BREAKING THE ICE: ANALYZING COLD START LATENCY IN vLLM

Huzaifa Shaaban Kabakibo [1] **Animesh Trivedi** [2] **Lin Wang** [1]

ABSTRACT

As scalable inference services become popular, the cold start latency of an inference engine becomes important. Today, vLLM has evolved into the de-facto inference engine of choice for many inference workloads. Although popular, due to its complexity and rapid evolution, there has not been a systematic study on the startup latency of its engine. With major architectural innovations under it (e.g., the `V1` API, introduction of `torch.compile`), in this paper, we present the first detailed performance characterization of vLLM startup latency. We break down the startup process into six foundational steps and demonstrate that this process is predominantly CPU-bound. Each step exhibits consistent and interpretable scaling trends with respect to model- and system-level parameters, enabling fine-grained attribution of latency sources. Building on these insights, we develop a lightweight analytical model that accurately predicts vLLM's startup latency for a given hardware configuration, providing actionable guidance for resource planning in large-scale inference environments. All our benchmarking datasets, analysis tools, and prediction scripts are open-sourced at https://github.com/upb-cn/vllm-startup-profiler.

1 INTRODUCTION

Despite the success of Large Language Models (LLMs) in various domains (Daivi, 2024; Anastasiya Zharovskikh, 2023; CellStrat, 2023), deploying LLMs at scale still poses significant challenges with respect to GPU resource provisioning, request scheduling, and performance scaling (Khare et al., 2025). To address these issues, serverless computing has emerged as an attractive paradigm for LLM serving (Fu et al., 2024; Hu et al., 2025; Lou et al., 2025; Qin et al., 2025; Zeng et al., 2025). In this paradigm, users provide LLMs while the serverless platform dynamically provisions resources to match workload variations, enabling a pay-as-you-go model that enhances cost efficiency by scaling automatically on-demand. Despite these advantages, serverless deployments face a critical challenge: **cold start latency**. Under bursty workloads, cloud providers frequently spin up new cold LLM container instances to handle traffic spikes, introducing significantly higher latency, often orders of magnitude greater than serving requests on warm instances (Zeng et al., 2025; Du et al., 2020; Oakes et al., 2018). This latency primarily impacts the Time-to-First-Token (TTFT), a key performance metric in LLM inference (Agrawal et al., 2024; Zeng et al., 2025; Fu et al., 2024).

A growing body of work has sought to mitigate the cold start latency through techniques such as accelerated checkpoint loading (Fu et al., 2024), reducing runtime initialization overhead (Akkus et al., 2018; Fuerst & Sharma, 2021; Li et al., 2022; Roy et al., 2022; Yu et al., 2024), fast state materialization (Zeng et al., 2025), and pipeline parallelism (Lou et al., 2025). However, these efforts focus on individual components of the startup process, with limited analysis of the process as a whole. This gap hinders our ability to design scalable and efficient serverless systems to meet the performance demands of LLM inference requests.

This gap is particularly evident in **vLLM** (Kwon et al., 2023), a widely adopted and rapidly evolving open-source framework for LLM inference. Despite its widespread adoption, the startup behavior of vLLM still lacks a clear structural understanding within the community. This is reflected in multiple user discussions and issue reports that attempt to diagnose or mitigate startup latency, often without a shared decomposition of the underlying steps (vLLM Project, 2025a;b;c;d;e). Only recently has a dedicated startup-time benchmark been added to the vLLM codebase, suggesting that this aspect of the system had not yet been systematically characterized (vLLM Project, 2025g). In this work, we provide the first detailed study of vLLM startup process. Specifically, *our goal is to characterize this process end-to-end, identifying the key steps, quantifying their performance dependencies, and analyzing their GPU-, CPU- and I/O-dependencies.* We argue that answering these questions is both challenging and timely for three key reasons:

Firstly, the popularity and complexity of vLLM. Over the past few years, vLLM has rapidly become one of the most

[1]Paderborn University, Paderborn, Germany [2]IBM Research Europe, Zurich, Switzerland. Correspondence to: Lin Wang <lin.wang@uni-paderborn.de>.

Proceedings of the 9th MLSys Conference, Bellevue, WA, USA, 2026. Copyright 2026 by the author(s).

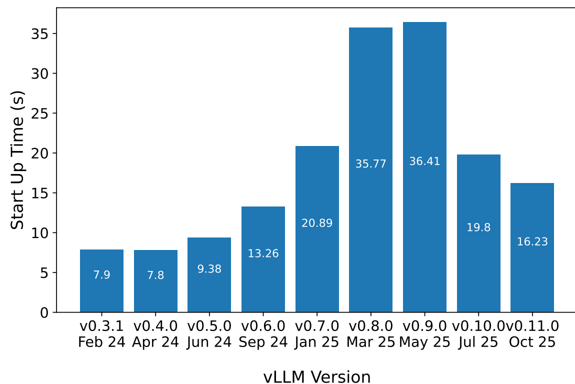

Figure 1. Startup times of different vLLM versions using the OPT-6.7B model on an H100 GPU (lower is better).

widely used inference engines, evolving quickly through frequent, community-driven releases (Kwon et al., 2023; Nar et al., 2025). It delivers a highly optimized inference path (Gordic, 2025) with techniques such as PagedAttention and prefix caching (Kwon et al., 2023), chunked prefill (Agrawal et al., 2023), disaggregated prefill-decoding (Zhong et al., 2024) and continuous batching (Yu et al., 2022). However, this fast evolution also complicates cold start optimizations, as prior techniques often become obsolete due to API changes or with the emergence of new features (if not forward ported) like the `V1` API and `torch.compile` (vLLM Project, 2025h;k;l). Furthermore, with a growing codebase (∼280K lines of Python code as of `v0.10`) and new integrations, it has become increasingly more difficult to systematically assess how changes to individual components affect the overall cold start performance (vLLM Project, 2025j). To quantify the impact of this complexity, in Figure 1 we show the vLLM startup latencies for the last nine major releases during the past 1.5 years. As evident from the figure, there is more than 4× variance in latencies, and a 2× latencies reduction observed between `v0.9` and `v0.10`. These results indicate that there is a need systematically characterize and understand the startup process of vLLM.

Secondly, heterogeneous inference ecosystem. Modern inference deployments are heterogeneous, where a variety of hardware (CPUs, GPUs, and storage devices) and software (framework, ecosystem, workload, models) parameters can influence resource management efficiency—an important factor in serverless computing. From a hardware point of view, vLLM supports a wide variety of AI accelerators with more than a dozen reported in active usage in 2024 (Hex, 2026). In terms of software, different versions of vLLM are used in the wild. We perform an analysis using *pepy.tech* website (pepy.tech, 2025) that reports `pip install` usage of various Python packages, including vLLM. Over the past three months, all vLLM versions shown in Figure 1 have been actively installed, ranging from `v0.3` (∼30K downloads) to `v0.10` (around 1.3 million downloads). Moreover, serving systems commonly host multiple models with diverse sizes, families, architectures (e.g., transformers, mixture-of-experts, and hybrid designs), and popularity (vLLM Project, 2025f; Yu et al., 2025). Furthermore, in terms of resource management, LLM requests remain highly bursty in nature, making provisioning for the peak demand both costly and inefficient. We analyze multiple publicly available LLM production traces (Microsoft Azure (Cortez et al., 2017), Shanghai AI Lab (Hu et al., 2024), Mooncake AI (Qin et al., 2025), and Alibaba (Chen et al., 2025) traces) and report peak-to-mean ratios of 2-20×, revealing significant variance in request arrival rates, thus making resource provisioning challenging. Similar numbers were also reported in the past literature (Khare et al., 2025). The diversity in hardware, software, models, and workloads underscores the importance of developing a systematic understanding of the startup process to ensure efficient resource management across this heterogeneous ecosystem.

Lastly, emergence of containerized, scalable distributed inference blueprints. Since early 2025, LLM ecosystem has seen a rapid emergence of end-to-end, containerized frameworks designed to support distributed and scalable inference. These frameworks provide full blueprints for running production-level LLM services with tightly integrated components, such as an inference request router, a KV-Cache manager, and worker autoscalers. Examples include NVIDIA Dynamo (NVIDIA, 2025b), Red Hat LLM-D (Red Hat, 2025), AIBrix (AIBrix, 2025), and vLLM Production Stack (vLLM Production Stack, 2025). A key requirement shared across these frameworks is the ability to scale GPU workers efficiently, which in turn requires an accurate model of the startup cost for each worker instance (LLM-D, 2025). For example, NVIDIA Dynamo recommends performing multi-hour offline profiling of inference environments, followed by periodic online monitoring to build detailed performance models for efficient resource provisioning (NVIDIA, 2025c). Such profiling captures end-to-end inference behavior, including model execution, scheduling, and runtime dynamics, to inform autoscaling and workload placement decisions. However, these system-level models typically treat startup as part of a larger inference lifecycle rather than isolating it as a distinct component of container initialization (Balamurugan et al., 2025). Hence, an accurate and detailed startup characterization is an essential contribution for designing impactful autoscaler policies for containerized, serverless inference frameworks.

With these trends, in this paper, we take a step back and focus on developing a comprehensive and systematic understanding of vLLM's startup process. We focus specifically on the vLLM engine initialization phase, while intentionally controlling for distributed factors such as container startup, remote storage, and network effects, in order to

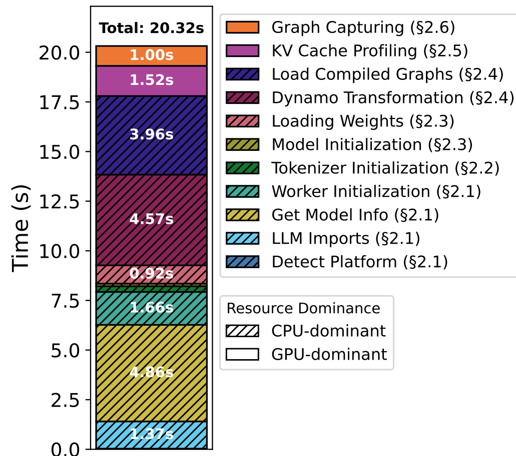

Figure 2. vLLM startup latency breakdown with Llama3.2-3B.

isolate the core system behavior. We decompose the startup process into six foundational steps and identify that the overall process is largely CPU-bounded. By examining the scaling characteristics of each step, we uncover consistent and interpretable relationships among model configuration, system environment, and startup latency. Leveraging these insights, we develop a lightweight analytical model capable of predicting vLLM startup time for a given hardware and model configurations. This predictive model enables more informed scheduling and autoscaling decisions in serverless deployments, allowing cloud platforms to plan resource allocation and mitigate cold starts effectively.

2 Performance Characterization of vLLM Startup Process

We start by conducting a detailed performance characterization of vLLM's startup process. Throughout this paper, we define the startup time as the duration between the start of the inference engine's initialization and the point at which the engine becomes fully operational and ready to serve inference requests (e.g., when vLLM prints *Application startup complete*). Our goal in performance characterization is to (i) identify unique steps involved in the startup process; (ii) synthesize their performance and scaling dependencies on vLLM-internal configuration and external factors. We perform our analysis with 22 LLMs, detailed in Table 1, with a variety of architectures and configurations on NVIDIA H100 (§2) and L40S (§3) GPUs (NVIDIA, 2025d;e) using vLLM `v0.10.1.1`. More details of our system environments are shown in Table 2. Unless mentioned otherwise, all experiments are conducted on node `n1` equipped with H100 GPU and AMD EPYC 9354 CPU. For clarity and space reasons, models are selectively omitted in certain plots without loss of generality. To ensure visual consistency, each model is represented using a fixed color across all figures, following the mapping illustrated in Table 1. All experiments are conducted five times, and the reported results represent the average time.

Our analysis reveals six unique steps in vLLM's startup process as shown in Figure 2 for Llama3.2-3B (Meta AI, 2023). The figure also shows the dominant resource type for each step and its contributions towards startup latency of 20.32 secs. As shown, all steps are CPU-bound, except for the final two steps (e.g., KVCache profiling and CUDA graph capture), which are GPU-bound, thus establishing that the overall startup process is predominantly bounded by CPU. In the following subsections, we provide a detailed explanation of each step, covering both, their functional role in the startup process, and their potential performance implications. We believe that the upcoming vLLM releases will optimize these steps instead of eliminating them, thus ensuring longevity of this analysis and insights produced.

2.1 vLLM Framework Bootstrapping

The first step of vLLM's startup process is Framework Bootstrap, which configures the runtime environment before any model components are loaded. In this step, vLLM initializes the runtime environment and launches an OpenAI-compatible inference API server responsible for managing inference requests. It comprises four substeps:

Detect Platform. vLLM probes the available hardware backend by importing runtime modules (e.g., CUDA or CPU) and confirming device support (NVIDIA, 2025a). This determines the appropriate execution environment for subsequent stages.

Dependencies Imports. After backend detection, the framework loads its core dependencies, such as PyTorch, Transformers, tokenizer libraries, and vLLM-specific plugins. The dynamic import and symbol resolution of these packages can introduce noticeable latency of several seconds.

Get Model Info. The API server retrieves the model's configuration and tokenizer metadata from a local repository or via remote querying (e.g., HuggingFace). This step involves file or network I/O and JSON parsing of files such as `config.json` and `model_index.json`, which define the model architecture and supported tasks.

Worker Initialization. Finally, vLLM spawns its main worker via Ray or Python multiprocessing, setting up interprocess communication, shared memory, and GPU contexts to prepare the runtime for model loading and inference.

Overall, the Framework Bootstrap step is mainly governed by vLLM's internal implementation and shows stable latency within the same environment across all models, regardless of their parameters. Recent optimizations, such as caching the `Model_Class` metadata after the first run (vLLM Project, 2025i), have reduced the latency of the

Table 1. Comparison of models architectures and configurations. MoE = Mixture of Experts, MHA = Multi-Head Attention, GQA = Grouped Query Attention, MQA = Multi-Query Attention, MLA = Multi-Latent Attention, $K = 1000$. Color boxes correspond to the colors used in all Figures.

Model	Layers	Hidden Size	FFN Dim	Heads (Q / KV)	Attention Type	Vocab Size	Tokenizer Size	MoE?
LLaMA 2-7B (Touvron et al., 2023)	32	4096	11008	32 / 32	Full MHA	32K	1.8MB	No
LLaMA 2-13B (Touvron et al., 2023)	40	5120	13824	40 / 40	Full MHA	32K	1.8MB	No
LLaMA 3-3B (Meta AI, 2023)	28	3072	8192	24 / 8	GQA (3:1)	128K	8.7MB	No
Falcon-7B (tiiuae, 2023)	32	4544	18176	71 / 1	MQA	65K	2.7MB	No
Qwen-0.5B (Bai et al., 2023)	24	1024	2816	16 / 16	Full MHA	152K	6.8MB	No
Qwen-1.8B (Bai et al., 2023)	24	2048	5504	16 / 16	Full MHA	152K	6.8MB	No
Qwen-4B (Bai et al., 2023)	40	2560	6912	20 / 20	Full MHA	152K	6.8MB	No
Qwen-7B (Bai et al., 2023)	32	4096	11008	32 / 32	Full MHA	152K	6.8MB	No
Qwen-14B (Bai et al., 2023)	40	5120	13696	40 / 40	Full MHA	152K	6.8MB	No
Yi-6B (01.AI et al., 2024)	32	4096	11008	32 / 4	GQA (8:1)	64K	3.5MB	No
Yi-9B (01.AI et al., 2024)	48	4096	11008	32 / 4	GQA (8:1)	64K	3.5MB	No
Falcon-11B (Malartic et al., 2024)	60	4096	16384	32 / 8	GQA (4:1)	65K	2.7MB	No
Mistral-7B (Jiang et al., 2023)	32	4096	14336	32 / 8	GQA (4:1)	32K	1.8MB	No
Qwen-MoE-14.3B-A2.7B (Qwen, 2024)	24	2048	1408	16 / 16	Full MHA	152K	6.8MB	Yes
Gemma-7B (Google, 2024)	28	3072	24576	16 / 16	Full MHA	256K	17.5MB	No
GPT-OSS-20B (OpenAI, 2025)	24	2880	2880	64 / 8	GQA (8:1)	201K	27MB	Yes
Granite3.3-8B-Instruct (Granite Team, IBM, 2025a)	40	4096	12800	32 / 8	GQA (4:1)	50K	3.4MB	No
Granite4.0-h-small-32B (Granite Team, IBM, 2025c)	40	4096	1536	32 / 8	GQA (4:1)	100K	6.9MB	Yes
Granite4.0-h-mirco-3B (Granite Team, IBM, 2025b)	40	2048	8192	32 / 8	GQA (4:1)	100K	6.9MB	No
DeepSeek-V2-Lite-16B (DeepSeek-AI, 2024)	27	2048	1408	16 / 16	MLA	102K	4.4MB	Yes
DeepSeek-R1-Distill-Llama-8B (DeepSeek-AI, 2025)	32	4096	14336	32 / 8	GQA (4:1)	128K	8.7MB	No
DeepSeek-R1-Distill-Qwen-7B (DeepSeek-AI, 2025)	28	3584	18944	28 / 4	GQA (7:1)	152K	6.8MB	No

Table 2. Hardware and software configurations for the nodes used in experiments.

	node1 (`n1`)	node2 (`n2`)	node3 (`n3`)	node4 (`n4`)
CPU	AMD EPYC 9354 (32C)	AMD EPYC 9354 (32C)	2x Intel Xeon Platinum 8568Y+ (2x48C)	2x Intel Xeon Gold 5520+ (2x28C)
GPU	H100 NVL	L40S	H100	L40S
DRAM	DDR5 251GB	DDR5 251GB	DDR5 2TB	DDR5 2TB
OS	Debian 12	Debian 12	Red Hat Enterprise Linux (RHEL) 9	Red Hat Enterprise Linux (RHEL) 9
Kernel	`6.1.0-40-amd64`	`6.1.0-40-amd64`	`5.14.0-503.34.1.el9_5`	`5.14.0-503.34.1.el9_5`
Python	3.11.2	3.11.2	3.12	3.12
CUDA	12.6 (Driver 580.82.07)	12.6 (Driver 580.82.07)	12.8 (Driver 570.124.06)	12.8 (Driver 570.124.06)
PyTorch	2.7.1+cu126	2.7.1+cu126	2.7.1+cu126	2.7.1+cu126
vLLM	v0.10.1.1	v0.10.1.1	v0.10.1.1	v0.10.1.1
SSD	–	–	4x PCIe 5.0 SSDs	–
FS	–	–	XFS/LVM with RAID-0 mirror	–

Get Model Info substep from roughly 4.47 secs, to about 0.12 secs. Since this feature was introduced in `v0.11`, it was not enabled in our experiments, which covered releases up to `v0.10.1.1`.

Summary: vLLM framework bootstrapping latency depends mainly on its implementation, and is independent of the model used. Users have little direct control over this step, though on-going optimizations are reducing its impact in newer vLLM versions.

2.2 Tokenizer Initialization

The second step of vLLM's startup process is tokenizer initialization, which prepares the component that converts raw text into a numerical form that the model can process (Singh & Strouse, 2024). This step can be divided into two parts: (i) loading the tokenizer's vocabulary files and configurations, and (ii) mapping user input text into model-compatible token IDs. However, during the startup process, only the first part occurs where vLLM loads the tokenizer's vocabulary and builds the internal logic required for tokenization.

The bar chart in Figure 3 shows the tokenizer initialization time across models. We observe that models like Llama2-7B and Falcon-7B initialize much faster (0.08 secs), while models with larger tokenizers, such as Llama3-3B, take longer (0.29 secs). Additionally, the Qwen series, all sharing the same tokenizer size (6.8 MB), exhibit identical initialization time (0.25 secs), regardless of the model size.

The inset plot in Figure 3 shows a strong linear relationship between tokenizer size and initialization time, confirmed by a regression fit with a Pearson correlation coefficient (PCC)

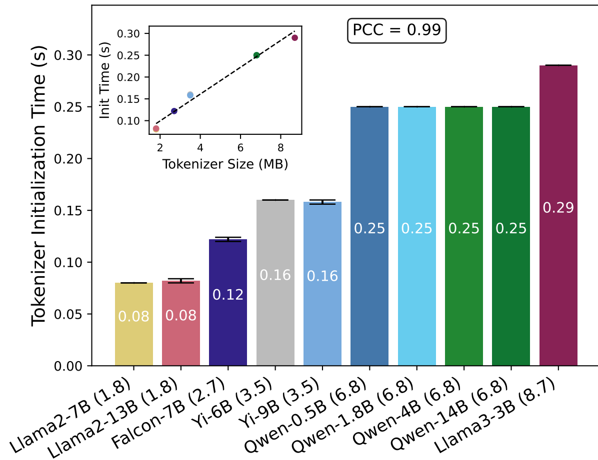

Figure 3. Strong linear relationship between the tokenizer size (shown in parentheses) and tokenizer initialization time.

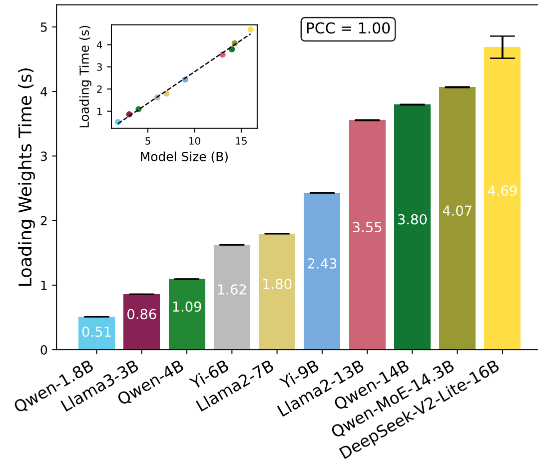

Figure 4. Strong linear relationship between the model size and loading weights time.

of 0.99. PCC is a measure ranging from -1.0 (no corelation) to +1.0 (perfect corelation) indicating the strength of a linear relationship between two variables. This shows that initialization latency is primarily governed by the size of tokenizer files, which contain the vocabulary, merge rules, and metadata. As detailed in Table 1, tokenizer size mainly depends on vocabulary size, with minor effects from tokenizer type and encoding format (e.g., JSON vs. binary). Although different models employ distinct tokenization methods (e.g., SentencePiece for LLaMA, BPE for Falcon), the tokenizer type itself has little impact; overall, tokenizer's file size dominates initialization performance.

Summary: Tokenizer initialization time scales linearly with the tokenizer size, which in turn is determined by the model vocabulary size.

2.3 Model Loading

The third step is model loading. It occurs in two steps: initializing the model structure and loading the pretrained weights. An LLM consists of two main components: its architecture (e.g., layers, activation functions, and attention blocks) and its pretrained weights (i.e., the parameters that define how the model performs its tasks). These components are usually distributed in separate files, which are read and loaded into the GPU memory during the model loading step.

Initializing The Model Structure. In this step, vLLM creates the model architecture in memory, setting up layers, attention blocks, and activation functions as defined by the model's configuration file. We identify this step to be independent of model parameter size and other model parameters (e.g., hidden size, FFN dimension). In our experiments, this step consistently takes about 0.1 ± 0.05 s across models, regardless of their size or architecture.

Loading The Weights. This step involves transferring the pretrained parameters (the weights for the attention block, FFN, etc.) from checkpoint files to GPU memory. Figure 4 shows the latency of this step across a variety of models loaded with `FP16` format. The size of the model depends primarily on the number of model parameters and the numeric precision used to store them. In the figure, the parameter count is reflected in the model names (e.g., 1.8B denotes 1.8 billion parameters). For example, a model such as Qwen-1.8B contains approximately 1.8 billion parameters; when stored in `FP16` format (2 bytes per parameter), the model requires about 3.6 GB of data to be loaded into GPU memory. In the bar chart, we see that models like Qwen-1.8B and Llama3-3B, with smaller parameter counts, have relatively low loading times around 0.5–1 secs, while larger models such as DeepSeek-V2-Lite-16B take longer, reaching nearly five seconds. Note that these experiments are done with a warm Linux buffer cache, i.e., model checkpoint files are effectively read from the system DRAM. We quantify the impact of SSD loading in §3.3.

The loading times in Figure 4 exhibit a clear trend: larger models require more time to load their weights, in direct proportion to their size. The inset plot further supports this observation by showing a linear relationship between the model size and loading time, with PCC = 1 and a regression line confirming that the loading time increases predictably as model size grows. Nonetheless, secondary factors such as quantization or precision format, may influence this step by reducing data volume and transfer time.

Summary: Model loading is dominated by the weight-loading time, which *linearly* depends on the model parameter size and the numeric precision.

2.4 Torch Compilation

The fourth step in vLLM's startup process is the `torch.compile` step, introduced in version `v0.7.0` as

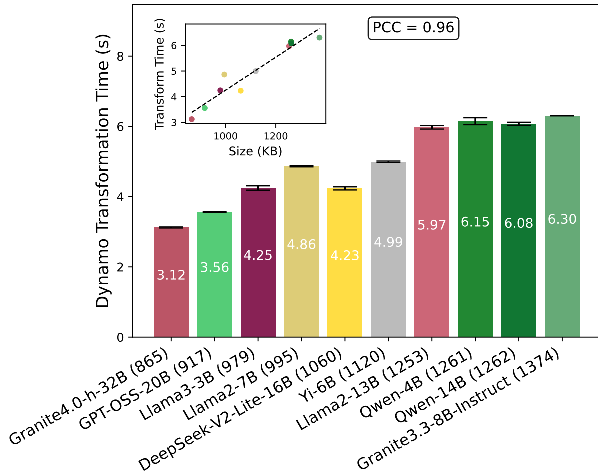

Figure 5. Strong linear relationship between compiled graphs size (shown in parentheses, in KB) and Dynamo transformation time.

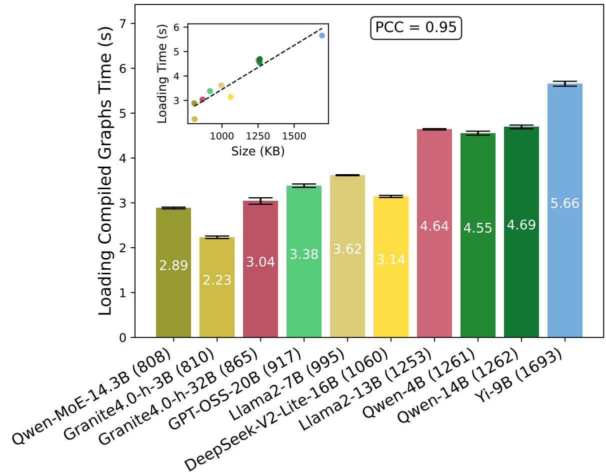

Figure 6. Strong linear relationship between compiled graphs size (shown in parentheses, in KB) and loading compiled graphs time.

a major optimization milestone. It leverages PyTorch's compilation infrastructure to convert Python-level execution into optimized, low-level kernels, reducing Python overhead and enabling kernel fusion for faster inference (PyTorch, 2023). In vLLM, this process includes two substeps: (i) *Dynamo Bytecode Transformation* and (ii) *Loading/Storing Compiled Graphs* (vLLM Docs, 2025).

Dynamo Bytecode Transformation. Dynamo captures and transforms Python bytecode at runtime to extract computational graphs from standard PyTorch programs (PyTorch, 2025a; Puneet Mangla, 2023). In regular execution, each model operation (e.g., matrix multiplication, activation functions, or attention layers) runs as a separate Python call, incurring interpreter overhead and limiting compiler-level optimizations (Ansel et al., 2024). Dynamo instead rewrites the execution into an *intermediate representation (IR)*, a static, compiler-friendly, graph-like form, enabling optimizations such as operator fusion and memory planning (Ansel et al., 2024; PyTorch, 2024; 2023).

We find that Dynamo transformation time grows with the number of layers, as each additional layer introduces more operations that must be traced and transformed. The time also depends on the complexity of each layer, which refers to the number of kernels, metadata and function wrappers that must be traced, meaning that complex layers need more time to be compiled. We observe that a good proxy for the complexity of a layer is the size of the generated compiled graph file. To verify this observation, Figure 5 reports this relationship across models, showing that transformation time increases with the total compiled graph files size, computed as the sum of all layers graph sizes. For example, Llama2-7B and Llama2-13B require 4.86 secs and 5.97 secs respectively. Although they have the same architecture (hence similar complexity), the latter takes more time due to higher number of layers (32 vs. 40). The inset plot confirms a near-linear scaling trend with PCC = 0.96, indicating that transformation cost is primarily determined by the layer count and per-layer complexity.

Loading/Storing Compiled Graphs. After Dynamo generates IR graphs, TorchInductor (PyTorch, 2025b) compiles them into highly optimized, low-level kernels that run efficiently on the GPU (PyTorch, 2025c). vLLM then stores these compiled artifacts in a file system cache, enabling subsequent runs to skip recompilation. Our measurements are performed with this cache already populated; §3.5 will show the impact of this caching on the startup latency.

As shown in Figure 6, loading time scales linearly with compiled graph size, from 2.89 secs for Qwen-MoE-14.3B (808 KB) to 5.66 secs for Yi-9B (1.69 MB). Models sharing the same architecture and number of layers, such as Qwen-4B and Qwen-14B, exhibit similar loading times (4.55 secs vs. 4.69 secs) despite different parameter counts. The inset confirms this linear dependence with PCC = 0.95, showing that the loading cost depends primarily on the compiled graph size rather than the model architecture or parameter size.

Summary: The `torch.compile` step time increases with both the number of layers and their complexity, which can effectively be approximated by the total size of the generated compiled graph files.

2.5 KVCache Profiling

The fifth step in vLLM's startup process is key–value (KV) cache profiling, which determines the optimal amount of GPU memory to allocate for the KVCache. The KVCache stores the past key and value tensors generated by the attention mechanism, allowing the model to reuse them efficiently across decoding steps. Since the cache for attention

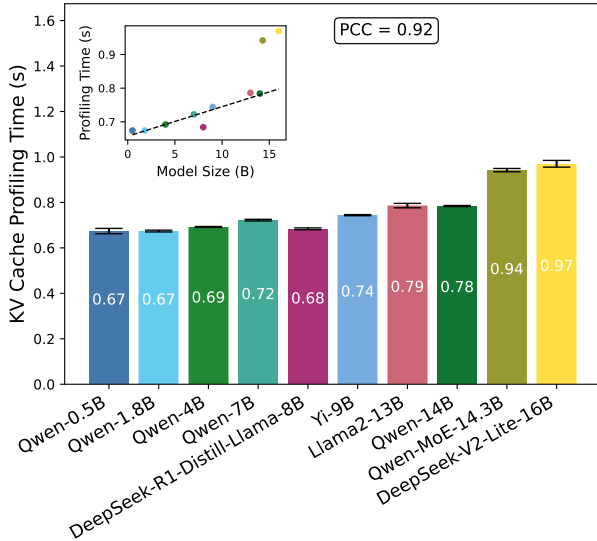

Figure 7. KVCache profiling time across different models. The inset shows a linear regression fitted only on non-MoE models (excluding Qwen and DeepSeek, shown as two yellow and olive dots in the top right in the inset) with `torch.compile` disabled.

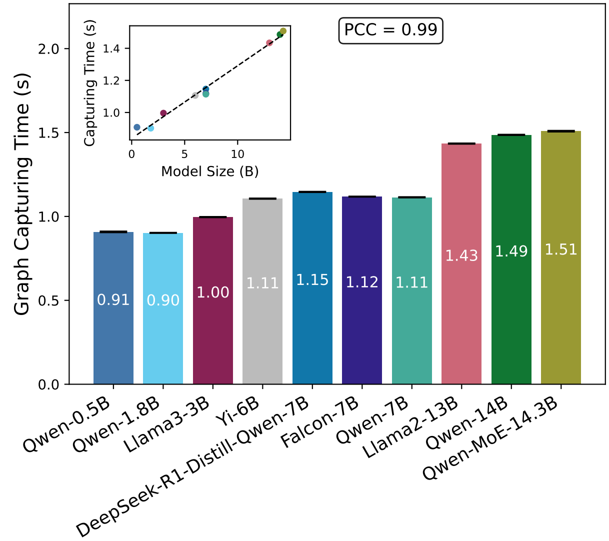

Figure 8. Strong linear relationship between model size and CUDA graph capturing time.

layers grows with each generated token, incorrect allocation can lead to out-of-memory errors. Profiling is therefore critical: vLLM executes a dummy forward pass to measure peak memory usage, and the remaining available GPU memory is then allocated for the KVCache. This ensures a balance between stability and maximum memory utilization during inference.

Since this step involves the first invocation of the model's forward pass through the dummy run, `torch.compile` is implicitly triggered, causing the profiling time reported by vLLM to include compilation overhead. Initially, we attempted to isolate the profiling duration by subtracting the measured `torch.compile` time from the logged values. However, this approach failed to yield a consistent trend. A closer analysis revealed that the inclusion of compilation overhead within the profiling stage distorted the measurements. To obtain a more accurate characterization of the intrinsic profiling behavior, we re-ran this step with `torch.compile` explicitly disabled, ensuring that only the dummy forward execution time is captured.

Figure 7 shows the measured profiling time across models after applying these adjustments. Smaller models such as Qwen-0.5B and Qwen-1.8B require about 0.67 secs, whereas medium-scale models like Qwen-4B and Qwen-7B take 0.69–0.72 secs. Larger transformer models including Yi-9B, Llama2-13B, and Qwen-14B exhibit profiling times between 0.74–0.79 secs, while the heaviest Mixture-of-Experts (MoE) models (Qwen-MoE-14.3B, DeepSeek-V2-Lite-16B) reach up to 0.94–0.97 secs. The results confirm that once the compilation overhead is removed, the KVCache profiling time follows a predictable linear trend with model size (PCC = 0.92), except for MoE models. Non-MoE models show a strong linear dependency on model size, consistent with expectations, since this step performs a dummy forward pass whose duration grows proportionally with the number of parameters shown in the inset that excludes fitting for the MoE models. In contrast, MoE models deviate due to their dynamic expert activation and load-balancing mechanisms, which introduce additional profiling complexity during the dummy run (Huang et al., 2024; Mu & Lin, 2025). Characterizing these non-linearities remains part of our ongoing work.

Summary: KVCache profiling scales linearly with the *transformer* model parameter size. MoE models deviate from this trend due to expert routing and varying activation patterns.

2.6 CUDA Graph Capturing

The final step of vLLM's startup process is CUDA graph capturing. During this step, vLLM performs a dummy forward pass to record the execution of inference kernels (including memory allocations, attention computations, and other CUDA operations) into a CUDA graph during the startup process. This graph encodes the exact sequence of GPU operations required for inference. Once captured, the graph can be replayed efficiently without re-launching individual kernels, significantly reducing CPU–GPU synchronization overhead and kernel launch latency. This makes CUDA graph capturing particularly important for achieving high inference throughput and low latency in production environments (Alan Gray, 2019).

To see the influence of different parameters on this step, we conduct two experiments. Figure 8 shows the first one,

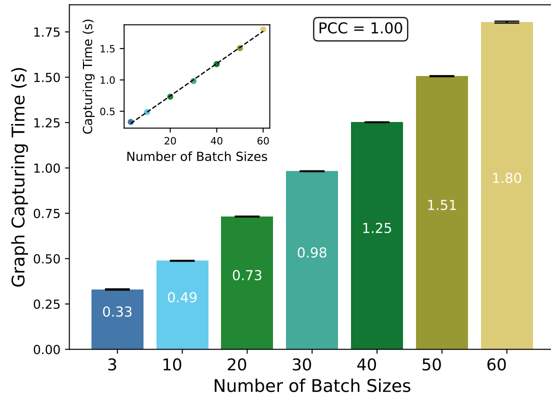

Figure 9. CUDA graph capturing time for different batch sizes using the Llama2-7B (Touvron et al., 2023) model.

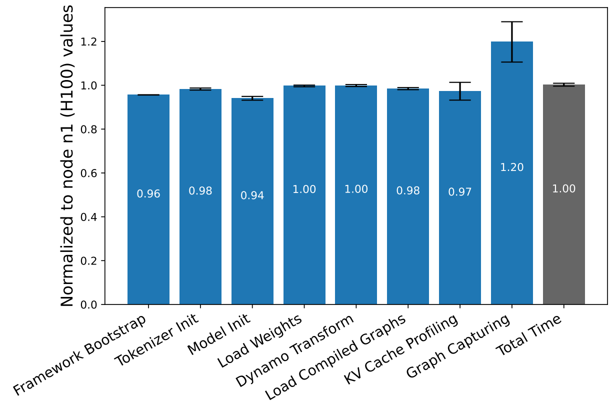

Figure 10. Startup steps comparison between H100 (n1) and L40S (n2) GPUs. All values are normalized to the H100 baseline.

where we measure CUDA capturing time for different models with different sizes. We observe that as the model size increases, so does the capturing time confirming a linear trend with PCC = 0.99. For instance, In Figure 8, Qwen-0.5B takes about 0.91 secs, while Qwen-MoE-14.3B requires around 1.51 secs for capturing.

In the second experiment, as shown in Figure 9, we analyze the effect of the batch size on the capturing time. Motivated by the fact that CUDA Graph capturing must be performed separately for each unique batch size (CUDA Graphs, 2025), we measure the capturing time for a single model, Llama2-7B (Touvron et al., 2023), using a range of batch sizes. The results show a familiar trend. The time for capturing the graph increases from 0.33 secs with three batch sizes to 1.8 secs with 60 batch sizes. These numbers reveal a gradual increase in capturing time as both model size and batch size grow, showing a linear relationship with PCC = 1.

By examining the inset graphs in both figures, we can deduce a clear linear relationship between the capturing time and the two parameters: model size and batch size. In both cases, the graphs demonstrate a steady increase in capturing time as either the model size or batch size increases.

Summary: CUDA graph capturing time scales linearly with both model size and the number of batches to be captured.

2.7 Summary

Our breakdown of vLLM's startup process reveals distinct and quantifiable dependencies across all six foundational steps to vLLM-internal (runtime initialization), and model-dependent (size, complexity, architecture) factors. We believe that these are foundational steps, and new releases of vLLM will not invalidate our insights but will build on them. Together, these observations demonstrate that each startup step has clear and interpretable scaling trends with respect to specific parameters. This systematic characterization establishes a strong empirical basis for modeling startup latency analytically to help build a predictive scheduler for a serverless autoscaler (§4).

3 Impact of Benchmarking Environment

In this section, we analyze the impact of the benchmarking environment (e.g., GPU, CPU, storage) and different configurations on our findings from §2. We also examined additional factors—-including containerization with Docker, varying PyTorch and Python versions, and modifying common runtime configuration flags (e.g., `--max-model-len` and OpenAI-compatible vLLM arguments)—-but observed no measurable or statistically significant impact on startup time and are thus omitted from discussion here.

3.1 Impact of Different GPUs

To validate the resource dependency findings presented in Figure 2, we repeat the previous experiments on node `n2`, which has the same system configuration as `n1` but with L40S GPU instead of H100 (see Table 2). These experiments are done on the first ten models listed in Table 1. Figure 10 illustrates the average speedup of each startup step across all evaluated models, calculated as the ratio of startup time on the H100 to the L40S (the y-axis).

As shown in the figure, most steps exhibit almost no speedup, indicating negligible performance gains when using the H100 over the L40S. The only notable exception is CUDA Graph Capturing, which demonstrates a speedup of 1.2× due to the forward pass run on the GPU during this step. These results are consistent with our earlier observations in Figure 2, reinforcing the conclusion that the startup process is predominantly CPU-bound. Notably, despite the different GPU architecture, and the H100's signifi-

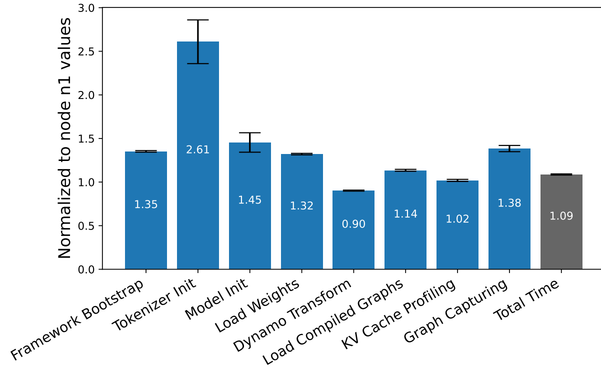

Figure 11. Comparison between AMD EPYC 9354 (`n1`) and Intel Xeon Platinum 8568Y+ (`n3`) CPUs with H100 GPUs.

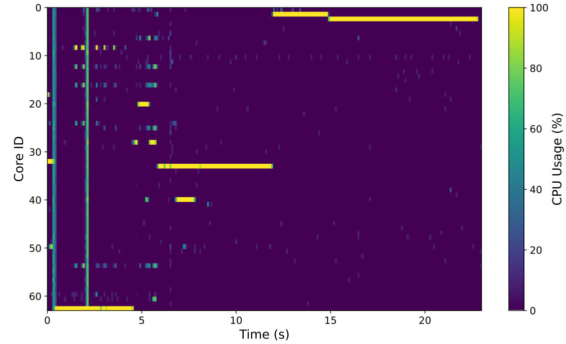

Figure 12. CPU usage per core over time during vLLM startup for the Qwen-4B model. Sampling interval: 100 ms.

cantly higher theoretical TFLOPs throughput compared to the L40S, (Liquid Web, 2025; Vast AI, 2024), this advantage does not translate into faster overall startup time. This observation indicates that the GPU performance has limited impact on the startup process.

3.2 Impact of Different CPUs

To further validate that vLLM's startup process is predominantly CPU-bound, we conduct experiments comparing environments with identical GPUs but different CPUs. Specifically, we compare our main node `n1` (AMD EPYC 9354 + NVIDIA H100), against node `n3`, equipped with an Intel Xeon Platinum 8568Y+ CPU and H100 GPU.

As shown in Figure 11, changing the CPU has a noticeably higher impact on startup time than changing the GPU (§3.1), reinforcing our earlier conclusion that the startup process is largely CPU-bound. However, the direction and magnitude of speedups across substeps varied considerably. For example, `n1` outperforms `n3` in *Tokenizer Initialization*, and *Model Initialization*, while `n3` is faster in *Graph Capturing* and *Dynamo Transformation*. The same experiment is also conducted between node `n2` and another node `n4`, equipped with Intel Xeon Gold 5520+ CPU and the same L40S GPU. Similarly, relative performance across steps fluctuates rather than following a consistent trend; the corresponding figure is omitted for space.

To better understand CPU involvement during the startup process, we monitored the utilization of individual CPU cores over time using a sampling interval of 100 ms. Figure 12 shows the per-core utilization during the startup of the Qwen-4B (Bai et al., 2023) model. The heatmap shows that at any given time, at least one CPU core reaches full (100%) utilization, indicating that vLLM continuously keeps one core saturated throughout the startup process. In contrast, there are very few instances where more than two cores are simultaneously fully utilized, suggesting that most startup operations are sequential or involve limited parallelism.

Overall, while these results confirm that CPU choice has a substantial effect on the vLLM startup latency, a precise attribution of the performance differences to the CPU (micro)architectural and systems-level factors (scheduling, runtime) remains part of our ongoing analysis.

3.3 Impact of SSDs

Our previous experiments used model weights and other configuration files cached in the warm Linux buffer memory (DRAM). To study the impact of loading directly from storage (SSDs), we repeat the experiments after flushing the buffer cache between runs, forcing all reads to be from the SSD. We conduct these experiments on node `n3`, equipped with an LVM volume spanning with mirroring four PCIe 5.0 SSDs, delivering read and write throughput of 25 and 15 GB/s, respectively with `fio` for large I/O transfers. In this experiment, we use the first ten models listed in Table 1, each repeated five times to account for variations. Figure 13 summarizes the averaged results across models, normalized to the DRAM baseline.

As expected, the only step that has a significant impact due to the data loading from SSDs is the Model Loading step. It slows down by a factor of 0.5×. The use of SSDs has minimal impact on all other steps, indicating that storage I/O does not significantly affect CPU-bound steps such as compilation or profiling. However, despite this considerable relative improvement in the weight-loading step, the overall startup time improves by only 1.04×. This modest total gain arises because the loading step constitutes only about 7–10% of the total startup duration in our measurements.

3.4 Impact of Model Weights' Loading Methods

vLLM supports multiple methods for loading model weights, each adopting distinct serialization and deserialization strategies. To understand how these methods affect startup latency, we evaluate three supported baselines: (i) **Safetensors** is the default method used in our previous experiments, and it serves as the baseline (HuggingFace, 2025). It stores

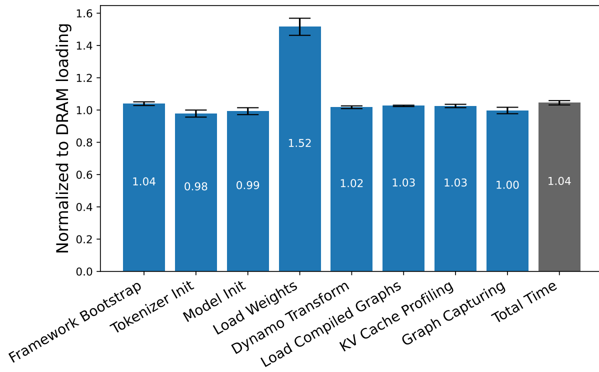

Figure 13. Impact of running the startup process while the model weights are retrieved from storage (SSD).

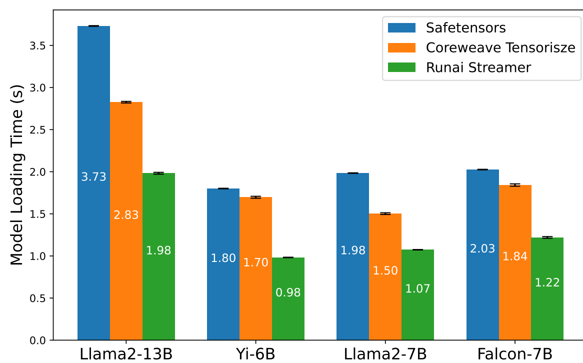

Figure 14. Model loading times across four models using different loading backends.

model tensors as pre-serialized binary files that are memory-mapped and loaded directly into CPU memory before GPU transfers. (ii) **Run:ai Model Streamer** enables concurrent reading and streaming of tensors directly into GPU memory, reducing I/O bottlenecks (Run-AI, 2025). (iii) **CoreWeave Tensorizer** uses a custom serialization format optimized for fast deserialization, allowing tensors to load directly into GPU without intermediate CPU staging (CoreWeave, 2025).

As shown in Figure 14, the *Model Loading* step exhibits substantial variation across different loading methods. Tensorizer consistently achieves the lowest latency, loading models up to 53–60% of Safetensors' time, while Run:ai Model Streamer provides moderate gains due to overlapping I/O and GPU transfers. These results demonstrate that weight loading is one of the few I/O-sensitive components of the startup process, and that optimizing data streaming and deserialization can yield meaningful reductions in the total startup time.

3.5 Impact of Non-Cached Compiled Graphs

By default, vLLM caches the compiled computation graphs generated during the `torch.compile` step after the first run, enabling subsequent runs to bypass costly graph transformations. To quantify the cost of a completely cold start, we disable this cache by setting the environment variable `VLLM_DISABLE_COMPILE_CACHE=1`, forcing vLLM to regenerate and store all compiled graphs at runtime. As shown in Figure 15, disabling the cache dramatically increases the latency of this step. The total graph storing time ranges between 11–21 secs across models, compared to 3–6 secs when cached (see Figure 6). Furthermore, as observed earlier, the compilation cost scales nearly linearly (PCC = 0.95) with the size of the compiled graphs, similar to our results in §2.4.

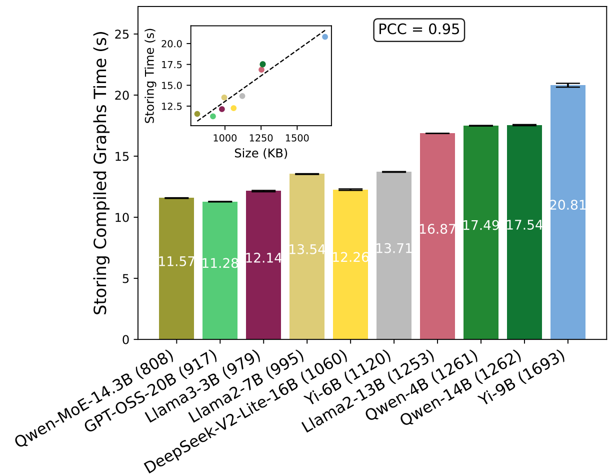

Figure 15. Strong linear relationship between compiled graphs size (shown in parentheses) and storing compiled graphs time.

4 ANALYTICAL PREDICTOR

Building on our detailed breakdown of vLLM's startup process, we now introduce a white-box regression-based analytical predictor for non-MoE models. This predictor estimates startup latency based on the model configurations and environment characteristics that we studied in §2 and §3. Figure 16 shows the overall working of the predictor consisting of the following four steps: (i) gathering of model configuration information for desired models; (ii) automated running of vLLM with models to collect the startup latency timing data from logs (takes typically a few hours); (iii) training step-specific predictors for the environment; and (iv) predicting the startup time.

Beyond estimating cold-start latency, the predictor serves as an interpretable conceptual model of the startup process that practitioners can use to reason about vLLM's startup behavior. For example, when diagnosing performance regressions or unexpected results, such as (vLLM Project, 2025b), the predictor provides a step-wise baseline against which empirical measurements can be compared, helping to identify the stages responsible for the slowdown. In addition, its step-wise structure enables estimating the performance

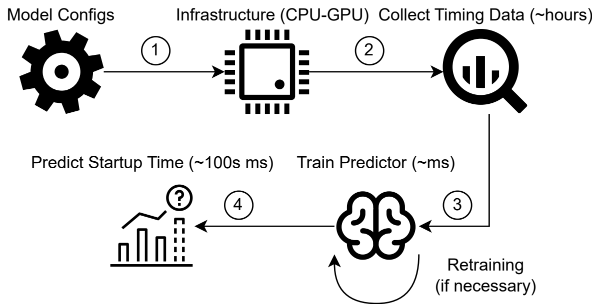

Figure 16. Workflow of the proposed predictor.

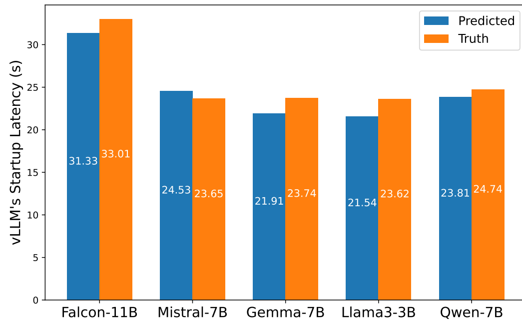

Figure 17. Validation of the predictor against measured startup latency across different models.

of optimizations that reuse parts of the initialization path. For instance, we confirm that the predictor can be used to estimate the performance of fast model re-initialization mechanisms such as vLLM's sleep mode, and that its estimates match the performance trends reported by the vLLM developers (vLLM, 2025).

Design Rationale. Our key insight from §2 is that each startup step exhibits a simple and often near-linear dependency on its corresponding parameters, such as model size for weight loading and graph size for compilation. Leveraging this observation, we design a *white-box decomposed predictor*: a lightweight regressor for each startup step, trained independently using linear regression. This modular formulation preserves interpretability, as the contribution of each parameter remains explicitly visible in its respective step model, while collectively achieving strong predictive accuracy when aggregating across steps.

An alternative approach here would be to build a monolithic black-box predictor that jointly captures the dependencies between all model- and vLLM-specific parameters. While such a global neural predictor could, in principle, learn these intricate interactions, it would increase complexity, and lack interpretability, thus making it difficult to attribute latency variations to specific model factors or startup steps (Molnar, 2020; Lipton, 2018). In contrast, our white-box approach enables us to retrain only specific step-specific regressors when a new model or hardware becomes available.

Validation. We report the accuracy of the predictor. We use all non-MoE models listed in Table 1, reserving Falcon-11B, Gemma-7B, Mistral-7B, Llama3-3B and Qwen-7B for validation and the remaining models for training the predictors. For each model, we ran the vLLM startup process five times, averaged the results, and used the data to fit the step-specific predictors. The final predictions are compared against the measured values in Figure 17. Despite its simplicity, the predictor achieves strong accuracy, with a mean squared error (MSE) of 2.42 secs and a maximum error of 2.08 secs, observed for Llama3-3B. We have also run and validate the predictor on `v0.11`, released on Oct 2nd, and report that our approach and predictor are still accurate (MSE of 2.62 secs), further validating the approach taken in this work for the analysis and modeling of vLLM's startup process.

5 Discussion

Benchmarking Scope. In a distributed inference service (e.g., serverless inference), cold-start latency is the result of multiple interacting factors: (i) distributed-environment overheads, such as networking, remote storage accesses, and container retrieval; (ii) node-local dynamics, including PCIe contention, OS scheduling, and container initialization; and (iii) the inference engine startup itself. Our study focuses on factor (iii), the vLLM engine initialization. We design our experimental setups so that the engine startup becomes the dominant bottleneck, thereby minimizing noise from external system components. The hardware and software stack used in our evaluation reflects a typical mid- to high-end GenAI deployment, where CPU-side processing is the primary contributor to startup latency. While factors (i) and (ii) are important in real-world distributed environments, they are orthogonal to the engine-level dynamics analyzed in this work. A full end-to-end characterization that integrates all factors is an important direction for future research.

Longevity of Insights. vLLM evolves rapidly, with frequent architectural changes that may alter the relative cost of individual startup steps. While such changes may require retraining of affected step-specific regressors, the modular nature of our predictor confines this retraining to localized components rather than the entire model. More importantly, our primary contribution lies in the methodology used to decompose and analyze the startup process, rather than in any specific parameter values. The six foundational steps identified in Figure 2 capture fundamental operations that are inherent to modern inference engines. Even as vLLM continues to evolve, these stages and their relationships to underlying computational and I/O factors remain grounded in persistent system principles. As a result, we expect the decomposition and analysis methodology to remain applicable, with only parameter re-tuning required to adapt to future vLLM versions.

Generalizability. The identified six steps correspond to the core initialization stages that are present across post-V1 vLLM releases, and similar startup phases have been reported in concurrent analyses of containerized inference services (Incubation, 2025). Moreover, our experimental findings remain consistent across a diverse set of configurations, including two GPU types, two CPU platforms, 22 models, and multiple versions of Python and PyTorch. These results suggest that the observed scaling trends and step-level behaviors are not tied to a single hardware or software stack, but instead reflect more general properties of modern LLM inference systems.

Assumption of Linearity. Our predictors assume predominantly linear relationships between model parameters and step latency. While justified by empirical evidences in §2, certain nonlinear behaviors found in MoE, SSM-transformer hybrid, diffusion models require more expressive analysis to capture their performance profile. Fortunately, this analysis is restricted to the KVCache Profiling step only.

Measurement Noise. The accuracy of the proposed predictor is inherently tied to the characteristics of the underlying CPU–GPU hardware and the conditions under which the data was collected. Predictors may need to be re-trained when deployed on substantially different infrastructures. Furthermore, training data was gathered under controlled experimental conditions; real-world deployments may experience background workload interference, NUMA effects, or scheduler contention that introduce additional noise not captured by our measurements. Additionally, timing analysis relies on vLLM's internal logging granularity, which can add up to hundreds of milliseconds of measurement error. To mitigate this, we averaged results over multiple runs, yet minor deviations may remain.

6 Related Work

Prior studies have explored cold start latency in the context of both traditional serverless platforms and emerging LLM inference systems. Early works such as SAND (Akkus et al., 2018) and SEUSS (Fuerst & Sharma, 2021) addressed general-purpose function startup overhead through lightweight containerization and runtime reuse. Other works in this domain focus on forecasting or avoiding cold start *occurrences*, rather than predicting their *duration* (Shiekhani et al., 2025; Golec et al., 2024; Jegannathan et al., 2022; Nguyen et al., 2025). More recent efforts around LLMs, including Sarathi-Serve (Agrawal et al., 2024), Llumnix (Sun et al., 2024), and DistServe (Zhong et al., 2024), investigate the throughput-latency tradeoff in large-scale model serving but primarily focus on steady-state inference rather than startup costs. With the emergence of scalable containerized LLM services, several works target cold start latency as a first-class optimization goal. For example, ParaServe (Lou et al., 2025) focuses on reducing model fetching and initialization delays in serverless LLM serving by overlapping parameter loading across GPU servers and exploiting pipeline parallelism. TIDAL uses fine-grained execution tracing to generate adaptive templates that bypass much of the cold start overhead in serverless settings (Cui et al., 2025). ServerlessLLM introduces techniques such as multi-tier checkpoint loading, locality-aware scheduling, and live migration to minimize delays before readiness (Fu et al., 2024). CSGO targets cold start latency in edge-distributed LLM systems by dynamically partitioning models and overlapping model loading, computation, and communication (Liu et al., 2025). Medusa accelerates serverless LLM inference by materializing frequently used execution states, enabling rapid function startup and reuse across invocations (Zeng et al., 2025).

In comparison to the past literature, our work is the first to systematically decompose and analyze the startup process of vLLM, a popular open-source inference engine used widely. While prior works examined isolated factors such as model loading or CUDA graph capturing, we provide a holistic breakdown across all startup steps and demonstrate the predominance of CPU-bound bottlenecks. Furthermore, our analytical predictor extends prior measurement studies by offering a practical, interpretable model for predicting cold start latency under varying hardware and software configurations. To the best of our knowledge, this is the first predictor that estimates the startup latency of an LLM inference engine in an interpretable and modular manner.

7 Conclusion

In this paper, we presented the first systematic characterization of `vLLM`'s startup latency. By decomposing the startup process into six key steps, we identified dominant bottlenecks and quantified their dependence on different model and hardware configurations. Our analysis showed that startup latency is largely CPU bounded and only marginally affected by GPU performance. Building on these findings, we proposed a regressor-based modular analytical predictor that estimates startup latency with high accuracy using model and system parameters.

In future work, we plan to integrate this predictor into real-world serverless orchestration frameworks to enable proactive cold start mitigation in multi-tenant environments. Additionally, our step-wise decomposition opens directions for further optimization, such as overlapping or parallelizing steps like `torch.compile`, KVCache profiling, and CUDA graph capturing. These extensions can help reduce startup latency even further, advancing toward adaptive, scalable, and low-latency LLM serving systems.

References

01.AI, :, Young, A., Chen, B., Li, C., Huang, C., Zhang, G., Zhang, G., Li, H., Zhu, J., Chen, J., Chang, J., Yu, K., Liu, P., Liu, Q., Yue, S., Yang, S., Yang, S., Yu, T., Xie, W., Huang, W., Hu, X., Ren, X., Niu, X., Nie, P., Xu, Y., Liu, Y., Wang, Y., Cai, Y., Gu, Z., Liu, Z., and Dai, Z. Yi: Open foundation models by 01.ai. https://arxiv.org/abs/2403.04652, 2024.

Agrawal, A., Panwar, A., Mohan, J., Kwatra, N., Gulavani, B. S., and Ramjee, R. Sarathi: Efficient llm inference by piggybacking decodes with chunked prefills. https://arxiv.org/abs/2308.16369, 2023.

Agrawal, A., Kedia, N., Panwar, A., Mohan, J., Kwatra, N., Gulavani, B., Tumanov, A., and Ramjee, R. Taming throughput-latency tradeoff in llm inference with sarathi-serve. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pp. 117–134, 2024.

AIBrix. Aibrix: Cost-efficient and pluggable infrastructure components for genai inference. https://github.com/vllm-project/aibrix, 2025. Accessed: 2026-03-17.

Akkus, I. E., Chen, R., Rimac, I., Stein, M., Satzke, K., Beck, A., Aditya, P., and Hilt, V. Sand: towards high-performance serverless computing. In *2018 USENIX annual technical conference (USENIX ATC 18)*, pp. 923–935, 2018.

Alan Gray. Nvidia: Getting started with cuda graphs. https://developer.nvidia.com/blog/cuda-graphs/, 2019. Accessed: 2026-03-17.

Anastasiya Zharovskikh. Best applications of large language models, 2023. URL https://indatalabs.com/blog/large-language-model-apps. Accessed: 2026-03-17.

Ansel, J., Yang, E., He, H., Gimelshein, N., Jain, A., Voznesensky, M., Bao, B., Bell, P., Berard, D., Burovski, E., et al. Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pp. 929–947, 2024.

Bai, J., Bai, S., Chu, Y., Cui, Z., Dang, K., Deng, X., Fan, Y., Ge, W., Han, Y., Huang, F., Hui, B., Ji, L., Li, M., Lin, J., Lin, R., Liu, D., Liu, G., Lu, C., Lu, K., Ma, J., Men, R., Ren, X., Ren, X., Tan, C., Tan, S., Tu, J., Wang, P., Wang, S., Wang, W., Wu, S., Xu, B., Xu, J., Yang, A., Yang, H., Yang, J., Yang, S., Yao, Y., Yu, B., Yuan, H., Yuan, Z., Zhang, J., Zhang, X., Zhang, Y., Zhang, Z., Zhou, C., Zhou, J., Zhou, X., and Zhu, T. Qwen technical report. https://arxiv.org/abs/2309.16609, 2023.

Balamurugan, B., Alexander, A., Raman, A., Gupta, K., Tan, W., Kreitzer, B., Isaza, E. T., Rao, H., and Liu, J. Accelerate generative ai inference with nvidia dynamo and amazon eks. https://aws.amazon.com/blogs/machine-learning/accelerate-generative-ai-inference-with-nvidia-dynamo-and-amazon-eks/, 2025. Accessed: 2025-10-30.

CellStrat. Real-world use cases for large language models (llms), 2023. URL https://cellstrat.medium.com/real-world-use-cases-for-large-language-models-llms-d71c3a577bf2. Accessed: 2026-03-17.

Chen, H., Li, X., Qian, K., Guan, Y., Zhao, J., and Wang, X. Gyges: Dynamic cross-instance parallelism transformation for efficient llm inference. https://arxiv.org/abs/2509.19729, 2025.

CoreWeave. Coreweave's tensorizer: Module, model, and tensor serialization/deserialization. https://github.com/coreweave/tensorizer, 2025. Accessed: 2026-03-17.

Cortez, E., Bonde, A., Muzio, A., Russinovich, M., Fontoura, M., and Bianchini, R. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pp. 153–167, 2017.

CUDA Graphs. Nvidia. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#cuda-graphs, 2025. Accessed: 2026-03-17.

Cui, W., Xu, Z., Zhao, H., Chen, Q., Li, Z., He, B., and Guo, M. Efficient function-as-a-service for large language models with tidal. https://arxiv.org/abs/2503.06421, 2025.

Daivi. 7 top large language model use cases and applications, 2024. URL https://www.projectpro.io/article/large-language-model-use-cases-and-applications/887. Accessed: 2026-03-17.

DeepSeek-AI. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. https://arxiv.org/abs/2405.04434, 2024.

DeepSeek-AI. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. https://arxiv.org/abs/2501.12948, 2025.

Du, D., Yu, T., Xia, Y., Zang, B., Yan, G., Qin, C., Wu, Q., and Chen, H. 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*, pp. 467–481, 2020.

Fu, Y., Xue, L., Huang, Y., Brabete, A.-O., Ustiugov, D., Patel, Y., and Mai, L. Serverlessllm: Low-latency serverless inference for large language models. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pp. 135–153, 2024.

Fuerst, A. and Sharma, P. Faascache: keeping serverless computing alive with greedy-dual caching. In *Proceedings of the 26th ACM international conference on architectural support for programming languages and operating systems*, pp. 386–400, 2021.

Golec, M., Walia, G. K., Kumar, M., Cuadrado, F., Gill, S. S., and Uhlig, S. Cold start latency in serverless computing: A systematic review, taxonomy, and future directions. *ACM Computing Surveys*, 57(3):1–36, 2024.

Google. gemma-7b. https://huggingface.co/google/gemma-7b, 2024. Accessed: 2026-03-17.

Gordic, A. Inside vLLM: Anatomy of a High-Throughput LLM Inference System. https://www.aleksagordic.com/blog/vllm/, 2025. Accessed: 2026-03-17.

Granite Team, IBM. Granite-3.3-8B-Instruct. https://huggingface.co/ibm-granite/granite-3.3-8b-instruct, 2025a. Accessed: 2026-03-17.

Granite Team, IBM. Granite-4.0-h-micro. https://huggingface.co/ibm-granite/granite-4.0-h-micro, 2025b. Accessed: 2026-03-17.

Granite Team, IBM. Granite-4.0-h-small. https://huggingface.co/ibm-granite/granite-4.0-h-small, 2025c. Accessed: 2026-03-17.

Hex. vllm weekly usage stats. https://app.hex.tech/019c4540-72b8-7005-9d68-08e0191ac583/app/vLLM-Weekly-Usage-Stats-032Vh7ZNLdI3OI2hNYJaPv/latest, 2026. Accessed: 2026-03-17.

Hu, J., Xu, J., Liu, Z., He, Y., Chen, Y., Xu, H., Liu, J., Meng, J., Zhang, B., Wan, S., Dan, G., Dong, Z., Ren, Z., Liu, C., Xie, T., Lin, D., Zhang, Q., Yu, Y., Feng, H., Chen, X., and Shan, Y. Deepserve: serverless large language model serving at scale. In *Proceedings of the 2025 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '25, USA, 2025. USENIX Association. ISBN 978-1-939133-48-9.

Hu, Q., Ye, Z., Wang, Z., Wang, G., Zhang, M., Chen, Q., Sun, P., Lin, D., Wang, X., Luo, Y., et al. Characterization of large language model development in the datacenter. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pp. 709–729, 2024.

Huang, H., Ardalani, N., Sun, A., Ke, L., Lee, H.-H. S., Bhosale, S., Wu, C.-J., and Lee, B. Toward efficient inference for mixture of experts. *Advances in Neural Information Processing Systems*, 37:84033–84059, 2024.

HuggingFace. Safetesnors: Ml safer for all. https://github.com/huggingface/safetensors, 2025. Accessed: 2026-03-17.

Incubation, L.-D. Llm-d benchmarking tool. https://github.com/llm-d/llm-d-benchmark, 2025. Accessed: 2026-03-17.

Jegannathan, A. P., Saha, R., and Addya, S. K. A time series forecasting approach to minimize cold start time in cloud-serverless platform. In *2022 IEEE International black sea conference on communications and networking (BlackSeaCom)*, pp. 325–330. IEEE, 2022.

Jiang, A. Q., Sablayrolles, A., Mensch, A., Bamford, C., Chaplot, D. S., de las Casas, D., Bressand, F., Lengyel, G., Lample, G., Saulnier, L., Lavaud, L. R., Lachaux, M.-A., Stock, P., Scao, T. L., Lavril, T., Wang, T., Lacroix, T., and Sayed, W. E. Mistral 7b. https://arxiv.org/abs/2310.06825, 2023.

Khare, A., Garg, D., Kalra, S., Grandhi, S., Stoica, I., and Tumanov, A. Superserve: fine-grained inference serving for unpredictable workloads. In *Proceedings of the 22nd USENIX Symposium on Networked Systems Design and Implementation*, NSDI '25, USA, 2025. USENIX Association. ISBN 978-1-939133-46-5.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J. E., Zhang, H., and Stoica, I. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

Li, Z., Guo, L., Chen, Q., Cheng, J., Xu, C., Zeng, D., Song, Z., Ma, T., Yang, Y., Li, C., et al. Help rather than recycle: Alleviating cold startup in serverless computing through inter-function container sharing. In *2022 USENIX annual technical conference (USENIX ATC 22)*, pp. 69–84, 2022.

Lipton, Z. C. The mythos of model interpretability: In machine learning, the concept of interpretability is both important and slippery. *Queue*, 16(3):31–57, 2018.

Liquid Web. A100 vs h100 vs l40s: A simple side-by-side and how to decide. https://www.liquidweb.com/gpu/a100-vs-h100-vs-l40s/, 2025. Accessed: 2026-03-17.

Liu, X., Xue, N., Bao, R., Sun, Y., Chen, Z., Tao, M., Xu, X., and Cui, S. Csgo: Generalized optimization for cold start in wireless collaborative edge llm systems. https://arxiv.org/abs/2508.11287, 2025.

LLM-D. Workload variant autoscaler. https://github.com/llm-d-incubation/workload-variant-autoscaler, 2025. Accessed: 2026-03-17.

Lou, C., Qi, S., Jin, C., Nie, D., Yang, H., Liu, X., and Jin, X. Towards swift serverless llm cold starts with paraserve. https://arxiv.org/abs/2502.15524v1, 2025.

Malartic, Q., Chowdhury, N. R., Cojocaru, R., Farooq, M., Campesan, G., Djilali, Y. A. D., Narayan, S., Singh, A., Velikanov, M., Boussaha, B. E. A., et al. Falcon2-11b technical report. https://arxiv.org/abs/2407.14885, 2024.

Meta AI. Llama 3.2 - 3b model. https://huggingface.co/meta-llama/Llama-3.2-3B, 2023. Accessed: 2026-03-17.

Molnar, C. *Interpretable machine learning*. Lulu. com, 2020.

Mu, S. and Lin, S. A comprehensive survey of mixture-of-experts: Algorithms, theory, and applications. https://arxiv.org/abs/2503.07137, 2025.

Nar, F. E., Pereira, G., Tang, Y., Shaw, R., and Asthana, A. Why vllm is the best choice for ai inference today, 2025. URL https://developers.redhat.com/articles/2025/10/30/why-vllm-best-choice-ai-inference-today. Accessed: 2026-03-17.

Nguyen, C., Bhuyan, M., and Elmroth, E. Taming cold starts: Proactive serverless scheduling with model predictive control. https://arxiv.org/abs/2508.07640, 2025.

NVIDIA. *NVIDIA CUDA C Programming Guide*. NVIDIA, 2025a. Accessed: 2026-03-17.

NVIDIA. Nvidia dynamo platform. https://developer.nvidia.com/dynamo, 2025b. Accessed: 2026-03-17.

NVIDIA. Nvidia dynamo: Adaptive load planning and gpu worker autoscaling. https://docs.nvidia.com/dynamo/latest/architecture/load_planner.html, 2025c. Accessed: 2026-03-17.

NVIDIA. Nvidia h100 tensor core gpu. https://www.nvidia.com/en-us/data-center/h100/, 2025d. Accessed: 2026-03-17.

NVIDIA. Nvidia l40s gpu. https://www.nvidia.com/en-us/data-center/l40s/, 2025e. Accessed: 2026-03-17.

Oakes, E., Yang, L., Zhou, D., Houck, K., Harter, T., Arpaci-Dusseau, A., and Arpaci-Dusseau, R. Sock: Rapid task provisioning with serverless-optimized containers. In *2018 USENIX annual technical conference (USENIX ATC 18)*, pp. 57–70, 2018.

OpenAI. gpt-oss-120b and gpt-oss-20b model card. https://arxiv.org/abs/2508.10925, 2025.

pepy.tech. pepy.tech: Python Package Download Statistics. https://pepy.tech, 2025. Accessed: 2026-03-17.

Puneet Mangla. What's behind pytorch 2.0? torchdynamo and torchinductor (primarily for developers). https://pyimagesearch.com/2023/04/24/whats-behind-pytorch-2-0-torchdynamo-and-torchinductor-primarily-for-developers/, 2023. Accessed: 2026-03-17.

PyTorch. torch.compiler overview. https://docs.pytorch.org/docs/stable/torch.compiler.html, 2023. Accessed: 2026-03-17.

PyTorch. Dynamo deep-dive. https://docs.pytorch.org/docs/stable/torch.compiler_dynamo_deepdive.html, 2024. Accessed: 2026-03-17.

PyTorch. Dynamo overview. https://docs.pytorch.org/docs/stable/torch.compiler_dynamo_overview.html, 2025a. Accessed: 2026-03-17.

PyTorch. torch.compiler. https://docs.pytorch.org/docs/stable/torch.compiler.html, 2025b. Accessed: 2026-03-17.

PyTorch. Writing graph transformations on aten ir. https://docs.pytorch.org/docs/stable/torch.compiler_transformations.html, 2025c. Accessed: 2026-03-17.

Qin, R., Li, Z., He, W., Cui, J., Ren, F., Zhang, M., Wu, Y., Zheng, W., and Xu, X. Mooncake: Trading more storage for less computation — a KVCache-centric architecture for serving LLM chatbot. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*, pp. 155–170, Santa Clara, CA, February 2025. USENIX Association. ISBN 978-1-939133-45-8. URL https://www.usenix.org/conference/fast25/presentation/qin.

Qwen. Qwen1.5-moe-a2.7b. https://huggingface.co/Qwen/Qwen1.5-MoE-A2.7B, 2024. Accessed: 2026-03-17.

Red Hat. Llm-d: Distributed large language model deployment framework. https://llm-d.ai/, 2025. Accessed: 2026-03-17.

Roy, R. B., Patel, T., and Tiwari, D. Icebreaker: Warming serverless functions better with heterogeneity. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 753–767, 2022.

Run-AI. Run:ai model streamer, 2025. URL https://github.com/run-ai/runai-model-streamer/. Accessed: 2026-03-17.

Shiekhani, L., Wang, H., Shi, W., Liu, J., Qiu, Y., Gu, C., and Ding, W. The hybrid model: Prediction-based scheduling and efficient resource management in a serverless environment. *Applied Sciences*, 15(14):7632, 2025.

Singh, A. K. and Strouse, D. Tokenization counts: the impact of tokenization on arithmetic in frontier llms. https://arxiv.org/abs/2402.14903, 2024.

Sun, B., Huang, Z., Zhao, H., Xiao, W., Zhang, X., Li, Y., and Lin, W. Llumnix: Dynamic scheduling for large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, 2024.

tiiuae. Falcon-7b. https://huggingface.co/tiiuae/falcon-7b, 2023. Accessed: 2026-03-17.

Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., et al. Llama 2: Open foundation and fine-tuned chat models. https://arxiv.org/abs/2307.09288, 2023.

Vast AI. Nvidia h100 vs. l40s: Power meets versatility. https://vast.ai/article/nvidia-h100-vs-l40s-power-meets-versatility, 2024. Accessed: 2026-03-17.

vLLM. Zero-reload model switching with vllm sleep mode. https://blog.vllm.ai/2025/10/26/sleep-mode.html, 2025. Accessed: 2026-03-17.

vLLM Docs. torch.compile integration. https://docs.vllm.ai/en/latest/design/torch_compile.html, 2025. Accessed: 2026-03-17.

vLLM Production Stack. Reference stack for production vLLM deployment. https://github.com/vllm-project/production-stack, 2025. Accessed: 2026-03-17.

vLLM Project. Performance analysis. GitHub Issue #23787. https://github.com/vllm-project/vllm/issues/23787, 2025a. Accessed: 2026-03-17.

vLLM Project. First call to llama model takes too much time. GitHub Issue #29676. https://github.com/vllm-project/vllm/issues/29676, 2025b. Accessed: 2026-03-17.

vLLM Project. vllm-compile warm-start time should be close to zero. GitHub Issue #20402. https://github.com/vllm-project/vllm/issues/20402, 2025c. Accessed: 2026-03-17.

vLLM Project. Improve startup time ux. GitHub Issue #19824. https://github.com/vllm-project/vllm/issues/19824, 2025d. Accessed: 2026-03-17.

vLLM Project. Add opentelemetry tracing for vllm start up phases. GitHub Issue #19318. https://github.com/vllm-project/vllm/issues/19318, 2025e. Accessed: 2026-03-17.

vLLM Project. vllm 2024 retrospective and 2025 vision. https://blog.vllm.ai/2025/01/10/vllm-2024-wrapped-2025-vision.html, 2025f. Accessed: 2026-03-17.

vLLM Project. Add a script to benchmark compilation time. https://github.com/vllm-project/vllm/pull/29919, 2025g. Accessed: 2026-03-17.

vLLM Project. Deprecation of vllm v0. GitHub Issue #18571. https://github.com/vllm-project/vllm/issues/18571, 2025h. Accessed: 2026-03-17.

vLLM Project. ModelClass Caching, Generate _ModelInfo properties file when loading to improve loading speed. GitHub PR #23558, https://github.com/vllm-project/vllm/pull/23558, 2025i. [Online; accessed 01-Oct-2025].

vLLM Project. Improve startup time ux in vllm. GitHub Issue #19824. https://github.com/vllm-project/vllm/issues/19824, 2025j. Accessed: 2026-03-17.

vLLM Project. vllm v1: A major upgrade to vllm's core architecture. https://blog.vllm.ai/2025/01/27/v1-alpha-release.html, 2025k. Accessed: 2026-03-17.

vLLM Project. Introduction to torch.compile and how it works with vllm. https://https://blog.vllm.ai/2025/08/20/torch-compile.html, 2025l. Accessed: 2026-03-17.

Yu, G.-I., Jeong, J. S., Kim, G.-W., Kim, S., and Chun, B.-G. Orca: A distributed serving system for {Transformer-Based} generative models. In *16th USENIX symposium*

on operating systems design and implementation (OSDI 22), pp. 521–538, 2022.

Yu, H., Basu Roy, R., Fontenot, C., Tiwari, D., Li, J., Zhang, H., Wang, H., and Park, S.-J. Rainbowcake: Mitigating cold-starts in serverless with layer-wise container caching and sharing. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pp. 335–350, 2024.

Yu, S., Xing, J., Qiao, Y., Ma, M., Li, Y., Wang, Y., Yang, S., Xie, Z., Cao, S., Bao, K., et al. Prism: Unleashing gpu sharing for cost-efficient multi-llm serving. https://arxiv.org/abs/2505.04021, 2025.

Zeng, S., Xie, M., Gao, S., Chen, Y., and Lu, Y. 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*, pp. 653–668, 2025.

Zhong, Y., Liu, S., Chen, J., Hu, J., Zhu, Y., Liu, X., Jin, X., and Zhang, H. Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving. https://arxiv.org/abs/2401.09670, 2024.

Notes: IBM is a trademark of International Business Machines Corporation, registered in many jurisdictions worldwide. Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries. Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both. Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates. Other products and service names might be trademarks of IBM or other companies.

A Artifact Appendix

A.1 Abstract

The artifact contains code and scripts required to reproduce the profiling experiments presented in the paper. The artifact is based on vLLM `v0.10.1.1` with additional profiling logs. It includes scripts to download all evaluated LLMs mentioned in Table 1, apply modifications to the vLLM runtime to include additional logs, and reproduce all figures from the paper automatically.

The provided workflow installs dependencies, downloads the required models, and executes figure-specific scripts to reproduce the experimental results. Each figure can be reproduced using a unified script interface. Some figures require multiple GPUs or machines to match the experimental setup.

A.2 Artifact check-list (meta-information)

- **Program:** Python, vLLM-based profiling scripts
- **Binary:** Pretrained LLM models
- **Model:** Multiple HuggingFace LLMs (∼600GB total)
- **Run-time environment:** Linux, Python 3.11, CUDA 12.x
- **Hardware:** NVIDIA H100, L40S GPUs; AMD EPYC and Intel Xeon CPUs
- **Execution:** Bash & Python scripts per figure
- **Metrics:** Throughput, latency, memory, and loading times
- **Output:** PDF figures
- **Experiments:** Figure reproduction scripts
- **How much disk space required (approximately)?:** ∼600GB
- **How much time is needed to prepare workflow (approximately)?:** 1–2 hours (model downloads)
- **How much time is needed to complete experiments (approximately)?:** Several hours to a day depending on hardware
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** Apache 2.0
- **Data licenses (if publicly available)?:** HuggingFace model licenses
- **Workflow automation framework used?:** Bash and Python scripts
- **Archived (provide DOI)?:** 10.5281/zenodo.19591523

A.3 Description

A.3.1 How to access

The artifact is publicly available on GitHub:

https://github.com/upb-cn/vllm-startup-profiler

A.3.2 Hardware & Software dependencies

The experiments were conducted on multiple GPUs and CPUs. The detailed configurations used are summarized in Table 2

A.3.3 Models

The artifact uses pretrained LLM weights downloaded from HuggingFace. The full set of models requires approximately 600 GB of disk space. A HuggingFace access token is required, and some models may require manual access approval.

Models are downloaded via:

```
python3 download_models.py <hf_token>
```

A.4 Installation

1. Clone the repository:

```
git clone https://github.com/upb-cn/vllm-startup-profiler
cd vllm-startup-profiler
```

2. Install dependencies:

```
pip install -r requirements.txt
```

3. Apply custom vLLM modifications:

```
python3 apply_vllm_changes.py
```

4. Download all required models:

```
python3 download_models.py <hf_token>
```

A.5 Experiment workflow

All figures can be reproduced using:

```
cd figures
bash run_figure.sh <num>
```

Where `<num>` is one of:

```
1, 2, 7, 9, 10, 11, 12, 13, 14,
15, 17, rest
```

The output figure will be generated at:

```
figures/figure-<num>/figure<num>.pdf
```

Special cases:

- **Figure 1:** Requires multiple vLLM versions using virtual environments.

- **Figure 10:** Requires two GPUs or two machines.
- **Figure 11:** Requires two different CPU systems.
- **Figure 13:** Compares RAM vs. SSD loading and requires cache clearing with sudo privileges.

A.6 Evaluation and expected results

Each script produces a PDF figure corresponding to the figures presented in the paper.

Due to hardware differences, results may exhibit small numerical variations. However, the overall trends, relative performance differences, and main conclusions should match those reported in the paper.

Successful reproduction is confirmed when:

- Scripts complete without errors.
- Output figures are generated.
- Trends align with the published results.

A.7 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-and-badging-current
- https://cTuning.org/ae