

A Shared KV Cache for vLLM: Reuse, Failover, and Elastic Scaling

Making the KV cache a property of the deployment, not the replica

Built on vLLM and llm-d

Mengmei Ye (mye@ibm.com)
Chen Wang (chen.wang1@ibm.com)
IBM Research

Cloud Native AI + Inference Day North America 2026

Today, every replica keeps its own cache

KV cache: the model's memory of text it has already read. Reusing it skips re-reading the prompt.

Replica 1

GPU memory
cache

CPU memory
its own overflow

Reads system prompt A
once

Replica 2

GPU memory
cache

CPU memory
its own overflow

Reads system prompt A
again, from scratch

Replica 3

GPU memory
cache

CPU memory
its own overflow

Reads system prompt B
once

The same work, twice

Replicas that receive the same long prompt each process it from scratch.

A failure loses the cache

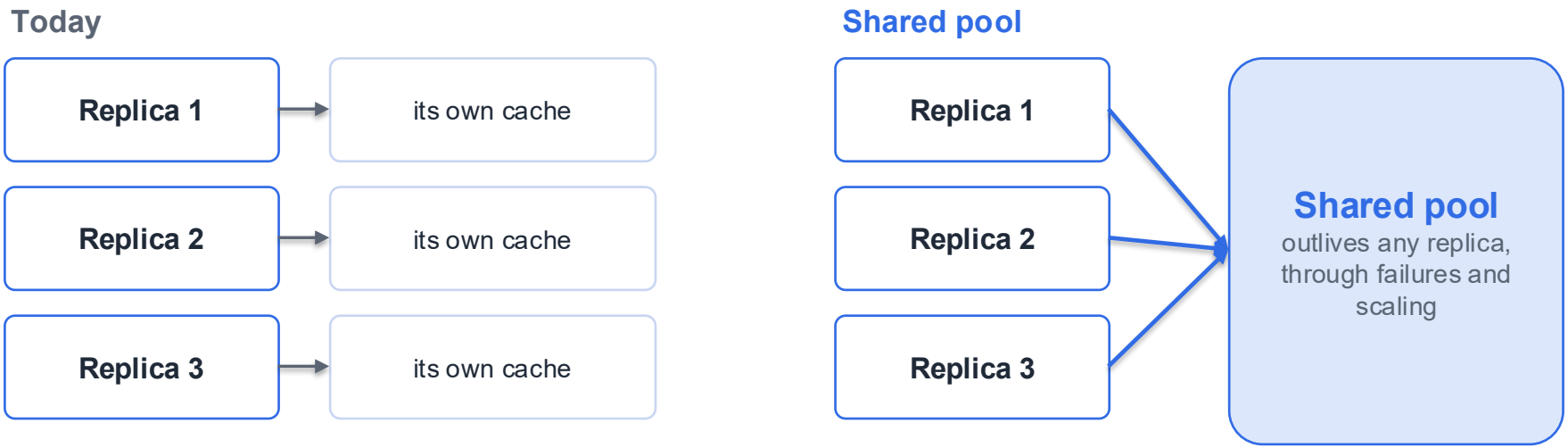
When a replica dies, everything it cached dies with it. Its traffic starts over.

New replicas start cold

Scaling down throws cached work away. Scaling up adds replicas with nothing cached.

The idea: the cache belongs to the deployment

One shared pool, outside the replicas, that every replica can read and write



- **Reuse:** a prompt one replica processed is ready for all of them
- **Failover:** a failed replica's cache stays available to its siblings
- **Elastic scaling:** adding or removing replicas keeps the fleet warm

Others share KV caches too

CLOUD NATIVE AI + INFERENCE DAY NORTH AMERICA

Shared caches exist. What is missing: an owner for the pool, and a router that knows it is shared.

System	How the cache is shared	Survives a replica failure?	What else you run	Does the router know?
LMCache	One cache server per node, shared by its pods	Documented for engine restarts	A cache server on each node	Not documented
Mooncake Store	Pooled memory and SSDs across machines	Yes, as a separate store	A metadata service and a fast network	Partly, via Dynamo
NVIDIA Dynamo	GPU, CPU, disk and object-storage tiers	Experimental	Several services and a fast network	For SGLang; vLLM in progress
SGLang HiCache	A shared storage level behind GPU and CPU	Likely, for the shared level	A storage backend	Partly; in design
AIBrix	Distributed cache across nodes	Not documented	A cache service and a fast network	Per engine
vLLM CPU offloading	Not shared: each replica's own CPU memory	No	Nothing	Per replica
vLLM shared-folder tier	Any folder all replicas can mount	Yes	Shared storage; vLLM never cleans it up	Only as the writer's cache
llm-d peer-to-peer	Pulls from another replica's CPU memory	No: that memory dies too	A fast network between pods	Yes, picks the peer
This work, built on llm-d	**One pool per node, shared by its replicas**	**Yes**	**One small service per node**	**Yes: credits every replica that can read it**

From each project's public documentation, November 2026. "Not documented" means we could not confirm it either way.

What we add: three small pieces

CLOUD NATIVE AI + INFERENCE DAY NORTH AMERICA

A working prototype built on llm-d: its router, its vLLM deployment and its benchmark tools

llm-d DEPLOYMENT, ONE PER NODE

An owner for the pool

A small service on each node creates the pool, keeps it from filling up by removing the oldest blocks, and deletes it only when no replica is using it.

vLLM

Safe joining and leaving

A replica checks that it found the right pool, stays registered while it uses it, and finishes saving its cache before it shuts down.

llm-d ROUTER

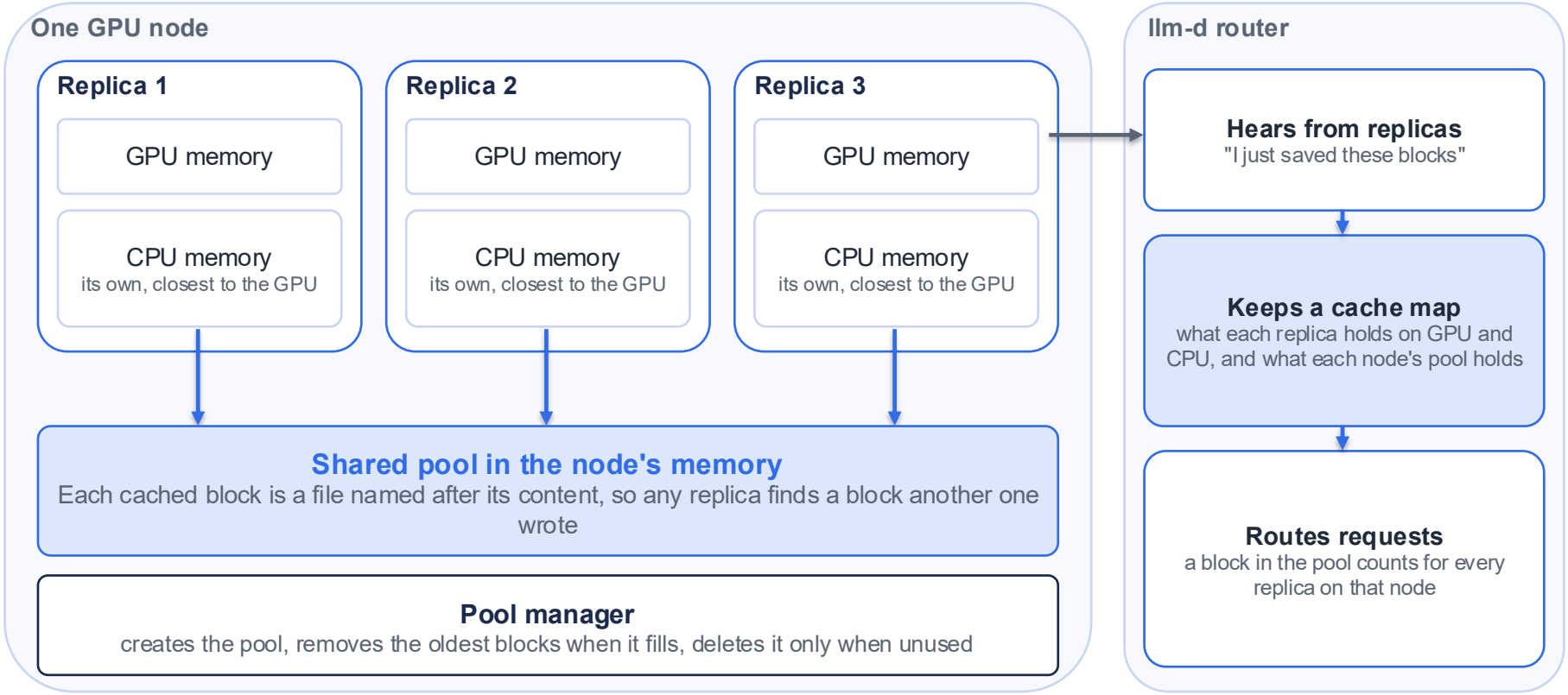
Routing that sees the pool

Blocks in a node's pool count for every replica on that node: after the replica that wrote them is gone, and for replicas that join later.

Why this design

- **No new data path:** blocks move through vLLM's own offloading, from GPU to CPU memory to the pool.
- **Nothing new on the network, and off by default:** without a pool, replicas work exactly as today.

How it fits together



The pool outlives every replica

CLOUD NATIVE AI + INFERENCE DAY NORTH AMERICA

Replicas only join and leave. The pool manager alone creates and deletes the pool.

Publish

Built completely, then made visible all at once. No replica ever sees half a pool.

Join

A replica checks it is the expected pool and stays registered while using it.

Clean up

When the pool fills, the least recently used blocks are removed first.

Retire

Only on request, only when no replica is registered, and only the exact pool named.

What happens when…	Result	On the cluster
a replica crashes	Its registration disappears automatically. Its cached blocks stay.	Blocks checked
every replica is gone	The pool and its blocks stay. The next replica starts warm.	Checked
someone deletes the pool while it is in use	Refused	Checked
a delete names an older version of the pool	Refused, so a newer pool is never removed by mistake	Checked
the model or cache format changes	A new, separate pool. Old blocks are never misread.	By design

Failover: a sibling picks up the failed replica's cache

1. Replica 1 answers conversations that share prompt G. Its cache for G flows from GPU to CPU memory to the pool. The router notes: the node's pool has G.
2. Replica 1 fails. The router forgets what was in replica 1's own memory, but keeps the note that the pool has G.
3. The next request for G goes to replica 2, which the router now sees as already having G nearby.
4. Replica 2 loads G's cache from the pool instead of reprocessing the whole prompt.

What is kept

- Every block the failed replica saved to the pool
- The router's knowledge of those blocks

What is not kept

- Requests that were in progress on the failed replica
- Blocks that never left its GPU
- The pool itself, if the whole node fails

Scaling up or down keeps the fleet warm

Adding a replica

- It joins the node's pool, which already holds its siblings' cache
- The router counts that cache for it as soon as it is ready
- Its first requests load from the pool instead of starting from zero

Removing a replica

- It finishes saving its cache, then leaves
- Its blocks stay in the pool, and the router still knows about them
- The remaining replicas keep serving those prompts warm

Where replicas run matters

- Replicas that share a pool must share a node, so we ask Kubernetes to place them together
- To survive losing a node, put the pool on storage every node can reach, at storage speed
- Works with any autoscaler: warmth lives in the pool, not in a pod

Demo: one node, two replicas, one failure

OpenShift with NVIDIA GPUs · Qwen3-8B, one GPU per replica · llm-d router · load from llm-d-benchmark

Two setups, same everything else

- **Today:** each replica caches on its own
- **Shared pool:** replicas share the node's pool, and the router knows

Traffic

Many conversations that share long system prompts, at a steady rate, for ten minutes.

Timeline of one run



Watch: how long requests wait for their first word after the failure, and how many pile up waiting.

Demo: the same failure, with and without the pool

CLOUD NATIVE AI + INFERENCE DAY NORTH AMERICA

One node, two replicas: replica 1 is killed at 0:00

-0:30

replay of two llm-d-benchmark runs, 3.5x speed

Per-replica cache (today)

- Replica 1: serving
- Replica 2: serving

1 requests waiting for a first word

0.2 s typical wait, last 30 s

Shared pool

- Replica 1: serving
- Replica 2: serving

2 requests waiting for a first word

0.3 s typical wait, last 30 s

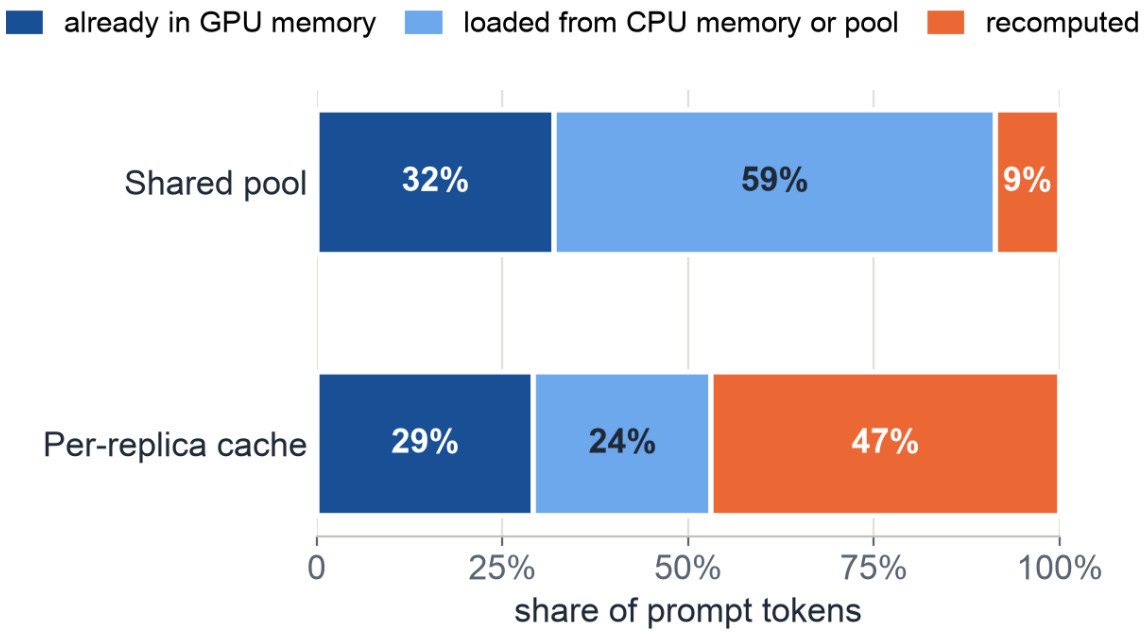




Qwen3-8B on OpenShift, one NVIDIA GPU per replica, 4 requests/s of conversations sharing 48 long system prompts. Filled dot: got its first word. Hollow dot: still waiting.

Result: the pool removes most repeated work

Where each replica got the prompt text it needed



47% → 9%

of prompt text recomputed: per-replica cache vs. shared pool

−35%

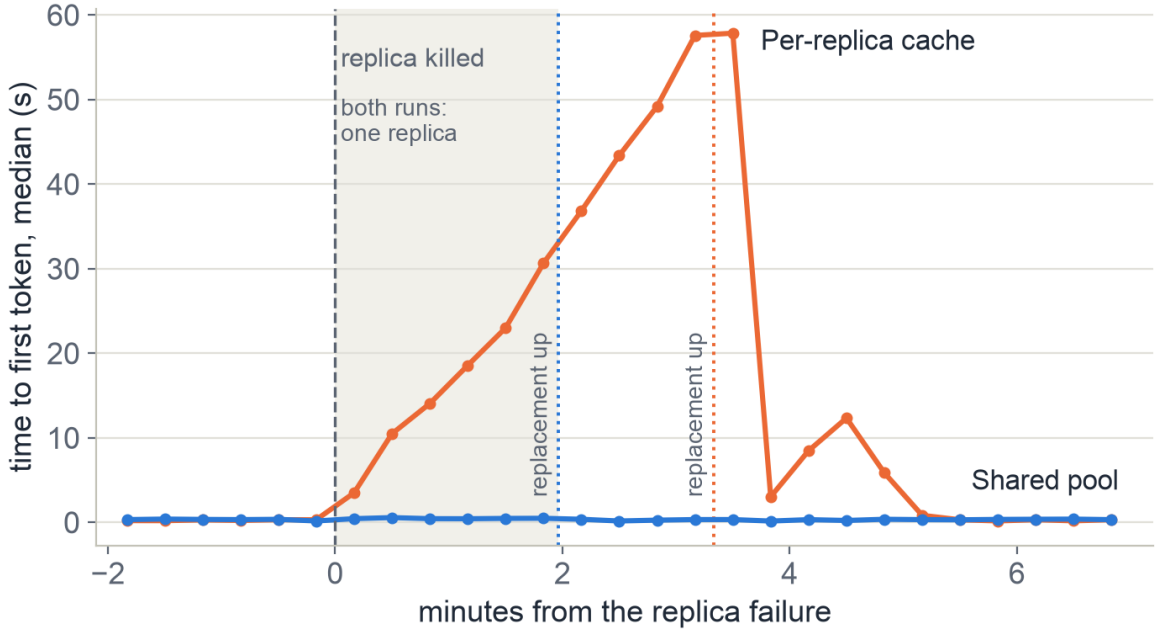
slowest 1% of first-word times (933 → 606 ms); slowest 10%: −15%

Typical first-word time did not change (304 vs. 312 ms). At this load the GPU had room to recompute; the saved work shows in the slowest requests.

llm-d with vLLM, Qwen3-8B, 2 replicas on one OpenShift node (1 NVIDIA GPU each), GPU cache capped at 64K tokens per replica, 48 shared 8K-token prompts at 4 requests/s for 10 minutes, measured with llm-d-benchmark. Each comparison: two cold-started runs, identical but for the pool.

Result: a failure no longer ripples through the fleet

A replica is killed 2 minutes into the run; the other one carries all traffic until a replacement is up



396 vs. 0

requests that waited more than 5 s for a first word in the 2 minutes with one replica (shaded): per-replica cache vs. shared pool

0.4 s

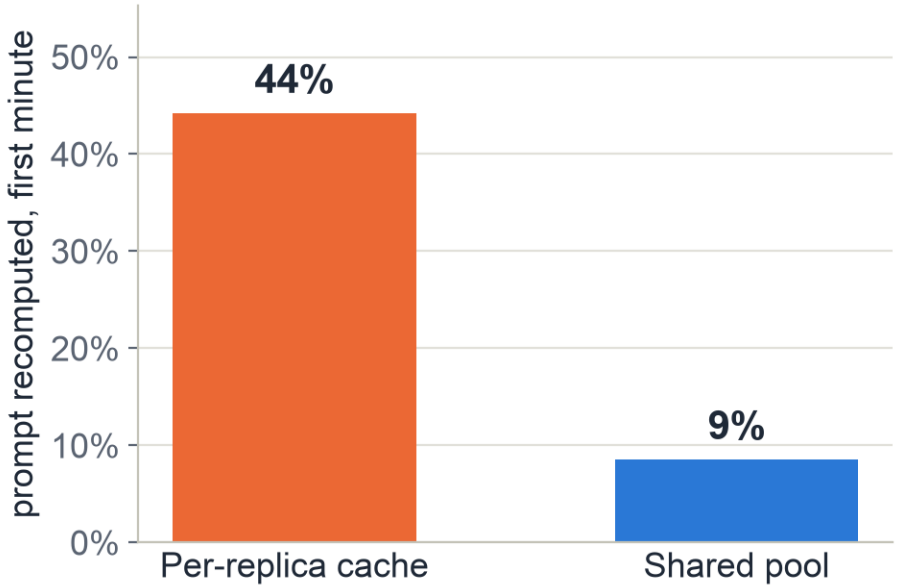
typical first-word time in those minutes with the pool, against 16 s without it

In both runs, the 5 requests in progress on the killed replica failed. Caching cannot save those; a retry can.

llm-d with vLLM, Qwen3-8B, 2 replicas on one OpenShift node (1 NVIDIA GPU each), GPU cache capped at 64K tokens per replica, 48 shared 8K-token prompts at 4 requests/s for 10 minutes, measured with llm-d-benchmark. Each comparison: two cold-started runs, identical but for the pool.

Result: a new replica starts warm

Scaling from 2 to 3 replicas during the run: what the new replica recomputed in its first minute



44% → 9%

of its prompt text recomputed in its first minute: per-replica cache vs. shared pool

79%

of the prompt text of its first 2 requests came from the pool: work its siblings did before it existed

At this load a fresh replica has a free GPU, so its first-word times were similar either way (171 vs. 159 ms). The saving is work, which matters when the scale-up is a response to overload.

llm-d with vLLM, Qwen3-8B, 2 replicas on one OpenShift node (1 NVIDIA GPU each), GPU cache capped at 64K tokens per replica, 48 shared 8K-token prompts at 4 requests/s for 10 minutes, measured with llm-d-benchmark. Each comparison: two cold-started runs, identical but for the pool.

When to share, and when not to

A shared pool helps when

- Many requests start with the same long text: agents, retrieval, long system prompts
- That shared text is too big for one replica's memory, but fits in the pool
- Replicas come and go often: failures, rollouts, autoscaling
- Replicas run side by side on the same nodes

Keep caches per replica when

- Prompts rarely repeat across replicas: the pool only adds copying
- Everything already fits in one replica's GPU memory, which is always fastest
- Replicas are spread one per node, so no siblings share a pool
- Node memory is already the bottleneck

Rule of thumb: share when the work saved by reusing prompts across replicas outweighs one extra memory copy per reuse.

Limits and what is next

Limits today

- A pool in node memory is lost if the node fails
- The same block can sit in memory twice: in the pool and in a replica's own CPU memory
- Requests in progress on a failed replica are not moved; they must be retried
- When the pool removes old blocks, the router finds out late, so some requests recompute
- One sharing scope at a time: per node or per cluster

Next

- A cluster-wide level behind the node pool, to survive node loss
- Let replicas use the pool directly as their CPU memory, so each block is kept once
- Tell the router the moment the pool removes blocks
- Retry requests cut off by a failure on a sibling
- Measure at larger scale: more nodes, longer agent sessions

Takeaways



1. **Make the KV cache outlive the replica**
   A shared pool keeps cached work through failures and scaling.

2. **Give the pool an owner**
   Create, clean up and delete it outside the model servers, and refuse unsafe deletes.

3. **Tell the router what is shared**
   Otherwise it routes as if the cache died with the replica.

4. **Share when prompts repeat across replicas**
   Keep caches per replica when they don't.

Built on vLLM and llm-d.

Thank you. Questions?