Adaptive Load Balancing for Distributed LLM Inference Using Ollama

Rodinei Martins Coelho
Universidade Federal de Mato Grosso do Sul
Três Lagoas, Brasil
rodinei.m.coelho@gmail.com

Vitor Mesaque Alves de Lima
Universidade Federal de Mato Grosso do Sul
Três Lagoas, Brasil
vitor.lima@ufms.br

Resumo

Modelos de linguagem de grande porte (LLMs) se tornaram componentes cruciais de sistemas de IA modernos, apoiando uma ampla gama de aplicações por meio de tarefas de geração de linguagem natural. Embora plataformas de execução de código aberto, como o Ollama, simplifiquem a implantação de LLMs em ambientes locais e *on-premises*, a distribuição eficiente das cargas de inferência entre múltiplas instâncias do Ollama ainda representa um desafio em aberto. Estratégias convencionais de balanceamento de carga, originalmente projetadas para serviços web sem estado, não consideram as características dinâmicas de execução da inferência de LLMs, resultando em uso subótimo de recursos e aumento de latência, especialmente em ambientes computacionais heterogêneos. Este artigo propõe o *OllamaRouter*, uma estratégia especializada de balanceamento de carga voltada à inferência distribuída de LLMs utilizando a API de geração de texto do Ollama. A estratégia proposta aloca dinamicamente as requisições com base em estimativas de tempo de processamento de *tokens* em tempo de execução e na carga das filas, adaptando-se ao desempenho heterogêneo dos nós sem introduzir sobrecarga computacional significativa. Para avaliar a eficácia do *OllamaRouter*, foi conduzido um experimento controlado comparando seu desempenho com as estratégias convencionais *Round Robin* e *Least Connection*, sob uma taxa constante de requisições. Os resultados mostram que o *OllamaRouter* proporciona maior vazão e menor latência média por requisição, especialmente à medida que o número de requisições em espera aumenta. As melhorias práticas observadas e o aumento da estabilidade destacam o potencial do balanceamento de carga adaptativo na otimização de cenários de execução distribuída de LLMs.

Abstract

Large Language Models (LLMs) have become crucial components of modern AI systems, supporting a wide range of applications through natural language generation tasks. While open-source serving platforms such as Ollama simplify the deployment of LLMs in local and on-premises environments, efficiently distributing inference workloads across multiple Ollama instances remains an open challenge. Conventional load balancing strategies, initially designed for stateless web services, fail to account for the dynamic execution characteristics of LLM inference, leading to suboptimal resource utilization and increased latency, particularly in heterogeneous computing environments. This paper proposes *OllamaRouter*, a specialized load balancing strategy tailored for distributed LLM inference using the Ollama text generation API. The proposed strategy dynamically allocates requests based on runtime estimates of token processing time and queue load, adapting to heterogeneous node performance without introducing significant computational overhead. To evaluate the effectiveness of *OllamaRouter*, we conducted a controlled experiment comparing its performance agains conventional Round Robin and Least Connection strategies, under a constant incoming request rate. The results show that *OllamaRouter* delivers higher throughput and lower average request latency, particularly as the number of waiting requests increases. The observed practical improvements and increased stability highlight the potential of adaptive load balancing in optimizing distributed LLM serving scenarios.

CCS Concepts

• **Software and its engineering** → **Software as a service orchestration system**; • **Computing methodologies** → **Natural language processing**; *Distributed computing methodologies*; • **Computer systems organization** → *Client-server architectures.*

Keywords

Ollama, Large Language Model, Load Balancer, AI Engineering.

1 Introduction

Large Language Models (LLMs) have become key enablers of modern natural language processing pipelines, supporting tasks ranging from conversational agents to domain-specific information extraction. Although the availability of open-source models and tools such as Ollama has democratized the deployment of LLMs, the computational cost of inference remains a significant challenge, especially in production environments where inference requests are frequent and workloads are dynamic [15]. Inference, rather than training, now dominates the runtime resource consumption in many applications, raising concerns about latency, infrastructure costs, and energy consumption [4].

Ollama emerged as a practical platform for running open-source LLMs locally, providing an accessible API and simplified deployment process. Its adoption spans various domains, including the deployment of automotive industry chatbots [8] and the development of an R wrapper for seamless API access [3]. While Ollama simplifies local serving, running it efficiently in distributed environments presents new challenges, particularly when managing heterogeneous computational resources and dynamically changing workloads.

The need for scalable and adaptive load balancing strategies becomes especially critical in applications where LLM inference pipelines must process fine-grained, real-time requests. A representative example of such a scenario is risk analysis for mobile applications, where frameworks such as *iRisk* [1] and MApp-IDEA [7] [9] leverage open-source LLM agents to extract and prioritize risks from user reviews. In such systems, inference requests are

short, numerous, and highly variable, demanding a responsive and efficient serving infrastructure. However, existing load balancing strategies are typically not designed to optimize inference latency and resource utilization in heterogeneous environments.

To efficiently support these workloads, we propose a two-fold architecture: (i) an application layer capable of generating dynamic inference workloads, such as risk analysis frameworks or other domain-specific tools; and (ii) a serving infrastructure powered by **OllamaRouter**, a specialized load-balancing framework tailored for distributed LLM inference using the Ollama API. While our experimental evaluation uses a risk analysis scenario as a case study, OllamaRouter is domain-agnostic because it operates solely on generic runtime signals—such as latency, queue length, and throughput—rather than on the semantics of the tasks being processed. As a result, it can support any system that requires scalable, distributed inference over open-source LLMs. It dynamically distributes inference requests based on runtime performance metrics, efficiently balancing load across heterogeneous nodes and improving throughput stability.

In our experimental setup, the challenges of executing risk analysis workloads on distributed Ollama instances highlighted the need for a more intelligent load balancing mechanism. Naive strategies, such as round-robin or least-connections, failed to fully utilize the available computational capacity, leading to resource underutilization in some nodes and overload in others. OllamaRouter addresses this limitation by dynamically adjusting routing decisions, considering runtime metrics such as estimated time per token and queue size.

Our contributions are two-fold:

- We propose an inference-oriented load balancing strategy designed for heterogeneous environments running open-source LLMs through Ollama;
- We empirically validate the OllamaRouter architecture using a real-world workload generated by a risk analysis framework, demonstrating improvements in throughput and latency stability compared to conventional balancing strategies.

By bridging the gap between scalable LLM inference and practical deployment in resource-constrained environments, OllamaRouter advances the state of the art in open-source LLM serving. It enables a wide range of applications — including, but not limited to, risk analysis — to operate efficiently without depending on proprietary cloud services, reducing both operational costs and privacy risks while supporting timely and scalable LLM-based inference.

2 Related Work

Load balancing and communication optimization are fundamental challenges in distributed machine learning systems, particularly in heterogeneous environments. Several strategies have been proposed to address the *straggler problem* and communication bottlenecks during training.

Li et al. [6] proposed the *Adaptive-Dynamic Synchronous Parallel* (A-DSP) model, integrating the *AdaptFR* load balancing strategy. A-DSP dynamically redistributes workloads and adjusts synchronization thresholds according to node performance, efficiently mitigating load imbalance in cloud environments. Compared to Bulk Synchronous Parallel (BSP) and Stale Synchronous Parallel (SSP), A-DSP achieves better throughput without compromising model accuracy.

Qi et al. [13] introduced *Nebula*, a bandwidth allocation strategy for mitigating intra-job network contention in distributed deep neural network (DNN) training. Nebula dynamically allocates network bandwidth between co-located Parameter Server (PS) and worker tasks, improving training stability and throughput in production GPU clusters. This approach distinguishes itself by targeting intra-job contention, complementing prior solutions that primarily addressed inter-job competition.

Ouyang et al. [11] presented a comprehensive survey on communication optimization strategies for distributed deep learning. They categorized techniques into algorithm-level optimizations (e.g., gradient compression, periodic communication) and network-level solutions (e.g., optimized AllReduce schemes). The survey also emphasized overlapping computation and communication, a crucial aspect for improving efficiency in large-scale distributed training.

Duan et al. [2] approached the problem from a data and parameter partitioning perspective. By leveraging data sparsity, they proposed a bipartite graph partitioning heuristic to minimize inter-machine communication and training time in parameter server architectures. Their solution demonstrated significant reductions in communication overhead without sacrificing load balancing.

Zhang et al. [17] tackled parameter synchronization in heterogeneous network topologies, proposing a topology-adaptive scheme based on near-optimal packing of Steiner trees. Their approach dynamically constructs communication trees tailored to available bandwidth, reducing synchronization latency compared to traditional AllReduce.

Jain et al. [4] address the limitations of treating LLM inference as monolithic jobs by explicitly modeling the distinct *prefill* and *decode* phases that contribute to load imbalance. They propose a reinforcement-learning-based router with a response-length predictor and a cost model to distribute queries across instances of the same LLM architecture, achieving improvements in end-to-end latency, Time-To-First-Token (TTFT), and Time-Between-Tokens (TBT). While their work represents an important step toward adaptive routing in inference workloads, it focuses on fine-grained phase-aware balancing within homogeneous model deployments and requires offline training for the router policy.

Another critical line of research focuses on optimizing the internal execution of LLM serving instances. The vLLM[5] serving system, utilizing the PagedAttention algorithm (inspired by operating system paging techniques), significantly increases throughput by achieving near-zero waste in Key-Value (KV) cache memory and allowing flexible sharing. This internal optimization allows vLLM to deliver 2 to 4 times throughput improvements compared to state-of-the-art systems. While vLLM focuses on intra-instance GPU memory efficiency, the proposed OllamaRouter addresses the distinct and complementary challenge of adaptive load balancing for distributed LLM inference.

Most approaches focus on training scenarios and assume either static topologies or pre-defined frameworks. In contrast with Jain et al.[4] approach, our work targets dynamic load balancing in heterogeneous inference environments, where instances are distributed

in hosts with different capabilities are exposed via API. Rather than relying on trained cost models or reinforcement learning, our approach offers a lightweight, adaptable routing strategy designed to handle runtime load variation across diverse LLM instances.

Our work proposes a load balancing strategy specifically designed for distributed inference using the Ollama API. By dynamically adapting to runtime load variations and optimizing both routing and execution latency, our approach improves throughput and stability in heterogeneous inference environments.

To better highlight the differences among the related works and our proposed approach, Table 1 summarizes the key characteristics of each strategy. While prior works have made significant contributions in training scenarios, focusing on load balancing, communication optimization, and synchronization strategies, our work is distinguished by its focus on distributed inference. Specifically, our approach targets dynamic load balancing and latency optimization for heterogeneous environments during inference requests, a scenario that remains underexplored despite its practical importance in API-based serving workflows.

3 OllamaRouter Strategy

The proposed load balancing strategy is guided by two fundamental principles: conceptual simplicity and computational efficiency. To meet these requirements, we designed Algorithm 1, which determines the optimal target instance to handle each incoming request.

The core idea is to estimate, for each instance, the expected waiting time that a new request would experience. This estimation combines two key elements: (i) the predicted number of tokens required to generate a response, and (ii) the instance's average processing time per token. By multiplying these quantities and adding the estimated number of tokens already queued on that instance, the algorithm accounts for both the expected computational cost of the new request and the current workload distribution.

The instance with the lowest estimated waiting time is selected. In the event of a tie, preference is given to the instance with the smaller queue size. Furthermore, instances with no prior time-per-token estimate and no queued requests are prioritized, allowing the system to collect performance metrics and improve future decisions.

The number of tokens, whether in the queue or in the current request, is approximated from the number of characters (excluding consecutive whitespaces) in the corresponding prompt(s), multiplied by a dynamic token estimation factor.

Formally, let I denote the set of available instances. For each instance $i \in I$, we define:

- q_i: total number of characters in the prompts currently queued on instance i;
- γ_i: dynamic queue weight of instance i;
- $\hat{t}_i$: estimated average time per token for instance i;
- $\hat{T}$: estimated number of tokens to be processed for the current request;
- p: number of characters in the current request prompt;
- f: token estimation factor.

The estimated number of tokens for the request and the estimated waiting time for instance i are given by:

$$\begin{aligned} r &= p \times f, \\ \hat{W}(i) &= (\gamma_i q_i f + \hat{T}) \times t_i \end{aligned} \tag{1}$$

The parameter γ_i serves as a dynamic weighting coefficient that adjusts the influence of the queue length on the estimated delay for instance i. Its value lies within the range $[0, 2]$, enabling adaptive sensitivity to the queue load according to observed performance characteristics. For example, instances capable of concurrent processing may operate with lower effective queue weights (closer to 0), since additional queued requests have minimal impact on latency. Conversely, instances susceptible to degradation under load are assigned higher weights (approaching 2), emphasizing the cost of queue accumulation in the estimation.

The algorithm selects the instance i^* that minimizes the estimated waiting time:

$$i^* = \arg\min_{i \in C} W(i) \tag{2}$$

where C is the set of candidate instances, defined as follows:

- If any instance has an available estimate for t_i, consider only those instances or those with empty queues;
- Otherwise, include all available instances.

If multiple instances yield the same minimum estimated waiting time, the instance with the smallest queue size q_i is chosen as the final target.

Computational Complexity. The selection process iterates over all available instances, resulting in a time complexity of $O(n)$, where n is the number of instances. This linear complexity ensures that the algorithm remains lightweight and suitable for runtime decision-making in distributed environments.

Algorithm 1 details the implementation of this strategy, outlining the steps for computing the estimated wait time, selecting candidate instances, and applying the tie-breaking rules to determine the optimal instance.

4 OllamaRouter Architecture

The OllamaRouter architecture has three main components:

(1) The load balancer itself;
(2) A Redis™ server;
(3) Ollama instances.

Figure 1 illustrates the overall architecture of OllamaRouter, highlighting the main components and the sequence of interactions between them. The diagram presents the flow of a request from its reception by the load balancer, through the queuing and instance selection processes, to the final response delivery. Each stage is designed to operate asynchronously, promoting scalability and efficient resource utilization in distributed LLM serving scenarios.

The load balancer was developed using NestJS, a Node.js framework, and BullMQ, a fast and robust background job processing library for Redis™. This choice was made due to the reliability of NestJS and the non-blocking I/O nature of Node.js. Using BullMQ makes it possible to control how many requests are currently being processed by the Ollama instances and also allows for future improvements, such as distributing the load balancer across multiple instances, implementing a dead-letter queue, and more.

Table 1: Comparison of Related Works and Proposed Approach

Approach	Scenario	Focus Area	Adaptivity	Target Problem
Li et al. [6] (A-DSP)	Training	Load balancing	Dynamic	Straggler mitigation in heterogeneous nodes
Qi et al. [13] (Nebula)	Training	Bandwidth allocation	Semi-dynamic	Intra-job network contention
Ouyang et al. [11]	Training	Communication optimization (survey)	-	Algorithmic and network-level optimizations
Duan et al. [2]	Training	Data/parameter partitioning	Static	Communication reduction via data sparsity
Zhang et al. [17]	Training	Synchronization topology optimization	Semi-dynamic	Bandwidth-aware parameter synchronization
Jain et al. [4]	Inference	Phase-aware load balancing	Dynamic	Latency optimization via prefill/decode routing
Kwon et al.[5]	Inference	KV Cache Memory Management / Attention Algorithm	Dynamic	KV cache fragmentation and memory redundancy, low throughput
This work (Proposed)	Inference	Load balancing and routing	Fully dynamic	Load distribution and latency optimization in distributed LLM inference

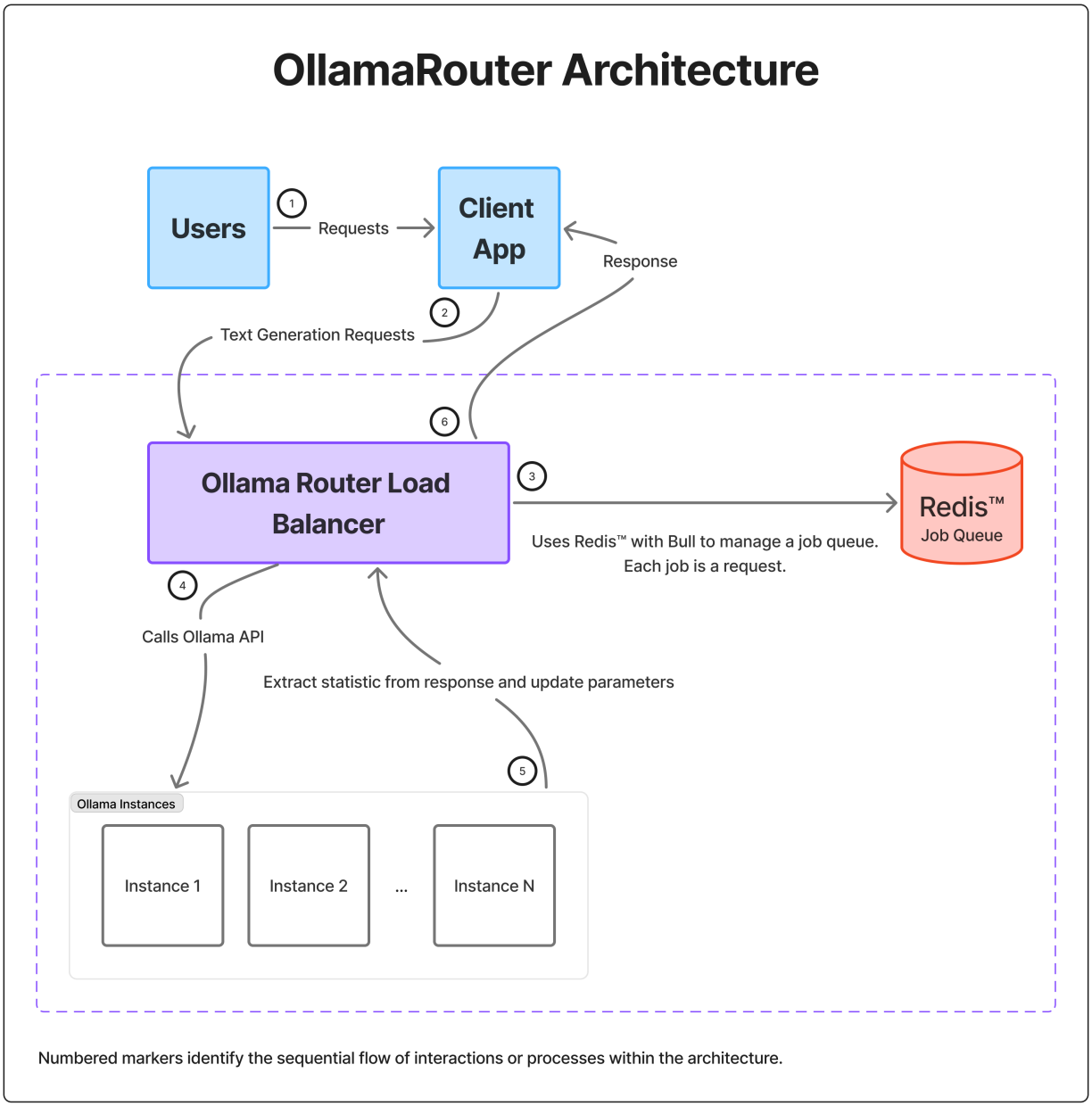

Figure 1: OllamaRouter architecture.

Algorithm 1 OllamaRouter Instance Selection

```
function chooseInstance(evaluatedTokenQuantityEstimate)
    chosenInstance ← NULL
    bestEstimatedWaitTime ← NULL
    hasInstancesWithTime ← checkAnyInstanceWithTime
    if hasInstancesWithTime then
        candidates ← filterCandidateInstances
    else
        candidates ← instances
    end if
    for all instance in candidates do
        queueLengthInCharacters ← instance.queueLengthInCharacters
        queueLengthInTokens ← calculateTokensByCharacters(queueLengthInCharacters)
        queueWeight ← instance.queueWeight
        timePerTokenEstimate ← instance.timePerTokenEstimate
        estimatedWaitTime ← (queueLengthInTokens × queueWeight + evaluatedTokenQuantityEstimate) × timePerTokenEstimate
        if isBetterCandidate(chosenInstance, estimatedWaitTime, queueLengthInTokens) then
            chosenInstance ← instance
            bestEstimatedWaitTime ← estimatedWaitTime
        end if
    end for
    return chosenInstance
end function

function isBetterCandidate(chosenInstance, estimatedWaitTime, queue)
    if chosenInstance = NULL then
        return TRUE
    else if estimatedWaitTime < bestEstimatedWaitTime then
        return TRUE
    else if estimatedWaitTime = bestEstimatedWaitTime and queue < chosenInstance.queueLengthInTokens then
        return TRUE
    else
        return FALSE
    end if
end function

function checkAnyInstanceWithTime
    return TRUE if ∃ instance in instances such that instance.timePerTokenEstimate ≠ NULL
end function

function filterCandidateInstances
    return [i ∈ instances | i.timePerTokenEstimate ≠ NULL or(i.timePerTokenEstimate = NULL and (i.queueLengthInTokens = NULL or
i.queueLengthInTokens = 0))]
end function

function calculateTokensByCharacters(charactersCount)
    return charactersCount × tokenEstimateFactor
end function
```

In the proposed architecture, requests go through a series of stages:

(1) Request reception and queuing;
(2) Ollama instance selection;
(3) Parameter updating;
(4) Response delivery.

The following subsections address each stage.

4.1 Request Reception and Queuing

The load balancer receives a request with prompt and model in the /ollama/generate endpoint. The BullMQ library is then used to enqueue the request using the Redis™ server and wait for the request processing.

4.2 Instance Selection

The load balancer handles incoming requests concurrently. An instance is selected according to the strategy described in Section 3, after which the request is forwarded to the chosen instance and the response is awaited.

If an error occurs during request transmission or while waiting for the response, the instance selection process is retried up to a maximum of four times.

4.3 Parameter Updating

When an instance returns a response, specific fields from the payload are used to update the parameters involved in the instance selection process. The following properties are extracted:

- `eval_count`: Number of tokens generated in the response;
- `promp_eval_count`: Number of tokens present in the prompt.

These values, together with the request's waiting time, are employed to update both the *token estimation factor* and the *time-per-token estimate* of the selected instance. The update procedure is based on an exponential moving average (EMA), allowing the system to gradually adapt to variations in performance while preserving stability.

Let T denote the total number of tokens, $\hat{t}_i$ the estimated time per token for instance i, and W the measured waiting time for the request. The updates are computed as follows:

$$
\begin{aligned}
T &= \texttt{eval\_count} + \texttt{promp\_eval\_count}, \\
t &= \frac{W}{T}, \\
\hat{t}_i &\leftarrow \alpha t + (1-\alpha)\hat{t}_i, \\
f &\leftarrow \alpha \frac{T}{p} + (1-\alpha) f, \\
\gamma &\leftarrow \begin{cases} 1, & \text{if } \hat{W} = 0, \\ \max\left(0, \min\left(2,\ \gamma\left(1 + \alpha\left(\frac{W}{\hat{W}} - 1\right)\right)\right)\right), & \text{otherwise.} \end{cases}
\end{aligned}
\tag{3}
$$

Here, α represents the EMA smoothing factor, p denotes prompt character count, and $\hat{W}$ is the estimated wait time. Together, these updates enable *OllamaRouter* to continuously refine its internal estimates of instance performance, improving future selection decisions.

Additional details about some of these variables and their roles can be found in Section 3.

4.4 Response delivery

Once the selected Ollama instance completes the inference process, its response is forwarded to the load balancer, where it is used to compose the final reply to the original client request. After delivering the response, the corresponding job is removed from the queue, maintaining the system's flow and ensuring that subsequent requests are processed efficiently.

5 Experiment Design

To evaluate the efficiency of the proposed strategy a quasi-experiment (a kind of empirical study where the assignment of treatments to subjects is not random) [16]. To do so, some research questions were raised and a test was conducted against our proposed solution and two conventional strategies.

5.1 Research Questions

The main research question is: for the usage of the Ollama text generation endpoint, can the proposed strategy bring better performance than conventional load balancing strategies in a scenario with computers with different performance capabilities? To answer this question, we formulated the following specific research questions:

- **RQ1**: Can *OllamaRouter* achieve better throughput than conventional strategies?
- **RQ2**: Can *OllamaRouter* achieve lower average request duration than conventional strategies?
- **RQ3**: Does *OllamaRouter* maintain stable performance over time as requests continuously arrive?

The experiment was designed to extract quantitative metrics to address these research questions.

5.2 Experiment Definition

The experiment is defined as follows:

- For a predetermined model,
- For each strategy,
- Send requests at a consistent rate over a specified duration,
- In a common context.

5.3 Preparing and Planning

For the preparation of the conducted tests, it was necessary the following steps:

- Choose a model;
- Generate the test workload;
- Prepare the test environment;
- Choose a tool and strategies to test against;
- Operate the tests scenarios.

5.4 Experimental Package

To promote reproducibility and support open science practices, the experimental `OllamaRouter` package has been made publicly available at https://github.com/rodineimcoelho/OllamaRouter. The repository contains a `README.md` file with detailed instructions for installing and running the package. Furthermore, the test scripts, configuration files, and experimental results are openly accessible at https://github.com/rodineimcoelho/OllamaRouter-tests.

5.5 Inference Model

To generate a realistic and complex inference workload capable of stressing the load balancing mechanism, we adopted the iRisk model [1], a modular framework designed to automate risk analysis from mobile app reviews. This model reflects the scenario that originally motivated this study, characterized by dynamic, fine-grained requests and heterogeneous computational demands, making it representative of typical distributed inference workloads.

The iRisk framework builds upon a fine-tuned version of the LLAMA3 [10] 8B language model, adapted with Low-Rank Adaptation (LoRA) and optimized using the Unsloth framework. The

fine-tuning was performed on a dataset of mobile app reviews annotated with issues, functionalities, and risk attributes, enabling the model—referred to as i-LLAMA—to extract structured risk assessments from unstructured text. The fine-tuned model uses 4-bit quantization to reduce memory consumption and improve inference efficiency, supporting deployment in resource-constrained or heterogeneous environments.

5.6 Workload Generation

The workload used in our experiments was generated using a modular framework previously developed to support automated risk analysis from app reviews [1]. This system continuously processes user reviews, extracting structured information regarding software issues and their associated risk levels. Its design enables the generation of inference requests that realistically simulate the variability, volume, and complexity of real-world app feedback.

The core inference task is formalized as a multi-output supervised mapping from a raw user review R_i to a tuple of extracted attributes:

$$f_\Theta(R_i) = (\hat{I}_i, \hat{F}_i, \hat{S}_i, \hat{P}_i) \tag{4}$$

where:

- $\hat{I}_i$: The identified issue or complaint described by the user;
- $\hat{F}_i$: The functionality affected by the issue;
- $\hat{S}_i \in \{1, 2, 3, 4, 5\}$: The predicted severity level of the issue;
- $\hat{P}_i \in [0, 100]$: The predicted likelihood of the issue occurring.

The system projects these predictions onto a two-dimensional risk matrix $\mathcal{M}(s, p)$, where each cell aggregates the number of occurrences of issues with specific severity-likelihood coordinates. This dynamic matrix evolves over time as new reviews are processed.

Each user review processed by the framework is submitted as a separate inference request, making the workload composed of fine-grained, heterogeneous tasks. The natural language inputs, combined with structured risk extraction, generate a highly variable and bursty workload that is well-suited to evaluate load balancing strategies in distributed LLM inference scenarios.

This workload setup ensures that our evaluation scenario reflects real-world patterns of user-generated feedback, capturing both high-volume and dynamic aspects of inference loads encountered in app ecosystems.

5.7 Environment

For the experimentation environment, the computers described in Table 2 were used.

Docker was used to run Ollama containers and the load balancers under test. A Redis™ container was used only when testing our proposed solution. Each computer ran one Ollama container, and Computer A ran the load balancer under test. When testing our proposed solution, Computer A also ran the Redis™ container. The figure 2 provides a diagram illustrating the test environment.

It is important to note that the computers used in the experiments have different performance capabilities, as this variation is intentional and directly related to the main research question.

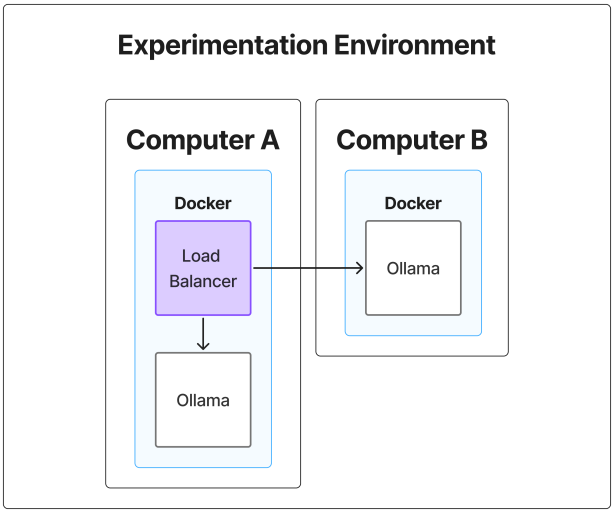

Figure 2: Environment Diagram

5.8 Choice of Reference Tool

To choose the tool that would be compared with our proposed solution, two requirements were defined:

- That it be a free and open-source tool, to facilitate reproducibility of the tests without costs;
- That it be a relevant tool.

The chosen tool is HAProxy, a high-performance open-source solution that meets the established criteria and offers many algorithms to choose from. [12, 14]

Among the algorithms offered by HAProxy, the round-robin and least connection strategies were selected for the experiment. These algorithms were chosen because they operate without requiring any modification to the request (e.g., additional headers) or any algorithm-specific preconfiguration. The Source, URL Parameter, and URI algorithms could not be used, as all requests originate from the same IP address and share the same URI.

In the round-robin algorithm, requests are sent to each instance in order, one after another. In the least-connections algorithm, requests are sent to the instance that currently has the fewest active connections. [12, 14]

5.9 Test Operation

To execute the tests, k6, a load testing tool owned by Grafana, was used to run test scripts and generate results.

The test was conducted with a constant arrival rate of one request every 40 seconds over a period of 40 minutes (0.025 requests per second). After this period, the test entered a 30-minute graceful stop phase to allow pending requests to complete, ensuring a well-defined termination.

Due to internal timing characteristics of k6, slight variations in the total number of requests may occur. As a result, some test runs produced 61 requests instead of 60. This minor discrepancy is inherent to the tool's scheduling mechanism and does not have a significant impact on the overall results or their interpretation.

Table 2: Computers

Computer	Processor	Memory	GPU	Operating System
A	Intel® Core™ i7-6700K 4.00GHz	64GB	NVIDIA Titan X (PASCAL)	Linux (Debian)
B	Intel® Xeon® E5-1650 v2 3.5GHz	64GB	N/A	Linux (Debian)

To ensure that a real improvement was achieved, the test was also conducted with Computer A as a single host. This allowed us to observe whether the least performant computer would negatively affect any of the strategies.

6 Results and Discussion

Table 4 presents the main result metrics collected during the test.

The Round Robin and Least Connection strategies were unable to complete all requests within the test duration. The testing tool, k6, calculated the metrics based only on the completed requests. Therefore, better metric values do not necessarily indicate better performance, as some requests were not successfully completed.

Figures 3, 4, and 5 provide a comparative view of the average request waiting time, completion rate and throughtput, respectively, over the test duration time for each strategy. The lines in the plotted charts starts on the first completed and request and ends on the last completed requests.

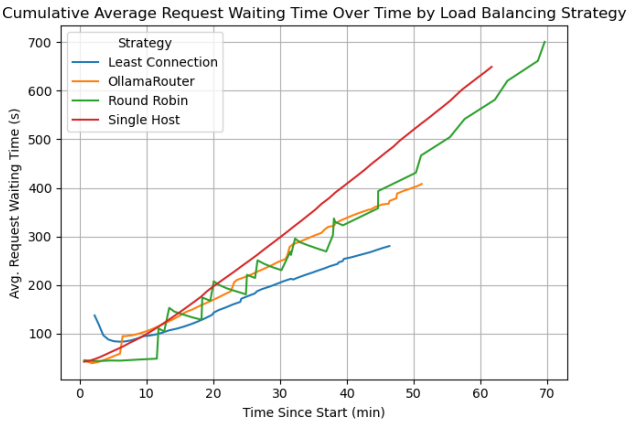

Figure 3: Cumulative Average Request Waiting Time

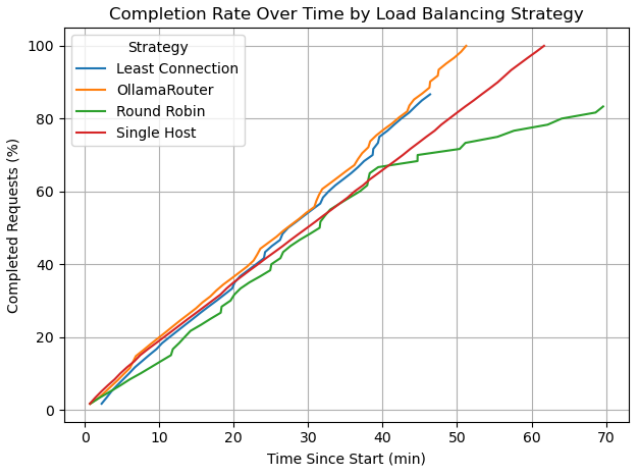

Figure 4: Completion Rate Over Time

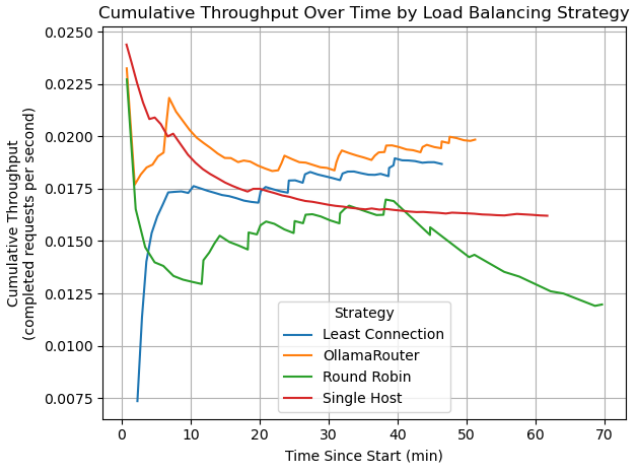

Figure 5: Cumulative Throughput

6.1 Request Waiting Time

The proposed *OllamaRouter* strategy achieved the second-lowest request waiting time, behind only the Least Connection strategy. However, Least Connection did not complete all requests within the test duration. This behavior suggests that using only the number of active connections as a parameter to decide which computer receives each request is not reliable in heterogeneous environments. Although Least Connection showed a lower mean waiting time, approximately 13.34% of the requests were not completed at all during the test. In terms of waiting time, *OllamaRouter* is, therefore, a more reliable and stable option.

In comparison, the average request waiting time of *OllamaRouter* was approximately 41.76% lower than that of Round Robin and 37.14% lower than when using a single host.

6.2 Completion Rate

The completion rate metric represents the proportion of successfully completed requests relative to the total number of requests issued. Among all evaluated strategies, only *OllamaRouter* and the single-host configuration achieved a 100% completion rate. Nevertheless, *OllamaRouter* reached full completion more than ten

Table 3: Comparative Results

Strategy	Requests Sent	Completed Requests	Completion Time (s)	Avg. Request Waiting Time (s)	Throughput (req/s)
Least Connection	60	52	N/A	280227.94	1.8685×10^{-2}
Round Robin	60	50	N/A	700479.01	1.1970×10^{-2}
Single Host	60	60	3701	649019.42	1.6212×10^{-2}
OllamaRouter (this work)	61	61	3073	407972.21	1.9850×10^{-2}

Table 4: Detailed Request Waiting Time Metrics (seconds)

Strategy	Avg.	Min.	Med.	Max.	P(90)	P(95)
Least Connection	280227.94	55074.77	305815.69	504110.54	440660.20	460342.73
Round Robin	700479.01	41495.86	57351.83	2637355.97	2171772.59	2428499.04
Single Host	649019.42	42255.43	645779.98	1341945.45	1210203.67	1264345.11
OllamaRouter (this work)	407972.21	34255.59	405956.28	932744.34	672525.00	698354.86

minutes earlier, indicating a consistent and significant improvement in performance.

6.3 Cumulative Throughput

Figure 5 illustrates the cumulative throughput, defined as the number of successfully completed requests divided by the elapsed time along the x-axis. During the initial phase of the experiment, the single-host configuration exhibited slightly higher throughput. However, as the pending workload accumulated, *OllamaRouter* progressively outperformed it, demonstrating superior overall efficiency.

Consistent with the results observed for the other metrics, the Round Robin strategy yielded the lowest performance. After approximately 40 minutes of execution, its throughput declined sharply, indicating instability under sustained load conditions.

6.4 Discussion

This section summarizes and interprets the main findings of the experimental evaluation, emphasizing the practical implications of adaptive load balancing in heterogeneous inference environments.

The consolidated results confirm that adaptive load balancing effectively improves both latency and throughput under heterogeneous conditions. The proposed *OllamaRouter* strategy consistently outperformed conventional approaches over time and is expected to maintain superior performance under higher concurrency levels.

These findings indicate that *OllamaRouter*'s adaptive instance selection mechanism, which considers token estimates and runtime execution time, enables more efficient request distribution across nodes with varying capabilities. Although the statistical evidence is constrained by the test scope and sample size, the observed trends align with the expected advantages of adaptive load balancing in heterogeneous systems.

Practically, *OllamaRouter* provides a lightweight and domain-agnostic solution for enhancing LLM inference with Ollama. In distributed environments where node performance varies—such as edge computing, on-premises clusters, or hybrid cloud deployments—static or naive load balancing strategies fail to adapt to runtime workload fluctuations. By dynamically routing inference requests based on runtime feedback, *OllamaRouter* improves responsiveness, stabilizes throughput, and optimizes resource utilization with minimal overhead. As workload intensity and heterogeneity increase, its adaptive mechanisms are expected to deliver even greater benefits in real-world deployments.

7 Threats to Validity

As with any empirical study, this work is subject to several threats to validity that may affect the generalizability and reliability of its results. We discuss the main threats categorized as internal, external, construct, and conclusion validity.

7.1 Internal Validity

Internal validity threats refer to potential factors that could affect the cause-effect relationship between the proposed load balancing strategy and the observed results. Since the experiments were conducted in a controlled environment, the influence of background processes or resource contention on the nodes cannot be entirely ruled out. To mitigate this, the test environment was isolated from unrelated workloads, and the same hardware conditions were maintained across all experiments. Nevertheless, the initialization phase of the LLM instances may introduce minor variability in response times, especially in the first inference requests.

7.2 External Validity

External validity concerns the generalizability of the results beyond the experimental context. Our experiments were performed using a specific workload generated by a risk analysis framework and executed on a limited number of machines with known performance differences. While the OllamaRouter strategy is designed to be domain-agnostic, the experiments did not evaluate its performance under different inference workloads, models, or hardware configurations. Future studies are needed to validate the strategy in other application domains, model types, and large-scale distributed environments.

7.3 Construct Validity

Construct validity relates to whether the measurements used accurately capture the intended constructs. In our study, we used throughput and request duration as proxies for balancing effectiveness and system responsiveness. While these are standard metrics for evaluating load balancers, other aspects, such as fairness in workload distribution or energy consumption, were not measured. Additionally, the use of simulated workloads generated by an automated framework may differ from real-world user behavior in production systems.

7.4 Conclusion Validity

Conclusion validity refers to the extent to which the statistical analysis supports the conclusions. Given the relatively small scale of the experiment and the deterministic nature of the workload, we did not perform statistical hypothesis testing. The observed trends in throughput and latency across multiple test steps provide empirical support for our conclusions, but larger-scale and repeated experiments would strengthen the confidence in these findings.

8 Final Remarks and Future Work

This paper presented *OllamaRouter*, a specialized load balancing strategy designed to improve distributed inference performance for the Ollama text generation API. By dynamically adapting to runtime execution metrics such as queue length and token processing time, OllamaRouter effectively distributes workloads across heterogeneous nodes, reducing response latency and increasing throughput stability.

Our experimental results, based on a real-world risk analysis workload, demonstrated that the proposed strategy achieves better throughput and request duration metrics compared to conventional strategies such as round-robin and least-connections. The strategy proved especially effective in mitigating performance variability under increasing loads, showing potential for scalable and cost-efficient LLM serving in resource-constrained environments.

Despite these positive results, this study has some limitations. The evaluation was performed in a controlled environment with a limited number of nodes and hardware configurations. Furthermore, the workload was limited to a single use case based on a risk analysis framework. Broader validation, covering different types of inference workloads and larger distributed environments, is necessary to fully assess the generalizability of the proposed approach.

As future work, we intend to extend OllamaRouter in several directions:

- Incorporate more advanced runtime metrics, such as GPU utilization and memory consumption, to refine the load balancing decisions;
- Experiment with predictive balancing strategies, using historical workload patterns to anticipate future bottlenecks;
- Examine the impact of parameter tuning on load balancing behavior, such as varying the alpha value in the exponential moving average;
- Incorporate logging features.
- Extend OllamaRouter with more resilience features, such as backoff in the retry mechanisms and dead-letter queue support;
- Evaluate the scalability of OllamaRouter in large-scale distributed environments with dozens of heterogeneous nodes;
- Integrate OllamaRouter with other LLM serving platforms, assessing its applicability beyond the Ollama ecosystem;
- Provide support for mixed workloads, combining multiple model types and inference tasks in the same distributed environment.

By continuing to evolve the OllamaRouter architecture, we aim to contribute to the broader adoption of open-source LLMs in distributed and heterogeneous environments, democratizing access to scalable and cost-effective language model inference.

References

[1] Vitor Mesaque Alves De Lima, Jacson Rodrigues Barbosa, and Ricardo Marcodes Marcacini. 2024. iRisk: A Scalable Microservice for Classifying Issue Risks Based on Crowdsourced App Reviews. In *2024 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 858–862. doi:10.1109/ICSME58944.2024.00091

[2] Yubin Duan, Ning Wang, and Jie Wu. 2021. Minimizing Training Time of Distributed Machine Learning by Reducing Data Communication. *IEEE Transactions on Network Science and Engineering* 8, 2 (2021), 1802–1815. doi:10.1109/TNSE.2021.3073897

[3] Johannes B. Gruber and Maximilian Weber. 2024. rollama: An R package for using generative large language models through Ollama. *arXiv e-prints*, Article arXiv:2404.07654 (April 2024), arXiv:2404.07654 pages. arXiv:2404.07654 [cs.CL] doi:10.48550/arXiv.2404.07654

[4] Kunal Jain, Anjaly Parayil, Ankur Mallick, Esha Choukse, Xiaoting Qin, Jue Zhang, Íñigo Goiri, Rujia Wang, Chetan Bansal, Victor Rühle, Anoop Kulkarni, Steve Kofsky, and Saravan Rajmohan. 2025. Performance Aware LLM Load Balancer for Mixed Workloads. In *Proceedings of the 5th Workshop on Machine Learning and Systems* (World Trade Center, Rotterdam, Netherlands) *(EuroMLSys '25)*. Association for Computing Machinery, New York, NY, USA, 19–30. doi:10.1145/3721146.3721947

[5] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. arXiv:2309.06180 [cs.LG] https://arxiv.org/abs/2309.06180

[6] Mingwei Li, Jilin Zhang, Jian Wan, Yongjian Ren, Li Zhou, Baofu Wu, Rui Yang, and Jue Wang. 2020. Distributed machine learning load balancing strategy in cloud computing services. *Wireless Networks* 26, 8 (2020), 5517–5533. doi:10.1007/s11276-019-02042-2

[7] Vitor Mesaque Alves de Lima, Jacson Rodrigues Barbosa, and Ricardo Marcondes Marcacini. 2023. MApp-IDEA: Monitoring App for Issue Detection and Prioritization. In *Proceedings of the XXXVII Brazilian Symposium on Software Engineering (SBES '23)*. Association for Computing Machinery, New York, NY, USA, 180–185. doi:10.1145/3613372.3613417

[8] Fei Liu, Zejun Kang, and Xing Han. 2024. Optimizing RAG Techniques for Automotive Industry PDF Chatbots: A Case Study with Locally Deployed Ollama Models. *arXiv e-prints*, Article arXiv:2408.05933 (Aug. 2024), arXiv:2408.05933 pages. arXiv:2408.05933 [cs.IR] doi:10.48550/arXiv.2408.05933

[9] Vitor Mesaque Alves de Lima, Jacson Rodrigues Barbosa, and Ricardo Marcondes Marcacini. 2024. Monitoring Temporal Dynamics of Issues in Crowdsourced User Reviews and their Impact on Mobile App Updates. In *2024 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 630–635. doi:10.1109/ICSME58944.2024.00064

[10] Meta LLaMA Team. 2024. Introducing Meta Llama 3: The most capable openly available LLM to date. https://ai.meta.com/blog/meta-llama-3/ Accessed: 2024-06-27.

[11] Shuo Ouyang, Dezun Dong, Yemao Xu, and Liquan Xiao. 2021. Communication optimization strategies for distributed deep neural network training: A survey. *J. Parallel and Distrib. Comput.* 149 (2021), 52–65. doi:10.1016/j.jpdc.2020.11.005

[12] Diego Pereira, Lucas Bezerra, Jacyana Nunes, Itamir Barroca Filho, and Frederico Lopes. 2023. Performance Efficiency Evaluation based on ISO/IEC 25010:2011 applied to a Case Study on Load Balance and Resilient. In *Proceedings of the 24th Workshop on Testing and Fault Tolerance* (Brasília/DF). SBC, Porto Alegre, RS, Brasil, 108–120. doi:10.5753/wtf.2023.787

[13] Qiang Qi, Fei Xu, Li Chen, and Zhi Zhou. 2021. Rationing bandwidth resources for mitigating network resource contention in distributed DNN training clusters.

CCF Transactions on High Performance Computing 3, 3 (2021), 171–185. doi:10.1007/s42514-021-00064-x

[14] Connor Rawls and Mohsen Amini Salehi. 2022. Load Balancer Tuning: Comparative Analysis of HAProxy Load Balancing Methods. *arXiv e-prints*, Article arXiv:2212.14198 (Dec. 2022), arXiv:2212.14198 pages. arXiv:2212.14198 [cs.DC] doi:10.48550/arXiv.2212.14198

[15] Siddharth Samsi, Dan Zhao, Joseph McDonald, Baolin Li, Adam Michaleas, Michael Jones, William Bergeron, Jeremy Kepner, Devesh Tiwari, and Vijay Gadepally. 2023. From Words to Watts: Benchmarking the Energy Costs of Large Language Model Inference. *2023 IEEE High Performance Extreme Computing Conference, HPEC 2023* (2023). doi:10.1109/HPEC58863.2023.10363447 Cited by: 81.

[16] Claes Wohlin, Per Runeson, Martin Hst, Magnus C. Ohlsson, Bjrn Regnell, and Anders Wessln. 2012. *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated.

[17] Zhe Zhang, Chuan Wu, and Zongpeng Li. 2021. Near-Optimal Topology-adaptive Parameter Synchronization in Distributed DNN Training. In *IEEE INFOCOM 2021*. IEEE, 1–9. doi:10.1109/INFOCOM42981.2021.9488678