SATURDAY 24 OCTOBER 2026 · BANGKOK

Ray Summit 2026 Recap

Opening & What's Ray and vLLM

+ Serving a Frontier Open Model with Ray & vLLM

by vLLM Bangkok x AGICAFET x Anyscale

Saturday 24 October 2026 · 09:00–12:00

ASIC Room, 31st floor, MS Siam Tower, Bangkok

Natdhanai Praneenatthavee (Job)

CEO & CAIO, AGICAFET · President & Co-Founder, vLLM Bangkok



HOSTED BY

vLLM BANGKOK COMMUNITY | AGICAFET | Anyscale

Venue sponsored by SVOA

SVOA EXPERT BEYOND IT

OPENING

Why a Thai recap of Ray Summit

> **The vLLM Bangkok team fosters an ecosystem for production-grade AI infrastructure.**

VLLM BANGKOK · VLLMBANGKOK.COM

1 It happened in San Francisco

24–26 August 2026, 2,000+ people. Most of us could not be there.

2 Retold in Thai, in one morning

Six talks by people who run this stack in production here, three hours, one room.

3 Connected to Thai teams

What the Ray and vLLM news means for teams serving models in Thailand.

HOUSEKEEPING

Community T-shirt: pick up at registration

Break 10:10 · networking from 11:20

Photos today — tell the desk to opt out

OPENING

vLLM Bangkok so far: three events in 2026

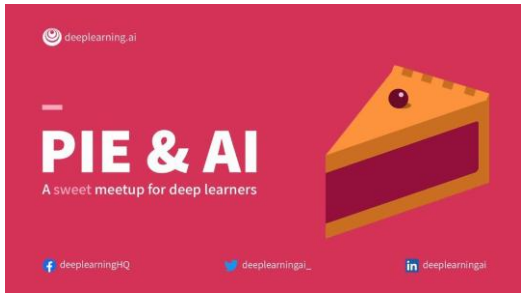


28 APR SESSION

IBM TechXchange Thailand x AI Alliance

IBM Thailand, SP Building

Our first stage: Embedded LLM gave the vLLM session in a half-day on multi-agent orchestration.



27 JUN MEETUP

Pie & AI: Bangkok — Embedded AI

Cleverse Rama 9

A DeepLearning.AI community meetup hosted by AGICAFET; five talks, slides online.



6 SEP FLAGSHIP

vLLM Bangkok Day

J.AI Arena, JAS Green Village

The flagship: 250 attendees, 11 sessions, speakers from Embedded LLM, Inferact, Red Hat, Tensormesh and more.



24 OCT TODAY

Ray Summit 2026 Recap

ASIC Room, MS Siam Tower

Today — six talks retelling Ray Summit, with Anyscale as partner and SVOA hosting us.

Slides, recordings and photos from every event: vllmbangkok.com/en/events · the Day's recap: vllmbangkok.com/en/recap

Source: vllmbangkok.com (events, recap)

vLLM Bangkok Day, 6 September 2026









J.AI ARENA · JAS GREEN VILLAGE RAMKHAMHAENG · 14:00–19:00

250 attendees

11 sessions

7 + 2 talks + panels

Introduction to vLLM	Tun Jian Tan · Embedded LLM
Inside vLLM: best practices & roadmap	Tiezhen Wang · Inferact
Scaling inference with llm-d	Jing Wen Ng · Red Hat
Supercharging vLLM with LMCache	Shaw Nguyen · Tensormesh
Thai and global perspective	CK Cheong · Fastwork
Panel · AI adoption in Thailand	KBTG · Embedded LLM · Tensormesh · moderated by FlowAccount
Panel · vLLM developers in Thailand	True Digital · Tsinghua · NVIDIA · moderated by Kanin Kearpimy

Partners: Embedded LLM · AGICAFET · Jaymart Group · SVOA · MSI · Red Hat · Inferact · Tensormesh · Thammasat · FlowAccount · ooca · NATACHAT · Recorded by @creatorsgarten · youtube.com/watch?v=fZ-VKvwM8xE

Source: vllmbangkok.com/en/recap · photos by the vLLM Bangkok team

Thank you to our venue sponsor



SVOA — venue sponsor

ASIC Room, 31st floor, MS Siam Tower

Hosts: vLLM Bangkok · AGICAFET · Partner: Anyscale

OPENING

This morning, 08:30–12:00

08:30		Registration · coffee	
09:00	01	**Opening & What's Ray and vLLM + Serving a Frontier Open Model with Ray & vLLM**	 **Natdhanai Praneenatthavee (Job)** AGICAFET · vLLM Bangkok
09:30	02	**State of vLLM 2026**	 **Paphanin Amnuaikitwanit** AI Engineer, vLLM Bangkok
09:50	03	**Beyond text-to-video: Omni Context Generation in a 33B Model**	 **Sattaya Singkul** True Digital Group · vLLM Bangkok
10:10		Break	
10:20	04	**Accelerating Large Language Models: vLLM on TPUs**	 **Kanin Kearpimy** PSF Fellow · vLLM Bangkok
10:40	05	**AMD and vLLM: What's New**	 **Passin Pornvoraphat** Chulalongkorn University
11:00	06	**NVIDIA and vLLM Full-Stack Collaboration for DeepSeek and MiniMax Performance**	 **Arisa Umparat** AI Engineer & NLP Researcher
11:20		Networking until 12:00	

Programme: vllmbangkok.com/en/events/ray-summit-recap-2026-10

Ray Summit 2026 in six numbers

24–26 August 2026 · San Francisco Marriott Marquis · hosted by Anyscale

2,000+

attendees

keynotes, breakouts and a hands-on training day

~3,200 GPUs

spun up for hands-on training

plus 36,000 CPUs across 830 workspaces over the two days, rebuilt over lunch

800+

people in training

up from about 515 in 2025; one session passed 400

4 topics

“four frontiers, one event”

Foundation Model Training · Multimodal Data Curation · Physical AI · LLM RL

1st

vLLM Conference, co-located

hosted by Inferact, 25–26 August; one pass covered both

3,000+

vLLM contributors

1,000+ hardware configurations, 5,000+ models (Simon Mo)

“Reinforcement learning has graduated from a research topic into a production engineering problem.” — the summit’s theme, in Anyscale’s words

Source: Anyscale, Ray Summit 2026 recap (anyscale.com/blog/ray-summit-2026-recap)

Keynotes: RL became a production problem

Six stories from the main stage — every one of them trains or runs models with RL on Ray

DAY 1 · ANYSCALE

RL → production

Robert Nishihara opens

The new workloads: RL loops that never stop, agents acting in the world, and physical simulation at scale.

DAY 1 · LILA SCIENCES

719 alloys in 5 weeks

Andrew Beam, CTO

Robots run the lab experiments; RL on Ray decides what to try next. Five weeks instead of seven months.

DAY 1 · TORC ROBOTICS

GPU 30–40% → ~90%

Felix Heide, Head of AI

Self-driving trucks: five training systems became one Ray cluster, and the GPUs finally stayed busy.

DAY 1 · INFERACT / VLLM

5,000+ models

Simon Mo, CEO

vLLM runs 5,000+ models on 1,000+ hardware setups with 3,000+ contributors — and a roadmap for every new chip.

DAY 2 · NVIDIA

550B / 55B active

Bryan Catanzaro, Nemotron

Nemotron 3 Ultra was post-trained with RL on 3,000+ GPUs, coordinated by Ray Core; smarter placement gave +13%.

DAY 2 · PHYSICAL AI · SCIENCE

Ray orchestrates

Bedrock · Periodic Labs

Construction robots and AI-for-science labs both use Ray as the conductor of their training pipelines.

Source: Anyscale, Ray Summit 2026 recap (anyscale.com/blog/ray-summit-2026-recap)

Breakouts worth one line each

Big companies, same pattern: one Ray cluster underneath data, training and serving

Google + Anyscale	`TPU slices`	Ray can now reserve whole TPU slices, and Ray Serve runs vLLM on TPUs across many hosts.
Microsoft AI	`2.85× faster`	One Ray layer across Kubernetes and Slurm; a new proxy (RELAY) starts actors 2.85× faster at 32,000 workers.
Spotify	`1M+ GPU-h / month`	About 21,000 batch LLM jobs a month on Ray Data and Ray Train, with vLLM doing the generation.
Capital One	`3–5 days → ~4 h`	Hyper-parameter search for a financial Transformer, on one KubeRay stack (Ray Data, Train, Tune).
Recursion	`Boltz-2 on Serve`	Protein-structure predictions served with Ray Serve for drug discovery.
Agentic API	`Rust`	Embedded LLM, Anyscale, Verda, Novita: lets Codex and Claude Code talk to open models served by vLLM.

Source: Anyscale, Ray Summit 2026 recap (anyscale.com/blog/ray-summit-2026-recap)

The first vLLM Conference, mirrored in Bangkok

Hosted by Inferact inside Ray Summit, 25–26 August 2026 — four of today's five talks retell one of its sessions

VLLM CONFERENCE · SAN FRANCISCO		TODAY · BANGKOK
State of vLLM 2026 Woosuk Kwon, Zachary Xi · Inferact	→	 **09:30 · State of vLLM 2026** Paphanin Amnuaikitwanit
Accelerating Large Language Models: vLLM on Google TPUs Qi Zhou · Google TPU	→	 **10:20 · vLLM on TPUs** Kanin Kearpimy
AMD and vLLM: What's New Douglas Lehr · AMD	→	 **10:40 · AMD and vLLM: What's New** Passin Pornvoraphat
NVIDIA and vLLM Full-Stack Collaboration for DeepSeek and MiniMax Performance Siyuan Fu · NVIDIA	→	 **11:00 · NVIDIA + vLLM for DeepSeek & MiniMax** Arisa Umparat
Also: PyTorch ❤ vLLM · llm-d (Red Hat) · GLM-5 RL training (Prime Intellect) · TorchSpec speculative-decoding training · Serving MiniMax M3 · Serving vLLM on agentic workloads	MADE IN THAILAND	 **09:50 · Omni Context Generation in a 33B Model** Sattaya Singkul · made in Thailand

Source: vLLM Conference 2026 programme (vllm.ai/events/ray-summit)

Three business moves behind the code

Who owns what — in plain words

30 JUL 2026

Nscale agrees to acquire Anyscale

A GPU-cloud company buys the company behind Ray (Bloomberg: about $1.65B; closing expected H2 2026). Anyscale keeps its name and team. Ray stays open source in the PyTorch Foundation.

JAN 2026

Inferact launches with a $150M seed

vLLM's creators (Simon Mo, Woosuk Kwon, Ion Stoica) start a company to support vLLM commercially — backed by a16z and Lightspeed. vLLM itself stays open.

OCT 2025

Ray joins the PyTorch Foundation

Ray, vLLM and PyTorch now live under one neutral foundation. No single company controls the projects we build on.

The talk retold in Part 3 comes from Nscale — the company buying Anyscale.

Sources: Nscale / Anyscale announcement 30 Jul 2026 · Bloomberg · Inferact launch, 22 Jan 2026 · PyTorch Foundation, Oct 2025

Why Ray? One machine stopped being enough

ONE MACHINE

A Python script on one box



1 machine · 1 GPU

- ✕ The model no longer fits one GPU
- ✕ The data no longer fits one disk
- ✕ A thousand experiments run one after another
- ✕ When the box dies, everything dies

WITH RAY

The same Python, on many machines



many machines · one program

RAY

- Ray decides where each function runs
- It moves data between machines for you
- It retries when a machine fails
- Laptop → Kubernetes → cloud, same code

Ray = a way to run ordinary Python on many machines as if they were one.

Source: docs.ray.io — Ray overview

Ray: Python-first distributed compute

Born at UC Berkeley RISELab, 2016–17 · open source · one core, five libraries

`laptop → Kubernetes (KubeRay) → any cloud`

RAY

LIBRARIES

Ray Data	Ray Train	Ray Tune	Ray Serve	RLlib
batch inference, Ray Data LLM	distributed training, TorchTPU	hyper-parameter search	online serving, Ray Serve LLM	reinforcement learning

RUNTIME

Ray Core — tasks · actors · objects

44.1k

GitHub stars

github.com/ray-project/ray · 23 Oct 2026

2.59

latest release · 2 Oct 2026

Ray Serve LLM + Ray Data LLM GA · vLLM 0.27.0

2.58

23 Aug 2026

TPU subslice placement groups · TorchTPU

Sources: docs.ray.io · Ray 2.58 / 2.59 release notes · github.com/ray-project/ray

Ray in one minute: tasks, actors, objects

```python
import ray
ray.init()                          # join the cluster, or start one locally

@ray.remote                         # a TASK: may run on any machine          1
def embed(batch):
    return model(batch)

@ray.remote(num_gpus=1)             # an ACTOR: own state, own GPU            2
class Engine:
    def generate(self, prompt): ...

futures = [embed.remote(b) for b in batches]   # runs in parallel
results = ray.get(futures)                     # OBJECTS, when ready          3
```

1. **Task** — a plain function; Ray picks a machine for it
2. **Actor** — a long-lived worker with state, e.g. one GPU engine
3. **Object** — results live in the cluster's shared memory; ray.get fetches them

BUILT ON THESE THREE

- Ray Data → batches over datasets
- Ray Train → distributed training
- Ray Tune → hyper-parameter search
- Ray Serve → online serving (and Ray Serve LLM)
- RLlib → reinforcement learning
- pip install "ray[default]"

Source: docs.ray.io — Ray Core walkthrough (illustrative snippet)

Who is Anyscale?

The company behind Ray — and our partner for this recap

2019 · SAN FRANCISCO

Founded by Ray's creators

Robert Nishihara, Philipp Moritz and Ion Stoica (with Michael I. Jordan) from UC Berkeley RISELab started Anyscale in 2019 to take Ray to production. They host Ray Summit every year.

WHAT THEY SELL

Ray as a managed service

The Anyscale platform runs Ray clusters for you on your own cloud, with RayTurbo, their optimised Ray runtime. You bring the Python; they run the machines.

RAY STAYS OPEN

Open source, foundation-governed

Ray itself is open source under the PyTorch Foundation (since Oct 2025). In July 2026 Anyscale agreed to join Nscale; the brand and the team continue.

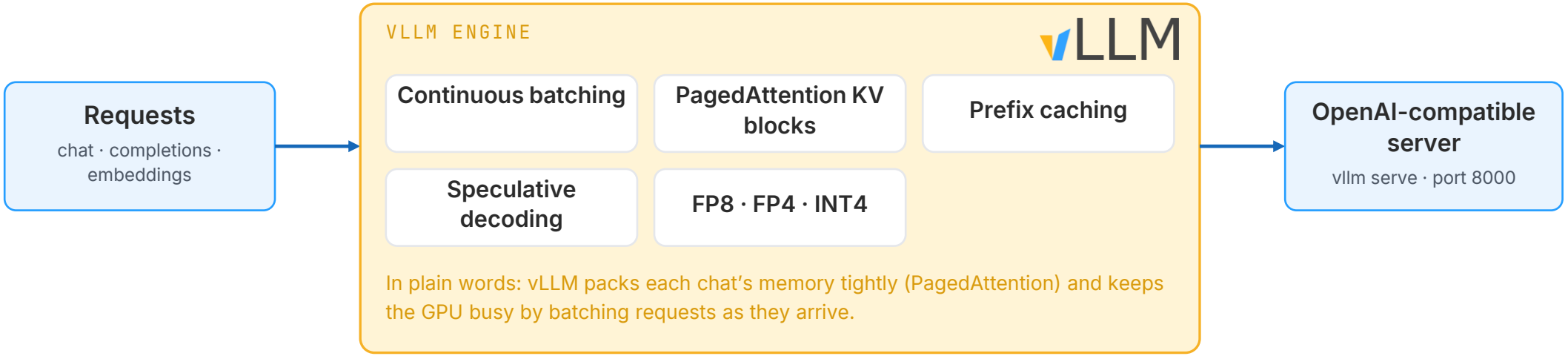
Thank you to Anyscale for partnering with vLLM Bangkok on today's recap.

Anyscale | RAY

Sources: anyscale.com/about · Anyscale launch release, Dec 2019 · PyTorch Foundation, Oct 2025 · Nscale, 30 Jul 2026

vLLM: the open LLM inference engine

The program that answers many chats at once on one GPU — UC Berkeley Sky Computing Lab, 2023



RUNS ON

NVIDIA · AMD ROCm · Google TPU · Intel · Apple Silicon · CPUs

2023
PagedAttention paper, vLLM 0.1

2025
V1 engine: new scheduler, zero-overhead prefix caching

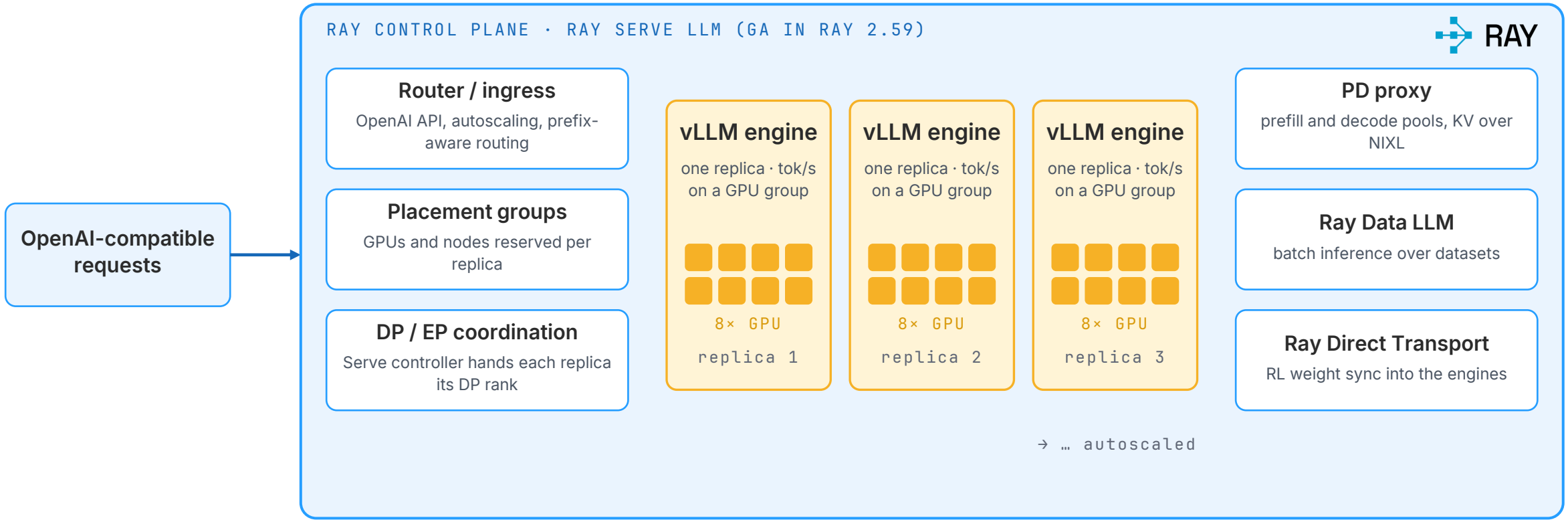
Mar 2026
Model Runner V2: modular, faster core

Oct 2026
v0.32.0 (18 Oct); day-0 MiniMax-M3.1-Flash this week

95.2k
GitHub stars
github.com/vllm-project/vllm · 23 Oct 2026

Sources: vLLM docs · Kwon et al., SOSP 2023 · vLLM blog (Model Runner V2, 24 Mar 2026) · github.com/vllm-project/vllm

vLLM is the engine, Ray is the control plane



vLLM answers "how fast". Ray answers "where, how many, and how they talk to each other".

Sources: Ray Serve LLM docs (Ray 2.59) · Anyscale, Ray Serve LLM wide-EP and P/D APIs (26 Nov 2025) · vLLM blog, RDT (22 Aug 2026)

What happens to one request

The same picture, told as a story — this is what "Ray around vLLM" means in practice

1

Your app sends a request

An OpenAI-style chat call to one URL — no idea which GPU will answer.

2

Ray Serve picks a replica

The router chooses a healthy replica, preferring one that already has your prefix cached.

3

vLLM batches it

Inside the replica, your request joins others already running on the GPUs.

4

Tokens stream back

Words arrive one by one while the next ones are still being computed.

5

Ray keeps it alive

More traffic → more replicas. A machine dies → Ray restarts the engine elsewhere.

RAY SERVE

VLLM

RAY SERVE

Ray does the "where, how many, keep it running". vLLM does the "fast".

Source: Ray Serve LLM docs (docs.ray.io/en/latest/serve/llm)

Ray Serve LLM: the whole API on one slide

```python
from ray import serve
from ray.serve.llm import LLMConfig, build_openai_app

llm = LLMConfig(
    model_loading_config={
        "model_id": "deepseek-ai/DeepSeek-V4-Flash",
        "model_source": "/models/DeepSeek-V4-Flash",   # shared PVC
    },
    deployment_config={
        "autoscaling_config": {"min_replicas": 1, "max_replicas": 4},
    },
    accelerator_type="B200",
    engine_kwargs={"tensor_parallel_size": 8,
                   "enable_expert_parallel": True,
                   "max_model_len": 262144},   # straight to vLLM
)
serve.run(build_openai_app({"llm_configs": [llm]}), blocking=True)
```

1. **LLMConfig + build_openai_app** — one OpenAI-compatible app per list of models
2. **deployment_config** — Ray Serve autoscaling, min and max replicas
3. **engine_kwargs** — every vLLM flag, by its EngineArgs name

ALSO IN THE BOX

- build_dp_deployment → wide expert parallelism
- build_pd_openai_app → prefill / decode pools
- chat · completions · embeddings · multi-LoRA
- TP · PP · EP · DP attention · prefix-aware routing
- vLLM or SGLang backend · metrics + Grafana
- pip install "ray[llm]"

Source: Ray Serve LLM docs, docs.ray.io/en/latest/serve/llm (Ray 2.59, GA 2 Oct 2026) · example as documented, verify on your version

PART 3 · SERVING A FRONTIER OPEN MODEL

Serving a frontier open model with Ray and vLLM

DeepSeek-V4-Flash on Nscale — deployment shapes on a single node, prefill/decode disaggregation, and the caches behind a 1M-token window.



scan for the full recording

Retold from the Ray Summit 2026 breakout by

Eyal Lantzman, VP of AI Services, Nscale

Recording: youtube.com/watch?v=bgem5_ooJqk · Anyscale channel · 17 Sep 2026 · 28:27

VISUAL RECREATION · SAME INFORMATION, OUR STYLE, CREDITED ON EVERY SLIDE

Four words you need for the next ten minutes

01

KV cache

The model’s memory of a conversation, kept on the GPU. Longer chats need more of it. When it runs out, fewer people can be served at once.

02

TP and EP: how to split a model

A model too big for one GPU gets cut up. TP slices every layer across GPUs. EP gives each GPU its own experts of a mixture-of-experts model.

03

Prefill vs decode (P/D)

Prefill reads your prompt: heavy compute, once. Decode writes the answer token by token: light compute, limited by memory speed. P/D = separate GPU pools for each.

04

tok/s · TTFT · TPOT

tok/s: tokens per second for everyone together. TTFT: how long until the first word. TPOT: the pause between words.

Remember these four and every slide that follows will make sense — the rest is detail.

DeepSeek-V4-Flash at a glance — big model, small footprint

Huge on paper, small at work: only a sliver of the model runs per token, and it fits one machine

284B / 13B

total vs active parameters

Sparse MoE: 257 experts per layer (1 shared + 256 routed), only 6 routed experts fire per token.

1M

token context window

A million-token window, kept affordable by the hybrid CSA + HCA attention.

CSA + HCA

compressed attention

Compressed Sparse Attention picks what to look at; Heavily Compressed Attention shrinks how it is stored: ~10% of V3.2's cache at 1M.

FP4 + FP8

ships already quantised

A 167 GB checkpoint (74 files), quantisation-aware rather than an afterthought.

DSpark

speculator built in

The speculative-decoding draft ships inside the same checkpoint — one vLLM flag, no second download.

1 node

everything fits one machine

Reference deployment is a single 4× GB300; the demo ran strategies A–C on one 8× B200 machine (D: two).

Every config pinned to vLLM v0.27.1 · recipes.vllm.ai/deepseek-ai/DeepSeek-V4-Flash · the 0731 release with the fused DSpark draft

Four ways to split it — and the four shapes tested

A model that does not fit one GPU has to be cut up somehow — these are the four knives



TP

tensor parallel

Slice every layer across the GPUs — each holds a piece of everything. They talk constantly, so it only works inside one NVLink domain. Fastest answers; least memory left for the cache.

IN THIS TALK

Part of strategy A's slicing — and strategy C's TP4 prefill pool



TEP

tensor + expert parallel

Slice attention like TP, but give each GPU whole experts of its own. Far less cross-GPU chatter; every expert runs at full speed. The natural fit for a mixture-of-experts model.

IN THIS TALK

Strategy A · TEP8 — one machine, the latency shape



PP

pipeline parallel

Cut the model into stages: machine 1 runs the first half of the layers, machine 2 the rest — an assembly line over InfiniBand or RoCE. Stations idle when traffic is light.

IN THIS TALK

Out of scope today — strategy D spans machines with whole replicas instead



DEP

data + expert parallel

Every GPU keeps a full copy of attention plus its share of the 256 experts; each token travels to wherever its experts live. Biggest throughput in theory — on 8× B200 it lost to TEP8.

IN THIS TALK

Strategy B · DP8×EP, the published recipe — and strategy C's decode pool, DP4×EP

STRATEGY C — prefill/decode disaggregation: two pools sized independently, the cache handed from prefill to decode per request.
STRATEGY D — whole replicas: one full engine per machine behind one Service; the only traffic between machines is complete requests.

NVLink = the fast link between GPUs in one domain (one machine here, a whole NVL72 rack elsewhere) · RoCE = RDMA over Converged Ethernet, a backend network between domains, like InfiniBand

Rules of thumb: when each strategy earns its keep

Cheat sheet: pick by what you need — speed, capacity or cost — then measure on your own hardware

Strategy	Reach for it when	Think twice when
TP **tensor parallel**	**Latency-sensitive serving inside one NVLink domain; dense layers shard cleanly.**	KV-hungry long context; links slower than NVLink; sizes that don't divide the heads. Measure it: the shape moved KV capacity far less than the engine build did (5.03M vs 5.05M tokens, TP8 vs DP8 — same 8 GPUs).
PP **pipeline parallel**	**Crossing the backend network (InfiniBand or RoCE) between NVLink domains; freeing HBM once TP is maxed out.**	Low-batch, latency-critical traffic — pipeline bubbles dominate at small batch sizes.
EP / TEP **expert parallel**	**Sparse MoE like V4-Flash (6 of 256): each GPU owns whole experts, so they run at full speed with less cross-GPU traffic than pure TP.**	Dense models (nothing to shard); expert routing that would have to leave the NVLink domain; imbalanced experts without EPLB.
DEP **data + expert**	**Throughput-per-dollar at scale; the decode pool in P/D setups. Measure before you believe it: here TEP8 beat DP8×EP, 2,209 vs 1,634 tok/s.**	Single-user latency (all-to-all bound); small setups where replicating attention wastes HBM.
DCP **decode context**	**Decode-heavy, very long context — shards decode KV across the TP ranks.**	Prefill-heavy traffic; the DCP size must divide the TP size.

KV (cache) = the model's stored working memory of the conversation · HBM = the GPU's onboard memory · EPLB = expert load balancer · P/D = prefill/decode split · DCP = decode context parallelism · all-to-all = every GPU exchanging data with every other GPU at once

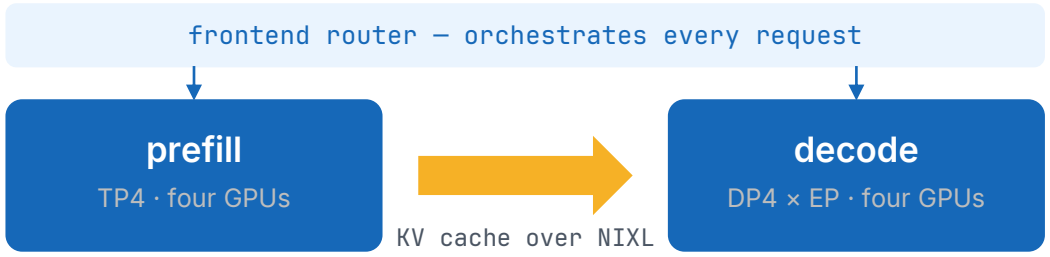
P/D disaggregation — the shape, and where it fits

Reading the prompt and writing the answer are different jobs — so give them different GPUs

WHY DISAGGREGATE

Prefill is compute-bound. Decode is memory-bandwidth-bound.

Running both on one engine means one deployment shape serving two opposite bottlenecks. Splitting them into separate pools lets you size, parallelise and scale each for its own bottleneck — and hand the cache from prefill workers to decode workers. The split has a price: a frontend router orchestrates every request across the pools — extra compute and latency in the equation.



HOW IT RAN IN THE DEMO

Two pools, sized for their own bottleneck. On 8 GPUs a 4+4 split needs TP4 — silently wrong for this model on v0.27.1 — so P/D belongs across two machines here, where each pool gets the known-correct TP8.

NIXL carries the cache between pools — paged KV blocks plus the sparse-attention indexer cache move from the prefill engine to the decode engine per request.

Different shapes, one connector — the pools use different parallelism, so they lay the cache out differently; the connector maps blocks between layouts.

The same pattern scales out — swap the decode bundle for "the decode machines" and nothing else changes; pools grow independently under load.

LIVE-DEMO LESSON

Strategy C reached ~1,700 tok/s — and answered "what is 2 + 2" with garbage. Eval the outputs, not just tok/s. Pin the engine version: one vLLM build was 45% slower on the same config.

KV (cache) = stored attention state per token · NIXL = NVIDIA's cache-transfer library · indexer cache = the sparse-attention lookup index kept alongside the KV cache

Long context: the cache is the product — and the knobs

A one-million-token window is real — but the memory behind it is what you actually pay for

Hybrid compression: CSA (Compressed Sparse Attention) narrows which tokens attention even looks at; HCA (Heavily Compressed Attention) shrinks how they are stored. Net effect: roughly 10% of V3.2's KV cache at 1M tokens. The cache is still the resource you manage — there is just ten times less of it.

⚠ fp8 KV and block-size 256 travel together

Every published V4-Flash command pins --kv-cache-dtype fp8 with --block-size 256. Keep the pair — don't cherry-pick one without the other.

⚠ There is a second cache: the indexer

Sparse attention keeps a lookup index alongside the KV blocks. --attention-config '{"use_fp4_indexer_cache": true}' shrinks it to 4-bit — remember it when budgeting memory.

⚠ Thinking eats context too

Three reasoning-effort levels (low / high / max); DeepSeek allows up to 384K output tokens at high and max. Long reasoning fills the cache like any other tokens — cap it per workload.

⚠ max-model-len is the context-vs-throughput dial

Native window 1M; the demo serves 256K. Same 8× B200, only this flag changed: at 1M the KV cache holds 15.6M tokens vs 5.0M at 256K — per-token cost falls 3.1×, so 4× the window costs 1.3× in concurrency (19.2× → 14.9×), not 4×. A needle at the midpoint of 996K tokens returned in 63.7 s.

KV (cache) = stored attention state per token · fp8 / FP4 = compact 8- and 4-bit number formats (smaller cache, same layout) · CSA / HCA = the two attention-compression stages defined above

The whole deployment is one command

```
# image: vllm/vllm-openai:v0.27.1
vllm serve /models/DeepSeek-V4-Flash-0731 \
  --distributed-executor-backend ray \
  --served-model-name deepseek-ai/DeepSeek-V4-Flash-0731 \
  --trust-remote-code \
  --kv-cache-dtype fp8 \                                  4
  --block-size 256 \
  --data-parallel-size 8 \                                1
  --enable-expert-parallel \
  --moe-backend deep_gemm_mega_moe \                      2
  --attention-config '{"use_fp4_indexer_cache":true}' \
  --speculative-config '{"method":"dspark",               3
      "num_speculative_tokens":7,
      "draft_sample_method":"greedy"}' \
  --tokenizer-mode deepseek_v4 \
  --tool-call-parser deepseek_v4 --enable-auto-tool-choice \
  --reasoning-parser deepseek_v4 \
  --max-model-len 262144                                  5
```

1. **DP8 × EP** — the published reference shape, widened from DP4 on 4× GB300 to 8× B200; here it loses to TEP8.
2. **deep_gemm_mega_moe** — the expert-routing engine built for the NVLink domain, where the expert traffic lives.
3. **DSpark, one flag** — the draft ships inside the checkpoint; k=7 gave ~1.2× aggregate throughput.
4. **fp8 KV + block-size 256** — the recipe pins these together. Keep the pair.
5. **--max-model-len 262144** — 256K for the demo; the native window is 1M. The context-vs-throughput dial.

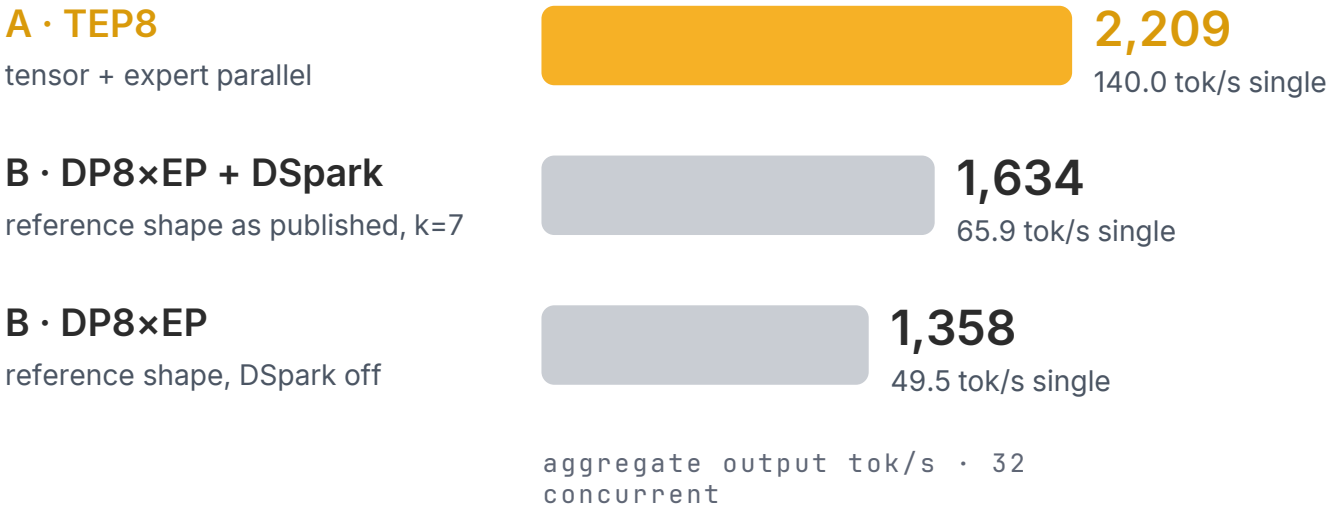
SAME FLAGS AS A KUBERAY RAYSERVICE (serveConfigV2)

```
applications:
- name: llm
  import_path: ray.serve.llm:build_openai_app
  args:
    llm_configs:
    - model_loading_config:
        model_id: deepseek-ai/DeepSeek-V4-Flash-0731
        model_source: /models/DeepSeek-V4-Flash-0731
      engine_kwargs: { data_parallel_size: 8,
        enable_expert_parallel: true, kv_cache_dtype: fp8,
        block_size: 256, max_model_len: 262144 }
```

MEASURED · 8× B200 · VLLM V0.27.1 · 32 CONCURRENT

The published reference shape lost to the simpler one

Same model, same 8 GPUs, same software — only the flags differ. The official recipe was not the fastest here.



	A · TEP8	B · DP8×EP	D · replicas
TTFT p50, single	0.141 s	0.193 s	0.145 s
TPOT p50, single	6.62 ms	14.27 ms	7.45 ms
TTFT p95 under load	0.709 s	0.795 s	1.031 s
KV cache (tokens)	5,029,381	5,046,561	5,029,381

Latency rows: the live benchmark run shown in the talk (32 concurrent; D at 64).

FOR REFERENCE

D · 2 replicas, one per machine — 4,647 tok/s at 32 req/replica — 2.10× of A, on twice the machines and twice the load (off-axis)

SO WHAT

- The published reference shape reached 74% of plain TEP8's throughput with 1.37× the TTFT.
- Its speculator is a win: DSpark adds 1.20×.
- So scale out the winner — D replicates A at 4,647 tok/s across two machines. Every shape is one kubectl apply away.

One PVC, one RayService per strategy — one apply

Download the weights once; each strategy is one YAML file; switching is one kubectl apply

Shared PVC (RWX)

A one-time Job pulls the 167 GB FP4+FP8 checkpoint (DSpark draft included) to shared storage. Every strategy mounts it read-only — switching never re-downloads.

RayCluster on B200

Each strategy is a RayService: KubeRay creates the cluster, Ray Serve runs the engine, and the head holds no GPUs — one 8× B200 worker group per strategy. Ray drives the GPUs; what differs between strategies is only the parallelism.

Four shapes, four applies

A · TEP8 on one machine (latency) → B · DP8×EP + MegaMoE + DSpark (the published reference) → C · P/D, TP4 prefill + DP4×EP decode → D · whole replicas, one per machine. Same weights, same PVC; engines restart in minutes.

`Kubernetes on Nscale, cluster in Norway · console.nscale.com`

PVC = a Kubernetes storage claim (a shared disk) · RWX = readable/writable by many machines at once · RayService = a Ray Serve application KubeRay manages as a Kubernetes resource

What to remember when you build this yourself

Four rules from the demo — and two more from what went wrong

01

Match the strategy to the wiring

Keep the chatty parallelism — expert routing, cache handoffs — inside the NVLink domain. Between NVLink domains, send only what travels light: pipeline activations or whole requests.

02

Disaggregation is a pattern, not a cluster size

P/D works on a single machine: two pools and a connector. Prove it small, tune each pool for its own bottleneck — the same NIXL handoff scales out unchanged.

03

The cache is still the product at 1M

CSA + HCA cut it roughly 10×; the rest is yours to manage — fp8 KV with block-size 256, the fp4 indexer cache, and an output cap for long thinking.

04

The flags ARE the architecture

Every deployment shape is a vllm serve argument away. Download the weights once to shared storage — switching strategies should be a kubectl apply, not another 200 GB.

From the live demo: benchmark performance AND outputs — fast junk is still junk. Test the exact engine version you will ship; versions differ in speed and in bugs.

CLOSING

What’s next for vLLM Bangkok

vllmbangkok.com

Events, recaps, slides and recordings — today’s decks go up after the event.

Directory

Teams running vLLM in production in Thailand. If yours does, add it.

Blog

Bilingual write-ups; guest posts welcome.

Membership

Join the community for the next meetup — and come speak at it.

vllmbangkok.com



Every ticket today includes the community T-shirt

Want to speak next time? Find me at networking from 11:20.

NEXT UP · 09:30

State of vLLM 2026

Paphanin Amnuaikitwanit

AI Engineer, vLLM Bangkok

Thank you — Natdhanai Praneenatthavee (Job) · AGICAFET · vLLM Bangkok

vLLM BANGKOK COMMUNITY | AGICAFET | Anyscale

