Breaking the Ice: Analyzing Cold Start Latency in vLLM

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

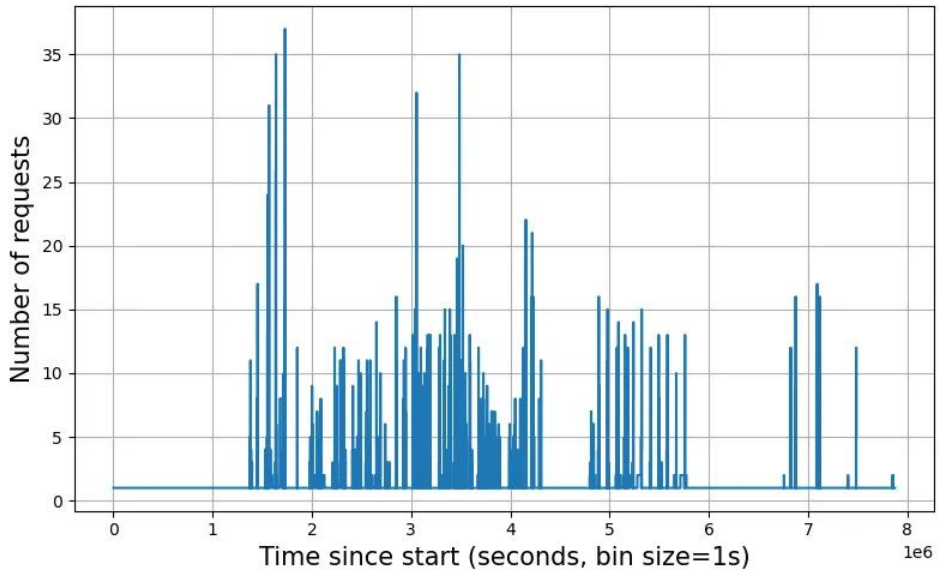
1. Paderborn University

2. IBM Research Europe, Zurich

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

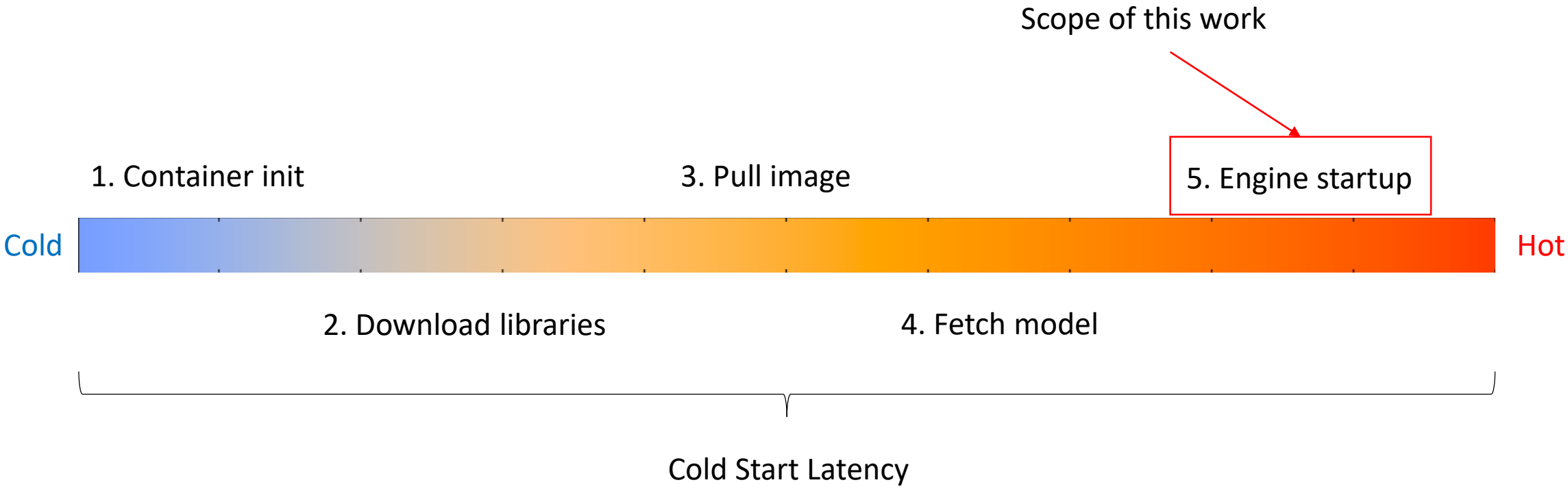
Motivation

Shanghai AI Labs trace



- LLMs are widely used among different domains (e.g., finance, coding, medical)
- Deploying at scale has challenges (e.g., GPU resource provisioning, request scheduling)
- Serverless computing emerged as an attracting paradigm
- Users upload their function, and providers handle autoscaling automatically
- This enables pay-as-you-go pricing that reduces both cost and operational complexity.
- **Problem:** Cold start latency, the time from user request until first token.
- LLM requests arrive in bursts, forcing cloud providers to spin up new instances frequently.

Scope


Scope of this work
1. Container init
3. Pull image
5. Engine startup
Cold
Hot
2. Download libraries
4. Fetch model
Cold Start Latency

Goals

Characterize End-to-End

Breakdown the startup process and identify key performance bottlenecks

Sneak peak →

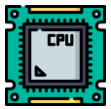
Break the startup process into 6 different steps

Identify Dependencies

Map each step to model & hardware parameters and identify linear relationships

→

Linear relationship with different model params (with few exceptions)



CPU vs. GPU

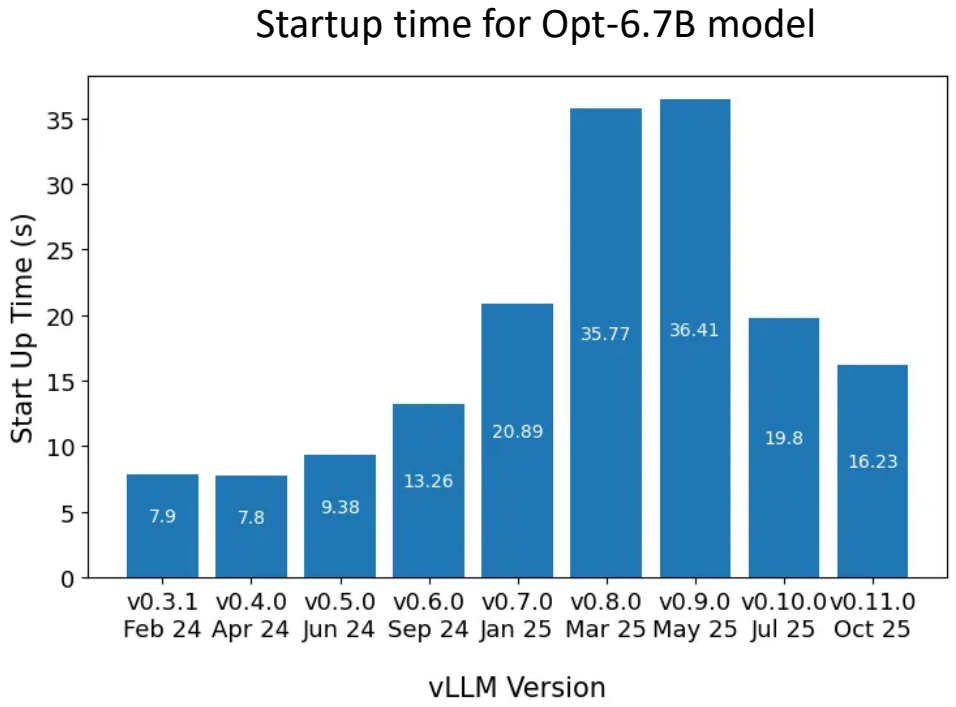
Determine whether each step is CPU-bound or GPU-bound

→

Startup process is predominantly CPU-bound!

Challenges

- vLLM's rapid evolution and growing complexity make consistent analysis difficult
- Heterogeneous inference ecosystem with varying hardware and software dependencies



Agenda

1. Methodology Overview
2. Discovery Process
3. Six Foundational Steps
   - Framework Bootstrap
   - Tokenizer Initialisation
   - Model Loading
   - Torch Compilation
   - KV Cache Profiling
   - CUDA Graph Capturing
4. CPU- or GPU-bound
5. Analytical Predictor
6. Takeaways

Methodology Overview

01

Decompose Startup Process

Break down initialization into six foundational steps

02

Identify Patterns

Identify trends relative to model and system parameters

03

Characterize Bottlenecks

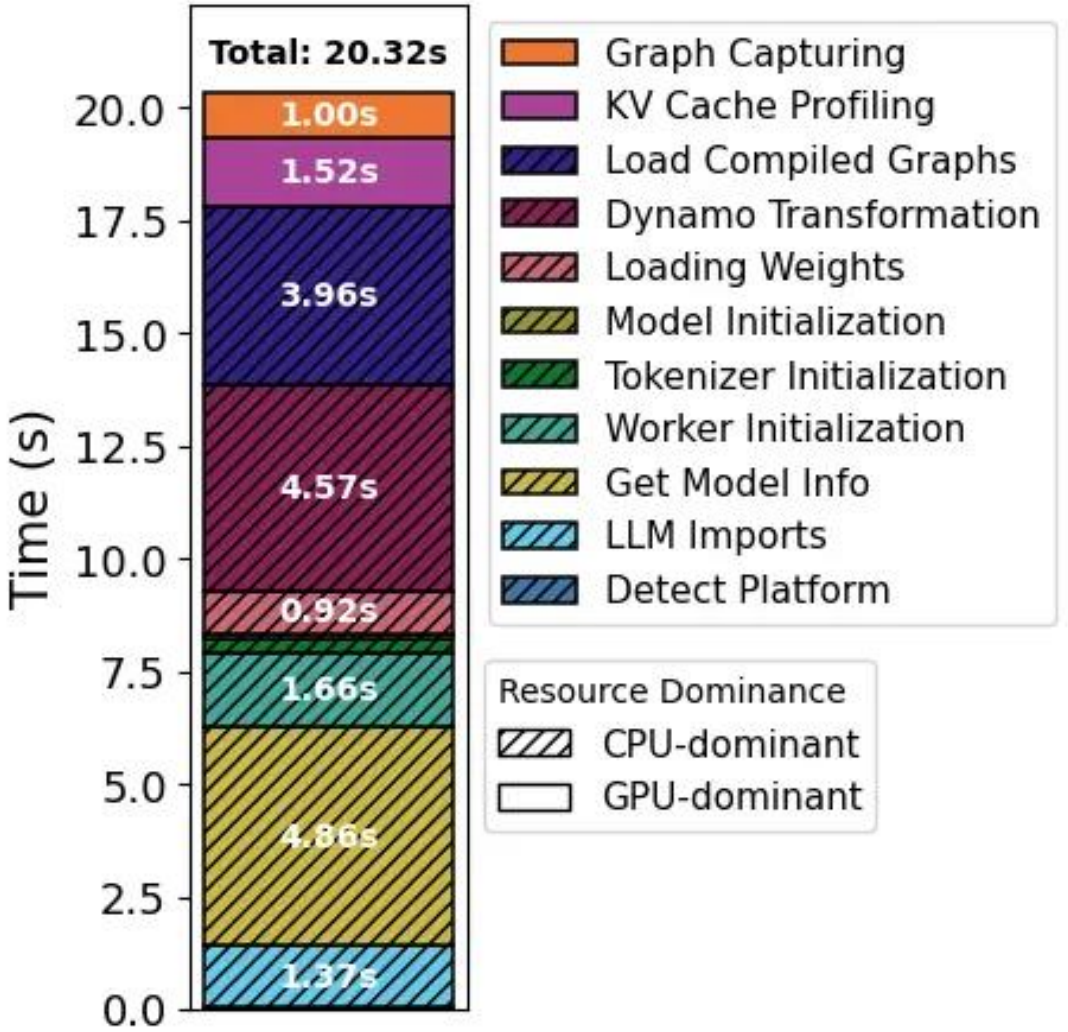
Demonstrate the process is predominantly CPU-bound

04

Build Predictor

Develop lightweight analytical model for accurate latency prediction

Cold start latency breakdown of Llama3.2-3B



Experimental Discovery Process

Systematic Testing

- We executed the startup process across different models with varying configurations, testing on diverse hardware platforms including different GPUs, CPUs, storage, and environment variables.
- For each test, we gathered detailed model configurations, analyzed output logs, and systematically matched each initialization step to specific architectural parameters.

Model Diversity

- LLaMA, Falcon, Qwen families
- Mistral, GPT, DeepSeek variants
- Full MHA, GQA, MQA attention
- Standard and MoE architectures

Our comprehensive model set ranged from 0.5B to 32B parameters, spanning multiple attention mechanisms and architectural patterns to ensure broad applicability of findings.

Table of Models

Model	Layers	Hidden Size	FFN Dim	Heads (Q / KV)	Attention Type	Vocab Size	Tokenizer Size	MoE
LLaMA 2-7B	32	4096	11008	32 / 32	Full MHA	32K	1.8MB	-
LLaMA 2-13B	40	5120	13824	40 / 40	Full MHA			-

Model	Layers	Hidden Size	FFN Dim	Heads (Q / KV)	Attention Type	Vocab Size	Tokenizer Size	MoE
LLaMA 2-7B	32	4096	11008	32 / 32	Full MHA	32K	1.8MB	-
LLaMA 2-13B	40	5120	13824	40 / 40	Full MHA			-
LLaMA 3-3B	28	3072	8192	24 / 8	GQA (3:1)	128K	8.7MB	-
Llama3.1-8B-Instruct	32	4096	14336	32 / 8	GQA (4:1)			-
Falcon-7B	32	4544	18176	71 / 1	MQA	65K	2.7MB	-
Falcon-11B	60	4096	16384	32 / 8	GQA (4:1)			-
Qwen 0.5B	24	1024	2816	16 / 16	Full MHA	152K	6.8MB	-
Qwen 1.8B	24	2048	5504	16 / 16	Full MHA			-
Qwen 4B	40	2560	6912	20 / 20	Full MHA			-
Qwen 7B	32	4096	11008	32 / 32	Full MHA			-
Qwen 14B	40	5120	13696	40 / 40	Full MHA			-
Qwen 14.3B 2.7A	24	2048	1408	16 / 16	Full MHA			60 experts, 4 active (2.7B)
Yi-6B	32	4096	11008	32 / 4	GQA (8:1)	64K	3.5MB	-
Yi-9B	48	4096	11008	32 / 4	GQA (8:1)			-
Mistral-7B	32	4096	14336	32 / 8	GQA (4:1)	32K	1.8MB	-
MPT-7B	32	4096	16384	32 / 32	Full MHA	50K	2.1MB	-
GPT-OSS-20B	24	2880	2880	64 / 8	GQA (8:1)	201K	27MB	32 experts, 4 active (3.6B)
DeepSeek-V2-Lite-16B	27	2048	1408 (first layer is 10944)	16 / 16	MLA	102K	4.4MB	64 experts, 2 shared, 6 active (2.4B)
DeepSeek-R1-Distill-Llama-8B	32	4096	14336	32 / 8	GQA (4:1)	128K	8.7MB	-
DeepSeek-R1-Distill-Qwen-7B	28	3584	18944	28 / 4	GQA (7:1)	152K	6.8MB	-
OLMoE-1B-7B	16	2048	1024	16 / 16	Full MHA	50K	2.1MB	64 experts, 8 active (1B)
Gemma-7B	28	3072	24576	16 / 16	Full MHA	256K	17.5MB	
Granite3.3-8B-Instruct	40	4096	12800	32 / 8	GQA (4:1)	50k	3.4MB	-
Granite4-h-32B	40	4096	1536	32 / 8	GQA (4:1)	100K	6.9MB	72 experts, 10 active (9B)
Granite4-h-3B	40	2048	8192	32 / 8	GQA (4:1)			-

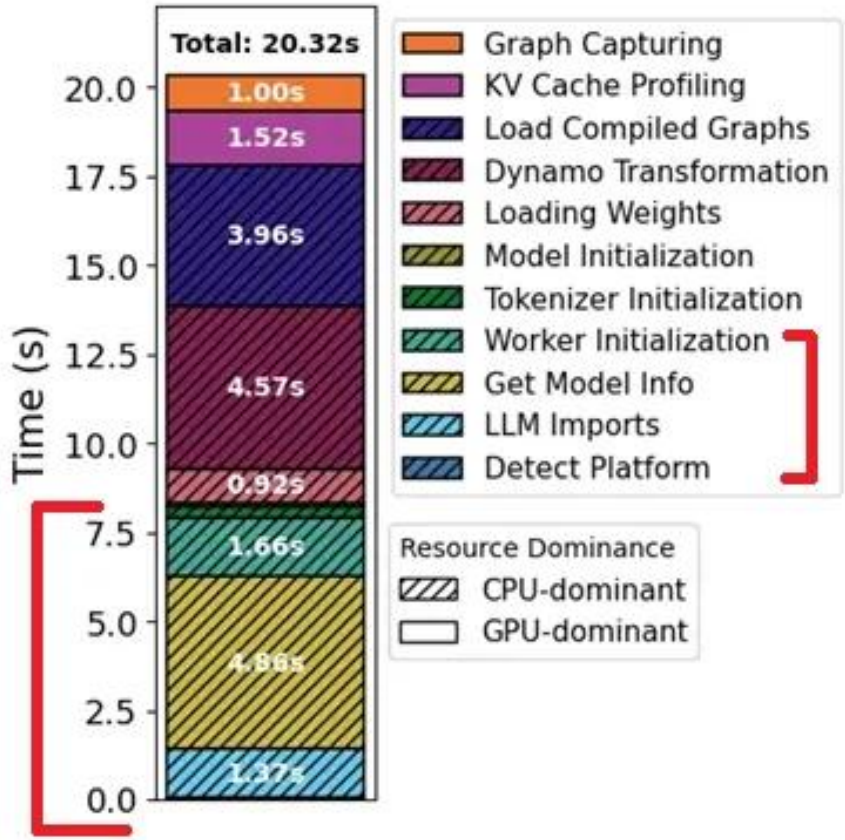
Experimental Setup

- All experiments are conducted on **n1** with **H100** GPU
- We also use L40S GPU on **n1** for GPU comparison, and **n2** for CPU comparison

Node	n1	n2
CPU	AMD EPYC 9354 (32C)	2x Intel Xeon Platinum 8568Y+ (2x48C)
GPU	H100 & L40S	H100
DRAM	DDR5 251GB	DDR5 2TB
Python	3.11.2	3.12
CUDA	12.6	12.8
PyTorch	2.7.1+cu126	2.7.1+cu126
vLLM	0.10.1.1	0.10.1.1

Step 1: Framework Bootstrap

1. **Platform Detection**
   Detect CUDA or CPU execution environment
2. **Import Dependencies**
   Load PyTorch, Transformers, and core libraries
3. **Model Metadata**
   Read model configs and tokenizer metadata
4. **Worker Setup**
   Spawn GPU contexts and shared memory structures

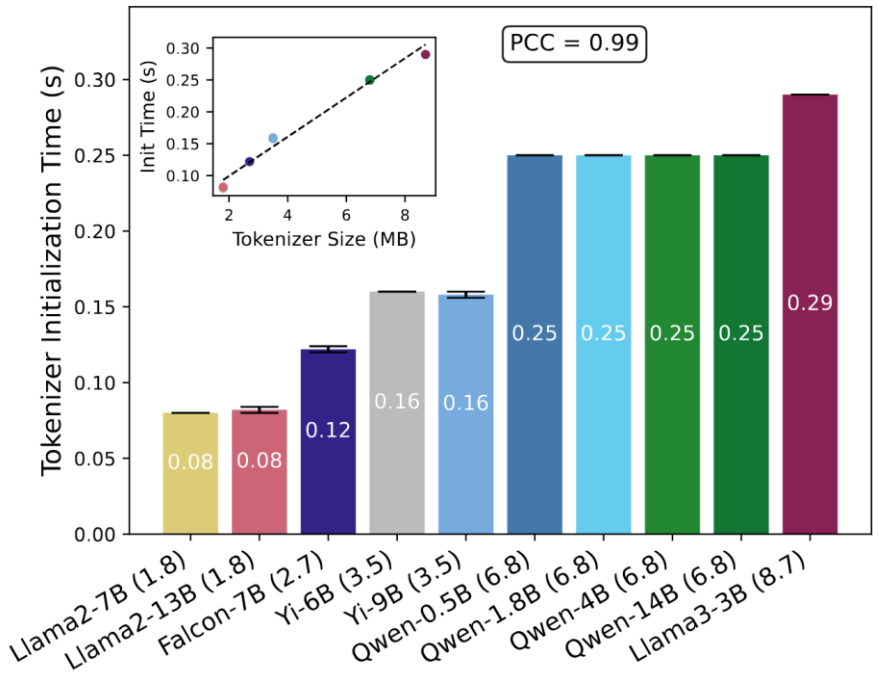


This step is constant, and depends mainly on vLLM's implementation

Steps 2-3: Tokenizer and Model Loading

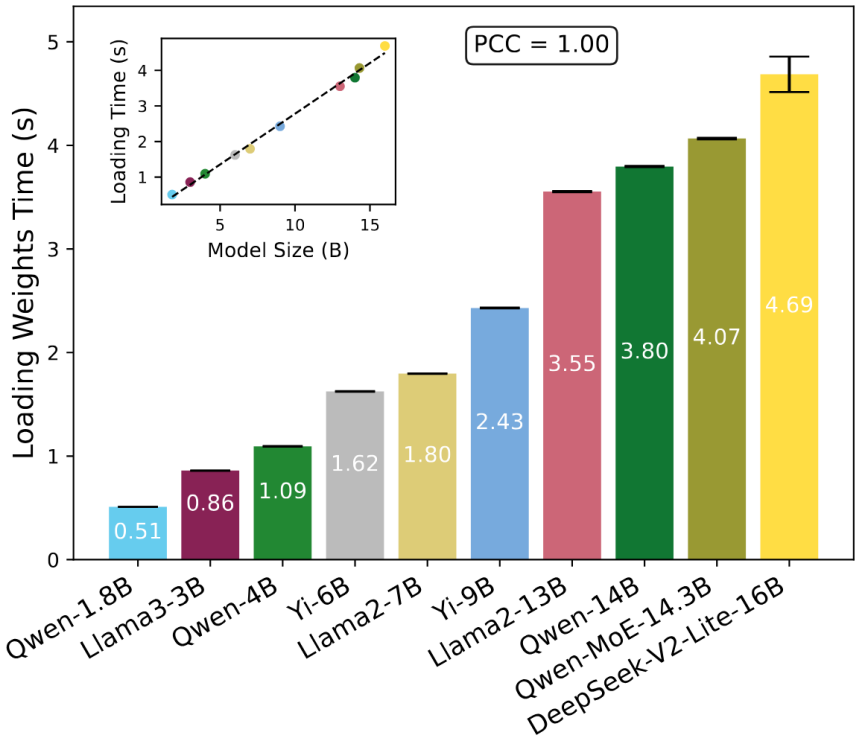
Tokenizer Initialization

Loads tokenizer configuration and vocabulary into memory. Analysis revealed initialization time scales linearly with tokenizer size, which is determined by vocabulary size



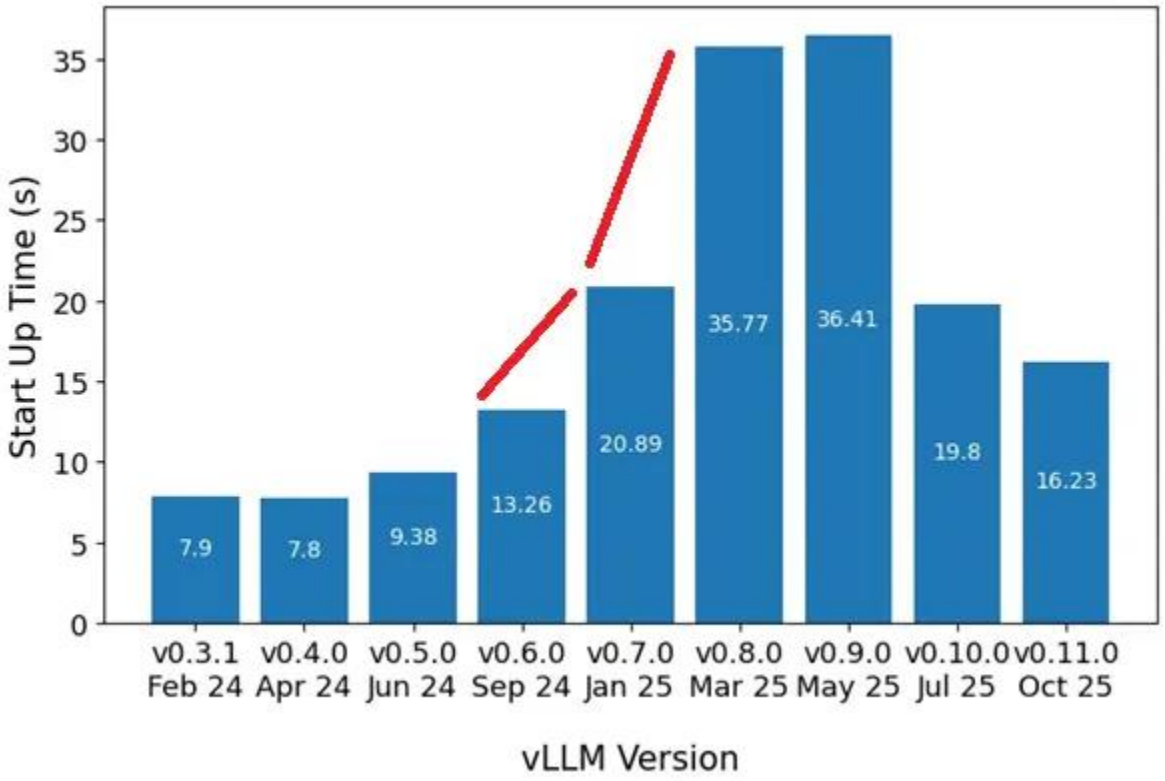
Model Loading

Creates model structure and loads weights into GPU memory. As expected, loading time exhibits strong linear dependence on total parameter count across all tested architectures.



Step 4: Torch Compilation

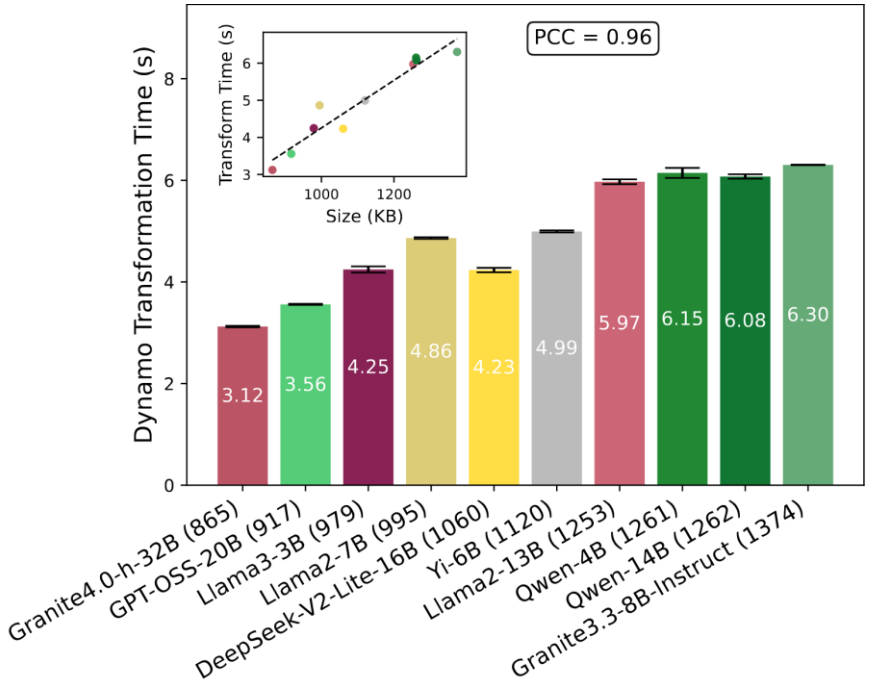
- This step was introduced in v0.7.0, almost a year ago
- This step has helped to improved throughput, but significantly increased latency
- It's split into 2 main sub-steps:
  - Dynamo Transformation
  - Graph Compilation



Step 4: Torch Compilation - Cont

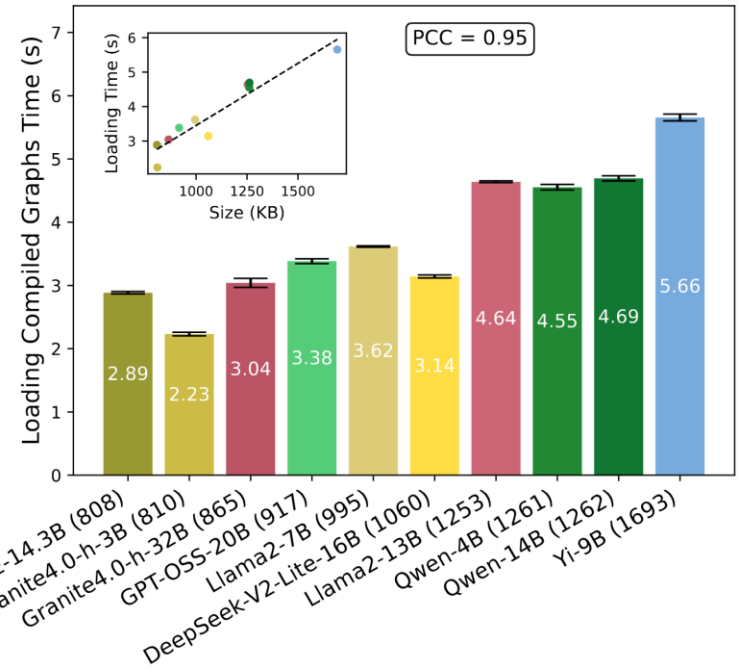
Dynamo Transformation

Transforms Python code into intermediate representation for optimization. Generates computation graph per layer. Time correlates with layer count and complexity.



Graph Compilation

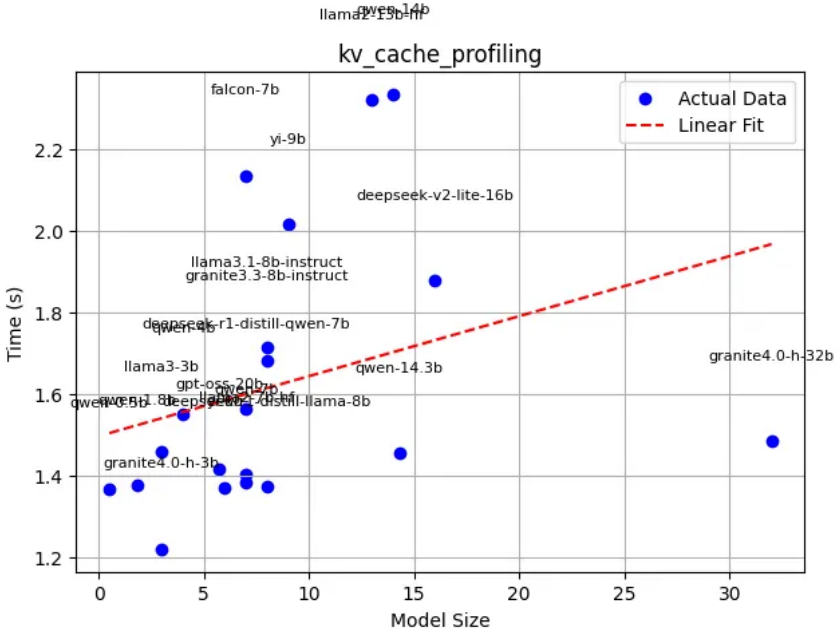
TorchInductor compiles IR graphs into optimized GPU kernels. Compilation time increases with number of layers and their complexity.



Step 5: KV Cache Profiling

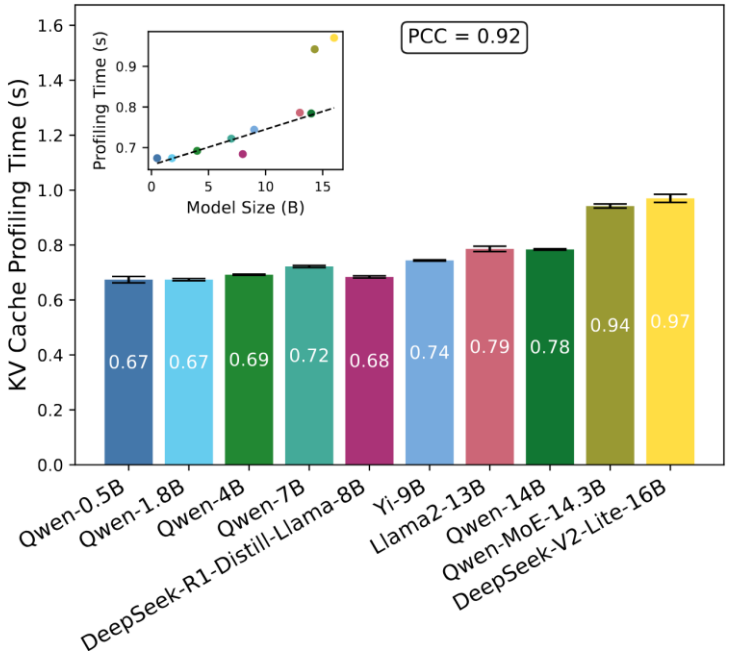
Early Analysis

vLLM runs a dummy run to profile the optimal amount of GPU memory to allocate for the KV cache.
During early experiments, this step included the timing of torch.compile



Findings

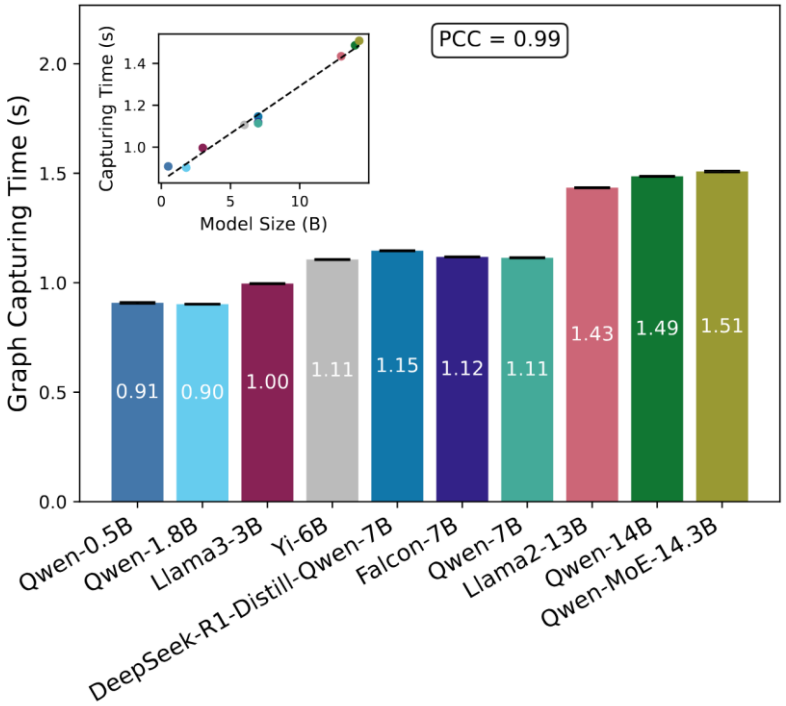
To get a better idea, we run experiments while disabling torch.compile
KV cache profiling scales linearly with the transformer model sizes, while MoE models deviate from this trend



Step 6: CUDA Graph Capturing

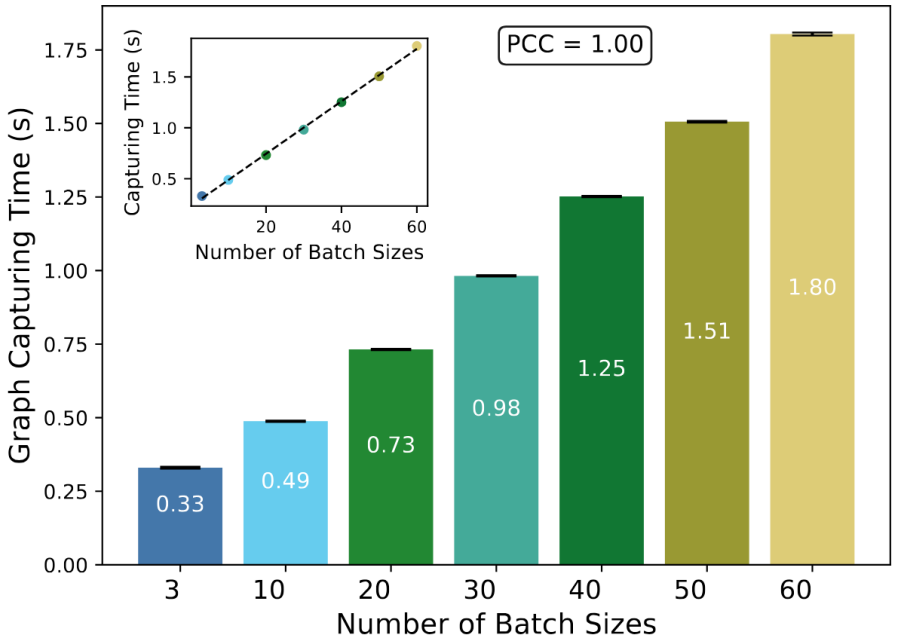
Relation to model size

vLLM performs a dummy run to encode all operations into a CUDA graph, which reduced the overhead of lunching each kernel individually.
As expected, larger models need longer, due to dummy run

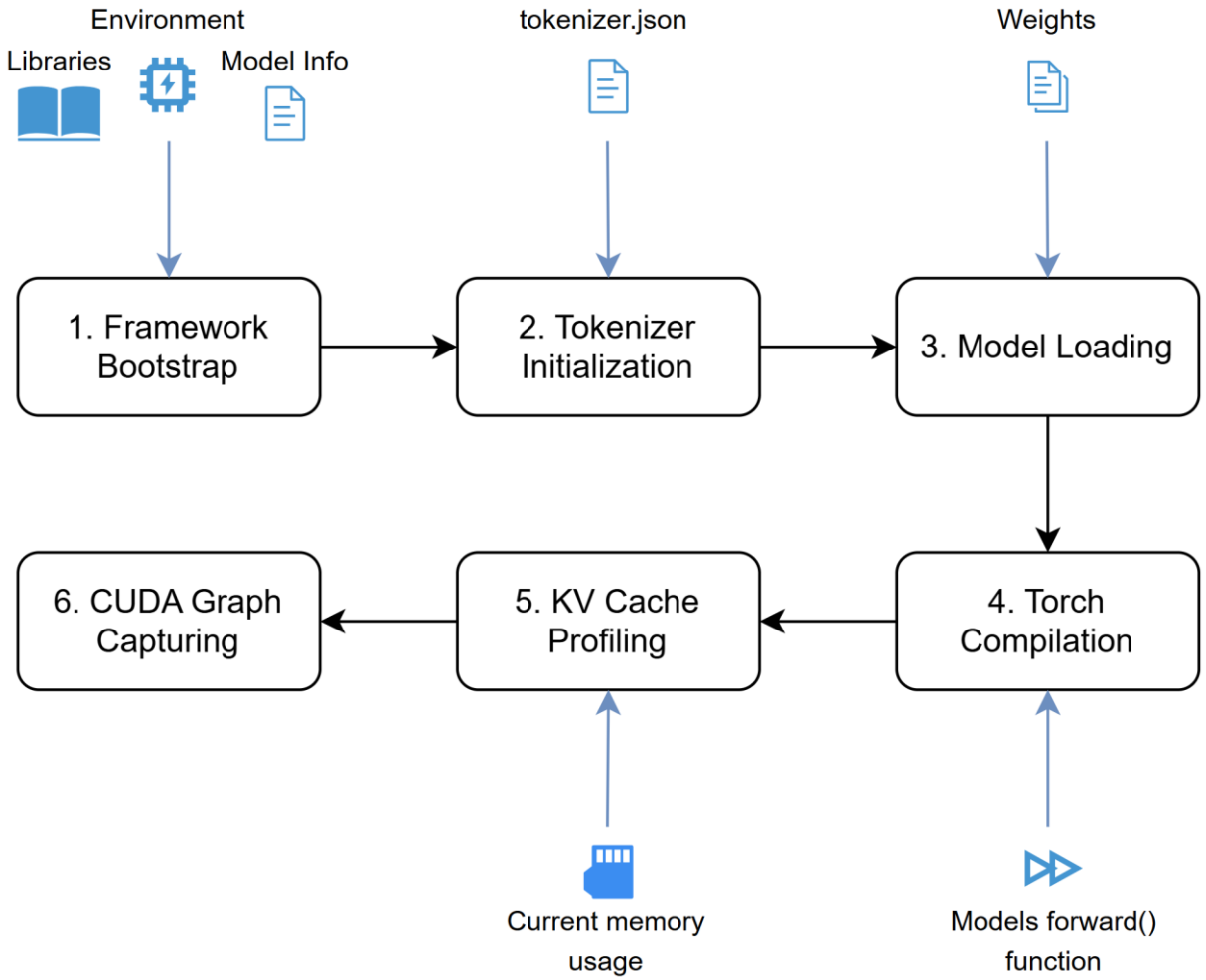


Relation to number of batch sizes

We have noticed that in the output logs, the graph is captured for each batch size
Therefore, longer time is needed for higher number of captured batch sizes



Startup Process Summary



Step	Parameter Dependency
1. Framework Bootstrap	Constant, regardless of model
2. Tokenizer Initialization	Tokenizer size (Vocab size)
3. Model Loading	Model Parameter Size
4. Torch Compilation	Number of layers, and complexity of model (Size of compiled graph)
5. KV Cache Profiling	Model Parameter Size
6. CUDA Graph Capturing	Model Parameter Size + Number of batch sizes

CPU- or GPU-bound

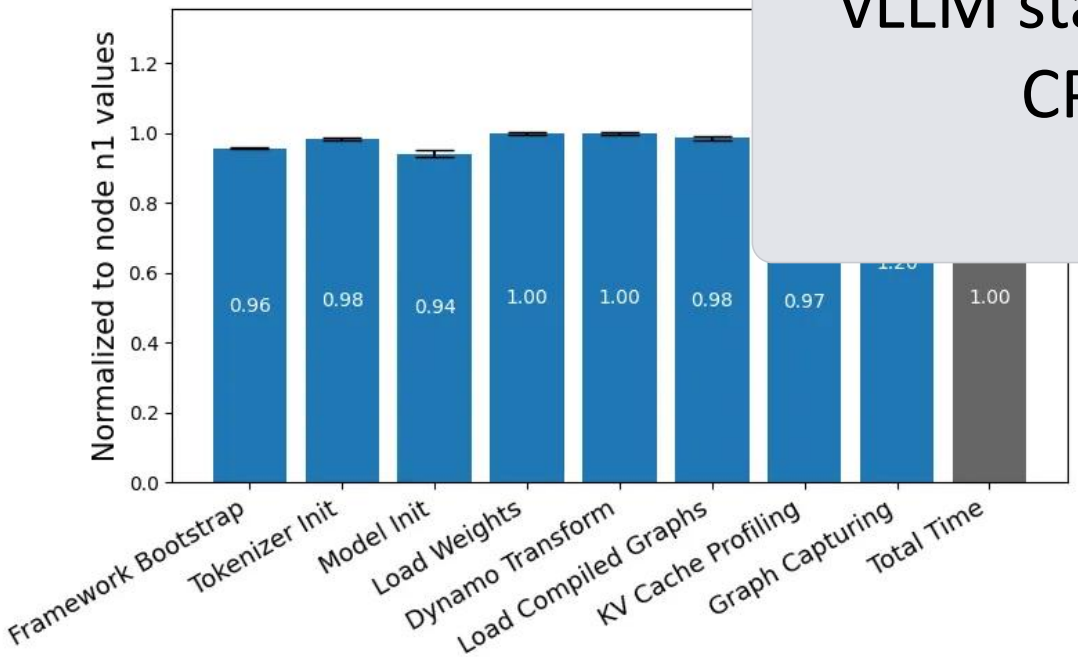
Different GPU

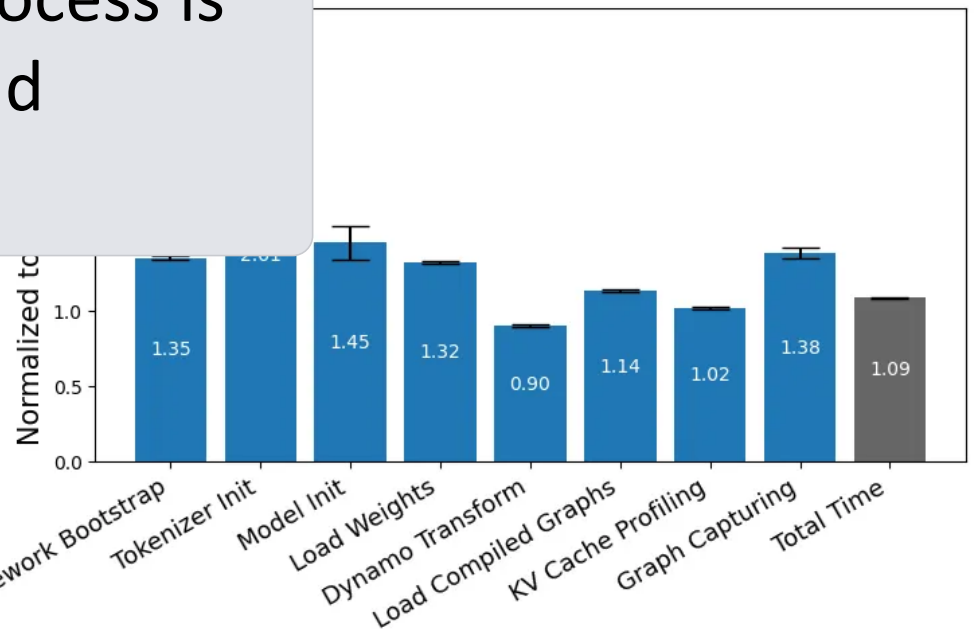
Compared experiments between H100 and L40S GPUs, across 10 models
All steps remained almost constant, e
Capturing!

Different CPU

Compared experiments between 2 CPUs, AMD EPYC 9354 and Intel Xeon Platinum 8568Y+ on same H100 GPU
s among all steps!

vLLM startup process is CPU-bound

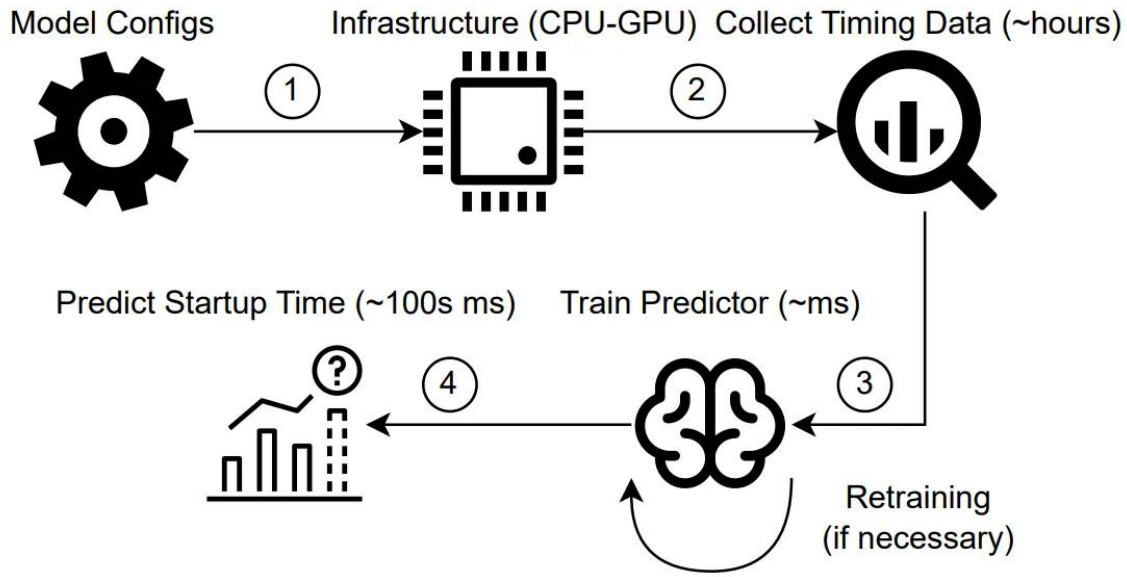




Analytical Predictor

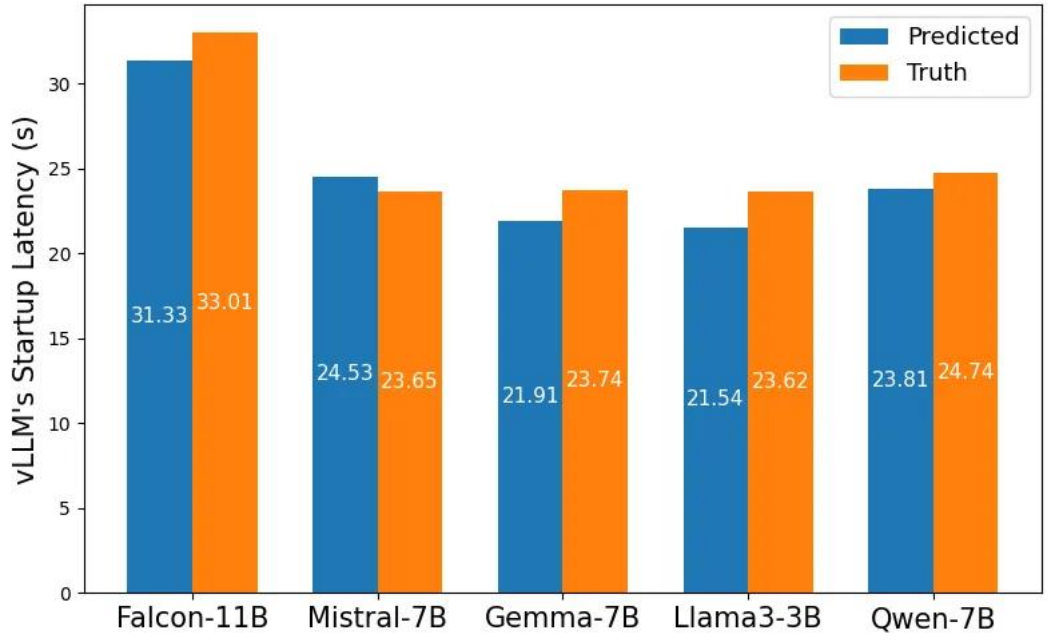
Process

We have decided to build a regression predictor for each step separately
This keeps the predictor simple, interpretable, and easy to adapt to future optimizations



Validation

Although the predictor is simple, it's able to accurately predict the startup latency of different models
We believe this could help to take better scheduling and autoscaling decisions



Main Takeaways

Longevity of Insights

Although vLLM is evolving rapidly, our analysis will still hold, as they are foundational to each step

Assumption of Linearity

Some steps (KV Cache Profiling) did not follow a clear linear trend for MoE models!

Measurement Noise

Background processes, scheduler contention may affect some results

Thank You

Any Questions?