapshot of an LLM inference container must preserve CPU process state, GPU state, and changes to the container's writable filesystem. CPU state includes application memory, threads, file descriptors, and inter-process communication; GPU state includes device contexts, allocations, mappings, and pending execution. The container runtime must reconstruct namespaces, mounts, GPU devices, and driver libraries, including resources configured through the Container Device Interface (CDI) for the target Pod's GPUs.

These components must be restored together because CPU processes retain references to GPU resources and depend on runtime-created devices and mounts. External connections, network identity, and load-balancer membership may need to be reestablished. The restored model must pass readiness checks before serving.

2.3 Infrastructure-Level Restore

Restoring a GPU-accelerated container requires coordination between the container runtime, process checkpointing system, GPU driver, and persistent storage. Before CRIU stops CPU processes for capture, its GPU plugin pauses GPU execution while the processes remain responsive to the driver. Without this ordering, the captured CPU and GPU state may represent different points in the model's execution.

Checkpointing involves several operations that can fail, including pausing GPU execution, capturing process and filesystem state, and storing the checkpoint. A failure must not leave the source workload paused or its GPU locked, and an incomplete checkpoint must not be made available for restore. If checkpointing fails, partial checkpoint data must be removed and the source workload returned to a well-defined state.

Restore also depends on the compatibility of the target environment with the saved checkpoint. The scheduler must place the workload on a target that satisfies the checkpoint's requirements, such as GPU count and type, device-memory capacity, device topology, and driver and runtime capabilities. An incompatible target should be detected before restore begins to avoid leaving the container in a partially restored state.

3 TUCS Design and Implementation

TUCS enables checkpoint and restore of GPU-accelerated containers by managing state across the container runtime, CPU processes, GPU driver, writable filesystem, and persistent storage. During checkpoint, CRIU and its GPU plugins coordinate CUDA locking, process suspension, and state capture to keep CPU and GPU state consistent. The snapshot becomes available for restore only after all required state has been saved to persistent storage. If any stage fails, the system discards the incomplete snapshot and marks the operation as failed.

3.1 Cross-Layer Checkpoint Protocol

Figure 4 shows the components involved in checkpointing and restoring a GPU-accelerated Pod. Kubernetes manages checkpoint metadata, Pod creation, scheduling, and GPU allocation. The kubelet invokes the container runtime, which uses CRIU and its GPU plugins to save process and device state to persistent storage and restore that state into the containers of a new Pod. The Kubernetes alpha proposal covers same-node Pod checkpoint/restore [14]. The GPU support shown here extends that scope through CRIU plugins [12].

To create a snapshot, TUCS launches a dedicated model-server instance from a Pod template. Once the inference server is fully initialized, it creates a `ready-for-checkpoint` marker file in a shared control volume. The Pod's readiness probe checks for this file, and TUCS waits for the probe to succeed before checkpointing the instance. The platform then creates a `PodCheckpoint` referencing the source Pod. The kubelet on the source node watches these requests and invokes the container runtime through the Pod-level `CheckpointPod` CRI operation. The runtime first freezes all running containers in the target Pod, then invokes CRIU to checkpoint each container. It also saves the Pod and container configuration, runtime metadata, and each container's writable filesystem layer. CRIU's GPU plugins capture device state during the same operation. The NVIDIA plugin includes this state in the captured memory pages, while the AMD plugin writes it to separate CRIU image files [12]. The kubelet provides a private directory on the node where the runtime saves the checkpoint. Once capture finishes,

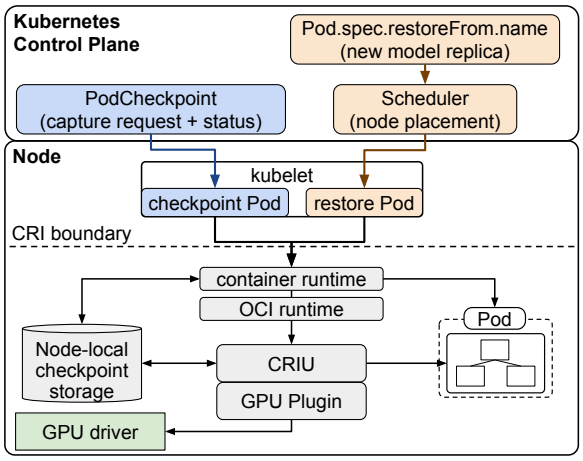


Figure 4. Architecture overview of TUCS.

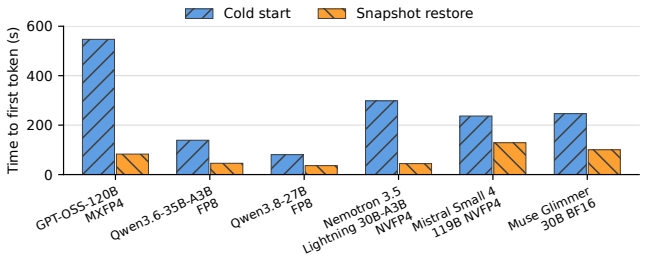


Figure 5. Cold-start and restore time to first token on B200.

the runtime resumes the source containers. The kubelet marks the `PodCheckpoint` ready after the operation succeeds. The platform can then remove the source Pod to release its GPUs.

A restore begins when a new Pod references a PodCheckpoint through `spec.restoreFrom.name`. After scheduling and GPU allocation, the kubelet calls the Pod-level `RestorePod` CRI operation. The runtime creates a new sandbox, prepares the containers and their writable filesystems, and returns their identifiers without executing the saved processes. The kubelet completes pre-start setup and calls `StartContainer` for each container. The runtime then invokes CRIU and its GPU plugins to restore CPU and device state using the source-to-target GPU mapping. The instance receives inference requests only after its serving readiness probe succeeds.

3.2 Snapshot Consistency

To keep the snapshot consistent, TUCS must prevent CPU and GPU state from diverging during checkpointing. The NVIDIA driver uses a dedicated thread inside each CUDA process to prepare the host-memory mappings required by the checkpoint. If the runtime has frozen a container, CRIU first thaws its cgroup, locks CUDA execution, and then stops the CPU processes. It temporarily resumes only the driver thread to copy GPU state into the host-memory mappings, stops it again, and saves CPU state. Because the NVIDIA driver saves CUDA state per process, CRIU identifies the CUDA processes within each container by querying the driver for each process's restore thread. For each CUDA process, CRIU uses the host-visible PID to invoke the driver and records the container-visible PID so that it can match the process after restore. For model servers that share GPU memory through CUDA inter-process communication (IPC), TUCS assigns all participating processes a shared driver checkpoint identity before CUDA initialization and saves their state sequentially. During restore, CRIU reconstructs the CUDA processes, maps their saved container PIDs to new host-visible PIDs, and restores GPU state.

3.3 Reconstructing the Pod Environment

In addition to restoring CPU and GPU state, the container runtime uses the snapshot to reconstruct the filesystems, mounts, devices, and namespaces required by each target container. For GPU access, it exposes the allocated GPUs and driver libraries inside each target container according to the new Pod's runtime configuration. CRIU also restores the contents of `/dev/shm` for the model-server processes. For networking, the runtime marks the network namespace as external to CRIU, so the processes join the namespace created for the new Pod rather than recreating the original network identity.

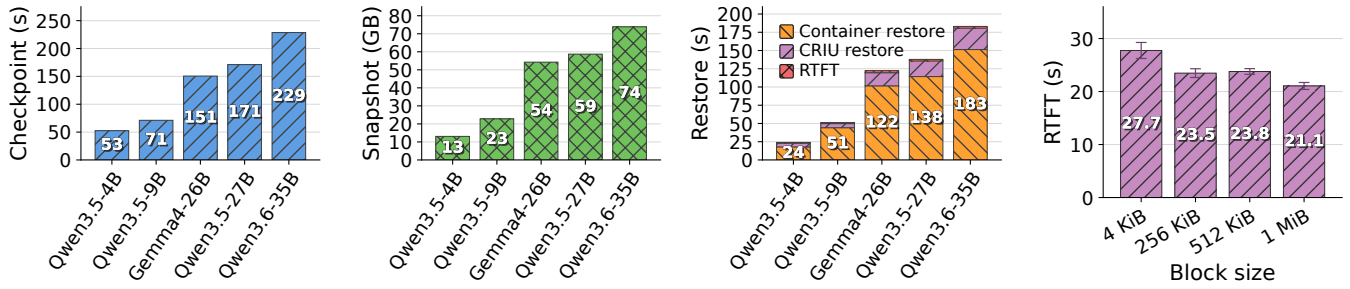
To avoid carrying external TCP connections into a new Pod, TUCS creates the snapshot before the model server establishes its request-routing and service-discovery connections. After restore, the model server establishes these connections using the new Pod's network identity. For established TCP connections between the captured processes, CRIU saves the socket state, while a TUCS plugin replaces the source Pod's IPv4 address with the new Pod's address during restore.

During restore, the runtime reads the mount mappings and exclusions, standard-I/O descriptors, writable-layer metadata, CUDA process identifiers, and source GPU identities recorded in the snapshot. To reconnect each container's root process to its new standard-I/O streams, the runtime supplies CRIU with the corresponding file descriptors. In addition, the snapshot contains files created or modified in each container's writable layer and deletion markers (overlay whiteouts) for files removed from that layer. Before invoking CRIU, the runtime applies these changes to each new container's writable layer. In contrast, the contents of bind-mounted volumes are excluded because the runtime recreates those mounts according to the new Pod's configuration.

If checkpointing fails, the runtime removes the staged data, resumes the containers it froze, and reports the failure to the kubelet. Similarly, if restore fails, the runtime removes the Pod sandbox and containers created during the operation before reporting the failure to the kubelet. The same cleanup applies if either operation exceeds the deadline set by the kubelet. Restore leaves the checkpoint unchanged.

3.4 Restore Validation and GPU Mapping

Before recreating the target containers, the runtime verifies that the checkpoint directory exists and contains a readable manifest. It obtains the new Pod's GPU allocation from the restore-time container



(a) Checkpoint latency. **(b) Snapshot size.** **(c) Restore-to-first-token latency.** **(d) Restore latency by block size.**

Figure 6. Checkpoint latency (a), snapshot size (b), and restore-to-first-token (RTFT) latency breakdown (c) across the evaluated models. In (d), we compare RTFT for Qwen3.5-4B across LZ4 block sizes using 16 decompression threads, with error bars indicating the standard error of the mean.

configuration supplied by the kubelet. Using this allocation, the runtime pairs each source GPU identity recorded in the snapshot with a target GPU. When the counts match, the runtime restores each source GPU's state to its mapped target GPU. However, TUCS's pre-restore hardware checks cover only GPU count and device mapping, so incompatibilities in GPU model, topology, memory capacity, or driver support may surface when the GPU plugin applies the saved state. The runtime also supplies CRIU with each target container's cgroup path so that the restored processes join the cgroups created for the new Pod. The design and implementation details are being discussed at the Kubernetes Checkpoint/Restore working group.[1]

4 Evaluation

To evaluate TUCS, we measure checkpoint and restore performance for five language models on an RTX PRO 6000 and six additional configurations up to 120B parameters on a B200.

Experimental Setup. We measure container-level startup and restore latency on two servers. One runs Ubuntu 22.04 with an NVIDIA RTX PRO 6000 GPU (96 GB), 16 logical CPUs from an AMD EPYC 9335 processor, and 141 GiB of host memory. It uses Linux 6.8, NVIDIA driver 610.43.02, and SGLang 0.5.16. The other has one NVIDIA B200 GPU (192 GB), 30 logical CPUs, and 180 GB of host memory, running Linux 5.15 and NVIDIA driver 615.71.09 with checkpoint archives on persistent ext4 storage. On this server, GPT-OSS and both Qwen configurations use SGLang 0.5.16, while Nemotron, Mistral, and Muse use 0.5.22. Both servers use Podman 6.0.2 and a CRIU development build based on version 4.2.1 with the CUDA plugin. We use 256 KiB LZ4 blocks and serial decompression, without additional archive compression.

Models. We evaluate the 4B, 9B, and 27B variants of Qwen3.5, together with Gemma 4 26B-A4B and Qwen3.6-35B-A3B, on the RTX PRO 6000. The six B200 configurations in Figure 5 extend the evaluation to 120B parameters. We run SGLang in eager mode for all controlled experiments.

Methodology. We use Podman to isolate container checkpoint/restore costs, with model files and checkpoint archives available locally. On the RTX PRO 6000, we run each path once to populate the filesystem cache, then collect five measurements using the same prompt, temperature zero, a fixed seed, and a 32-token limit. On the B200, we run both paths once before collecting three pairs of cold-start and restore measurements for GPT-OSS, both Qwen configurations, and Nemotron. These runs use pipeline parallelism of one (PP=1), a context limit of 2,048 tokens, and a 256-token output limit. After SGLang blocks new requests, aborts active ones, waits until idle, copies weights to host memory, and releases managed GPU memory, we time checkpointing from command invocation through archive export. Cold-start and restore latency extend from command invocation to the first streamed token. We divide restore-to-first-token (RTFT) latency into CRIU's reported restore time, remaining command time, and the interval from command completion to the first token, deriving the latter two from timestamps. These measurements exclude Kubernetes scheduling, kubelet processing, and remote snapshot transfer.

4.1 Startup and Restore Latency

On the RTX PRO 6000, mean TTFT is reduced from 61.1 to 24.0 seconds for Qwen3.5-4B and from 63.3 to 51.2 seconds for Qwen3.5-9B. Figure 6c shows the restore-time breakdown. Gemma 4 26B-A4B takes about two minutes to produce the first token with either path. Restoring Qwen3.5-27B and Qwen3.6-35B-A3B takes 11% and 19% longer than a cold start, respectively. This is because restore work outside CRIU (e.g., archive processing) outweighs the saved latency.

On the B200, restore reduces mean TTFT by 55–85% across the four configurations with three pairs of cold-start and restore measurements each (Figure 5). For GPT-OSS-120B, mean TTFT falls from about nine minutes to under a minute and a half. For Nemotron 3.5 Lightning, it falls from about five minutes to 44 seconds. Mistral Small 4 has one measured pair, while Muse Glimmer's cold-start and restore observations were collected separately.

4.2 Restore Latency Breakdown

For all five models, most RTFT occurs during the Podman restore command but outside CRIU's reported restore interval (Figure 6). This portion increases from 18 seconds for Qwen3.5-4B to 2.5 minutes for Qwen3.6-35B-A3B, while CRIU's reported restore time increases from 4.9 to 29.3 seconds. Podman processes the archive separately for runtime selection, container configuration, and state

[1] https://kubernetes.io/blog/2026/01/21/introducing-checkpoint-restore-wg

Table 1. Observed and estimated queueing for two production Qwen3.5-4B scale-from-zero events. All 28 requests were waiting when scale-up started. The estimate replaces the observed scale-up-to-live interval with the 24 seconds from starting container restore to the first token measured in our evaluation. Total queueing is reported in cumulative request-minutes.

Scale-up event	Waiting requests	Scale-up to live (s)	Restore time used (s)	Observed mean (s)	Estimated mean (s)	Observed total (request-min)	Estimated total (request-min)	Reduction
1	14	439.7	24.0	467.9	52.2	109.2	12.2	88.8%
2	14	144.8	24.0	183.7	62.9	42.9	14.7	65.8%
Combined	28	292.3	24.0	325.8	57.5	152.0	26.9	82.3%

extraction. The remainder includes archive processing, runtime operations, and GPU-state restoration after CRIU's reported interval. By comparison, the interval from Podman command completion to the first token remains between 1.2 and 2.8 seconds across the five models. In a separate experiment using PROCS [10], we evaluate Qwen3.5-4B with 16 decompression threads and LZ4 blocks from 4 KiB to 1 MiB. Increasing the block size from 4 KiB to 1 MiB reduces mean RTFT by 24%, from 27.7 to 21.1 seconds. This is a 65% reduction compared with model-server startup, which takes one minute in the same experiment.

4.3 Queueing Delay Estimate

We estimate queueing-delay savings for two production Qwen3.5-4B scale-from-zero events containing 28 requests. All requests were queued before scale-up began. Replacing each event's scale-up-to-live interval with the measured container restore time of 24 seconds, while retaining the remaining observed delay, reduces mean queueing delay from 5.4 minutes to 58 seconds (82%), saving 2.1 request-hours (Table 1). This optimistic projection assumes that container restore replaces the entire activation interval, without additional scheduling, snapshot transfer, or gateway-registration costs. It also holds subsequent request processing fixed, excluding changes in KV-cache state, routing, and batching. The estimate illustrates potential savings for these two events; production restore measurements are needed to establish the actual reduction.

4.4 Checkpoint Latency and Snapshot Size

Across the evaluated models, mean checkpoint latency and snapshot size increase from 53 seconds and 13.1 GB for Qwen3.5-4B to 3.8 minutes and 73.8 GB for Qwen3.6-35B-A3B (Figure 6). Although Gemma 4 activates only 4B of its 26B parameters for each token, its snapshot includes all model weights and reaches 54.3 GB, close to the 58.7 GB snapshot for dense Qwen3.5-27B, so storage reflects the complete model state rather than only the active parameters. Compared with snapshots created without CRIU memory-page compression, LZ4 reduces snapshot size by 12–46%, while mean checkpoint latency increases by 8% for Qwen3.5-4B, changes by less than 1% for Qwen3.5-9B, and decreases by 3–11% for the three larger models.

5 Discussion

Snapshots preserve a particular model revision, serving engine, and runtime configuration. The platform should regenerate them when these change and check compatibility after GPU or driver changes. Whether restore is faster than a cold start depends on snapshot size, storage throughput, and runtime overhead. For Qwen3.5-4B, creating a checkpoint takes 53 seconds, while restoring it saves 37 seconds compared with a cold start. After two restores, the total time saved exceeds the time spent creating the checkpoint. Storage costs and any additional transfer time must be considered separately. The platform must verify the source application's progress after a checkpoint failure before returning it to service, and snapshot files require authorization, encryption, retention, and deletion policies because they contain process and GPU memory. When several replicas restore from the same storage system, they share its bandwidth. We have not measured how concurrent restores affect activation latency or compared restore performance with cold and warm filesystem caches.

6 Conclusion and Future Work

TUCS enables checkpoint and restore of GPU-accelerated containers in Kubernetes while preserving consistent CPU, GPU, container, and filesystem state. For GPT-OSS-120B, snapshot restore reduces mean time to first token from about nine minutes to under a minute and a half. Future work will optimize archive processing in the container runtime, add earlier GPU compatibility checks, and evaluate Kubernetes restore for multi-GPU model servers.

Acknowledgments

We are grateful to Steven Gurfinkel, Jesus Ramos, and Ron Kahn for their contributions to GPU checkpoint/restore and their support for this work. We also thank Adrian Reber, Peter Hunt, Tim Allclair, Andrey Velichkevich, Kevin Hannon, David Eads, Mike Brown, David Porter, Dawn Chen, Filip Křepinský, Antonio Ojea, and Jordan Liggitt for their feedback and guidance on the design and implementation of Pod-level checkpoint/restore in Kubernetes.

References

[1] Xiaohu Chai, Tianyu Zhou, Keyang Hu, et al. 2025. Fork in the Road: Reflections and Optimizations for Cold Start Latency in Production Serverless Systems. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*. USENIX Association, Boston, MA, 199–218. https://www.usenix.org/conference/osdi25/presentation/chai-xiaohu

[2] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. 2024. ServerlessLLM: Low-Latency Serverless Inference for Large Language Models. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 135–153. https://www.usenix.org/conference/osdi24/presentation/fu

[3] Tanmaey Gupta, Sanjeev Krishnan, Rituraj Kumar, Abhishek Vijeev, Bhargav Gulavani, Nipun Kwatra, Ramachandran Ramjee, and Muthian Sivathanu. 2024. Just-In-Time Checkpointing: Low Cost Error Recovery from Deep Learning Training Failures. In *Proceedings of the Nineteenth European Conference on Computer*

Systems (Athens, Greece) *(EuroSys '24)*. Association for Computing Machinery, New York, NY, USA, 1110–1125. doi:10.1145/3627703.3650085

[4] Ben Holmes, Baltasar Dinis, Lana Honcharuk, et al. 2026. Rethinking Process Snapshots for Near-Warm Serverless Cold Starts. In *20th USENIX Symposium on Operating Systems Design and Implementation (OSDI 26)*. USENIX Association, Seattle, WA, 2007–2025. https://www.usenix.org/conference/osdi26/presentation/holmes

[5] Chiheng Lou, Sheng Qi, Chao Jin, Dapeng Nie, Haoran Yang, Yu Ding, Xuanzhe Liu, and Xin Jin. 2026. HydraServe: Minimizing Cold Start Latency for Serverless LLM Serving in Public Clouds. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*. USENIX Association, Renton, WA, 415–430. https://www.usenix.org/conference/nsdi26/presentation/lou

[6] Dharma Shukla, Muthian Sivathanu, Srinidhi Viswanatha, Bhargav Gulavani, Rimma Nehme, Amey Agrawal, Chen Chen, Nipun Kwatra, Ramachandran Ramjee, Pankaj Sharma, Atul Katiyar, Vipul Modi, Vaibhav Sharma, Abhishek Singh, Shreshth Singhal, Kaustubh Welankar, Lu Xun, Ravi Anupindi, Karthik Elangovan, Hasibur Rahman, Zhou Lin, Rahul Seetharaman, Cheng Xu, Eddie Ailijiang, Suresh Krishnappa, and Mark Russinovich. 2022. Singularity: Planet-Scale, Preemptive and Elastic Scheduling of AI Workloads. arXiv:2202.07848 [cs.DC]

[7] Viktória Spišaková, Radostin Stoyanov, Lukáš Hejtmánek, Dalibor Klusáček, Adrian Reber, and Rodrigo Bruno. 2026. Kubernetes Scheduling with Checkpoint/Restore: Challenges and Open Problems. In *Job Scheduling Strategies for Parallel Processing (Lecture Notes in Computer Science)*, Dalibor Klusáček, Julita Corbalán, and Gonzalo P. Rodrigo (Eds.). Springer Nature Switzerland, Cham, 41–62. doi:10.1007/978-3-032-10507-3_3

[8] Viktória Spišaková, Radostin Stoyanov, and Adrian Reber. 2025. Efficient Transparent Checkpointing of AI/ML Workloads in Kubernetes. KubeCon + CloudNativeCon Europe, conference presentation. https://radostin.io/files/KubeCon-2025-Efficient-Transparent-Checkpointing-of-AI-ML-Workloads-in-Kubernetes.pdf

[9] Steven Gurfinkel. 2025. CUDA Checkpoint and Restore Utility. https://github.com/NVIDIA/cuda-checkpoint.

[10] Radostin Stoyanov, Fergus Finn, Dan Feigin, Andrei Vagin, and Rodrigo Bruno. 2026. Parallel Restore-Optimized Snapshot Compression for Cost-Efficient Inference Serving. CANOPIE-HPC 2026.

[11] Radostin Stoyanov, Lorena Goldoni, Adrian Reber, et al. 2026. Forensic Analysis of Container Snapshot Chains for Post-Event Reconstruction. *Forensic Science International: Digital Investigation* 57 (2026), 302114. doi:10.1016/j.fsidi.2026.302114

[12] Radostin Stoyanov, Viktória Spišaková, Jesus Ramos, Steven Gurfinkel, Andrei Vagin, Adrian Reber, Wesley Armour, and Rodrigo Bruno. 2025. CRIUgpu: Transparent Checkpointing of GPU-Accelerated Workloads. arXiv:2502.16631 [cs.DC] https://arxiv.org/abs/2502.16631

[13] Radostin Stoyanov, Viktória Spišaková, Adrian Reber, Wesley Armour, Marcin Copik, and Rodrigo Bruno. 2025. Engine-Agnostic Model Hot-Swapping for Cost-Effective LLM Inference. In *Proceedings of the SC '25 Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC Workshops '25)*. Association for Computing Machinery, New York, NY, USA, 114–125. doi:10.1145/3731599.3767354

[14] Radostin Stoyanov, Viktória Spišaková, Adrian Reber, and Andrey Velichkevich. 2026. KEP-5823: Pod-Level Checkpoint/Restore. Kubernetes Enhancement Proposal. https://github.com/kubernetes/enhancements/blob/master/keps/sig-node/5823-pod-level-checkpoint-restore/README.md

[15] Biao Sun, Ziming Huang, Hanyu Zhao, et al. 2024. Llumnix: Dynamic Scheduling for Large Language Model Serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 173–191. https://www.usenix.org/conference/osdi24/presentation/sun-biao

[16] Xingda Wei, Zhuobin Huang, Tianle Sun, Yingyi Hao, Rong Chen, Mingcong Han, Jinyu Gu, and Haibo Chen. 2025. PhoenixOS: Concurrent OS-level GPU Checkpoint and Restore with Validated Speculation. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles* (Lotte Hotel World, Seoul, Republic of Korea) *(SOSP '25)*. Association for Computing Machinery, New York, NY, USA, 996–1013. doi:10.1145/3731569.3764813

[17] Yanning Yang, Dong Du, Haitao Song, and Yubin Xia. 2024. On-demand and Parallel Checkpoint/Restore for GPU Applications. In *Proceedings of the 2024 ACM Symposium on Cloud Computing* (Redmond, WA, USA) *(SoCC '24)*. Association for Computing Machinery, New York, NY, USA, 415–433. doi:10.1145/3698038.3698510

[18] Shaoxun Zeng, Tingxu Ren, Jiwu Shu, and Youyou Lu. 2026. GPU Checkpoint/Restore Made Fast and Lightweight. In *24th USENIX Conference on File and Storage Technologies (FAST 26)*. USENIX Association, Santa Clara, CA, 239–254. https://www.usenix.org/conference/fast26/presentation/zeng

[19] Shaoxun Zeng, Minhui Xie, Shiwei Gao, Youmin Chen, and Youyou Lu. 2025. Medusa: Accelerating Serverless LLM Inference with Materialization. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 653–668. doi:10.1145/3669940.3707285

[20] Yinmin Zhong, Shengyu Liu, Junda Chen, et al. 2024. DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 193–210. https://www.usenix.org/conference/osdi24/presentation/zhong-yinmin