THE UNIVERSITY OF CHICAGO

A SCALABLE APPROACH TO DISTRIBUTED LARGE LANGUAGE MODEL INFERENCE

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY
YIHUA CHENG

CHICAGO, ILLINOIS
MARCH 2025

CONTENTS

LIST OF FIGURES v

LIST OF TABLES vii

ACKNOWLEDGMENTS viii

ABSTRACT x

1 INTRODUCTION 1
- 1.1 Distributed LLM serving: Opportunities and challenges 2
  - 1.1.1 Observed opportunities 2
  - 1.1.2 Key challenges 2
- 1.2 Limitations of Prior Approaches 3
- 1.3 Our Technique: Decoupling KV Cache from LLMs 4
- 1.4 Organization 7

2 BACKGROUND 9
- 2.1 Transformer-based Large Language Models (LLMs) and their serving infrastructure 9
  - 2.1.1 LLM Basics 9
  - 2.1.2 Today's LLM Serving Infrastructure 10
- 2.2 LLM Serving Workloads in the Wild 12
  - 2.2.1 Scale Analysis of Real-World LLM Workloads 12
  - 2.2.2 Context Reuse in Real-World LLM Workloads 14
- 2.3 Prior Work on LLM Serving Engine Optimization 16
- 2.4 Prior Work on Distributed LLM Serving 18
- 2.5 Summary 19

3 OVERVIEW 21
- 3.1 Goal of Distributed LLM Serving 21
- 3.2 Key Insight: Decoupling KV Caches from Serving Engines 22
- 3.3 End-to-End System Design of LMStack 23
  - 3.3.1 System Architecture 23
  - 3.3.2 Contrast to Prior Works 25
- 3.4 Challenges in LMStack 26
- 3.5 Summary 27

4 KV CACHE OFFLOADING INTERFACE 28
- 4.1 Design goals 28
- 4.2 Design of KV Cache Offloading Interface 29
  - 4.2.1 Data structure for offloaded KV cache 29
  - 4.2.2 Destination device to offload 30

4.2.3 Timing of interface calls . . . 31
4.3 Integration with vLLM . . . 32
4.4 Performance Evaluation . . . 33
4.5 Summary . . . 36

5 KNOWLEDGE DELIVERY NETWORK . . . 37
5.1 Motivating KDN . . . 37
5.2 Practical Challenges and Overview of KDN . . . 39
5.3 Interface Design of KDN . . . 40
5.4 KV Cache Compression . . . 43
5.5 Flexible Composition of KV Cache . . . 46
5.6 Performance Evaluation of KDN . . . 48
5.7 Discussion and Future Work . . . 52
5.8 Summary . . . 53

6 GLOBAL ORCHESTRATOR FOR DISTRIBUTED LLM SERVING . . . 54
6.1 KV Cache Prefetching . . . 54
6.1.1 Motivation . . . 54
6.1.2 Key Design . . . 55
6.2 KV-Cache-Aware Load Balancing . . . 56
6.2.1 Motivation . . . 56
6.2.2 Key Design . . . 57
6.3 Flexible request migration . . . 58
6.3.1 Motivation . . . 59
6.3.2 Design . . . 60
6.3.3 Better Fault Tolerance . . . 60
6.4 Implementation and production deployment of LMStack . . . 61
6.5 Summary . . . 64

7 CONCLUSION . . . 65
7.1 Contributions . . . 65
7.2 Future Works . . . 66

REFERENCES . . . 67

LIST OF FIGURES

1.1 Comparison between existing approaches and this dissertation: existing works treat the serving engines as black boxes, while we decouple the KV cache (serving engine's internal states) to provide better performance and manageability. . . . 4
1.2 The main contribution of this dissertation is to present a suite of solutions (bottom) to address the key challenges of distributed LLM deployments (top). The key insight (middle) is that decoupling the KV cache from the serving engines is the key to building an efficient distributed LLM serving system. 7

2.1 Illustration of Key & Value tensors (KV caches). 10
2.2 How today's serving engines manage KV caches. 11
2.3 Scale analysis of real-world LLM workloads. (a) shows the average input length in tokens, (b) shows the average output length in tokens, and (c) shows the average GPU time to process the requests across 3 featured use cases. 13

3.1 Comparison of today's LLM and our proposed abstraction. 22
3.2 System architecture of LMStack 23
3.3 The technical roadmap of this dissertation towards making LMStack practical . 27

4.1 Measuring the KV cache offloading and injection speed of naive implementation and our optimized implementation. The figure also shows the speed of LLM generating the KV caches. 34
4.2 Impact on serving engine's prefill and decoding speed when doing KV cache offloading with or without the optimizations. 35

5.1 The architecture of Knowledge Delivery Network (KDN). 40
5.2 Main components inside KDN. 42
5.3 Visualizing the K cache generated by Llama-3.1-8B model. The horizontal "stripes" in the figure show that the adjacent tokens have similar values in the same channel. 44
5.4 Erasing values in different layers only negligibly impacts the quality of LLM-generated output. The accuracy is measured from 100 questions randomly sampled from LongChat [84] dataset. 45
5.5 An illustrative example of an LLM input with two text chunks prepended to a query. If we directly concatenate the KV cache of the text chunks together, the LLM will give wrong answers. 47
5.6 Across 8 datasets from long-document Q&A and RAG-based Q&A use cases, KDN helps reduce the time-to-first-token (TTFT) by 2.1-4.6×. 50
5.7 Across 4 different datasets, the serving engine can serve 1.7-3× more queries using the same hardware with the help of KDN. 51
5.8 Compared to prefilling the text on GPU, KDN reduces the end-to-end serving cost by storing and reusing the KV cache. 52

6.1 Design of the KV-cache-aware load balancing router 57
6.2 Illustration of how global orchestrator does the request migration. 59

6.3 Main components in LMStack when deploying in a Kubernetes cluster. 61
6.4 End-to-end performance evaluation shows LMStack can process 2.5-4× more requests with 4× lower TTFT compared to the opensource and commercial alternatives. 63

LIST OF TABLES

2.1 Selected LLM applications and corresponding datasets. 13

3.1 Comparison between LMStack and prior works. 25

4.1 Typical bandwidth between GPU and different destinations. 30
4.2 How fast the serving engine can generate KV cache. 30
4.3 Offloading time, throughput, and the ratio of kernel launching overhead when offloading the KV cache of 256 tokens using different chunk sizes from GPU to page-locked memory. 31

5.1 Comparing the time of loading KV cache from different storage devices and prefilling the text. 43
5.2 In KDN, our GPU-based KV cache compression and decompression kernel achieves 100× speed up compared to the original implementation in CacheGen, making it practical to compress and decompress KV caches in real-time. 46
5.3 **Quality impact**: Across 8 datasets, KDN only creates a negligible impact on the LLM's output quality. 51

6.1 With task migration, LMStack can provide a much better user experience when failure happens in one of the serving engines—the user session is not interrupted and the generation stall time can be decreased from 7.2 seconds to 1.7 seconds (4.2× improvement). 61

ACKNOWLEDGMENTS

First, I would like to express my deepest appreciation to my Ph.D. advisor, Prof. Junchen Jiang, for his unwavering guidance, patience, and encouragement throughout my research. His mentorship and insightful research taste have helped me a lot during the 5-year journey of my Ph.D. Beyond academic mentorship, his kindness and willingness to invest in my growth both as a researcher and individual have left a lasting impact on me.

I'm extremely grateful to Prof. Hui Zhang, for not only joining my dissertation committee but also for providing valuable advice to my academics, work, and life. His supportive guidance and useful feedback have helped me get through many challenges through the years.

In completing this dissertation, I would also like to extend my great appreciation to Kexin Pei, for being one of my committee members and providing valuable assistance. His insightful comments have broadened my perspective and made my dissertation more solid.

Many thanks to Prof. Vyas Sekar, Prof. Nick Feamster, Prof. Francis Y. Yan, and Prof Ion Stoica, for their constructive feedback and helpful discussions over the years. They have provided plenty of support across the projects during my Ph.D., and their insights have significantly shaped my research direction and problem-solving approach.

Throughout my Ph.D. years, I have been learning a lot of things during my internship at Conviva. I would like to give my thanks to my colleagues, Jibin Zhan, Haijie Wu, Henry Milner, Rui Zhang, Yan Li, Han Zhang, and Evan Chan, for helping me understand the background knowledge in the state-of-the-art industry.

Thanks should also go to my lab mates at the University of Chicago, Xu Zhang, Zhengxu Xia, Kuntai Du, Yuhan Liu, Siddhant Ray, Jiayi Yao, Haodong Wang, Yuyang Huang, Shaoting Feng, Qizheng Zhang, Hanchen Li, for their assistance in discussing ideas, building artifacts, conducting experiments, and writing papers. The collaborative spirit in the group has been critical to the success of all my projects.

A special thanks go to my roommate, Lai Wei, who has always been supportive of the issues in daily life. Also thank you to my loyal companion, Tuss, whose unwavering affection, playful energy, and comforting presence have been a constant source of joy and stress relief throughout my Ph.D. journey.

Finally, I want to thank my family and friends, whose love and support have been a strong foundation that kept me grounded and motivated throughout this journey. Also, a great special thanks to my love, Caicai, for her patience, understanding, encouragement, and unreserved support across the Pacific Ocean throughout the years.

I am deeply grateful to everyone who has contributed to this journey in ways both big and small. If I have unintentionally left anyone out, please know that your support has not gone unnoticed and is sincerely appreciated.

ABSTRACT

As the use of large language models (LLMs) expands rapidly, so does the intensity and scale of the workloads required to query LLMs. Thus, the requirements for serving LLMs evolve beyond single-instance deployment to large-scale distributed deployment. As most of today's LLM serving system optimizations focus only on speeding up a single serving instance, key techniques for distributed LLM deployments are still missing.

The key contribution of this dissertation is the design and implementation of an efficient system for distributed LLM serving engine deployment. Our thesis is that by decoupling the inference states (KV caches) from the LLM serving engine, the performance of distributed LLM inference can be substantially improved. Unlike prior work, which treats the LLM serving engine as a black box and builds global orchestrators for serving engines, our approach uses a separate module to transfer, store, and share the KV caches across different serving engines.

To prove this thesis, this dissertation provides a suite of techniques to address the following fundamental challenges. First, we need to offload the KV caches from serving engines and insert them back without impacting the LLM's runtime performance. Second, we need an efficient way to store, transmit, and compose KV caches across the whole distributed cluster. Third, we need a scalable way to manage the serving engines and schedule the requests to maximize the utilization of computing resources.

Our key insight is that many KV caches can be reused across serving engines in the large-scale distributed setup. With the reused KV cache, the LLM can skip computationally intensive prefills and start generating outputs immediately. Based on the insight, we develop LMStack, an end-to-end software stack for distributed LLM deployment that can fully realize the potential of KV cache reuse. We have shown that our solution can substantially reduce the computational cost on real-world workloads, as well as provide better production-ready guarantees, including easy-deployment, automatic scaling, and fault tolerance.

We have fully open-sourced LMStack at `https://github.com/LMCache/LMCache` and `https://github.com/vllm-project/production-stack`, where the former implements the KV cache offloading, storage, and transmission component and the latter includes the distributed cluster management.

CHAPTER 1

INTRODUCTION

Recently, the use of large language models (LLMs) [52, 120, 138, 55] has expanded rapidly and LLM applications have been widely deployed into many aspects of daily life, such as chatbots [50, 27, 10, 9], document summarization [65, 129], information retrieval [25, 26], programming copilot [14, 34], and workflow automation [126, 16]. Unlike traditional machine learning models [68, 117, 90, 91, 67, 108] that learn all the knowledge from the training data, LLMs often take in external knowledge from a long input prompt. For instance, chatbots [50, 10, 9] and agent systems use the chatting histories as supplementary knowledge to generate personalized responses; enterprises use LLM agents to answer queries based on their internal databases, as new knowledge, using retrieval-augmented generation (RAG) [82, 76, 48, 124, 85]; and search engines use LLM to read fresh and relevant web data from the internet for each user query [30, 17, 11].

LLM serving engine is a stateful application that takes the prompt as input, updates its internal states (which is also known as ***KV cache*** [131, 64, 92]), and generates the output based on the KV cache. However, the fundamental difference between LLM and traditional stateful applications is that it takes a tremendous amount of time and computation cost to generate the KV cache when the input prompt is long. For instance, it takes more than 20 seconds for a small-sized LLM (*e.g.,* Llama-34B model [57]) to process a 50-paged document on an industry-standard server equipped with Nvidia A100 GPUs [92]. In real-world workloads, a service provider needs to process such long-input requests at a rate of tens of requests per second, which requires deploying multiple LLM serving engine instances on a large-scale distributed cluster.

1.1 Distributed LLM serving: Opportunities and challenges

With the increased workload size and deployment scale, more opportunities are emerging to optimize the efficiency of LLM inference. In the meantime, the distributed setup also incurs new challenges.

1.1.1 Observed opportunities

In real-world workloads, the input prompts may contain many recurring segments. For example, the same set of documents can be queried many times with different questions in knowledge base management applications [104, 21, 22]; the same context of a user is added to the prompt of a personalized assistant service [25, 26, 27]; the same conversation history will be reused every time the user talks to a chatbot [10, 9].

In response, modern LLM serving engines (*e.g.,* vLLM [79], SGLang [141]) support storing and reusing the *KV cache* of the prompt to skip redundant computation when the prompt is used again. By reusing the KV cache, the serving engine can skip the time-consuming processing of the long input context, which drastically improves the efficiency of the serving engine.

As the scale of the distributed LLM deployment grows, so does the size of the system's workload. Prior works [115] have shown that the frequency of the prompt reuse will increase if the workload size grows higher. Therefore, a scalable distributed LLM deployment may benefit more from reusing the prompts.

1.1.2 Key challenges

However, building an efficient system that runs LLM serving engines in distributed clusters still faces practical challenges.

Firstly, although input prompts may contain reusable parts, their KV cache often cannot

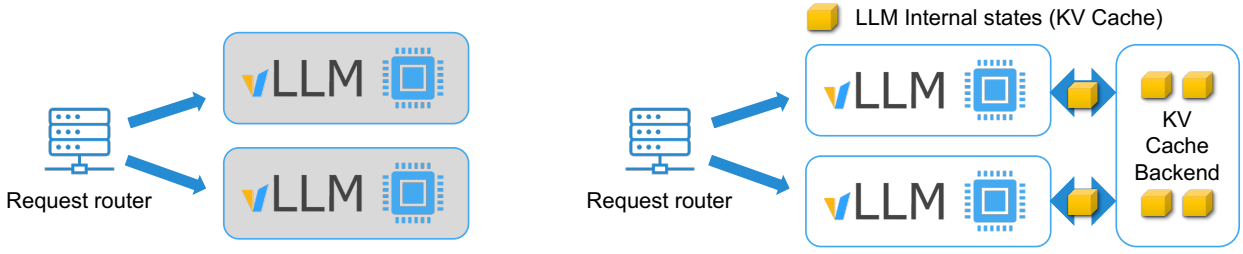
be reused by the serving engines. In modern serving engines, KV caches are stored in GPU memory for rapid reuse. However, the KV cache, composed of large high-dimensional floating-point tensors, can consume a significant amount of GPU memory (e.g., tens of gigabytes) [92]. Moreover, the size of the KV cache increases with both the context length and the model size. Consequently, a context's KV cache may be evicted from GPU memory within tens of seconds to minutes in real-world workloads. Without the KV cache, the serving engine must spend additional time and GPU cycles to process the same prompt again, which substantially decreases the system's efficiency.

Secondly, failures can frequently occur in a distributed cluster [70], necessitating a highly fault-resilient system. However, today's serving engines [79, 141, 71] lack checkpointing or fault tolerance mechanisms, meaning that if a failure occurs, all requests running on that node will be lost. Resuming the affected request on other serving engines not only consumes significant time but also impairs the performance of other requests.

Thirdly, the system should be sufficiently flexible to handle fluctuating workloads. In real-world scenarios, there can be an order-of-magnitude difference between peak and base request rates [125]. This requires the system to quickly adapt to workload changes and be able to scale up or down the serving engine instances dynamically. However, bootstrapping the serving engine instance from scratch may take up to ten minutes, as it needs to load tens or hundreds of gigabytes of the model weights [24] into GPU memory. This delay in bootstrapping time makes it challenging to maintain optimal performance during sudden spikes in demand, potentially leading to significant service degradation or downtime.

1.2 Limitations of Prior Approaches

Recent works attempt to improve the efficiency of distributed LLM serving; however, they fail to address the aforementioned challenges. These works mainly fall into two categories:



(a) Existing approaches: treat serving engines as black boxes.

(b) Our approach: decouple internal states (KV cache) from the serving engine.

Figure 1.1: Comparison between existing approaches and this dissertation: existing works treat the serving engines as black boxes, while we decouple the KV cache (serving engine's internal states) to provide better performance and manageability.

- *Single-instance serving engine optimizations* improve efficiency in executing requests within a single serving engine through techniques such as optimizing GPU kernels [54, 53, 132], reducing compute complexity [140, 74, 75], and enhancing request scheduling [79, 141]. Although these techniques can boost single-instance performance, they completely miss the opportunity to reuse contexts across distributed instances.

- *Smarter algorithms for request routing* treat the serving engines as black boxes and aim to improve KV cache reuse by routing incoming requests to one of the serving engine instances [115, 72]. However, these approaches are fundamentally limited by the fact that each serving engine can only hold a limited number of short-lived KV caches. If the KV caches of the prompts are evicted from the serving engines, the routing algorithms will not be effective.

Moreover, the fault-tolerance and elastic scaling challenges remain unsolved in prior works.

1.3 Our Technique: Decoupling KV Cache from LLMs

This dissertation is based on the key insight that *decoupling the internal states (KV cache) from the LLM serving engines can substantially improve the performance of distributed LLM*

inference (Figure 1.1).

More specifically, we argue the need for a separate module to transmit, store, and manage the KV caches apart from the distributed serving engine instances. Installing such a module brings the following benefits in distributed LLM deployments.

- First, storing KV in a separate module bypasses the limit of GPU memory and enables the sharing of the KV cache across different serving engine instances. A larger KV cache storage pool across different serving instances can hold the KV caches for a longer time, and as a result, more KV cache can be reused across instances.

- Second, the module provides a way for serving engines to checkpoint the processed prompts. When a failure occurs in one of the serving engines, other serving engines can easily resume the job from the existing KV cache stored inside the module without needing to process the input prompt again. This avoids repetitive computation and stalls in generation.

- Third, this module enables seamless workload migration when a new serving engine launches. This allows the newly joined instance to directly take the unfinished work from the existing instances.

Building on the insight, this dissertation develops LMStack, an end-to-end framework for distributed LLM serving with the following concrete solutions to address the aforementioned challenges.

- **A standard interface for KV cache offloading** (Chapter 4).While all the state-of-the-art serving engines implement the KV cache mechanism, how to manage the KV caches is very different across different serving engines. For instance, vLLM [79] splits the KV cache into small "pages" containing 16 tokens and uses a "page table" to maintain all the pages, while SGLang [141] uses a radix tree to store the KV caches where each tree node could be the KV cache of just a single token. To decouple the

KV cache management component and the implementation details within the serving engine, we have designed a standard interface to offload the KV cache from the serving engine and inject them back. We have also provided an efficient implementation of the interface in vLLM using hand-written CUDA [12] kernels.

- **An efficient module to manage KV caches** (Chapter 5). Despite the benefits, efficiently managing KV caches still faces some technical challenges. The efficiency of storing and transmitting KV caches suffers from the considerable size of KV caches. Also, KV cache chunks are composable as text segments, limiting the versatility of reusing it across prompts with different prefixes. To address those limitations, we have developed Knowledge Delivery Network (***KDN***), a separate KV-cache management system, which dynamically compresses, composes, and modifies KV caches to optimize the storage and delivery of KV caches and the LLM inference based on KV caches. In KDN, we have also implemented the mechanism to manage KV cache in multi-tier storage (including GPU memory, CPU memory, local disk, and remote storage server) within a distributed environment.

- **A global orchestrator to better manage the requests and the serving engines** (Chapter 6). Although KDN provides modular and efficient management for KV caches, it does not directly solve the aforementioned challenges of distributed LLM serving in production (*e.g.,* request scheduling, fault tolerance, and autoscaling). Therefore, we have built LMStack, a software stack to support a better production-ready deployment of LLMs. The global orchestrator provides a KV-cache-aware request router, which routes the request based on the KV cache information retrieved from KDN, minimizing the inference and network cost while keeping the load balanced across different instances. Moreover, we have also implemented live task migration in LMStack, which can quickly migrate a request to another instance and resume generation without repetitive computation when failures happen or a new instance joins, providing better fault

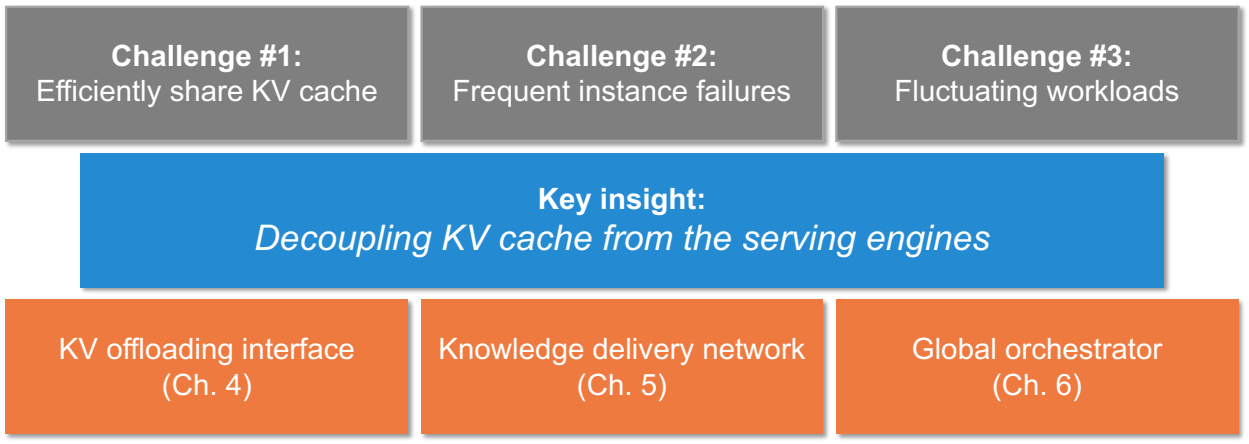


Figure 1.2: The main contribution of this dissertation is to present a suite of solutions (bottom) to address the key challenges of distributed LLM deployments (top). The key insight (middle) is that decoupling the KV cache from the serving engines is the key to building an efficient distributed LLM serving system.

tolerance and auto-scaling features.

LMStack uses container technologies [100] and helm charts [19] to provide a cloud-native, hardware-agnostic way for users to deploy the distributed LLM service with one key. As we will see in the following chapters, LMStack achieves higher efficiency and provides better manageability in distributed LLM serving compared to state-of-the-art solutions. By storing, composing, and reusing the KV cache in KDN, LMStack can serve the requests with a 3.1-4× time-to-first-token across multiple standard benchmarks. Moreover, with the smart request scheduling in the global orchestrator, LMStack can reduce the cost of distributed serving by 2.5-4× compared to the state-of-the-art open-source and commercial distributed LLM deployment solutions.

1.4 Organization

The rest of this dissertation is organized as follows. Chapter 2 begins with the background information of the LLMs, serving engines, and today's LLM workloads. It then discusses two recent research directions closely related to this dissertation: single-instance serving engine

optimizations and distributed LLM instance management.

Chapter 3 presents an overview of the main insights of this dissertation: decoupling the KV cache from serving engines can substantially improve the performance of the distributed LLM serving. It then discusses the high-level design of LMStack and its main components.

The next three chapters present the details of the three key components in the LMStack: the KV cache offloading interface (Chapter 4), the Knowledge Delivery Network (KDN) (Chapter 5), and the global orchestrator (Chapter 6).

Finally, Chapter 7 summarizes the contributions of the dissertation and ends with the discussion about future works.

CHAPTER 2

BACKGROUND

Before we talk about the solutions for distributed LLM serving, it would be helpful to *(i)* introduce the basics of LLM and motivate the need for improving the distributed LLM serving on today's workloads, and *(ii)* understand the existing efforts in this area and their limitations.

The first part of this chapter introduces the basics of LLMs and today's infrastructure for LLM serving (Section 2.1). It then discusses the need for distributed serving as well as the opportunities by analyzing the real-world workloads (Section 2.2).

The second part of this chapter discusses research directions and existing works that are close to this dissertation for optimizing the distributed LLM serving (Section 2.3 and Section 2.4).

2.1 Transformer-based Large Language Models (LLMs) and their serving infrastructure

We begin by introducing the basic concepts in transformer-based large language models (Section 2.1.1), and then talk about the specialized infrastructure to serve LLM inferences—LLM serving engines (Section 2.1.2).

2.1.1 LLM Basics

Transformers [122, 47, 51] are the state-of-the-art models for most generative AI services. At a high level, a transformer takes as input a list of input tokens (which can be quickly transformed from text, images, or videos), and outputs a sequence of new tokens through two phases: *prefill* and *decode*.

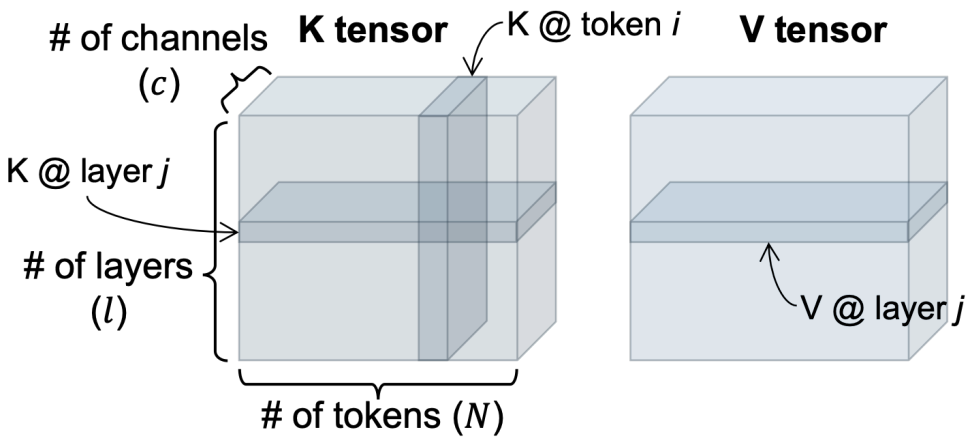


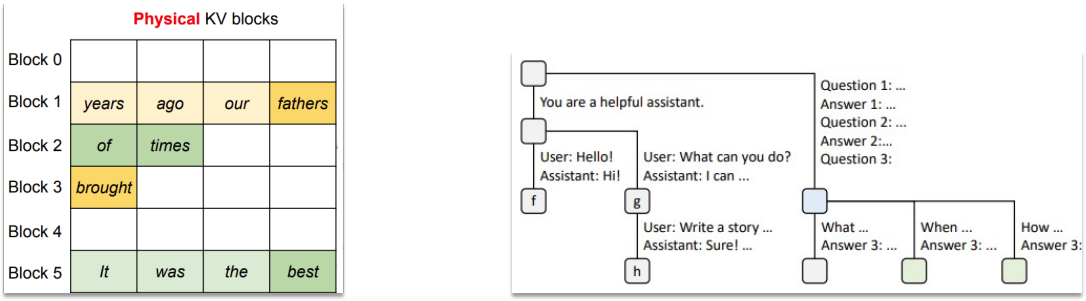
Figure 2.1: Illustration of Key & Value tensors (KV caches).

During the prefill phase, a neural network with multiple attention layers takes in the input token. Then each of the attention layers produces two three-dimensional tensors, a key (K) tensor and a value (V) tensor (shown in Figure 2.1). These K and V tensors contain information essential for LLM to utilize the context later. All the KV tensors across different layers are together called the *KV cache*. In all mainstream models, the compute overhead of the prefill phase grows superlinearly with the input length.

During the generation phase, also called the decoding phase, the KV cache is used to compute the attention score between every pair of tokens, which constitute the attention matrix, and generate output tokens in an autoregressive manner. For performance reasons, the KV cache, which has a large memory footprint [80], is usually kept in GPU memory during this phase and released afterward. Some emergent optimizations save and reuse the KV cache across different LLM requests, as we will explain shortly in Section 2.3.

2.1.2 Today's LLM Serving Infrastructure

While LLMs like GPT-4 [18] and Llama [24] have revolutionized natural language processing with their extraordinary abilities, deploying these models efficiently in real-world applications requires specialized infrastructure known as **LLM serving engines**, such as vLLM [79],



(a) vLLM: store the KV cache as non-continuous blocks to reduce memory fractions

(b) SGLang: discrete KV cache segments in a "radix-tree" for faster prefix lookup

Figure 2.2: How today's serving engines manage KV caches.

SGLang [141], TensorRT LLM [32], and TGI [71].

An LLM serving engine is a software framework designed to run the inference process of large language models efficiently. At a high level, the serving engine coordinates three core tasks: request handling, inference execution, and resource management. When a user submits a request, it will be put into the serving engine's internal request queue, which is operated by the request scheduling algorithm inside the serving engine. The request scheduling algorithm assigns priorities to the requests to maximize, balancing the latency across requests. When a request is scheduled for inference, the serving engine executes the models on the hardware accelerators (*e.g.,* GPUs and TPUs) by running the specialized hardware-optimized codes (*e.g.,* CUDA kernels). Additionally, many serving engines also support dynamic request batching [141, 71, 79, 133] and request preemption [79] to maximize the utilization of the underlying hardware resources.

A key challenge for the serving engine is to manage the KV cache tensors of the requests properly. Unlike the tensors in the traditional deep learning workloads, the KV cache will dynamically grow as the model generates new tokens, and its lifetime is hard to predict [79]. Therefore, one cannot pre-allocate a fixed-sized GPU buffer to hold the KV cache of a request, and the varying size of the KV cache creates severe memory fragmentation issues. To address this challenge, different serving engines apply different strategies:

- **Paged KV cache management** (Figure 2.2a) is adopted by vLLM [79] and TGI [71]. Inspired by operating systems' memory management, Paged KV cache management divides requests' KV cache into blocks, each with a fixed number of tokens. It employs a paged table to manage the map between tokens and KV cache blocks. Paged KV cache management eliminates external memory fragmentation since all the blocks are the same size. It also reduces memory usage by sharing the KV cache blocks across requests with the same prompts.

- **Radix-tree-based KV cache management** (Figure 2.2b) is adopted by SGLang [141]. This approach manages the KV cache as a traditional cache and uses a radix tree to efficiently find the corresponding KV cache for a given sequence of tokens. In the meantime, the radix tree also provides an interface for fast insertion and eviction of the KV caches.

Besides the KV cache management, the serving engines also include optimizations from different aspects, which we will shortly introduce in Section 2.3.

2.2 LLM Serving Workloads in the Wild

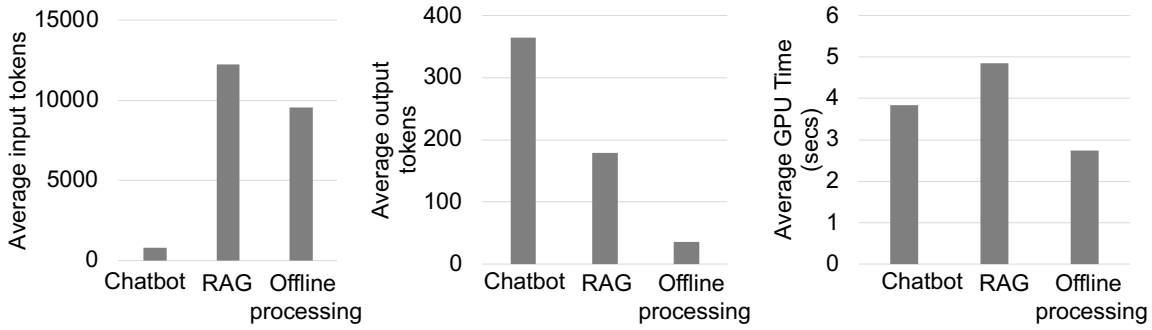
Next, we motivate the need for distributed LLM serving by analyzing multiple real-world LLM workloads. Specifically, we answer the following two questions: *(i)* how many serving engines are needed to serve the requests in real-world LLM applications (Section 2.2.1), and *(ii)* what are the potential benefits brought by deploying serving engines in a distributed environment (Section 2.2.2).

2.2.1 Scale Analysis of Real-World LLM Workloads

We start by analyzing the input-output characteristics of a few featured real-world LLM applications, including chatbot service, RAG-based Q&A application, and offline document

Application	Dataset
Chatbot service	BurstGPT [125], ShareGPT [49]
RAG-based Q&A	RAGBench [61]
Offline document processing	LongChat [84]

Table 2.1: Selected LLM applications and corresponding datasets.



(a) Average *input* length of three different use cases

(b) Average *output* length of three different use cases

(c) Average *GPU time* to process the requests in each use case

Figure 2.3: Scale analysis of real-world LLM workloads. (a) shows the average input length in tokens, (b) shows the average output length in tokens, and (c) shows the average GPU time to process the requests across 3 featured use cases.

processing. Table 2.1 shows the dataset we used for each application.

Figure 2.3(a) and Figure 2.3(b) show the average number of input and output tokens per request in the above three applications. In the chatbot service, LLMs usually take a short input from the users: the average input length is only 793 tokens. However, the LLM generates a relatively long output to answer the user's request compared to RAG and offline document processing use cases. In contrast, both input and output are long in RAG-based Q&A services. This is because the related knowledge is usually embedded in the input prompt in the RAG application. Although offline document processing does not need to generate too many tokens, the input documents usually take much longer.

We further profiled the GPU time to process each request by running a small-sized LLM (Llama-34B [24]) via vLLM [80] on a server with a Nvidia A100 GPU. Figure 2.3(c) shows the average GPU time to process the requests are 3.83, 4.85, and 2.74 seconds for the

requests from the chatbot service, RAG-based Q&A, and offline document processing use cases respectively. Based on the analysis, a single serving engine running on a GPU server with 8× A100 GPUs can only achieve 2.1, 1.6, and 2.9 requests per second for chatbot service, RAG-based Q&A application, and offline document processing.

As shown in prior works and industry reports [125, 62], the request rate to the application provider can easily achieve hundreds of requests per second. Therefore, it is vital to deploy LLM serving engines into a distributed cluster in order to serve real-world workloads.

2.2.2 Context Reuse in Real-World LLM Workloads

As shown in prior works, LLMs may generate low-quality or hallucinated answers when the response requires knowledge not already embedded in the models [134, 124, 85, 56]. Thus, many LLM applications and users supplement the LLM input with additional texts, referred to as the **context** [63, 83]. The LLM can read the context first and use its in-context learning capability to generate high-quality responses. The contexts in LLM input can be used for various purposes.

- a user question can be supplemented with a document about specific domain knowledge, to produce better answers [10, 110], including using the latest news to answer fact-checking inquiries [30, 36], using case law or regulation documents to offer legal assistance;

- code analysis applications retrieve context from a code repository to answer questions or generate a summary about the repository [45, 73, 77], and similarly, financial companies use LLMs to generate summaries or answer questions based on detailed financial documents [99];

- gaming applications use the description of a particular character as context so that the LLM can generate character dialogues or actions matching the character's personal-

ity [101, 126, 112];

- in few-shot learning, a set of question-answer pairs are used as context to teach the LLM to answer certain types of questions [39, 94, 114];
- in chatting apps, the conversational history with a user is often prepended as the context to subsequent user input to produce consistent and informed responses [78, 43, 111, 116], etc.;

We observe that in practice, contexts are often **long** and often **reused** to supplement different user inputs.

Long contexts are increasingly common in practice. For example, those contexts discussed above, such as case law documents, financial documents, news articles, code files, and chat history accumulated in a session, easily contain thousands of tokens or more. Intuitively, longer contexts are more likely to include the right information and hence may improve the quality of the response. Retro [46] similarly shows that the generation quality (perplexity) improves significantly when the context increases from 6K tokens to 24K.

Moreover, these long contexts are often *reused* by different inputs. In the financial analysis example, consider two queries, "summarize the change in the revenue of the company last year" and "what leads to the most increase or decrease in the company's income last year"; the same earning reports are likely to be supplemented to both queries as the contexts. Similarly, the same law enforcement document or latest news article can be used to answer many different queries in legal assistant or fact-checking apps. As another example, during a chat session, early chat content will keep getting reused as part of the context for every later chat input.

Longer contexts can lead to higher prefill computations and cause longer delays, however, it is promising to reduce such computation and delay by storing the KV cache of the contexts in the serving engine. Most modern serving engines [80, 141, 71, 32] support storing KV

caches in GPU memory. That said, a single serving engine can only store the KV cache of a limited number of contexts. This is because KV caches, as a high-dimensional tensor, can easily take 10s GB of memory, while the available memory to store KV cache on a single GPU is usually less than 100 GB.

Given that a single serving engine has limited capability to store KV cache, we argue that *reusing KV caches across distributed LLM serving engines can substantially improve the efficiency of the inference*. For example, in the RAG-based Q&A use case above, increasing the KV cache reusing ratio from 0% to 90% can reduce the average processing time from 4.85 seconds to 2.01 seconds, meaning that the cost can be reduced by 2.41×.

2.3 Prior Work on LLM Serving Engine Optimization

Many recent research works have focused on optimizing the efficiency inside the LLM serving engine. The optimization covers multiple different aspects from hardware acceleration to request-level scheduling.

Operator acceleration: The most direct way to improve the efficiency of the serving engine is to provide better operators for inference computation. FlashAttention [54, 53] is an I/O-aware optimization algorithm that accelerates attention operation in transformer-based LLMs. It minimizes the reads/writes to GPU HBMs by fusing multiple GPU kernels into one. It also employs the tiling technique to reduce the runtime memory footprint, increase the maximum batch sizes, and improve the generation throughput. Some recent works also quantize the model weights or KV caches to accelerate the GPU operators [87, 88, 93]. By embracing vectorized instructions, the GPU can process multiple numerical operations within one clock cycle, boosting the operator's speed.

Improved Parallelism: Since the decoding stage is usually bounded by GPU memory bandwidth instead of computation, recent works also explore how to improve parallelism during generation. *Speculative decoding* [97, 81] accelerates LLM inference by using a smaller

draft model to propose token sequences, which are then verified in parallel by the target LLM. This approach reduces latency by leveraging the draft model's speed while preserving the quality of the output. *Chunked prefill* [38, 42] improves the parallelism of decoding by batching the decoding with prefilling by dividing long prefill requests into small chunks. It also implements specialized GPU operators to batch prefill and decode requests together.

Better request scheduling: State-of-the-art serving engines employ different optimizations for better request scheduling. vLLM [80] uses a priority-based scheduling system, which enables interactive requests to preempt batch workloads while preserving their KV cache to avoid recomputation. It also dynamically adjusts priorities to ensure fairness and minimize latency across requests with the same types. SGLang [141] employs the scheduling algorithm proposed by NanoFlow [143]. The idea is to overlap the CPU scheduling with the GPU computation by letting the scheduler run one batch ahead and prepare all the metadata required for the next batch.

Complexity reduction: Transformer inference is the most costly component in the serving engine since its compute complexity scales quadratically as the input context length grows. Therefore, existing works also explore how to reduce the complexity of the attention operation. Routing Transformer [109] reduces the attention's complexity for N tokens from N^2 to $N^{1.5}$ by a sparse routing module based on online k-means while preserving the quality of the output. The Mixture of Expert (MoE) [107, 89] technology runs the inference on a subset of the model weights, which drastically reduces the computation complexity of transformer inference without decreasing the model's ability. Other works have also explored how to reduce the number of tokens in the KV cache by maintaining a fix-sized KV [127, 128], evicting the less important tokens [140, 86, 60, 106, 37], merging multiple KV caches together [113, 123], or dropping tokens from the prompt [74, 75].

In summary, recent advancements in LLM serving engine efficiency span hardware-aware operator acceleration, speculative and chunked parallelism, adaptive request scheduling, and

attention complexity reduction. However, all of the optimizations are constrained within a single serving engine instance. In this dissertation, we will focus on the optimizations across serving engine instances to make distributed LLM serving more efficient.

2.4 Prior Work on Distributed LLM Serving

Because of the rising need for distributed LLM serving, recent work has also examined how to efficiently manage serving engines in a distributed environment. The works mainly fall into two categories: *prefill-decode disaggregation* and *smart request routing*.

DistServe [142] and SplitWise [102] propose the idea of prefill-decode disaggregation. Their key insight is that the prefill and decode are very different requirements in memory access pattern, compute resource usage, and scheduling objectives, and such difference leads to severe prefill-decode interference, which slows down the system's performance. Therefore, their key idea is to eliminate such interference by separating the prefill and decoding computation into different serving engine instances.

However, such disaggregation also has limitations. First, it requires a static configuration (*e.g.,* which instance will be used for prefill or decode) before launching the LLM service, and such configuration cannot be changed online. Therefore, the service cannot scale when the workload increases, and will stop working if failure happens in one of the instances. Second, it creates intensive data transmissions between prefill and decode instances, since the KV cache needs to be transferred. In the meantime, it fails to utilize the bandwidth within prefill instances (or decode instances), causing resource underutilization. Third, there is no KV cache sharing between prefill instances, meaning that this approach cannot benefit from the increased context reuse ratio when the number of serving engine instances increases.

Another set of works assumes that the serving engine instances are replicated in the distributed cluster, and then explores algorithms to route requests to different serving engine instances. ScaleLLM [130] implements a Rust-based request router, which monitors the load

of each serving engine instance and routes the new request to the serving engine with the lightest load. Jain *et. al* [72] improves the end-to-end distributed serving performance by a reinforcement-learning-based request router. It uses a lightweight model to predict the decode length for each request, and trains an RL model to select the best-suited serving engine instance. Preble [115] proposes a prefix-aware-routing algorithm, which considers the KV cache inside each serving engine instance. The router first estimates the computation needed to execute the request on each serving engine by considering the KV cache reuse ratio and request length, and then selects the instance with the lowest needed computation. It also takes into account the load of each serving engine instance to avoid overloading one of the serving engine instances. SGLang [141] open sources a Rust-based request router, which simulates the radix-tree-based KV cache management algorithm running in each serving engine instance but at a coarser granularity. The router will route the request to the instances that can maximize the KV cache reuse ratio.

Although the smart routing techniques can achieve better load balancing or optimize efficiency by improving the KV cache reuse ratio, they all share the same fundamental limitation: *(i)* the improvement in KV cache reuse ratio is incremental since they cannot share KV cache across different serving engine instances, and *(ii)* they do not address the key challenge in distributed LLM serving—fault-tolerance and autoscale.

Contrasting to the prior work, this dissertation introduces a new component into the distributed LLM serving system to enable efficient KV cache sharing across all the serving engine instances (Chapter **??**). Based on the new component, the dissertation also includes the solutions to fault-tolerance and autoscale in distributed LLM serving (Chapter **??**).

2.5 Summary

In the first part of this chapter, we have introduced the basics of large language models and the LLM serving engines. Then, we have used real-world workloads to show the importance

of distributed LLM serving as well as the opportunities brought by scaling up the serving engines into a distributed environment.

In the second part of this chapter, we have discussed the prior works that try to improve the efficiency of the LLM serving. From the discussion, we have shown that the fully functional and efficient solution for distributed LLM serving is still missing. We will introduce how our solution addresses all the challenges and thus is superior to prior works in distributed LLM serving.

CHAPTER 3

OVERVIEW

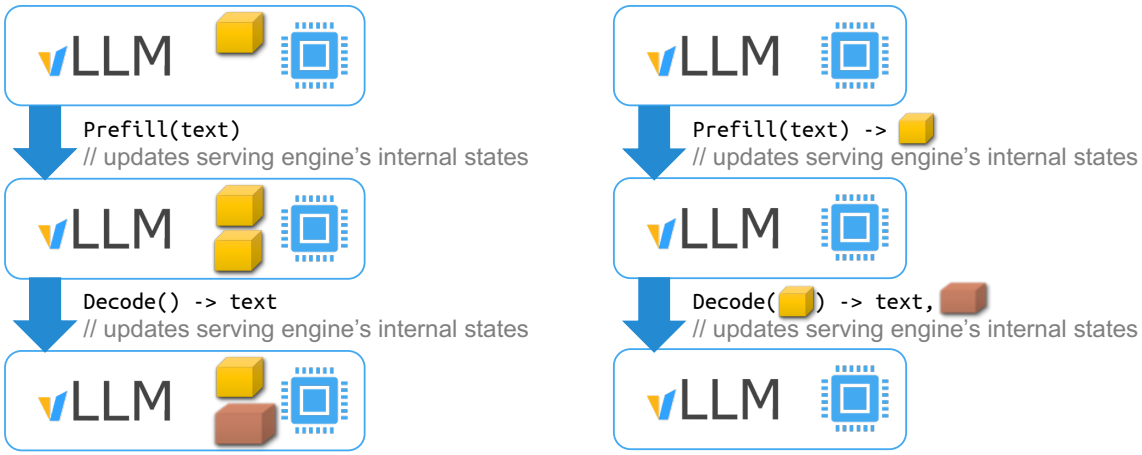
The main contribution of this dissertation is the design and implementation of an efficient system for distributed LLM serving engine deployment. To achieve this, our key insight is to *decouple the KV caches from the serving engines*. Based on this insight, we build LMStack with multiple key components to better serve LLMs in a distributed environment. This chapter is organized as follows. We begin with discussing the goal of distributed LLM serving (Section 3.1). Then, we illustrate our key insight and explain what kind of benefits it can bring (Section 3.3). Finally, we present the design of our system LMStack (Section 3.3) and discuss the challenges in building the system.

3.1 Goal of Distributed LLM Serving

We argue that a good distributed LLM serving system should achieve the following goals.

Realize the potential of KV cache reusing across the whole system: As we discussed in Section 2.2, the input prompts share a lot of recurring segments and their KV cache can potentially be reused to improve the efficiency of LLM inference. A good distributed LLM serving system should be able to fully realize such potential. Moreover, as the scale of the system increases, it should benefit more from reusing the KV caches because the available storage for KV caches also increases.

Resilient to failures in serving engine instances: In real production environments, failures happen frequently [70]. A good distributed LLM serving system should be able to minimize the impact of those failures on the running requests. More specifically, when a serving engine fails, the requests on that serving engine should be quickly migrated and resumed on other living serving engines. In the meantime, the migrated requests should have minimal impact on the target serving engine while waiting for a new serving engine to



(a) Abstraction of today's serving engine: KV cache are kept as ***internal states***

(b) Our proposal: ***decoupling*** KV cache from the serving engine, making the it become ***stateless***

Figure 3.1: Comparison of today's LLM and our proposed abstraction.

bootstrap.

Quickly respond to changes in the workloads: In real-world scenarios, there can be an order-of-magnitude difference between peak and base request rates [125]. A good distributed system should quickly adapt to workload changes and be able to scale up or down the serving engine instances dynamically.

3.2 Key Insight: Decoupling KV Caches from Serving Engines

LLM serving engines are *stateful*, and the states are the KV caches. The KV cache (state) of a request will be generated after the prefill phase finishes and will be managed by the serving engine. During the decode phase, the serving engine takes the KV cache as input, generates a new token, and appends the new token's KV cache to the end of the old KV cache. Figure 3.1a demonstrates this process.

In today's serving engines, the KV caches are maintained internally inside the engine and are not accessible by outside modules. This hinders the sharing of KV cache across multiple serving engine instances and thus fundamentally limits the feasibility of achieving the goals

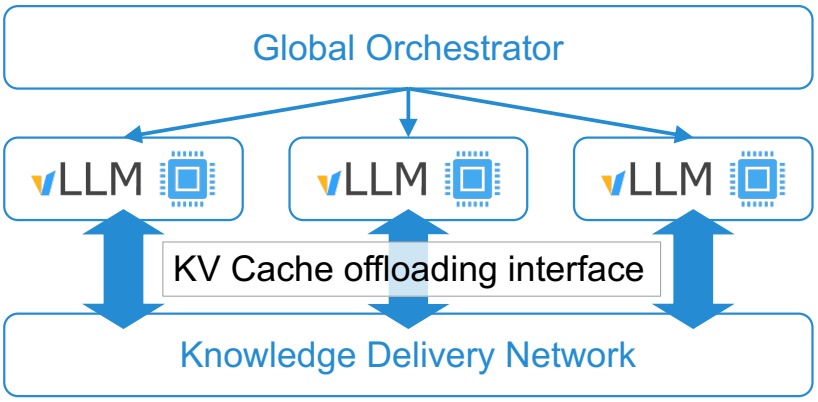


Figure 3.2: System architecture of LMStack

listed in Section 3.1.

Therefore, our key insight is that in order to build a good distributed serving system, one should *decouple the KV caches from the serving engines*. Figure 3.1b illustrates the new interface based on our insight: contrasting to maintaining the KV cache internally, now the prefill and decode phase become purely functional without any side effects.

3.3 End-to-End System Design of LMStack

With the insight of decoupling the KV cache from the serving engine, we have designed LMStack, the end-to-end distributed LLM serving system.

3.3.1 System Architecture

Figure 3.2 shows the system architecture of LMStack, which consists of the following key components:

- **Serving engine instances**: LMStack uses the state-of-the-art serving engine (*e.g.*,, vLLM or SGLang) to provide the functionality of LLM inference.
- **KV cache offloading interface**: The KV cache offloading interface enables the serving engine to offload the KV cache to other modules as well as load the KV cache

back from outside.

- **Knowledge Delivery Network** (KDN): KDN is a separate KV cache management system that shares the KV cache across different serving engine instances. It dynamically compresses, composes, and modifies KV caches to optimize the storage and delivery of KV caches. It also explores different storage options including GPU memory, CPU memory, disk, and cloud storage, which substantially increase the system's capability to store more KV caches.

- **Global orchestrator**: The global orchestrator monitors each component in the system, initiating the KV cache transfers, dealing with failures, and automatically scaling up and down the serving engine instances when the workload changes. It also provides the functionality of request routing and load balancing.

When a new request comes into the system, the global orchestrator queries KDN about the existence of the KV cache, determines which serving engine instance to route to based on the load and the KV cache locality, and tells KDN to prepare the KV cache into the target serving engine. Once the KV cache is ready, the request will be sent to the target serving engine. As the serving engine processes the request, the newly generated KV cache will be offloaded to KDN via the KV cache offloading interface.

The global orchestrator also coordinates with KDN to migrate the KV cache across different instances, which enables better fault tolerance in the distributed LLM serving system. Once failures happen in a serving engine, the global orchestrator immediately starts migrating the request from that serving engine to others. Upon the start of the migration, KDN transfers the KV cache to the target serving engines, and the global orchestrator will re-route the impacted request to the target serving engine after the KV cache transfer finishes. In this case, the impacted request can immediately get resumed by the new serving engine with the migrated KV cache.

	KV Cache Reuse	Resilient to Failures	Adapt to Workload Changes
Single-instance serving engine optimizations	Low	No	No
Prefill-decode disaggregation	Low	No	No
Smart request routing	Medium	No	No
LMStack	**High**	**Resilient**	**Quickly adapt**

Table 3.1: Comparison between LMStack and prior works.

3.3.2 Contrast to Prior Works

Table 3.1 shows the comparison between LMStack and prior works that optimize the distributed LLM serving on the dimensions we discussed in Section 3.1. LMStack is superior to the prior works in the following dimensions:

- *Much higher KV cache reuse*: By decoupling the KV cache from the serving engines, LMStack enables sharing the KV cache globally across the whole cluster. KDN's capability to store KV in different storage devices also increases the potential KV cache reuse ratio.

- *Better resilience to failures*: When failure happens in one of the serving engines, the global orchestrator will collaborate with KDN to quickly migrate the unfinished requests to other serving engines. With the migrated KV cache, the request can immediately be resumed instead of being executed from scratch again, substantially reducing the request's stall time and alleviating the load increase in the new serving engine.

- *Faster adaptation when workload changes*: LMStack provides a bunch of optimizations to reduce the delay of bootstrapping the serving engine, making it faster to launch new instances when the workload increases.

3.4 Challenges in LMStack

Despite its promise, there are also some fundamental challenges that have to be addressed before we can unleash the full potential of LMStack. This section presents our technical roadmap (depicted in Figure 3.3) towards making LMStack practical.

The first challenge is that we should be able to *efficiently offload the KV cache from the serving engines (and load them back) without impacting the performance of LLM inference*. In today's serving engine, KV caches are stored in GPU memory but maintained by different data structures in different serving engines (*e.g.,*, page table in vLLM [80] and radix tree in SGLang [141]). Simply copying out the KV cache from those data structures could block the GPU by hundreds of milliseconds or more than a second, leading to stalls in LLM generation and GPU underutilization.

The second challenge is that we should be able to efficiently store, transfer, and compose the KV cache given that the size of the KV cache is millions of times bigger than its corresponding text. Transmitting such a large KV cache between the serving engine instances could take up to ten seconds without any optimizations [92]. Moreover, to reuse the KV cache, today's serving engine requires that the text of the KV cache must be the prefix of the LLM input. However, in many real-world use cases such as retrieval-augmented generation (RAG) and LLM-based agents, the reused text is often prepended with different system prompts or composed with other contexts, meaning that they will not share the same prefix.

The third challenge is how to smartly maintain the load of each serving engine so that instance failures or workload increases will not bloat the system. Failures or sudden increases in the workloads may easily overload the distributed system. Therefore, the system should maintain the load of the instances in a fair range to avoid being overloaded by incidents.

As we will see in the next chapters, LMStack provides a bunch of technical innovations to address the above challenges.

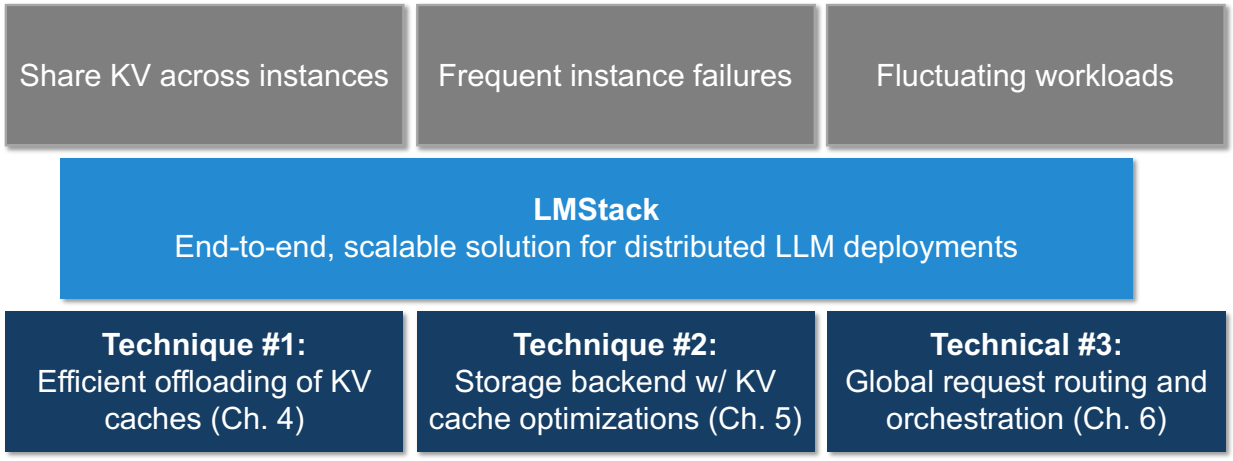


Figure 3.3: The technical roadmap of this dissertation towards making LMStack practical

3.5 Summary

In this section, we have first discussed the goals when building the system for distributed LLM serving, including realizing the potential of KV cache reuse, being resilient to failures, and adapting to workload changes quickly. To achieve this goal, we have proposed our key insight that we should decouple the KV cache from the serving engines.

Based on the insight, we have presented the system design of LMStack. Inside LMStack, we apply some concrete ideas (which will be presented in the next chapters) to address the practical challenges.

CHAPTER 4

KV CACHE OFFLOADING INTERFACE

In this chapter, we will present the design and implementation of the KV cache offloading interface in LMStack. First, we will discuss the design goals in Section 4.1. Then, we present the design of the KV cache offloading interface in LMStack by discussing the following concrete questions (Section 4.2): *(i)* what data structure we should use to store the offloaded KV cache, *(ii)* where the offloaded KV cache should be stored, and *(iii)* when the serving engine should call the KV cache offloading interface.

We will show how we integrate this KV cache offload interface with vLLM [80], one of the state-of-the-art serving engines (Section 4.4), and evaluate the performance of the KV cache offloading interface in (Section 4.4).

4.1 Design goals

The objective of the KV cache offloading interface is to bridge the gap between KDN and serving engines. At a high level, the interface should be able to *efficiently* extract the KV cache from the serving engine and pass it to KDN as well as load the KV cache from KDN and inject it back into the serving engine.

In the meantime, different serving engines have very different ways to maintain the KV caches internally. However, we do not want operations in KDN to be coupled with how serving engines manage the KV cache.

Therefore, we summarize the design goal of the KV cache offloading interface as follows:

- The interface should offload or inject KV caches as quickly as possible without impacting the performance of LLM inference.
- The offloaded KV cache should be agnostic to how serving engines manage the KV cache internally.

4.2 Design of KV Cache Offloading Interface

Based on the discussion in Section 4.1, we present the design of the KV cache offloading interface in LMStack by answering the following questions

- What data structure should we use to store the offloaded KV cache (Section 4.2.1)?
- Where should the offloaded KV cache be stored (Section 4.2.2)?
- When should the serving engine call the KV cache offloading interface (Section 4.2.3)?

4.2.1 Data structure for offloaded KV cache

As shown in Figure 2.1, by definition, a KV cache of an N token context is a continuous three-dimensional tensor consisting of shape $L \times N \times D$, where L is the number of attention layers and D is the hidden dimensions that vary across different LLMs. Given a KV cache tensor `t`, the KV cache of the i^{th} token in the context can be expressed as `t[:, i, :]`.

Today's serving engines, such as vLLM and SGLang, use different data structures to maintain KV caches internally, which makes the KV cache of a context non-continuous. For instance, vLLM divides the big KV cache tensor into blocks that contain a fixed number of tokens, and SGLang breaks the KV caches by token indexes and stores them in a radix tree. However, in order to make optimizations in KDN (which we will introduce in the next chapters) agnostic to those serving-engine-specific data structures, the KV cache offloading interface should provide a unified data structure for the offloaded KV cache.

In LMStack, our design is to convert the offloaded KV cache back to the continuous 3-D tensor representation. This design gives us a few benefits. First, no matter how a serving engine manages the KV cache, it can always be converted to the continuous 3-D tensor representation. This ensures the interface can be integrated into any serving engine while providing the unified view of the KV cache to KDN. Second, as we will see, KDN employs a bunch of optimizations to better store, transmit, and manage the KV cache. As most of

Destination device	Bandwidth
GPU memory	500 GB/s
Page-locked CPU memory	15 GB/s
Paged CPU memory	1.5 GB/s
Local SSD	1 GB/s
Remote server	0.6 GB/s

Table 4.1: Typical bandwidth between GPU and different destinations.

Model	Prefill throughput (tokens / sec)	Decode throughput (tokens / sec)	KV cache generation speed (GB/s)
Llama-3.2-1B	50000	3000	1.61
Llama-3.2-8B	10000	1000	1.35
Llama-3-70B	3500	250	1.14

Table 4.2: How fast the serving engine can generate KV cache.

the optimizations require the KV cache to be a continuous tensor, it is better to store the offloaded KV cache in a continuous tensor.

4.2.2 Destination device to offload

KV cache can be offloaded to different devices, including GPU memory, page-locked CPU memory [31], paged CPU memory, local disk, or even remote storage servers. However, the different destinations have very different performance implications.

We argue that the bandwidth from the GPU memory to the destination should be larger than the speed at which the serving engine generates the KV cache. Table 4.1 shows the typical bandwidth between GPU memory and potential offloading destinations when using Nvidia A100 GPU.

To understand how fast a model generates a KV cache, we selected a few typical models with different scales and profiled their KV cache generation speed when running on Nvidia A100 GPUs. The speed is shown in Table 4.2.

Based on the above analysis, only GPU and page-locked CPU memory can be a good destination for KV cache offloading. However, in distributed LLM serving, GPU memory is

Number of tokens	Time to offload	Throughput	Kernel launch overhead
1	3.944 ms	7.9 GB/s	20.7%
16	1.889 ms	16.6 GB/s	2.71%
256	1.565 ms	19.9 GB/s	0.20%

Table 4.3: Offloading time, throughput, and the ratio of kernel launching overhead when offloading the KV cache of 256 tokens using different chunk sizes from GPU to page-locked memory.

much more valuable than CPU memory, because *(i)* the total size of GPU memory is much smaller than CPU memory, and *(ii)* the LLM model weights take a great portion of GPU memory. Therefore, in LMStack, the destination of KV cache offloading is the page-locked CPU memory.

4.2.3 Timing of interface calls

Ideally, every time the serving engine generates a new token, the corresponding KV cache should be offloaded. This essentially provides a checkpointing mechanism for serving engines, and when failure happens in the serving engine instance, other serving engines can seamlessly resume the generation without losing any progress.

However, calling the offload with the KV cache of a single token every time may incur performance overheads. First, the size of the KV cache of a single token usually ranges from tens to hundreds of KBytes. Copying such a small amount of data cannot fully saturate the PCIE bandwidth between GPU and page-locked CPU. Second, offloading the KV cache requires launching GPU kernels, but every kernel launching will have an overhead of a few microseconds [136]. This overhead will dominate the transferring time if each transfer is small.

Table 4.3 shows the transferring time, throughput, and the ratio of kernel launching overhead when offloading the KV cache with different numbers of tokens. We measured the numbers by running Llama-3.1-8B model [35] on Nvidia A100 GPU. In practice, we offload

the KV cache every 256 tokens, minimizing the performance overhead while keeping a fine enough granularity of checkpointing.

Based on the discussions above, the KV cache offloading interface in LMStack consists of the following functions:

```
offload(tokens: Tensor, mask: Tensor, dst_kv_buffer: Tensor, **kwargs)
inject(tokens: Tensor, mask: Tensor, src_kv_buffer: Tensor, **kwargs)
```

`tokens` is a tensor that contains all tokens in the context. `mask` is a boolean tensor indicating the KV cache of which subset of the tokens should be offloaded or injected. `dst_kv_buffer` and `src_kv_buffer` are the buffers for offloaded KV cache to write to or read from. `kwargs` contains the serving-engine-specific data structures for KV cache management.

4.3 Integration with vLLM

Based on the design, we have integrated the KV cache offloading interface into vLLM, a state-of-the-art serving engine. Figure **??** illustrates the details of the integration. Before the LLM inference, vLLM first asks KDN if the KV cache of the request exists. If so, we will parse the request's page table, making sure the KV cache will be injected into the correct places, and then call the `inject()` function to load the KV cache into vLLM's paged KV buffer. Finally, the request's metadata will be updated to notify vLLM of the existence of the new KV cache. After the LLM inference, we parse the page table again to locate the newly generated KV cache in vLLM's paged table, and then call the `offload()` function to offload the KV cache into the destination buffer.

During the process, we have also applied a few optimizations to make sure the KV cache offloading and injection do not hurt the performance of normal LLM inference.

Use pre-allocated destination buffer: Whenever calling `offload()`, it needs to store the offloaded KV cache into `dst_kv_buffer` which is allocated on the page-locked CPU memory.

In practice, allocating page-locked CPU memory is much slower than allocating normal CPU memory by `malloc`. This is because page-locked memory requires the operating system to bypass paging mechanisms, ensuring that the allocated memory is always in physical RAM and is not swapped out. This process involves system calls and direct OS kernel-level operations, making it much slower.

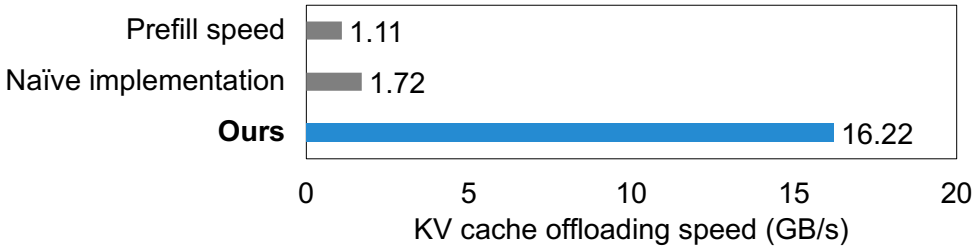
To mitigate this issue, we implemented a technique to pre-allocate a large page-locked CPU memory buffer when the serving engine is initializing. When calling the `offload()`, an application-level memory allocator will try allocating the `dst_kv_buffer` from the large page-locked memory buffer. After `offload()` finishes, the offloaded KV will be quickly consumed by KDN so that the buffer can be released and used by future calls to `offload()`.

Use separate CUDA stream to offload / inject KV cache: By default, offloading or injecting the KV cache will block the execution of other GPU kernels. This could significantly impact the performance of LLM inference. Our implementation leverages CUDA stream [13] to address this problem. When running in different streamings, computation kernels and GPU-CPU memory copies can be perfectly overlapped and executed simultaneously.

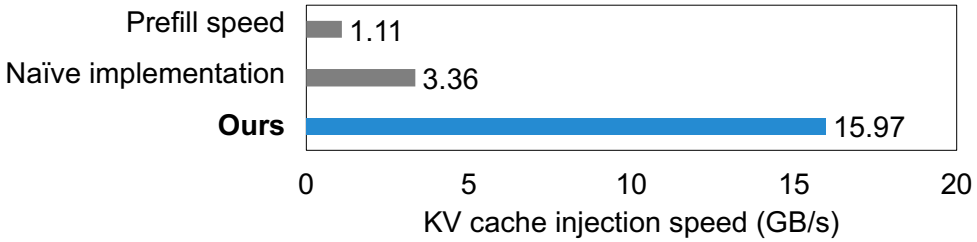
Implementation: The implementation consists of two parts: *(i)* 500 lines of python code that manipulates the KV-cache-related data structures in vLLM, and *(ii)* 300 lines of C code and CUDA kernels to efficiently offload KV cache from paged memory to a continuous page-locked buffer. The code is accessible at `https://github.com/LMCache/LMCache.git`

4.4 Performance Evaluation

To evaluate the performance of the KV cache offloading interface, we focus on answering the following two questions: *(i)* is the KV cache offloading interface fast enough to offload all the LLM generated KV cache in time, and *(ii)* will the KV cache offloading interface impact the performance of normal LLM inference? We conduct real experiments to measure the



(a) With the optimizations, our implementation can improve the KV cache **offloading throughput** by **9.43×**



(b) With the optimizations, our implementation can improve the KV cache **injection throughput** by **4.75×**

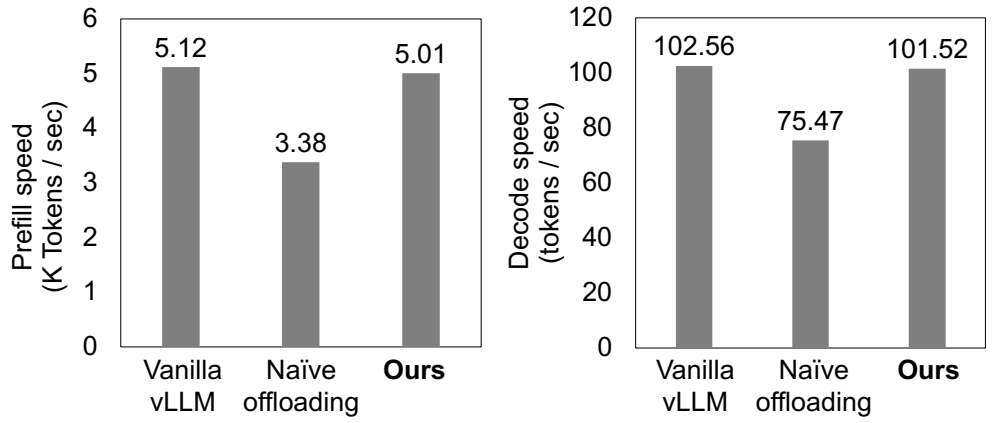
Figure 4.1: Measuring the KV cache offloading and injection speed of naive implementation and our optimized implementation. The figure also shows the speed of LLM generating the KV caches.

performance impact of the KV cache offloading interface on vLLM.

Dataset: We use the LongChat [84] dataset for the queries sent to the serving engine. The dataset contains 200 queries with 9-10K tokens long contexts plus a question like "What was the first topic we discussed". We send the queries to the serving engine at a rate of one query per second.

Baselines: The experiment uses the following baselines:

- *Vanilla vLLM*: directly run vLLM engine without any modification.
- *vLLM w/ naive KV offloading*: run vLLM with KV cache offloading but without the optimizations in Section 4.3. The naive implementation is based on the KV cache swapping module in the vLLM's official implementation [33].
- *vLLM w/ optimized KV offloading*: run vLLM with the optimized KV cache offloading



(a) LLM prefill speed **(b)** LLM decode speed

Figure 4.2: Impact on serving engine's prefill and decoding speed when doing KV cache offloading with or without the optimizations.

implementation.

Testbed setup: We run vLLM on a server with a single Nvidia A100 80GB PCIE GPU, and the serving engine runs Llama-3.1-8B model [35].

Offloading performance comparison: Figure 4.1 compares the KV cache offloading and injection speed between the naive implementation and our implementation. The naive implementation achieves 1.72 GB/s and 3.36 GB/s when offloading and injecting the KV cache respectively. With the optimizations in Section 4.2 and Section 4.3, our implementation achieves 16.22 GB/s and 15.97 GB/s tokens per second for KV cache offloading and injection, which is 9.43× and 4.75× better than the naive implementation. To put the numbers in context, the serving engine's prefill speed is around 1.1 GB/s. Therefore, our implementation for the KV cache offloading interface is fast enough to offload all the LLM-generated KV cache.

Impact on LLM inference: Figure 4.2 shows the comparison of the prefill speed and decode speed between the baselines. Without the optimizations, the KV offloading process will significantly decrease the LLM inference performance—the prefilling speed and decoding speed decreased by 33.4% and 26.5%, respectively. In contrast, with the optimizations, the

KV offloading interface in LMStack only incurs negligible performance overhead compared to vanilla vLLM—the decrease in the prefilling speed and decoding speed is less than 2%.

4.5 Summary

In this chapter, we have first discussed the design goals of the KV offloading interface, including minimizing the performance impact on LLM inference and being agnostic to the serving engine's internal implementation. Then, we have used empirical evidence to justify the choices we made in designing the KV cache offloading interface in LMStack.

We have also presented the concrete integration of the interface to a state-of-the-art serving engine. By running real experiments, our evaluation results show that the implementation can offload the KV cache from the serving engines with negligible overheads.

CHAPTER 5

KNOWLEDGE DELIVERY NETWORK

In previous chapters, we have discussed how the KV cache offloading interface efficiently offloads the KV cache from the serving engines as well as injects them back. However,

To address the problem, this chapter presents the **Knowledge Delivery Network** (KDN), another key component in LMStack to dynamically compress, compose, and modify KV caches to optimize the storage and delivery of KV caches.

The organization of this chapter is as follows. Section 5.1 discusses why we need a separate module to manage and deliver KV caches across different serving engines in the distributed LLM deployment. Section 5.2 talks about the challenges in building KDN and gives a high-level overview of KDN's architecture. Section 5.3 presents the key interface design of KDN, including how it interacts with serving engines and storage devices, and how it manages KV caches internally. Section 5.4 and 5.5 introduce the two key optimizations we implement in KDN, and we evaluate the performance of KDN in Section 5.6. Finally, we talk about the future works in KDN in Section 5.7.

5.1 Motivating KDN

As we mentioned earlier, the idea of storing KV cache has gained increasing attention in LLM services [141, 28, 105].

Why storing knowledge in KV caches pays off?: On the surface, the use of KV caches may seem merely a space-time tradeoff (trading KV cache storage space for shorter prefill), but the tradeoff is favorable for two reasons:

- KV caches are reused a lot. Many contexts, especially long contexts, are frequently reused by different queries and different users. This can be viewed as the LLM's version of Pareto's rule: an LLM, like humans, uses 20% of the knowledge for 80% of the time,

which means knowledge is frequently reused. Just consider that if a user asks the LLM to read a book, it is unlikely that they will only ask one book-related question.

- The size of KV caches increases slower than the prefill delay. As the context increases, the KV cache size grows linearly whereas the prefill delay grows superlinearly. And as the LLM gets bigger, more compute will happen at the feedforward layers which do not affect the KV cache size.

Why storing KV cache locally is not enough?: We have already shown how to efficiently offload the KV cache locally in Chapter 4. However, only offloading the KV cache to local storage is far from enough in practice.

First, the local storage in the serving engine is very limited and thus cannot store too many KV caches. In production deployments, a serving engine instance usually only has less than one hundred GBs of CPU memory and disk [7]. Even if we use all of the storage, a serving engine instance can only store the KV cache of 500K tokens (approximately 5-6 PDF reports) when using a small model such as Llama-34B.

Second, a locally stored KV cache will be much less useful if it cannot be shared across different serving engine instances. In real-world workloads, requests with the same context are very likely to be executed by different serving engine instances. For instance, a knowledgebase management application will use the same document page as the context to answer the questions from different users; in chatbot applications, the user's request could be routed to a new serving engine instance due to load balancing while the KV cache of the chatting history is still kept in the old serving engine.

Therefore, to fully unleash the potential of KV cache reuse, it is necessary to have a distributed KV cache management component to provide a large enough storage pool for KV caches and share the KV cache across different serving engines.

5.2 Practical Challenges and Overview of KDN

Despite the promise, there are still some unclear design questions and technical limitations in building such a distributed KV cache management component.

- *Substantial size of KV caches:* As a high-dimensional tensor, the size of the KV cache is millions of times larger than its corresponding text, and can easily scale larger than 10 GB when the number of tokens increases. Directly storing/loading the KV caches to disk or remote servers would significantly suffer from the limited bandwidth for loading the KV caches into GPU memory.

- *Prefix-only reusing:* To reuse the KV cache, most systems require that the text of the KV cache must be the *prefix* of the LLM input. Even though reusing the KV cache of the prefix is naturally supported by LLMs, this assumption of "sharing prefix only" severely limits its use cases. For instance, retrieval-augmented generation (RAG) concatenates *multiple* retrieved text chunks in the LLM input, so most reused texts will not be the prefix.

- *Degraded quality with long contexts:* Finally, as more texts are added to the input as LLM's context (*i.e.,* long context), the LLM's capability to capture important information might degrade, lowering the inference quality. Thus, when the KV caches are reused by more queries repeatedly, the degraded quality will also affect more queries.

To address the above issues, we present Knowledge Delivery Network (KDN), a distributed KV cache management system, which dynamically compresses, composes, and modifies KV caches to optimize the storage and delivery of KV caches. As depicted in Figure 2, the envisioned architecture of a KDN consists of three main modules:

- The ***storage*** module stores the KV caches keyed by various texts and offline *modifies* the KV cache content such that the inference quality will be improved when the KV caches are fed to the LLM.

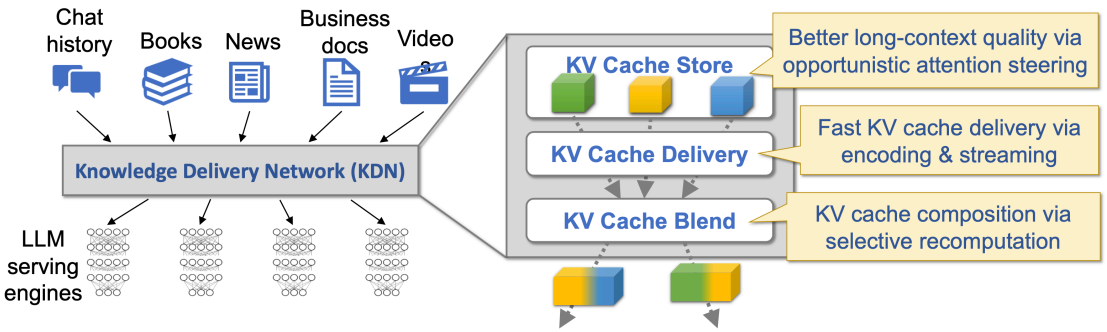


Figure 5.1: The architecture of Knowledge Delivery Network (KDN).

- The ***delivery*** module transmits the *compressed* KV caches from the storage device to the server running the LLM and decompresses them in GPU to be used by the LLM serving engine.
- The ***blending*** module dynamically *composes* multiple KV caches corresponding to different texts when these texts are put together in the context of an LLM query.

The following sections will introduce the detailed designs and implementations of the three components in KDN.

5.3 Interface Design of KDN

In this section, we discuss the interface design of KDN by answering the following two questions: *(i)* how does the serving engine interact with KDN, and *(ii)* how does KDN manage the KV cache internally.

External interfaces towards serving engines: KDN provides 4 functions for serving engines to interact with, including:

```
store(tokens: Tensor, mask: Tensor, kv_tensor: Tensor)
retrieve(tokens: Tensor, mask: Tensor) -> Tensor
lookup(tokens: Tensor, mask: Tensor) -> Tensor
prefetch(tokens: Tensor, mask: Tensor)
```

The serving engine usually calls `store()` after it offloads the KV cache via the `offload()` interface we introduced in Chapter 4. This function is used to store the KV cache tensor into KDN. To avoid blocking the execution of the serving engine, `store` is designed to be a non-blocking function. Similarly, `retrieve()` is called to retrieve the KV cache into a tensor stored in local page-locked CPU memory before the serving engine calls `inject()` to inject the KV cache back to itself. But unlike `store`, `retrieve` is a blocking function. This is because we made an assumption that if the serving engine needs a KV cache, it cannot do anything else but wait for this KV cache to be ready.

`lookup()` is used for checking the existence of the KV cache in KDN.

KDN also supports `prefetch()`, which launches a background task to retrieve the KV cache from remote storage to local CPU RAM. The prefetch functionality can significantly improve GPU utilization as it avoids letting the serving engine wait for loading the KV cache from the remote server (which may take a long time).

Internal interfaces to manage KV caches: Internally, KDN manages the KV caches by *chunks*. Each KV cache chunk corresponds to a fixed number of tokens and is indexed by its corresponding tokens plus all the prefix tokens. When a new KV chunk comes, KDN will serialize it to an array of bytes (will be introduced later), and store the bytes in one of the storage devices (will be introduced later). KDN also stores the metadata for each KV cache chunk, which includes the model-related information, where the KV cache is stored, what kind of serialization technique is applied, and the positional information of the corresponding tokens. The metadata also serves as an extendable interface for developers to implement new management logics in KDN.

The rationale behind chunk-based KV cache management is that the requests will only share the prompts *partially*. For instance, if we store the serialized KV cache of a long document as a single chunk, we will need to deserialize the whole KV cache and return only a part of it to the serving engine when a new request only uses the first half of the document.

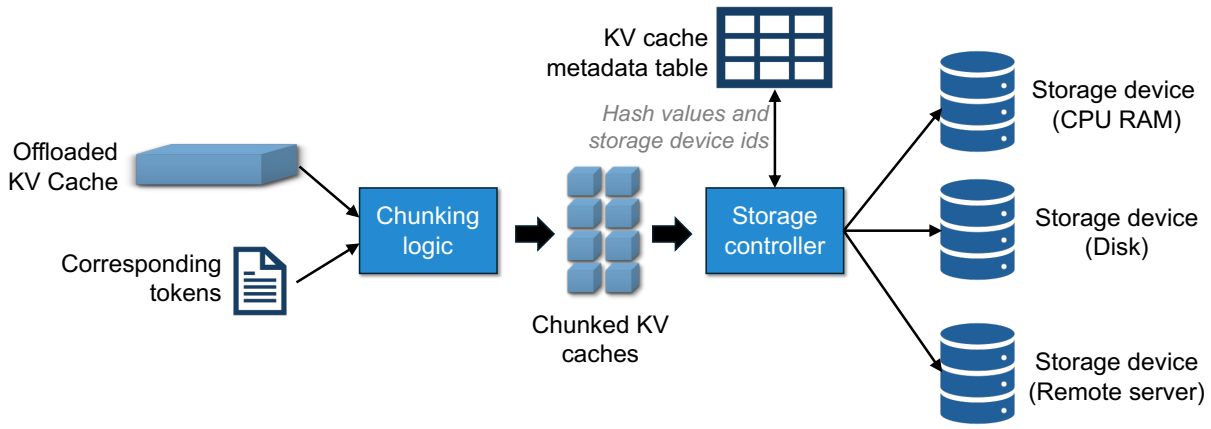


Figure 5.2: Main components inside KDN.

Chunking the KV caches and storing them chunk-by-chunk provides more flexibility to match partially used prompts. In practice, we use a chunk size ranging from 128 to 512 tokens, as it provides a decent hit rate while keeping a minimized overhead by avoiding having too many chunks.

Multi-tier KV cache storage: In a distributed setup, the cluster may have multiple different tiers of storage devices with different capacities and bandwidth, including GPU memory, CPU memory, local SSD, and remote storage. In KDN, different storage devices all share the same abstraction of a key-value store, where the key is generated based on the hash of the tokens plus the metadata, and the value is the serialized KV chunk. We have implemented a storage controller to manage the KV caches across different storage devices. When a KV cache chunk comes, the storage controller tries to store the KV cache in the fastest storage device. If the target storage device is full, the storage controller will use LRU to evict some KV chunks to other storage devices (usually the slower but larger ones). A KV cache chunk will be moved to faster storage when requested by a serving engine.

Based on the design above, Figure 5.2 presents the main components and the workflow of KDN. When a KV cache comes, the chunking logic will first chunk the KV cache and the tokens. Then, the storage controller determines where to store those KV cache chunks

Operation	Bandwidth	Time
Prefill on GPU	N/A	5.8s
Load from paged-locked CPU memory	15 GB/s	0.4s
Load from SSD	1 GB/s	6s
Load from remote storage server	0.6 GB/s	10s

Table 5.1: Comparing the time of loading KV cache from different storage devices and prefilling the text.

and evict recently not-used KV cache chunks to slower storage devices if necessary. The storage controller also calculates the key for each KV cache chunk based on the tokens and the metadata, and updates the metadata table to reflect the locations of each KV cache chunk. It also updates the metadata table to reflect the locations of each KV cache chunk.

When a retrieve request arrives at the KDN, the chunking logic splits the tokens into chunks. Then, the storage controller looks up the metadata table and loads the KV cache chunks from the storage devices. Finally, the chunking logic concatenates the KV cache chunks and returns the concatenated KV cache tensor.

5.4 KV Cache Compression

As we mentioned in Section 5.2, the size of the KV cache is much larger than its corresponding texts, making it inefficient to transmit and expensive to store. To better understand the problem, Table 5.1 shows the speed comparison of loading a KV cache of 25K tokens from different storage devices vs. prefilling the original text using GPU. The number is measured on the server that has an Nvidia A100 GPU and runs a small LLM (Llama-34B), and the size of the KV cache is 6 GB. As shown in the table, loading the KV cache from the remote storage server is almost two times slower than prefilling it on the GPU. Therefore, it would be impractical if we were to directly store the KV cache to slow storage devices as is.

To address this problem, our key insight is that *the KV cache is compressible*. More specifically, we make the following observations:

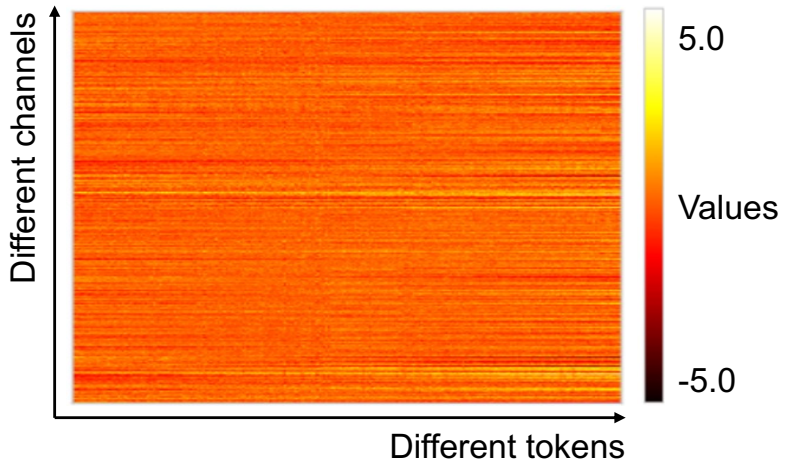


Figure 5.3: Visualizing the K cache generated by Llama-3.1-8B model. The horizontal "stripes" in the figure show that the adjacent tokens have similar values in the same channel.

- *Values in the KV cache are not randomly distributed.* Figure 5.4 shows a snapshot of the values in one of the K cache layers generated by the Llama-3.1-8B model. It shows a clear pattern (the "strips " in the figure) that adjacent tokens have similar values in the same channel, while values in different channels are very different. Therefore, compared to compressing all the values as a single group, grouping the values of adjacent tokens and compressing them group by group can drastically reduce the size of the compressed KV cache. This is because values in each group are much more similar and thus have lower entropy (*i.e.,* bits per value after compression).

- *LLMs are not sensitive to some data loss in KV cache.* More specifically, we observe that slightly changing the values in the KV cache only creates a negligible impact on the LLM's output quality. Furthermore, the LLM is less sensitive to changes in deeper layers than shallower layers. Figure **??** shows that when we erase the values in some of the layers, the quality drop in LLM's output will be small. Therefore, one can apply quantization [139, 92] to reduce the entropy of the KV cache without making large impacts on the quality of LLM's outputs.

Based on the observations, we argue for the opportunity to compress the KV cache by

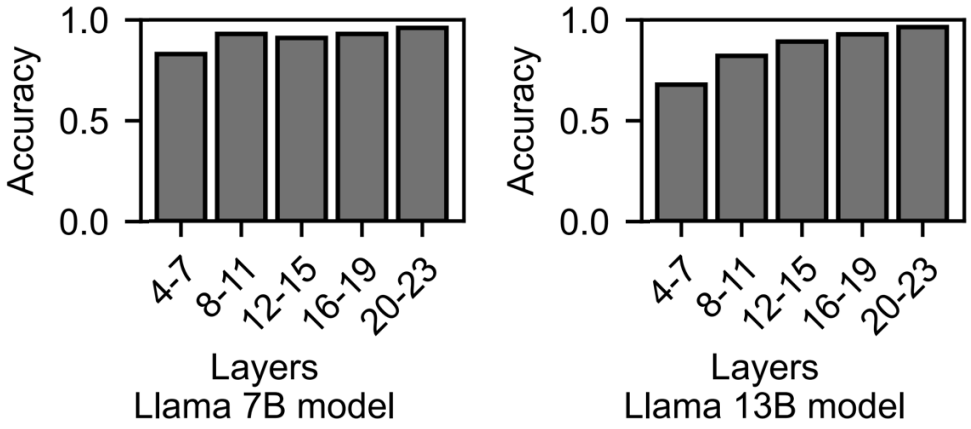


Figure 5.4: Erasing values in different layers only negligibly impacts the quality of LLM-generated output. The accuracy is measured from 100 questions randomly sampled from LongChat [84] dataset.

applying quantization and group-by-group compressions without impacting the normal LLM inference.

In practice, KDN adopts the compression techniques from CacheGen [92], which compresses the KV cache to a byte stream via the following steps. First, it calculates the "delta tensors" between the K and V tensors of nearby tokens, where the delta tensor stores the delta of the value between two consecutive tokens. Second, it applies different levels of quantization to different layers of the delta tensors. Third, it groups the values by each layer and each channel and then runs a lossless arithmetic coder [119] to encode the quantized delta tensors into bitstreams.

CacheGen provides a good algorithm to compress KV cache by 3-6× without impacting the performance of LLM output quality. However, it suffers from performance overheads in practice. CacheGen relies on `torchac` [95] to execute the arithmetic encoding and decoding step, where every value in the KV cache will be encoded (or decoded) sequentially using CPU. Compressing a KV cache of 10K tokens from a 7B model takes tens of seconds when using torchac. In KDN, the KV cache compression speed should also match the speed at which the serving engine generates the KV cache. Therefore, we have implemented performance

	Compression time	Decompression time
Original CacheGen implementation	7.81s	4.52s
Our implementation	0.071s **(110× faster)**	0.062s **(73× faster)**

Table 5.2: In KDN, our GPU-based KV cache compression and decompression kernel achieves 100× speed up compared to the original implementation in CacheGen, making it practical to compress and decompress KV caches in real-time.

optimizations to make the compression practical in KDN.

GPU-based compression: We have implemented specialized CUDA [12] kernels for running the arithmetic coding step on the KV cache chunks. Instead of sequentially processing every value in the KV cache, we break the values into many small groups, map each GPU thread process to a group, and run arithmetic coding within each group in parallel. Each small group contains the values from 256 consecutive tokens of the same layer and same channel. By grouping, we can utilize massive GPU threads to run the arithmetic coding process in parallel, substantially speeding it up. For instance, the KV cache from the Llama-3.1-8B model has 32 layers and 1024 channels. Therefore, to compress the tokens, our implementation can launch 32768 threads at the same time, which is orders of magnitude faster than using the torchac library. Table 5.2 shows that our GPU-based implementation is 70-100× faster than the original CacheGen when compressing and decompressing the KV cache, which is much faster than the serving engine's speed of generating KV caches.

As we will show in Section 5.6, the KV cache compression technique can significantly improve the efficiency of KDN and speed up the LLM inference under a distributed environment.

5.5 Flexible Composition of KV Cache

In today's LLM workloads, the reused part of the input prompt is located not only at the beginning but also in the middle or at the end of the prompt. For instance, in multi-turn

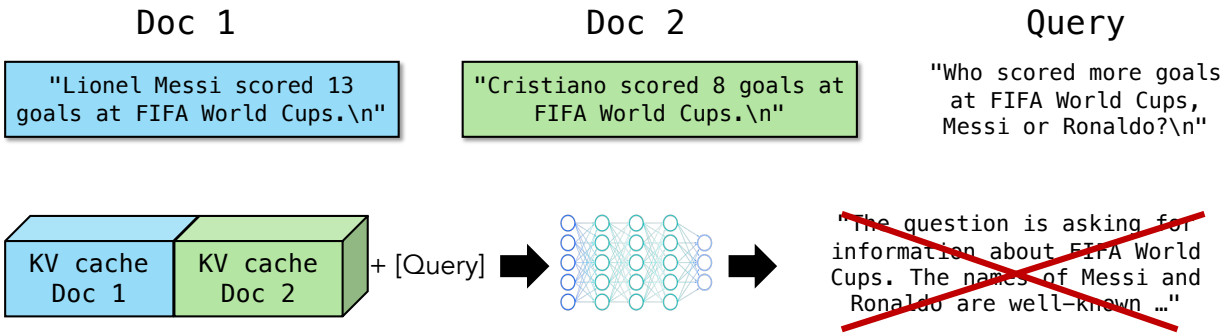


Figure 5.5: An illustrative example of an LLM input with two text chunks prepended to a query. If we directly concatenate the KV cache of the text chunks together, the LLM will give wrong answers.

chat applications, the user's chatting history is reused every time the user types a new query, but the application usually prepends a system prompt, some parts of which may change over time, such as timestamps. Another example is retrieval-augmented generation (RAG), which concatenates multiple retrieved chunks together in the input prompt so that they do not become the prefix except the first chunk.

However, transformers do not support reusing the non-prefix KV cache. Moreover, when the KV caches of multiple prompts are directly concatenated together, the LLM cannot understand the input well and will start generating wrongful outputs. Figure 5.5 gives a real example.

Such a limitation in transformers seems to hinder the practical use of KDN fundamentally. Fortunately, some recent research works improve the composability of the KV caches. CacheBlend [129], for instance, enables arbitrarily composing different KV caches by recomputing the cross-attention between KV caches, where the recomputation only needs 10% computation of prefilling the full text. PromptCache [64] lets users define a prompt template with different segments, which allows each segment's KV cache to be reused at different positions rather than prefix-only.

We have integrated the techniques in CacheBlend to KDN. Based on CacheBlend, KDN opens a new interface named `compose()`, which takes in a list of prompts as input and

returns the composed KV cache back to the serving engine for generation.

5.6 Performance Evaluation of KDN

In this section, we evaluate the benefits brought by KDN. Concretely, we show that

- Across 8 different datasets for long document Q&A and RAG-based Q&A use cases, KDN reduces the response delay 2.1-4.6× while maintaining the same response quality.
- Under the same hardware and same SLO requirements, KDN helps the serving engine serve 1.7-3× more queries.
- Compared to prefilling the text using GPU, storing KV cache in KDN can save the end-to-end cost by 5-20*times* when the KV cache reuse ratio ranges from 50% to 100%.

Baseline: We mainly compare between the following baselines:

- *Official vLLM*: running the official vLLM without any modifications.
- *vLLM with KDN*: running the vLLM engine with KDN as a backend to provide the KV caches.

Testbed setup: We use an NVIDIA A40 GPU server with four GPUs to run a Llama-3-70B model with tensor parallelism of 4 in vLLM. The server is equipped with 384GB of memory and two Intel(R) Xeon(R) Gold 6130 CPUs with Hyper-threading and Turbo Boost enabled by default. In the meantime, we set KDN with a remote storage server of 500 GB capacity as the storage device. The bandwidth between the serving engine and the remote storage server is 5 Gbps.

Datasets: We evaluate the performance of the baselines across 8 different datasets from 2 featured LLM use cases: long-document Q&A and RAG-based Q&A.

For long-document Q&A, we used the following datasets:

- *LongChat:* The task is recently released [84] to test LLMs on queries like "What was the first topic we discussed?" by using all the previous conversations as the context. Most contexts are around 9-9.6K tokens.

- *TriviaQA:* The task tests the reading comprehension ability of the LLMs [44], by giving the LLMs a single document (context), and letting it answer questions based on it. The dataset is part of the LongBench benchmark [44] suite.

- *NarrativeQA:* The task is used to let LLMs answer questions based on stories or scripts, provided as a single document (context). The dataset is also part of LongBench.

- *Wikitext:* The task is to predict the probability of the next token in a sequence based on the context consisting of relevant documents that belong to a specific Wiki page [96].

For RAG-based Q&A, we used the following datasets:

- *2WikiMQA* [69]: This dataset aims to test LLM's reasoning skills by requiring the model to read multiple paragraphs to answer a given question. We included 200 test cases, following the dataset size of previous work [**?**].

- *Musique* [121]: This is a multi-document question-answering dataset. It is designated to test LLM's multi-hop reasoning ability where one reasoning step critically relies on information from another and contains 150 test cases.

- *SAMSum* [66]: This dataset comprises multiple pairs of dialogues and summaries, and requires the LLM to output a summary to a new dialogue. It is intended to test the few-shot learning ability of language models and contains 200 test cases.

- *MultiNews* [59]: This dataset consists of news articles and human-written summaries of these articles from the site newser.com. Each summary is professionally written by editors and includes links to the original articles cited and contains 60 sampled cases.

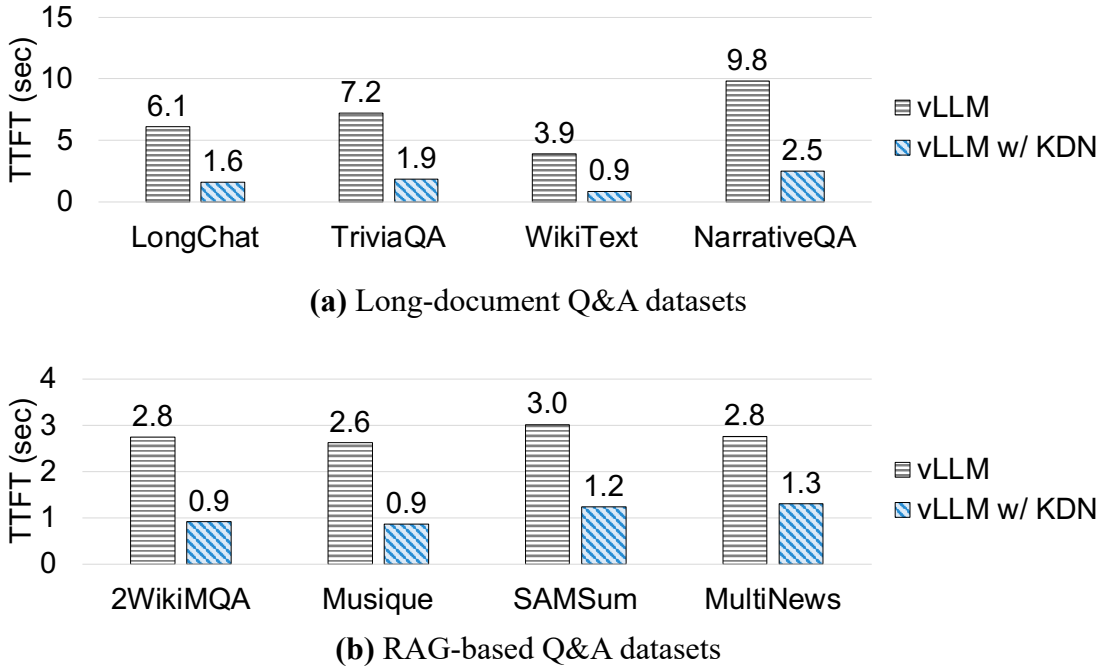


Figure 5.6: Across 8 datasets from long-document Q&A and RAG-based Q&A use cases, KDN helps reduce the time-to-first-token (TTFT) by 2.1-4.6×.

TTFT reduction: Figure 5.6 shows the end-to-end performance gain of KDN. Across 4 documents in the long-document Q&A use case, KDN reduces the time-to-first-token (TTFT, *i.e.,* LLM's response delay) by 3.8-4.6× compared to official vLLM. With the capability to flexibly compose the KV caches, KDN also reduces the TTFT by 2.1-3.1× across 4 datasets in the RAG-based Q&A use case.

Impact on LLM output quality: Although KV cache compression and KV cache composition techniques in KDN may impact the LLM's output quality, Table 5.3 shows that the quality degradation is negligible, ranging from 0.3-2.0%.

Serve more queries with KDN: By reusing the KV cache, KDN allows the serving engine to skip the GPU-intensive prefill stage. Therefore, with KDN, the serving engine can handle more queries using the same hardware compared to the baseline. Figure 5.7 compares the maximum queries per second without violating the SLO on 4 different datasets from the long-document Q&A and RAG-based Q&A use cases, where the SLO is set to TTFT less

Dataset	Quality metric	Quality w/o KDN	Quality w/ KDN
LongChat	Accuracy, higher is better	0.97	0.95
TriviaQA	F1 score (%), higher is better	89.9	89.5
WikiText	Perplexity, lower is better	4.25	4.35
NarrativeQA	F1 score (%), higher is better	24.1	24.0
2WikiMQA	F1 score (%), higher is better	32.8	32.4
Musique	F1 score (%), higher is better	30.8	30.9
SAMSum	RougeL-score, higher is better	0.388	0.383
MultiNews	RougeL-score, higher is better	0.205	0.201

Table 5.3: **Quality impact**: Across 8 datasets, KDN only creates a negligible impact on the LLM's output quality.

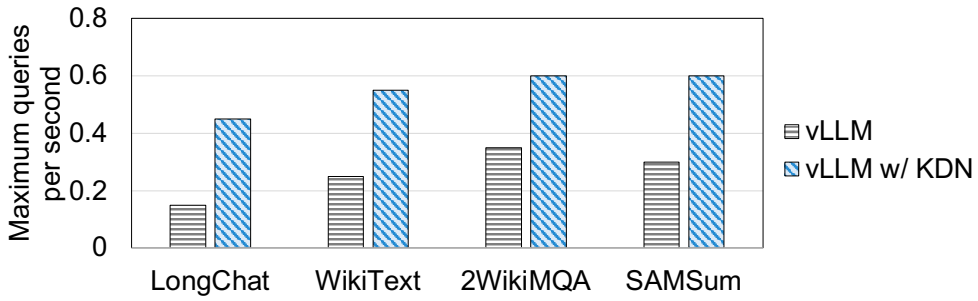


Figure 5.7: Across 4 different datasets, the serving engine can serve 1.7-3× more queries using the same hardware with the help of KDN.

than 10 seconds and generation speed larger than 25 tokens per second per request. With the help of KDN, the serving engine can serve 1.7-3× more queries using the same hardware compared to running the official vLLM.

End-to-end serving cost: Although reusing the KV cache can skip the costly prefill stage, storing them also introduces extra storage costs in practice. Therefore, We further analyzed the end-to-end serving cost based on the on-demand GPU and storage prices on Amazon Web Services [8]. Figure 5.8 shows the cost per hour if we query the serving engine at 1 request per second using the WikiText [96] dataset. By storing and reusing the KV cache, KDN can save the cost by 4.31×, 11.6×, and 21.52× respectively when the ratio of the reused contexts are 50%, 80%, and 100%.

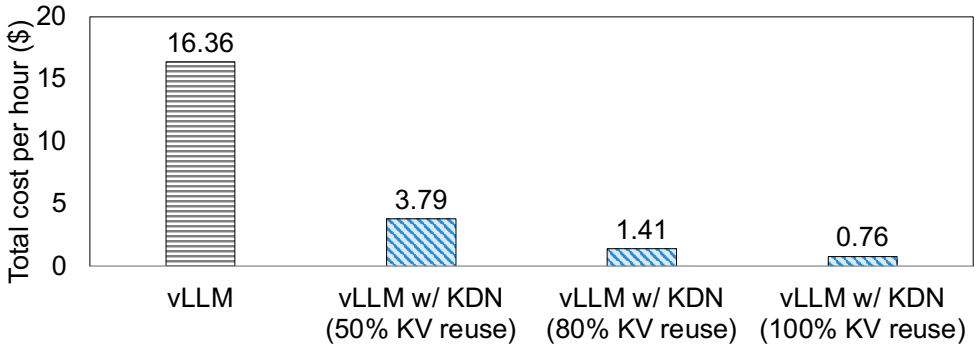


Figure 5.8: Compared to prefilling the text on GPU, KDN reduces the end-to-end serving cost by storing and reusing the KV cache.

5.7 Discussion and Future Work

More compression techniques: The recent KV cache compression techniques make it possible to cheaply store and quickly load KV caches outside GPU and CPU memory. KDN implements CacheGen [92], which compresses the KV cache by encoding it into binary strings. In the meantime, there are a lot of KV cache compression techniques that are complementary to CacheGen. LLMLingua [74] introduces a smaller language model to identify and remove non-essential tokens in the knowledge's text, thus reducing the size of the corresponding KV cache. H2O [140] directly removes elements in the KV cache based on their importance calculated during the inference. By combining the above techniques with CacheGen, the memory footprint of the KV cache can be reduced by over 10×, drastically improving the loading speed and the storage cost of the KV cache.

Improving output quality by KV cache editing: By separating KV cache management from the serving engines, KDN opens up new possibilities to improve inference quality. Recent works have shown that if the attention matrix of an LLM input is appropriately modified, the inference quality can significantly improve [137, 41]. In other words, when a KV cache is "deposited" to KDN, KDN not only stores it but also can actively influence the LLM inference quality by offline editing the KV cache and returning the edited KV cache when it is retrieved next time, all of which is done without any change to the model itself or the input prompt.

Cache performance optimizations: Although KDN already provides substantial performance benefits as shown in Section 5.6, there are still a lot of opportunities to improve its performance. First, while KDN still uses traditional TCP/IP stack to transmit KV caches, emerging hardware techniques such as NVLink [29], InfiniBand [103], GPU Direct Storage [118], and GPU Direct RDMA [15] can help transmit KV cache between GPU and different storage devices much more efficiently. Second, existing works show that a good caching policy can boost the performance of a multi-tier storage system [40, 58]. As a multi-tier storage system, KDN can also benefit from those techniques.

5.8 Summary

In this chapter, we have first argued that storing the KV cache locally cannot fully exploit the opportunity of prompt reuse, thus motivating the solution to store KV caches in KDN, a distributed and shared system. Then, we have analyzed the challenges in KDN.

We have presented the design of KDN by talking about how it interacts with the LLM serving engines, how it stores the KV cache internally, and how it manages different storage devices. To address the practical challenges, we have shown two key techniques used in KDN: KV cache compression and KV cache composition. We have used real-world datasets and workloads to evaluate the performance gain introduced by KDN.

We have also discussed the future works to further optimize the performance of KDN, and we believe that having a standalone KV cache management module like KDN is the key to serving distributed LLM inference service efficiently.

CHAPTER 6

GLOBAL ORCHESTRATOR FOR DISTRIBUTED LLM SERVING

In this chapter, we will introduce how LMStack orchestrates a distributed LLM inference stack by the *global orchestrator*. At a high level, the global orchestrator receives the requests from the users, forwards the user's request to the serving engines, and returns the LLM's response to users.

The following of this chapter introduces the three key functionalities of the global orchestrator. Section 6.1 presents how the global orchestrator coordinates the serving engine and KDN when doing KV cache prefetching. Section 6.2 talks about the design and the implementation of the KV-cache-aware load balancing algorithm in the global orchestrator. Section 6.3 shows how the global orchestrator migrates the request from a serving engine instance to another one and enables better fault tolerance based on the request migration.

Finally, we present the implementation of LMStack in Section 6.4. We also show how LMStack is integrated into Kubernetes, the state-of-the-art production deployment platform.

6.1 KV Cache Prefetching

In this section, we present how the global orchestrator embraces the prefetch functionality provided by KDN to improve the efficiency of distributed LLM inference.

6.1.1 Motivation

As we discussed in previous chapters, KV cache is large, and it would take a long time to load a KV cache of a long context into GPU. Therefore, it will be too late if KDN starts loading the KV cache after the request is getting executed in the serving engine—the serving engine has to keep the GPU idle and wait for the KV cache to be ready in GPU memory

before inference. This could significantly decrease GPU utilization as most KV caches reside in slow but high-capacity devices such as disk or remote storage servers, and thus it requires a long time to load them into GPU even with all the optimizations in KDN.

Therefore, the prefetch functionality provided by KDN can significantly improve GPU utilization as it allows the serving engine to execute other requests during the loading of the KV cache.

6.1.2 Key Design

Despite the promises of prefetch, a practical problem is when and where to call the `prefetch` function in KDN. A straightforward design is to let the serving engine call the prefetch function since KDN is directly integrated with the serving engines. However, this design may lead to multiple problems.

First, today's serving engine uses convoluted scheduling algorithms to determine when to execute which request, and each engine implements this algorithm differently. Integrating the prefetching logic into the serving engine means that we need to couple the prefetch logic with the implementation details inside each serving engine. This not only increases the development complexity but also limits evolvability, as functionality updates in the serving engine schedulers may break the prefetching logic.

Second, some serving engines, such as vLLM and HuggingFace TGI, implement synchronized schedulers, meaning that the scheduler and the model inference are executed sequentially. However, prefetch logic is asynchronous, as it launches a prefetch task in parallel with the normal serving engine execution. It also requires the caller to be able to execute callback functions after the KV cache is ready.

Therefore, in LMStack, we decide to implement the prefetching functionality in the global orchestrator instead of inside the serving engine. Figure **??** shows the workflow of how global orchestrator manages the prefetching. When the global orchestrator receives a request, it

first looks up the KV cache by calling the `lookup` interface provided by KDN (Section 5.3). If the KV cache exists in KDN, the global orchestrator will then call KDN's `prefetch` interface to prefetch the KV cache into a serving engine (how global orchestrator determines which serving engine to prefetch to in the next section will be discussed in the next section). KDN will notify the global orchestrator after the prefetch is finished, and then the controller will send the request to the serving engine.

In this case, the prefetch process will not bother the serving engine's workflow because the serving engine will not receive the requests whose KV cache is not prefetched.

6.2 KV-Cache-Aware Load Balancing

In this section, we present how the global orchestrator routes the request to different serving engine instances to balance the load between the instances while maximizing the efficiency by KV cache reuse.

6.2.1 Motivation

The computation requirement of a request will be the same across all the serving engine instances if the KV cache has already been prefetched to the instance. However, smartly routing the request to different serving engine instances is still important because of the following reasons.

- *Achieve better load balancing*: In practice, different serving engine instances may have very different loads at the same time, because requests may have very different prompt lengths and generation lengths. Therefore, smartly routing a request to the instance with a low load can achieve better load balancing and avoid overloading some serving engines.

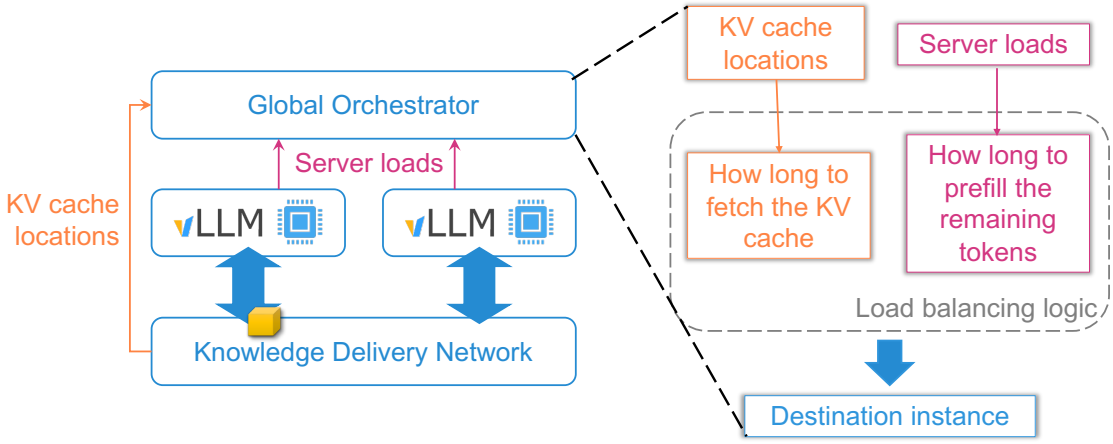


Figure 6.1: Design of the KV-cache-aware load balancing router

- *Reduce the network cost*: Although KDN supports prefetching the KV cache to any serving engine instances, the cost of such prefetching may differ for different serving engines. This is because the KV cache may reside in storage that is local to one of the serving engine instances. Routing the request to that instance will avoid extra network transmission, and alleviate the load of transmitting large KV caches across the cluster.

- *Reduce the end-to-end delay*: When the KV cache is stored local to a serving engine instance, routing the request to that instance will be faster, and thus having reduced end-to-end delay than routing to other instances.

Therefore, the global orchestrator implements a KV-cache-aware load balancing algorithm for request routing, which balances the load of the serving engine instances while minimizing the overall cost of running the request.

6.2.2 Key Design

Figure 6.1 illustrates how global orchestrator supports KV-cache-aware load-balancing. The serving engine monitor keeps monitoring the load of the serving engines, including the number of running requests, the number of queuing requests, internal batch size, and the KV cache usage. In the meantime, the global orchestrator counts the rate of incoming requests and

finished requests for each serving engine instance. The global orchestrator aggregates the above information to determine the set of the serving engine instances that are available to take new requests. More specifically, a serving engine that meets one of the following conditions will be considered as "non-available":

- Have more than 1 queuing request in the last 30 seconds.
- Internal batch size hits the maximum batch size in the last 30 seconds.
- The rate of incoming request is larger than 1.2× rate of finished requests in the last 30 seconds.

Once the global orchestrator receives a new request, it will first query KDN for the existence and storage location of the KV cache. It will then estimate a time that sums the following two parts:

- The time to load the KV cache to the GPU of that serving engine instance.
- The time to refill the request on the serving engine with the loaded KV cache.

The first part can be calculated by dividing the size of the KV cache by the bandwidth between the storage device and the target GPU. The second part can be estimated by the number of tokens that need to be prefilled divided by the prefilling throughput at the serving engine, where the prefilling throughput can also be monitored from the serving engine.

We have implemented the KV-cache-aware load balancing logic with 2000 lines of Python code, and it is open-sourced at `https://github.com/vllm-project/production-stack/tree/main/src/vllm_router`.

6.3 Flexible request migration

In this section, we present the request migration functionality in global orchestrator. This functionality allows a request to be quickly migrated to another server without extra com-

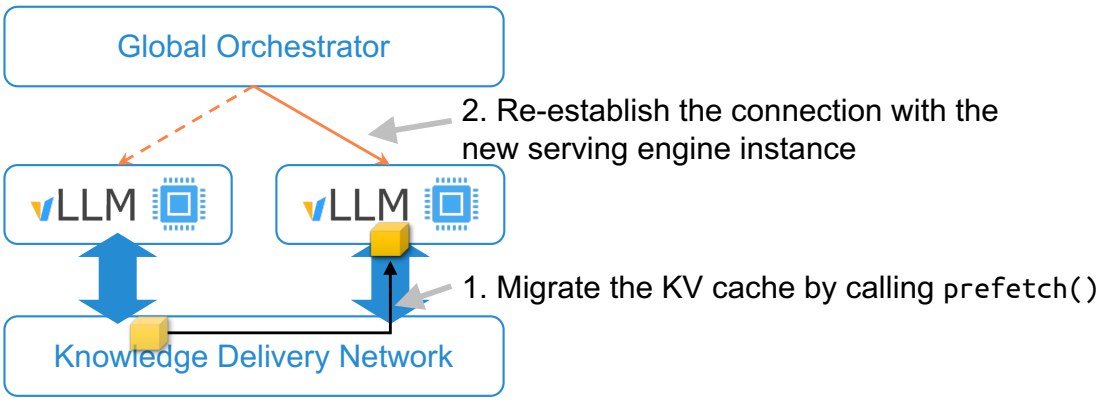


Figure 6.2: Illustration of how global orchestrator does the request migration.

putation overhead. We also show that such migration can enable better fault tolerance when failures happen in the serving engines.

6.3.1 Motivation

In the production deployment of LLMs, the serving engines frequently shut down due to failures or runtime errors.

On the one hand, when the serving engine shuts down, the connection between users and the serving engine will be suddenly closed and cannot be resumed. The user has to resend a new query to another LLM instance again. Moreover, all the KV caches stored in the serving engine will be lost, meaning that the progress of the unfinished request will all be lost. Therefore, when the request is processed by another serving engine, the user has to wait for a long prefill delay again.

We argue that there is a key missing functionality—seamlessly migrating a request with its progress (*i.e.,* KV cache) from an existing serving engine to a new serving engine. With such functionality, the requests can be migrated to a new serving engine when the original serving engine shuts down without interrupting the users.

6.3.2 Design

The state of a running request consists of three parts: the input prompt, the LLM-generated texts, and the KV cache of the request. Therefore, to migrate the request, one has to transfer all three parts to the new serving engine. In LMStack, the migration of the KV cache is done by KDN, and the other two parts will be migrated by the global orchestrator.

Figure 6.2 illustrates the workflow of migrating a task in LMStack. The migration is triggered by calling `migrate`, an interface provided by the global orchestrator. When the interface is called, global orchestrator first calls KDN to launch the prefetch job that transfers the KV cache to the new serving engine. When the prefetch job finishes, the global orchestrator creates a new connection to the new serving engine, and sends the prompt plus the already generated text through the new connection. Since users directly interact with the global orchestrator rather than the serving engine, such request migration is transparent to the users.

6.3.3 Better Fault Tolerance

With the request migration support in the global orchestrator, LMStack provides much better fault tolerance for the distributed LLM serving that does not interrupt users when failure happens in the serving engine.

As we discussed in earlier sections, the serving engine will offload the KV cache to KDN during the execution of the request. Therefore, the KV cache can still be available in KDN even if the serving engine fails. In the meantime, when the global orchestrator detects the failure in the serving engine (*e.g.,* when the connection breaks), it will call the request migration logic to migrate the requests to other serving engines. To determine which serving engine instance to migrate to, the global orchestrator uses the KV-cache-aware routing algorithm described in Section 6.2.

Table 6.1 shows the benefit under a real workload. The numbers represent the stall time

	User session interrupted	Generation stall time
w/o LMStack	Interrupted	N/A (need session restart)
LMStack w/o KV migration	Not interrupted	7.2 secs
LMStack w/ KV migration	Not interrupted	1.7 secs

Table 6.1: With task migration, LMStack can provide a much better user experience when failure happens in one of the serving engines—the user session is not interrupted and the generation stall time can be decreased from 7.2 seconds to 1.7 seconds (4.2× improvement).

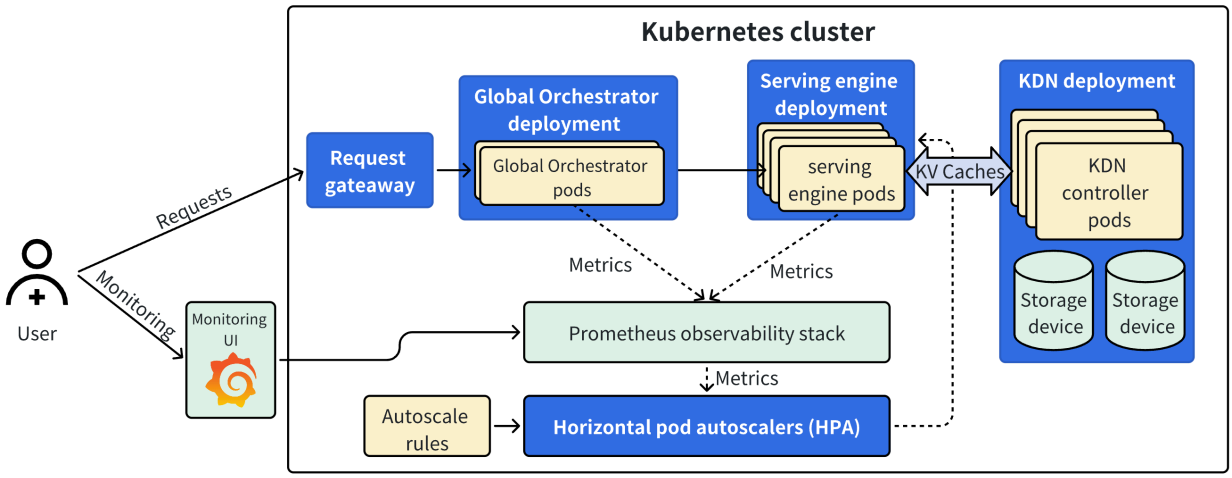


Figure 6.3: Main components in LMStack when deploying in a Kubernetes cluster.

caused by the failure. With LMStack, the stall time can be reduced by 4.2×, from 7.2 seconds to 1.7 seconds.

6.4 Implementation and production deployment of LMStack

In the production environment, people do not directly deploy the applications on physical machines. Instead, most of the deployments rely on *Kubernetes* [5], an open-source system that helps people deploy, scale, and manage containerized applications.

To align with the industry practice, the implementation of LMStack supports Kubernetes-native deployments. Figure 6.3 shows the architecture of LMStack when deploying on a Kubernetes cluster. LMStack uses the Kubernetes Gateway API [3] to handle the requests that are sent to the Kubernetes cluster, and the requests will be directly passed into the

global orchestrator. The global orchestrator, serving engines, and KDN are all packaged into containers and run as different pods [6] inside the cluster. LMStack uses the Kubernetes deployment [1] to manage the pods to provide a high availability. The deployment restarts the pod when the failure happens in the pod.

LMStack also embeds the Prometheus Observability Stack [20] to monitor the loads and statistics of the serving engine. The metrics are used for two purposes. First, they are passed to the horizontal pod autoscalers [4], a component in Kubernetes to scale the number of pods automatically. Second, the metrics are also used to power a dashboard for users to monitor the status of the cluster in real time.

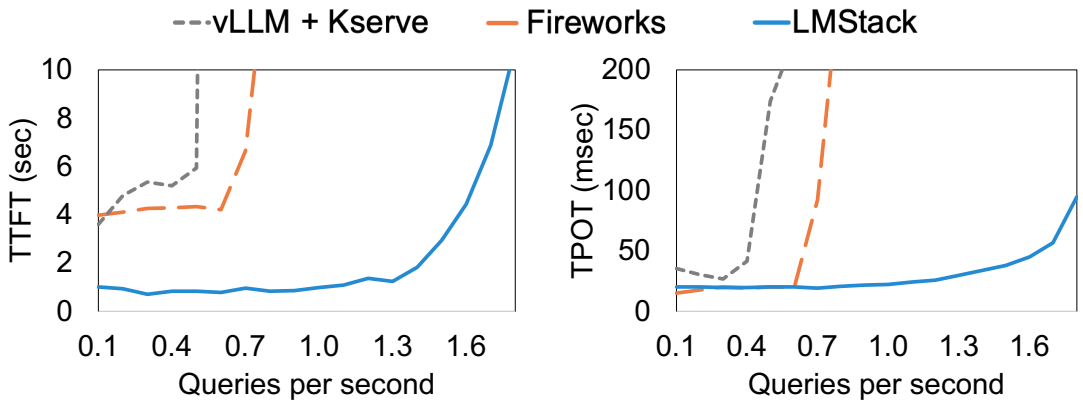
In order to simplify the deployment process of LMStack, we wrapped the Kubernetes configuration files by Helm [135]. Helm lets users deploy multiple Kubernetes components in a single command by generating the Kubernetes configurations from a set of pre-defined templates. With Helm, the LMStack can easily deployed to any Kubernetes in a single command.

Serving engine pods and their IPs can be frequently changed due to automatic scaling and pod failures. To avoid sending requests to dead serving engines or ignoring the newly added pods, the global orchestrator uses Kubernetes API to watch the pod creation and pod failures in real time.

The implementation of LMStack is fully open-sourced at `https://github.com/vllm-project/production-stack`.

Performance evaluation of LMStack: We compare the performance of LMStack with two baselines:

- *Fireworks* [2] is a commercial LLM endpoint service that allows users to deploy production-ready LLM on a distributed cluster.
- *KServe* [23] is an open-source model inference platform for serving predictive and generative AI models on Kubernetes,



(a) Time-to-first-token (**TTFT**) **(b)** Time-per-output-token (**TPOT**)

Figure 6.4: End-to-end performance evaluation shows LMStack can process 2.5-4× more requests with 4× lower TTFT compared to the opensource and commercial alternatives.

To keep a fair comparison, we set up all the baselines on a 2-node cluster with each node having a single Nvidia A100 80G GPU and 256 GB CPU RAM. When running LMStack, we configure the underlying KDN to use a shared KV cache storage of 300 GB in CPU memory.

We run a multi-turn chat workload based on the ShareGPT [49] dataset. In the workload, 80 concurrent sessions interact with the LLM service simultaneously, and each session has a chatting history of 20K tokens.

Figure 6.4 compares the time-to-first-token (TTFT) and the time-per-output-token (TPOT) between the baselines. With all the optimizations described above, LMStack has 4× TTFT compared to other two baselines. In the meantime, by storing and reusing more KV caches of the chatting histories in KDN, LMStack can skip the expensive prefills and thus be able to handle more requests using the same hardware—it achieves 1.6 queries per second without a significant increase in TTFT and TPOT, which is 4.1× and 2.5× better than KServe and Fireworks, respectively.

6.5 Summary

In this chapter, we have introduced the design and implementation of the global orchestrator in LMStack. The global orchestrator handles the requests from the users and coordinates the serving engines and KDN to process the requests. We have presented three key designs in the global orchestrator, including KV cache prefetching (Section 6.1, KV-cache-aware load balancing (Section 6.2), and flexible request migration (Section 6.3).

Finally, we have talked about the implementation of LMStack and how it is integrated into Kubernetes, the standard solution for running distributed services in production.

CHAPTER 7

CONCLUSION

7.1 Contributions

This dissertation contributes by designing and implementing an efficient system for distributed LLM serving. As most of today's LLM serving system optimizations focus only on speeding up a single serving instance, we argue that there are optimization opportunities and techniques unique to distributed deployment. Specifically, a distributed cluster can hold much more KV cache than a single instance, and reusing the KV cache across the instances can substantially improve the system's efficiency by skipping expensive LLM inference. Our key insight is that the performance of distributed LLM inference can be substantially improved by decoupling the KV cache from the LLM serving engines. Based on such insight, we propose LMStack, an end-to-end software stack to deploy and manage the distributed LLM serving engines, which can fully realize the potential of KV cache reuse.

Besides the serving engine itself, LMStack introduces three new components: the KV cache offloading interface, the Knowledge Delivery Network (KDN), and the global orchestrator. The KV cache offloading interface provides an efficient solution to detach the KV caches from the serving engines and inject them back when needed. KDN is a standalone module that dynamically compresses, composes, and modifies KV caches to optimize the storage and delivery of KV caches. The global orchestrator implements optimizations specific to the *distributed* LLM deployments, including cluster-wide KV cache prefetching, KV-cache-aware request routing, load balancing, and fault tolerance.

Putting the pieces together, we implement LMStack to support end-to-end cloud-native distributed LLM serving engine deployments. By realizing the benefit of KV cache reuse and smartly scheduling the incoming requests, LMStack can achieve 3.1-4× lower response delay compared to state-of-the-art solutions. To serve the same number of users, LMStack

can reduce the computation cost by 2.5-4×. In the meantime, LMStack provides better fault tolerance by hiding the serving engine failures from the users and reducing the failure-induced stall time from tens of seconds to less than one second.

7.2 Future Works

There are multiple potential directions related to LMStack that can be explored in the future.

Global-scale KDN: The benefit of reusing the KV cache grows when the capacity of KV cache storage increases. Therefore, we envision a global-scale KDN, which is similar to the content delivery networks (CDNs) nowadays. The key challenge to increasing the scale of KDN is how to efficiently transmit the KV caches given the tremendous size of the KV caches. There is already some research on how to reduce the size of the KV cache [140, 74]. We believe that global-scale KDN will be practical if we efficiently incorporate those methods into KDN.

On-premise deployment on heterogeneous hardware: Although the deployment of LMStack is compatible with Kubernetes, the industry standard solution for distributed production deployments, there are still many scenarios that the current LMStack does not support. For instance, many companies use Ray [98], an emerging distributed framework for AI applications, to deploy their LLM services. Many companies also have specialized hardware that is not natively supported by Kubernetes, such as InfiniBand [103] and GPU direct RDMA [15]. The key to addressing the challenge is to find a minimal common set of interfaces to support different frameworks and heterogeneous hardware, and we will leave this as future work.

REFERENCES

[1] Deployments | kubernetes. URL https://kubernetes.io/docs/concepts/workloads/controllers/deployment/. [Online; accessed 2025-02-09].

[2] Fireworks - fastest inference for generative ai. URL https://fireworks.ai/. [Online; accessed 2025-02-15].

[3] Gateway api | kubernetes. URL https://kubernetes.io/docs/concepts/services-networking/gateway/. [Online; accessed 2025-02-09].

[4] Horizontal pod autoscaling | kubernetes. URL https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/. [Online; accessed 2025-02-09].

[5] Kubernetes. URL https://kubernetes.io/. [Online; accessed 2025-02-09].

[6] Pods | kubernetes. URL https://kubernetes.io/docs/concepts/workloads/pods/. [Online; accessed 2025-02-09].

[7] Using kubernetes — vllm. URL https://docs.vllm.ai/en/latest/deployment/k8s.html. [Online; accessed 2025-02-02].

[8] Cloud computing services - amazon web services (aws). URL https://aws.amazon.com/. [Online; accessed 2025-02-15].

[9] character.ai | personalized ai for every moment of your day. https://character.ai/, . (Accessed on 09/07/2024).

[10] Introducing chatgpt | openai. https://openai.com/index/chatgpt/, . (Accessed on 09/07/2024).

[11] Search - consensus: Ai search engine for research. https://consensus.app/search/. (Accessed on 09/07/2024).

[12] Cuda toolkit documentation 12.8, . URL https://docs.nvidia.com/cuda/index.html. [Online; accessed 2025-01-31].

[13] Cuda runtime api :: Cuda toolkit documentation, . URL https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__STREAM.html. [Online; accessed 2025-02-02].

[14] Cursor - the ai code editor. URL https://www.cursor.com/. [Online; accessed 2025-02-09].

[15] 1. overview — gpudirect rdma 12.8 documentation. URL https://docs.nvidia.com/cuda/gpudirect-rdma/. [Online; accessed 2025-02-08].

[16] [2304.03442] generative agents: Interactive simulacra of human behavior. https://arxiv.org/abs/2304.03442. (Accessed on 09/21/2023).

[17] Enterprise search: an llm-enabled out-of-the-box search engine. https://io.google/2023/program/27cce05f-df4c-4ab2-9a59-5b466bdae0f9/. (Accessed on 09/07/2024).

[18] Gpt-4 api general availability and deprecation of older models in the completions api. https://openai.com/blog/gpt-4-api-general-availability. (Accessed on 09/21/2023).

[19] Helm, . URL https://helm.sh/. [Online; accessed 2025-01-31].

[20] helm-charts/charts/kube-prometheus-stack at main · prometheus-community/helm-charts, . URL https://github.com/prometheus-community/helm-charts/tree/main/charts/kube-prometheus-stack. [Online; accessed 2025-02-09].

[21] Ai assistant for your enterprise | glean work ai, . URL https://www.glean.com/product/assistant?utm_source=google&utm_medium=paid-search&utm_campaign=US-general-ai-productivity-assistant&utm_term=ai%20business%20productivity%20software&gad_source=1&gclid=CjwKCAiAwaG9BhAREiwAdhv6YzYbjW-e0ZnqbQMW8xGYlXWKpRKrPD16nq48atXj3TB-MlIDy1F4UBoCf5MQAvD_BwE. [Online; accessed 2025-02-09].

[22] Ai assistant for your enterprise | glean work ai, . URL https://www.glean.com/product/assistant?utm_source=google&utm_medium=paid-search&utm_campaign=US-general-ai-productivity-assistant&utm_term=ai%20business%20productivity%20software&gad_source=1&gclid=CjwKCAiAwaG9BhAREiwAdhv6YzYbjW-e0ZnqbQMW8xGYlXWKpRKrPD16nq48atXj3TB-MlIDy1F4UBoCf5MQAvD_BwE. [Online; accessed 2025-02-09].

[23] kserve/kserve: Standardized serverless ml inference platform on kubernetes. URL https://github.com/kserve/kserve. [Online; accessed 2025-02-15].

[24] [2302.13971] llama: Open and efficient foundation language models. https://arxiv.org/abs/2302.13971. (Accessed on 09/21/2023).

[25] Applications of large language models - indata labs. https://indatalabs.com/blog/large-language-model-apps, . (Accessed on 09/21/2023).

[26] 12 practical large language model (llm) applications - techopedia. https://www.techopedia.com/12-practical-large-language-model-llm-applications, . (Accessed on 09/21/2023).

[27] 7 top large language model use cases and applications. https://www.projectpro.io/article/large-language-model-use-cases-and-applications/887, . (Accessed on 09/21/2023).

[28] Perplexity. https://lmcache.github.io/2024-09-17-release/. (Accessed on 10/14/2024).

[29] Nvlink & nvswitch: Fastest hpc data center platform | nvidia. URL https://www.nvidia.com/en-us/data-center/nvlink/. [Online; accessed 2025-02-08].

[30] Perplexity. https://www.perplexity.ai/. (Accessed on 09/07/2024).

[31] A guide on good usage of non_blocking and pin_memory() in pytorch — pytorch tutorials 2.6.0+cu124 documentation. URL https://pytorch.org/tutorials/intermediate/pinmem_nonblock.html. [Online; accessed 2025-02-02].

[32] Nvidia/tensorrt-llm: Tensorrt-llm provides users with an easy-to-use python api to define large language models (llms) and build tensorrt engines that contain state-of-the-art optimizations to perform inference efficiently on nvidia gpus. tensorrt-llm also contains components to create python and c++ runtimes that execute those tensorrt engines. URL https://github.com/NVIDIA/TensorRT-LLM. [Online; accessed 2025-02-01].

[33] vllm-project/vllm: A high-throughput and memory-efficient inference and serving engine for llms. URL https://github.com/vllm-project/vllm. [Online; accessed 2025-02-15].

[34] Github copilot · your ai pair programmer, 12 2024. URL https://github.com/features/copilot?ef_id=_k_CjwKCAiAwaG9BhAREiwAdhv6Y1cYMjWJfhN7mmtXkWmTSCJ7yT64yFbPF5dnhK1Sh227zRFPabC62xoCTV4QAvD_BwE_k_&OCID=AIDcmmb150vbv1_SEM__k_CjwKCAiAwaG9BhAREiwAdhv6Y1cYMjWJfhN7mmtXkWmTSCJ7yT64yFbPF5dnhK1Sh227zRFPabC62xoCTV4QAvD_BwE_k_&gad_source=1&gclid=CjwKCAiAwaG9BhAREiwAdhv6Y1cYMjWJfhN7mmtXkWmTSCJ7yT64yFbPF5dnhK1Sh227zRFPabC62xoCTV4QAvD_BwE. [Online; accessed 2025-02-09].

[35] meta-llama/llama-3.1-8b-instruct · hugging face, 12 2024. URL https://huggingface.co/meta-llama/Llama-3.1-8B-Instruct. [Online; accessed 2025-02-02].

[36] pathwaycom/llmapp. https://github.com/pathwaycom/llm-app, 2024. Accessed: 2024-01-25.

[37] Muhammad Adnan, Akhil Arunkumar, Gaurav Jain, Prashant Nair, Ilya Soloveychik, and Purushotham Kamath. Keyformer: Kv cache reduction through key tokens selection for efficient generative inference. *Proceedings of Machine Learning and Systems*, 6:114–127, 2024.

[38] Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, and Ramachandran Ramjee. Sarathi: Efficient llm inference by piggybacking decodes with chunked prefills, 2023.

[39] Toufique Ahmed and Premkumar Devanbu. Few-shot training llms for project-specific code-summarization. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450394758. doi:10.1145/3551349.3559555. URL https://doi.org/10.1145/3551349.3559555.

[40] Abubakr O Al-Abbasi, Vaneet Aggarwal, and Moo-Ryong Ra. Multi-tier caching analysis in cdn-based over-the-top video streaming systems. *IEEE/ACM Transactions on Networking*, 27(2):835–847, 2019.

[41] Anonymous. Model tells itself where to attend: Faithfulness meets automatic attention steering. In *Submitted to ACL Rolling Review - June 2024*, 2024. URL `https://openreview.net/forum?id=vFf7ZDoLA2`. under review.

[42] Anonymous. Chunkattention: Efficient attention on KV cache with chunking sharing and batching, 2024. URL `https://openreview.net/forum?id=9k27IITeAZ`.

[43] AuthorName. Can chatgpt understand context and keep track of conversation history. `https://www.quora.com/Can-ChatGPT-understand-context-and-keep-track-of-conversation-history`, Year. Quora question.

[44] Yushi Bai, Xin Lv, Jiajie Zhang, Hongchang Lyu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, Yuxiao Dong, Jie Tang, and Juanzi Li. Longbench: A bilingual, multitask benchmark for long context understanding. *arXiv preprint arXiv:2308.14508*, 2023.

[45] Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Vageesh D C, Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, B. Ashok, and Shashank Shet. Codeplan: Repository-level coding using llms and planning, 2023.

[46] Sebastian Borgeaud, Arthur Mensch, Jordan Hoffmann, Trevor Cai, Eliza Rutherford, Katie Millican, George van den Driessche, Jean-Baptiste Lespiau, Bogdan Damoc, Aidan Clark, Diego de Las Casas, Aurelia Guy, Jacob Menick, Roman Ring, Tom Hennigan, Saffron Huang, Loren Maggiore, Chris Jones, Albin Cassirer, Andy Brock, Michela Paganini, Geoffrey Irving, Oriol Vinyals, Simon Osindero, Karen Simonyan, Jack W. Rae, Erich Elsen, and Laurent Sifre. Improving language models by retrieving from trillions of tokens, 2022.

[47] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language Models are Few-Shot Learners, 2020.

[48] Jiawei Chen, Hongyu Lin, Xianpei Han, and Le Sun. Benchmarking large language models in retrieval-augmented generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 17754–17762, 2024.

[49] Lin Chen, Jinsong Li, Xiaoyi Dong, Pan Zhang, Conghui He, Jiaqi Wang, Feng Zhao, and Dahua Lin. Sharegpt4v: Improving large multi-modal models with better captions. In *European Conference on Computer Vision*, pages 370–387. Springer, 2024.

[50] Wei-Lin Chiang, Lianmin Zheng, Ying Sheng, Anastasios Nikolas Angelopoulos, Tianle Li, Dacheng Li, Hao Zhang, Banghua Zhu, Michael Jordan, Joseph E Gonzalez, et al. Chatbot arena: An open platform for evaluating llms by human preference. *arXiv preprint arXiv:2403.04132*, 2024.

[51] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022.

[52] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. *arXiv preprint arXiv:1901.02860*, 2019.

[53] Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. 2023. URL `https://arxiv.org/abs/2307.08691`.

[54] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness, 2022.

[55] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[56] Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Zhiyong Wu, Baobao Chang, Xu Sun, Jingjing Xu, and Zhifang Sui. A survey on in-context learning. *arXiv preprint arXiv:2301.00234*, 2022.

[57] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[58] Tyler Estro, Pranav Bhandari, Avani Wildani, and Erez Zadok. Desperately seeking... optimal {Multi-Tier} cache configurations. In *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*, 2020.

[59] Alexander R Fabbri, Irene Li, Tianwei She, Suyi Li, and Dragomir R Radev. Multi-news: A large-scale multi-document summarization dataset and abstractive hierarchical model. *arXiv preprint arXiv:1906.01749*, 2019.

[60] Yuan Feng, Junlin Lv, Yukun Cao, Xike Xie, and S. Kevin Zhou. Ada-kv: Optimizing kv cache eviction by adaptive budget allocation for efficient llm inference. 2025. URL `https://arxiv.org/abs/2407.11550`.

[61] Robert Friel, Masha Belyi, and Atindriyo Sanyal. Ragbench: Explainable benchmark for retrieval-augmented generation systems. 2025. URL `https://arxiv.org/abs/2407.11005`.

[62] Bin Gao, Zhuomin He, Puru Sharma, Qingxuan Kang, Djordje Jevdjic, Junbo Deng, Xingkun Yang, Zhou Yu, and Pengfei Zuo. Cost-efficient large language model serving for multi-turn conversations with cachedattention. 2024. URL https://arxiv.org/abs/2403.19708.

[63] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Qianyu Guo, Meng Wang, and Haofen Wang. Retrieval-augmented generation for large language models: A survey, 2024.

[64] In Gim, Guojun Chen, Seung seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. Prompt cache: Modular attention reuse for low-latency inference, 2023.

[65] In Gim, Guojun Chen, Seung seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. Prompt cache: Modular attention reuse for low-latency inference, 2023.

[66] Bogdan Gliwa, Iwona Mochol, Maciej Biesek, and Aleksander Wawer. Samsum corpus: A human-annotated dialogue dataset for abstractive summarization. *arXiv preprint arXiv:1911.12237*, 2019.

[67] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks. *Communications of the ACM*, 63(11):139–144, 2020.

[68] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[69] Xanh Ho, Anh-Khoa Duong Nguyen, Saku Sugawara, and Akiko Aizawa. Constructing a multi-hop qa dataset for comprehensive evaluation of reasoning steps. *arXiv preprint arXiv:2011.01060*, 2020.

[70] Qinghao Hu, Peng Sun, and Tianwei Zhang. Understanding the workload characteristics of large language model development.

[71] HuggingFace. text-generation-inference, 2024. URL https://github.com/huggingface/text-generation-inference.

[72] Kunal Jain, Anjaly Parayil, Ankur Mallick, Esha Choukse, Xiaoting Qin, Jue Zhang, Íñigo Goiri, Rujia Wang, Chetan Bansal, Victor Rühle, et al. Intelligent router for llm workloads: Improving performance through workload-aware scheduling. *arXiv preprint arXiv:2408.13510*, 2024.

[73] Naman Jain, Tianjun Zhang, Wei-Lin Chiang, Joseph E. Gonzalez, Koushik Sen, and Ion Stoica. Llm-assisted code cleaning for training accurate code generators, 2023.

[74] Huiqiang Jiang, Qianhui Wu, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. Llmlingua: Compressing prompts for accelerated inference of large language models. *arXiv preprint arXiv:2310.05736*, 2023.

[75] Huiqiang Jiang, Qianhui Wu, Xufang Luo, Dongsheng Li, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. Longllmlingua: Accelerating and enhancing llms in long context scenarios via prompt compression, 2023.

[76] Zhengbao Jiang, Frank F Xu, Luyu Gao, Zhiqing Sun, Qian Liu, Jane Dwivedi-Yu, Yiming Yang, Jamie Callan, and Graham Neubig. Active retrieval augmented generation. *arXiv preprint arXiv:2305.06983*, 2023.

[77] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues?, 2023.

[78] jwatte. How does chatgpt store history of chat. `https://community.openai.com/t/how-does-chatgpt-store-history-of-chat/319608/2`, Aug 2023. OpenAI Community Forum.

[79] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 611–626, 2023.

[80] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

[81] Yaniv Leviathan, Matan Kalman, and Y. Matias. Fast Inference from Transformers via Speculative Decoding. In *International Conference on Machine Learning*, 2022. URL `https://api.semanticscholar.org/CorpusID:254096365`.

[82] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 33:9459–9474, 2020.

[83] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks, 2021.

[84] Dacheng Li*, Rulin Shao*, Anze Xie, Lianmin Zheng Ying Sheng, Joseph E. Gonzalez, Ion Stoica, Xuezhe Ma, and Hao Zhang. How long can open-source llms truly promise on context length?, June 2023. URL `https://lmsys.org/blog/2023-06-29-longchat`.

[85] Jiarui Li, Ye Yuan, and Zehua Zhang. Enhancing llm factual accuracy with rag to counter hallucinations: A case study on domain-specific queries in private knowledge-bases. *arXiv preprint arXiv:2403.10446*, 2024.

[86] Yuhong Li, Yingbing Huang, Bowen Yang, Bharat Venkitesh, Acyr Locatelli, Hanchen Ye, Tianle Cai, Patrick Lewis, and Deming Chen. Snapkv: Llm knows what you are looking for before generation. *arXiv preprint arXiv:2404.14469*, 2024.

[87] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. Awq: Activation-aware weight quantization for on-device llm compression and acceleration. *Proceedings of Machine Learning and Systems*, 6:87–100, 2024.

[88] Yujun Lin, Haotian Tang, Shang Yang, Zhekai Zhang, Guangxuan Xiao, Chuang Gan, and Song Han. Qserve: W4a8kv4 quantization and system co-design for efficient llm serving. *arXiv preprint arXiv:2405.04532*, 2024.

[89] Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Dengr, Chong Ruan, Damai Dai, Daya Guo, et al. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. *arXiv preprint arXiv:2405.04434*, 2024.

[90] Haotian Liu, Chunyuan Li, Qingyang Wu, and Yong Jae Lee. Visual instruction tuning. *Advances in neural information processing systems*, 36, 2024.

[91] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. Ssd: Single shot multibox detector. In *Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11–14, 2016, Proceedings, Part I 14*, pages 21–37. Springer, 2016.

[92] Yuhan Liu, Hanchen Li, Kuntai Du, Jiayi Yao, Yihua Cheng, Yuyang Huang, Shan Lu, Michael Maire, Henry Hoffmann, Ari Holtzman, et al. Cachegen: Fast context loading for language model applications. *arXiv preprint arXiv:2310.07240*, 2023.

[93] Zirui Liu, Jiayi Yuan, Hongye Jin, Shaochen Zhong, Zhaozhuo Xu, Vladimir Braverman, Beidi Chen, and Xia Hu. Kivi: A tuning-free asymmetric 2bit quantization for kv cache. *arXiv preprint arXiv:2402.02750*, 2024.

[94] Sathiya Kumaran Mani, Yajie Zhou, Kevin Hsieh, Santiago Segarra, Trevor Eberl, Eliran Azulai, Ido Frizler, Ranveer Chandra, and Srikanth Kandula. Enhancing network management using code generated by large language models. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*, HotNets '23, page 196–204, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400704154. doi:10.1145/3626111.3628183. URL `https://doi.org/10.1145/3626111.3628183`.

[95] Fabian Mentzer, Eirikur Agustsson, Michael Tschannen, Radu Timofte, and Luc Van Gool. Practical full resolution learned lossless image compression. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019.

[96] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer Sentinel Mixture Models, 2016.

[97] Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Rae Ying Yee Wong, Zhuoming Chen, Daiyaan Arfeen, Reyna Abhyankar, and Zhihao Jia. SpecInfer: Accelerating Generative LLM Serving with Speculative Inference and Token Tree Verification. *arXiv preprint arXiv:2305.09781*, 2023.

[98] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. Ray: A distributed framework for emerging {AI} applications. In *13th USENIX symposium on operating systems design and implementation (OSDI 18)*, pages 561–577, 2018.

[99] Author's Name. Llms in finance: Bloomberggpt and fingpt - what you need to know. Medium, Year of Publication. URL `https://12gunika.medium.com/llms-in-finance-bloomberggpt-and-fingpt-what-you-need-to-know-2fdf3af29217`.

[100] Claus Pahl, Antonio Brogi, Jacopo Soldani, and Pooyan Jamshidi. Cloud container technologies: a state-of-the-art review. *IEEE Transactions on Cloud Computing*, 7(3): 677–692, 2017.

[101] Joon Sung Park, Joseph C. O'Brien, Carrie J. Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. Generative agents: Interactive simulacra of human behavior, 2023.

[102] Pratyush Patel, Esha Choukse, Chaojie Zhang, Íñigo Goiri, Aashaka Shah, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative llm inference using phase splitting. *arXiv preprint arXiv:2311.18677*, 2023.

[103] Gregory F Pfister. An introduction to the infiniband architecture. *High performance mass storage and parallel I/O*, 42(617-632):10, 2001.

[104] Maciej Pondel, Iwona Chomiak-Orsa, Małgorzata Sobińska, Wojciech Grzelak, Artur Kotwica, Andrzej Małowiecki, Kamila Łuczak, Andrzej Greńczuk, Peter Busch, David Chudán, et al. Ai tools for knowledge management–knowledge base creation via llm and rag for ai assistant. In *European Conference on Artificial Intelligence*, pages 3–15. Springer, 2024.

[105] Ruoyu Qin, Zheming Li, Weiran He, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. Mooncake: Kimi's kvcache-centric architecture for llm serving. *arXiv preprint arXiv:2407.00079*, 2024.

[106] Isaac Rehg. Kv-compress: Paged kv-cache compression with variable compression rates per attention head. *arXiv preprint arXiv:2410.00161*, 2024.

[107] Carlos Riquelme, Joan Puigcerver, Basil Mustafa, Maxim Neumann, Rodolphe Jenatton, André Susano Pinto, Daniel Keysers, and Neil Houlsby. Scaling vision with sparse

mixture of experts. *Advances in Neural Information Processing Systems*, 34:8583–8595, 2021.

[108] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution image synthesis with latent diffusion models. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 10684–10695, 2022.

[109] Aurko Roy, Mohammad Saffar, Ashish Vaswani, and David Grangier. Efficient content-based sparse attention with routing transformers. *Transactions of the Association for Computational Linguistics*, 9:53–68, 2021.

[110] Ohad Rubin and Jonathan Berant. Long-range Language Modeling with Self-retrieval. *arXiv preprint arXiv:2306.13421*, 2023.

[111] Ayesha Saleem. Llm for lawyers, enrich your precedents with the use of ai. Data Science Dojo, July 2023. URL `https://datasciencedojo.com/blog/llm-for-lawyers/`.

[112] Zijing Shi, Meng Fang, Shunfeng Zheng, Shilong Deng, Ling Chen, and Yali Du. Cooperation on the fly: Exploring language agents for ad hoc teamwork in the avalon game, 2023.

[113] Woomin Song, Seunghyuk Oh, Sangwoo Mo, Jaehyung Kim, Sukmin Yun, Jung-Woo Ha, and Jinwoo Shin. Hierarchical context merging: Better long context understanding for pre-trained llms. 2024. URL `https://arxiv.org/abs/2404.10308`.

[114] Yisheng Song, Ting Wang, Puyu Cai, Subrota K. Mondal, and Jyoti Prakash Sahoo. A comprehensive survey of few-shot learning: Evolution, applications, challenges, and opportunities. *ACM Comput. Surv.*, 55(13s), jul 2023. ISSN 0360-0300. doi:10.1145/3582688. URL `https://doi.org/10.1145/3582688`.

[115] Vikranth Srivatsa, Zijian He, Reyna Abhyankar, Dongming Li, and Yiying Zhang. Preble: Efficient distributed prompt scheduling for llm serving. 2024.

[116] Pavlo Sydorenko. Top 5 applications of large language models (llms) in legal practice. Medium, 2023. URL `https://medium.com/jurdep/top-5-applications-of-large-language-models-llms-in-legal-practice-d29cde9c38ef`.

[117] Mingxing Tan, Ruoming Pang, and Quoc V Le. Efficientdet: Scalable and efficient object detection. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 10781–10790, 2020.

[118] Adam Thompson. Gpudirect storage: A direct path between storage and gpu memory | nvidia technical blog, 8 2019. URL `https://developer.nvidia.com/blog/gpudirect-storage/`. [Online; accessed 2025-02-08].

[119] Contributors to Wikimedia projects. Arithmetic coding - wikipedia, 5 2001. URL https://en.wikipedia.org/wiki/Arithmetic_coding. [Online; accessed 2025-02-06].

[120] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. LLaMA: Open and Efficient Foundation Language Models, 2023.

[121] Harsh Trivedi, Niranjan Balasubramanian, Tushar Khot, and Ashish Sabharwal. Musique: Multihop questions via single-hop question composition, 2022.

[122] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention Is All You Need, 2023.

[123] Zhongwei Wan, Ziang Wu, Che Liu, Jinfa Huang, Zhihong Zhu, Peng Jin, Longyue Wang, and Li Yuan. Look-m: Look-once optimization in kv cache for efficient multimodal long-context inference. 2024. URL https://arxiv.org/abs/2406.18139.

[124] Cunxiang Wang, Xiaoze Liu, Yuanhao Yue, Xiangru Tang, Tianhang Zhang, Cheng Jiayang, Yunzhi Yao, Wenyang Gao, Xuming Hu, Zehan Qi, et al. Survey on factuality in large language models: Knowledge, retrieval and domain-specificity. *arXiv preprint arXiv:2310.07521*, 2023.

[125] Yuxin Wang, Yuhan Chen, Zeyu Li, Xueze Kang, Zhenheng Tang, Xin He, Rui Guo, Xin Wang, Qiang Wang, Amelie Chi Zhou, and Xiaowen Chu. Burstgpt: A real-world workload dataset to optimize llm serving systems, 2024. URL https://arxiv.org/abs/2401.17644.

[126] Dekun Wu, Haochen Shi, Zhiyuan Sun, and Bang Liu. Deciphering digital detectives: Understanding llm behaviors and capabilities in multi-agent mystery games, 2023.

[127] Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks. *arXiv preprint arXiv:2309.17453*, 2023.

[128] Guangxuan Xiao, Jiaming Tang, Jingwei Zuo, Junxian Guo, Shang Yang, Haotian Tang, Yao Fu, and Song Han. Duoattention: Efficient long-context llm inference with retrieval and streaming heads. *arXiv preprint arXiv:2410.10819*, 2024.

[129] Jiayi Yao, Hanchen Li, Yuhan Liu, Siddhant Ray, Yihua Cheng, Qizheng Zhang, Kuntai Du, Shan Lu, and Junchen Jiang. Cacheblend: Fast large language model serving with cached knowledge fusion. *arXiv preprint arXiv:2405.16444*, 2024.

[130] Yuhang Yao, Han Jin, Alay Dilipbhai Shah, Shanshan Han, Zijian Hu, Yide Ran, Dimitris Stripelis, Zhaozhuo Xu, Salman Avestimehr, and Chaoyang He. Scalellm: A resource-frugal llm serving framework by optimizing end-to-end efficiency. 2024. URL https://arxiv.org/abs/2408.00008.

[131] Lu Ye, Ze Tao, Yong Huang, and Yang Li. Chunkattention: Efficient self-attention with prefix-aware kv cache and two-phase partition. *arXiv preprint arXiv:2402.15220*, 2024.

[132] Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, et al. Flashinfer: Efficient and customizable attention engine for llm inference serving. *arXiv preprint arXiv:2501.01005*, 2025.

[133] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for {Transformer-Based} generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538, 2022.

[134] Zhenrui Yue, Honglei Zhuang, Aijun Bai, Kai Hui, Rolf Jagerman, Hansi Zeng, Zhen Qin, Dong Wang, Xuanhui Wang, and Michael Bendersky. Inference scaling for long-context retrieval augmented generation. *arXiv preprint arXiv:2410.04343*, 2024.

[135] Ahmed Zerouali, Ruben Opdebeeck, and Coen De Roover. Helm charts for kubernetes applications: Evolution, outdatedness and security risks. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, pages 523–533. IEEE, 2023.

[136] Lingqi Zhang, Mohamed Wahib, and Satoshi Matsuoka. Understanding the overheads of launching cuda kernels. *ICPP19*, pages 5–8, 2019.

[137] Qingru Zhang, Chandan Singh, Liyuan Liu, Xiaodong Liu, Bin Yu, Jianfeng Gao, and Tuo Zhao. Tell your model where to attend: Post-hoc attention steering for llms. *arXiv preprint arXiv:2311.02262*, 2023.

[138] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. OPT: Open Pre-trained Transformer Language Models, 2022.

[139] Yanqi Zhang, Yuwei Hu, Runyuan Zhao, John C. S. Lui, and Haibo Chen. Unifying kv cache compression for large language models with leankv. 2024. URL `https://arxiv.org/abs/2412.03131`.

[140] Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Re, Clark Barrett, Zhangyang Wang, and Beidi Chen. H2O: Heavy-Hitter Oracle for Efficient Generative Inference of Large Language Models. In *Workshop on Efficient Systems for Foundation Models @ ICML2023*, 2023. URL `https://openreview.net/forum?id=ctPizehA9D`.

[141] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Jeff Huang, Chuyue Sun, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. Efficiently programming large language models using sglang. *arXiv preprint arXiv:2312.07104*, 2023.

[142] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving. *arXiv preprint arXiv:2401.09670*, 2024.

[143] Kan Zhu, Yilong Zhao, Liangyu Zhao, Gefei Zuo, Yile Gu, Dedong Xie, Yufei Gao, Qinyu Xu, Tian Tang, Zihao Ye, et al. Nanoflow: Towards optimal large language model serving throughput. *arXiv preprint arXiv:2408.12757*, 2024.