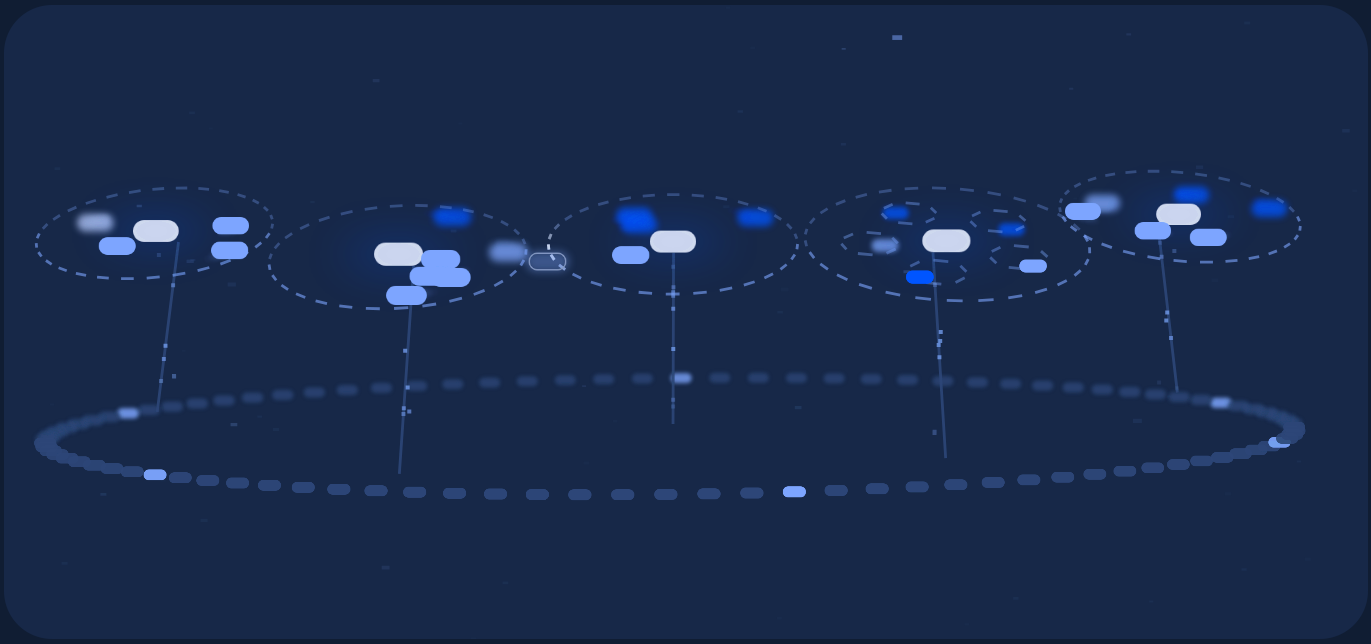
Rillor

EXPLAINER

Deploying LLMs on Kubernetes: serving, scaling and rollbacks

LLM deployment on Kubernetes, layer by layer: model servers, inference-aware routing, scaling on queues and token backlog, scale to zero and rollbacks by weight and revert.



Contents

01 Summary

02 Background

03 Method

04 Findings

05 Implications

06 Limits of this analysis

07 Sources

Summary

LLM deployment on Kubernetes runs on the platform's standard objects, and four parts need settings built for inference traffic: routing, scaling, startup and rollback.

- Kubernetes is a common home for inference. In its January 2026 survey announcement, the CNCF reported that 66% of organizations that host generative models use Kubernetes for some or all of their inference, and that 7% deploy models daily [1].
- Round-robin load balancing does not see what an LLM request costs. Replicas differ in queue length, in free memory for the key-value (KV) cache each request holds and in which prompt prefixes they already have cached, and a single request can generate tokens for over a minute [2, 4, 9, 10].
- The Gateway API Inference Extension replaces a Kubernetes Service with an InferencePool, a set of model server Pods that share one base model and accelerator type, and an endpoint picker that chooses a replica for each request from its queue and cache metrics [4]. InferencePool has been v1 since 9 September 2025 [5].
- GPU utilization is a weak scaling signal for LLM servers. llm-d, a project for serving LLMs on Kubernetes that builds on InferencePool, says in its documentation that GPU utilization can sit near 100% during active batching whether a server is lightly loaded or saturated, and it scales on queued requests, token backlog or estimated latency against a target instead [6, 7].
- Scale to zero has been on by default in the Kubernetes HorizontalPodAutoscaler since v1.37 (26 August 2026), for metrics that do not come from the Pods themselves. Kubernetes Services do not buffer requests while no Pods are ready, so an LLM endpoint at zero needs a separate layer that holds them [20, 21].
- Startup time sets the price of scaling out. In one llm-d example, a replica of an 8-billion-parameter model on one GPU took about 240 seconds to become Ready [8]. Image volumes, stable since Kubernetes v1.36, let model files travel through the same registries as container images [18, 19].
- Models too large for one node run as LeaderWorkerSet groups: a leader Pod and worker Pods that scale and roll out as one unit [17].
- Rollbacks rest on fixed versions: image digests that cannot move [26], weighted routes that shift traffic between two model versions and back [14, 24], and a desired state kept in Git with its full history [35].

Background

What changes when the service is an LLM

llm-d's documentation sets out why LLM requests differ from ordinary web traffic. Ordinary HTTP requests are fast, uniform and cheap, and standard round-robin scheduling balances them well. LLM requests are multi-turn, sending the same growing prefix again and again. They are slow, since a single request can take over a minute generating tokens [9]. They are non-uniform, ranging from thousands of reasoning tokens to more than 100,000 tokens of context [9]. The June 2025 Kubernetes blog post that introduced the Gateway API Inference Extension adds that inference sessions are often long-running, resource-intensive and partially stateful, and that a GPU-backed model server may keep several sessions active with in-memory token caches [2].

Two properties of the engines explain that state. The first is the KV cache. The 2023 PagedAttention paper describes the KV cache memory of each request as huge and as growing and shrinking dynamically, and shows that wasted cache memory limits the batch size [33]. Its vLLM engine improved throughput 2 to 4 times at the same latency against the systems it compared [33]. The second is the split between prefill and decode. Prefill, the evaluation of the prompt, is compute-bound. Decode, the generation of output tokens, is memory-bandwidth-bound [11]. The 2024 DistServe paper names the latency measure of each phase: time to first token (TTFT) for prefill, and time per output token (TPOT) for decode [32].

Adoption

The CNCF's announcement of its 2025 Annual Cloud Native Survey, published on 20 January 2026, reports that 82% of container users run Kubernetes in production, up from 66% in 2023 [1]. Among organizations that host generative models, 66% use Kubernetes to manage some or all of their inference [1]. 44% of respondents reported that they did not run model workloads on Kubernetes [1]. On frequency, 7% of organizations deploy models daily and 47% deploy occasionally [1]. The same announcement reports that 58% of the survey's "cloud native innovators" use GitOps principles extensively, against 23% of its "adopters" [1].

Who this is for

Platform engineers asked to host models on Kubernetes, and ML engineers who own model servers.

Method

What was examined. Eight parts of an LLM deployment on Kubernetes, one per finding below, each read in the documentation of the projects that define it.

Sources. Kubernetes documentation and release posts [18, 19, 20, 21, 22, 25, 26, 27, 28], the Gateway API and its inference extension [2, 3, 4, 5, 24], llm-d's documentation and release notes [6, 7, 8, 9, 10, 11, 12], KServe's documentation and release notes [13, 14, 15, 16], LeaderWorkerSet [17], KEDA [23], vLLM [29, 30], CNCF publications [1, 31, 34], OpenGitOps [35] and two papers [32, 33].

How. Release dates come from each project's release page. Features are described by what they do and the Kubernetes or project version in which they are on by default. Figures carry their source, and numbers computed or derived for this report are labelled. Models are described by size, never by family or vendor. No benchmarks were run for this report, and the tools are described without a ranking.

Findings

1. Four layers, and the work each one owns

Layer	Its job	Object or tool	Sources
Inference engine	Loads model weights, executes forward passes, keeps the KV cache	vLLM, SGLang or TensorRT-LLM in a container	[6, 33]
Pod	Holds one engine and the GPUs it requests	A Pod with a GPU limit, a model cache volume, shared memory and health probes	[27, 29]
Model server set	Replicas that share a compute configuration, accelerator type, base model and model server	An InferencePool	[4]
Gateway	Sends each request to one replica	A Gateway, an HTTPRoute and an endpoint picker	[3]

Table 1. The layers of an LLM endpoint on Kubernetes, as the sources describe them.

The engine. llm-d's architecture lists model servers that load model weights and execute forward passes, with vLLM as the default and SGLang and TensorRT-LLM as alternatives [6]. This report stays at the platform layer and does not compare engines.

The Pod. GPU scheduling has been stable since Kubernetes v1.26 [27]. Administrators install the vendor's drivers and device plugin, after which the cluster exposes a custom schedulable resource for that vendor's GPUs [27]. GPUs are specified in the limits section, and requests, if given, must equal the limits. Node labels and node selectors place Pods on nodes with a given GPU type [27]. vLLM's Kubernetes example is a Deployment that requests one GPU, a 50 GiB persistent volume used as a model cache, a memory-backed `/dev/shm` volume because the engine needs shared memory for tensor-parallel inference, and liveness and readiness probes on `/health` with an initial delay of 60 seconds [29].

The model server set and the gateway. The Gateway API Inference Extension calls the set of replicas an InferencePool and puts a gateway with an endpoint picker in front of it [3, 4]. The extension works with gateways that support Envoy's External Processing and the Gateway API, which has more than 25 implementations [3].

What Kubernetes does. Kubernetes places each Pod on a node with the GPUs it requests [27], removes a Pod whose readiness probe fails from the EndpointSlices of every Service that matches it [28], rolls out new Pod templates [25] and changes the replica count through the HorizontalPodAutoscaler (HPA) [22]. The endpoint picker joins the cluster to the engine, since it reads metrics that only the engine knows, such as cache use and queue length [4].

2. Round-robin routing does not see what an LLM request costs

The 2025 Kubernetes blog post states the gap directly: traditional load balancers focused on HTTP path or round-robin lack the capabilities that long-running, partially stateful inference sessions need [2]. Three kinds of state differ between replicas at any moment.

- **Queue.** vLLM reports the number of requests in the running and waiting states on its Prometheus endpoint [30].
- **Cache memory.** vLLM reports `vllm:kv_cache_usage_perc`, the fraction of KV cache blocks in use, from 0 to 1 [30]. Each request's cache grows and shrinks as it runs [33], so free cache memory changes from moment to moment.
- **Prefixes.** Requests in a conversation resend the same growing prefix [9]. Routing a request to a replica that already holds that prefix in cache avoids redundant prefill computation [10]. A request sent elsewhere repeats that prefill (derived from [10]).

Request counts also hide the size of each request. llm-d's documentation notes that a request count rates an 8,192-token prompt the same as a 512-token one, while the prefill work differs by 16 times [8]. Two replicas that receive the same number of requests can carry very different prefill loads (derived from [8]).

No source reviewed puts a number on the GPU time that round-robin routing loses. The evidence for routing on model server state is the comparison in Finding 3 and the results in Finding 4.

3. The inference extension replaces the Service with an InferencePool

The Gateway API Inference Extension describes itself as an official Kubernetes project that optimizes self-hosting of generative models on Kubernetes, driven by WG-Serving and SIG-Network [3]. Its API overview says that in practice an InferencePool replaces a Kubernetes Service [4].

What an InferencePool holds. Pods in an InferencePool share the same compute configuration, accelerator type, base language model and model server [4]. A pool is configured with three parts: a `selector` whose labels must exactly match the model server Pods, `targetPorts`, and `endpointPickerRef` [4]. Until release v1.5.0 the picker reference was required, and it is now optional [4]. One endpoint picker serves one InferencePool, while an HTTPRoute may reference several pools as backends [4].

How a request is routed. The Gateway and HTTPRoute select the InferencePool, and the endpoint picker tells the gateway which endpoint should receive the request [3]. The picker tracks the KV-cache utilization, the queue of waiting requests and the active LoRA adapters of each model server, and routes to a replica based on those metrics [4]. The project lists further features: routing on model names, serving priority (chat above summarization, in its example) and model rollouts by traffic splitting on model names [3].

Versions. The extension's v1.0.0 release on 9 September 2025 marked the v1 of Inference Gateway and promoted the InferencePool resource to v1 [5]. Its releases continued through v1.6.2 on 17 September 2026 [5]. The project's own endpoint picker is a lightweight one for conformance testing, and the documentation

tells production users to implement their own or use an existing one, such as llm-d's router [3].

Measured effect. The June 2025 blog post compared the extension with a standard Kubernetes Service on a vLLM deployment of 10 model replicas on 80 GB GPUs, with traffic ramped from 100 to 1,000 queries per second [2]. Throughput was roughly on par. Latency per output token at p90 was significantly lower at 500 queries per second and above [2].

Conformance. A CNCF conformance checklist for platforms that run model workloads on Kubernetes v1.37, dated 26 August 2026, requires support for the Gateway API with an implementation that enables weighted traffic splitting and header-based routing for inference services [34]. It recommends an implementation of the Gateway API Inference Extension that makes routing decisions such as KV-cache-aware routing [34].

4. Prefix-aware routing and separate prefill and decode

llm-d's router. llm-d's endpoint picker scores and selects model server Pods on real-time metrics, KV-cache affinity and configured policies, and a latency predictor uses online regression models to predict TTFT and TPOT against latency targets [6].

Prefix-cache-aware routing. llm-d routes requests to replicas that already hold the KV cache for a prompt's prefix, so the prefill is not computed again. It expects model servers that keep the cache across requests, such as vLLM's Automatic Prefix Caching [10]. It offers two implementations. The approximate one splits the prompt into fixed-size blocks, 64 tokens by default and approximated as characters, and keeps an in-memory index of recent blocks, which can diverge from the servers' real state when a Pod evicts a prefix under memory pressure [10]. The precise one has model servers emit cache events, from which an indexer keeps an exact view of which token blocks sit on which Pods, at the cost of more infrastructure [10]. llm-d's optimized baseline combines prefix-cache affinity, with an override that spreads load when endpoints get hot, and a scorer of the total prefill token load on each server [9].

Disaggregated serving. Prefill is compute-bound and decode is memory-bandwidth-bound, and running both phases on one accelerator leads to pipeline interference and head-of-line blocking [11]. Disaggregated serving runs prefill and decode on separate instances and moves KV-cache blocks between them over high-speed RDMA networks [11]. The DistServe paper found that systems which colocate and batch the two phases suffer strong interference between them and couple their resource plans. Against the systems it compared, DistServe served 7.4 times more requests, or met 12.6 times tighter latency objectives, while keeping over 90% of requests within their latency constraints [32].

llm-d v1.0.0. The release of 20 October 2026 lists router high availability with a hot standby and safe rolling updates, a DisaggregatedSet that manages prefill and decode replicas as one logical unit with coordinated rollouts and lockstep upgrades, a router configuration API at v1, and priority- and fairness-aware flow control with controlled load shedding for multi-tenant quality of service [12].

KServe's LLMInferenceService. KServe's documentation (version 0.20) describes LLMInferenceService as a Kubernetes custom resource built on the foundation of llm-d [13]. Its older InferenceService stays for predictive models. It can serve LLMs only in basic single-node deployments, without prefill and decode separation, multi-node orchestration or its scheduling features [13]. LLMInferenceService supports single-node serving, multi-node serving with LeaderWorkerSet, disaggregated prefill and decode, and data and expert parallelism. Its specification has `spec.worker` for multi-node workers, `spec.prefill` for a prefill-only workload, and a router section with the Gateway, the HTTPRoute and the scheduler [13]. KServe v0.21.0 was released on 25 September 2026, and its notes list canary deployment tests for LLMInferenceService [13].

5. Models bigger than one node run as LeaderWorkerSet groups

LeaderWorkerSet (LWS) is an API for deploying a group of Pods as a unit of replication, built for multi-host inference in which the model is sharded across devices on several nodes [17]. Its documentation lists these features [17]:

- a unique index from 0 to n-1 for each Pod in a group
- one template for the leader and one for the workers
- a scale subresource, so the HPA can scale the number of groups
- rollouts and rolling updates at the group level
- topology-aware placement, as an option
- all-or-nothing restart of a group when one Pod fails, as an option

Because the scale endpoint counts groups, one added replica of a multi-node model is a whole group of Pods and the GPUs they hold (derived from [17]). LWS also defines DisaggregatedSet, which updates two to ten roles in lockstep and keeps their capacity ratios through the update [17]. LWS v1.0.0 was released on 16 October 2026, and its notes say the release improves rollout coordination and scaling for DisaggregatedSet [17].

The higher-level tools build on it. llm-d uses LeaderWorkerSet for multi-node accelerator Pod groups with a leader-worker topology and all-or-nothing restarts, and DisaggregatedSet for prefill and decode as synchronized, versioned slices [6]. In KServe, declaring `spec.worker` in an LLMInferenceService creates a LeaderWorkerSet [13].

6. Scaling signals: queues, token backlog and latency

Why GPU utilization misleads. llm-d's documentation says that "accelerator utilization is an unreliable proxy for LLM load: GPU utilization can sit near 100% during active batching whether the server is lightly loaded or saturated" [7]. A utilization-based autoscaler therefore cannot reliably tell spare serving capacity from an overloaded batch [8]. llm-d scales on queued requests, active concurrency, token backlog or estimated latency against a target [7].

Signal	What it counts	Sources
In-flight requests	Requests being served, llm-d's default signal	[8]
Queue depth	Requests waiting for a server	[8, 30]
Pool saturation	Load as a fraction of capacity, from 0.0 to above 1.0	[8]
Token backlog	In-flight tokens and KV cache occupancy, set as seconds of prefill queue wait, so a long prompt weighs more than a short one	[8]
Estimated latency	Predicted latency against a target	[6, 8]

Table 2. Scaling signals in llm-d's documentation.

The pipeline. In llm-d's setup, metrics flow from the endpoint picker to Prometheus, through KEDA's Prometheus scaler and the External Metrics API, to an HPA that KEDA generates [8]. KEDA's Prometheus scaler reads the picker's metrics and creates and owns that HPA [7]. Multiple HPAs targeting the same Deployment conflict with one another [8].

What the HPA does with a signal. The HPA runs a control loop every 15 seconds by default [22]. It computes desired replicas as the current replicas times the ratio of the current metric value to the target, rounded up, and changes within a tolerance of 0.1 are skipped by default [22]. For example, four replicas averaging 12 waiting requests each against a target of 4 give ceil(4 × 12 / 4) = 12 desired replicas (computed for this report from the formula in [22]). Scale-downs use a stabilization window of five minutes by default, which approximates a rolling maximum [22].

What KEDA adds. KEDA feeds event-source data to the HPA. Scaling between zero and one replica is KEDA's own decision, and scaling from one to N belongs to the HPA that KEDA generates [23].

Overshoot. A replica can take minutes to become Ready, about 240 seconds in llm-d's one measured example of an 8-billion-parameter model at tensor parallelism of one [8]. To limit overshoot while replicas start, the example sets scale-up and scale-down stabilization windows of 300 seconds and advises sizing the scale-up window near the measured cold-start time of the target Deployment [8]. Two pools that draw on one contended accelerator budget can both scale up and together ask for more accelerators than exist [8]. llm-d handles shared accelerator budgets below the autoscaler, with Kueue [7].

A deprecation to note. KServe's documentation (version 0.20) describes autoscaling LLMInferenceService with the Workload Variant Autoscaler (WVA), which decides on metrics such as KV cache utilization, queue depth and request saturation [15]. llm-d's v1.0 documentation marks the Workload Variant Autoscaler as deprecated [6, 7]. This report records both and recommends neither.

Conformance. The same CNCF checklist for model workloads requires that, on a platform that supports the HPA, it works correctly for Pods that use accelerators, including scaling on custom metrics [34].

7. Scale to zero needs a buffer, and startup time sets the price

The HPA. The Kubernetes v1.37 release announcement of 26 August 2026 says that HPA scale to zero, first introduced in v1.16, is now enabled by default, and that for workloads using object or external metrics it lets HPAs scale down to zero Pods when idle [20]. The 2 September 2026 blog post on the feature adds the history: v1.36 added the ScaledToZero condition, and v1.37 turns the feature on by default in the API server and the controller manager [21]. `minReplicas: 0` requires at least one object or external metric, because CPU and memory come from running Pods and at zero there are no Pods left to measure [21]. The post ties the savings to Pods that reserve expensive resources such as dedicated GPUs, and names cold-start time as the trade-off [21].

The buffer. Kubernetes Services do not buffer requests while no Pods are ready, so HTTP and other request-driven workloads need a separate buffering layer [21]. In llm-d, the endpoint picker's flow control buffers requests while no model server endpoint is ready, which makes scale-from-zero possible. A request queued while the pool is at zero is lost if the picker restarts before a model server becomes Ready [8]. llm-d supports scale to zero through KEDA, without the HPA feature gate [7]. Its documentation advises enabling scale to zero only after scaling from one to N is validated, because model cold-start time becomes request latency and clients must allow time for the model to load [8]. KServe's documentation draws a similar line: its HPA actuator keeps at least one replica, while KEDA supports idle scale-down through `idleReplicaCount` and offers `initialCooldownPeriod` for model load time [15].

What a new replica has to do. A CNCF member post of 6 November 2026 lists the work between starting a process and serving a response: loading weights, initializing GPU resources, compiling kernels, capturing execution graphs and warming up the engine [31]. Restarts, model switches and scale-out can repeat it [31]. The post groups what can be reused instead of rebuilt.

What is reused	How	Source
Model files	Pre-staging and efficient loaders	[31]
Compiled artifacts	Persistent caches	[31]
Processed weights	Daemons and weight transfer	[31]

What is reused	How	Source
A live engine	Sleep and wake, warm instances	[31]
Initialized state	Snapshots	[31]

Table 3. Startup work that can be reused, from the CNCF member post [31].

The post reports that vLLM's weight daemon and its `vllm preload` command have been merged, and describes vLLM's snapshots of an initialized engine as experimental, limited to one GPU with tensor parallelism of one on the same machine [31]. As of 6 November 2026, the post reports that KEP-5823, the Kubernetes proposal for Pod-level checkpoint and restore, targets v1.38, with device state and restore on another node outside its first scope [31].

llm-d's documentation states the cost: warmup is paid at every scale-up, with model load and compile time between the scaling decision and the new capacity [8].

Getting model files to the Pod. Image volumes mount content from OCI registries inside containers. The feature has been stable since Kubernetes v1.36, is enabled by default and was first available in v1.31, and the container runtime must support it [18]. The v1.36 release announcement of 22 April 2026 says developers and platform engineers can package models as OCI artifacts and deliver them to Pods through the same registries and versioning workflows as container images [19]. KServe's Modelcars feature packages model files in an OCI image for the same reason: fetching large models from object storage delays startup in autoscaling, and an image avoids repeated downloads. Modelcars is not enabled by default in KServe [16].

Probes during startup. Kubernetes runs no liveness or readiness probes until the startup probe succeeds, which gives an application time to finish initializing [28]. A readiness probe suits applications that load files and warm caches before serving, and liveness probes that are configured wrongly can lead to cascading failures [28]. vLLM's documentation warns that a startup or readiness failure threshold that is too low for the server's start time gets the container killed, and advises raising the threshold after timing how long the server takes to be ready [29]. vLLM's Rust frontend has a gRPC health service that reports liveness and readiness under separate names, so a paused engine can leave the load balancer without being restarted [29]. Its readiness service reports not serving while the engine is paused, asleep or partially awake, or when a weight update has not finished successfully [29].

8. Rollouts and rollbacks: digests, weights and revisions in Git

Pin the version. Image digests are hashes of an image's content and are immutable, while tags can be moved to point to different images. Specifying an image by digest means a change at the registry cannot change the code that runs [26]. Kubernetes' documentation advises against the `:latest` tag in production,

since it makes the running version harder to track and harder to roll back [26]. vLLM's Kubernetes example uses the `latest` tag [29], so a production copy of it needs a pinned version (derived from [26, 29]).

Roll out a Deployment. A Deployment's rollout starts if and only if its Pod template changes, and scaling it does not start one [25]. By default at least 75% of the desired Pods stay up and no more than 125% exist during the update [25]. With four replicas of one GPU each, the defaults allow one Pod down and one extra Pod, and the extra Pod can start only where a fifth GPU is free (derived from [25, 27]). A rollback with `kubectl rollout undo` restores only the Pod template of an earlier revision, and `--to-revision` picks the revision [25]. A Deployment keeps 10 old revisions by default, and with a history limit of 0 it cannot roll back [25]. A model version that the Pod template does not fix, such as weights behind a mutable tag or in a separate object, does not come back with the template (derived from [25, 26]).

Shift traffic by weight. An HTTPRoute can give weights to several backends to shift traffic during rollouts, canaries or emergencies. Each backend gets traffic in proportion to its weight, and the default weight is 1 [24]. A header match can send test traffic to the new version before any user traffic [24]. The inference extension lists model rollouts by traffic splitting on model names among its features [3].

KServe's documentation gives the lifecycle for LLMInferenceService. Without a canary, runtime upgrades and model rotations are all or nothing, so a bad image or a model regression can take down the whole endpoint [14]. With one, each version declares a group and a weight under `spec.router.route`, and the controller writes weighted backends into the HTTPRoutes. Weights of 9 and 1 give 90/10 [14]. The new version starts with a small weight, waits for Ready and ramps while its metrics are watched. Promotion force-stops the old version, which frees its GPUs and keeps the resource for rollback, and decommissioning follows [14]. At any point, lowering the new version's weight or restarting the old one rolls back. A new version at weight 0 may see transient connection failures when its weight later rises, so the documentation starts it at weight 1 [14].

Weights shift traffic. They do not check the answers a model gives, so testing a new version on tasks with known answers is separate work (derived from [14, 24]).

Revert in Git. The OpenGitOps principles set four conditions: desired state expressed declaratively, stored so that it is versioned and immutable with a complete version history, pulled automatically, and continuously reconciled by software agents that observe actual state and apply the desired state [35]. Under those conditions a reverted commit is the new desired state, and the agents bring the cluster back to it (derived from [35]).

Roll back the cluster itself. The v1.37 scale-to-zero post warns that before turning the feature gate off or downgrading to a version without the condition-based implementation, `minReplicas` should be set to 1 or higher and any workload at zero scaled to at least one replica [21].

Implications

For platform teams. Put an InferencePool and an endpoint picker in front of model servers that differ in queue and cache state, since round-robin routing does not take either into account [2, 4]. Pin images by digest and keep the Deployment's revision history above zero, so that a rollback has a version to return to [25, 26]. Plan GPU headroom for surge Pods during a rollout, or set the rollout limits to match the free GPUs (derived from [25, 27]). Run one autoscaler per Deployment [8].

For ML engineers who own model servers. Expose queue, cache and latency metrics, because the router and the autoscaler both depend on them [4, 7, 30]. Measure the time a replica takes to become Ready and set the startup probe and the scale-up window from that measurement [8, 29]. Enable scale to zero only after scaling from one to N works, and give clients a timeout that covers model load [8].

For those who own the budget. Scale to zero saves more where each Pod holds GPUs, and the price is cold-start time [21]. Faster startup through pre-staged model files, image volumes and warm instances lowers that price [18, 31].

For anyone choosing tools. Read each feature's state by version and date, as in this review, and check the documentation version: llm-d's v1.0 documentation and KServe's 0.20 documentation disagree on whether the Workload Variant Autoscaler is current [7, 15].

Platform work. Rillor designs and builds Kubernetes platforms for superintelligence (SI) work on GPUs, on servers the client controls. Every change ships from Git, and a revert rolls it back.

Limits of this analysis

- This is an explainer of public documentation and papers. No benchmark was run for it, and it ranks no tool.
- No source reviewed quantifies the GPU time lost to round-robin routing. The 2025 Kubernetes blog post reports significantly lower p90 latency per output token without a percentage [2].
- No independent comparison of KServe, llm-d and a plain Deployment and Service was found.
- The only measured startup time is one llm-d example for one model size on one GPU [8].
- The DistServe and PagedAttention results come from their papers' own setups and comparisons [32, 33].
- KServe's documentation is cited at version 0.20, while its newest release is v0.21.0 [13]. llm-d's and LWS's documentation is cited at v1.0 [6, 17].
- The checkpoint and restore proposal is a v1.38 target as reported on 6 November 2026, and its scope may change [31].
- The CNCF announcement gives no sample size or fieldwork dates for its survey [1].

Sources

1. Kubernetes Established as the De Facto 'Operating System' for AI as Production Use Hits 82% in 2025 CNCF Annual Cloud Native Survey. Cloud Native Computing Foundation, 20 January 2026, viewed 7 November 2026. https://www.cncf.io/announcements/2026/01/20/kubernetes-established-as-the-de-facto-operating-system-for-ai-as-production-use-hits-82-in-2025-cncf-annual-cloud-native-survey/

2. Introducing Gateway API Inference Extension. Daneyon Hansen, Kaushik Mitra, Jiaxin Shan and Kellen Swain, Kubernetes Blog, 5 June 2025, viewed 7 November 2026. https://kubernetes.io/blog/2025/06/05/introducing-gateway-api-inference-extension/

3. Introduction, Gateway API Inference Extension documentation. Kubernetes SIGs, viewed 7 November 2026. https://gateway-api-inference-extension.sigs.k8s.io/

4. InferencePool, Gateway API Inference Extension documentation. Kubernetes SIGs, viewed 7 November 2026. https://gateway-api-inference-extension.sigs.k8s.io/api-types/inferencepool/ (API Overview: https://gateway-api-inference-extension.sigs.k8s.io/concepts/api-overview/)

5. Release v1.0.0, kubernetes-sigs/gateway-api-inference-extension. GitHub, 9 September 2025, viewed 7 November 2026. https://github.com/kubernetes-sigs/gateway-api-inference-extension/releases/tag/v1.0.0 (Release v1.6.2, 17 September 2026: https://github.com/kubernetes-sigs/gateway-api-inference-extension/releases/tag/v1.6.2)

6. Architecture, llm-d documentation v1.0. llm-d, viewed 7 November 2026. https://llm-d.ai/docs/architecture

7. Autoscaling, llm-d documentation v1.0. llm-d, viewed 7 November 2026. https://llm-d.ai/docs/architecture/autoscaling

8. KEDA with EPP Metrics, llm-d documentation v1.0. llm-d, viewed 7 November 2026. https://llm-d.ai/docs/architecture/autoscaling/keda-epp

9. Deploy an Optimized Baseline, llm-d documentation v1.0. llm-d, viewed 7 November 2026. https://llm-d.ai/docs/well-lit-paths/foundations/optimized-baseline

10. Prefix-Cache Aware Routing, llm-d documentation v1.0. llm-d, viewed 7 November 2026. https://llm-d.ai/docs/architecture/kv-management/prefix-cache-aware-routing

11. Disaggregation, llm-d documentation v1.0. llm-d, viewed 7 November 2026. https://llm-d.ai/docs/architecture/disaggregation

12. Release v1.0.0, llm-d/llm-d. GitHub, 20 October 2026, viewed 7 November 2026. https://github.com/llm-d/llm-d/releases/tag/v1.0.0

13. Understanding LLMInferenceService, KServe documentation 0.20. KServe, viewed 7 November 2026. https://kserve.github.io/website/docs/model-serving/generative-inference/llmisvc/llmisvc-overview (Release v0.21.0, kserve/kserve, GitHub, 25 September 2026: https://github.com/kserve/kserve/releases/tag/v0.21.0)

14. Canary Rollout for LLMInferenceService, KServe documentation 0.20. KServe, viewed 7 November 2026.

https://kserve.github.io/website/docs/model-serving/generative-inference/llmisvc/canary-rollout

15. Autoscaling LLMInferenceService with WVA, KServe documentation 0.20. KServe, viewed 7 November 2026. https://kserve.github.io/website/docs/model-serving/generative-inference/llmisvc/autoscaling/llmisvc-autoscaling

16. Serving Models with OCI Images (Modelcars), KServe documentation 0.20. KServe, viewed 7 November 2026. https://kserve.github.io/website/docs/model-serving/storage/providers/oci

17. Distributed Inference APIs: LeaderWorkerSet and DisaggregatedSet, LWS documentation v1.0.0. Kubernetes SIGs, viewed 7 November 2026. https://lws.sigs.k8s.io/stable/docs/overview/ (Release v1.0.0, kubernetes-sigs/lws, GitHub, 16 October 2026: https://github.com/kubernetes-sigs/lws/releases/tag/v1.0.0)

18. Use an Image Volume With a Pod. Kubernetes Documentation, last modified 3 February 2026, viewed 7 November 2026. https://kubernetes.io/docs/tasks/configure-pod-container/image-volumes/

19. Kubernetes v1.36: ハル (Haru). Kubernetes v1.36 Release Team, Kubernetes Blog, 22 April 2026, viewed 7 November 2026. https://kubernetes.io/blog/2026/04/22/kubernetes-v1-36-release/

20. Kubernetes v1.37: Garhwal. Kubernetes Blog, 26 August 2026, viewed 7 November 2026. https://kubernetes.io/blog/2026/08/26/kubernetes-v1-37-release/

21. Kubernetes v1.37: Scale Workloads to Zero with HorizontalPodAutoscaler. Johannes Würbach, Kubernetes Blog, 2 September 2026, viewed 7 November 2026. https://kubernetes.io/blog/2026/09/02/kubernetes-v1-37-hpa-scale-to-zero-beta/

22. Horizontal Pod Autoscaling. Kubernetes Documentation, last modified 3 August 2026, viewed 7 November 2026. https://kubernetes.io/docs/concepts/workloads/autoscaling/horizontal-pod-autoscale/

23. Scaling Deployments, StatefulSets & Custom Resources, KEDA documentation 2.21. KEDA, viewed 7 November 2026. https://keda.sh/docs/2.21/concepts/scaling-deployments/

24. HTTP traffic splitting, Gateway API user guide. Kubernetes SIGs, viewed 7 November 2026. https://gateway-api.sigs.k8s.io/guides/user-guides/traffic-splitting/

25. Deployments. Kubernetes Documentation, last modified 7 July 2026, viewed 7 November 2026. https://kubernetes.io/docs/concepts/workloads/controllers/deployment/

26. Images. Kubernetes Documentation, last modified 14 February 2026, viewed 7 November 2026. https://kubernetes.io/docs/concepts/containers/images/

27. Schedule GPUs. Kubernetes Documentation, last modified 20 September 2024, viewed 7 November 2026. https://kubernetes.io/docs/tasks/manage-gpus/scheduling-gpus/

28. Liveness, Readiness, and Startup Probes. Kubernetes Documentation, last modified 30 June 2026, viewed 7 November 2026. https://kubernetes.io/docs/concepts/workloads/pods/probes/

29. Using Kubernetes, vLLM documentation. vLLM project, 2 November 2026, viewed 7 November 2026. https://docs.vllm.ai/en/latest/deployment/k8s/

30. Metrics, vLLM design documentation. vLLM project, 30 October 2026, viewed 7 November 2026. https://docs.vllm.ai/en/latest/design/metrics/

31. Faster LLM inference startup: What you need to know. Baofa Fan, CNCF Blog (member post), 6 November 2026, viewed 7 November 2026. https://www.cncf.io/blog/2026/11/06/faster-llm-inference-startup-what-you-need-to-know/

32. DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving. Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu and others, OSDI 2024, arXiv:2401.09670, version 3, 6 June 2024 (first version 18 January 2024), viewed 7 November 2026. https://arxiv.org/abs/2401.09670

33. Efficient Memory Management for Large Language Model Serving with PagedAttention. Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu and others, SOSP 2023, arXiv:2309.06180, 12 September 2023, viewed 7 November 2026. https://arxiv.org/abs/2309.06180

34. Kubernetes AI Conformance Checklist, v1.37 (AIConformance-1.37.yaml). Cloud Native Computing Foundation, cncf/k8s-ai-conformance on GitHub, 26 August 2026, viewed 7 November 2026. https://github.com/cncf/k8s-ai-conformance/blob/main/docs/AIConformance-1.37.yaml

35. GitOps Principles v1.0.0. OpenGitOps, GitOps Working Group (CNCF), viewed 7 November 2026. https://opengitops.dev/

About this report

Published by Rillor on 8 November 2026. Superintelligence (SI) tools assist research and drafting, and every figure cites its source. Rillor builds and secures SI systems, supplies the data they learn from and delivers the infrastructure they run on. We apply SI to financial and compute markets.

Cite as: Rillor. Deploying LLMs on Kubernetes: serving, scaling and rollbacks. 8 November 2026. https://rillor.com/insights/deploying-llms-on-kubernetes

Notices

Rillor is not a registered investment adviser or commodity trading advisor and does not provide investment or trading advice. Research, forecasts, data and software described on this site are for research and engineering use. Nothing here is an offer or recommendation to buy or sell any security, commodity interest or digital asset. Past or simulated results do not indicate future results.

NVIDIA, AMD, Ceph, Kubernetes and related product names are trademarks of their owners and are used only to identify products. No affiliation or endorsement is implied.

Work with Rillor

Send the workloads, the GPUs and servers they run on, the teams that deploy to the cluster and who runs it afterward. Contact us at rillor.com/contact?type=kubernetes or contact@rillor.com. More on Kubernetes platforms: rillor.com/compute/kubernetes