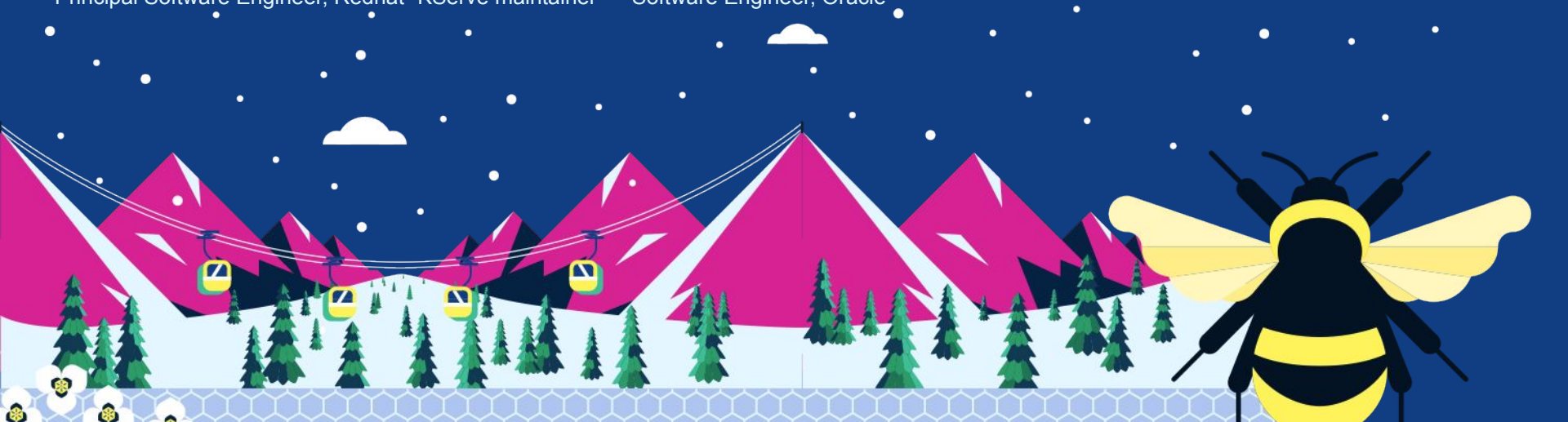
KubeCon CloudNativeCon North America 2026

#KubeCon #CloudNativeCon

Reduce vLLM Pod Cold Starts with KServe Kernel Caching

Jooho Lee
Principal Software Engineer, Redhat· KServe maintainer

Vishwa Hiren Gandhi
Software Engineer, Oracle



What is vLLM?

Open-Source LLM Serving Engine

Runs pretrained models and serves inference requests

High Performance

Optimized for high-throughput, low-latency inference

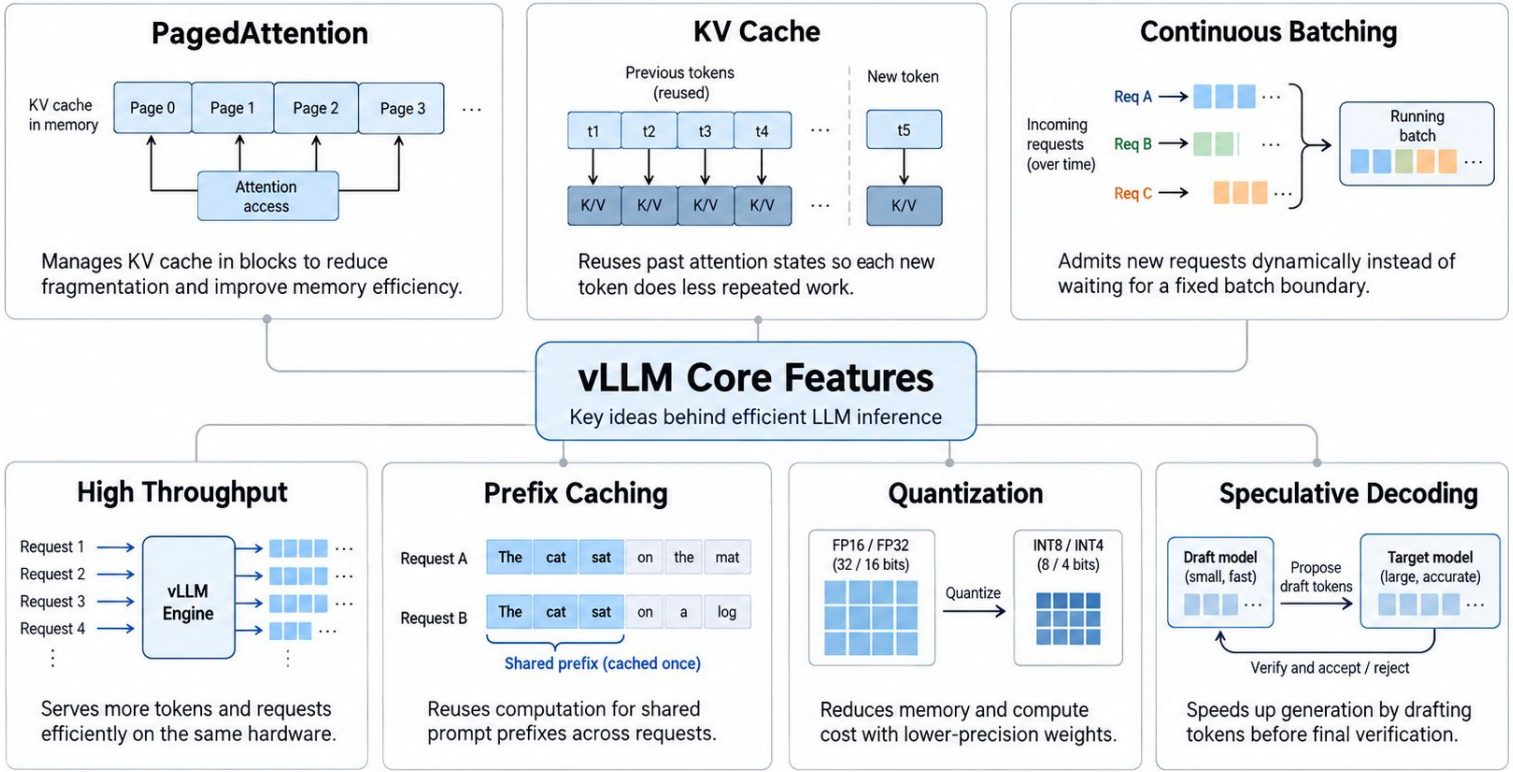
Hardware & Model Support

Supports a wide range of models and GPU architectures

Key Optimization Techniques

Uses techniques such as Paged Attention, continuous batching, and model compilation

vLLM Core Features



Why vLLM?

De facto choice

A widely adopted inference engine for serving large language models.



Fast model adoption

Rapid support for new models and architectures.



High Performance inference

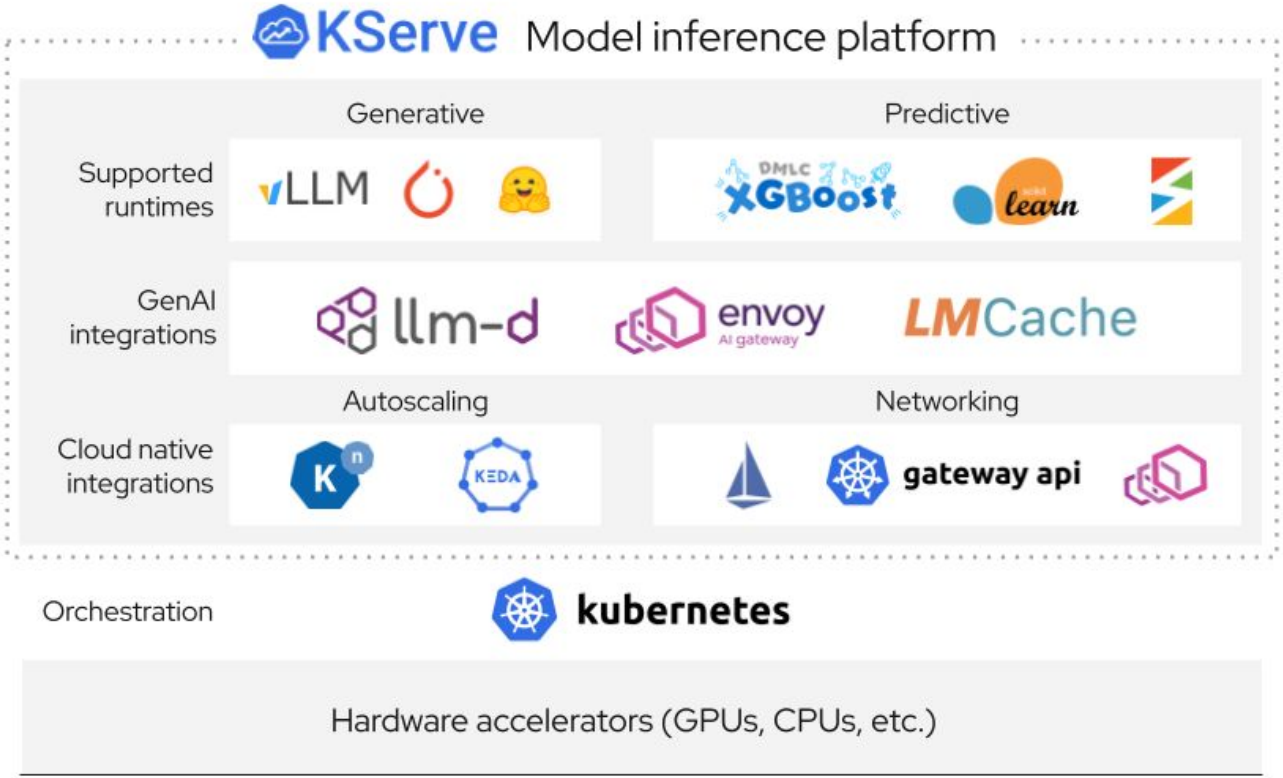
Support across GPU platforms and diverse deployment environments.



High-performance inference is valuable
but starting the server can still be expensive.

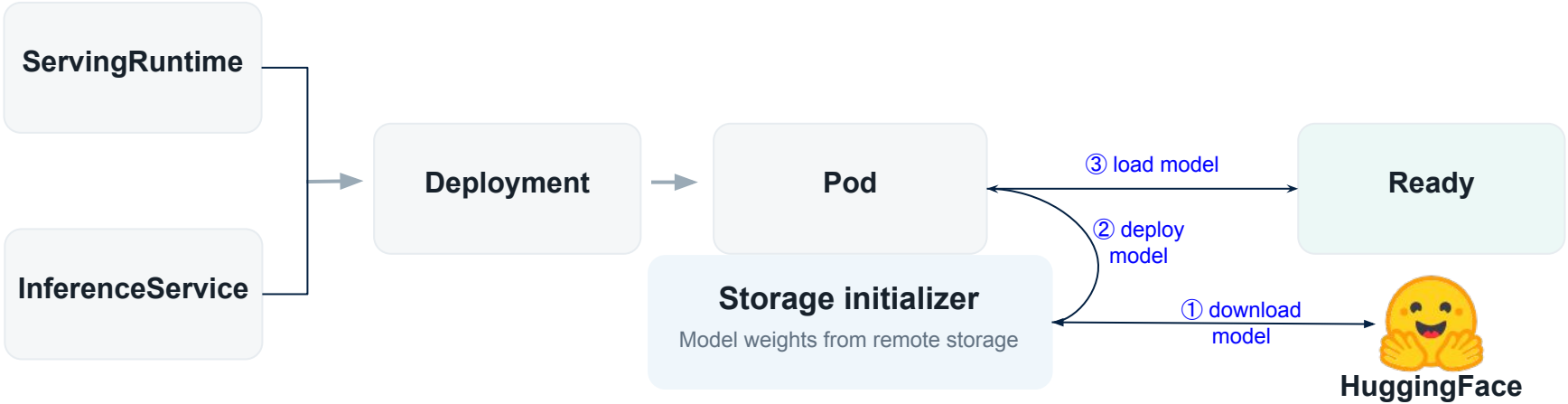
What is KServe?

KServe is a CNCF project for deploying and managing model inference on Kubernetes.



How do we deploy vLLM using KServe?

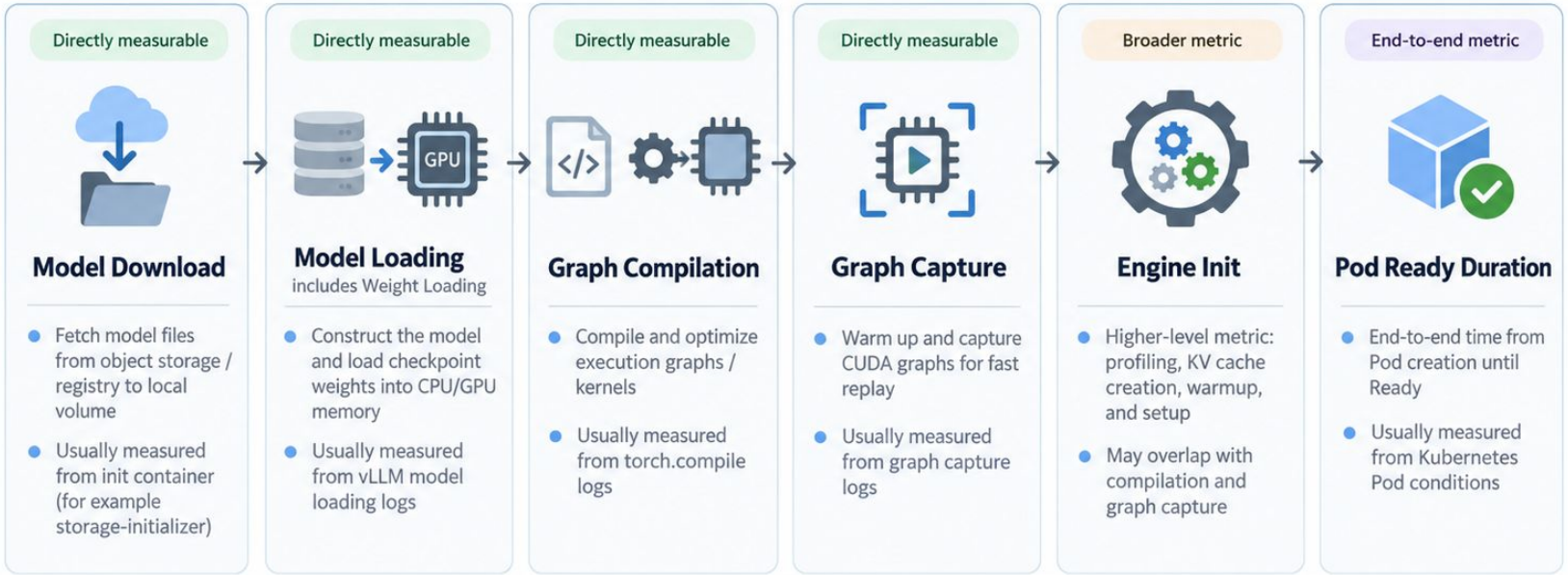
Standard deployment mode



KServe manages the Kubernetes resources; vLLM still performs its own runtime initialization.

vLLM StartUp Metrics on Kubernetes

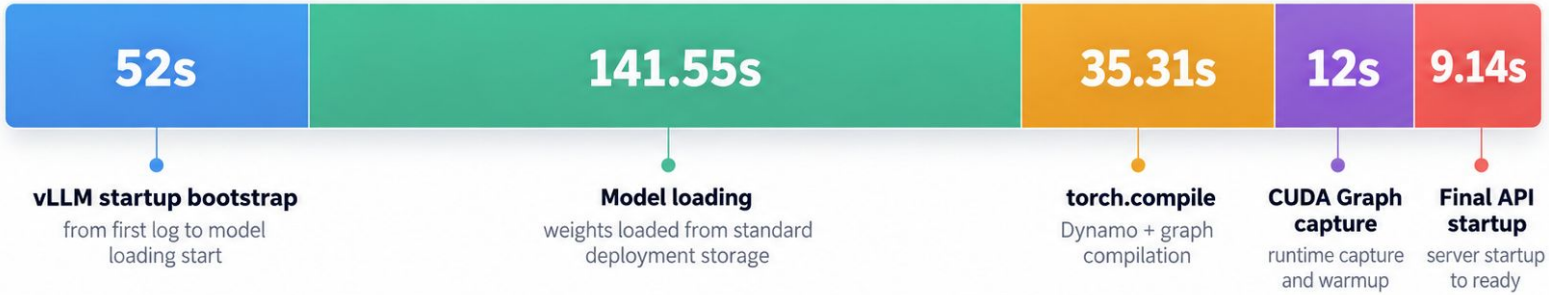
What each metric means and where it is usually measured



Initial deployment: where time goes

It shows pure vllm deployment time only

Total startup: 250s



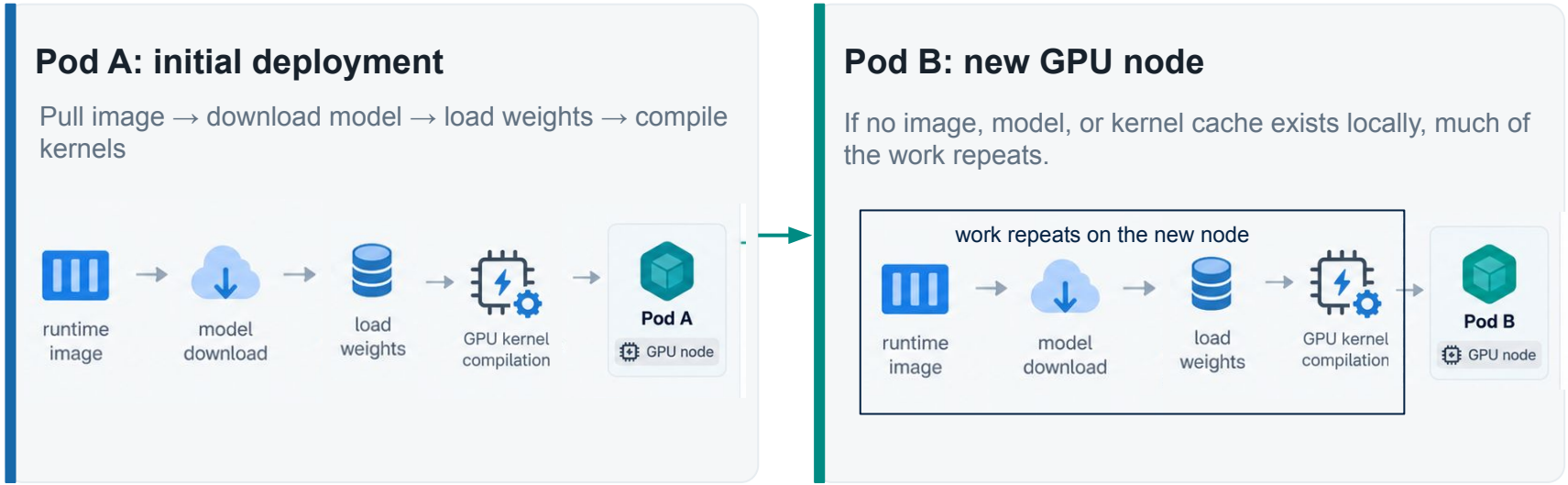
What stands out

1. Model loading is the biggest part of startup.
2. Compilation is still a significant cost.
3. This is the baseline before LocalModel and KServe Kernel Cache optimizations.

Share of total time (≈250s)

Model loading	57%
torch.compile	14%
CUDA Graph capture	5%
Other startup overhead (vLLM bootstrap + final API startup)	24%

What happens when we scale out?



A new Pod does NOT automatically inherit the previous Pod's compiled artifacts.

A cold start has multiple costs

01 Downloading Runtime image

Pull the vLLM container image from the registry before starting execution.

02 Downloading Model weights

Download large model parameters and check artifact integrity.

03 Compiling GPU kernels

Compile and cache optimized GPU execution artifacts dynamically.

04 Loading weight to GPU memory

Load weights into VRAM and initialize the runtime context for execution.

Which startup costs can we reduce?

Model download	Pre-stage or locally cache weights	**KServe Local Model Cache** 
Image pull	Pre-pull runtime images	
Kernel compilation	Reuse previously compiled artifacts	
Model weight loading	More constrained by runtime and hardware	**Not available**

Today's focus: make compiled GPU kernels reusable.

Why compile GPU kernels at runtime?

To specialize execution for the actual GPU and model configuration.

Runtime environment

- GPU architecture (SM capability)
- Model shapes, dtype & settings
- vLLM / PyTorch / Triton stack
- CUDA / driver compatibility

→

Runtime compilation

torch.compile / Triton

→

Specialized GPU kernels

- GPU-specific instructions
- Shape-specialized operations
- Optimized execution paths
- Potential throughput & latency gains

Benefit: better runtime performance

Trade-off: extra startup latency

What if we could reuse optimized kernels without compiling them on every pod startup?

Compile once. Reuse when compatible.

First startup

vLLM compiles kernels and writes artifacts to its cache directory.

Initial Build



Cache

Compiled Artifacts



Subsequent startup

vLLM finds compatible compiled artifacts and can skip redundant compilation.

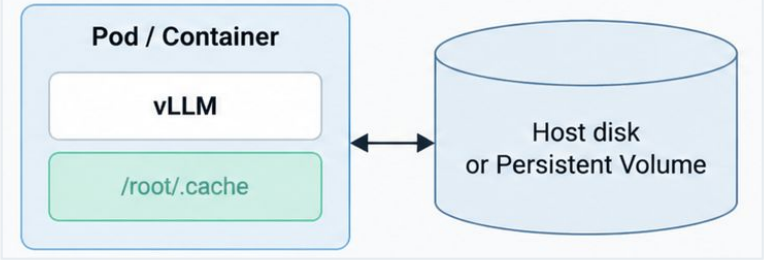
Fast Startup

Note: Reuse depends on compatibility — not every artifact works for every model, runtime, or GPU.

Cache reuse depends on storage

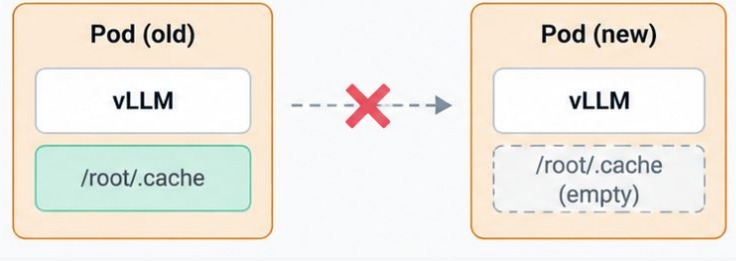
Persistent host or volume

A cache directory can survive process or container replacement if backed by persistent storage.



Disposable Pod storage

A newly created Pod may start without the artifacts compiled by its predecessor.

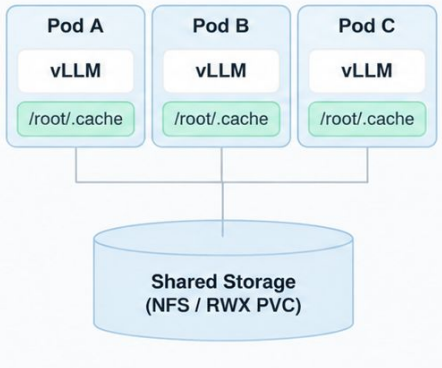


Compatible artifacts must persist and remain accessible across pods

Could we just share the cache directory?

Shared network filesystem

A ReadWriteMany volume can expose cached artifacts to multiple Pods.



Pros

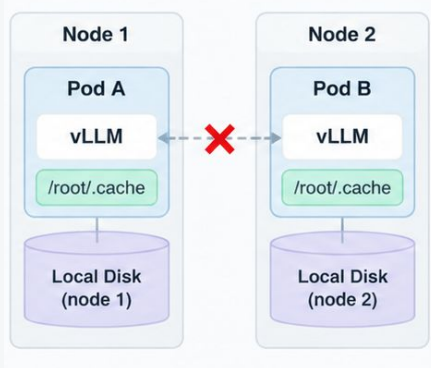
- Multiple Pods can access the same cache
- Survives Pod replacement

Cons

- Additional infrastructure (e.g., NFS/RWX)
- Network latency
- Single point of failure
- May not scale for large artifacts

Node-local filesystem

Node-local storage can preserve artifacts but complicates scheduling and host access.



Pros

- High performance (local disk)
- No additional infrastructure

Cons

- Not accessible from other nodes
- Pods are tied to a specific node
- Cache not available on new nodes
- Lost when node is drained

Enterprise question: who created the cached code, and should we trust it?

Make the cache portable and trusted

01

Package

Store compiled cache artifacts as OCI images.

02

Sign

Apply artifact signatures with a trusted signing identity.

03

Verify

Enforce signature verification before use.

04

Distribute

Pull and reuse caches across compatible GPU nodes.

Signature verification establishes provenance; compatibility checks still matter.

KServe Adopts GPU Kernel Caching

From GPU Kernel Manager to a native, optional capability in KServe.

Earlier direction

Persistent Volumes for cache reuse

KServe integration

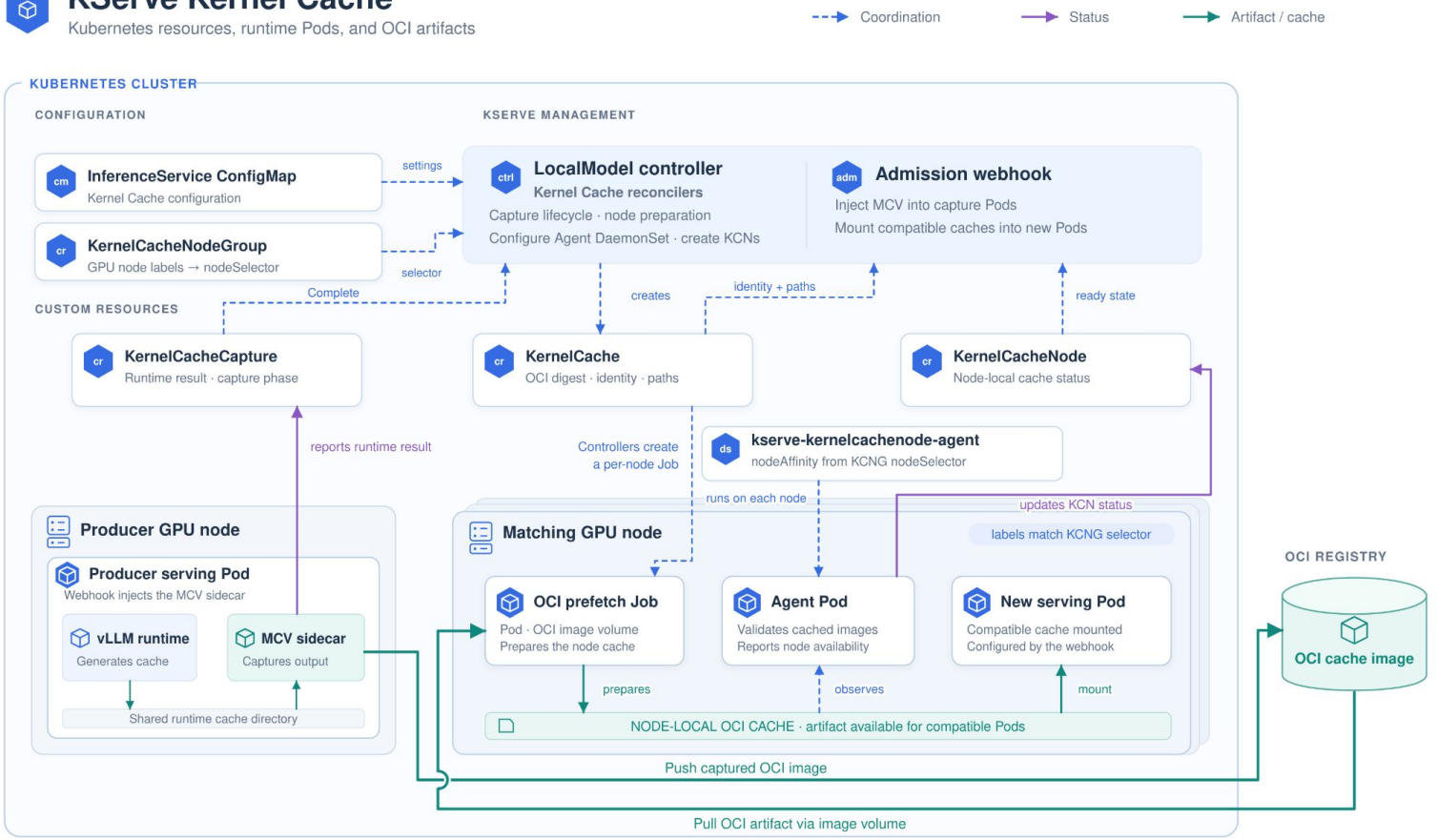
OCI-based distribution and signing / verification

Operational controls

Registry connectivity and authentication options

KServe Kernel Cache

Kubernetes resources, runtime Pods, and OCI artifacts



Make the cache portable and trusted in KServe Kernel Cache

KServe Kernel Cache / From compilation to verified reuse across GPU nodes

PRODUCER

Capture → Package → Sign

01



Capture

Compiled kernels from first Pod

02



Package

Bundle cache as OCI artifact

03



Sign

Trusted signing identity

Output: signed, portable cache artifact

PUSH

OCI REGISTRY

Store & distribute



Signed OCI cache artifact

Artifact + signature stored in registry

PULL

Prefetch

CONSUMER

Verify → Prefetch → Reuse

04



Verify

Signature

Compatibility

05



Reuse

On compatible GPU nodes

Result: cache ready before Pod startup

Portable through OCI • Trusted through signatures • Safe to reuse through compatibility checks

Cache reuse requires a compatible workload

Artifact identity

SHA-256 fingerprints identify cache origin and reuse conditions.

Workload footprint

Identifies a cache created for the same workload

Namespace · Workload kind/name · Runtime image

Model URI hash · Command hash · Args hash

Compatibility footprint

Identifies reuse across compatible workloads

Runtime image · Model URI hash

dtype · quantization · max model length

Block size · Attention backend · Compile config / env

Compatibility decision

Find a safe match for the new workload, in priority order.

1. **Exact match**
   Workload + compatibility footprints match
   Reuse existing cache

2. **Fallback reuse**
   Compatibility footprint matches another workload
   Reuse cache across workloads

3. **No compatible cache**
   No acceptable workload or compatibility match
   Start a new capture, if eligible

Cache identity determines reuse. Trust alone is not enough.

Startup Performance Test Setup

Measuring real vLLM startup time across different storage and caching strategies

Qwen2.5-32B-Instruct-AWQ

vLLM 0.28.0 · NVIDIA L40S (48 GB) (AWS EC2 g6e.2xlarge) · KServe

Same model

18.00 GiB checkpoint

Same runtime

AWQ · TP=1 · max len 32,768

One test at a time

Storage and cache strategy vary

Readiness here = vLLM first log → API server start. Pod creation, pulls and initContainers excluded.

Four runs, two optimizations

Each scenario adds one change to the previous deployment

#1	**Standard**	Regular ISVC · XFS checkpoint	**No kernel cache**
#2	**LocalModel**	Pre-staged weights mounted from NFS	**Compile from scratch**
#3-A	**KernelCache Capture**	Same NFS path + MCV sidecar	**Compile, then capture artifacts**
#3-B	**KernelCache Reuse**	Same NFS path + cache linker init	**Load precompiled AOT artifacts**

#3-A and #3-B form the controlled comparison for kernel cache reuse.

LocalModel reduces model loading time

Standard ISVC vs cached model on NFS



Model loading	torch.compile
141.55 → 60.46s	35.31 → 35.73s
57.3% lower	No improvement

Observed startup: 250s → 169s. XFS vs NFS differs; do not attribute all loading gains to caching.

Capture adds no clear compile penalty

Compare LocalModel alone with LocalModel + MCV sidecar



torch.compile

35.73 → 35.27s

Nearly unchanged

Engine initialization

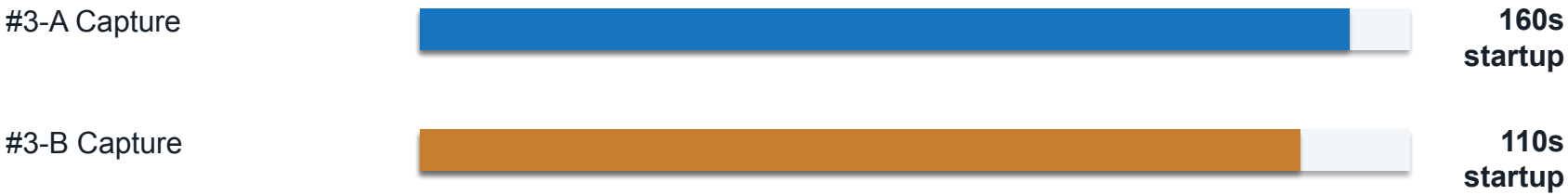
55.83 → 55.40s

Nearly unchanged

#3-A still compiles normally; MCV captures the generated kernel cache for future reuse.

Reusing Kernel Cache reduced startup time

Direct comparison: #3-A Capture vs. #3-B Reuse



torch.compile

35.73 → 1.16s

96.7% reduction

Engine initialization

55.40 → 18.29s

67% reduction

Evidence: “Directly load AOT compilation” · 3 artifacts reconstructed · 65 submodules

What do the measurements mean?

Nested measurements — do not add compilation or CUDA Graph capture to Engine Initialization

vLLM startup (overall measured interval) — **Standard 250s | Cache reuse 110s**

Model loading

Load model weights
and initialize model state

141.55s → 51.74 ~ 56.35s

Engine initialization

Includes the following measured operations:

torch.compile
35.31s → 1.16s

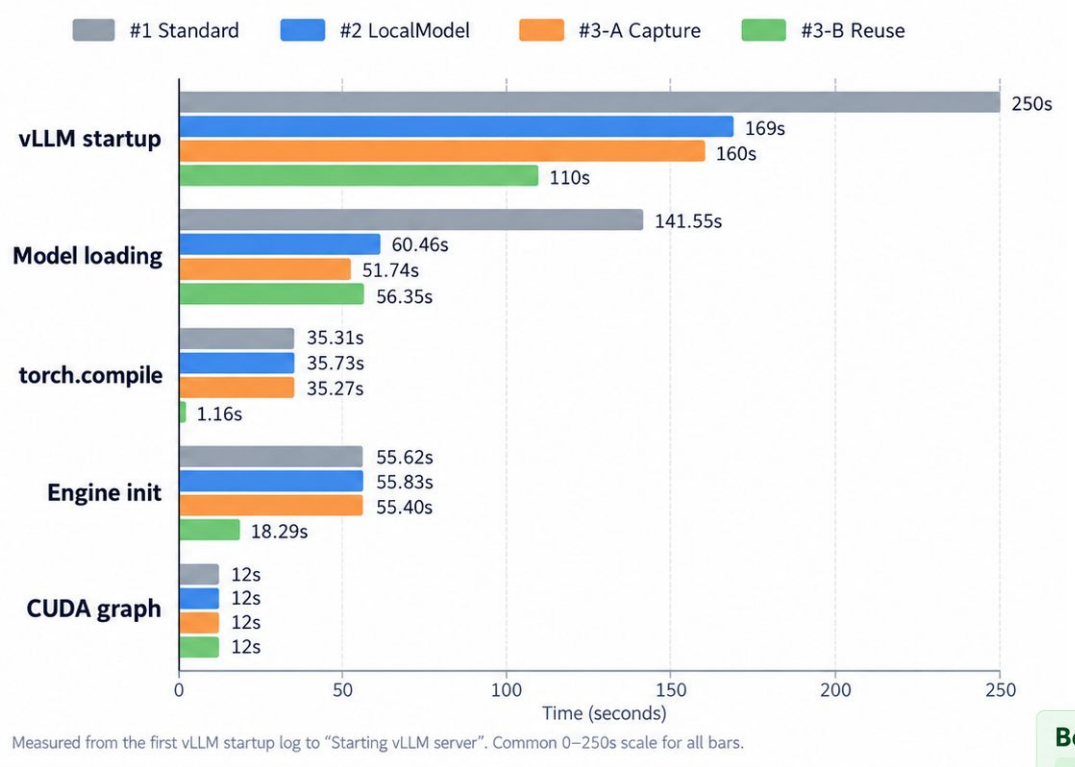
CUDA Graph capture
12s → 12s

Engine total: 55.62s → 18.29s (not additive)

Timing: first vLLM API server log → “Starting vLLM server”. Values are from separate standard and cache-reuse tests.

Outside this measured interval Runtime image pull • Model download • Init containers

Measured outcome across startup stages



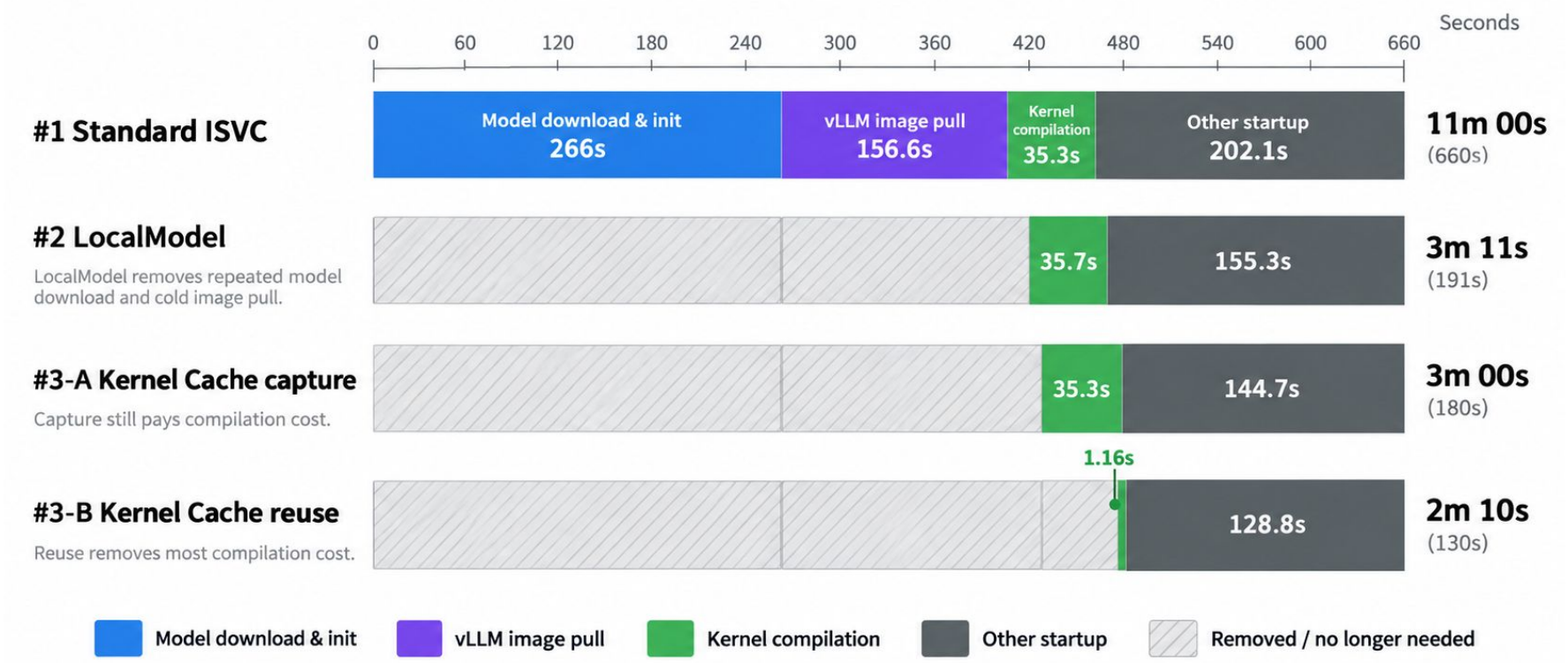
Measured from the first vLLM startup log to "Starting vLLM server". Common 0–250s scale for all bars.

Bottom line

Best observed result:
250s → 110s startup

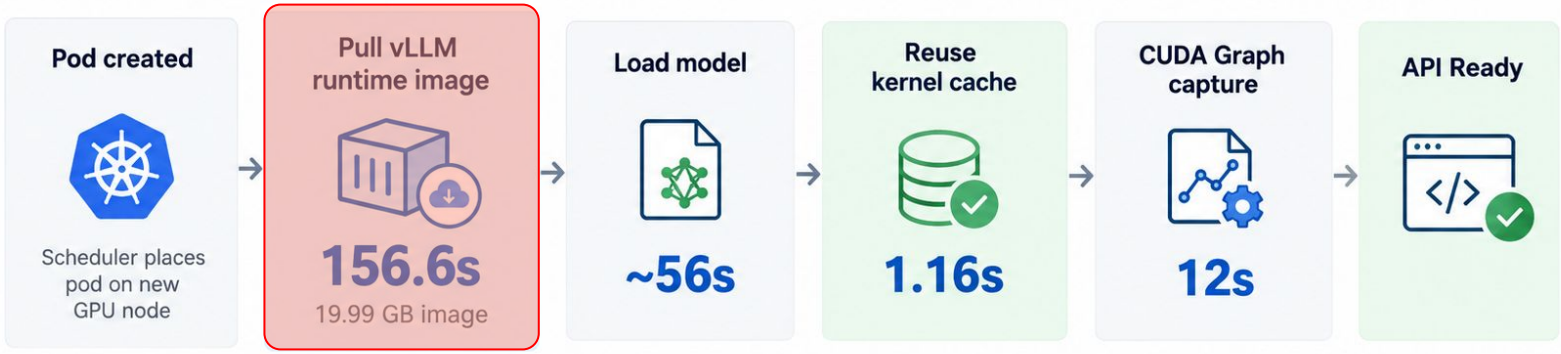
Which startup costs disappear?

Observed pod creation → Ready time across four tests



Bonus finding: another cold-start cost remains

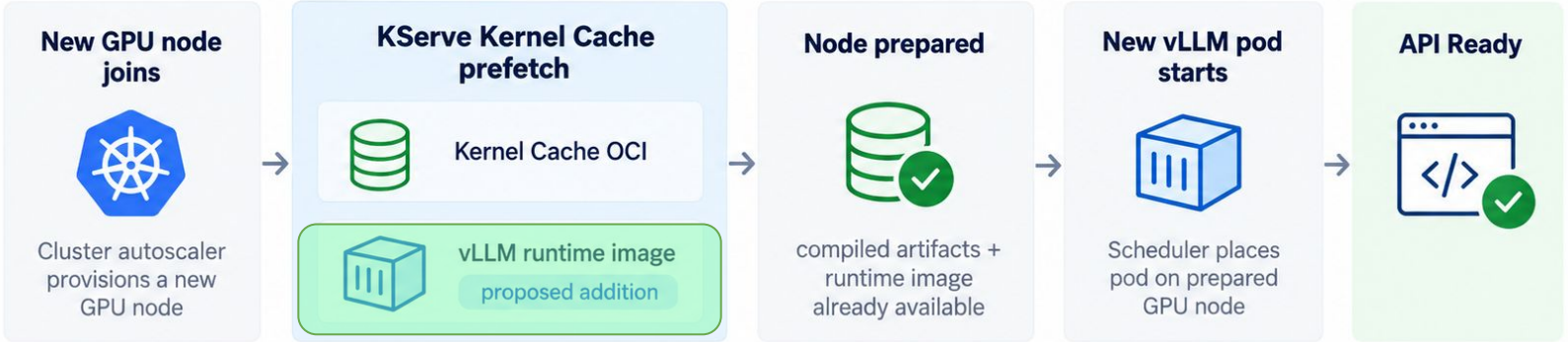
Even after KServe Kernel Cache reuse, a new GPU node still pays for the vLLM runtime image pull.



Kernel compilation is largely solved, but runtime image pull can still dominate cold starts on a new node.

Bonus idea: prefetch the runtime image too

KServe Kernel Cache already knows the runtime image. We can use that to prepare the node before the pod starts



Proposed optimization — image pull is shifted earlier to node preparation time; end-to-end benefit still to be validated.

Key takeaways

Reducing vLLM cold starts with KServe

01 Measure the entire cold start

Image pull, model preparation, compilation, and engine startup all add latency.

Pod creation → Ready

not just vLLM startup

02 Compile once, reuse safely

Package, sign, verify, and reuse compiled artifacts on compatible GPU nodes.

35.27s → 1.16s

96.7% less compile time

03 Remove startup costs layer by layer

LocalModel avoids re-downloads; runtime-image prefetch is the next opportunity.

~11m → 2m 10s

observed Pod Ready across tests

Measured in one test environment. E2E results span different storage and image-cache conditions; the full gain is not solely from Kernel Cache.

KubeCon CloudNativeCon North America 2026

#KubeCon #CloudNativeCon

Thank you. Questions?

GitHub	github.com/kserve/kserve
Meetings	Biweekly, Thursdays 12 PM EST
Slack	#kserve on CNCF Slack

https://github.com/kserve/kserve

