



Autoscaling LLM Inference: Queue, KV Cache, Concurrency

Plan and build private AI compute with GPU Smith. We connect workload requirements with GPU hardware, networking, deployment and operating decisions so your infrastructure fits the work it must perform.

Published by GPU Smith · 2026-10-29 · autoscale vllm kubernetes keda · hpa custom metric vllm num_requests_waiting · llm autoscaling queue depth kv cache usage · dynamo planner autoscaling tft itl · gpu autoscaling kubernetes inference metrics · hpa scale to zero

Original article: <https://gpusmith.com/articles/en/llm-autoscaling-signals-queue-depth-kv-usage-concurrency>



In this article

1. [Why GPU utilization lies for LLM serving](#)
 2. [Signal candidates](#)
 3. [What vLLM exposes](#)
 4. [HPA math with custom and external metrics](#)
 5. [KEDA Prometheus trigger](#)
 6. [Stabilization, flapping and warmup](#)
 7. [Dynamo Planner](#)
 8. [Prefill and decode pools](#)
 9. [Load test to find thresholds](#)
 10. [Failure modes and guardrails](#)
-

Why GPU utilization lies for LLM serving

The default Kubernetes autoscaling signal, CPU utilization, says almost nothing about an inference pod whose work happens on a GPU. The obvious replacement, GPU utilization, is not much better. NVML defines the `gpu utilization` field as the "percent of time over the past sample period during which one or more kernels was executing on the GPU" ([NVML API reference](#)). It measures whether the device is busy, not how much more work it can absorb.

A continuous-batching engine such as vLLM launches a forward pass on every scheduler step while any request is in flight. By the NVML definition, a replica serving a handful of streams and a replica serving its maximum batch can both read close to 100%, because in both cases a kernel is almost always executing. That is our reading of the definition, and it matches Red Hat's guidance for KServe and vLLM, which notes that "a high GPU utilization might indicate efficient usage of accelerators or it can also signify reaching a saturated state" ([Red Hat Developer, 23 September 2025](#)).

The signals that move with load sit inside the serving engine. vLLM's metrics design notes say that "a common use case for our metrics is to support automated scaling of vLLM instances" ([vLLM metrics design](#)).

Signal candidates

Signal	vLLM metric	Shape	Strength	Weakness
Queue depth	<code>vllm:num_requests_waiting</code>	Unbounded; zero until saturation	Clear overload signal	Silent until the replica is full; noisy at low counts
Running requests	<code>vllm:num_requests_running</code>	Bounded by batch and KV limits	Tracks load below saturation	Flat past the batch ceiling
KV cache usage	<code>vllm:kv_cache_usage_perc</code>	Bounded 0 to 1	Sees long-context load	Bounded ratio caps each HPA step

Signal	vLLM metric	Shape	Strength	Weakness
In-flight concurrency	running + waiting	Unbounded; arrival rate × latency	Tracks load on both sides of the knee	Needs a load-tested target
Latency SLO	vllm:time_to_first_token_seconds, vllm:inter_token_latency_seconds	Histograms	What users feel	Lagging; hard to invert into replicas with HPA

The first four feed HPA or KEDA directly; latency targets suit a capacity-modelling controller such as Dynamo's Planner. Metric definitions are in our [inference metrics guide](#).

What vLLM exposes

vLLM serves Prometheus metrics on the `/metrics` endpoint of its OpenAI-compatible API server ([vLLM production metrics](#)). The current list includes the gauges above, `vllm:num_requests_waiting_by_reason`, the counter `vllm:num_preemptions` and `vllm:request_queue_time_seconds` ([vLLM production metrics](#)). The design page defines `vllm:kv_cache_usage_perc` as the "fraction of used KV cache blocks (0–1)" ([vLLM metrics design](#)).

Names change across releases, and queries copied from older posts break silently. As of 2026-10-28:

vLLM release (source, vllm/v1/metrics/loggers.py)	KV cache gauge exported
up to v0.9.1 (2025-06-10)	<code>vllm:gpu_cache_usage_perc</code>
v0.9.2 (2025-07-07) to v0.11.2 (2025-11-20)	both; <code>gpu_cache_usage_perc</code> deprecated
v0.12.0 (2025-12-03) to v0.32.0 (2026-10-18)	<code>vllm:kv_cache_usage_perc</code> only

The Red Hat KServe example from September 2025 still lists `vllm:gpu_cache_usage_perc` among its candidate metrics ([Red Hat Developer](#)). The swap-related `vllm:num_requests_swapped` and `vllm:cpu_cache_usage_perc` are described as legacy from a preemption mode that is "no longer relevant in v1" ([vLLM metrics design](#)). vLLM's policy is that metrics deprecated in version X.Y are hidden in X.Y+1 and can be re-enabled with `--show-hidden-metrics-for-version=X.Y` ([vLLM production metrics](#)).

Check the running binary before writing a query:

```
curl -s http://<pod-ip>:8000/metrics | grep -E '^vllm:
(num_requests_(running|waiting)|kv_cache_usage_perc|gpu_cache_usage_perc|num_preemptions)'
```

A KEDA query that returns no series for a renamed metric does not fail loudly: `ignoreNullValues` defaults to `true`, "which means ignoring the empty value list from Prometheus"; set it to `false` and the scaler returns an error when the target is lost ([KEDA Prometheus scaler](#)).

HPA math with custom and external metrics

The HPA controller computes ([Kubernetes HPA docs](#)):

```
desiredReplicas = ceil( currentReplicas × currentMetricValue / desiredMetricValue )
```

It skips scaling when the ratio is within a tolerance of 1.0, which is 0.1 by default. The per-HPA behavior `.scaleUp.tolerance / .scaleDown.tolerance` field is stable since Kubernetes v1.37 (first available in v1.33); without it, the cluster-wide `--horizontal-pod-autoscaler-tolerance` flag applies ([Kubernetes HPA docs](#)). The controller evaluates every 15 seconds by default (`--horizontal-pod-autoscaler-sync-period`), and with several metrics it computes a replica count for each and takes the largest ([Kubernetes HPA docs](#)).

Two details matter for GPU pods. For external and object metrics, the replica count used is the number of Running and Ready Pods ([Kubernetes HPA docs](#)), so pods still loading weights do not count. And for an external metric with an `AverageValue` target, the metric value is divided by the pod count before comparison ([Kubernetes HPA docs](#)). The query should therefore return a fleet total (`sum( . . . )`), not an average, when the target is per-pod.

Worked example 1: queue depth

Target: 2 waiting requests per Ready replica (`AverageValue: 2`), the same threshold Red Hat's example uses ([Red Hat Developer](#)). Four Ready replicas.

- Total waiting = 20. Average = $20 / 4 = 5$. Ratio = $5 / 2 = 2.5$. Desired = $\text{ceil}(4 \times 2.5) = \mathbf{10}$.
- Total waiting = 9. Average = 2.25. Ratio = 1.125, outside the 0.1 tolerance. Desired = $\text{ceil}(4.5) = \mathbf{5}$.
- Total waiting = 8.6 (a rate-smoothed query). Average = 2.15. Ratio = 1.075, inside tolerance. **No action.**

The default scale-up policy allows the larger of +100% or +4 pods per 15 seconds ([Kubernetes HPA docs](#)), so $4 \rightarrow 10$ takes two periods ($4 \rightarrow 8 \rightarrow 10$). Weight loading, not arithmetic, is the bottleneck.

Queue depth has a structural problem: it is an integral of overload. While new pods load, the queue keeps growing, the HPA keeps computing against only the Ready pods, and the desired count keeps rising. If the queue reaches 60 before any new pod is Ready, the next evaluation asks for $\text{ceil}(4 \times 60/4/2) = 30$ replicas. Without a sensible `maxReplicas`, queue depth overshoots after slow warmups.

Worked example 2: KV cache usage is a bounded ratio

Target: average KV usage 0.80 (Value type on `avg(vllm:kv_cache_usage_perc)`). Because the metric cannot exceed 1.0, the largest ratio it can produce is $1.0 / 0.80 = 1.25$. Each decision can therefore add at most 25% of current Ready replicas:

Step	Ready replicas	KV avg	Desired
1	4	1.00	$\text{ceil}(4 \times 1.25) = 5$
2	5	1.00	$\text{ceil}(5 \times 1.25) = 7$

Step	Ready replicas	KV avg	Desired
3	7	1.00	$\text{ceil}(7 \times 1.25) = 9$
4	9	1.00	$\text{ceil}(9 \times 1.25) = 12$

Each step waits for the previous pods to become Ready, so 4 to roughly 10 replicas takes three warmups. Pair KV usage with an unbounded signal; since HPA takes the maximum across metrics, KV then acts as a floor.

Worked example 3: in-flight concurrency and Little's law

In-flight requests (running + waiting) are proportional to load. Little's law, $L = \lambda W$ (Little, Operations Research, 2011), says the average number in the system equals arrival rate times average time in the system. The numbers below are hypothetical; replace them with your load test results.

- Load test: one replica meets the TTFT and ITL p99 SLO up to 48 concurrent requests (the knee). Mean request duration $W = 12$ s.
- Target set at 40 per replica (about 83% of the knee).
- Arrivals $\lambda = 10$ req/s $\rightarrow L = 10 \times 12 = 120$ in flight $\rightarrow 120 / 40 = \mathbf{3}$ replicas.
- Arrivals rise to $\lambda = 16$ req/s $\rightarrow L = 192$ (assuming W stays at 12 s) $\rightarrow$ with 3 replicas the average is 64, ratio 1.6, desired = $\text{ceil}(3 \times 1.6) = \mathbf{5}$. Check: $192 / 5 = 38.4$ per replica, under the knee.

In practice W rises near saturation, so L grows faster than λ : the signal strengthens just when it matters.

KEDA Prometheus trigger

KEDA's Prometheus scaler takes a `serverAddress`, a query that "must return a vector/scalar single element response," a threshold ("value to start scaling for," float allowed) and an optional `activationThreshold` (default 0) (KEDA Prometheus scaler, v2.21). Trigger `metricType` defaults to `AverageValue`; `pollingInterval` defaults to 30 seconds; `cooldownPeriod` defaults to 300 seconds and applies only when scaling to zero; `minReplicaCount` defaults to 0 (KEDA ScaledObject spec). Scaling between 1 and N replicas is delegated to an HPA that KEDA creates, and `advanced.horizontalPodAutoscalerConfig.behavior` is passed straight to the HPA's `behavior` field (KEDA ScaledObject spec).

A `ScaledObject` that combines the concurrency and KV signals from the examples above:

```
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: llama-vllm
spec:
  scaleTargetRef:
    name: llama-vllm
  minReplicaCount: 2
  maxReplicaCount: 12      # = GPU quota / GPUs per replica
  pollingInterval: 15
  advanced:
    horizontalPodAutoscalerConfig:
```

```

behavior:
  scaleDown:
    stabilizationWindowSeconds: 900
    policies:
      - type: Pods
        value: 1
        periodSeconds: 120
  triggers:
    - type: prometheus
      metricType: AverageValue # sum / Ready pods vs threshold
      metadata:
        serverAddress: http://prometheus.monitoring:9090
        query: |
          sum(vllm:num_requests_running{namespace="llm",model_name="llama"})
          + sum(vllm:num_requests_waiting{namespace="llm",model_name="llama"})
        threshold: "40"
        ignoreNullValues: "false"
    - type: prometheus
      metricType: Value # fleet average vs target
      metadata:
        serverAddress: http://prometheus.monitoring:9090
        query: avg(vllm:kv_cache_usage_perc{namespace="llm",model_name="llama"})
        threshold: "0.8"

```

The thresholds and the 900-second window are the illustrative values from this article, not recommendations for your hardware. For scale-to-zero, `activationThreshold` decides the $0 \leftrightarrow 1$ transition and "has more priority than the scaling value"; activation happens only when the value is strictly greater than the setting ([KEDA scaling deployments](#)). A metric served by the pods themselves reads nothing at zero replicas, so scale-from-zero needs a signal that exists without pods, such as a gateway or queue metric. Native HPA has the same rule: from Kubernetes v1.37 `minReplicas: 0` is beta and on by default, but only for object and external metrics ([Kubernetes HPA docs](#)). The cost side of that decision is in our [scale-to-zero and cold start analysis](#).

Stabilization, flapping and warmup

The HPA defaults are asymmetric. Scale-down uses a 300-second stabilization window and allows removal of 100% of replicas per 15 seconds; scale-up has no stabilization window and is applied immediately, up to +100% or +4 pods per 15 seconds, whichever is larger ([Kubernetes HPA docs](#)). Within the window the controller picks the highest recommendation, so scale-downs are gradual ([Kubernetes HPA docs](#)).

Those defaults suit pods that start in seconds. A GPU replica that takes minutes to load weights and capture CUDA graphs pays a full warmup for every premature scale-down, and a burst during warmup becomes a backlog.

Backlog arithmetic

Continue the hypothetical from example 3. Each replica's capacity at the knee is $\mu = 48 / 12 = 4$ req/s, so 3 replicas serve 12 req/s. Arrivals jump to 16 req/s, and a new replica takes 240 s to become Ready (measure yours; the [cold start analysis](#) breaks the stages down).

- Excess arrivals: $16 - 12 = 4$ req/s.
- Backlog at the moment new capacity arrives: $4 \times 240 = \mathbf{960}$ **queued requests**.

Three ways to absorb it:

1. **Headroom in the target.** A per-replica target T against a knee K absorbs a step of up to $K/T - 1$ with no queueing. At $T = 40$, $K = 48$ that is 20%. To absorb a 60% step ($10 \rightarrow 16$ req/s) for the whole warmup, $T \leq 48 / 1.6 = 30$, which means $120 / 30 = 4$ replicas at baseline instead of 3: 33% more GPUs held warm.
2. **Shorter warmup.** Local weight caches and GPU memory snapshots shrink the window.
3. **Shed or defer.** Gateway admission control, or a delay-tolerant queue for batch traffic.

Scale-down settings

Our judgment for slow-warming GPU pods: a scale-down window at least as long as warmup plus the typical gap between bursts, and a Pods policy removing one replica per period. Kubernetes gives a pod 30 seconds to terminate gracefully by default ([Kubernetes pod lifecycle](#)). A request with `max_tokens` 4096 at a 50 ms ITL needs $4096 \times 0.05 = 204.8$ s to finish, so a 30-second grace period cuts off long generations on every scale-down. Size `terminationGracePeriodSeconds` from your longest allowed generation, and set vLLM's `--shutdown-timeout`. Its default of 0 makes SIGTERM abort in-flight requests; a value above 0 rejects new requests and drains running ones for up to that many seconds (flag added in v0.18.0, 2026-03-20; behaviour read in v0.32.0 source, `vllm/v1/engine/core.py`). Keep it below the grace period.

Dynamo Planner

NVIDIA Dynamo's Planner is an SLA-aware alternative to HPA/KEDA for Dynamo deployments. Facts below are from the v1.6.1 documentation (the latest URL redirects to a dev build as of 2026-10-28) ([Dynamo Planner v1.6.1](#)):

Setting	v1.6.1 behavior
<code>optimization_target: throughput</code> (default)	"Maximizes throughput by scaling based on queue depth and KV cache utilization"
<code>optimization_target: latency</code>	Scales up at lower utilization thresholds to keep queues short
<code>optimization_target: load</code>	User-defined prefill queue token and decode KV utilization thresholds
<code>optimization_target: sla</code>	Targets <code>ttft_ms</code> (default 500.0) and <code>itl_ms</code> (default 50.0)
Load-based adjustment	Every 5 s by default (<code>load_adjustment_interval_seconds</code>)
Throughput-based adjustment	Every 180 s by default (<code>throughput_adjustment_interval_seconds</code>), <code>sla</code> only
Both enabled	Throughput-based scaling "provides a capacity floor for long-term planning"
<code>max_gpu_budget</code>	Default 8, "Maximum GPUs across all workers"
Backends	<code>sglang</code> , <code>trtllm</code> , <code>vllm</code>

The `sla` target requires kube-prometheus-stack on Kubernetes, and profiling is recommended rather than required ([Dynamo Planner v1.6.1](#)). The default `max_gpu_budget` of 8 is hit quickly by a multi-GPU model, so set it on purpose. The dual loop mirrors the headroom pattern above: a slow forecast-driven floor plus a fast reactive layer.

Prefill and decode pools

Prefill is compute-bound and decode is memory-bound; Dynamo's docs state that "a single replica count doesn't capture both" and let you set separate floors with `prefill_min_endpoint` and `decode_min_endpoint` ([Dynamo Planner, dev; v1.6.1](#)). In a disaggregated deployment, scale each pool on the signal that saturates it:

Pool	Saturating resource	Signal
Prefill	Compute, per prompt token	Queued prompt tokens (Dynamo load target); request-count queue depth as a fallback, which undercounts long prompts
Decode	KV cache memory and bandwidth	KV usage plus running requests; preemptions as the overload alarm

The ratio between the pools follows the input/output token mix, not a fixed number. Whether to disaggregate at all is a cost question covered in [prefill/decode disaggregation break-even](#).

Load test to find thresholds

Every threshold above depends on a per-replica knee measured on your stack and traffic mix:

1. Fix the replica configuration (model, `tensor_parallel_size`, `max_num_seqs`, `max_num_batched_tokens`, `gpu_memory_utilization`).
2. Replay a prompt/output length distribution sampled from production. `vllm bench serve` supports `--request-rate` (Poisson arrivals by default, `inf` sends everything at once), `--burstiness` for Gamma-distributed arrivals, and `--max-concurrency` to cap outstanding requests ([vLLM benchmark CLI](#)).
3. Step rate or concurrency upward, recording client TTFT/ITL p99 and the vLLM gauges and preemption rate.
4. The knee `K` is the highest step that still meets both p99 SLOs with zero preemptions.
5. Set the concurrency target at $K / (1 + \text{expected step size during one warmup})$, per the headroom arithmetic. Set the KV target slightly below the KV usage observed at `K`.
6. Re-run after engine upgrades.

KV capacity per replica can be estimated beforehand with our [vLLM KV cache capacity planner](#).

Failure modes and guardrails

Failure	Mechanism	Guardrail
Metric renamed or scrape lost	Query returns empty; KEDA ignoreNullValues defaults to true, ignoring the empty result (KEDA)	Set ignoreNullValues: "false"; alert on absent series; pin queries to engine version
Partial metric failure	If a metric can't be computed and the rest suggest scale-down, HPA skips scaling; scale-up can still occur (Kubernetes HPA docs)	Keep at least one robust signal from the gateway
Pods Pending for lack of GPUs	Replicas requested beyond free GPUs never become Ready	$\text{maxReplicas} \times \text{GPUs per replica} \leq \text{quota}$; namespace quota such as <code>requests.nvidia.com/gpu</code> (Kubernetes ResourceQuota)
Preemption storm	When KV space runs out, V1 preempts requests and recomputes them later (vLLM optimization)	Alert on <code>rate(vllm:num_preemptions[5m]) > 0</code> ; lower the KV target
Queue-driven overshoot	Queue grows during warmup; HPA counts only Ready pods	Bounded maxReplicas; concurrency signal instead of raw queue
Flapping	Bursty traffic shorter than warmup	Longer scale-down window; Pods scale-down policy
Cut-off generations on scale-down	30 s default grace period	Grace period $\geq$ longest allowed generation

GPU Smith works with teams sizing and operating private inference clusters where these thresholds have to be measured rather than guessed.

Source links

Links cited in this article, in order of first appearance. Full addresses are provided for readers using extracted text.

1. https://docs.nvidia.com/deploy/nvml-api/api/structnvmlUtilization__t.html
2. <https://developers.redhat.com/articles/2025/09/23/how-set-kserve-autoscaling-vllm-keda>
3. <https://docs.vllm.ai/en/latest/design/metrics>
4. <https://docs.vllm.ai/en/latest/usage/metrics>
5. <https://keda.sh/docs/latest/scalers/prometheus>
6. <https://kubernetes.io/docs/concepts/workloads/autoscaling/horizontal-pod-autoscale>
7. <https://pubsonline.informs.org/doi/10.1287/opre.1110.0940>
8. <https://keda.sh/docs/latest/reference/scaledobject-spec>
9. <https://keda.sh/docs/latest/concepts/scaling-deployments>
10. <https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle>
11. <https://docs.nvidia.com/dynamo/v1.6.1/knowledge-base/modular-components/planner/overview>
12. https://docs.nvidia.com/dynamo/latest/planner/planner_intro.html
13. <https://docs.vllm.ai/en/latest/benchmarking/cli>

14. <https://kubernetes.io/docs/concepts/policy/resource-quotas>
15. <https://docs.vllm.ai/en/latest/configuration/optimization>

THE PUBLISHER

About GPU Smith

GPU Smith provides independent engineering and research for private AI infrastructure. We help technical buyers and operators reason about GPU workload sizing, hardware selection, procurement, deployment and inference operations, from the compute system to the networking, power and cooling requirements around it.

Start with the workload

Useful infrastructure decisions begin with the models, data, throughput, latency and operating constraints a team actually has. GPU Smith helps connect those requirements with system design and deployment choices. The goal is a privately controlled AI environment whose capacity and operational demands are understood before a purchasing decision.

Hardware and supplier research

Our public reference library covers GPUs, complete systems and networking components. The server-vendor directory and manufacturing research help teams investigate suppliers, compare options and follow sources behind technical claims. We distinguish manufacturer specifications from measured benchmarks and advertised capabilities from tested configurations. Reference pages are research resources, not a live inventory listing or a binding equipment quote.

Deployment and operations

GPU Smith's engineering scope includes infrastructure integration, networking, power, cooling and the practical operation of inference workloads. Our published articles explain the tradeoffs behind deployment, procurement and ongoing operations so teams can ask better questions and document decisions.

Work with GPU Smith

Explore the [hardware library](#), [GPU references](#), [networking references](#) and [vendor research](#). [Contact GPU Smith](#) with your workloads and deployment constraints to discuss a project and confirm current scope, pricing and availability.

Inclusion of a manufacturer or product in our research does not imply a partnership, certification or endorsement.

Website: <https://gpusmith.com>

This article is educational. Verify consequential medical, legal, financial, regulatory and technical decisions with appropriate professionals and the cited primary sources. Company services are described separately from the article's research findings.