sLLM: Accelerating LLM Inference using Semantic Load Balancing with Shared Memory Data Structures

anonymous authors

***Abstract*—As Large Language Models (LLMs) are increasingly deployed to support a broad spectrum of applications, enhancing inference efficiency and minimizing costs have become critical areas of focus. To address these challenges, researchers have explored optimizing the Key-Value (KV) cache within LLMs. However, existing approaches have not considered the potential benefits of sharing KV caches across multiple requests in a cluster environment. Addressing this gap, we introduce sLLM, a novel system that integrates an efficient shared-memory-based Semantic Load Balancer with a KV cache sharing mechanism. This design significantly reduces the need for recomputation during LLM inference, which enhances inference performance. Our evaluation of the sLLM system showcases its effectiveness: the Semantic Load Balancer achieves up to a $7\times$ reduction in latency when dispatching requests, while the system as a whole can decrease the Time-To-First-Token (TTFT) for LLM inferences by $30-58\%$.**

***Index Terms*—component, formatting, style, styling, insert**

I. Introduction

Transformer-based Large Language Models (LLMs) have gained significant attention due to their ability to generate human-like text. These models, trained on vast datasets, are transforming a wide variety of applications such as content creation, coding, personal assistant, and more [1], [2]. As LLMs are deployed in production to support real-world use cases, the efficiency and performance of LLM inference become critical for operation cost and user experience.

In many applications of LLM today, the questions given to the LLMs, often referred to as "prompts", are made using a set of standard templates. These prompt templates are used repeatedly for various requests. These templates provide a consistent structure to the prompts to ensure accurate interpretation and response, which is crucial for production applications. It has been found that using these standard prompt templates helps the LLM perform better for a wide variety of applications such as healthcare, robotics, marketing, and computer networking. [3]–[5]

The repeated use of templates or entire prompts in different requests gives rise to the following question: *Can we reduce the unnecessary re-computation across requests during model inference to reduce the inference latency and cost?*

Previous research explored various methods to reduce redundant computation for LLM inference [6], [7]. One popular approach is the use of a Key-Value (KV) cache. Most LLMs are autoregressive, where one token is generated at a time based on previous tokens. Here a token is a word, part of a word, or even punctuation. The KV cache technique involves storing and reusing the key-value attention states of known tokens to reduce the need of having to recompute the attention states of all the tokens when generating the next new token. Existing works have looked into optimizing the KV cache to improve performance by using techniques such as paged attention [7] and compression [8], [9]. However, these works only consider leveraging the KV cache for a single LLM inference request in a single machine. In other words, the KV cache is kept only during the inference of a single request and destroyed once the request is completed. In a cluster environment, these inference servers work individually and do not collaborate and share the KV cache. As a result, existing techniques do not consider leveraging the shared text between requests in a cluster environment to further improve inference performance.

To this end, we propose sLLM, a system that aims to reduce the inference latency in large-scale clusters by reusing KV cache across multiple requests. sLLM achieves this by efficiently and semantically load-balancing the inference request to the most appropriate inference server. Each inference server process uses a shared KV cache that is kept across requests to minimize the recomputation of the key value attention states of previously seen tokens. One of the key challenges here is the performance of the semantic load balancer. A semantic load balancer does not simply forward the request to a server, but instead also looks into the data to decide the best server selection. Thus it is more challenging to maintain high through and low latency. To tackle this challenge, we build the semantic load balancer on top of lock-free shared memory data structures, to allow multiple load balancer processes to operate in parallel with high-performance read/write access to the shared forwarding information.

The sLLM system is implemented using Rust and Python and has undergone extensive testing on a server cluster with real-world traces. The evaluation results demonstrate the high performance of the system. Compared to utilizing Redis for in-memory caching of forwarding information in load balancers, our semantic load balancer achieves 6X higher throughput and 7X lower latency. Overall, by sharing the KV cache across multiple requests, our sLLM system significantly reduces the time-to-first-token(TTFT) inference latency by 30-58%.

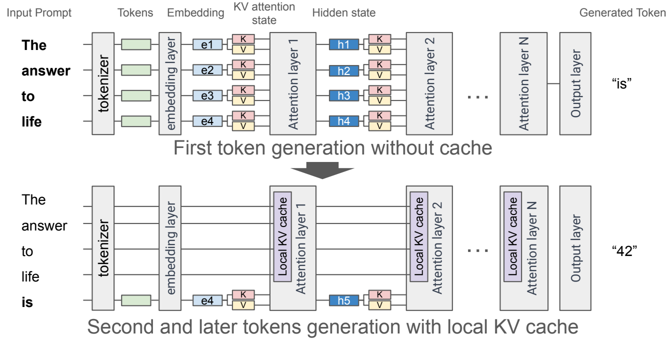

Fig. 1. Operation of traditional autoregressive transformers

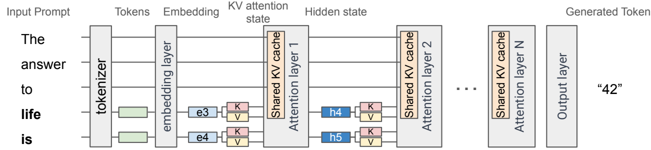

Fig. 2. Ours: Operation of transformers with shared KV across requests

II. Background and Related Works

A. LLM and Transformer Basics

Most of the popular LLM models, such as GPT [10], LLama [11], OPT [12], are built based on the transformer architecture [13]. At its core, the transformer model relies on a mechanism known as "attention", which enables the model to focus on different parts of the input sequence when producing an output, thereby capturing the context more effectively. During the inference process of an LLM model, the input prompt is first tokenized into tokens, each representing a word, part of a word, or punctuation. These tokens each go through an embedding layer to create a word embedding. Next, embeddings (including positional encoding) would be used as input to the LLM's attention layer, where each embedding is sent through three separate networks (often with a fully connected layer) to obtain a key, value, and query. We then compute the dot product of each query and its preceding token's keys to obtain the "importance" of each of the preceding tokens. Subsequently, the output is normalized and multiplied by the values of all the preceding tokens to obtain the output of the attention layer. Notice that an LLM model often contains multiple attention layers, thus the output of an attention layer is used as an input to the next attention layer. Then last transformer layer output is processed to complete the generation of the next token. Figure 1 shows the high-level ideas of this inference procedure.

B. Autoregressive Process

From the above description, we can see that a single pass through a transformer generates one token at a time. To generate a complete response from the input prompt, modern LLM models adopt an autoregressive process, which generates text one token at a time based on all the preceding tokens (including both input and generated tokens). This process is run sequentially until the max number token is reached or an end-of-sequence token is generated.

C. KV Cache

As a result, the autoregressive process requires us to execute the transformer multiple times to obtain the final response text, which involves a significant amount of recomputation. To reduce recomputation, researchers created the Key-Value (KV) cache to store the attention states. The KV cache essentially stores the key and value vectors of the processed token, so when the LLM model generates the next token, it does not need to compute the key and value vectors of the previously seen tokens. More specifically, when the LLM receives an input sequence of tokens and starts to generate the first new token, the LLM model needs to compute the key, value, and query of all the tokens in all attention layers. Then based on the autoregressive process, when the LLM generates the second token, it only needs to compute the key and value vector of the last generated token, because the keys and values of the other tokens can be retrieved directly from the KV cache. Figure 1 shows the process of the first token generation and the second token generation in an autoregressive manner. Once the LLM model completes the token generation for one request, it usually discards the KV cache.

To accelerate LLM inference, researchers have looked into various ways to optimize the KV cache. [7] tries to reduce the memory footprint of the KV cache on GPUs. It uses paging techniques to improve memory management and reduce internal and external fragmentation of KV cache. [8] tries to improve inference performance by pruning non-important entries in the KV cache. It proposes a KV eviction policy to keep the KVs of the recent and most important tokens. [14] and [9] also explore ways to compress the KV cache with pruning techniques. Most of these works focus on optimizing the KV cache of a single server to improve inference performance. This study focuses on improving the inference performance of an inference server cluster by efficiently load-balancing the request across the servers.

D. LLM Inference Cluster Deployment

To deploy LLM or Machine Learning inference in a cluster, it is a common practice to run multiple inference servers with a load balancer in front of them to distribute the incoming requests among the inference servers [15]–[17]. HA proxy [18] and Nginx [19] are some of the popular choices for load balancers. Some systems, such as [16], also develop their specialized load balancing to support application requirements. Although many of these load balancers are highly efficient, they only support simple load balancing strategies, such as round robin, least request scheduling, and batching. These strategies do not look into the content of the request itself, so they can not load balance the requests based on the content and the conditions of the KV caches of the inference servers. In our work, we aim to build a semantic load-balancing solution

that is aware of the incoming prompt and also highly efficient and scalable.

III. Method

In this section, we provide the detailed design of our sLLM system. We will first introduce the high-level system design, the KV cache sharing mechanism, followed by the internal design of the semantic load balancer.

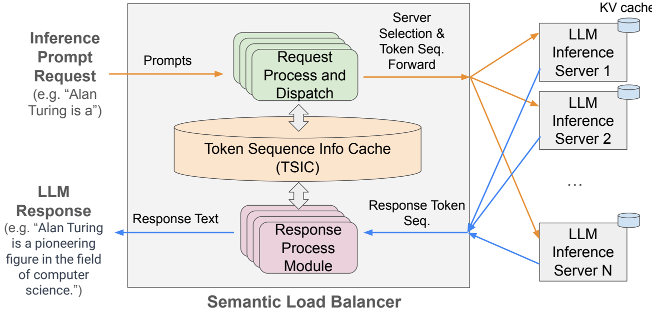

Fig. 3. sLLM High-level System Design Diagram

A. System Design

The high-level system design of the sLLM system is shown in Figure 3. The system contains two types of components:

- **Semantic Load Balancer**: This component acts as the initial recipient of inference requests. Its primary function is to analyze and direct each request to the most appropriate LLM inference server. The selection criterion is based on the availability of relevant cached keys and values attention states that match the request's context.
- **LLM Inference Servers**: These servers are the workers of the system. Upon receiving a request from the Semantic Load Balancer, an inference server executes its LLM model to process the request and generate response tokens. During this process, it utilizes and updates a KV cache. This KV cache is shared across multiple requests and multiple inference processes.

In practice, the workflow begins with the inference request being sent to the Semantic Load Balancer. After processing and determining the most suitable LLM inference server (based on the cache state), the request is forwarded. The chosen server then executes the LLM model, generating a response while simultaneously updating the shared memory KV cache with new attention states. Finally, the response is relayed back to the Semantic Load Balancer, which in turn delivers it to the user. The Semantic Load Balancer also caches the response from the inference server to facilitate future lookups. This architecture is able to leverage the KV caches residing in the inference servers to reduce the inference latency by forwarding the request to the most appropriate inference server.

B. KV Cache Sharing Mechanism

One of the main requirements is to share the KV cache across multiple requests. We achieve this by searching for the long prefix match. More specifically, for a given new input sequence of tokens, we aim to find the cached token sequence that has the longest prefix match to the input sequence. Then we leverage the cache of the longest prefix to reduce the amount of recomputation. With templated prompts [3], [20], the amount of prefix matches across prompts is significant and thus this method has a substantial impact on reducing the inference latency. Compared to techniques such as KV cache compression where the optimization methods change the model output, our solution in fact keeps the output of the model exactly the same, so it is applicable to a wide variety of models. Figure 2 shows how the transformer works when the keys and values of some prefix tokens are cached.

C. Semantic Load Balancer

As shown in Figure 3, the Semantic Load Balancer contains mainly three components:

- **Token Sequence Info Cache (TSIC)**: This is a shared memory cache that stores mapping from a token sequence to an LLM server that has the KV cache of that token sequence. This module supports the lookup of the longest prefix match mechanism mentioned above, and provides high throughput and low latency read/write capability to support the other two modules in the Semantic Load Balancer.
- **Request Process and Dispatch Module**: This module processes the prompt input, tokenizes the prompt, and accesses the TSIC to look for the inference server that has the longest prefix match token KV cache. Then it forwards the token sequence to the corresponding server.
- **Response Process Module**: This module receives the response token from the LLM inference servers, converts it into text, and sends the response back to the user. At the same time, it updates the TSIC to reflect the cached token sequence of the response LLM server.

During operation, user input prompts are sent to the Request Process and Dispatch Module to make inference server selection decisions by reading the TSIC. After token generation at the selected LLM server, the responses are transmitted to the Response Process Module, which writes to the TSIC module and returns the response to the user.

One aspect to emphasize in this design is the support of parallelism execution of both the Request Process and Dispatch module and Response Process module. The TSIC supports high-throughput and low-latency concurrent read/write access from multiple processes. Notice that the TSIC is not a traditional data store or database. Instead, it uses shared memory techniques to ensure the high performance of the load balancer. With such a design, all read and write accesses to the TSIC involve only memory access without any Inter Process Communication (IPC). More implementation details of the TSIC is provided in the next Section.

IV. Implementation

In this section, we provide the implementation details of the sLLM system. The sLLM system is implemented using Rust and Python to support both high-performance execution and compatibility with machine learning packages.

Next, we will separately introduce the implementation details of the TSIC, Request Process and Dispatch module, and the Response Process module.

A. Implementation of TSIC

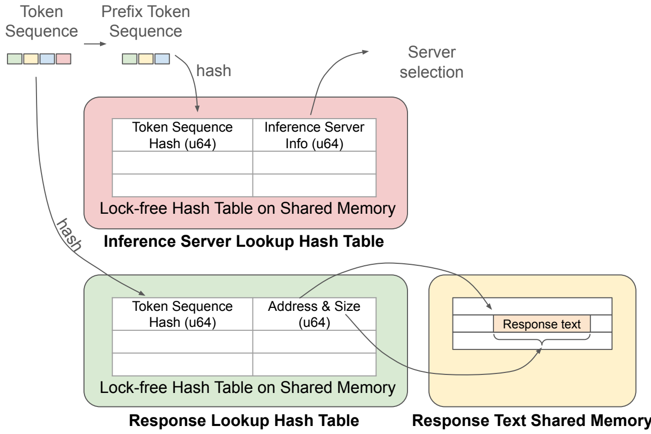

Fig. 4. Implementation of TSIC

To support high throughput and low latency concurrent read/write operation, TSIC is designed using lock-free data structures built on top of shared memory provided by the Operating System. More specifically, we implemented a lock-free hash table [21] on top of shared memory to support a fast concurrent look-up and update based on a given token sequence. Locking mechanism is a popular method used by developers to control the concurrent access to a shared piece of memory to ensure data integrity and prevent issues like race conditions. However, locking mechanisms have limitations like deadlock risk and performance issues in high contention, making them unsuitable for highly concurrent real-time applications. Lock-free data structure, on the other hand, combines Operating System support and algorithm design to provide an efficient way for concurrent read/write access with high throughput and low latency.

The detailed implementation of the TSIC is shown in Figure 4. It contains two lock-free hash tables with shared memory and one regular shared memory. The Inference Server Lookup Hash Table maps from the hash value of a token sequence to the info (e.g. hostname or IP address) of an inference server that has the corresponding KV cache to the token sequence. The Response Lookup Hash Table is designed to respond to prompts that have an exact match to the previously seen prompts. The goal is to directly reply with the response text to the user without having to forward the request to an inference server. It maps the hash of a token sequence to a memory address and the size of the response. The memory address points to a location in the Response Text Shared Memory that stores the response text.

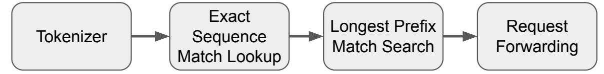

Fig. 5. Execution Pipeline of Request Process and Dispatch Module

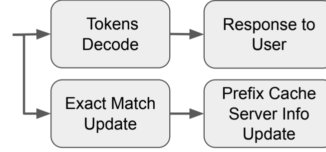

Fig. 6. Execution Pipeline of Response Process Module

B. Implementation of Request Process and Dispatch module

With the concurrent read access support from TSIC, the Request Process and Dispatch module follows the flow in Figure 5. The prompt is first tokenized, and then we search for an exact match in the Response Lookup Hash Table. If not found, the Prefix Search module tries to find the longest prefix match token sequence. If a match is found, the request is forwarded to the server with the corresponding KV cache for token generation. A basic longest prefix search algorithm is used in our design. It starts with the first token and gradually increases the number of tokens and checks if there is a match until no match is found.

C. Implementation of Response Process module

The processing pipeline of the Response Process is provided in Figure 6. After receiving the generated token sequence from an inference server, the Response Process first decodes the tokens into text and replies to the user. In parallel, it adds the response tokens to the Response Text Shared Memory and updates the info to the Response Lookup Hash Table. Then it also updates the Inference Server Lookup Hash for all the prefixes.

V. Evaluation

To evaluate the sLLM system, we mainly consider two areas: 1. the performance of the Semantic Load Balancer and; 2. the LLM inference latency.

A. Experiment Setup

The sLLM is evaluated with the OPT 2.7B model [12] with the Alpaca dataset [22]. The experiments are done in a compute cluster. The inference servers are deployed on Ubuntu servers with Intel Xeon Gold 5218 CPU and 192G of RAM. The Semantic Load Balancer is deployed on an AMD Ryzen 5 Server with 32G of RAM. To simulate a large number of inference servers, we use profiling traces collected from running the inference tasks on the Intel Xeon servers. These

traces allow us to test the performance of the Semantic Load Balancer on a large scale.

B. Semantic Load Balancer

We evaluate the performance of the Semantic Load Balancer in terms of its throughput and latency.

Baseline - Redis-based Semantic Load Balancing: We compare our Semantic Load Balancer to a functionally similar load balancing solution that is implemented using Redis [23] as the in-memory caching layer. We select Redis because it is a popular choice for efficient in-memory cache.

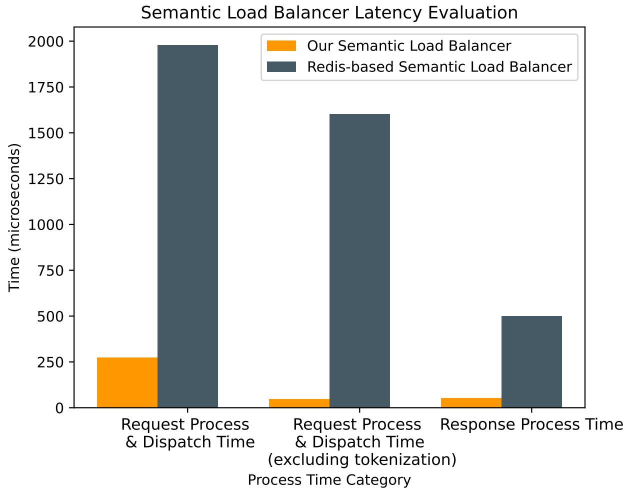

Fig. 7. Semantic Load Balancer Latency Evaluation

Figure 7 shows the latency of different parts of our Semantic Load Balancer versus the Redis-based Semantic Load Balancer. Our implementation is able to achieve a significant speedup compared to the Redis-based solution. When considering the Request Process and Dispatch pipeline, our solution is over 7X faster (30x faster if not considering the tokenization step) compared to the baseline. This demonstrates the efficiency and low overhead of our lock-free shared-memory Semantic Load Balancer.

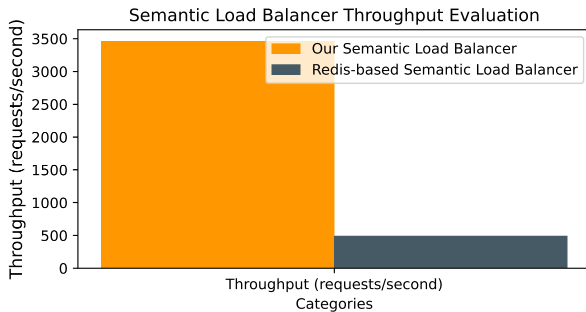

Fig. 8. Semantic Load Balancer Throughput Evaluation

Figure 8 shows the throughput of our Semantic Load Balancer compared to the baseline. Our implementation is able to achieve over 6x higher throughput compared to the baseline, showing the ability of the load balancer to support a large number of inference servers efficiently.

C. LLM inference latency

For the evaluation of the overall LLM inference latency, we focus on the metric of Time-To-First-Token (TTFT), which is defined as the time between when the input prompt is received and the generation of the first token. TTFT reflects the responsiveness of the system as it measures the wait time from the user's input and the first output, thus it has a significant impact on the user experience. [24]

Baseline - Basic Inference Cluster: We compare sLLM inference time with the case where the KV cache is not shared across requests, and a basic load balancer is used.

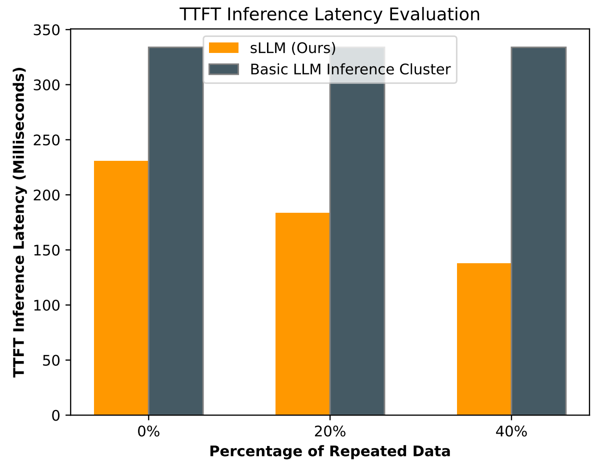

Fig. 9. TTFT Inference Latency under Different Percentages of Repeated Prompts

To evaluate the inference latency under different usage patterns, we augment the Alpaca dataset with different percentages of repeated prompts (i.e. identical prompts). Repeated prompts are realistic as prompts are commonly reused in real-world applications. The Percentage of repeated prompts is defined as the percentage of prompts in the dataset that have one or more previous occurrences. Figure 9 shows the TTFT latency under different percentages of repeated prompts. With our system that enables KV cache sharing across prompts, it is able to reduce the inference latency by around 30% even if no prompts in the dataset are exactly the same. With more repeated prompts, we can further reduce the latency as the cached response text can be replied directly to the user without running inference.

VI. Conclusion

In conclusion, the sLLM system provides a method for improving LLM inference for real-world applications. By combining a Semantic Load Balancer with a shared Key-Value (KV) cache mechanism, sLLM improves inference efficiency and reduces computational costs in a cluster environment. Our evaluations underscore sLLM's ability to dramatically reduce latency and Time-To-First-Token (TTFT).

REFERENCES

[1] "How generative ai is changing creative work." [Online]. Available: https://hbr.org/2022/11/how-generative-ai-is-changing-creative-work

[2] "Chatgpt."

[3] D. McDuff, M. Schaekermann, T. Tu, A. Palepu, A. Wang, J. Garrison, K. Singhal, Y. Sharma, S. Azizi, K. Kulkarni *et al.*, "Towards accurate differential diagnosis with large language models," *arXiv preprint arXiv:2312.00164*, 2023.

[4] S. Vemprala, R. Bonatti, A. Bucker, and A. Kapoor, "Chatgpt for robotics: Design principles and model abilities," *Microsoft Auton. Syst. Robot. Res*, vol. 2, p. 20, 2023.

[5] J. Lin, K. Dzeparoska, A. Tizghadam, and A. Leon-Garcia, "Appleseed: Intent-based multi-domain infrastructure management via few-shot learning," in *2023 IEEE 9th International Conference on Network Softwarization (NetSoft)*. IEEE, 2023, pp. 539–544.

[6] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-lm: Training multi-billion parameter language models using model parallelism," *arXiv preprint arXiv:1909.08053*, 2019.

[7] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with pagedattention," in *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023, pp. 611–626.

[8] Z. Zhang, Y. Sheng, T. Zhou, T. Chen, L. Zheng, R. Cai, Z. Song, Y. Tian, C. Ré, C. Barrett *et al.*, "H _2 o: Heavy-hitter oracle for efficient generative inference of large language models," *arXiv preprint arXiv:2306.14048*, 2023.

[9] S. Ge, Y. Zhang, L. Liu, M. Zhang, J. Han, and J. Gao, "Model tells you what to discard: Adaptive kv cache compression for llms," *arXiv preprint arXiv:2310.01801*, 2023.

[10] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[11] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar *et al.*, "Llama: Open and efficient foundation language models," *arXiv preprint arXiv:2302.13971*, 2023.

[12] S. Zhang, S. Roller, N. Goyal, M. Artetxe, M. Chen, S. Chen, C. Dewan, M. Diab, X. Li, X. V. Lin *et al.*, "Opt: Open pre-trained transformer language models," *arXiv preprint arXiv:2205.01068*, 2022.

[13] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[14] Z. Liu, A. Desai, F. Liao, W. Wang, V. Xie, Z. Xu, A. Kyrillidis, and A. Shrivastava, "Scissorhands: Exploiting the persistence of importance hypothesis for llm kv cache compression at test time," *arXiv preprint arXiv:2305.17118*, 2023.

[15] "Kserve." [Online]. Available: https://kserve.github.io/website/latest

[16] "Ray serve." [Online]. Available: https://docs.ray.io/en/latest/serve/index.html

[17] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica, "Clipper: A {Low-Latency} online prediction serving system," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017, pp. 613–627.

[18] "Haproxy." [Online]. Available: https://www.haproxy.org/

[19] "Nginx." [Online]. Available: https://www.nginx.com/

[20] G. Li, H. A. A. K. Hammoud, H. Itani, D. Khizbullin, and B. Ghanem, "Camel: Communicative agents for" mind" exploration of large scale language model society," *arXiv preprint arXiv:2303.17760*, 2023.

[21] M. M. Michael, "High performance dynamic lock-free hash tables and list-based sets," in *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, 2002, pp. 73–82.

[22] R. Taori, I. Gulrajani, T. Zhang, Y. Dubois, X. Li, C. Guestrin, P. Liang, and T. B. Hashimoto, "Alpaca: A strong, replicable instruction-following model," *Stanford Center for Research on Foundation Models. https://crfm. stanford. edu/2023/03/13/alpaca. html*, vol. 3, no. 6, p. 7, 2023.

[23] "Redis." [Online]. Available: https://redis.io/

[24] Z. Lew, J. B. Walther, A. Pang, and W. Shin, "Interactivity in online chat: Conversational contingency and response latency in computer-mediated communication," *Journal of Computer-Mediated Communication*, vol. 23, no. 4, pp. 201–221, 2018.