

ScaleLLM: A Resource-Frugal LLM Serving Framework by Optimizing End-to-End Efficiency

Yuhang Yao¹, Han Jin¹, Alay Dilipbhai Shah¹, Shanshan Han¹,
Zijian Hu¹, Dimitris Stripelis¹, Yide Ran¹, Zhaozhuo Xu¹,
Salman Avestimehr¹, Chaoyang He¹

¹TensorOpera Inc.

Correspondence: yuhang@tensoropera.com

Abstract

Large language models (LLMs) have surged in popularity and are extensively used in commercial applications, where the efficiency of model serving is crucial for the user experience. Most current research focuses on optimizing individual sub-procedures, *e.g.* local inference and communication, however, there is no comprehensive framework that provides a holistic system view for optimizing LLM serving in an end-to-end manner. In this work, we conduct a detailed analysis to identify major bottlenecks that impact end-to-end latency in LLM serving systems. Our analysis reveals that a comprehensive LLM serving endpoint must address a series of efficiency bottlenecks that extend beyond LLM inference. We then propose ScaleLLM, an optimized system for resource-efficient LLM serving. Our extensive experiments reveal that with 64 concurrent requests on Mixtral 8x7B, ScaleLLM achieves a $4.3 \times$ speed up over vLLM and outperforms state-of-the-arts with $1.5 \times$ higher throughput¹.

1 Introduction

Large language models (LLMs) have significantly changed the field of natural language processing and have been widely used in commercial applications. However, serving LLMs effectively remains challenging due to system latency, query concurrency, and computational resources constraints. LLM applications are typically deployed as online services where users expect real-time responses, while any delay can impact user experience, making low latency to be crucial. Also, the computationally intensive nature of LLMs, which involve inference with billions of parameters, requires substantial computational resources. Moreover, achieving scalability to handle multiple concurrent requests without performance degradation further complicates the serving process.

¹[https://tensoropera.ai/prod/model/mistralai/
ScaleLLM-Mixtral-8x7B](https://tensoropera.ai/prod/model/mistralai/ScaleLLM-Mixtral-8x7B)

Latency in LLM serving primarily arises from the processing at the serving engine as well as the gateway. The serving engine is the core component responsible for executing the LLM inference tasks. It optimizes resource allocation to handle the intensive computational workload of LLMs to efficiently utilize computational resources, such as GPUs. The gateway manages communication between clients (*e.g.*, end-users or applications) and LLM instances. It handles incoming requests, directs them to the LLM instances, and ensures that responses are returned correctly and efficiently.

Existing research focuses on optimizing individual subprocedures of LLM serving, especially accelerating local inference speeds (Dao et al., 2022; VLLM AI; NVIDIA). However, in commercial LLM applications, end-to-end latency, introduced from functionalities of the gateway, becomes the most significant bottleneck. Meanwhile, commercial LLM applications have specific requirements on serving, which directly accessing a single LLM instance fails to address. In practice, commercial LLM applications must satisfy several critical requirements for efficient and reliable inference: *i)* *fault tolerance*: there must be replicas of LLMs to ensure that the serving system can select appropriate replica upon receiving requests under a specific resource constraint, thereby maintaining service reliability even when individual replica instance fails; *ii)* *inference control*: the serving system should manage the inference process to ensure that the models are accessed with authentication and can produce responses that are appropriate and safe while adapting to different user demands; *iii)* *low latency*: to ensure the user experience, the serving system should process inferences efficiently and deliver responses in real-time; *iv)* *concurrency*: small batch sizes and high throughput for individual requests become impractical in real-world LLM services such as ChatGPT, where the queries can be frequent, *e.g.*, with queries per second (QPS)

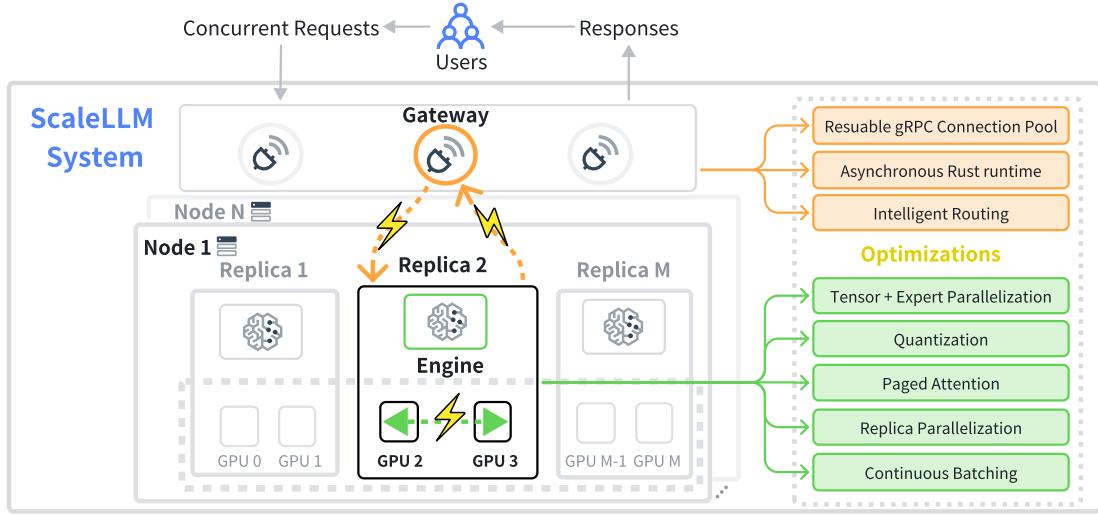


Figure 1: Overview of ScaleLLM Serving System. ScaleLLM provides an optimized gateway for balancing workloads of user requests to different inference replicas and an efficient serving engine for promptly response with high concurrent requests.

often exceeding 200 (Lammertyn, 2024); v) *frugal computational resource usage*: given the substantial computational demands, optimizing resource utilization is crucial to prevent excessive costs and ensure the reliable operation of the serving system. Thus, a comprehensive LLM serving system must balance computational efficiency, concurrency, and latency to manage the high volume of requests.

To address the efficiency of LLM serving comprehensively, we present ScaleLLM, an optimized LLM serving system, as well as an end-to-end measurement, to meet real-world requirements of commercial LLM applications. As shown in Figure 1, to address different challenges in commercial LLM applications, ScaleLLM optimizes two crucial modules, including *i*) a Routing Module that efficiently does replica level load balancing and data transmission; and *ii*) a strong LLM engine to inference promptly with high concurrent requests. Our contributions are summarized as follows.

- We go beyond optimizing the latency of LLM inference and measure the end-to-end time and resource cost of maintaining an LLM serving endpoint. Moreover, we present a breakdown of the end-to-end LLM serving endpoint to showcase the overhead introduced in each component.
- We optimize LLM serving for both the local inference and the gateway, and provide a recipe for efficient LLM serving frameworks for commercial applications. Specifically, instead of random selection, we evaluate different gateways in §4.2

and choose Rust as the backend due to its superior performance in terms of latency, concurrency handling, and resource efficiency.

- Extensive experiments highlight that with 64 concurrent requests on Mixtral 8x7B, ScaleLLM achieves a $4.3 \times$ speed up over vLLM and outperforms the state-of-the-arts with $1.5 \times$ higher throughput (Fireworks AI; Together AI).
- Lastly, we synthesize our insights and findings from extensive experiments into the *blueprint design* of a dynamic inference load balancing system engineered to adapt to varying workloads to address the critical requirements of contemporary production environments.

2 Related Work

Many pre-trained open LLMs have been released since last year, where the most commonly used models include Mixtral 8x7B (Jiang et al., 2024) and Llama-3 (Touvron et al., 2023)). Such open-source models motivate the industry to build public LLM-serving endpoints (Together AI; Fireworks AI) and empower researchers to work on speeding up the inference speed. FlashAttention (Dao et al., 2022) is proposed to approximate the attention calculation to reduce memory usage with fast computation. By representing the weights and activations with low-precision data types, Model Quantization (Lin et al., 2024; Liu et al., 2024) is also widely adopted to reduce memory and computation costs.

During LLM serving, the key-value cache (KV cache) memory for each request is huge and grows and shrinks dynamically. Page attention (Kwon et al., 2023) is proposed for efficient management of KV cache memory blocks with exact model computation. Built on top of PagedAttention, vLLM (VLLM AI) is proposed as a high-throughput distributed LLM serving engine that aims to increase GPU utilization and hence speeds up the throughput of LLM serving. TensorRT-LLM (NVIDIA) provides industrial-level integration of these state-of-the-art optimization methods with Python and C++ runtimes to perform inference efficiently on NVIDIA GPUs.

However, these serving engines primarily focus on accelerating local LLM computation, neglecting other crucial components such as gateway and routing. To the best of our knowledge, our proposed ScaleLLM is the first to offer an end-to-end latency measurement and optimization specifically for resource-efficient LLM serving.

3 Benchmark LLM Serving Solutions

We first provide the end-to-end system breakdown of serving latency in §3.1, then provide the benchmark results of baselines in §3.2.

3.1 System Breakdown

To optimize the user’s experience with low latency, there are two components to be focused on.

Replica Router. In practical applications, the serving endpoint is not a single instance but consists of multiple replicas and schedulers to facilitate load balancing. The router functions as a crucial module that mediates request and response transformation between the engine and the end user. Given the high concurrency of user requests, the router typically operates under significant pressure.

Inference Engine within Replica. A replica represents the smallest unit of resource allocation and is designed to be homogeneous. Each replica houses an instance of the inference engine, utilizing one or more GPUs with a specific parallelism pattern, such as tensor parallelism or process parallelism.

3.2 Performance of Baseline Solutions

For the routing gateway, FastAPI is widely adopted due to its user-friendliness and ease of setup. For the serving engine, there are two baselines, including Huggingface Transformer (Wolf et al., 2019) and vLLM (Kwon et al., 2023). Benchmark results

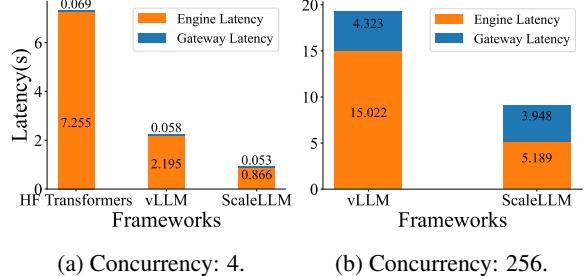


Figure 2: Comparisons with the two baseline solutions. ScaleLLM is applied without gateway optimization.

in Figure 2 indicate that with 4 concurrent requests, engine latency is the primary bottleneck. However, at 256 concurrent requests, the gateway latency becomes the predominant bottleneck.

4 Optimizations

This section discusses the optimization goal, then decomposes the latency into engine latency and gateway latency, and optimizes each component.

Optimization Goal. Our goal is to leverage various optimization techniques on both the inference engine and the replica router to improve the end-to-end serving performance. The inference engine is applied with different frameworks and optimization methods to increase the throughput and decrease the latency. For the replica router, we break down the latency to engine latency and gateway routing latency. The goal is to decrease the engine latency, especially when the concurrency is high.

4.1 Optimize Inference Engine

We mainly focus on optimizing the Mixture of Experts (Jiang et al., 2024) LLMs that are being widely used nowadays.

Model Parallelization. We utilize parallel processing across multiple GPUs to accommodate models with multiple experts (MoEs), as the model may not fit within the memory of a single GPU. As shown in Figure 9 in §Appendix, TensorRT engine (NVIDIA) offers three approaches for achieving parallelism, including Tensor Parallel, Expert Parallel, and a hybrid of the two. Tensor parallelism (TP) is a method for distributing a model’s computation across multiple GPUs by splitting tensors into non-overlapping pieces, which allows different parts of the tensor to be processed simultaneously on separate GPUs. Expert Parallelism (EP), on the other hand, distributes experts of an MoE across GPUs. We found that a hybrid mode for balancing

TP and EP can be $1.5\times$ faster than the original TP solution; see **Exp4** in §5 for details.

Model Quantization. During model inference, each parameter of the original LLM model is stored as a float number with 32-bit (fp32), resulting in significant GPU memory consumption and slower inference speeds. However, applying quantization techniques using 16-bit (fp16) and 8-bit (fp8) floating point numbers can substantially reduce memory usage and accelerate inference speeds, while maintaining nearly the same model accuracy as fp32 (Liu et al., 2024; Lin et al., 2024).

Continuous Batching and Batch Scheduler. To efficiently handle asynchronous user requests, we use a continuous batching strategy that batches requests for simultaneous processing by the engine. This method addresses variability in user input characteristics, such as input length, which can cause inefficiencies in static batching. Furthermore, our experiments with scheduling policies revealed that setting policy to max utilization, when in-flight sequence batching is enabled, significantly enhances GPU utilization by processing the maximum number of requests per iteration. However, this aggressive approach may require pausing requests if the KV cache size limit is reached, a trade-off to consider in production systems.

Other Optimizations. We adopt Flash Attention (Dao et al., 2022) for operator fusion and Paged Attention (Kwon et al., 2023) to boost the performance

4.2 Optimize Replica Router

To effectively manage high concurrent requests, the gateway must exhibit superior performance in handling extensive Network I/O, database I/O, and CPU-intensive operations, including authentication processes, routing algorithms, and token filtering for security purposes. The efficient execution of these resource-bound tasks is critical, as they significantly impact the system’s overall latency and throughput. Optimizing the gateway’s capacity to handle these diverse and demanding operations is essential for maintaining system performance and scalability under high-load conditions. To address these requirements, we replace the baseline router framework, which is based on FastAPI (Python), with Axum (Rust). In terms of transaction protocol, we migrate from HTTP/1.1 to the gRPC protocol. The architecture is shown in Figure 3.

CPU Bound Job Optimization. For CPU-bound jobs, the FastAPI gateway in the baseline imple-

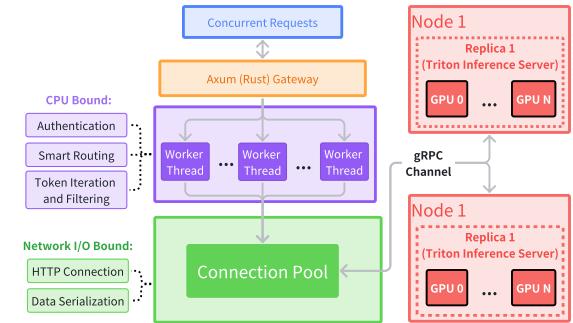


Figure 3: ScaleLLM Gateway Architecture

mentation is constrained by the Global Interpreter Lock (GIL), which limits its ability to utilize multiple CPU cores effectively. We refactor the gateway using Tokio (Lerche et al., 2017) for multi-task execution across multiple worker threads and Axum (Pedersen, 2021) for web development.

Network I/O Bound Job Optimization. We implement a gRPC connection pool based on Tonic (Franco, 2020), a robust and efficient gRPC framework. This approach allows new requests to reuse existing connection channels, thereby reducing connection establishment overhead. Additionally, by utilizing Protocol Buffers for data serialization, we further decreased associated costs.

4.3 Safety and Observability Module

ScaleLLM incorporates a comprehensive Safety Module that addresses key security concerns like user authentication, rate limiting, and sensitive content detection. Token-based authentication is securely managed in Redis to prevent unauthorized access. Rate limiting controls user requests to ensure fair usage, while advanced algorithms detect and filter harmful content for system integrity.

Additionally, the Observability Module tracks performance and operational metrics, which are stored locally to meet production compliance standards. This module enables detailed monitoring for analysis and troubleshooting. Rust’s multi-threading boosts performance by supporting efficient concurrent processing, minimizing latency, and optimizing high-traffic handling.

5 Experiments

Experimental settings. We employ 8 NVIDIA DGX H100 GPUs, connected via 18 NVLink links, each providing a bandwidth of 26.562 GB/s. We select Mixtral 8x7B (Jiang et al., 2024) as the inference LLM and set the maximum tokens generation

length to 512, the temperature to 0.5, and the top-p parameter to 0.7. We optimize the ScaleLLM engine based on TensorRT-LLM (NVIDIA). Our evaluations use OpenOrca dataset (Lian et al., 2023) that contains question-response pairs for LLMs, as well as predefined system prompts. We simulated the user’s behavior of submitting a prompt in OpenAI API format (OpenAI, 2024) to the system, in a concurrent and continuous manner. Figure 4 illustrates the typical lifecycle of concurrent requests in comparison to one request.

Compared Endpoints. We utilized several endpoints for comparisons, including *i) Huggingface Endpoint* that is deployed with Huggingface transformer (Wolf et al., 2019) and FastAPI gateway; *ii) vLLM Endpoint* that is deployed with vLLM (VLLM AI) and FastAPI gateway; and *iii) Fireworks and Together AI Endpoints* (Fireworks AI; Together AI).

5.1 Evaluation Metrics

We define metrics to evaluate the efficiency of LLM serving frameworks. To explain the definitions clearly, we illustrate different stages of LLM inference in Figure 4.

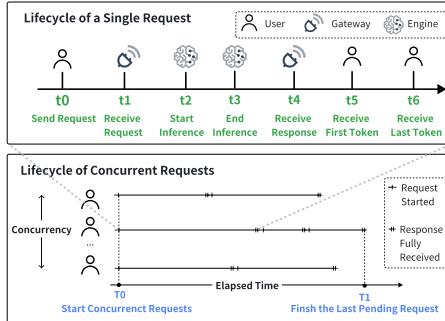


Figure 4: Lifecycle of Concurrent and Single Request

For the rest of §5.1, we denote t_0 as the timestamp the user submits a request, t_1 as the timestamp for the router to receive that request, t_2 as the start time for the engine’s local inference, t_3 as the engine finished the inference, t_4 as the time gateway received the response from engine, t_5 as the time for the user to receive the first token, and t_6 as the timestamp that they receive the full output.

of Concurrency Requests: The upper bound of the number of ongoing requests at a single moment.

of Requests: In order to fulfill the system during an elapsed time period, this number is set to be $20 \times c$ where c is the number of concurrency requests.

Average Latency: The average waiting time for a user to see the full output, computed as $t_5 - t_0$.

Gateway Latency: The time cost for processing and routing requests and LLM responses between the user and the inference engine, defined as $(t_2 - t_0) + (t_5 - t_3)$, where $t_2 - t_0$ is the time for processing and routing a user request to the inference engine, and $t_5 - t_3$ is the time for transferring the response from the engine to the user.

Engine Latency: The time for the engine to process a local inference, computed as $t_3 - t_2$.

Throughput: The number of tokens that the whole system generates within a certain time frame, computed as $\frac{N_t}{T_1 - T_0}$, where N_t is the number of generated tokens, T_1 is the timestamp to finish the last request, and T_0 is the time that the concurrent requests start.

Time to First Token (TTFT): The elapsed time between the user to submit a new request and to receive the first token, computed as $t_4 - t_0$.

Time Between Tokens (TBT): The average wait time to the next generated token after the first generated token, computed as $\frac{(N_g - 1)}{(t_6 - t_5)}$, where N_g is the number of generated tokens for one request.

5.2 Serving Performance Evaluation

We first provide the comparison with the state-of-the-art endpoints, then make a detailed comparison for non-streaming and streaming generation.

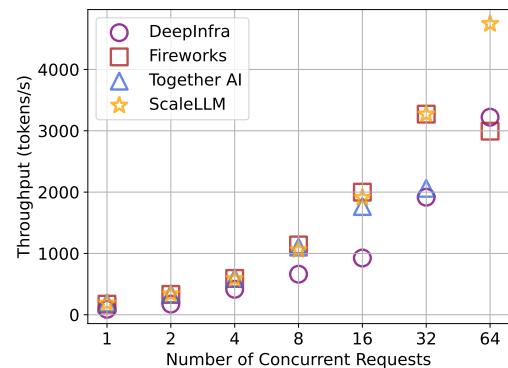


Figure 5: Endpoint Throughput Comparison.

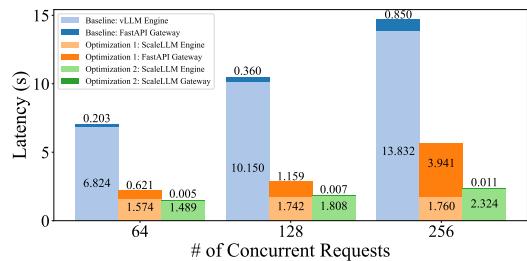
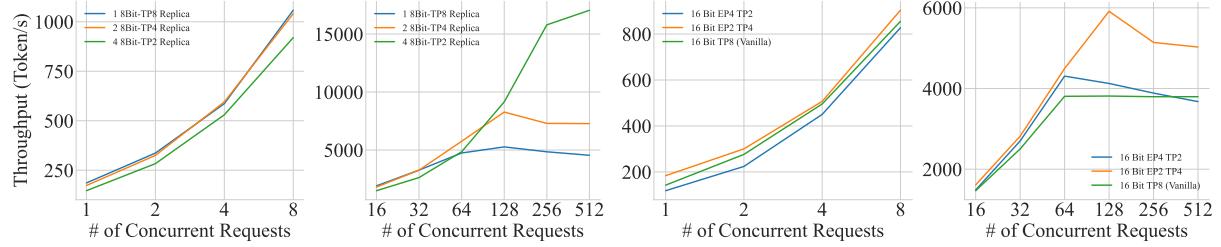


Figure 6: System latency vs # of concurrent requests.

Table 1: TTFT and TBT for end to end streaming requests on 2 H100s. Smaller TTFT means faster response for the first token and smaller TBT means faster generation of tokens. Timeout: 90% of the users’ requests cannot complete in 60s.

Concurrent Requests	Huggingface Endpoint		vLLM Endpoint		ScaleLLM	
	TTFT/ms	TBT/ms	TTFT/ms	TBT/ms	TTFT/ms	TBT/ms
1	315.6	83.4	48.4	16.5	25.0 (1.9x)	8.5 (1.9x)
2	637.2	218.3	51.9	16.7	25.3 (2.1x)	8.7 (1.9x)
4	1157.8	506.4	55.1	21.1	25.5 (2.2x)	10.4 (2.0x)
8	Timeout	Timeout	70.2	30.1	25.9 (2.7x)	12.2 (2.5x)
16	Timeout	Timeout	93.1	38.3	26.7 (3.5x)	13.4 (2.9x)
32	Timeout	Timeout	135.8	50.1	29.8 (4.5x)	14.6 (3.4x)
64	Timeout	Timeout	285.4	70.8	99.4 (2.9x)	16.5 (4.3x)



(a) Low conc. with 1-4 replica. (b) High conc. with 1-4 replica. (c) Low conc. with 1 replica. (d) High conc. with 1 replica.
Figure 7: Throughputs for different replica settings and varying # of concurrency (conc) requests for batch size 64.

Exp1. Endpoints Throughput Comparison. We compare the throughput of ScaleLLM against DeepInfra, Fireworks, and Together AI across different levels of concurrency. ScaleLLM utilizes 8 H100 GPUs for creating the endpoint. As shown in Figure 5, ScaleLLM performs comparably to other endpoints at lower concurrency levels. However, ScaleLLM significantly outperforms the endpoints as the concurrency scales up, and surpasses all other endpoints by a huge margin for batch size 64.

Exp2. Non-Streaming Generation Evaluation. We conducted a comprehensive latency breakdown evaluation for Mixtral 8x7B running on two H100 GPUs, examining various levels of concurrent requests. The averaged latency decomposition is shown in Figure 6. The result shows that with ScaleLLM, the engine latency is reduced compared to the baseline engine. However, at concurrency levels of 64/128/256, the baseline gateway latency increases when connected to the ScaleLLM Engine, compared to its connection with the Baseline engine, making it the new bottleneck. This is attributed to the baseline gateway’s inability to keep pace with the ScaleLLM Engine’s generation speed due to CPU bound task and Network I/O task as mentioned in §4.2. However, we observe a significant reduction in latency upon swapping the baseline gate-

ways out with ScaleLLM Gateway, indicating that ScaleLLM Gateway matches the engine’s generation speed, thereby shifting the bottleneck back to the engine. The result of the concurrency level from 1 to 32 is in Appendix §E.

Exp3. Streaming Generation Evaluation. To provide an intuitive perspective from the user’s point of view, we compared the time to the first token (TTFT) and the time between tokens (TBT) on ScaleLLM with Huggingface Transformer and vLLM. In order to simulate the realistic user’s waiting threshold, we set the timeout of generating all the tokens to be 60 seconds. The results in Table 1 show that the HuggingFace Endpoint has the highest TTFT and TBT, where over 90% of the user’s requests get timeout after 60 seconds when the concurrency is 8. On the contrary, vLLM has lower TTFT and TBT but ScaleLLM improved over 1.9× lower TTFT and TBT compared with the vLLM Endpoint.

Exp4. Parallelism Comparisons. We experiment with replicas and computations parallelism. For computation parallelism, we test three combinations: Vanilla Tensor Parallelism 8, MOE Expert Parallelism 4 with Tensor Parallelism 2, and MOE Expert Parallelism 2 with Tensor Parallelism 4. We present results in Figure 7 and explain in details in Appendix §B and §C.

6 Blueprint Architecture of Dynamic Inference Load Balancing System

Our experiments have revealed that different engine parameters are suited for different throughput loads, thereby emphasizing the need for a dynamic load balancing system for AI inference unifying the strengths of these heterogeneous configurations and averaging out weakness. We propose a blueprint for such a dynamic inference load balancing system, designed to optimize resource allocation by efficiently distributing inference requests across these heterogeneous replicas, thereby maintaining consistently high throughput regardless of the concurrency scale.

The core component of the proposed system is a dynamic inference balancing router that handles incoming inference requests and intelligently routes them to the appropriate replica based on a routing policy, mapping request concurrency levels to throughput ranges and selecting the replica best suited to manage the specific workload range.

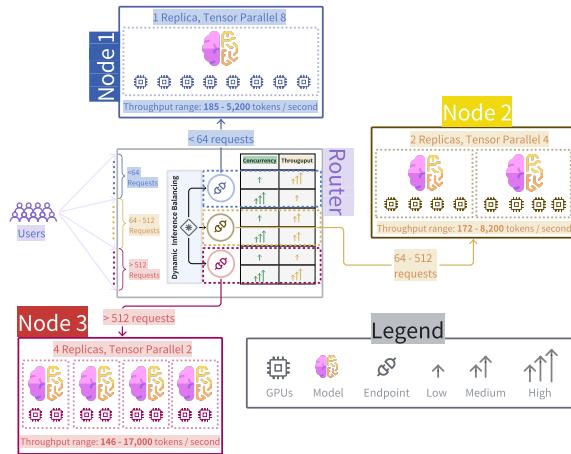


Figure 8: Blueprint Architecture of Dynamic Inference Load Balancing System.

The dynamic routing policy illustrated in Figure 8, showcasing the blueprint architecture and the policy breakdown follows a general rule of thumb:

Low concurrency (< 64 requests). Route requests to nodes with fewer replicas but higher tensor parallelism to optimize resource utilization for smaller batch computations.

High concurrency (≥ 64 requests). Route requests to nodes with more replicas but lower tensor parallelism, effectively distributing the workload to squeeze everything out of available compute by leveraging the power of replica parallelism.

7 Discussion

7.1 Serving Any LLMs

The paper primarily uses the Mistral 8x7B model as an example of the serving engine to demonstrate its effectiveness. The reason is that Mistral 8x7B features 8 experts and utilizes 2 experts per token generation, making it more complex and suitable for showcasing the effectiveness of the proposed optimizations. However, the methodology of ScaleLLM is not limited to this specific model. In addition to the optimization on the mixture of experts, the optimizations on Gateway and Serving Engine can be applied to any LLMs, and the framework is designed with generalizability in mind.

7.2 Serving Cost Analysis

The system is deployed on an industry cloud platform with H100 GPUs, where the current market price for a dedicated H100 GPU is \$2.2 per hour. ScaleLLM achieves a 4.3 \times speed-up in throughput compared to vLLM while using the same 8 H100s, which means that ScaleLLM can save 4.3 \times computation cost for the same system throughput. Dynamic pricing on volatile instances can also be an interesting direction for future research.

7.3 Fault Tolerant

ScaleLLM's architecture incorporates replica-level load balancing and dynamic routing, ensuring inherent fault tolerance. This design minimizes service degradation and supports seamless recovery from replicas or infrastructure failures. The Gateway's gRPC channels employ timeouts and error codes to detect replica failures, triggering the reconciler to initiate recovery actions such as restarting components or migrating them to other nodes. Further exploration of failure scenarios presents a promising avenue for future security research.

8 Conclusion and Future work

In this paper, the proposed ScaleLLM framework optimizes both the LLM serving engine and the platform. As LLM applications grow in complexity, platform latency becomes increasingly critical. Instead of focusing solely on local inference speed, industrial research should prioritize reducing end-to-end latency by streamlining the serving gateway and optimizing the platform-level performance. LLM can also be deployed in a federated way to further reduce the latency (Yao et al., 2024).

References

- Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, 35:16344–16359.
- Fireworks AI. Fireworks ai. <http://fireworks.ai>. Accessed: 2024-07-16.
- Lucio Franco. 2020. Tonic: 0.1 has arrived! <https://luciofran.co/tonic-0-1-release/>.
- Albert Q Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, et al. 2024. Mixtral of experts. *arXiv preprint arXiv:2401.04088*.
- Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 611–626.
- Marina Lammertyn. 2024. 60+ chatgpt statistics and facts you need to know in 2024. <https://blog.invgate.com/chatgpt-statistics>.
- Carl Lerche, Alex Crichton, and Aaron Turon. 2017. Announcing tokio 0.1. <https://tokio.rs/blog/2017-01-tokio-0-1>.
- Wing Lian, Bleys Goodson, Eugene Pentland, Austin Cook, Chanvichet Vong, and "Teknium". 2023. Openorca: An open dataset of gpt augmented flan reasoning traces. <https://huggingface.co/OpenOrca/OpenOrca>.
- Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. 2024. Awq: Activation-aware weight quantization for on-device llm compression and acceleration. *Proceedings of Machine Learning and Systems*, 6:87–100.
- Zirui Liu, Jiayi Yuan, Hongye Jin, Shaochen Zhong, Zhaozhuo Xu, Vladimir Braverman, Beidi Chen, and Xia Hu. 2024. Kivi: A tuning-free asymmetric 2bit quantization for kv cache. *arXiv preprint arXiv:2402.02750*.
- NVIDIA. Tensorrt-llm. <https://github.com/NVIDIA/TensorRT-LLM>. Accessed: 2024-07-16.
- OpenAI. 2024. Openai api. <https://platform.openai.com/docs/api-reference/introduction>.
- David Pedersen. 2021. Announcing axum. <https://tokio.rs/blog/2021-07-announcing-axum>.
- Together AI. Together ai. <http://together.ai>. Accessed: 2024-07-16.
- Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.
- VLLM AI. Vllm ai. <http://vllm.ai>. Accessed: 2024-07-16.
- Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pieric Cistac, Tim Rault, R'emi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2019. Huggingface's transformers: State-of-the-art natural language processing. *Preprint*, arXiv:1910.03771.
- Yuhang Yao, Jianyi Zhang, Junda Wu, Chengkai Huang, Yu Xia, Tong Yu, Ruiyi Zhang, Sungchul Kim, Ryan Rossi, Ang Li, et al. 2024. Federated large language models: Current progress and future directions. *arXiv preprint arXiv:2409.15723*.

A User-Oriented Metrics

From the user’s point of view, the metrics to measure an inference system are close to their intuitive feeling, which is the following four metrics:

- Throughput: The number of output tokens per second that an inference server can generate across all users and requests.
- Latency: The overall time for a user to get the full response from the system.
- Time to first token (TTFT): For streaming response format, the time that the user receives the first generated token.
- Time between tokens (TBT): After the first token gets generated, the time of generating each following token.

B Experiment results for Tensor Parallel and Expert Parallel for Mixture of Experts LLMs.

We conducted a series of experiments to assess the performance of a variety of computation parallelism techniques as depicted in Figure 9. The tested configurations, using a Mixtral 8x7B model include: *i*) Vanilla Tensor Parallelism 8 (TP8) *ii*) MOE Expert Parallelism 4 (EP4) with Tensor Parallelism 2 (TP2); and *iii*) MOE Expert Parallelism 2 (EP2) with Tensor Parallelism 4 (TP4)

Our findings illustrated in Figure 7c and 7d indicate that MOE-EP2-TP4 consistently outperformed all other methods across the entire concurrency spectrum, demonstrating a particularly significant advantage at higher concurrency levels, specifically beyond 128 concurrent requests. While TP8 showed superior performance compared to MOE-EP4-TP2 at lower concurrency levels, it was eventually surpassed by MOE-EP4-TP2 as concurrency increased beyond 16 requests.

These results underscore the effectiveness of MOE-EP2-TP4 in managing high-concurrency scenarios, establishing it as the optimal configuration for deployments intended to handle large-scale concurrency.

C Throughput for different replica settings and varying # of concurrency requests for batch size 64.

In our study, we evaluated the impact of combining replica parallelism with tensor parallelism to provide a thorough assessment of performance under different parallelism strategies. Specifically,

we tested the following configurations using an 8-bit quantized Mixtral 8x7B model: *i*) One replica with Tensor Parallelism 8 (TP8), utilizing 8 GPUs for a single replica *ii*) Two replicas with Tensor Parallelism 4 (TP4), utilizing 4 GPUs per replica; and *iii*) Four replicas with Tensor Parallelism 2 (TP2), utilizing 2 GPUs per replica. These configurations were chosen to equalize the utilization of the computational resource for each setup, ensuring a comprehensive but fair evaluation.

As illustrated in Figure 7a, at lower concurrency levels, fully utilizing the available compute for tensor parallelism, without any replica parallelism demonstrates superior performance compared to configurations combining tensor and replica parallelism. However, as shown in Figure 7b, the trend shifts significantly at higher concurrency levels, favoring configurations with higher degrees of replica parallelism. Notably, the configuration with four replicas and Tensor Parallelism 2 (TP2) significantly outperforms both the two-replica TP4 and single-replica TP8 configurations. Specifically, the four-replica TP2 setup achieves markedly high throughput as the concurrency level exceeds 128 requests while the single-replica TP8 configuration exhibits the poorest performance. The two-replica TP4 configuration shows a modest improvement over the single-replica TP8 configuration. This study highlights the importance of replica parallelism for handling high concurrency levels, and conversely, highlights the effectiveness of tensor parallelism at lower concurrency levels.

D Throughput vs # of concurrency requests.

We evaluated the throughput differences between the ScaleLLM Engine and the vLLM Engine, as well as their integration with the FastAPI Gateway and the optimized ScaleLLM Gateway. The complete results (with Concurrency from 1 to 256) are illustrated in Figure 10. The findings indicate that engine optimization leads to significant improvements in throughput; Additionally, the optimization of the Gateway contributes to further notable performance enhancements, demonstrating the cumulative impact of both engine and gateway optimizations on overall system performance.

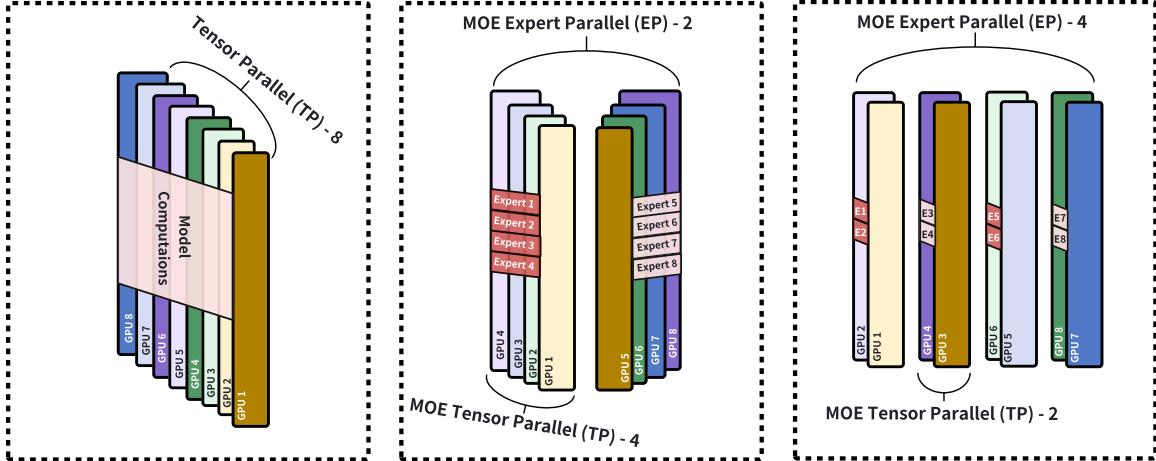


Figure 9: Tensor Parallel and Expert Parallel for Mixture of Experts LLMs.

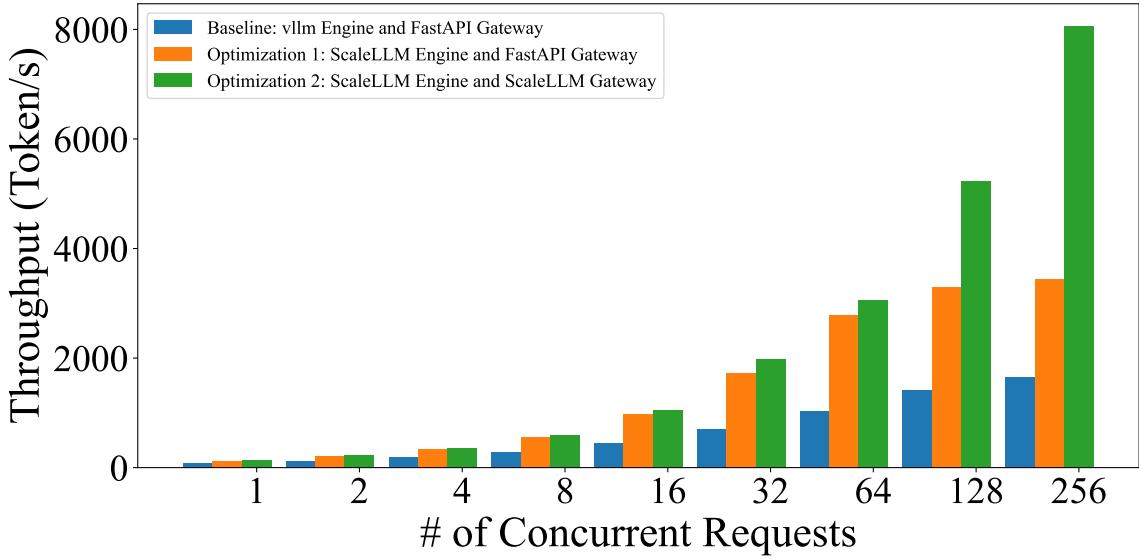


Figure 10: Throughput vs # of concurrent requests.

E Comprehensive Result of System latency vs # of concurrency requests.

Building upon our findings presented in §3.2, where we discussed the latency characteristics of the Gateway, we now provide a more comprehensive examination of this phenomenon. Figure 11 illustrates a detailed analysis of the relationship between system latency and the number of concurrent requests. Our results demonstrate a notable trend: the Gateway’s latency increases substantially when the number of concurrent requests exceeds 32. This observation provides crucial insights into the system’s performance characteristics and scalability limitations.

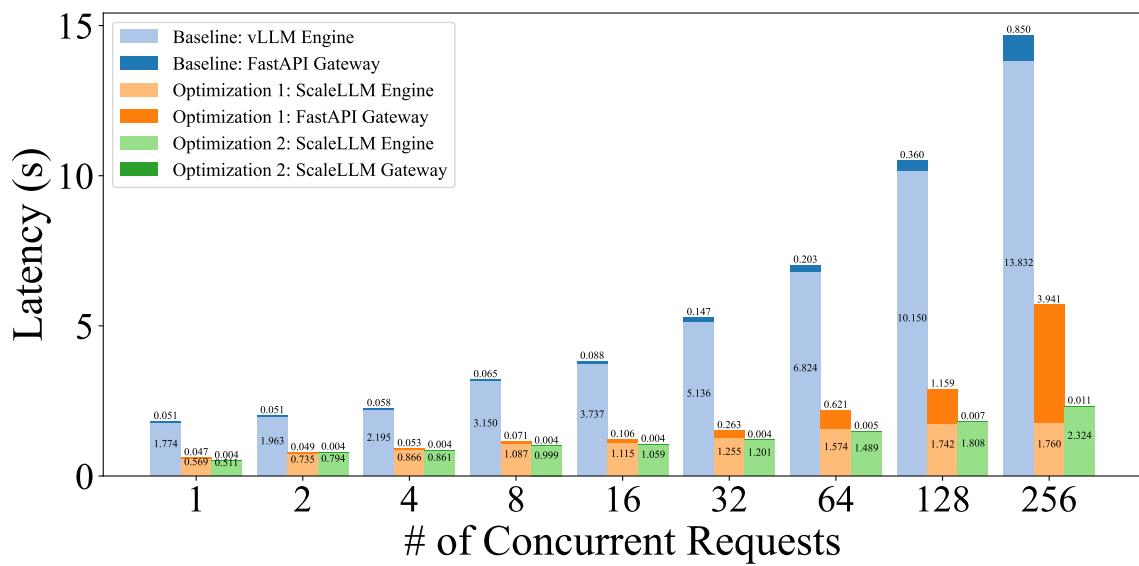


Figure 11: System latency vs # of concurrent requests.