

CLOUD NATIVE AI +

INFERENCE DAY

NORTH AMERICA



Best Practices for LLM Autoscaling

from the llm-d ecosystem

Jiazhou Gao, Anthropic · **Abhishek Malvankar**, IBM Research

Cloud Native AI + Inference Day · KubeCon NA · Nov 9, 2026 · 2:05–2:30 MST

How an LLM answers a request

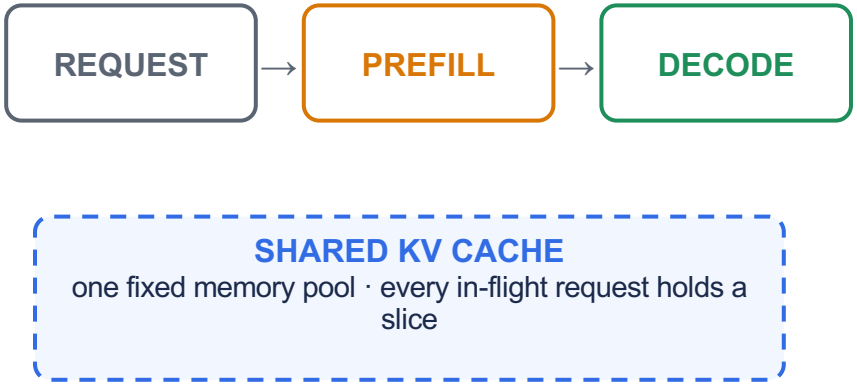
FUNDAMENTALS

Prefill

Reads the whole prompt at once · compute-heavy · short · builds the KV cache

Decode

Writes one token at a time · reads model + KV each step · memory-heavy · hundreds of steps



Serving cost varies between long and short context. Memory runs out before compute. A new replica takes minutes to become useful.

Unique problems in LLM inference autoscaling

WHY AUTOSCALING WITH TRADITIONAL METRICS FAILS

01 · SIGNAL

GPU busy ≠ capacity

99.8%

engine active while tensor cores sit at 16.6%

GPU util hits 100% at one request.

02 · TIMING

The feedback loop is slow

30–90 s

metric age, plus 15 s HPA sync, plus 2–8 min pod startup

A 30-second burst is over before capacity lands.

03 · PRESSURE

KV pressure is silent

errors = 0

while preemptions rise

Latency climbs long before the first 503.

Sources: Elastic Observability Labs (A10G, vLLM, Aug 2026); llm-d autoscaling docs.

Why plain HPA fails for LLM serving

THE NAIVE FIX

```
desired = ceil( current × metric / target )
```

ASSUMPTION 1

The metric tracks spare capacity

GPU util reads 100% at one request and at fifty. HPA sees the same ratio either way.

ASSUMPTION 2

New capacity arrives in time

A model server takes 2–8 minutes to start (image, weights, GPU load, CUDA graphs). Load stays high meanwhile, so every 15 s sync asks for more: +2 → +2 → +2 → overshoot, then flapping.

The autoscaler scales up for capacity that is already on its way.

Startup time bounds autoscaling

CLOUD NATIVE AI + INFERENCE DAY NORTH AMERICA

TIMING

ModelExpress P2P — Copy weights GPU-to-GPU, not from storage — **30.3 s → 0.93 s**

One seed pod downloads the model; new replicas pull weights from its GPU over RDMA (NIXL). Load time drops below a second, and the gain grows with pool size.

Snapshot / restore — Checkpoint a warmed engine, restore it later — **~117 s → ~5.5 s**

CRIU and cuda-checkpoint save a fully started vLLM process to local NVMe. A new pod restores it, skipping weight loading, compilation and CUDA graph capture. Still a prototype.

Measure, then tune — Find where vLLM's startup time goes — **90.7 s → 43.3 s**

Timing each startup phase found three config fixes: a writable bytecode cache, forkserver workers and fastsafetensors loading. CUDA graph capture (11.4 s) is the biggest cost left.

At minutes: long windows and idle buffers. At seconds: tight thresholds, burst reaction, scale to zero by default.

Where model weights load from

COLD START

Download at start	Each pod pulls weights from Hugging Face Simple, but each replica re-downloads tens of GB. Startup depends on network speed and hub rate limits.	**Slowest: every pod, every time**
Shared PVC	Download once, mount on every pod Removes the download, but load speed is capped by the storage backend (NFS, Ceph, cloud file). Needs ReadWriteMany.	**No repeat downloads**
hostPath / local NVMe	Pre-stage weights on each node's disk No network hop on load. You must pre-populate the nodes and schedule pods only where the weights are.	**Fastest reads**
safetensors + caches	Load faster, skip recompiling safetensors loads memory-mapped; fastsafetensors reads straight to GPU. Persist vLLM's compile cache (VLLM_CACHE_ROOT) so restarts skip recompiling.	**Skip work on restart**

Put weights as close to the GPU as you can, and cache anything computed at startup.

How llm-d solves it: KEDA + EPP

SOLUTION · THE SCALING LOOP

EPP the router → **Prometheus** scrapes EPP → **KEDA** one ScaledObject → **HPA** created by KEDA → **Model servers** pods move

EPP measures

Every request passes the router, so it sees demand for each model before any pod does.

KEDA decides

One ScaledObject turns PromQL queries on those metrics into a replica count. KEDA creates and owns the HPA.

Kubernetes acts

The HPA scales a Deployment, LeaderWorkerSet or DisaggregatedSet. No custom controller, no Prometheus Adapter.

The whole loop is EPP metrics, Prometheus and one ScaledObject.

What EPP measures

CLOUD NATIVE AI + INFERENCE DAY NORTH AMERICA

SIGNALS · ALL FROM THE ROUTER

SIGNAL	ANSWERS	EPP METRIC	WHY USE IT
In-flight requests	How busy is the pool now?	`llm_d_epp_inflight_requests`	Default. Works with the stock router
Queue depth	How many requests are waiting?	`llm_d_epp_flow_control_queue_size`	Unmet demand. Also wakes 0 → 1. Needs Flow Control
Pool saturation	How close to full is the pool?	`llm_d_epp_flow_control_pool_saturation`	max(queue/5, KV/0.8): reacts before requests queue
Token backlog	How much prefill work is waiting?	`llm_d_epp_inflight_tokens`	Counts tokens, so an 8K prompt weighs 16× a 512 one
Predicted latency	Will we miss the SLO?	`llm_d_epp_request_predicted_ttft_seconds`	Scales on the goal itself

Scale on what users wait for: requests, tokens and latency, not GPU utilization.

Four ways to scale

STRATEGIES · PICK BY WORKLOAD

	In-flight DEFAULT	Queue / saturation NEEDS FLOW CONTROL	Token-aware	SLO-aware
SCALES ON	In-flight requests	Queue depth or pool saturation, plus running requests	Token backlog for prefill, KV cache use for decode	Predicted latency ÷ SLO
BEST FOR	Getting started. Nothing beyond the stock router	Scaling on unmet demand, or before requests queue	Prompts of very different lengths. One calibration run per model and GPU	Clients with latency targets. One Prometheus recording rule
HOW IT DECIDES	**One trigger: in-flight ÷ 16 per replica**	**Two triggers. HPA uses the larger replica count**	**Two triggers, larger wins. P/D: one ScaledObject per role**	**One formula (scalingModifiers) returns the replica count, with a hold band**

Start with in-flight. Add Flow Control for queue depth, tokens when prompt sizes vary, SLO-aware for latency targets.

How HPA and KEDA make a scaling decision

CLOUD NATIVE AI + INFERENCE DAY NORTH AMERICA

TUNING · ONE PASS, EVERY 15 S

```
SCALING DECISION  ·  HPA, every 15 s
│
├─ 1 SIGNAL ──────────── what is the load?
│     ├─ target / threshold                  what one replica should handle
│     └─ metricType                          AverageValue (KEDA default) | Value
│
├─ 2 TOLERANCE ─────────── is the change big enough?
│     └─ scaleUp / scaleDown.tolerance       ±10%
│
├─ 3 STABILIZATION ──── has the change lasted?
│     ├─ scaleUp.stabilizationWindowSeconds   0 s                  → set ≈ cold start
│     └─ scaleDown.stabilizationWindowSeconds 300 s                → keep warm pods
│
├─ 4 RATE LIMIT ──────── how fast can it move?
│     ├─ scaleUp.policies                     4 pods or 100% / 15 s  → cap GPU bursts
│     ├─ scaleDown.policies                   100% / 15 s
│     └─ selectPolicy                         Max | Min | Disabled
│
└─ 5 BOUNDS ──────────── how far can it go?
      └─ minReplicaCount / maxReplicaCount    0 / 100 (KEDA)         → max = GPU quota

SCALING TO ZERO  ·  KEDA on any cluster · HPA itself on Kubernetes 1.37+ (HPAScaleToZero)
├─ activationThreshold        wake up   0 → 1, no delay
├─ cooldownPeriod 300 s       go idle   1 → 0 once triggers stay inactive this long
└─ initialCooldownPeriod 0 s  grace period after the ScaledObject is created
```

One ScaledObject, one trigger

CLOUD NATIVE AI + INFERENCE DAY NORTH AMERICA

INTEGRATION

```
kind: ScaledObject                  # KEDA
spec:
  scaleTargetRef: { name: vllm-decode }  # 1
  minReplicaCount: 1                     # 2
  maxReplicaCount: 8
  advanced:
    horizontalPodAutoscalerConfig:
      behavior:                          # 3
        scaleUp:   { stabilizationWindowSeconds: 300 }
        scaleDown: { stabilizationWindowSeconds: 300 }
  triggers:
  - type: prometheus                     # 4
    metricType: AverageValue
    metadata:
      query: sum(llm_d_epp_inflight_requests{…})
      threshold: "16"
```

1 · TARGET

The vLLM Deployment to scale. KEDA creates and owns its HPA; no custom controller.

2 · BOUNDS

Keep one warm replica so the first request skips a cold start. Set max to your GPU quota.

3 · BEHAVIOR

Scale-up window ≈ cold start, so loading pods aren't counted as new demand. Left unset: no window, and up to max(2×, 4) pods every 15 s.

4 · TRIGGER

Requests in flight across the pool, 16 per replica: 40 in flight → 3 replicas. Works with the stock router.

Need more? Add a trigger, such as queue depth with Flow Control, and HPA scales to whichever asks for more replicas.

Pick the workload to scale

SCALE TARGETS · BY MODEL SHAPE

	Deployment	LeaderWorkerSet	DisaggregatedSet
USE WHEN	**Model fits on one node**	**One model copy spans several nodes**	**Prefill and decode run in separate pods**
ONE REPLICA IS	One vLLM pod (tensor parallel inside the node)	A leader + worker group, gang-scheduled and restarted together	A slice: prefill + decode roles (each an LWS) on one revision and network domain
KEDA SCALES	The Deployment. Adds or removes pods	The LWS. Adds or removes whole groups	One role at a time: scaling.mode: External, slices: 1, then target <ds>-<role>
IN LLM-D	Optimized baseline, prefix-cache routing, every autoscaling guide	Multi-host TPU v7, tiered prefix cache multi-host, wide-EP on XPU	P/D disaggregation, wide-EP on GPU, DeepSeek V4, GLM-5.2

Fits on one node → Deployment. Spans nodes → LeaderWorkerSet. Prefill and decode split → DisaggregatedSet.

Benchmark your autoscaler

CLOUD NATIVE AI + INFERENCE DAY NORTH AMERICA

MEASURE · LLM-D-BENCHMARK

1 Deploy the guide

```
llmdbenchmark --spec guides/epp-keda-saturation standup
```

2 Run load

```
llmdbenchmark --spec guides/epp-keda-saturation run \
  -l inference-perf --monitoring
```

3 Clean up

```
llmdbenchmark --spec guides/epp-keda-saturation teardown
```

WHAT IT DEPLOYS

Qwen3-32B, EPP and a KEDA ScaledObject. No WVA, no Prometheus Adapter.

OTHER SPECS

`keda-epp-token-aware`

`keda-epp-token-aware-pd-disaggregation`

`optimized-baseline` no autoscaling

github.com/llm-d/llm-d-benchmark · **Swap --spec to benchmark another guide.**



CLOUD NATIVE AI +
INFERENCE DAY
NORTH AMERICA

Demo

Recorded on a MacBook · Kind + llm-d inference simulator · no GPU · reproducible from the repo

Best practices

TAKEAWAYS

1. **Scale on what users wait for**
   EPP requests, tokens and latency, not GPU utilization.

2. **Size the scale-up window to cold start**
   A loading pod is capacity on its way. Don't ask for more.

3. **Cut cold start first**
   Weights close to the GPU, compile caches, snapshots.

4. **Start with one trigger**
   In-flight requests, 16 per replica. Add more only when you need them.

5. **One ScaledObject per workload**
   KEDA owns the HPA. Never add a second autoscaler.

Where we need you

CONTRIBUTE

REPO + ISSUES

Good first issues in the autoscaling repo

`github.com/llm-d/llm-d-autoscaling`

#SIG-AUTOSCALING

Wednesdays, 12:30 ET, biweekly

`llm-d.ai/slack`

Try it on Kind, no GPU needed, and tell us what broke.



CLOUD NATIVE AI +
INFERENCE DAY
NORTH AMERICA

Q&A

What's breaking in your cluster?

Jiazhou Gao · Abhishek Malvankar · github.com/llm-d · llm-d.ai/slack



CLOUD NATIVE AI +
INFERENCE DAY
NORTH AMERICA

Backup

Quick start · Benchmarks · Tuning · Quota · Speculative decoding

Your first 30 minutes

CLOUD NATIVE AI + INFERENCE DAY NORTH AMERICA

QUICK START

01 **Install KEDA first**

```
helm install keda kedacore/keda -n keda-system
```

02 **Enable EPP Flow Control**

```
helm upgrade <release> … -f keda-epp/optimized-baseline/router.values.yaml
```

03 **Apply the ScaledObject**

```
kubectl kustomize keda-epp/optimized-baseline/overlays/k8s/queue | envsubst | kubectl apply -f -
```

04 **Watch the HPA**

```
kubectl get hpa keda-hpa-optimized-baseline -o jsonpath='{.status.currentMetrics}'
```

No GPU: `make create-kind-cluster` in llm-d-autoscaling runs the stack on Kind.

Reproducible benchmarks

CLOUD NATIVE AI + INFERENCE DAY NORTH AMERICA

MEASURE · LLM-D-BENCHMARK

```
llm-d-benchmark/
├── config/
│   ├── specification/guides/ one per llm-d guide
│   ├── scenarios/guides/    its settings
│   └── templates/values/defaults.yaml
├── workload/
│   ├── harnesses/  inference-perf, guidellm,
│   │               vllm-benchmark, aiperf
│   └── profiles/   sanity, chatbot, prefill-
│                   and decode-heavy shapes
├── experiments/  parameter sweeps (DoE),
│                 token-aware-autoscaling.yaml
├── llmdbenchmark/ the CLI: standup, run,
│                 experiment, teardown
└── docs/  quickstart on Kind, reproducibility
```

1 Install

```
./install.sh && source .venv/bin/activate
```

2 Preview, no cluster changes

```
llmdbenchmark --spec guides/keda-epp-token-aware \
  --dry-run standup
```

3 Stand up the guide's stack

```
llmdbenchmark --spec guides/keda-epp-token-aware standup
```

4 Run the sweep

```
llmdbenchmark --spec guides/keda-epp-token-aware run \
  --harness inference-perf --monitoring \
  --experiments experiments/token-aware-autoscaling.yaml
```

5 Tear down

```
llmdbenchmark --spec guides/keda-epp-token-aware teardown
```

Same spec, same result: config merges defaults → scenario → env → CLI, and each run keeps its rendered manifests and run_metadata.yaml. No GPU? Use --spec cicd/kind.

Scale before the queue builds

TIMING

SCALE EARLY

0.7

pool saturation threshold: scale before requests queue

WAIT FOR COLD START

300 s

scale-up stabilization window, so a loading replica isn't counted as unmet demand

WAKE FROM ZERO

0 → 1

EPP holds requests in its queue while KEDA activates the first replica

Cut cold start first: model streaming, compile cache in the image, warm-up.

Scaling and quota are separate jobs

DIVISION OF LABOR

```
request → EPP → Prometheus → KEDA → HPA → Deployment
```

AUTOSCALING · KEDA + EPP

How many replicas for this signal?

EPP Flow Control publishes queue and running-request metrics. KEDA reads them directly; no controller in between.

QUOTA · KUEUE · SLURM · LSF

Who gets how much GPU?

Borrowing, preemption and grouped quotas, with the tenants and budgets you already configured.

Keep one source of truth for compute: your quota system.

Speculative decoding: main model + drafter

HOW IT WORKS

```
drafter guesses → main model checks → keep matches
```

MAIN MODEL (TARGET)

The large model you actually serve

Checks the drafter's guesses in a single forward pass. Only tokens it agrees with are kept, so the output is exactly what the main model alone would produce.

DRAFTER (DRAFT MODEL, EAGLE, MTP)

A small, fast model that guesses ahead

Proposes the next few tokens cheaply. Every accepted guess saves a full main-model step. It is a separate module with its own weights.

Good guesses ≈ 2× faster decode. Bad guesses: same output, no speedup.

Why the drafter can't be snapshotted

SNAPSHOT LIMITATION

01

Snapshots start by dropping all weights

vllm snapshot create reuses sleep level 2: main-model and drafter weights are freed before the CRIU dump.

02

Restore reloads only the main model

On wake, reload_weights walks the main model's parameters. The drafter is a separate module, so it is skipped.

03

The drafter runs on empty memory

Its weights are never written back. It still runs, but every guess it makes is garbage.

04

Silent 2× slowdown

The main model rejects every guess: output stays correct, decode drops from 111.9 to 51.8 tok/s, and no error fires.

Tracked in vLLM issue #52479: `github.com/vllm-project/vllm/issues/52479`