



# Q1 Update

The vLLM Meetup at Beijing

The vLLM Team

# Agenda

- **vLLM V1** (Chen Zhang)
- **Q1 Roadmap Update** (Chen Zhang)
- **DeepSeek Enhancements** (Kaichao You)
- **Ecosystem Projects** (Kaichao You)

# VLLM's Goal

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

# vLLM Today



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

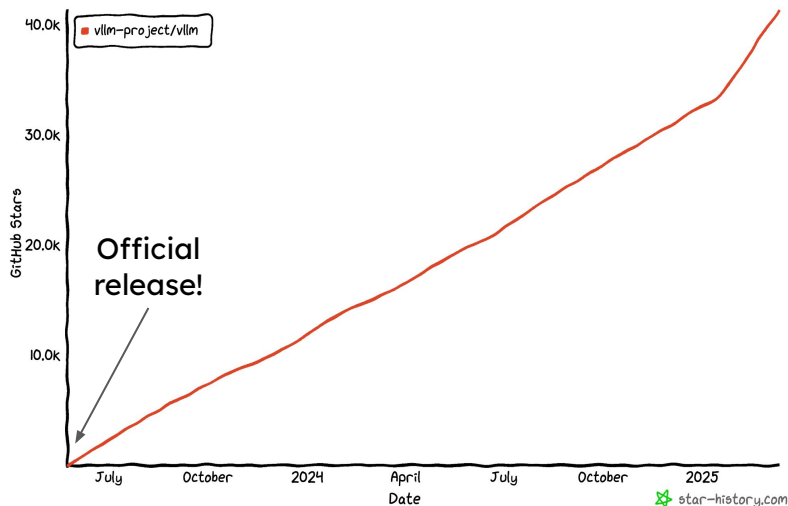


```
$ pip install vllm
```



**41.5K Stars**

v Star History



# vLLM API (1): LLM class

A Python interface for offline batched inference

```
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

[vLLM V1: A Major Upgrade to vLLM's Core Architecture](#)

# 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
    - `vllm/v1/core/scheduler.py` 608 LOC (>2k LOC in v0)
  - 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

<https://blog.vllm.ai/2025/01/27/v1-alpha-release.html>

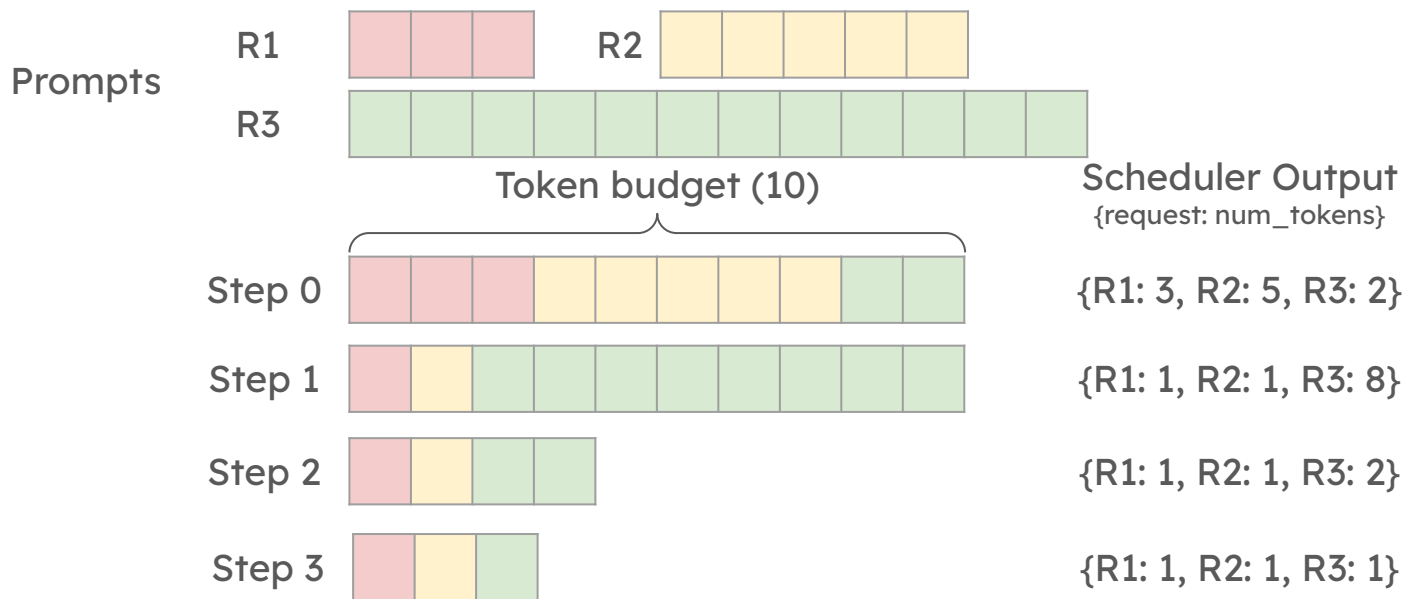
# Key changes in vLLM V1

1. Optimized engine loop & API server
2. Simplified scheduler
3. Clean implementation of distributed inference
4. Piecewise CUDA graphs

<https://blog.vllm.ai/2025/01/27/v1-alpha-release.html>

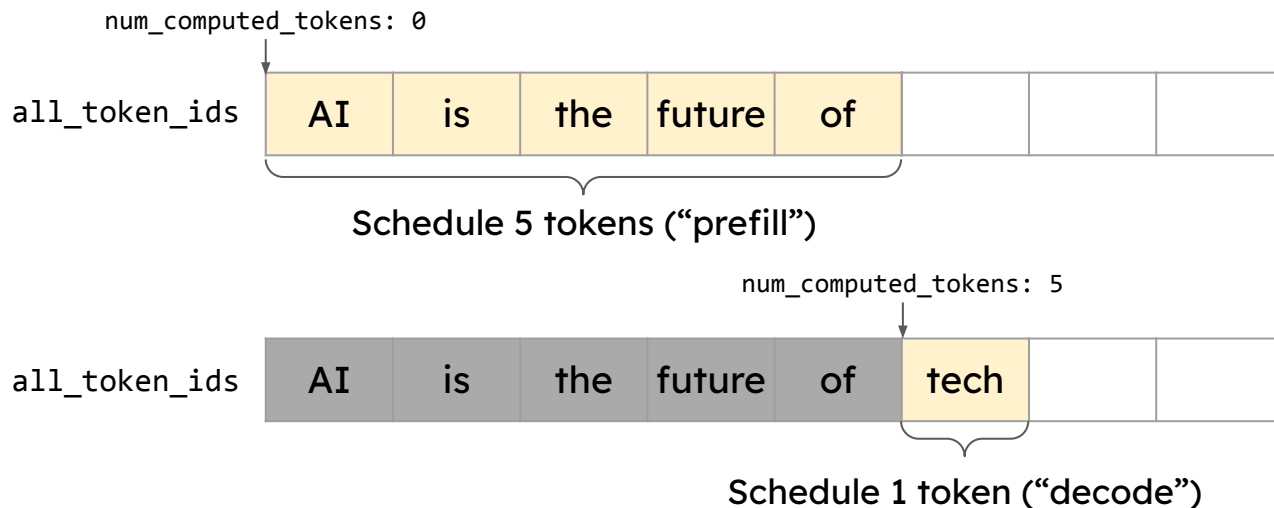
# Simplified Scheduler

- 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}`



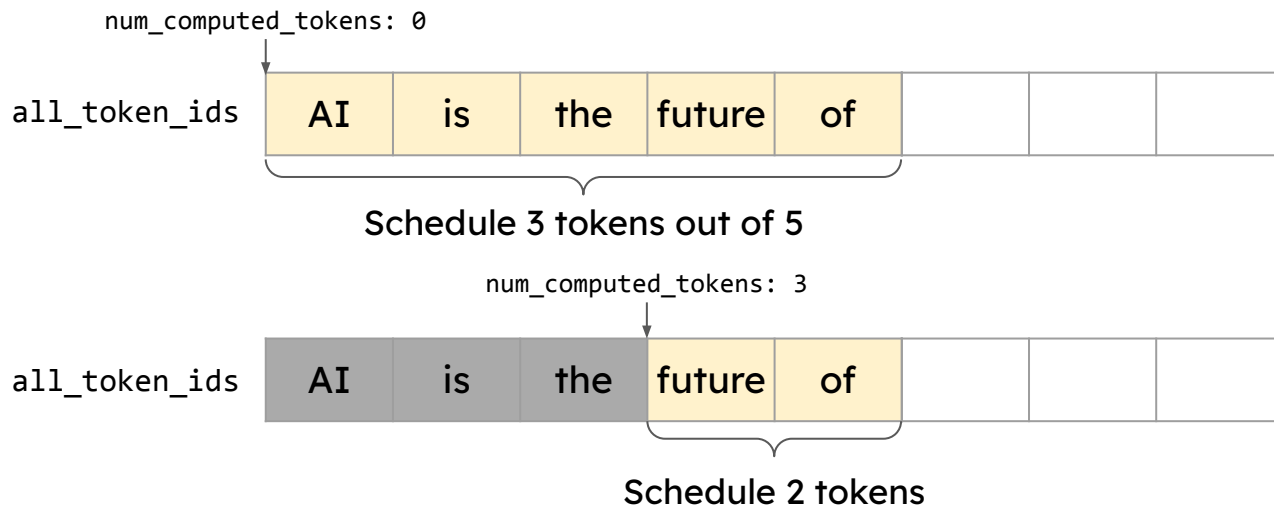
# Simplified Scheduler (cont'd)

- 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”



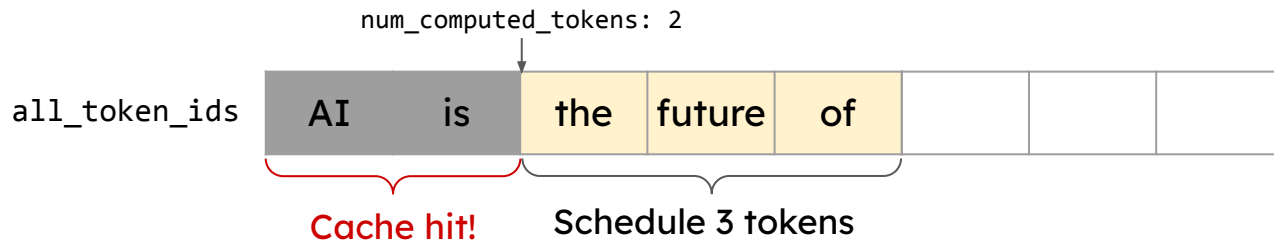
# Simplified Scheduler (cont'd)

- 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

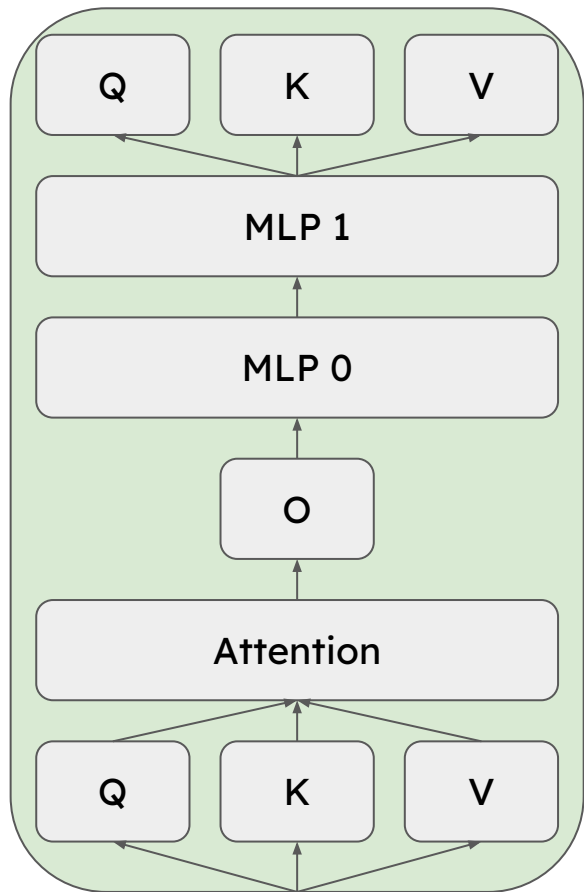


# Simplified Scheduler: Next Steps

- 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

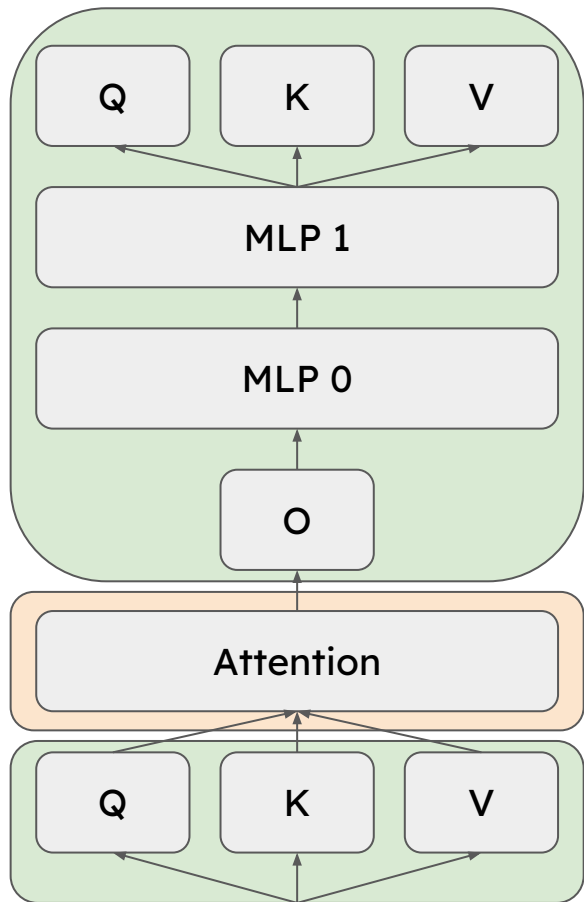


# Piecewise CUDA Graphs



- 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)



CUDA graph N

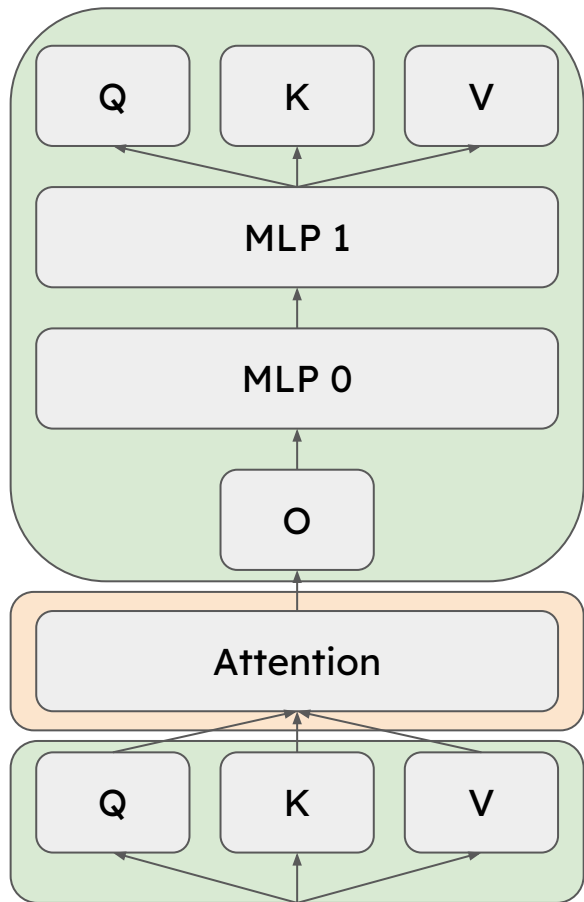
PyTorch Eager

CUDA graph N-1

Graph split using [torch.compile](https://pytorch.org/docs/stable/torch.compile.html)

- 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

## Piecewise CUDA Graphs (cont'd)



- 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**

# Potential Use Cases of Piecewise CUDA Graphs

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










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

# V1 Current Status

- Set `VLLM_USE_V1=1` to use the V1 engine
  - Same end-user APIs as V0 (OpenAI server & LLM class)
- **On by default** for supported use cases in the upcoming v0.8.0 release.
- **Supported models**
  - Decoder-only Transformers (e.g., Llama, Mixtral) & MOE
  - Llava-style LMMs (e.g., Qwen2.5-VL, Pixtral)
- **Features**
  - Supported: chunked prefills, prefix caching, tensor parallelism, LoRA, spec decoding (n-gram only for now), pipeline parallelism, structured outputs
- **Only supports NVIDIA GPUs for now**
  - AMD, TPU, HPU work in progress



# Q1 Roadmap

# vLLM Core





-  Ship a performant and modular V1 architecture
  -  V1 on by default
  -  Spec decode
  -  Hybrid memory allocator
  -  Documentation and Design Docs
- Support large and long context models
  -  Attention DP and EP
  -  Disaggregated prefill support
- Improved performance in batch mode
  -  RLHF
  -  Long Generation

# 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 and Ascend

- **Optimizations**

-  AsyncTP/Flux
-  FlashAttention3

- **Usability**

-  Multi-platform wheels and distributions



# DeepSeek Support



🔗 **deepseek-ai/DeepSeek-R1**

📄 Text Generation • Updated 19 days ago • 📄 2.29M • ⚡ • ❤️ 11.3k

🔗 **deepseek-ai/DeepSeek-R1-Zero**

📄 Text Generation • Updated 19 days ago • 📄 9.91k • ❤️ 864

🔗 **deepseek-ai/DeepSeek-R1-Distill-Llama-70B**

📄 Text Generation • Updated 19 days ago • 📄 358k • ⚡ • ❤️ 630

🔗 **deepseek-ai/DeepSeek-R1-Distill-Qwen-32B**

📄 Text Generation • Updated 19 days ago • 📄 1.59M • ⚡ • ❤️ 1.26k

🔗 **deepseek-ai/DeepSeek-R1-Distill-Qwen-14B**

📄 Text Generation • Updated 19 days ago • 📄 693k • ⚡ • ❤️ 467

🔗 **deepseek-ai/DeepSeek-R1-Distill-Llama-8B**

📄 Text Generation • Updated 19 days ago • 📄 1.53M • ⚡ • ❤️ 642

🔗 **deepseek-ai/DeepSeek-R1-Distill-Qwen-7B**

📄 Text Generation • Updated 19 days ago • 📄 1.25M • ❤️ 545










🔗 **deepseek-ai/DeepSeek-R1-Distill-Qwen-1.5B**

📄 Text Generation • Updated 19 days ago • 📄 1.62M • ⚡ • ❤️ 1.04k

- Qwen ✓
- Llama ✓

# Status of DeepSeek Model Support

## 4 Major Performance Levers:

1. MLA: Multi-head Latent Attention ( v0,  v1)
  - a. FlashMLA 
2. MTP: Multi-Token Prediction ( v0,  v1)
3. EP: Expert Parallelism ( v0,  v1)
4. DP: Attention Data Parallelism ( v0,  v1)

## vLLM's unique features:

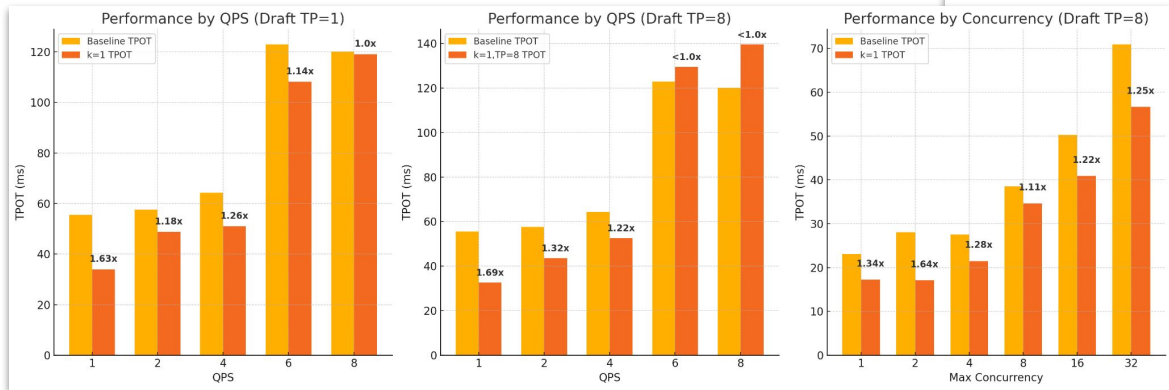
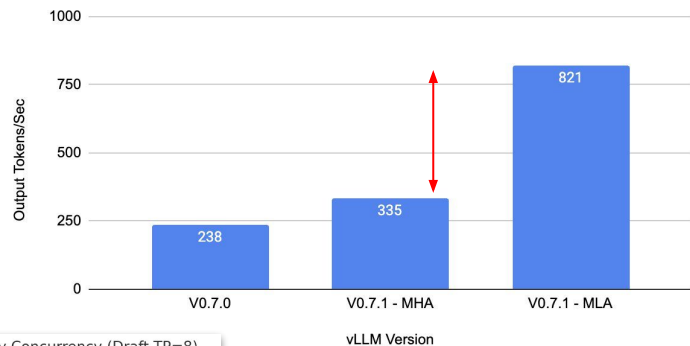
- Pipeline Parallelism
- Integration with SOTA kernels (FlashMLA, FlashInfer, FlashAttention)
- More spec decode methods (ngrams, draft based, etc)
- Ecosystem of RLHF integrations, offline inference & serving infra

# V0 Results of MLA and MTP

Multi-head Latent Attention: 2.4x  
decoding throughput against MHA

DeepSeek-R1 vLLM Throughput

NVIDIA 8xH200 | 1000 Input Tokens | 3000 Output Tokens



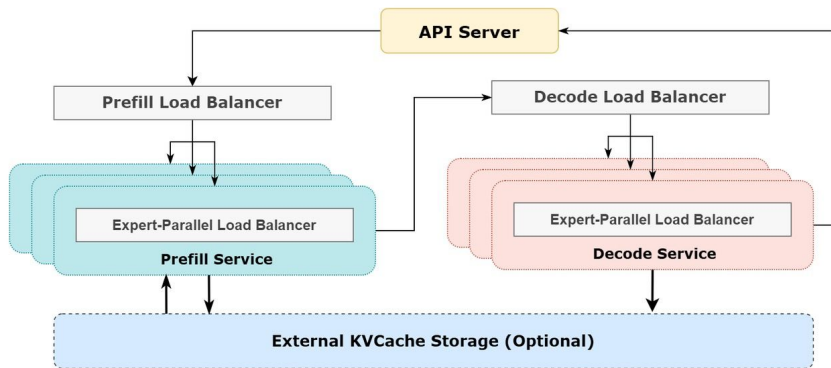
Multi-token prediction (k=1):  
Improve TPOT with low QPS

# Next Step for DeepSeek Models

## Efficient Serving of MoE models

1. DeepEP integration
2. Expert Load Balance
3. PD disaggregation

## Efficient support without loss of generality



Contribution and  
collaboration is welcome!

# Ecosystem Projects



 **vllm** A high-throughput and memory-efficient inference and serving engine for LLMs

---

 **vllm-spyre** Community maintained hardware plugin for vLLM on Spyre

---

 **llm-compressor** Transformers-compatible library for applying various compression algorithms to LLMs for optimized deployment with vLLM

---

 **aibrix** Cost-efficient and pluggable Infrastructure components for GenAI inference

---

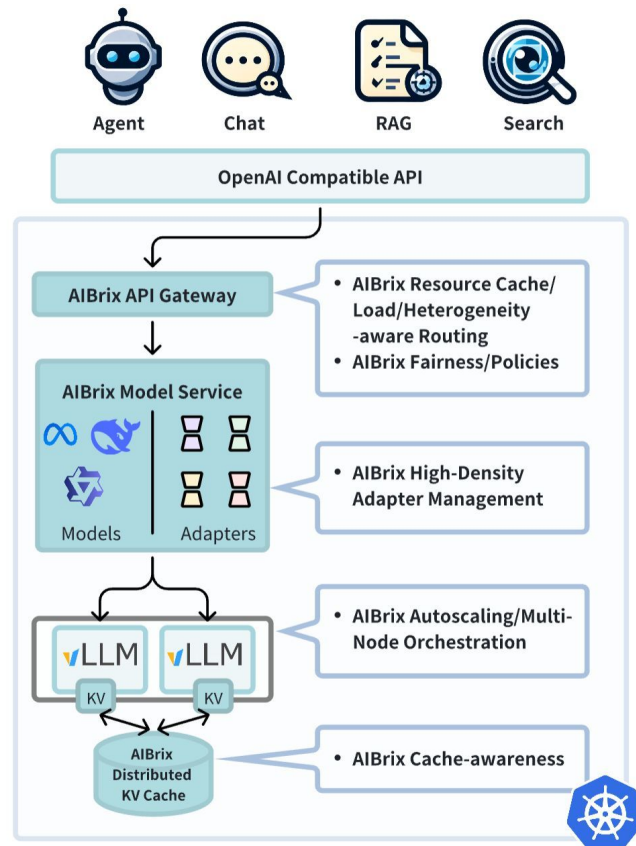
 **production-stack** vLLM's reference system for K8S-native cluster-wide deployment with community-driven performance optimization

---

 **vllm-ascend** Community maintained hardware plugin for vLLM on Ascend

# AI Brix

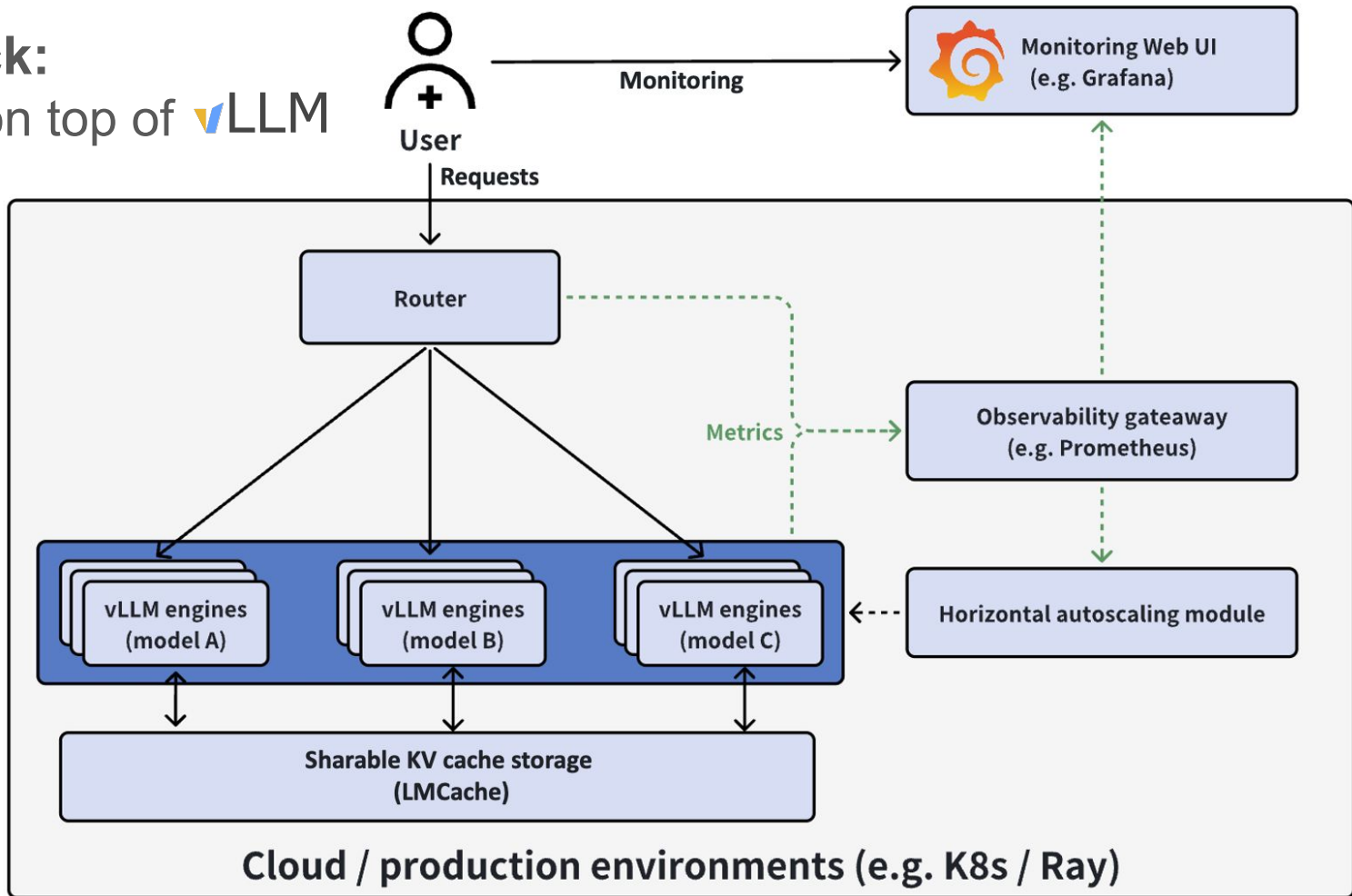
- High-Density LoRA Management
- LLM Gateway and Routing
- LLM App-Tailored Autoscaler
- Unified AI Runtime
- Distributed Inference
- Distributed KV Cache
- Cost-efficient Heterogeneous Serving
- GPU Hardware Failure Detection





# LLMStack:

A **Stack** on top of vLLM



# Scale out LLM inference with Ray

- Pythonic and extendible **ray.serve** application for scaling LLM engines for online inference

## Features

- ⚡ Automatic scaling and load balancing
- 🌐 Unified multi-node multi-model deployment
- 🔌 OpenAI compatible
- 🔄 Multi-LoRA support with shared base models
- ☁ Cloud storage as model source (AWS, GCP)

## Try out today

<https://docs.ray.io/en/master/serve/llm/overview.html>



```
from ray import serve
from ray.serve.llm.configs import LLMConfig
from ray.serve.llm.deployments import VLLMService, LLMRouter

llm_config = LLMConfig(
    model_loading_config=dict(
        model_id="Qwen/Qwen2.5-0.5B-Instruct",
    ),
    deployment_config=dict(
        autoscaling_config=dict(
            min_replicas=1, max_replicas=2,
        )
    ),
    # Pass the desired accelerator type (e.g. A10G, L4, etc.)
    accelerator_type="A10G",
    # You can customize the engine arguments (e.g. vLLM engine kwargs)
    engine_kwargs=dict(
        tensor_parallel_size=2,
    ),
)

# Deploy the application
deployment = VLLMService.as_deployment().bind(llm_config)
llm_app = LLMRouter.as_deployment().bind([deployment])
serve.run(llm_app)
```

# Partner Projects



PyTorch

...

Thank you sponsors (funding compute!)



Anyscale



Crusoe



databricks



deepinfra



Dropbox



Google Cloud



Lambda



NEBIUS



novita.ai



NVIDIA

SEQUOIA



Replicate

ROBLOX



RunPod



Trainy



ZhenFund  
真格基金



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



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



<https://slack.vllm.ai>



<https://www.linkedin.com/company/vllm-project>



[https://twitter.com/vllm\\_project](https://twitter.com/vllm_project)



<https://opencollective.com/vllm>



<https://www.zhihu.com/people/vllm-team>



Search “vLLM” or scan this QR code



More events in China in the future!