

Georg-August-Universität Göttingen

Institute of Computer Science

AI Inference Gateways in Kubernetes: Comparing Higress, NGINX, and Direct Model Access

Seminar Report

Author: Mohammad Hasnain Sayyed

Matrikel number: 13465202

Supervisor: Mojtaba Akbari

Date: September 25, 2026

Abstract

Large language models are increasingly served as network services inside cloud and Kubernetes environments. This creates new traffic management needs because one inference request can differ greatly from another in prompt size, generated output, model cost, and processing time. AI Gateways aim to extend normal gateway functions with model aware routing, token based control, provider management, and AI specific observability. This seminar report studies these ideas and evaluates Higress in a small Kubernetes cluster. The practical setup uses one control plane node and one worker node. Ollama serves TinyLlama as the model backend. Three request paths are compared with ApacheBench. The first path accesses Ollama directly. The second path uses NGINX as a conventional gateway. The third path uses Higress as an AI Gateway. Each path processes 1000 sequential requests at concurrency one. The measured mean latency was 8.168 seconds for direct access, 7.412 seconds for NGINX, and 7.191 seconds for Higress. All requests completed without reported failures. These values show measurable differences inside the experiment, but they do not prove that AI awareness caused the lower Higress latency. The paths were tested one after another and only one complete run was recorded for each path. The main practical finding is therefore that Higress added AI Gateway functions without showing a large latency penalty in this resource constrained setup.

I hereby declare that all uses of tools and AI as well as assistance from third parties in this work have been fully and accurately disclosed in the Transparency Report in Appendix Section A.

Missing or incorrect information may be considered an attempt to cheat.

Contents

1	Introduction	1
2	Background and Terminology	2
2.1	Kubernetes and model serving	2
2.2	Regular gateways	2
2.3	AI Gateways	2
3	AI Gateway Landscape and Higgs	3
4	System Design and Implementation	4
4.1	Cluster setup	4
4.2	Request paths	4
4.3	Monitoring	5
5	Experimental Evaluation	5
5.1	Benchmark design	5
5.2	Results	6
6	Discussion	7
7	Conclusion	8
	References	9
A	Transparency Report	10

1 Introduction

Large language models are now used through network APIs in many applications. A user sends a prompt to a model service and waits for generated text. At first sight, this looks similar to a normal web API. The traffic still uses HTTP and it still passes through common network components. However, the work behind one model request can vary strongly. A short prompt may finish quickly, while a longer prompt with more generated tokens may need more compute time and more memory. Research on large language model serving also shows that inference performance depends on request length, scheduling, batching, and memory management [1, 2].

Kubernetes is widely used to run container based applications. It provides deployment, service discovery, resource control, access control, and scaling. It can also expose hardware devices such as GPUs through device plugins [3]. Horizontal Pod Autoscaling can react to resource usage or other metrics when a workload needs more capacity [4]. These features make Kubernetes a useful platform around AI inference. Still, Kubernetes itself does not interpret prompts or token usage. The platform knows that an HTTP request reached a service, but it does not automatically know whether the request asked for one model or another, or whether it consumed a small or large token budget.

This gap has encouraged work on AI Gateways. The Kubernetes community describes an AI Gateway as gateway infrastructure with extra capabilities for AI workloads [5]. The goal is not to create a completely separate networking layer. Instead, the gateway extends familiar traffic management with model aware routing, AI specific policies, and richer observability. The Gateway API Inference Extension follows a similar direction for self hosted inference and adds concepts for model aware backend selection [6].

This report studies the difference between a conventional gateway and an AI aware gateway in a small practical environment. The central research question is:

Does an AI aware API gateway, Higress, provide measurably different request latency compared with a conventional gateway, NGINX, and direct inference access when serving a large language model on a resource constrained Kubernetes cluster?

This report focuses on three main aspects. First, it explains why inference traffic creates different gateway requirements from normal HTTP traffic. Second, it compares several current AI Gateway approaches and describes the design of Higress. Third, it reports a practical benchmark with 1000 requests for direct Ollama access, NGINX, and Higress on the same TinyLlama backend.

The rest of the report is structured as follows. Section 2 introduces the main background terms. Section 3 discusses current AI Gateway approaches and the role of Higress. Section 4 presents the Kubernetes implementation. Section 5 describes the experiment and the measured results. Section 6 discusses what the results mean and where the experiment is limited. Section 7 concludes the report.

2 Background and Terminology

2.1 Kubernetes and model serving

Kubernetes manages applications as a set of declarative resources. Pods run containers. Services provide stable network access to groups of pods. Ingress controllers or Gateway API implementations can then expose selected services to clients. This model is useful for AI systems because the model server can be managed like another application while still receiving platform features such as scheduling, isolation, and monitoring.

AI inference adds resource pressure that is less predictable than many simple web services. A model request has a prompt processing phase and a token generation phase. The time needed for these phases depends on input size, output size, model design, and available hardware [2]. Serving systems therefore try to reduce memory waste and improve scheduling because poor resource use can reduce throughput or increase latency [1]. These properties explain why backend state matters when routing inference traffic.

The model runtime in this project is Ollama. Ollama exposes a generate API that accepts model input and generation options [7]. TinyLlama is used as the model because it is a compact 1.1 billion parameter language model that can run with much lower hardware requirements than larger models [8]. This was important because the worker node in the experiment had four virtual CPUs and 16 GB of RAM, with no GPU.

2.2 Regular gateways

A conventional gateway acts as a controlled entry point to backend services. It can route requests by host, path, header, or other HTTP information. It can also apply TLS, authentication, and request based rate limits. NGINX Ingress is a common example in Kubernetes. It can expose request duration, response duration, request counts, process information, and other metrics when Prometheus monitoring is enabled [9].

These functions are useful for AI APIs as well. An AI request still needs a secure network path and a backend destination. A conventional gateway can forward AI traffic without difficulty, but it normally works with general HTTP information. Model names, provider formats, prompts, and token usage only become available for policy decisions when additional AI specific logic is added.

2.3 AI Gateways

An AI Gateway extends the normal gateway role with functions that are designed for model traffic. Current projects commonly provide unified access to several model providers, model aware routing, AI specific authentication, token based limits, fallback, and AI focused metrics [10, 11, 12]. The Kubernetes AI Gateway Working Group also frames AI Gateways as normal gateway infrastructure with added policy support for AI workloads [5].

Token based control is an important example. Request based limits treat every HTTP request as an equal unit. This can be misleading for language models because two requests can consume very different numbers of tokens. Higress therefore supports policies that count token use and apply token quotas or rate limits [13]. Such controls are closer to the

resource and cost model of LLM services.

AI specific routing is another difference. The Gateway API Inference Extension introduces model focused routing concepts and can consider information about inference backends instead of using only normal path based routing [6]. This becomes more useful when several model replicas or providers are available. The experiment in this report has only one TinyLlama backend, so advanced model selection is discussed as a capability rather than measured experimentally.

3 AI Gateway Landscape and Higress

Several gateway projects now provide AI focused functions. The landscape is still developing and the products do not all follow the same design.

Kong adds AI functions through plugins that can proxy and route model requests [11]. This keeps the AI functionality within an established API gateway, although the broader plugin based setup was more than this small experiment required.

Istio approaches the problem from a service mesh perspective. Together with the Gateway API Inference Extension, it can use inference aware endpoint selection for backend routing [14]. This is particularly relevant when Istio is already part of the platform, while the environment in this project only required a gateway comparison on a small cluster.

Envoy AI Gateway follows a more focused approach by extending Envoy Gateway with APIs and policies for generative AI traffic [12]. It provides another open source option for handling AI traffic, while Higress was selected for the practical implementation in this project.

Higress was selected for the implementation in this report because it combines Kubernetes gateway functions with AI focused features in one project. Higress provides unified model access, model routing, fallback, semantic caching, token level rate limiting, authentication, and observability [10]. Its AI statistics plugin can expose metrics such as input tokens, output tokens, total request time, and first token time when the required AI proxy integration is present [15]. This feature set made Higress suitable for studying an AI aware gateway without replacing the Kubernetes environment with a larger service mesh design.

In the deployed path, a client sends a request to Higress. The Envoy based data plane receives the traffic and applies routing and optional Wasm based policies. Higress then forwards the request to the configured model backend. In this project, the backend is Ollama and the model is TinyLlama. The response returns through the same gateway path.

Higress also provided the monitoring path used in the experiment. Prometheus collected gateway metrics and Grafana displayed them. NGINX can expose Prometheus metrics too, but the controller must be configured to export them and Prometheus must be configured to collect them [9]. The comparison in this report therefore does not claim that monitoring is exclusive to Higress. The practical difference is that the Higress deployment already provided the monitoring view used during the project.

4 System Design and Implementation

4.1 Cluster setup

The test environment used two virtual machines. The control plane node had two virtual CPUs, 4 GB of RAM, and a 40 GB disk. The worker node used the `m2.large` flavor with four virtual CPUs, 16 GB of RAM, and an 80 GB disk. The control plane ran the Kubernetes API server, scheduler, etcd, and cluster management components. The worker node ran the application workloads.

The cluster was created with `kubeadm`. `Containerd` ran the application containers. `Flannel` provided the pod network so that pods on the cluster could communicate through one common network. The worker node hosted Higress, NGINX, Ollama, TinyLlama, Prometheus, and Grafana.

TinyLlama was selected because its small size made local inference possible on the available worker node [8]. No GPU was available. This makes the setup intentionally resource constrained and also means that model generation time is expected to be a large part of end to end latency.

4.2 Request paths

The experiment kept the model backend fixed and changed only the access path. The three paths are shown in Figure 1.

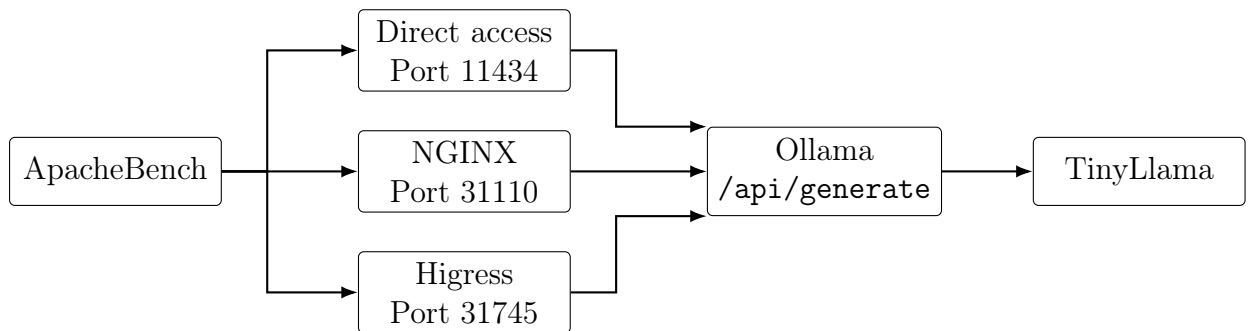


Figure 1: Validated request paths used in the experiment. All paths reached the same Ollama and TinyLlama backend.

The direct baseline connected ApacheBench to Ollama on port 11434. The conventional gateway path connected ApacheBench to NGINX on port 31110. NGINX then forwarded the request to Ollama. The AI Gateway path connected ApacheBench to Higress on port 31745, and Higress forwarded the same request to Ollama. Each path used the endpoint `/api/generate`.

The long benchmark was started inside a `tmux` session. This allowed the command to continue even if the interactive terminal was disconnected. `tmux` was only used to keep the process running and did not change the network path.

4.3 Monitoring

Prometheus and Grafana were used to inspect the Higress gateway. The dashboard in Figure 2 shows request volume, success rate, request duration percentiles, TCP traffic, memory use, and CPU use. The disk panel had no data and is not used for any conclusion.

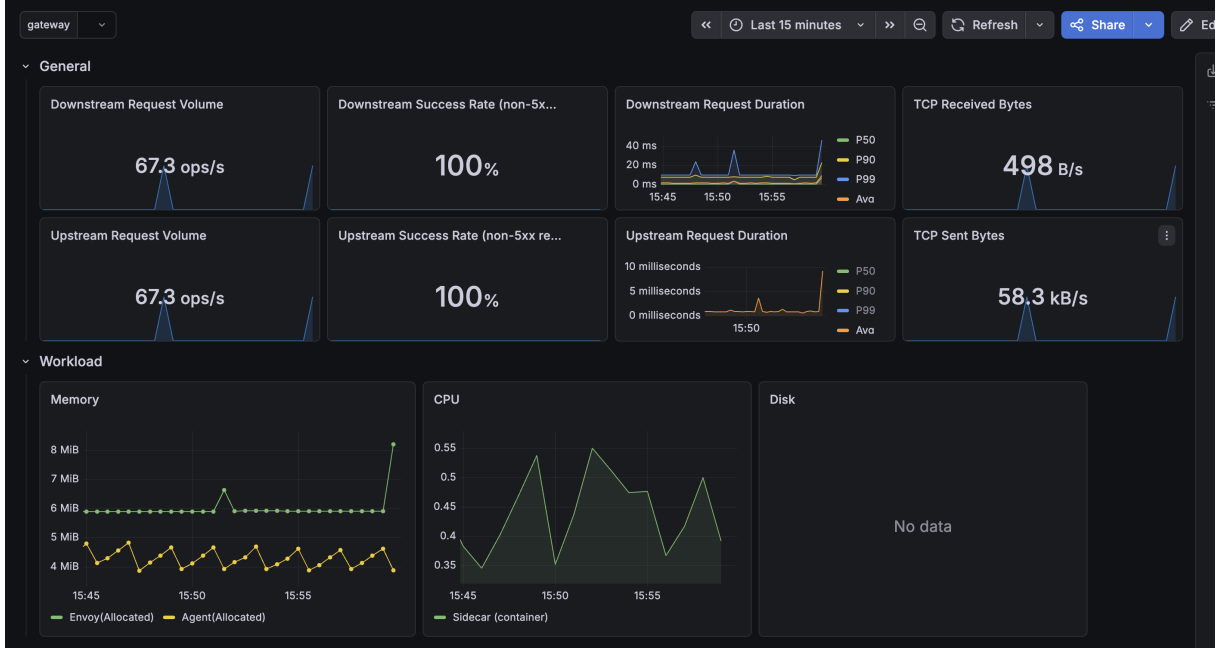


Figure 2: Grafana dashboard from the deployed Higress monitoring environment.

The dashboard showed 100 percent non 5xx success rate during the displayed interval. It also showed downstream and upstream request volume and request duration values. These gateway metrics are useful because end to end client timing alone cannot show where time is spent.

5 Experimental Evaluation

5.1 Benchmark design

The benchmark compared direct Ollama access, NGINX, and Higress. The Higress fallback test with the Phi3 model was excluded because it used a different model and was not part of the research question.

Each measured path processed 1000 requests with ApacheBench. Concurrency was fixed at one, so only one request was active at a time. ApacheBench therefore measured a long sequence of individual inference requests rather than a high concurrency load test. ApacheBench is designed to report request time, throughput, connection timing, and percentile information [16].

The same request body was used for the three paths. The generation request included `num_predict = 100` to limit the maximum generated output and reduce variation between requests. Ollama documents generation options through its generate API [7]. The

total request body sent by ApacheBench was 335000 bytes for every path, which supports that the input workload size was the same.

The paths were executed in a fixed order. Direct Ollama ran first, NGINX ran second, and Higress ran third. The benchmark log records a 30 second cooling interval before NGINX and before Higress. It also records successful HTTP 200 warm up responses for NGINX and Higress. This fixed order is important when interpreting the results because later runs may benefit from a warmer model runtime, operating system cache, or other changing system state.

5.2 Results

ApacheBench completed all 1000 requests for each path without recording failures. Since the benchmark used a concurrency of one, this result should only be understood as an observation from the measured runs and not as evidence of reliability under heavier load. The response document length was marked as variable for every run. The main focus of the experiment was therefore the latency comparison shown in Table 1.

Table 1: ApacheBench results for 1000 sequential requests per path.

Metric	Direct	NGINX	Higress
Mean latency	8168 ms	7412 ms	7191 ms
Median latency	8084 ms	7407 ms	7196 ms
P90 latency	8813 ms	7807 ms	7698 ms
P95 latency	9004 ms	7981 ms	7800 ms
P99 latency	9204 ms	8198 ms	7995 ms
Maximum latency	9701 ms	8491 ms	8200 ms
Standard deviation	472 ms	346 ms	364 ms
Requests per second	0.12	0.13	0.14
Failed requests	0	0	0

The direct path had a mean latency of 8168 ms. The NGINX run had an observed mean latency of 7412 ms. Higress had the lowest observed mean at 7191 ms. Relative to the direct run, the NGINX mean was about 9.3 percent lower and the Higress mean was about 12.0 percent lower. Higress was about 3.0 percent lower than NGINX in the measured mean.

The tail latency values followed the same broad order. P90 was 8813 ms for direct access, 7807 ms for NGINX, and 7698 ms for Higress. The maximum recorded request was 9701 ms for direct access, 8491 ms for NGINX, and 8200 ms for Higress.

The amount of generated response data was also very close across the runs. ApacheBench reported 1379679 bytes of HTML transferred for direct access, 1379248 bytes for NGINX, and 1379124 bytes for Higress. The difference between the largest and smallest value was less than one tenth of one percent. This supports that the overall response volume was similar. It does not prove that every request generated exactly the same number of tokens.

6 Discussion

The research question asks whether Higress provides measurably different request latency from NGINX and direct access. The experiment produced different measured values. Higress had the lowest mean latency and the lowest P90 latency in the recorded runs. NGINX was second and direct Ollama access was slowest.

These measurements should not be interpreted as proof that the gateway made inference faster. A gateway adds another processing step, so a causal speed improvement would need stronger evidence. The three paths were not tested in random order and the experiment recorded only one long run for each path. The direct path always ran first. NGINX ran after it, and Higress ran last. The later paths may have benefited from model warm state, operating system caches, CPU behavior, or lower background load. The 1000 requests provide many observations inside each run, but they do not replace independent repeated runs with different orders.

The result is still useful. Higress did not show a large latency penalty in this setup. Its measured mean was close to NGINX and lower than the direct run. These results are consistent with gateway processing being small compared with model generation time on this CPU based worker. However, the experiment does not isolate gateway processing time directly. The result is consistent with the observation that LLM serving time is strongly shaped by model execution and scheduling [1, 2].

The experiment also shows why performance alone is not the full product case for an AI Gateway. NGINX can route the same inference request successfully and can expose Prometheus metrics when configured [9]. Higress adds AI focused functions such as token management, model access rules, fallback, semantic caching, and AI statistics [10, 13, 15]. These functions matter most when several users, models, or providers share the same gateway.

Several limitations remain. Concurrency was one, so the test did not create queue pressure from simultaneous clients. Only one model backend was used, so model aware load balancing and fallback were not evaluated. The hardware had no GPU, which limits how well the results represent production model serving. Exact token counts were not collected for every request. The `num_predict` setting limited output length, but it did not force identical responses. Finally, the experiment did not repeat each path several times with randomized order.

Future work should address these limits. Each path should be repeated in several independent runs, and the run order should change between repetitions. Concurrency should increase to show how the gateways behave under shared load. Exact input token counts, output token counts, and model generation duration should be collected. A second model backend would allow a real test of AI aware routing and fallback. A further comparison with Envoy AI Gateway or the Istio Inference Extension would also help separate Higress specific behavior from general AI Gateway behavior.

7 Conclusion

This seminar report compared three ways to access the same TinyLlama model in a small Kubernetes cluster. The direct path used Ollama without a gateway. The conventional path used NGINX. The AI aware path used Higress.

The 1000 request benchmark found a mean latency of 8.168 seconds for direct access, 7.412 seconds for NGINX, and 7.191 seconds for Higress. Higress also had the lowest P90 value in the recorded runs. ApacheBench did not record failures during these runs, but since concurrency was limited to one, this should not be interpreted as evidence of reliability under higher load.

The research question can therefore be answered in a limited way. Higress produced measurably different latency in this experiment and did not show a clear latency penalty compared with NGINX or direct access. However, the experiment does not show that AI awareness caused the difference. The fixed run order and lack of independent repeated runs prevent a strong causal conclusion.

The wider value of an AI Gateway is found in traffic governance rather than raw latency alone. AI Gateways add model and token aware controls to normal gateway functions. In a larger environment with several models, providers, and users, these functions can support fair resource use, central policy, and richer monitoring. The current experiment shows that such a gateway can be deployed on limited Kubernetes infrastructure without an observed latency penalty, while also showing why stronger experiments are needed before making general performance claims.

References

- [1] Woosuk Kwon et al. “Efficient Memory Management for Large Language Model Serving with PagedAttention”. In: *Proceedings of the 29th Symposium on Operating Systems Principles*. 2023, pp. 611–626. DOI: 10.1145/3600006.3613165.
- [2] Amey Agrawal et al. “Taming Throughput Latency Tradeoff in LLM Inference with Sarathi Serve”. In: *18th USENIX Symposium on Operating Systems Design and Implementation*. 2024. URL: <https://arxiv.org/abs/2403.02310>.
- [3] Kubernetes. *Device Plugins*. URL: <https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/device-plugins/>.
- [4] Kubernetes. *Horizontal Pod Autoscaling*. URL: <https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/>.
- [5] Kubernetes. *Announcing the AI Gateway Working Group*. 2026. URL: <https://kubernetes.io/blog/2026/03/09/announcing-ai-gateway-wg/>.
- [6] Kubernetes. *Introducing Gateway API Inference Extension*. 2025. URL: <https://kubernetes.io/blog/2025/06/05/introducing-gateway-api-inference-extension/>.
- [7] Ollama. *Generate a Response*. URL: <https://docs.ollama.com/api/generate>.
- [8] Peiyuan Zhang et al. “TinyLlama: An Open Source Small Language Model”. In: *arXiv preprint arXiv:2401.02385* (2024). URL: <https://arxiv.org/abs/2401.02385>.
- [9] Ingress NGINX Controller. *Prometheus and Grafana Installation*. URL: <https://kubernetes.github.io/ingress-nginx/user-guide/monitoring/>.
- [10] Higress. *Higress AI Gateway*. URL: <https://higress.ai/en/ai-gateway>.
- [11] Kong. *AI Gateway Documentation*. URL: <https://developer.konghq.com/index/ai-gateway/>.
- [12] Envoy AI Gateway. *Envoy AI Gateway Documentation*. URL: <https://aigateway.envoyproxy.io/docs/>.
- [13] Higress. *Token Management*. URL: <https://higress.ai/en/docs/ai/scene-guide/token-management/>.
- [14] Istio. *Gateway API Inference Extension*. URL: <https://istio.io/latest/docs/tasks/traffic-management/ingress/gateway-api-inference-extension/>.
- [15] Higress. *AI Statistics*. URL: <https://higress.ai/en/docs/latest/plugins/ai/api-only/ai-statistics/>.
- [16] Apache Software Foundation. *Apache HTTP Server Benchmarking Tool*. URL: <https://httpd.apache.org/docs/2.4/programs/ab.html>.

A Transparency Report

AI tools were used as a supporting resource during this project. The main model used was GPT-5.6 Sol. It was mainly used to research the topic, understand AI Gateway concepts, and support the setup of the Kubernetes cluster and its required environment. AI was also used to review possible configuration steps and help identify technical issues during the implementation.

The experimental setup, benchmark execution, and evaluation of the results were carried out and reviewed manually. AI suggestions were not accepted without verification. Configuration steps and technical recommendations were checked against the actual system behavior and relevant documentation before they were used. The final conclusions of the report are based on the measurements collected from the implemented system and on my own review of the experiment.