vLLM

V1 Update & Q1 Roadmap

The vLLM Meetup at Google Cloud

The vLLM Team

Agenda

- V1 Deep Dive (Woosuk)
- vLLM’s Q1 Roadmap (Simon)

vLLM's Goal

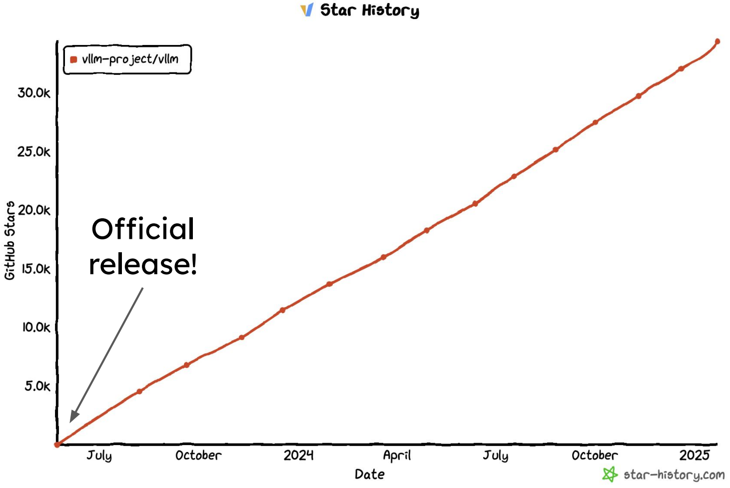
Build the fastest and easiest-to-use open-source LLM inference & serving engine

vLLM Today

https://github.com/vllm-project/vllm

`$ pip install vllm`

34.5K Stars



vLLM API (1): `LLM` class

A Python interface for offline batched inference

```python
from vllm import LLM

# Example prompts.
prompts = ["Hello, my name is", "The capital of France is"]
# Create an LLM with HF model name.
llm = LLM(model="meta-llama/Meta-Llama-3.1-8B")
# Generate texts from the prompts.
outputs = llm.generate(prompts) # also llm.chat(messages)]
```

vLLM API (2): OpenAI-compatible server

A FastAPI-based server for online serving

Server

```
$ vllm serve meta-llama/Meta-Llama-3.1-8B
```

Client

```
$ curl http://localhost:8000/v1/completions \
    -H "Content-Type: application/json" \
    -d '{
        "model": "meta-llama/Meta-Llama-3.1-8B",
        "prompt": "San Francisco is a",
        "max_tokens": 7,
        "temperature": 0
    }'
```

vLLM V1

Woosuk Kwon

What is vLLM V1?

Re-architect the "core" of vLLM

based on the lessons from V0 (current version)

Unchanged

- **User-level APIs**
- **Models**
- GPU Kernels
- Utility functions
- …

Changed

- Scheduler
- Memory Manager
- Model Runner
- API Server
- …

Why vLLM V1?

- Main goals:
    - Simple & easy-to-hack codebase
    - High performance with near-zero CPU overheads
    - Combining all key optimizations & enabling them by default

Key changes in vLLM V1

1. Optimized engine loop & API server

2. Simplified scheduler

3. Clean implementation of distributed inference

4. Piecewise CUDA graphs

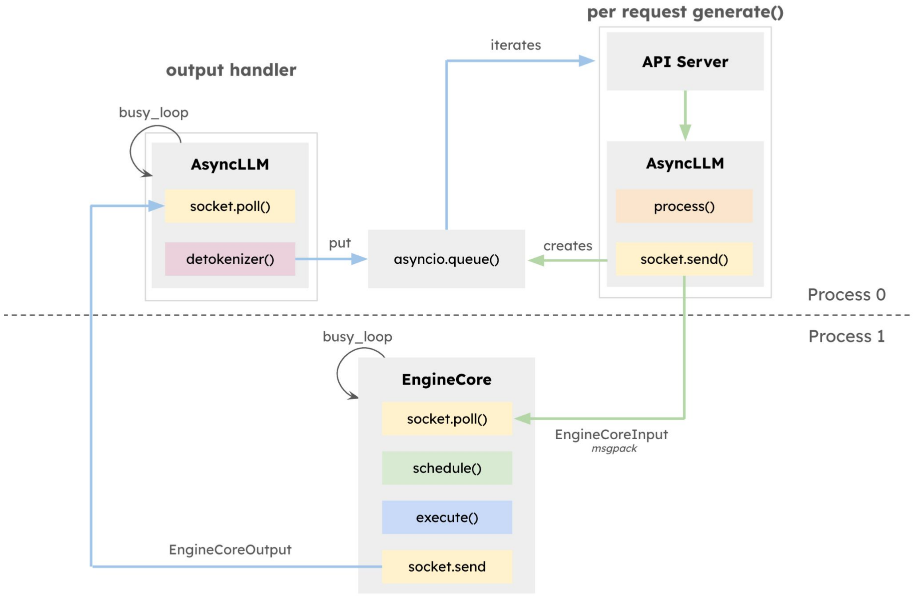
Optimized Engine Loop & API Server

Goal: Make sure GPU is not stalled

- By pre-processing
  - E.g., converting JPEG images into input tensors (resizing, cropping, …)
- By post-processing
  - E.g., de-tokenizing output token IDs into output strings
- By HTTP request handling
  - E.g., streaming outputs to 100s of concurrent users

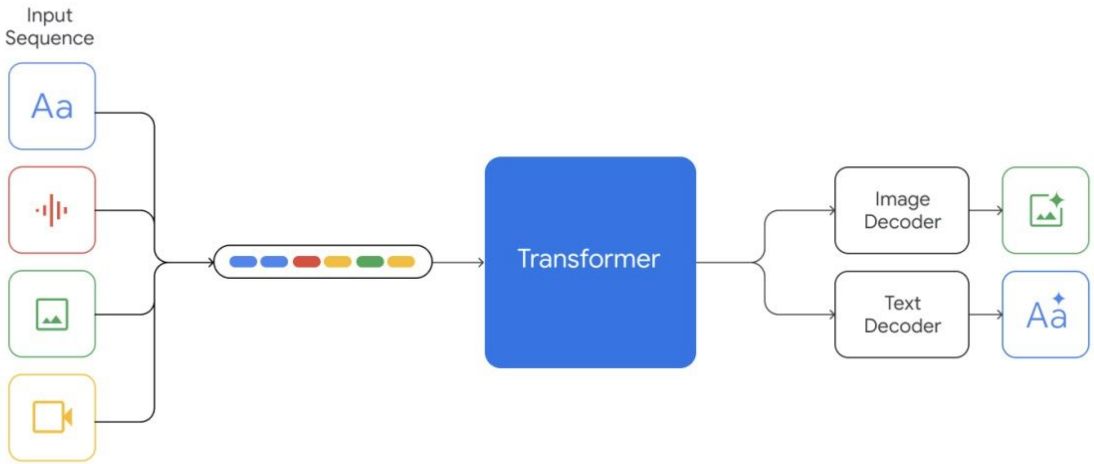
Optimized Engine Loop & API Server (cont'd)

- Two-process approach
  - Process 0 (Frontend): Pre-/post-processing & API Server
    - Importantly, de-tokenization happens in Process 0
  - Process 1 (EngineCore): Schedule & execute the model every step
    - A busy loop that is NOT blocked by Process 0



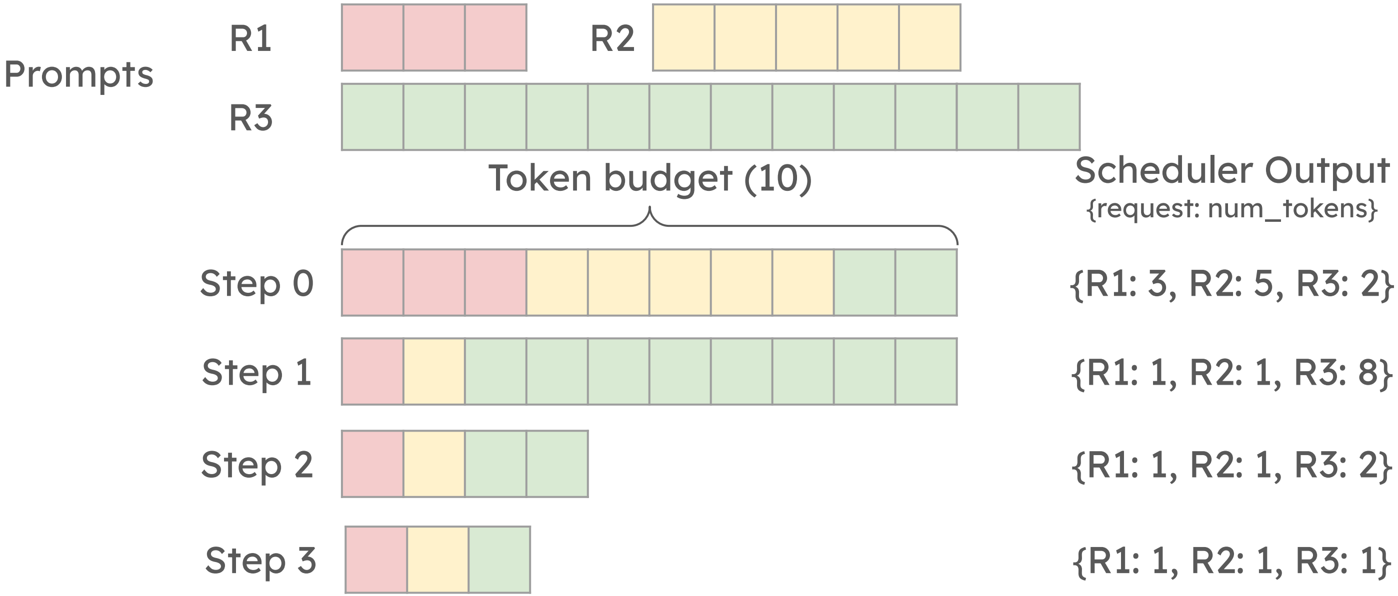
Optimized Engine Loop & API Server (cont'd 2)

- We are not there yet!
  - How to handle emerging modalities such as videos and audios?
  - It would be increasingly difficult to build a fast & reliable server
- We would like to solicit for more community engagement & contributions



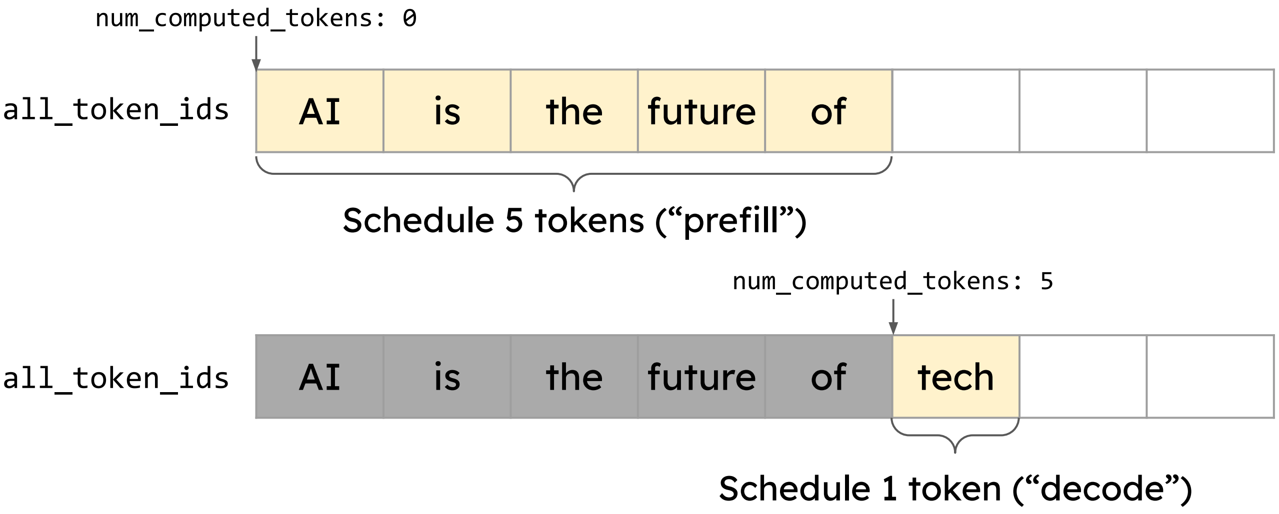
Simplified Scheduler

- Synchronous single-step scheduler
- Chunked prefills (aka Dynamic SplitFuse) by default
    - The scheduling decision is simply represented as a dictionary of `{request_id: num_tokens}`



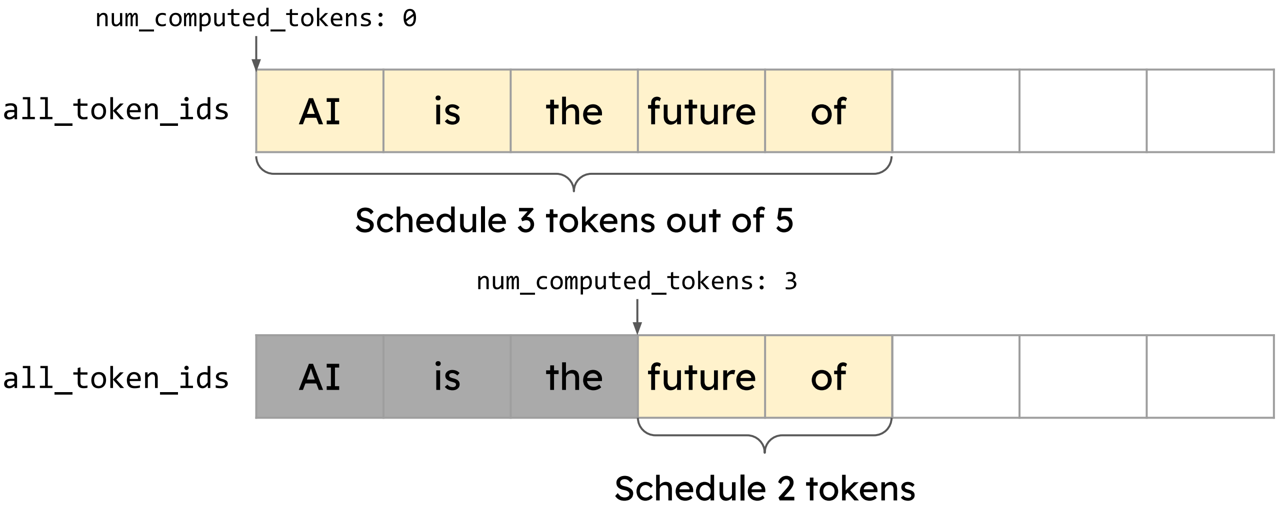
Simplified Scheduler (cont'd)

- Unification of "prefill" and "decode"
  - In V1, there's no concept of prefill and decode
  - Schedule based on the difference between `num_compute_tokens` and `len(all_token_ids)`
- Ex1) "Prefill" & "Decode"



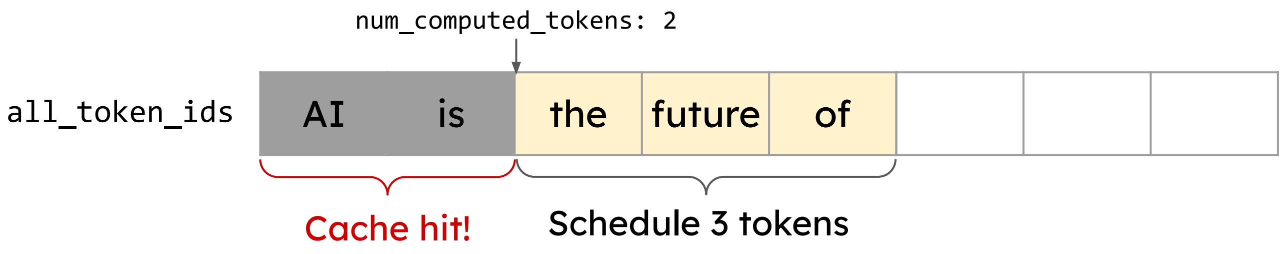
Simplified Scheduler (cont'd)

- Unification of "prefill" and "decode"
  - In V1, there's no concept of prefill and decode
  - Schedule based on the difference between `num_compute_tokens` and `len(all_token_ids)`
- Ex2) Chunked prefills



Simplified Scheduler (cont'd)

- Unification of "prefill" and "decode"
  - In V1, there's no concept of prefill and decode
  - Schedule based on the difference between `num_compute_tokens` and `len(all_token_ids)`
- Ex3) Prefix caching



Simplified Scheduler: Next Steps

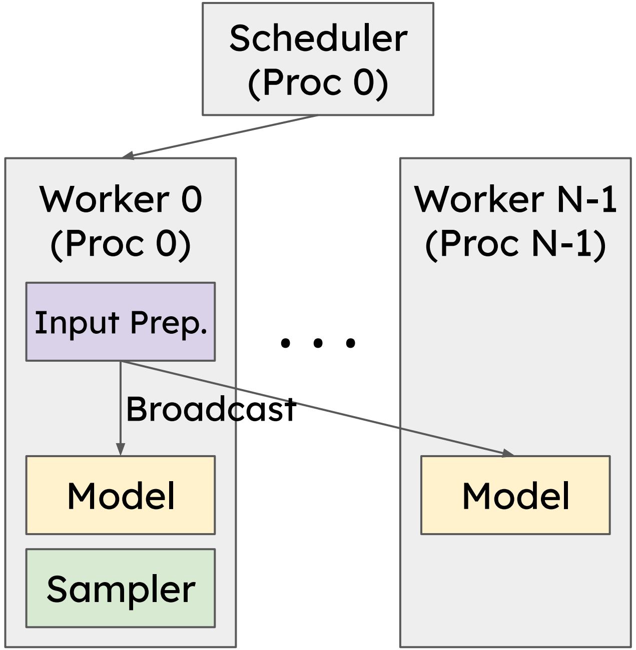
- Current: First-come-first-served policy is baked in the scheduler

- Next step 1: Support various scheduling policies
  - Priority-based scheduling
  - Fair scheduling
  - Predictive scheduling

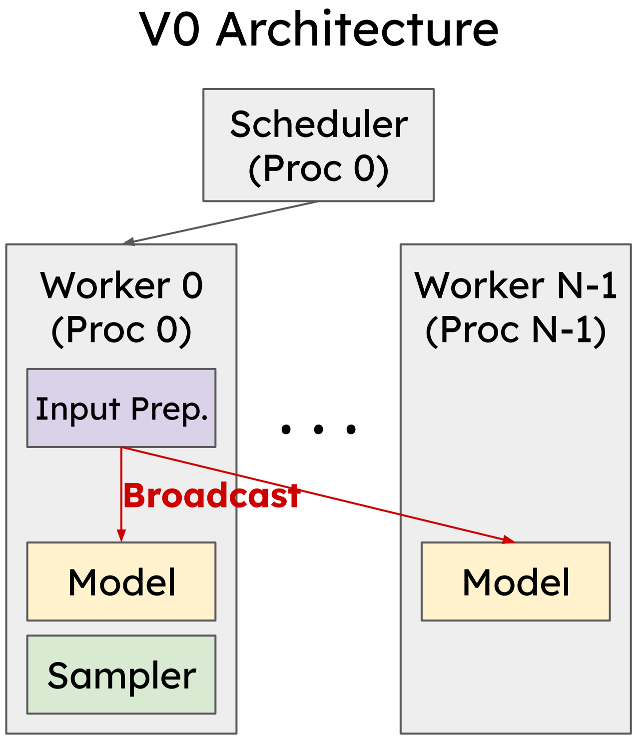
- Next step 2: Pluggable scheduler
  - E.g., workload-specific scheduler
  - E.g., different schedulers for different hardware backends

Clean Implementation of Distributed Inference

V0 Architecture

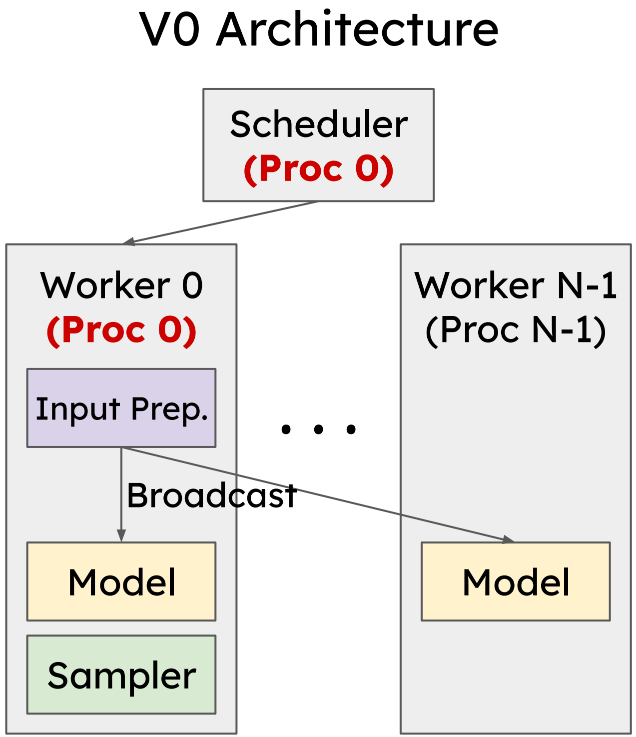


Clean Implementation of Distributed Inference



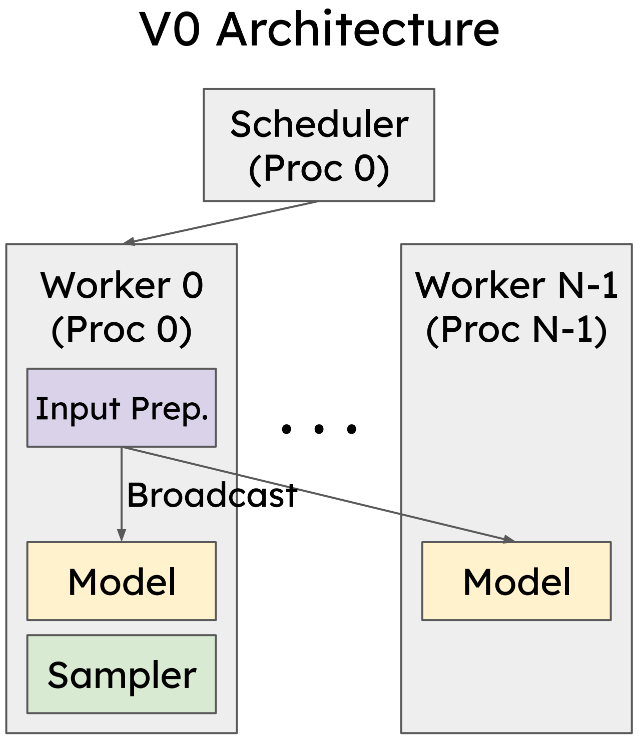
- Problem 1: Broadcasting whole input tensors every step
  - Significant communication overheads

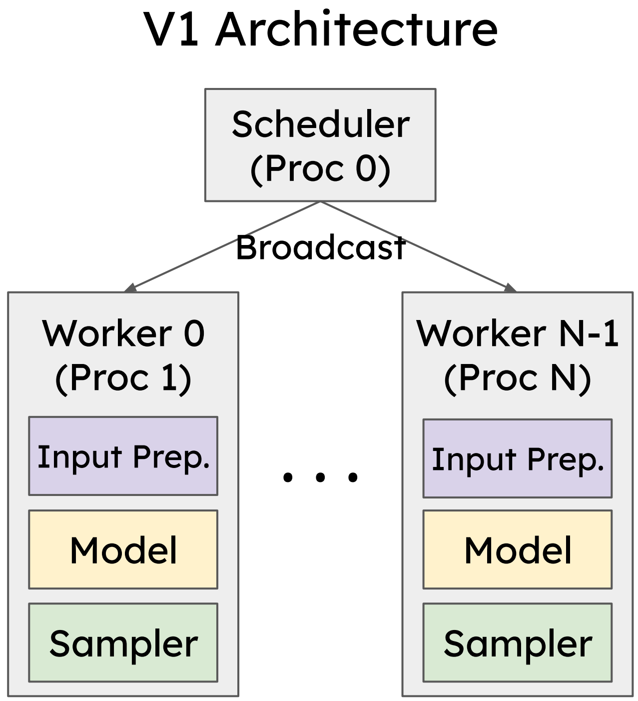
Clean Implementation of Distributed Inference



- Problem 1: Broadcasting whole input tensors every step
  - Significant communication overheads
- Problem 2: Scheduler and Worker 0 share the same process
  - Increased complexity due to the asymmetric architecture

Clean Implementation of Distributed Inference

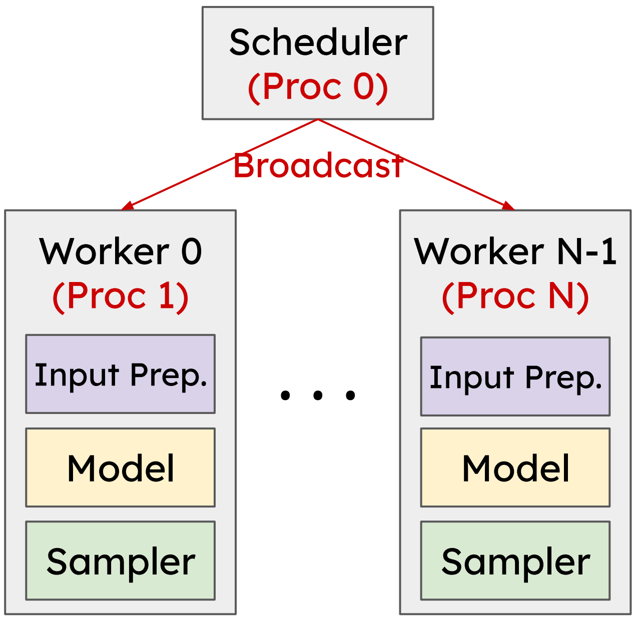




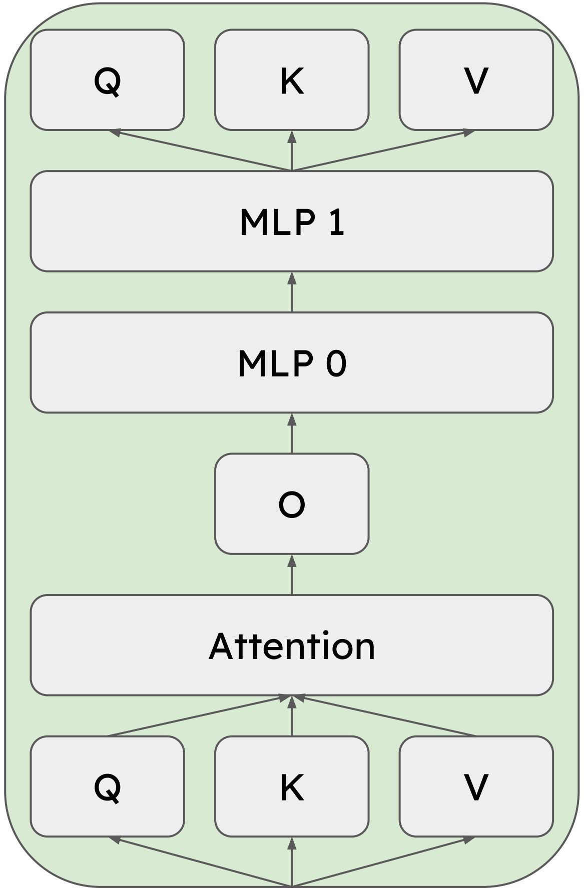
Clean Implementation of Distributed Inference

- Broadcasting the scheduler outputs (Python data structures)
  - Key optimization: Only sending the diff every step by caching the request states in the worker
- Symmetric architecture
  - Each worker has its own process
  - No code divergence between worker 0 and other workers
- Overall, most of the code for distributed inference is abstracted away

V1 Architecture



Piecewise CUDA Graphs



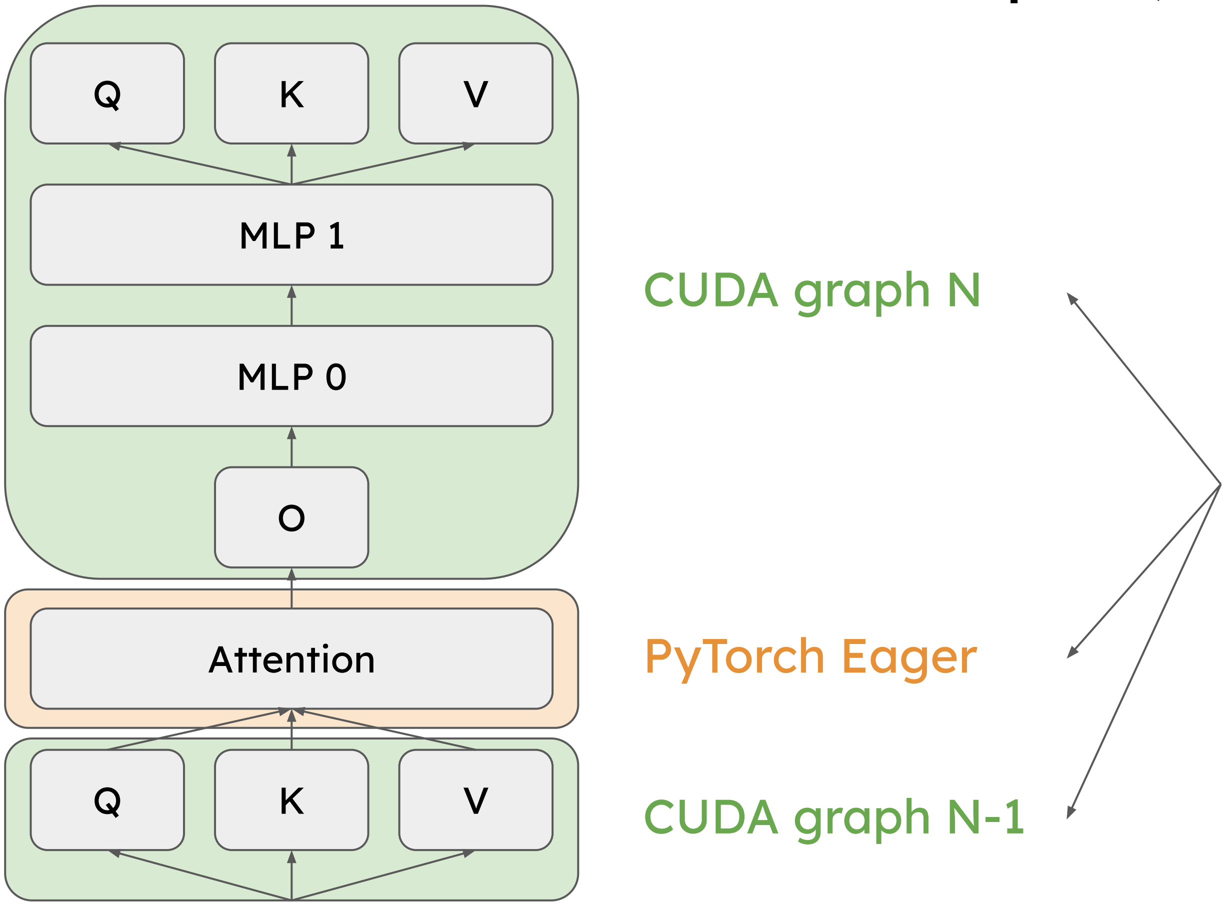
- V0: Single CUDA graph for the entire model

- Pros: Minimal CPU overheads in model execution

- Cons: Limited flexibility
  - Static shapes are required
  - No CPU operations are allowed

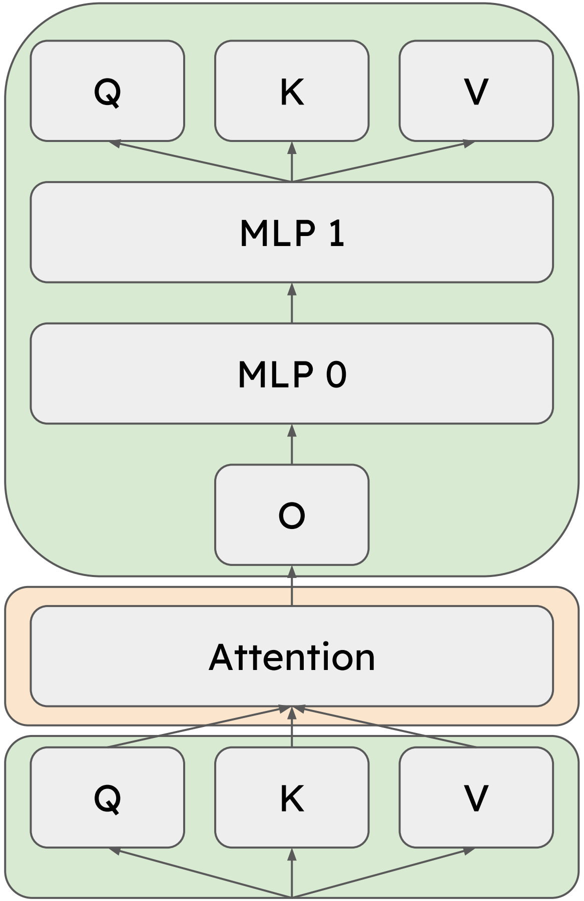
  → Increased development burden

Piecewise CUDA Graphs (cont'd)



Graph split using [torch.compile]()

Piecewise CUDA Graphs (cont'd)



- V1: Splits the model into pieces
  - Runs the attention op in eager-mode PyTorch
  - Runs other ops with CUDA graphs
    - Easy to capture, since the ops are token-wise
    - Critical to capture the all-reduce op

- Pros: Maximum freedom in implementing the attention op
  - No restriction on shapes
  - Any CPU operations are allowed

- Cons: CPU overheads unhidden by CUDA graphs could slow down the model execution
  - Negligible for 8B+ models on H100

Use Case: Cascade Attention for System Prompts

- What is system prompt?
  - A text prefix describing the task, style, tools, safety rules, etc.
  - Typically shared for all requests to the model
  - The length varies a lot by models & applications
    - E.g., Copilot: ~570 tokens, ChatGPT-4o: ~1200 tokens, Sonnet 3.5: ~2400 tokens

```
You are an AI programming assistant.
When asked for your name, you must respond with "GitHub Copilot".
Follow the user's requirements carefully & to the letter.
Follow Microsoft content policies.
Avoid content that violates copyrights.
...
```

Use Case: Cascade Attention for System Prompts

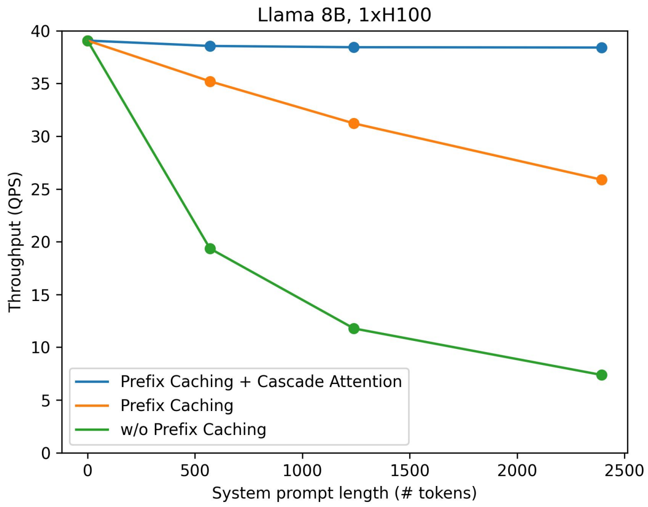
- What is system prompt?
  - A text prefix describing the task, style, tools, safety rules, etc.
  - Typically shared for all requests to the model
  - The length varies a lot by models & applications
    - E.g., Copilot: ~570 tokens, ChatGPT-4o: ~1200 tokens, Sonnet 3.5: ~2400 tokens

- What is cascade attention?
  - Proposed by Zihao Ye (UW) et al.
  - A GPU kernel trick to optimize attention with shared KV cache
  - Was difficult to integrate because the trick makes the kernel more dynamic, making it less compatible with CUDA graph

Use Case: Cascade Attention (cont'd)

- Thanks to piecewise CUDA graphs, vLLM V1 was able to smoothly integrate cascade attention and enable it by default
  - Automatically detects the system prompt & applies the optimization

- Performance improvements
  - ShareGPT + system prompts of varying lengths
  - 0–50% throughput increase
  - Makes the system prompt almost free



Potential Use Cases of Piecewise CUDA Graphs

Piecewise CUDA graphs will allow vLLM to easily integrate new optimizations such as:

- KV cache offloading to CPU memory (#11532)
- KV cache offloading to disk
- Sparse KV cache
- Brand-new attention algorithms
- . . .

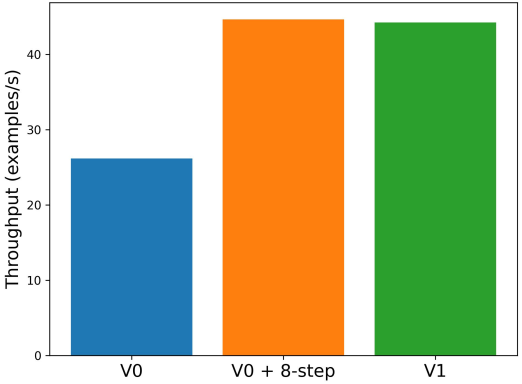
Other Important Enhancements in V1

- FlashAttention 3 integration
  - Also available in V0
- Prefix caching for image inputs
  - Useful for multi-turn chat with images
- Miscellaneous Python-level optimizations
  - Avoid frequently creating new objects
  - Minimize garbage-collection overhead
- . . .

V1 Engine Performance (Llama 3 8B)

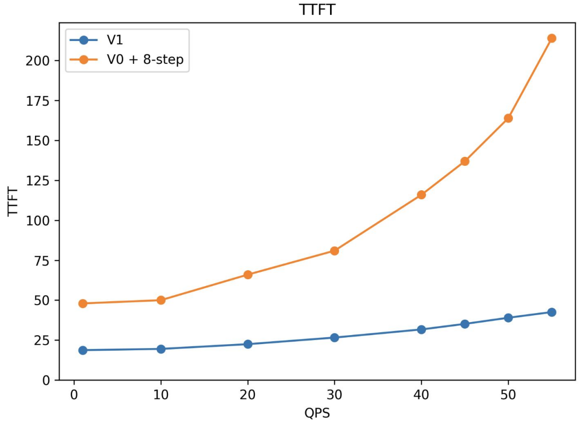
Offline Inference (ShareGPT)

Throughput

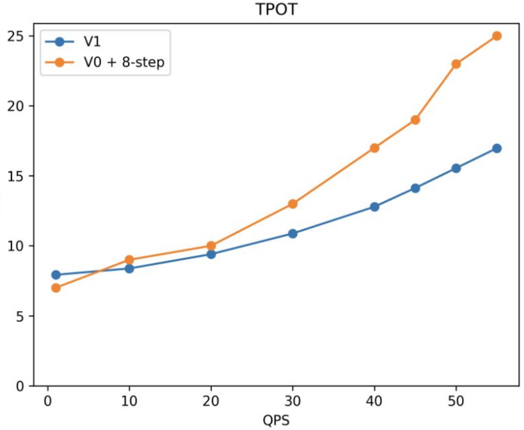


Online Serving (ShareGPT)

Time-to-first-token



Time-per-output-token



- Same throughput as V0 + 8-step scheduling
- Lower TTFT & TPOT than V0 + 8-step scheduling

V1 Current Status

- Will do **alpha** release very soon
  - The code can be found in [vllm/v1/]
- Set `VLLM_USE_V1=1` to use the V1 engine
  - Same end-user APIs as V0 (OpenAI server & `LLM` class)

- Supported models
  - Decoder-only Transformers (e.g., Llama, Mixtral)
  - Llava-style VLMs (e.g., Pixtral, Qwen2-VL)
- Features
  - Supported: chunked prefills, prefix caching, tensor parallelism
  - WIP: LoRA, spec decoding, pipeline parallelism, structured outputs
- Only supports NVIDIA GPUs for now
  - Actively working to also integrate other hardware backends

Q1 Roadmap

Simon Mo

vLLM 2025 Vision

1. Support **Emerging** Models
2. **Production** Deployments
3. **Open Architecture**

1. Support Emerging Models

...GPT4o level model in 1 GPU, and very large models

- **Small yet mighty models + large models coexist**
- **System Optimizations for:**
  - KV Cache
  - MoE
  - Long Context
- **Tailored Inference System for:**
  - Reasoning
  - Coding
  - Agents
  - Creative Writing
- **vLLM being used in data curation and RLHF**

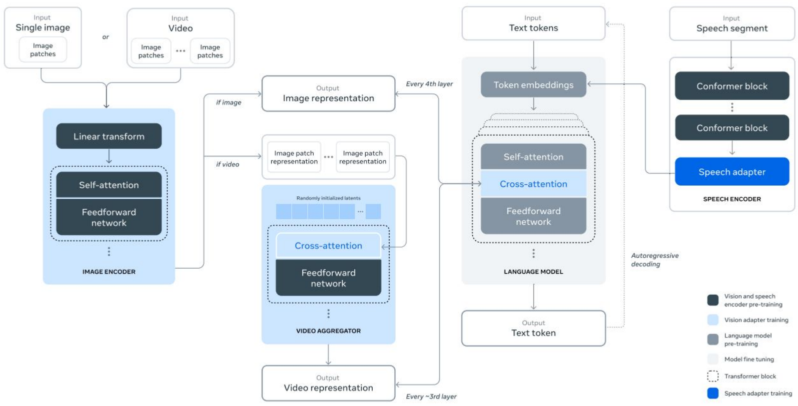
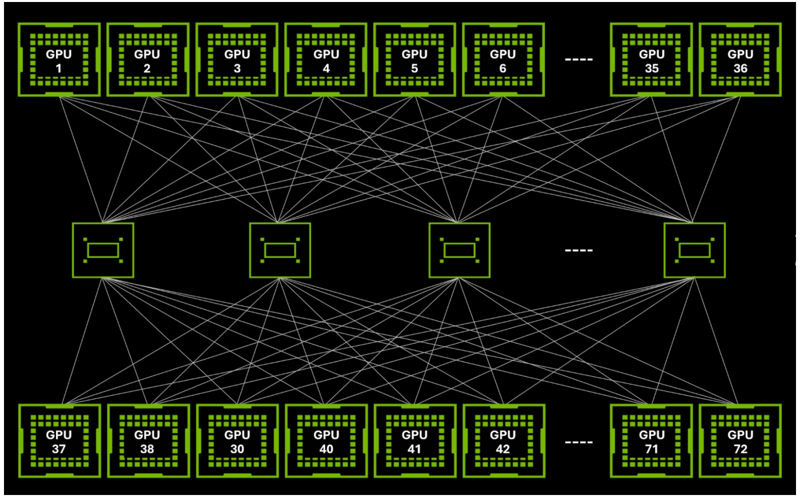


Illustration of adding multimodal capabilities to Llama 3

2. Production Deployment

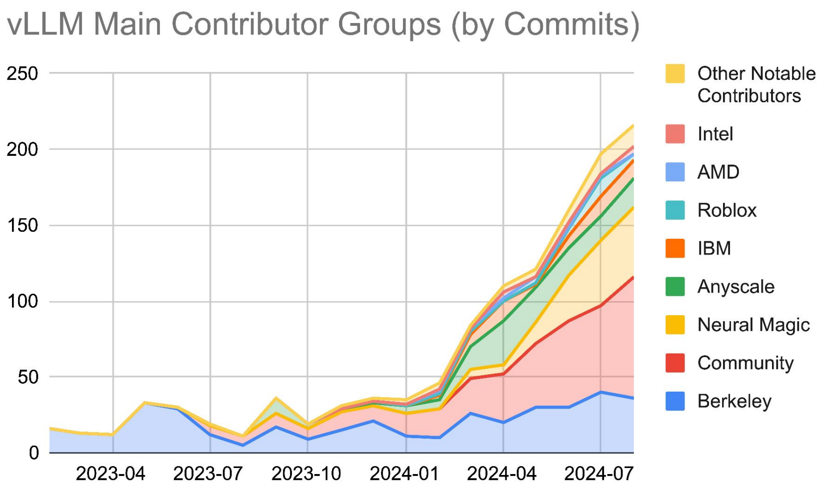
…hundreds of thousands of GPUs scale

- **The year of productionization of scaled deployments**
- **Recipes and ecosystems for routing, caching, and scaling**
- **High performance by default:**
  - Quantization (< FP8, KV Cache, Attention)
  - Speculative Decoding
  - Predicted and Structured Outputs
- **Cluster solutions increase in importance; support for prefill disaggregation**
- **vLLM being tuned for specific models, leading hardwares, and extreme workloads**



3. Open Architecture is Our Key to Adoption

- **A community of passionate users, adopters, and contributors**
- **vLLM can be easily extended and modified**
  - Forks and monetization are welcomed
- **Technically**
  - We will ship a performant yet modular V1 architecture.
  - torch.compile as an extension point for fusion
  - Researchers will build on it for new ideas
  - Users will extend it with private use cases
- **A coordinated engineering organization spanning multiple companies**
- **A positive ROI for everyone involved**



Our Q1 Roadmap

- roadmap.vllm.ai
- vLLM Core
- Emerging Features
- Ecosystem Projects

vLLM Core

- Ship a performant and modular V1 architecture
  - V1 on by default, spec decode, hybrid memory allocator and more!
- Support large and long context models
  - Sparsity in attention and MoEs
  - Disaggregated prefill support
- Improved performance in batch mode
  - RLHF
  - Long Generation

Built-in RLHF support

- If vLLM is an OS,
  - User requests are normal processes running on the OS
  - And the OS also provides "system call" to control the behavior of the system
- RLHF community has been a key driving-factor for adding new "system calls", and vLLM has been their primary inference engine
  - OpenRLHF, veRL, open_instruct, LLaMA-Factory , …
- vLLM provides built-in "system calls" that are difficult to implement out-of-tree
  - Sleep mode: unmap the kv cache and model weights, put vLLM in sleep mode. Give up GPU resource temporarily for training or other models.
- RLHF frameworks can also easily customize and add new "system calls"
  - Provide a new worker class, and use llm.collective_rpc to call the new functions.
  - For more details, please check the PR.

Features

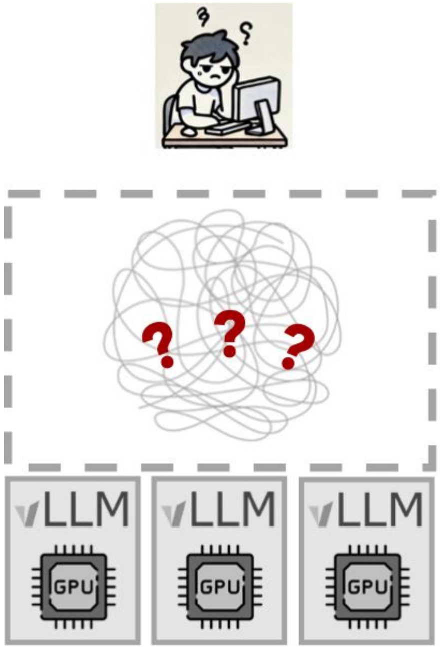
- Model Support
  - Arbitrary HF model
  - Alternative checkpoint format
- Hardware Support
  - Blackwell
  - Improved Tranium/Inferentia, Gaudi
  - Productionize and support large scale deployment of vLLM on TPU
  - Out of tree support for IBM Spyre, Ascend, Tenstorrent
- Optimizations
  - AsyncTP/Flux
  - FlashAttention
- Usability
  - Multi-platform wheels and distributions

Ecosystem Projects

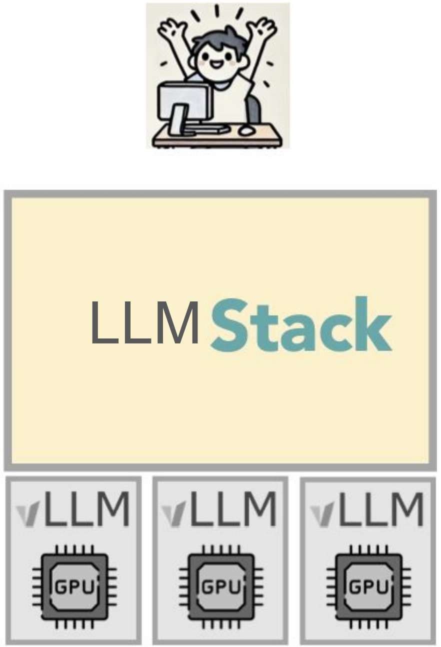
- Distributed batch inference
- Large scale serving
- Prefix aware router
- Multi-modality output

vLLM ecosystem project: LLM inference stack

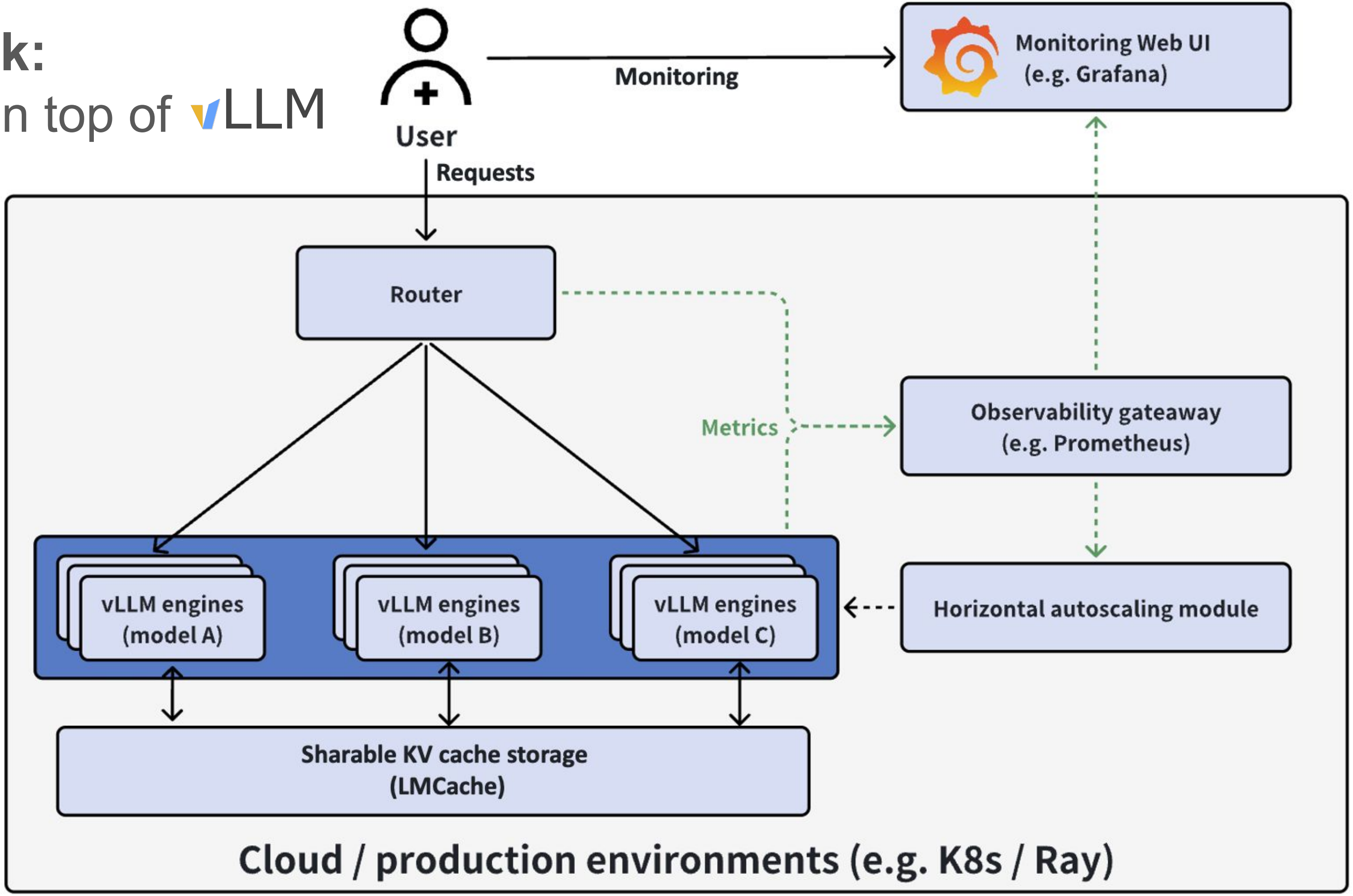
vLLM $\xrightarrow{???}$ LLM service



A stack on top of vLLM!



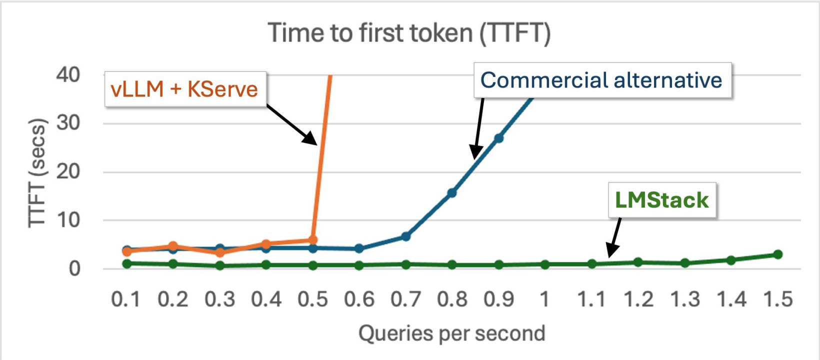
LLMStack:
A **Stack** on top of vLLM



Key features of **LLMStack**

- **One-click deployable** on kubernetes via `helm install`

- **Optimized performance**
  - **Smart router & KV cache offloading (LMCache)**
  - Handle **2x** more QPS with lower TTFT than commercial alternative!



- Cluster-wide **observability dashboard**

Thank you sponsors (funding compute!)

a16z
AMD
anyscale
aws
Crusoe
databricks
deepinfra
Dropbox
Google Cloud
Lambda
NEBIUS
novita.ai
NVIDIA
SEQUOIA
天工 Skywork
replicate
ROBLOX
RunPod
Trainy
THE·UNIVERSITY·OF·CALIFORNIA·BERKELEY·1868· LET THERE BE LIGHT
UNIVERSITY OF CALIFORNIA · SAN DIEGO · LET THERE BE LIGHT
ZhenFund 真格基金

Our Goal

Build the fastest and easiest-to-use open-source LLM inference & serving engine vLLM

Building the fastest and easiest-to-use open-source LLM inference & serving engine!

https://github.com/vllm-project/vllm

https://twitter.com/vllm_project

https://slack.vllm.ai

https://opencollective.com/vllm