Engine-Agnostic Model Hot-Swapping for Cost-Effective LLM Inference

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

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

Adrian Reber
Red Hat
Stuttgart, Germany
areber@redhat.com

Wesley Armour
University of Oxford
Oxford, United Kingdom
wes.armour@eng.ox.ac.uk

Marcin Copik
ETH Zurich
Zurich, Switzerland
marcin.copik@inf.ethz.ch

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

Abstract

The widespread adoption of Large Language Models (LLMs) has led to an increased demand for large-scale inference services, presenting a unique set of challenges for the HPC community. These services are characterized by moderate-scale models that require dedicating expensive GPUs to handle bursty inference requests, leading to high costs and resource underutilization. In this paper, we propose SwapServeLLM — a novel engine-agnostic hot-swapping method for cost-effective inference. This model hot-swapping approach is enabled by recent driver capabilities for transparent GPU checkpointing. SwapServeLLM optimizes resource utilization by dynamically allocating GPU resources with two key mechanisms: (1) a demand-aware preemption leveraging information about concurrent requests, and (2) efficient request routing with memory reservation minimizing inference latency. Our evaluation demonstrates that SwapServeLLM optimizes model loading for state-of-the-art inference engines by 31× compared to vLLM and up to 29% compared to Ollama, enabling cost-effective inference.

CCS Concepts

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

Keywords

Cloud Computing, Containers, LLM Inference, GPU Checkpointing

ACM Reference Format:
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 *Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC Workshops '25), November 16–21, 2025, St Louis, MO, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3731599.3767354

*Also with Red Hat.

SC Workshops '25, St Louis, MO, USA

ACM ISBN 979-8-4007-1871-7/25/11
https://doi.org/10.1145/3731599.3767354

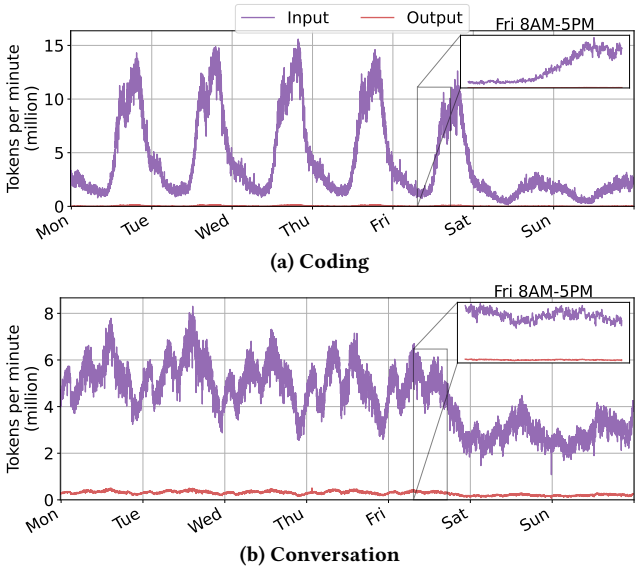
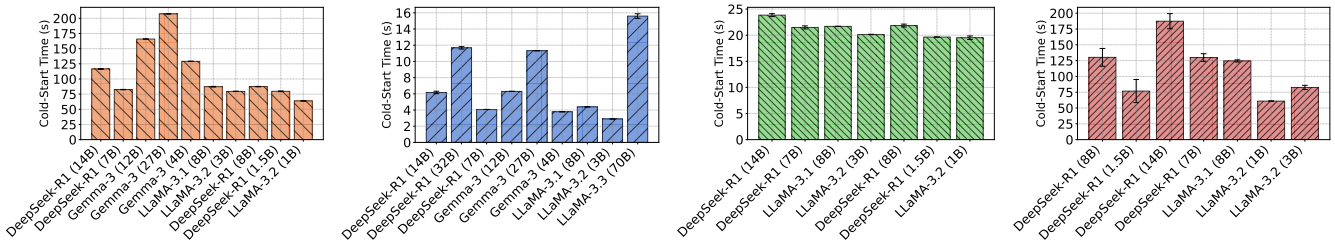


Figure 1: LLM token volume analysis illustrating the number of input (context) and output (generated) tokens over a week for Coding and Conversational workloads in Microsoft Azure [53], with a zoomed-in view highlighting the daily usage patterns of a typical workday (Friday, 8 AM – 5 PM).

1 Introduction

The increasing popularity of Large Language Models (LLMs) is driving unprecedented demand for inference services across different domains, from conversational bots [4, 17, 30, 37] and customer support agents [65] to programming assistants [21, 46]. These inference services host multiple models running on expensive GPU accelerators [9, 27, 58, 59] and often experience unpredictable bursts of inference requests that fluctuate throughout the day [29, 41, 57, 64], as illustrated in Figure 1. Even if invocations can be anticipated, the compute and memory requirements are highly input-dependent. Requests with a large number of input tokens and a small amount of output tokens are compute-intensive, while those with few input tokens and many output tokens are memory-bound [53, 72].



(a) vLLM (b) Ollama (c) SGLang (d) TensorRT-LLM

Figure 2: Cold-start latency including container startup and model loading on H100 (80 GB HBM3).

Serverless platforms such as SageMaker [3], KServe [8], and Hugging Face [12] have emerged as a promising solution that allows developers to upload their models and configurations, while provisioning of the underlying infrastructure is fully automated. This approach aims to reduce operational costs by sharing compute resources between multiple workloads and avoiding expensive long-term GPU reservations. However, a key challenge for these serverless inference workloads is the attainment of strict latency Service Level Objectives (SLOs) while minimizing operational costs to remain economically viable [19, 69].

Furthermore, the autoregressive nature of inference workloads, where each output token is generated sequentially based on previously generated tokens, coupled with the unpredictable output lengths, makes it challenging to accurately estimate the computational requirements of each request [32, 40, 49]. This challenge is further exacerbated by the heterogeneity across LLM models, GPU architectures, and the variability of request types with inputs ranging from short prompts to long documents [19, 20].

To adapt to fluctuating load, serverless platforms dynamically adjust the number of inference engine instances for serving requests (i.e., scaling in and out). However, the overheads of starting a new instance are often intolerable if performed on the critical path of request serving [10, 22, 48, 74]. As shown in Figure 2, these overheads can take from tens of seconds to a few minutes across inference engines. The cold-start latency of inference engines is prolonged due to the large model sizes [6, 27] and high-throughput optimizations that require complex initialization phase [23].

A practical solution proposed in the literature [69, 73] and adopted by inference engines such as Ollama [31], relies on dynamic loading of LLM models into GPU memory. However, these mechanisms are tightly integrated with the GPU memory management functionality of the inference engine and introduce additional overhead in the critical path of request serving. These limitations prevent high-throughput LLM engines such as vLLM [23] and TensorRT-LLM [35] from easily adopting dynamic model loading.

An alternative approach of allocating dedicated GPUs has been introduced in many cloud systems [62, 72, 73]. As shown in Figure 3, this approach often results in continuously reserved GPU resources [51], while actual compute utilization remains low due to insufficient incoming requests to fully utilize the allocated GPUs. Consequently, service providers encounter a trade-off between adopting a cost-effective serverless approach with high cold-start latencies, and allocating dedicated GPU resources with continuously running inference engines to meet SLO requirements.

In this work, we explore a novel engine-agnostic approach to LLM model hot-swapping for cost-efficient inference serving. Our approach leverages NVIDIA's driver capabilities for transparent checkpoint/restore of GPU workloads [52, 55, 56], in combination with the Linux cgroup freezer mechanism [13, 43], to suspend and resume containerized applications. This approach enables transparent host-to-GPU *hot-swapping* of inference workloads that works across different models and engines [54]. We design and implement SwapServeLLM by integrating this hot-swapping mechanism with an OpenAI-compatible request router, a demand-aware preemption policy, and GPU memory reservation mechanism to provide an engine-agnostic framework for efficient LLM serving. This framework provides a platform that reduces cold-start latency and improves GPU utilization while optimizing the cost-efficiency of multi-model deployments across heterogeneous inference engines.

We evaluate SwapServeLLM with a state-of-the-art inference engines (vLLM [23], Ollama [31], SGLang [75], and TensorRT-LLM [35]) and using a set of LLaMA, DeepSeek, and Gemma models with different sizes, architectures, and quantization running on A100 (SXM4 80GB) and H100 (HBM3 80GB) GPUs. Our evaluation results show that for larger DeepSeek-R1 14B FP16 models, SwapServeLLM is about 29% faster than the cold-start latency of baseline Ollama, while with smaller LLaMA 3.2 1B FP16 model, it is approximately 2.6× faster. When compared to cold-start latencies with vLLM, SwapServeLLM significantly reduces the overhead associated with model loading, resulting in a speedup of approximately 18× to 31× faster than cold-start times, depending on the model size and GPU memory usage. These results show that SwapServeLLM enables cost-efficient inference deployments with larger number of LLM models running on fewer GPUs. Our framework is open source and available at https://github.com/rst0git/SwapServeLLM.

In summary, we make the following contributions:

- Characterize the initialization phase and model loading methods of state-of-the-art inference engines and analyze the trade-offs and limitations of different strategies for efficient model serving (§2).
- Propose SwapServeLLM, an engine-agnostic model hot-swapping framework for LLM inference (§3).
- Introduce a dynamic memory reservation mechanism and demand-aware preemption policy for efficient request serving with SwapServeLLM (§4).
- Evaluate SwapServeLLM with four inference engines (vLLM, Ollama, SGLang, TensorRT-LLM) and a set of DeepSeek, LLaMA, and Gemma models with different architectures, sizes, and quantization (§5).

2 Background and Motivation

In this section, we begin by highlighting the challenges with model loading for inference serving. We then provide an overview of the methods for efficient and on-demand model loading.

2.1 Challenges with Inference Serving

With the proliferation of specialized LLM models, efficient multiplexing of inference requests has become crucial [28]. For example, organizations and inference providers maintain multiple models designed for tasks such as math and science reasoning, handling of text and images, faster inference, as well as code analysis and generation. However, in today's commercial platforms, inference serving with custom, fine-tuned models is more expensive than base models due to several challenges that lead to increased memory footprint and low resource utilization [20, 38, 68].

Inefficient GPU Utilization. A naive approach for inference serving with multiple LLM models is to dedicate a group of GPUs for each model. As shown in Figure 3, this approach is particularly inefficient when the number of inference requests is sporadic and low in volume, as it limits the batch size and leads to high operational costs. Efficiently allocating and distributing workloads across GPUs is crucial to avoid resource underutilization and enable cost-efficient inference serving. In some cases, maximizing GPU utilization requires assigning models to suboptimal GPU configurations. Although this may not be ideal for individual models or requests, it ultimately improves the overall performance [20].

High Model Loading Latency. Conventional model loading approaches incur significant overheads due to the expensive initialization phase for high-throughput LLM inference, as shown in Figure 2. For instance, loading LLaMA 3.1-8B takes 87 seconds with vLLM and 124 seconds with TensorRT-LLM on H100 GPU. This overhead far exceeds the latency requirements for token generation, with industry target for high-quality interactive experience typically below the average human reaction time in the range of 200-250 ms [41]. The unpredictable nature of model invocations further prevents effective prefetching as it is hard to predict when a request for particular model will arrive. These limitations result in prolonged first-token latency and inefficient GPU utilization.

LLM Serving with Heterogeneous Engines. Inference service providers often need to support multiple engines and configurations to optimize for different use-cases (e.g., multi-turn conversions, question answering, text summarization), as well as performance requirements and hardware compatibility across a range of models and deployment environments [11]. The distinct characteristics of each engine stem from differences in the core design and architecture. However, the memory management, scheduling and batching strategies of multiple engines are often incompatible and this leads to inefficient resource sharing [20, 63].

Autoscaling Inference Engines. Scaling LLM inference presents unique challenges, as demand spikes require rapid GPU provisioning and session-aware load balancing. Although GPU utilization is conventionally used as a metric for automatic scaling, workload-specific performance metrics such as batch size, queue size, and decode latencies have been shown to be more effective for autoscaling of LLM inference engines [16]. These metrics are often more reliable for traffic fluctuations across a range of GPU hardware setups. However, the high cold start latencies of LLM engines remain a challenge, significantly reducing elasticity [14, 53].

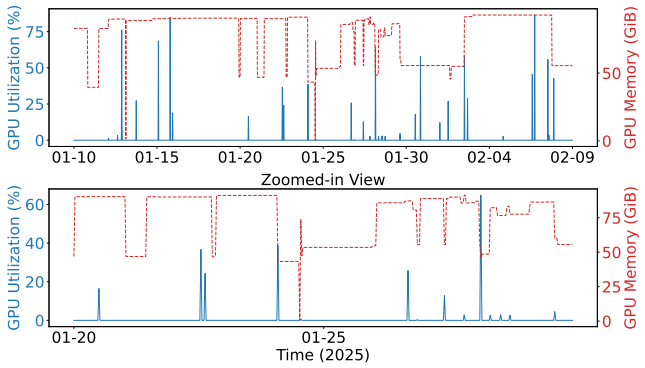


Figure 3: The GPU utilization and memory consumption during serving of six models in the e-INFRA CZ Kubernetes cluster with single NVIDIA H100, used by a small group of academics and researchers over a month.

2.2 Efficient Model Loading for Inference

The massive size of LLM models creates a bottleneck in both I/O and CPU-bound operations, making efficient model loading crucial for inference engines to minimize cold-start latency. Modern formats such as *SafeTensors* [18] enable *memory-mapping* (zero-copy) optimizations, allowing the model weights to be loaded directly into GPU memory without an intermediate copy. When these techniques are combined with *model sharding* (splitting the model into multiple smaller files) and *parallel loading*, they significantly reduce I/O bottlenecks and maximize the utilization of available GPU bandwidth. To further reduce cold-start latency, techniques such as *quantization* are often integrated to compress the model size through lower-precision data types. While these techniques reduce I/O bottlenecks, runtime optimizations such as *graph capture* (recording GPU operations into CUDA or HIP graphs) [33, 34], *just-in-time* (JIT) compilation (compiling optimized kernels specific to the model and hardware) [36], and *speculative decoding* (using a smaller auxiliary model to propose token sequences for parallel validation) [24], introduce significant initialization overheads. This results in a fundamental tradeoff between minimizing cold-start latency and achieving high-throughput inference, as the cost of these runtime optimizations is typically too large to be tolerable on the critical path of serving requests [53].

2.3 On-Demand Model Loading

On-demand model swapping enables inference engines to serve multiple models concurrently with limited GPU memory by dynamically loading requested models. Inference engines such as Ollama [31] adopt this mechanism through llama.cpp *runners* [15]. Each runner handles requests for a single model and aims to maximize availability without requiring all models to fit in memory simultaneously. When the available GPU memory is insufficient to load a model, the Ollama scheduler selects which models to unload.

The scheduler continuously tracks all active runners and prioritizes unloading least recently used (LRU) runners. This approach enables multi-model serving of concurrent requests by keeping only active models in GPU memory. However, it relies on the fast model loading and initialization of llama.cpp, which comes at the cost of sacrificing runtime optimizations. An initial performance analysis of Ollama by Red Hat researchers [60] shows that this leads to significantly lower throughput when compared to highly optimized inference engines such as vLLM [23] and TensorRT-LLM [35]. Thus, realizing cost-efficient inference serving requires addressing the aforementioned challenges while maintaining runtime optimizations.

3 Engine-Agnostic Model Swapping

At a high level, our goal is to provide an efficient, engine-agnostic mechanism that dynamically loads and swaps LLM models on demand while preserving the performance benefits of optimizations such as CUDA graph capture and JIT compilation. We enable hot-swapping via transparent GPU checkpointing [56] by creating inference engines snapshots immediately after LLM models have been initialized. This approach is decoupled from specific engine implementations and allows to resume inference serving upon request arrival without the costly overhead of reinitialization.

3.1 System Overview

Figure 4 shows the system architecture of SwapServeLLM. It consists of the following main components: OpenAI API router, request handler, model workers, scheduler, task manager, GPU monitor, and LLM engine controller. The *router* provides an OpenAI-compatible API endpoint that acts as a proxy, multiplexing inference requests for multiple models and inference engines (backends). On arrival, each inference request is added to a queue for processing by the *request handler*. The request handler is responsible for accepting incoming requests, creating response channel objects, and enqueuing both to a backend queue. Each backend has a *model worker* instance that reads from its queue, and forwards requests and responses between the inference engine and connected clients. If the backend has been swapped out when an inference request arrives, the worker issues a request to the *scheduler* to resume its execution. The scheduler then reserves the required GPU memory with the *task manager*. When necessary, the task manager identifies suitable candidates for preemption. It uses the *GPU monitor* to observe memory utilization and inform the scheduling decisions. Once the requested GPU memory becomes available, the scheduler triggers a swap-in operation via the *engine controller*. The engine controller then restores the backend GPU state and resumes its execution, allowing to continue processing requests without incurring the overhead of full reinitialization.

3.2 SwapServeLLM Initialization

The initialization phase of SwapServeLLM begins with loading of a configuration that specifies runtime parameters and list of models. Each model configuration is then validated to verify that all required parameters have been specified and supported by SwapServeLLM.

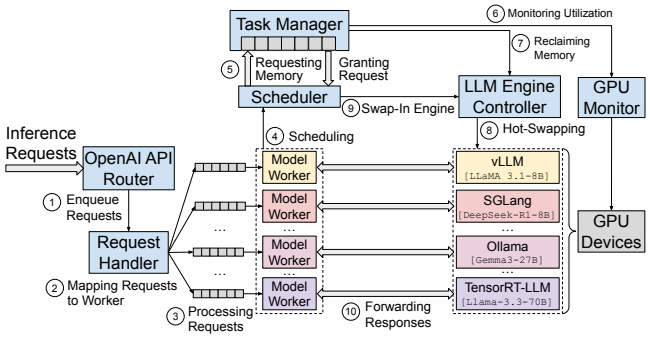


Figure 4: An overview of the SwapServeLLM architecture. Modules in boxes with dashed border are on the critical path of serving requests. All other modules do not impact serving latency of models loaded in GPU memory. Numbered circles correspond to the life-cycle of serving inference requests.

Once the configuration has been validated, SwapServeLLM initializes the task manager and GPU monitor, obtaining information about the system continuously monitoring GPU utilization. SwapServeLLM then initializes a worker and request queues for each model, as well as inference engine backends as individual containers. Each model is configured with *global* and *local* parameters. Global parameters include engine-specific options such as response timeout, KV cache type, and authentication tokens, while local parameters are model-specific options such as model name, container image, GPU memory utilization, and initialization timeout. During the initialization phase, SwapServeLLM creates and runs a container for each configured model. Once the inference engine and model have been fully initialized, it creates an in-memory snapshot of the GPU state and leaves the container in a paused state. Information about container instances (e.g., unique identifier, IP address, published TCP port) are stored in an index data structure that during inference efficiently maps the model name to model worker and inference backend. Once the task manager, model workers, and inference backends have been fully initialized, SwapServeLLM starts the request handler and OpenAI API router to begin serving requests.

3.3 Model Hot-Swapping Workflow

To address the challenges in Section 2, SwapServeLLM continuously monitors the activity and resource utilization of LLM models and selectively hot-swaps inference engines in and out of GPU memory to effectively serve inference requests.

When incoming requests arrive for a model whose engine is not currently running (i.e., not loaded in GPU memory), the OpenAI router and the request handler validate and accept the incoming request (①). The request handler checks if the corresponding backend has available queue capacity and creates a response channel with the associated metadata (②). The model workers actively poll requests from their queue (③), and if on request arrival the backend is *not running*, the worker makes a *swap-in* request to the scheduler (④). The scheduler then obtains the required GPU memory needed to resume the model, and makes a reservation request to the task manager (⑤). The task manager continuously monitors

the available GPU resources (⑥) and keeps track of memory reservations with priority queue. The task manager is also responsible for reclaiming GPU resources to accommodate new requests (⑦). If not enough GPU memory is available, the engine controller applies demand-aware preemption policy to select the best candidates to *swap-out* (⑧). Once enough GPU resources become available, the memory reservation request is granted by the task manager. The scheduler then triggers a *swap-in* operation for the corresponding backend (⑨). When the swap-in operation completes, the model worker verifies that the inference engine API is active, forwards the enqueued request, and begins relaying responses directly to the client, avoiding the overhead of request processing (⑩).

3.4 Model Swapping Mechanism

When an inference request arrives and the corresponding backend is not currently in a running state, a swap-in operation is used to resume LLM serving without full model reinitialization. This swap-in operation is performed asynchronously by model workers, requesting the required GPU memory (previously saved during the swap-out operation) via the task manager. The task manager utilizes a priority queue for memory reservations to ensure that multiple requests can be handled concurrently. For example, if SwapServeLLM is running on a server with a single `A100` GPU with 80GB of memory, and inference requests for Gemma 7B (requiring 16GB) and DeepSeek Coder 6.7B (requiring 14 GB) arrive at the same time, both swap-in operations can be performed as sufficient memory is available for both models to reside on the GPU. If a request for LLaMA 3.3 70B FP8 (requiring 75 GB) subsequently arrives, the task manager must enqueue the swap-in operation and swap-out both Gemma and DeepSeek models to free the required GPU memory before granting the memory reservation request.

3.5 Preemption Policy

Whenever a *swap-in* request for a model cannot be fulfilled due to insufficient GPU memory, the task manager invokes a demand-aware preemption policy implemented by the engine controller. This policy iterates over currently running backends, using a two-tiered hybrid metric to select candidates for eviction. The first tier prioritizes backends with shorter request queues that are less likely to disrupt ongoing user interactions. If multiple backends have equal queue lengths, the second tier is applied. This tier prioritizes candidates based on the oldest *last-accessed* time, effectively implementing a traditional *least recently used* (LRU) tie-breaker.

Each candidate for preemption is write-locked immediately before eviction to prevent race conditions with concurrent request handling. This locking mechanism ensures that no new requests are forwarded to the inference engine during the swap-out operation. After completing each swap-out operation, the task manager checks the available GPU memory and, once the memory requirement is satisfied, grants the memory reservation request for the swap-in operation. If insufficient memory is available after a swap-out operation, the task manager attempts to evict the next best candidate for preemption. This preemption policy allows SwapServeLLM to dynamically adapt to workload patterns and to maximize throughput while minimizing latency when serving multiple models with constrained memory resources.

4 Implementation

We implement SwapServeLLM as a serverless LLM framework built using the Go bindings for Podman [44], an open-source tool for managing containerized workloads. This framework incorporates the system components described in Section 3 and provides backends for vLLM [23], Ollama [31], SGLang [75], and TensorRT-LLM [35]. We selected these engines to cover a wide range of deployment use-cases: vLLM for high-throughput inference with optimized memory management, Ollama for lightweight model hosting, SGLang for structured generation and control over outputs, and TensorRT-LLM for low-latency serving in production environments. The request routing, queuing, and scheduling mechanisms are detailed in Section 4.1, while additional information on the hot-swapping mechanism and concurrent requests handling is provided in Section 4.2.

4.1 API Router and Request Handler

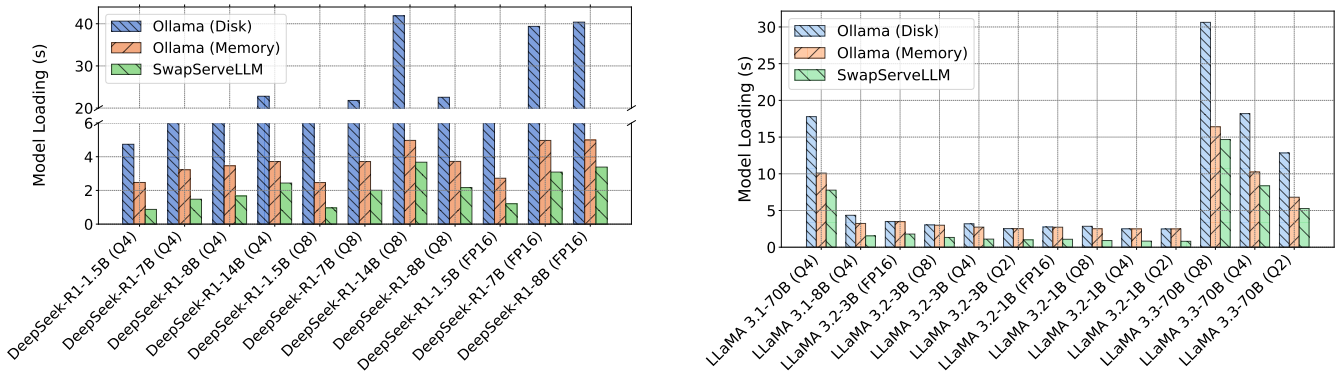
The SwapServeLLM API router is designed to efficiently serve concurrent requests across multiple inference engines and LLM models. This component is implemented using Gin [26], a high-performance web framework developed in Go, and provides an API endpoint compatible with the OpenAI specification [39]. It adopts a non-blocking, asynchronous architecture that effectively decouples routing, request handling, and response generation to improve scalability, reduce latency and maximize throughput. Upon receiving a request, the router processes and validates its payload. It then extracts the requested model name, verifies the corresponding backend availability, and queues the request for processing.

The request handler saves arrival timestamps and updates the last-accessed metadata of each backend to keep track of utilization. It then creates a dedicated response channel using an asynchronous mechanism that enables low-latency communication between the inference engine and the client. This channel allows for non-blocking transmission of responses with real-time streaming. The request handler then encapsulates the inference request, response channel, and relevant metadata into an object that is enqueued in a model-specific queue, as illustrated in Figure 4.

A per-model worker continuously monitors each queue, and when requests arrive, it verifies that the client connection is still active (i.e., not canceled due to disconnection), and forwards the requests to the inference engine. This approach allows concurrent processing of multiple requests, handling cancellations and timeouts, and maximizing overall throughput, while the API router and request handler accept and dispatch incoming requests without blocking. If the backend for the requested model is not in a running state, the model worker coordinates with the scheduler, task manager and engine controller. These components use per-model synchronization mechanisms to reserve the required GPU memory and perform swap-in operations concurrently.

4.2 On-demand Model Hot-Swapping

The hot-swapping functionality of SwapServeLLM is designed for efficient GPU memory management through dynamic loading and unloading of LLM inference engines. This mechanism ensures that LLM models reside in GPU memory only when actively needed and swapping them in with either explicit API calls or incoming inference requests.



(a) DeepSeek-R1 models **(b) LLaMA models**

Figure 5: Comparison of Ollama model cold-start latencies when loading from disk, memory, and SwapServeLLM in-memory snapshots on an NVIDIA A100 (SXM4 80 GB) server.

Model Preemption. When GPU memory is insufficient to perform a swap-in operation, the task manager utilizes a demand-aware LRU policy to select candidates for preemption. This preemption operation utilizes a locking mechanism to signal the model worker to stop forwarding inference requests. The task manager then suspends CPU execution of the inference engine via the Linux cgroup freezer mechanism and saves the amount of GPU memory in use. It then creates an in-memory snapshot using the CUDA checkpoint functionality, thus freeing GPU capacity for other workloads. This preemption mechanism is further accelerated with engine-specific optimizations, such as vLLM's `sleep` API [61]. These optimizations efficiently unload the model weights to CPU memory and reduce both GPU state size and checkpoint/restore latency.

5 Evaluation and Analysis

In this section, we evaluate the performance of SwapServeLLM with a set of models and inference engines. Our evaluation aims to answer the following question: *How does the latency of on-demand model hot-swapping with SwapServeLLM compare to the cold-start latency of inference engines?*

To address this questions, we first establish a performance baseline by characterizing the cold-start latencies of several inference engines (§5.2). We then measure the swap-in latency of SwapServeLLM to demonstrate the efficiency of the model hot-swapping mechanism (§5.3).

5.1 Experiment Setup

We use two server configurations to evaluate the performance of SwapServeLLM. For the experiments in Figure 5, we use a system with an NVIDIA A100 GPU (SXM4 80GB), 12-core Intel Xeon Gold 6342, 1 TB SSD, running Ubuntu 22.04 with kernel v6.2, CUDA 12.8, and driver v570.86. For cold-start and swap-in latency measurements in Figure 6 and Table 1, we use a server equipped with an NVIDIA H100 GPU (HBM3 80GB), 26-core Intel Xeon Platinum 8480, 221 GB RAM, 2.8 TiB storage, running Ubuntu 22.04 with kernel v6.8, Podman v5.7, runc v1.3, CUDA 13 and driver v580.65.

Models. We evaluate SwapServeLLM with a set of LLaMA, DeepSeek, and Gemma models with varying sizes, architectures, and quantization levels. For Figure 2, Figure 6 and Table 1 we used vLLM v0.9.2, SGLang v0.4.9, Ollama v0.9.6, and TensorRT-LLM v1.0rc0 inference engines. To obtain the evaluation results in Figure 5 we used Ollama v0.5.7.

Language Model Parameters. To ensure deterministic and reproducible results, we set a fixed *temperature* and *seed* parameters for all inference requests. This temperature parameter controls the randomness (creativity) of the model's output and influences how it chooses among possible next tokens when generating text. By setting the temperature value of all requests to 0, we instruct the model to always pick the most likely next token and generate consistent output.

5.2 Impact of Cold-Starts on Inference Latency

To establish a baseline for our evaluation, we first characterize the cold-start latency of four state-of-the-art inference engines: vLLM, Ollama, SGLang, and TensorRT-LLM, shown in Figure 2. These results include both container startup and model initialization times, highlighting cold-start latency across inference engines, ranging from tens of seconds to a few minutes. For instance, loading LLaMA 3.1-8B takes 4.38 seconds with Ollama, 21.68 seconds with SGLang, 87.28 seconds with vLLM, and 124.48 seconds with TensorRT-LLM.

We further analyze a detailed breakdown of the initialization times for vLLM across various models. We chose vLLM as it provides a set of optimizations for high-performance LLM serving that balance memory efficiency, compute acceleration, and hardware compatibility. The results in Table 1 show that runtime optimization mechanisms, such as PyTorch kernel compilation and CUDA graph capturing, are the two largest contributors to the overall initialization latency. As the model size increases, the times for these two optimization steps also increase. These results highlight the impact of runtime optimization mechanisms on the prolonged cold-start latency of inference workloads.

In Figure 5, we analyze the model loading latency for several DeepSeek-R1 and LLaMA models with Ollama and SwapServeLLM on `A100` GPU. In comparison to vLLM, SGLang, and TensorRT-LLM, the Ollama inference engine is highly optimized for local deployments on consumer hardware and efficient use of limited GPU memory. Thus, we chose this inference engine as the strongest baseline as it has the lowest cold-start latency across models. For

these experiments, Ollama is configured with a default disk storage as well as a memory-backed filesystem. The results show that, across all models and quantization levels, the memory-backed storage configuration outperforms the disk-based one, but remains slower than SwapServeLLM. For the smallest model, DeepSeek-R1 1.5B, Ollama with loading from disk ranges from 4.7 to 11.3 seconds, when loading from memory it is 2.46 to 2.72 seconds, and SwapServeLLM loading mechanism is from 0.87 to 1.21 seconds. This represents a 70–90% latency improvement compared to disk, and a 50–60% when compared to memory. Medium size model, such as DeepSeek-R1 7B and 8B, show disk latencies of approximately 13 - 40 seconds, memory latencies of 3.2–5 seconds, and SwapServeLLM latencies of 1.48–3.39 seconds. These results reflect similar latency improvement. Large models, such as DeepSeek-R1 14B, have disk latencies between 22.8 and 41.9 seconds, memory latencies of 3.7–5 seconds, and SwapServeLLM latencies of 2.44–3.68 seconds. These results indicate 80–90% and 25–35% improved latency when compared to Ollama model loading from disk and memory, respectively. These results also highlight the significant impact of quantization level on loading times. For instance, models with a lower bit-width (e.g., Q4) are smaller in size and thus load considerably faster than those with a higher bit-width (e.g., Q8) or unquantized FP16 models.

5.3 Hot-Swapping Latency of SwapServeLLM

We evaluate the hot-swapping effectiveness of SwapServeLLM by measuring the swap-in latency when serving requests with vLLM and Ollama inference backends. In comparison to the cold start latency shown in Figure 2, this mechanism avoids the initialization phase of the inference engine and LLM model, and resumes inference serving as described in Section 3.

As illustrated in Figure 6a, the swap-in latency with the vLLM backend ranges from about 5.5 to 7.5 seconds. The corresponding models utilize between 72 and 73 GB of GPU memory. These results reveal a correlation between the size of occupied GPU memory and the swap-in latency: smaller models like LLaMA-3.2 (1B FP16) have swap-in latency of 5.5 seconds, whereas larger models, such as DeepSeek-R1 (14B FP16), incur swap-in latencies of 7.5 seconds. In comparison, vLLM cold-start times for these models range between 1 minute 41 seconds and 2 minutes 53 seconds, respectively. This indicates that SwapServeLLM significantly reduces the overhead of model loading compared to cold-start times, and the time needed to resume inference serving primarily depends on the GPU memory utilized by the inference engine. In Figure 6b, we analyze and compare the swap-in latency of SwapServeLLM with the model loading times of Ollama. These results show that SwapServeLLM consistently outperforms Ollama across different model sizes, with swap-in latency in the range of 4.6 seconds for DeepSeek-R1 (14B FP16) and 0.75 seconds for LLaMA 3.2 (1B FP16). The amount of GPU memory utilized by these models is 30.5 GB and 3.6 GB, with model loading times of 5.93 and 1.96 seconds, respectively.

6 Discussion

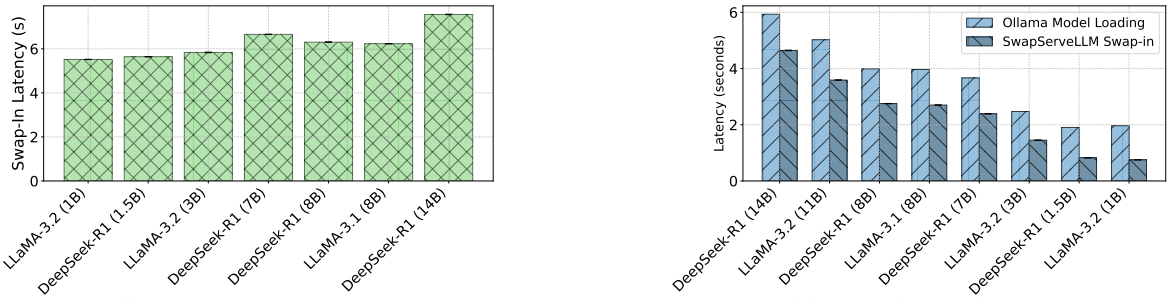
The evaluation results of SwapServeLLM show a significant reduction in model loading latency, with approximately 18× to 31× speedup over vLLM and up to 29% compared to Ollama. This approach allows platforms to keep the benefits of high-performance engines, such as vLLM and TensorRT-LLM, with high elasticity by quickly instantiating models through hot-swapping.

Model	Total (s)	Load (s)	Compile (s)	CG (s)
DS-14B	82.39	5.17	43.18	21.00
DS-8B	55.17	3.05	29.13	17.00
DS-7B	51.03	2.88	26.58	16.33
DS-1.5B	49.81	1.01	26.52	16.00
G3-27B	160.30	9.11	79.67	32.33
G3-12B	123.71	4.35	63.42	27.00
G3-4B	89.26	1.91	47.50	22.00
L3.1-8B	55.41	3.11	29.33	17.00
L3.2-3B	49.41	1.48	26.38	16.00
L3.2-1B	34.14	0.85	16.85	14.00

Table 1: Baseline vLLM initialization time breakdown (in seconds) for DeepSeek (DS), Gemma (G3), and LLaMA (L3) models. *Total* is the full engine initialization time. *Load* refers to loading model weights. *Compile* is the *torch.compile* duration. *CUDA Graphs* (CG) is the capturing time.

The key to these performance gains is the ability of SwapServeLLM to bypass the time-consuming steps of engine and model initialization (e.g., PyTorch compilation and CUDA graph capture). As shown in Table 1, these two steps are the largest contributors to overall initialization latency for our vLLM baseline. By using transparent GPU checkpointing to create an in-memory snapshot of the GPU state immediately after a model has been initialized, SwapServeLLM can restore the engine upon request without repeating these costly phases. This approach enables hosting of a larger set of LLM models on a reduced number of GPUs in multi-tenant environments, thus mitigating the need for over-provisioning and long-term GPU reservations. Such optimizations directly translate into lower operational costs and improved resource utilization.

Multi-GPU Orchestration. As the size of LLM model weights continues to grow, leveraging multiple GPUs becomes essential to meet the computational requirements of inference services. Single model inference on multi-GPU systems is typically achieved through parallelization strategies such as tensor parallelism (TP), pipeline parallelism (PP), data parallelism (DP), or hybrid approaches combining these techniques. To effectively handle resource sharing in multi-GPU systems, SwapServeLLM defines the GPU topology of each inference engine (backend) during initialization, when GPU resources are allocated. The subsequent preemption and swap-in operations use similar memory reservation methods for each GPU, as described above. To prevent memory overcommitment during model swapping, memory reservations are managed using scoped acquire-release semantics. Each model swapping operation issues a *reservation request* followed by corresponding *memory release* signal, propagated upon eviction, where the task manager is responsible for handling pending signals and allocations. This approach maximizes resource utilization by enabling model hot-swapping while avoiding contention across shared resources and costly initialization overhead.



(a) Swap-in latency with vLLM backend. **(b) Ollama (baseline) vs SwapServeLLM.**

Figure 6: (a) On-demand swap-in latency with vLLM and (b) comparison between Ollama model loading and SwapServeLLM.

7 Related Work

Serverless inference systems. Platforms such as AWS Lambda [47], Azure Functions [5], and Alibaba Function Compute [2], as well as open-source systems such as KServe [8] provide support for serverless inference. Many academic works attempt to optimize inference serving in such serverless platforms [1, 7, 14, 25, 42, 45, 67, 71]. However, these systems focus primarily on resource efficiency and scheduling, avoiding the cold-start problem or using caching policies, multi-tier storage systems, or predictive strategies to mitigate it. In contrast, SwapServeLLM is the first work that focuses on reducing the model loading times during inference serving with containerized engine-agnostic hot-swapping.

Cold-start optimizations. Previous research explores mechanisms for model swapping between CPU and GPU memory using mechanisms however for SLO-aware scheduling and queuing policies with asynchronous GPU API redirection [50, 69, 70, 73]. These approaches are implemented atop the Alibaba Cloud and SnowFlake commercial serverless platforms and rely on proprietary infrastructure features. In contrast, SwapServeLLM is built on open-source container runtime and integrates with state-of-the-art inference engines. There are prior works on GPU runtimes implementing cold-start optimizations that selectively restore essential states from checkpoint images [66]. However, these works focus on general-purpose snapshotting mechanism for serverless functions that captures both CPU and GPU states. In contrast, SwapServeLLM is designed for accelerating the model loading times of LLM inference serving workloads and leverages engine-specific optimizations.

8 Conclusion

The inherent challenges of inference serving in traditional deployments often lead to resource over-provisioning, inefficient GPU utilization, and high operational costs. In this paper, we propose SwapServeLLM, a framework for cost-efficient LLM serving with an engine-agnostic model hot-swapping. SwapServeLLM leverages recent driver capabilities for transparent GPU checkpointing, along with a demand-aware preemption policy to enable dynamic swapping of LLM models in and out of GPU memory. Our evaluation results demonstrate that SwapServeLLM achieves a significant reduction in model loading latencies approximately 18× to 31× faster than vLLM and 29% faster than Ollama. These optimizations enabled with SwapServeLLM allow operators to deploy a larger number of models with fewer GPUs, and to improve resource utilization in multi-tenant environments.

Acknowledgments

We sincerely thank Jesus Ramos and Steven Gurfinkel for their insightful feedback on the CUDA checkpointing functionality. We would also like to thank Andrei Vagin and Lukáš Hejtmánek for their invaluable help. This work was supported in part by the European Union's Horizon Europe research and innovation program under Grant Agreement No. 101189689.

References

[1] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. 2020. BATCH: Machine Learning Inference Serving on Serverless Platforms with Adaptive Batching. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15. doi:10.1109/SC41405.2020.00073

[2] Alibaba Cloud. 2025. *Alibaba Cloud Function Compute.* https://www.alibabacloud.com/product/function-compute Accessed: 15-09-2025.

[3] Amazon Web Services. 2025. Amazon SageMaker. https://aws.amazon.com/sagemaker Accessed: 15-09-2025.

[4] Anthropic. 2025. Claude. https://claude.ai.

[5] Microsoft Azure. 2025. Azure AI Foundry. https://ai.azure.com Accessed: 15-09-2025.

[6] Xiao Bi et al. 2024. DeepSeek LLM: Scaling Open-Source Language Models with Longtermism. arXiv:2401.02954 [cs.CL] https://arxiv.org/abs/2401.02954

[7] Seungbeom Choi, Sunho Lee, Yeonjae Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. 2022. Serving Heterogeneous Machine Learning Models on Multi-GPU Servers with Spatio-Temporal Sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 199–216. https://www.usenix.org/conference/atc22/presentation/choi-seungbeom

[8] Cloud Native Computing Foundation. 2025. KServe: Standardized Serverless ML Inference Platform on Kubernetes. https://github.com/kserve/kserve. Accessed: 15-09-2025.

[9] DeepSeek-AI, Aixin Liu, Bei Feng, et al. 2025. DeepSeek-V3 Technical Report. arXiv:2412.19437 [cs.CL] https://arxiv.org/abs/2412.19437

[10] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. 2020. Catalyzer: Sub-millisecond Startup for Serverless Computing with Initialization-less Booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 467–481. doi:10.1145/3373376.3378512

[11] Hugging Face. 2025. Deploy with your own container. https://huggingface.co/docs/inference-endpoints/en/engines/custom_container. Accessed: 15-09-2025.

[12] Hugging Face. 2025. Hugging Face Inference. https://huggingface.co/docs/inference-providers/providers/hf-inference. Accessed: 15-09-2025.

[13] Freezer Subsystem. 2025. Linux Kernel Documentation. https://www.kernel.org/doc/Documentation/cgroup-v1/freezer-subsystem.txt. Accessed: 15-09-2025.

[14] 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

[15] Georgi Gerganov. 2025. llama.cpp. https://github.com/ggerganov/llama.cpp.

[16] Google Cloud. 2025. Autoscaling Machine Learning Inference Workloads on Google Kubernetes Engine. https://cloud.google.com/kubernetes-engine/docs/best-practices/machine-learning/inference/autoscaling Accessed: 15-09-2025.

[17] Google DeepMind. 2025. Gemini. https://gemini.google.com. Google's multimodal chatbot integrated across Workspace and Search, hundreds of millions of users.

[18] Hugging Face. 2025. Simple, safe, zero-copy tensor storage. https://github.com/huggingface/safetensors

[19] Shashwat Jaiswal, Kunal Jain, Yogesh Simmhan, Anjaly Parayil, Ankur Mallick, Rujia Wang, Renee St. Amant, Chetan Bansal, Victor Rühle, Anoop Kulkarni, Steve Kofsky, and Saravan Rajmohan. 2025. Serving Models, Fast and Slow: Optimizing Heterogeneous LLM Inferencing Workloads at Scale. arXiv:2502.14617 [cs.DC] https://arxiv.org/abs/2502.14617

[20] Youhe Jiang, Fangcheng Fu, Xiaozhe Yao, Guoliang He, Xupeng Miao, Ana Klimovic, Bin Cui, Binhang Yuan, and Eiko Yoneki. 2025. Demystifying Cost-Efficiency in LLM Serving over Heterogeneous GPUs. arXiv:2502.00722 [cs.DC] https://arxiv.org/abs/2502.00722

[21] Majeed Kazemitabaar, Runlong Ye, Xiaoning Wang, Austin Zachary Henley, Paul Denny, Michelle Craig, and Tovi Grossman. 2024. CodeAid: Evaluating a Classroom Deployment of an LLM-based Programming Assistant that Balances Student and Educator Needs. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI '24)*. Association for Computing Machinery, New York, NY, USA, Article 650, 20 pages. doi:10.1145/3613904.3642773

[22] Sumer Kohli, Shreyas Kharbanda, Rodrigo Bruno, Joao Carreira, and Pedro Fonseca. 2024. Pronghorn: Effective Checkpoint Orchestration for Serverless Hot-Starts. In *Proceedings of the Nineteenth European Conference on Computer Systems* (Athens, Greece) *(EuroSys '24)*. Association for Computing Machinery, New York, NY, USA, 298–316. doi:10.1145/3627703.3629556

[23] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 611–626. doi:10.1145/3600006.3613165

[24] Yaniv Leviathan, Matan Kalman, and Yossi Matias. 2023. Fast Inference from Transformers via Speculative Decoding. In *Proceedings of the 40th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 202)*, Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (Eds.). PMLR, 19274–19286. https://proceedings.mlr.press/v202/leviathan23a.html

[25] Jie Li, Laiping Zhao, Yanan Yang, Kunlin Zhan, and Keqiu Li. 2022. Tetris: Memory-efficient Serverless Inference through Tensor Sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA. https://www.usenix.org/conference/atc22/presentation/li-jie

[26] Bo-Yi Wu Manu Martinez-Almeida, Javier Provecho et al. 2025. Gin Web Framework. https://github.com/gin-gonic/gin

[27] Meta AI. 2025. LLaMA 4: Large Language Model. https://www.llama.com/models/llama-4. Accessed: 15-09-2025.

[28] Microsoft. 2024. How to Customize an LLM: A Deep Dive to Tailoring an LLM for Your Business. https://techcommunity.microsoft.com/blog/azure-ai-foundry-blog/how-to-customize-an-llm-a-deep-dive-to-tailoring-an-llm-for-your-business/4110204 Accessed: 15-09-2025.

[29] Microsoft. 2025. Azure LLM Inference Traces. https://github.com/Azure/AzurePublicDataset.

[30] Microsoft. 2025. Copilot. https://www.bing.com/chat.

[31] Jeffrey Morgan and Michael Chiang. 2025. Ollama. https://ollama.ai/.

[32] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Anand Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. 2021. Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM. arXiv:2104.04473 [cs.CL] https://arxiv.org/abs/2104.04473

[33] Chanh Nguyen. 2025. CUDA Graph Capture Support in vLLM. https://github.com/vllm-project/vllm/pull/16072

[34] Vinh Nguyen, Michael Carilli, Sukru Burc Eryilmaz, Vartika Singh, Michelle Lin, Natalia Gimelshein, Alban Desmaison, and Edward Yang. 2021. Accelerating PyTorch with CUDA Graphs. https://pytorch.org/blog/accelerating-pytorch-with-cuda-graphs/. Accessed: 15-09-2025.

[35] NVIDIA. 2025. TensorRT-LLM: Accelerated Large Language Model Inference. https://github.com/NVIDIA/TensorRT-LLM.

[36] NVIDIA. 2025. TensorRT-LLM Build Workflow. https://nvidia.github.io/TensorRT-LLM. Accessed: 15-09-2025.

[37] OpenAI. 2025. ChatGPT. https://chat.openai.com.

[38] OpenAI. 2025. *OpenAI Pricing*. https://openai.com/api/pricing Accessed: 15-09-2025.

[39] OpenAI. 2025. Specification for the OpenAI API. https://platform.openai.com/docs/api-reference Accessed: 15-09-2025.

[40] Daon Park and Bernhard Egger. 2024. Improving Throughput-oriented LLM Inference with CPU Computations. In *Proceedings of the 2024 International Conference on Parallel Architectures and Compilation Techniques* (Long Beach, CA, USA) *(PACT '24)*. Association for Computing Machinery, New York, NY, USA, 233–245. doi:10.1145/3656019.3676949

[41] Pratyush Patel, Esha Choukse, Chaojie Zhang, et al. 2024. Splitwise: Efficient Generative LLM Inference Using Phase Splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. Institute of Electrical and Electronics Engineers, Buenos Aires, Argentina, 118–132. doi:10.1109/ISCA59077.2024.00019

[42] Qiangyu Pei, Yongjie Yuan, Haichuan Hu, Qiong Chen, and Fangming Liu. 2023. AsyFunc: A High-Performance and Resource-Efficient Serverless Inference System via Asymmetric Functions. In *Proceedings of the 2023 ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '23)*. Association for Computing Machinery, New York, NY, USA, 324–340. doi:10.1145/3620678.3624664

[43] Podman Documentation. 2025. Pause one or more containers. https://docs.podman.io/en/latest/markdown/podman-pause.1.html. Accessed: 15-09-2025.

[44] Red Hat, Inc. 2025. Podman. https://podman.io/ Accessed: 15-09-2025.

[45] Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. 2021. INFaaS: Automated Model-less Inference Serving. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 397–411. https://www.usenix.org/conference/atc21/presentation/romero

[46] Steven I. Ross, Fernando Martinez, Stephanie Houde, Michael Muller, and Justin D. Weisz. 2023. The Programmer's Assistant: Conversational Interaction with a Large Language Model for Software Development. In *Proceedings of the 28th International Conference on Intelligent User Interfaces* (Sydney, NSW, Australia) *(IUI '23)*. Association for Computing Machinery, New York, NY, USA, 491–514. doi:10.1145/3581641.3584037

[47] Amazon Web Services. 2025. AWS Lambda. https://aws.amazon.com/lambda Accessed: 15-09-2025.

[48] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, Santa Clara, CA, 205–218. https://www.usenix.org/conference/atc20/presentation/shahrad

[49] Haihao Shen, Hanwen Chang, Bo Dong, Yu Luo, and Hengyu Meng. 2023. Efficient LLM Inference on CPUs. arXiv:2311.00502 [cs.LG] https://arxiv.org/abs/2311.00502

[50] Snowflake. 2024. Snowflake LLM Inference: Model Hotswapping. https://www.snowflake.com/engineering-blog/llm-interference-model-hotswapping/ Accessed: 15-09-2025.

[51] Viktória Spišaková, Radostin Stoyanov, Lukáš Hejtmánek, Dalibor Klusáček, Adrian Reber, and Rodrigo Bruno. 2025. Kubernetes Scheduling with Checkpoint/Restore: Challenges and Open Problems. In *Job Scheduling Strategies for Parallel Processing*. Springer Nature Switzerland.

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

[53] Jovan Stojkovic, Chaojie Zhang, Íñigo Goiri, Josep Torrellas, and Esha Choukse. 2025. DynamoLLM: Designing LLM Inference Clusters for Performance and Energy Efficiency. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 1348–1362. doi:10.1109/HPCA61900.2025.00102

[54] Radostin Stoyanov. 2025. Transparent Hot-Swapping of Containerized AI/ML Workloads. In *High Performance Container Workshop*.

[55] Radostin Stoyanov, Adrian Reber, and Viktória Spišaková. 2025. Efficient Transparent Checkpointing of AI/ML Workloads in Kubernetes. In *KubeCon + CloudNativeCon Europe 2025*. https://kccnceu2025.sched.com/event/1tx7i

[56] 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

[57] Biao Sun, Ziming Huang, Hanyu Zhao, Wencong Xiao, Xinyi Zhang, Yong Li, and Wei Lin. 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

[58] Gemini Team. 2024. Gemini: A Family of Highly Capable Multimodal Models. arXiv:2312.11805 [cs.CL] https://arxiv.org/abs/2312.11805

[59] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. arXiv:2302.13971 [cs.CL] https://arxiv.org/abs/2302.13971

[60] Harshith Umesh. 2025. Ollama vs. vLLM: A deep dive into performance benchmarking. *Red Hat Developers* (8 Aug. 2025). https://developers.redhat.com/articles/2025/08/08/ollama-vs-vllm-deep-dive-performance-benchmarking Accessed: 2025-08-14.

[61] vLLM. 2025. Sleep Mode. https://docs.vllm.ai/en/latest/features/sleep_mode.html. Accessed: 15-09-2025.

[62] Jiali Wang, Yankui Wang, Mingcong Han, and Rong Chen. 2025. Colocating ML Inference and Training with Fast GPU Memory Handover. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*. 1657–1675.

[63] Yiding Wang, Kai Chen, Haisheng Tan, and Kun Guo. 2023. Tabi: An Efficient Multi-Level Inference System for Large Language Models. In *Proceedings of the Eighteenth European Conference on Computer Systems* (Rome, Italy) *(EuroSys '23)*. Association for Computing Machinery, New York, NY, USA, 233–248. doi:10.1145/3552326.3587438

[64] Yuxin Wang, Yuhan Chen, Zeyu Li, Xueze Kang, Yuchu Fang, Yeju Zhou, Yang Zheng, Zhenheng Tang, Xin He, Rui Guo, Xin Wang, Qiang Wang, Amelie Chi Zhou, and Xiaowen Chu. 2025. BurstGPT: A Real-world Workload Dataset to Optimize LLM Serving Systems. arXiv:2401.17644 [cs.DC] https://arxiv.org/abs/2401.17644

[65] Zhentao Xu, Mark Jerome Cruz, Matthew Guevara, Tie Wang, Manasi Deshpande, Xiaofeng Wang, and Zheng Li. 2024. Retrieval-Augmented Generation with Knowledge Graphs for Customer Service Question Answering. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval* (Washington DC, USA) *(SIGIR '24)*. Association for Computing Machinery, New York, NY, USA, 2905–2909. doi:10.1145/3626772.3661370

[66] 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

[67] Yanan Yang, Laiping Zhao, Yiming Li, Huanyu Zhang, Jie Li, Mingyang Zhao, Xingzhen Chen, and Keqiu Li. 2022. INFless: a native serverless system for low-latency, high-throughput inference. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 768–781. doi:10.1145/3503222.3507709

[68] Xiaozhe Yao, Qinghao Hu, and Ana Klimovic. 2024. DeltaZip: Efficient Serving of Multiple Full-Model-Tuned LLMs. arXiv:2312.05215 [cs.DC] https://arxiv.org/abs/2312.05215

[69] Minchen Yu, Ao Wang, Dong Chen, Haoxuan Yu, Xiaonan Luo, Zhuohao Li, Wei Wang, Ruichuan Chen, Dapeng Nie, and Haoran Yang. 2024. FaaSwap: SLO-Aware, GPU-Efficient Serverless Inference via Model Swapping. arXiv:2306.03622 [cs.DC] https://arxiv.org/abs/2306.03622

[70] Minchen Yu, Ao Wang, Dong Chen, Haoxuan Yu, Xiaonan Luo, Zhuohao Li, Wei Wang, Ruichuan Chen, Dapeng Nie, Haoran Yang, and Yu Ding. 2025. Torpor: GPU-Enabled Serverless Computing for Low-Latency, Resource-Efficient Inference. arXiv:2306.03622 [cs.DC] https://arxiv.org/abs/2306.03622

[71] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. 2019. MArk: Exploiting Cloud Services for Cost-Effective, SLO-Aware Machine Learning Inference Serving. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 1049–1062. https://www.usenix.org/conference/atc19/presentation/zhang-chengliang

[72] Hong Zhang, Yupeng Tang, Anurag Khandelwal, and Ion Stoica. 2023. SHEPHERD: Serving DNNs in the Wild. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 787–808. https://www.usenix.org/conference/nsdi23/presentation/zhang-hong

[73] Jeff Zhang, Sameh Elnikety, Shuayb Zarar, Atul Gupta, and Siddharth Garg. 2020. Model-Switching: Dealing with Fluctuating Workloads in Machine-Learning-as-a-Service Systems. In *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*. USENIX Association. https://www.usenix.org/conference/hotcloud20/presentation/zhang

[74] Yanqi Zhang, Íñigo Goiri, Gohar Irfan Chaudhry, Rodrigo Fonseca, Sameh Elnikety, Christina Delimitrou, and Ricardo Bianchini. 2021. Faster and Cheaper Serverless Computing on Harvested Resources. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 724–739. doi:10.1145/3477132.3483580

[75] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. 2024. SGLang: Efficient Execution of Structured Language Model Programs. arXiv:2312.07104 [cs.AI] https://arxiv.org/abs/2312.07104

Appendix: Artifact Description

Artifact Description (AD)

A Overview of Contributions and Artifacts

A.1 Paper's Main Contributions

C_1 A framework that enables cost-efficient inference serving with engine-agnostic scheme for LLM model hot-swapping, leveraging the pause/resume functionality of container runtimes in combination with transparent GPU checkpointing.

C_2 SwapServeLLM — a prototype of the engine-agnostic model hot-swapping framework designed to efficiently handle concurrent inference requests with multiple engines (backends) and models.

C_3 Extension to SwapServeLLM that integrates a demand-aware preemption policy and dynamic GPU memory reservation mechanism to optimize resource utilization and minimize inference latency.

C_4 An extensive evaluation, comparing SwapServeLLM with cold-start and model loading latencies of four state-of-the-art inference engines (vLLM, Ollama, SGLang, TensorRT-LLM) across multiple LLM models.

A.2 Computational Artifacts

A_1 https://github.com/rst0git/SwapServeLLM

Artifact ID	Contributions Supported	Related Paper Elements
A_1	$C_1 - C_4$	Figure 5 Figure 6

B Artifact Identification

B.1 Computational Artifact A_1

Relation To Contributions

Supports all four contributions by implementing the SwapServeLLM framework, which consists of the OpenAI-compatible API router, model hot-swapping logic, concurrent inference request routing, and GPU memory management. For the production of the results in the paper, SwapServeLLM provides a set of configurations and evaluation scripts. It also includes the raw CSV data of our evaluation, scripts for analyzing the data, and plotting the results. These artifacts are included in our repository and organized as follows:

- **internal/**, **pkg/**, **vendor/**: The core logic and helper functions in private and public Go modules, as well as third-party packages for reproducible build.
- **evaluation/**: Configurations and scripts for evaluation and data analysis to reproduce the results in the paper.

Expected Results

The components, evaluation scripts, and conflagrations are used in combination for the production of the results in the paper. The evaluation scripts reproduce the swap-in latency improvements and model loading benchmarks as shown in Figure 5 and Figure 6.

Expected Reproduction Time (in Minutes)

Artifact Setup: The expected computational time for deploying the setup, installing all dependencies, and downloading all inference engines and LLM models is approximately 120-240 minutes, depending on hardware availability and network bandwidth.

Artifact Execution: The expected computational time depends mostly on the available hardware, model sizes, and number of times each experiment is repeated to calculate mean values and standard deviation. In our case, the total time considering all experiments is approximately 240 minutes.

Artifact Analysis: The analysis process for this artifact is automated through a set of scripts that extract the evaluation data into CSV format and plot the results. It takes approximately 10 minutes.

Artifact Setup (incl. Inputs)

Hardware. The configurations, scripts, and LLM models used in our evaluation require NVIDIA GPUs with 80 GB of memory such as A100 or H100.

Software. The following software must be installed:

- Git: https://github.com/git-guides/install-git
- runc: https://github.com/opencontainers/runc
- Podman: https://podman.io/docs/installation
- CRIU: https://criu.org/Installation
- cuda-checkpoint: https://github.com/NVIDIA/cuda-checkpoint
- CUDA: https://developer.nvidia.com/cuda-downloads
- NVIDIA Driver: https://www.nvidia.com/en-gb/drivers/
- NVIDIA Container Toolkit: https://docs.nvidia.com/datacenter/cloud-native/container-toolkit
- Go: https://go.dev/doc/install
- Python 3: https://www.python.org/downloads

Datasets / Inputs. The inputs used in our evaluation are included as part of the artifact.

Installation and Deployment. Other than the software requirements, it is necessary to clone our GitHub repository, download vLLM, SGLang, TensorRT-LLM, and Ollama container images, install the configuration files for SwapServeLLM.

Artifact Execution

This artifact's workflow consists of two tasks T_1 and T_2, that do not have dependency between them. T_1 may deploy the vLLM, SGLang, TensorRT-LLM, and Ollama inference engines with Podman and evaluate the cold-start latency of each engine running as a local bare-metal container. T_2 may clone our GitHub repository, compile our SwapServeLLM prototype and make sure it is working properly with the available example.

Artifact Analysis (incl. Outputs)

The outputs of the experiments are log files containing the measurements of cold-start, swap-in and swap-out latencies as well as memory utilization. To process these outputs we use a set of

Python scripts that extract the measurements into files with *CSV* format and plot the data to produce Figure 5 and Figure 6.