



What's New in the Kubernetes Ecosystem

AI Inference Gateways, Kubernetes, and Gateway Comparison

Hasnain Sayyed

Talk Structure

- 1 Why Kubernetes is relevant for AI workloads
- 2 Why AI inference traffic is different from normal HTTP traffic
- 3 How regular gateways and AI Gateways differ
- 4 Why Higress was selected for the implementation
- 5 How the Kubernetes test setup was built
- 6 What the direct, Higress, and NGINX benchmark results show
- 7 What can be concluded carefully from the data

Aim of the Talk

- Explain why AI workloads create new requirements for Kubernetes gateways.
- Compare regular gateways with AI-aware gateways.
- Present the deployed system using Higress, Ollama, and TinyLlama.
- Evaluate three access paths:
 - ▶ direct Ollama,
 - ▶ NGINX Ingress,
 - ▶ Higress AI Gateway.
- Discuss the measured latency, overhead, and practical trade-offs.

Key point: The goal is to assess both gateway capabilities and measured performance in a small Kubernetes cluster.

Why Kubernetes Is Relevant for AI

- AI inference still needs normal platform functions: deployment, networking, isolation, scaling, and monitoring.
- Kubernetes provides these functions through a common operating model.
- Device plugins can expose accelerators such as GPUs to workloads.
- Autoscaling can react to resource usage or custom metrics.
- Namespaces, RBAC, quotas, and network policies support multi-tenant operation.

Key point: Kubernetes manages the environment around the model; it does not natively understand prompts, models, or tokens.

Why AI Inference Traffic Is Different

- In many APIs, one request is treated as one similar unit of work.
- In LLM inference, request cost depends on prompt length, generated output, model size, and backend load.
- A short prompt and a long prompt may both count as one HTTP request.
- Request-count limits therefore do not describe real inference cost well.
- Useful AI metrics include model name, input tokens, output tokens, latency, and backend behavior.

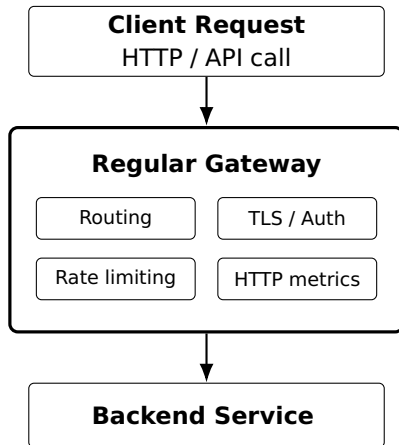
Key point: The cost of an LLM request is hidden inside the request body and generation process.

What a Regular Gateway Does

- Provides a controlled entry point into services.
- Routes traffic using host, path, headers, or weights.
- Applies TLS, authentication, and request-based limits.
- Measures HTTP latency, status codes, and availability.

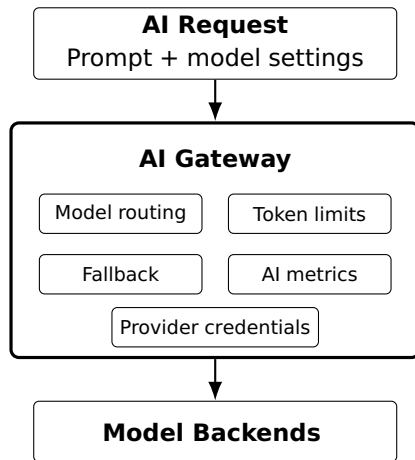
Limitation:

It sees HTTP requests, but does not automatically understand models, prompts, or token usage.



What an AI Gateway Adds

- Understands model and provider information.
- Routes traffic between model backends.
- Applies token budgets and model-access policies.
- Centralizes provider credentials and fallback rules.
- Exposes model, token, and generation metrics.



Why Token-Based Control Matters

- Request-count rate limiting treats all requests equally.
- LLM cost depends strongly on the number of tokens processed and generated.
- A user can send a few expensive requests instead of many small ones.
- Token-based limits are closer to the real cost driver of the workload.
- This helps with fairness, budget control, and operational safety in shared model-serving environments.

Key point: For LLM traffic, token budgets can be more meaningful than request budgets.

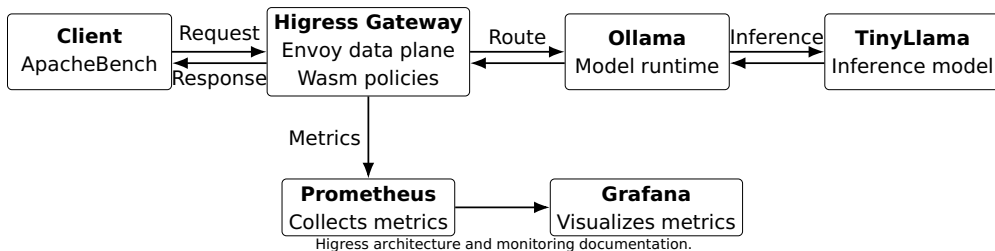
AI Gateway Options Considered

Option	Core approach	Fit for this project
Kong AI Gateway	Adds AI capabilities through plugins for model proxying, routing and rate limiting.	Mature and feature-rich, but some advanced AI plugins require an enterprise licence.
Istio + Inference Extension	Uses an endpoint picker to select suitable model backends based on current inference conditions.	Strong for large service-mesh environments, but too complex for this small two-node setup.
Envoy AI Gateway	Extends Envoy Gateway with APIs and policies designed specifically for generative-AI traffic.	Focused and open-source, but introduces another gateway stack to deploy and evaluate.
Higress	Combines an Istio/Envoy data plane with AI plugins for model proxying, fallback, policies and monitoring.	Selected because it is Kubernetes-native, open-source and practical for the available infrastructure.

Documentation: Kong AI Gateway, Istio Inference Extension, Envoy AI Gateway and Higress.

How Higress Processes an AI Request

- 1 The client sends the inference request to Higress.
- 2 Higress applies routing rules and optional Wasm-based policies.
- 3 The request is forwarded to the configured model backend.
- 4 The response returns through Higress to the client.
- 5 Gateway metrics are exposed to Prometheus and visualized in Grafana.



What Distinguishes Higress

- **Unified model access:** connects local models and external LLM providers through one gateway.
- **AI-aware traffic control:** supports token-based limits, model routing, fallback, and semantic caching.
- **Integrated observability:** exposes gateway and AI-related metrics for Prometheus and Grafana.
- **Combined gateway functions:** provides Kubernetes ingress, API management, and AI traffic control in one system.
- **Extensible design:** uses Envoy, Istio, and Wasm plugins to add policies and processing logic.
- **Project background:** originated from Alibaba's gateway technology and is now a CNCF Sandbox project.

Higress documentation and CNCF project page.

Deployed System

- **Cluster:** one Kubernetes control-plane node and one worker node.
- **Runtime/networking:** kubeadm, containerd, kubelet, kubectl, and Flannel CNI.
- **Model backend:** Ollama serving TinyLlama through `/api/generate`.
- **Gateways:** Higress as AI Gateway path, NGINX as regular gateway path.
- **Monitoring:** Prometheus and Grafana.
- **Benchmarking:** ApacheBench with JSON POST requests.

Node Setup

Node	Role	vCPU	RAM	Workload
master-NTHD	Control plane	2	4 GB	API server, scheduler, etcd, cluster management
worker-NTHD	Application node	4	8 GB	Higress, NG- INX, Ollama, TinyLlama, Prometheus, Grafana

- The worker node runs gateway, model, and monitoring workloads.
- No GPU is included in the stated configuration.

Validated Request Architecture

- 1 **Direct baseline:** ApacheBench → Ollama 11434 → TinyLlama
 - 2 **Regular gateway path:** ApacheBench → NGINX 31110 → Ollama service → TinyLlama
 - 3 **AI Gateway path:** ApacheBench → Higress 31745 → Ollama service → TinyLlama
- The endpoint path is always `/api/generate`.
 - The model backend is the same for all three paths.
 - Only the access path changes.

Benchmark Setup

- Each path uses ten POST requests: `-n 10`.
- Concurrency is fixed at one: `-c 1`.
- Socket timeout is fixed at 120 seconds: `-s 120`.
- The same JSON request body is sent to `/api/generate`.
- The request includes `"num_predict": 100` to make generated output length more comparable.
- Compared ports: direct 11434, Higress 31745, NGINX 31110.

ApacheBench Length Mismatches

- ApacheBench reported nine failed requests in each ten-request run.
- The detailed categories show zero connection failures, zero receive failures, and zero exceptions.
- All reported failures are Length mismatches.
- This means the response length differed from the first response.
- For LLM responses, variable response length is expected even with `num_predict=100`.

Key point: These are length mismatches, not demonstrated inference or network failures.

ApacheBench manual: dynamic response length is counted unless the `-l` option is used.

Benchmark Results

Metric	Direct	Higress	NGINX
Port	11434	31745	31110
Complete requests	10	10	10
Total time	305.281 s	300.206 s	319.358 s
Mean latency	30.528 s	30.021 s	31.936 s
P50 latency	29.599 s	32.000 s	33.184 s
P90 latency	42.802 s	49.445 s	44.602 s
Requests/sec	0.03	0.03	0.03
First response length	2,241 B	2,064 B	4,746 B
Connect / receive / exceptions	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0
Length mismatches	9	9	9

Interpreting the Benchmark Results

Path	Mean	Median	P90	Mean vs. Direct
Direct Ollama	30.53 s	29.60 s	42.80 s	Baseline
Higress	30.02 s	32.00 s	49.45 s	−1.7%
NGINX	31.94 s	33.18 s	44.60 s	+4.6%

- Mean latency differed by less than two seconds across all paths, indicating that gateway processing was small compared with model generation time.
- Higress had the lowest mean latency, but the highest P90 latency. It was therefore not consistently faster across all requests.
- NGINX had the highest mean and median latency, but one run with ten requests is insufficient to prove a systematic performance difference.

Key point: No substantial Higress latency penalty was observed, but repeated runs and exact token counts are required for a statistically stronger conclusion.

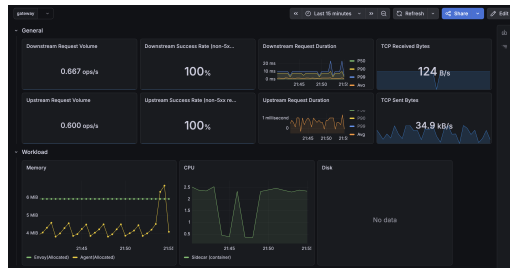
Observability with Prometheus and Grafana

What was visible in Grafana

- Gateway request volume
- Upstream and downstream rate
- Request latency: P50, P90, P99
- Gateway CPU and memory usage

Important distinction

- Higress offers integrated Prometheus/Grafana monitoring.
- NGINX metrics require separate setup.



Screenshot from the deployed Higress monitoring dashboard.

Sources: Higress monitoring docs; NGINX Ingress monitoring docs; Ollama usage API docs.

Key Takeaways

- 1 Kubernetes provides the operational base for AI inference workloads.
- 2 Regular gateways solve HTTP routing and general policy problems.
- 3 AI Gateways add model-, provider-, and token-aware control.
- 4 Higress was feasible for this two-node proof-of-concept setup.
- 5 The benchmark shows similar mean latency across direct, Higress, and NGINX paths.

Future Work

- Repeat the benchmark with more runs and higher concurrency.
- Add token metrics and Ollama generation-duration metrics.
- Compare request-count rate limiting with token-based rate limiting.
- Add multiple model backends to test routing and fallback.
- Compare Higress with Envoy AI Gateway or Istio's Inference Extension.