Performance Evaluation of a Kubernetes-Deployed Chatbot Microservice

Cameron Black · Anthony Chapov

Mar 13, 2025

1 Introduction

In recent years, microservices architectures have become the cornerstone of modern data center operations. Their inherent scalability, modularity, ease of deployment, and robustness against failures have significantly transformed software development and operational workflows. However, deploying complex, resource-intensive applications, such as chatbots, within a microservices framework introduces various challenges, particularly concerning resource allocation, load balancing, scaling, the lack of specialized hardware (CPU, NO GPU), and overall performance management.

Chatbots inherently rely on computationally heavy NLP techniques, including text parsing, machine learning inference, and document retrieval methods like Term Frequency-Inverse Document Frequency (TF-IDF). Such complexity means that deploying chatbots on container orchestration platforms like Kubernetes necessitates meticulous management of computing resources and careful consideration of queuing, completed requests, and latency behaviors under varying workloads and setups.

This labs primary objective is to systematically evaluate the performance implications of deploying a chatbot microservice within a Kubernetes-managed environment. Specifically, we aim to investigate the impacts of horizontal scaling—adding multiple replicas to handle concurrent requests—and vertical scaling—increasing CPU and memory resources allocated to each individual replica when GPU is not available (leveraging potential setups for CPU hosting). Our methodology employs rigorous benchmarking techniques using widely accepted performance testing tools, such as `wrk2` for HTTP-based load tests and `ghz` for gRPC performance evaluations. We analyzed real world LLM applications and came to conclusion that our system needs to be tested under various conditions other than increasing/decreasing requests/second, parameter. In real world, many things like single-threading/multi-threading, number of concurrent users (requests) and finally rate of requests may all be more important than the others in different situations. Say, number C (conc requests) might be more important to stress test then rate, since many users send requests to LLM models at quite slow rates with less than 1 request per second (reading message, typing message, sending new response.)

This comprehensive performance analysis addresses critical questions relevant to data center operations and apps running on them, including identifying system bottlenecks, analyzing the latency and throughputs of different workload distributions, and explaining observed behaviors through principles drawn from queuing theory.

2 Design and Implementation

The chatbot application evaluated in this study is structured into two independently scalable microservices: an exposed to clients via REST question-answering model (AI Inference) implemented using FastAPI and Huggin Face QA model, and a backend document retrieval service implemented in Go utilizing the gRPC protocol to send the document as a context to QA model.

The frontend service provides a REST API for clients, enabling user interactions and query processing. Once the user submits a query, the frontend validates and forwards this query via gRPC to the backend retrieval service.

The backend retrieval service leverages TF-IDF computations to efficiently rank and retrieve relevant documents corresponding to user queries about e-commerce shop and all the information about it. Using Go's efficient concurrency capabilities combined with gRPC's low-latency communication, this backend service aims to ensure minimal processing delays. Both microservices are containerized using Docker, facilitating their deployment, scaling, and management within the Kubernetes cluster on VM.

Kubernetes, selected due to its powerful orchestration and self-healing capabilities, manages these microservices through explicit configurations, including defined CPU and memory limits and various number of pods. During our experiments, Horizontal Pod Autoscaler (HPA) and VM-level autoscaling functionalities were intentionally disabled to maintain controlled, reproducible conditions for performance testing. This allowed us to isolate and clearly evaluate the performance impacts attributable solely to scaling strategies.

To enforce CPU resource isolation and ensure accurate performance measurements, we utilized Kubernetes resource constraints alongside explicit CPU affinity mechanisms provided by the underlying Linux kernel. We also heavily analyzed the available hardware on VM (such as a number of cores, the amount of memory it has, the running processes and cpu percentage during the instance, and so on.) This helped us to come up with experimental testing strategies (such as changing cores per replica, replicas, memory, etc.)

2.1 Sample Usage

Below is an example demonstrating typical interaction with the chatbot microservice using the `curl` command-line tool:

Request:

```
curl -X POST
"http://10.103.251.191:5000/chat"
-H "Content-Type: application/json"
-d '{"query":
"What is your return policy?"}'
```

An example JSON response from the chatbot service might look like this:

Response:

```
{
  "response": "You can return unused
  items in their original packaging
  within 30 days of purchase."
}
```

This illustrates the typical interaction flow, highlighting the simplicity of requesting information and receiving clear, structured JSON responses from the deployed chatbot microservice.

3 Experimental Results and Analysis

We used different vertial and hozintal scailing strategies, such as 5 replicas with 1 core per each for chatbot service, and 2 replicas with 3 cores for each for chatbot service. TFIDF service had 1 replica and 1 core since it was not a bottlenck. C is concurrent requets, R is Req/Sec, t is number of threads (where is 1 is single-threaded). Our experiment analyzed different setups for different conditions such as different num theads, num of conc requests, and rate.

3.1 Comparison of Configurations

Table 1: Performance for r=1, t=1, c=5

Metric	A (5×1)	B (2×3)	Winner
Avg Latency	814.18 ms	379.46 ms	B
50th Pctl Latency	793.09 ms	361.47 ms	B
99th Pctl Latency	1.65 s	681.47 ms	B
Throughput	4.90 req/s	4.92 req/s	B
Completed Req	294	295	B

Vertical scaling (fewer replicas with more CPU cores each) notably reduced latency, although

Table 2: Performance for r=1, t=1, c=10

Metric	A (5×1)	B (2×3)	Winner
Avg Latency	11.60 s	11.93 s	A
50th Pctl Latency	12.81 s	12.79 s	B
99th Pctl Latency	21.84 s	22.94 s	A
Throughput	0.50 req/s	0.50 req/s	Tie
Completed Req	30	30	Tie

Table 3: Performance for r=1, t=5, c=10

Metric	A (5×1)	B (2×3)	Winner
Avg Latency	35.08 s	35.20 s	A
50th Pctl Latency	34.90 s	34.44 s	B
99th Pctl Latency	59.4 s	59.4 s	Tie
Throughput	11.59 req/s	2.22 req/s	A
Completed Req	696	133	A

Table 4: Performance for r=1, t=1, c=5

Metric	A (5×1)	B (2×3)	Winner
Avg Latency	716.77 ms	1.54 s	A
50th Pctl Latency	647.68 ms	1.50 s	A
99th Pctl Latency	1.45 s	2.39 s	A
Throughput	0.92 req/s	0.92 req/s	Tie
Completed Req	55	55	Tie

Table 5: Performance for r=1, C=1, t=1

Metric	A (5×1)	B (2×3)	Winner
Avg Latency	145.09 ms	63.42 ms	B
50th Pctl Latency	117.25 ms	51.55 ms	B
99th Pctl Latency	217.09 ms	115.20 ms	B
Throughput	0.85 req/s	0.89 req/s	B
Replicas Used	5	2	B

Table 6: Performance for r=1, C=8, t=8

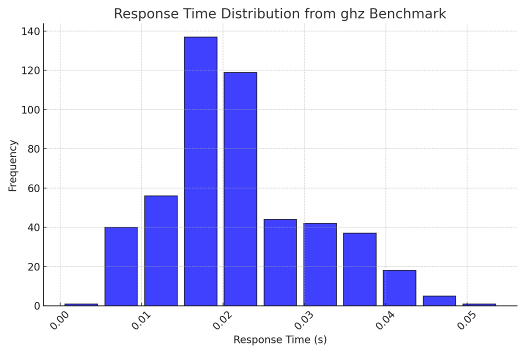
Metric	A (5×1)	B (2×3)	Winner
Avg Latency	34.67 s	35.17 s	A
50th Pctl Latency	33.62 s	35.00 s	A
99th Pctl Latency	58.2 s	59.4 s	A
Throughput	1.77 req/s	2.20 req/s	B
Completed Req	106	132	B

CPU allocation beyond three cores showed diminishing returns, reflecting practical limitations of vertical CPU scaling (Table 7).

Table 7: Vertical Scaling Configurations

Component	Resources Allocated
Frontend Replicas	2 replicas @ 3 cores each
TF-IDF Replica	1 replica @ 1 core

Performance analysis clearly showed that backend TF-IDF retrieval was not the bottleneck, consistently achieving rapid response times under heavy load. Instead, inference computations in the frontend limited performance. Benchmarks with `wrk2` demonstrated stable latency under constant workloads but severe latency spikes under exponential (bursty) conditions, aligning with theoretical predictions of queuing theory.



Future research should explore GPU-based inference acceleration, as GPU resources could significantly reduce latency for computation-intensive tasks, particularly under bursty conditions [2, 3]. These findings reinforce the necessity of careful scaling strategy selection and resource allocation in real-world Kubernetes deployments to maintain acceptable performance and user satisfaction.

4 Conclusion

Our analysis shows that while scaling both ways slighty improved chatbot performance in Kubernetes it was not material. AI inference, CPU-bounded task, remained a major bottleneck, with weak CPU and no GPU leading to rising latency over time. Testing multiple configurations, including those from lecture, revealed that the lack of specialized hardware, such as GPUs, severely limited scalability. Future optimizations, including GPU accelerationa and optimized code, will be essential to reduce latency and improve efficiency in production environments.

References

[1] Casalicchio, Emiliano, and Valeria Perciballi. "Auto-scaling of containers: The impact of relative and absolute metrics." *IEEE Transactions on Cloud Computing*, vol. 9, no. 1, pp. 374–387, 2017.

[2] Hazelwood, Kim, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, et al. "Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective." *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018.

[3] Gan, Yu, Yanqi Zhang, Dailun Cheng, and Yuankai Liu. "An open-source benchmark suite for microservices and their hardware-software implications for cloud and edge systems." *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2019.