RESEARCH-ARTICLE

Mooncake: A KVCache-centric Disaggregated Architecture for LLM Serving

RUOYU QIN, Tsinghua University, Beijing, China

ZHEMING LI

WEIRAN HE

JIALEI CUI

HEYI TANG

FENG REN, Tsinghua University, Beijing, China

View all

Accepted: 23 October 2025
Revised: 28 July 2025
Received: 28 July 2025

Mooncake: A KVCache-centric Disaggregated Architecture for LLM Serving

RUOYU QIN*, Computer Science and Technology, Tsinghua University, Beijing, China
ZHEMING LI*, Moonshot AI, Beijing, China
WEIRAN HE, Moonshot AI, Beijing, China
JIALEI CUI, Moonshot AI, Beijing, China
HEYI TANG, Moonshot AI, Beijing, China
FENG REN, Computer Science and Technology, Tsinghua University, Beijing, China
TENG MA, Alibaba Cloud Computing, Hangzhou, China
SHANGMING CAI, Alibaba Cloud Computing, Hangzhou, China
YINENG ZHANG, Independent Researcher, Berkeley, United States
MINGXING ZHANG†, Tsinghua University, Beijing, China
YONGWEI WU, Computer Science and Technology, Tsinghua University, Beijing, China
WEIMIN ZHENG, department of computer science and technology, Tsinghua University, Beijing, China
XINRAN XU†, Moonshot AI, Beijing, China

Mooncake is the serving platform for Kimi, an LLM chatbot service developed by Moonshot AI. This platform features a KVCache-centric disaggregated architecture that not only separates prefill and decoding clusters but also efficiently utilizes the underexploited CPU, DRAM, SSD and NIC resources of the GPU cluster to establish a disaggregated KVCache. At the core of Mooncake is its KVCache-centric global cache and a scheduler designed to maximize throughput while adhering to stringent latency-related Service Level Objectives (SLOs).

Our experiments demonstrate that Mooncake excels in scenarios involving long-context inputs. In tests using real traces, Mooncake increases the effective request capacity by 59%~498% when compared to baseline methods, all while complying with SLOs. Currently, Mooncake is operational across thousands of nodes, processing over 100 billion tokens daily. In practical deployments, Mooncake's innovative architecture enables Kimi to handle 115% and 107% more requests on NVIDIA A800 and H800 clusters, respectively, compared to previous systems.

*Ruoyu Qin's part of work done as an intern at Moonshot AI, contributed equally with Zheming Li.
†Corresponding authors.

Authors' Contact Information: Ruoyu Qin, Computer Science and Technology, Tsinghua University, Beijing, Beijing, China; e-mail: qinry24@mails.tsinghua.edu.cn; Zheming Li, Moonshot AI, Beijing, Beijing, China; e-mail: lizheming@moonshot.cn; Weiran He, Moonshot AI, Beijing, Beijing, China; e-mail: heweiran@moonshot.cn; Jialei Cui, Moonshot AI, Beijing, Beijing, China; e-mail: cuijialei@moonshot.cn; Heyi Tang, Moonshot AI, Beijing, Beijing, China; e-mail: tangheyi@moonshot.cn; Feng Ren, Computer Science and Technology, Tsinghua University, Beijing, Beijing, China; e-mail: renfeng.chn@outlook.com; Teng Ma, Alibaba Cloud Computing, Hangzhou, Zhejiang, China; e-mail: sima.mt@alibaba-inc.com; Shangming Cai, Alibaba Cloud Computing, Hangzhou, Zhejiang, China; e-mail: csmthu@gmail.com; Yineng Zhang, Independent Researcher, Berkeley, California, United States; e-mail: me@zhyncs.com; Mingxing Zhang, Tsinghua University, Beijing, Beijing, China; e-mail: zhang_mingxing@mail.tsinghua.edu.cn; Yongwei Wu, Computer Science and Technology, Tsinghua University, Beijing, Beijing, China; e-mail: wuyw@tsinghua.edu.cn; Weimin Zheng, department of computer science and technology, Tsinghua University, Beijing, Beijing, China; e-mail: zwm-dcs@tsinghua.edu.cn; Xinran Xu, Moonshot AI, Beijing, Beijing, China; e-mail: xuxinran@moonshot.cn.

ACM 1553-3093/2025/11-ART
https://doi.org/10.1145/3773772

CCS Concepts: • **Computer systems organization** → *Distributed architectures*; • **Computing methodologies** → *Artificial intelligence.*

Additional Key Words and Phrases: Machine learning system, LLM serving, KVCache

1 Introduction

With the rapid adoption of large language models (LLMs) in various scenarios [6, 35, 37, 46], the workloads for LLM serving have become significantly diversified. These workloads differ in input/output length, distribution of arrival, and, most importantly, demand different kinds of Service Level Objectives (SLOs). As a Model as a Service (MaaS) provider, one of the primary goals of Kimi [32] is to solve an optimization problem with multiple complex constraints. The optimization goal is to maximize overall effective throughput, which directly impacts revenue, while the constraints reflect varying levels of SLOs. These SLOs typically involve meeting latency-related requirements, mainly the time to first token (TTFT) and the time between tokens (TBT).

To achieve this goal, a prerequisite is to make the best use of the various kinds of resources available in the GPU cluster. Specifically, although GPU servers are currently provided as highly integrated nodes (e.g., DGX/HGX supercomputers [34]), it is necessary to decouple and restructure them into several disaggregated resource pools, each optimized for different but collaborative goals. For example, many other researchers [19, 39, 52] have suggested separating prefill servers from decoding servers, because these two stages of LLM serving have very different computational characteristics.

Further advancing this disaggregation strategy, we have engineered a disaggregated KVCache by pooling CPU, DRAM, SSD and RDMA resources of the GPU cluster, referred to as Mooncake Store. This novel architecture harnesses underutilized resources to enable efficient near-GPU prefix caching, significantly enhancing the global cache capacity and inter-node transfer bandwidth. The resultant distributed KVCache system embodies the principle of **trading more storage for less computation**. Thus, as demonstrated in Figure 1, it substantially boosts Kimi's maximum throughput capacity in meeting the required SLOs for many important real-world scenarios. Later in this paper, we will first delve into a mathematical analysis of this strategy's benefits for LLM serving and empirically assess its efficacy using real-world data (§2.2). Then, we will detail the design choices made in implementing this petabyte-level disaggregated cache, which is interconnected via an RDMA network up to 8×400 Gbps (§3.2).

Building on this idea, we also found that the scheduling of KVCache is central to LLM serving, and hence propose a corresponding disaggregated architecture. Figure 2 presents our current **KVCache-centric disaggregated architecture** for LLM serving, named Mooncake. For each request, the global scheduler (Conductor) will select a pair of prefill and decoding instances and schedule the request in the following steps: *1)* transfer as much reusable KVCache as possible to the selected prefill instance; *2)* complete the prefill stage in chunks/layers and continuously stream the output KVCache to the corresponding decoding instance; *3)* load the KVCache and add the request to the continuous batching process at the decoding instance for generating request outputs.

Although this process seems straightforward, the selection policy is complex due to many restrictions. In the prefill stage, the main objective is to reuse the KVCache as much as possible to avoid redundant computation. However, the distributed KVCache pool faces challenges in terms of both capacity and access latency. Thus Conductor is responsible for scheduling requests with KVCache-awareness and executing scheduling operations such as swapping and replication accordingly. The hottest blocks should be replicated to multiple nodes to avoid fetching congestion, while the coldest ones should be swapped out to reduce reserving costs. In contrast, the decoding stage has different optimization goals and constraints. The aim is to aggregate as many tokens as possible in a decoding batch to improve the Model FLOPs Utilization (MFU). However, this objective is restricted not only by the TBT SLO but also by the total size of the aggregated KVCache that can be contained in the VRAM.

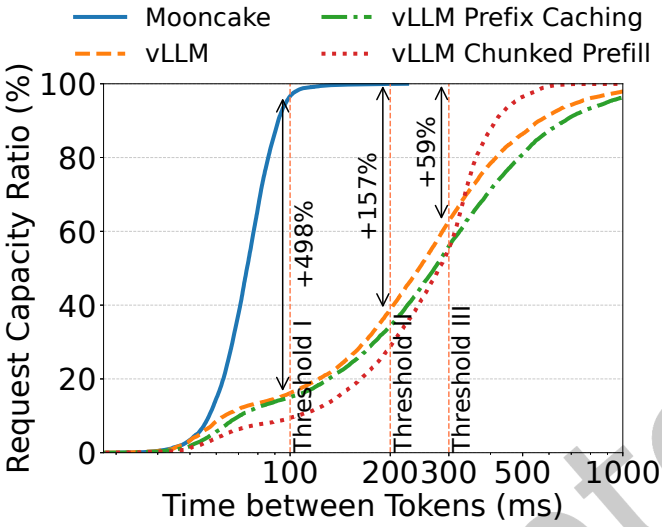


Fig. 1. The experiment of the effective request capacity of Mooncake under the real-world conversation workload and different TBT SLOs. In this experiment, Mooncake and three baseline systems utilize 16 8×A800 nodes each. More on §7.2.

In §4, we will detail our KVCache-centric request scheduling algorithm, which balances instance loads and user experience as measured by TTFT and TBT SLOs. This includes a heuristic-based automated hotspot migration scheme that replicates hot KVCache blocks without requiring precise predictions of future KVCache usage. Additionally, we introduce new scheduling optimizations—Early Rejection (§4.3.2) and Early Rejection based on Prediction (§4.3.4)—to address the issue of underutilized resources in disaggregated serving systems during overload scenarios. Experimental results show that our enhanced KVCache-centric scheduling significantly reduces TTFT and increases service capacity in real-world settings.

We will also describe the main design choices made during its implementation, especially those not covered in current research. For example, regarding P/D disaggregation, there are currently debates on its feasibility in large-scale practice due to bandwidth requirements and trade-offs associated with chunked prefill (e.g., Sarathi-Serve [1]). We demonstrate, through comparison with vLLM, that with a highly optimized transfer engine, the communication challenges can be managed, and P/D disaggregation is preferable for scenarios with stringent SLO limits (§7.2). Based on P/D disaggregation, we propose a dynamic P/D switch algorithm (§5.1) that automatically balances workloads by adapting to dynamic conditions. Additionally, we discuss how to implement a separate prefill node pool that seamlessly handles the dynamic distribution of context length. We employ a chunked pipeline parallelism (CPP) mechanism to scale the processing of a single request across multiple nodes, which is necessary for reducing the TTFT of long-context inputs. Compared to traditional sequence parallelism (SP) based solutions, CPP reduces network consumption and simplifies the reliance on frequent elastic scaling (§3.3). Furthermore, our workload-aware cache strategy (§5.2) effectively addresses declining cache hit rates caused by evolving upper-layer systems. We also detail improvements in fault tolerance mechanisms (§5.3), significantly enhancing system reliability.

Mooncake is currently the serving platform of Kimi and has successfully handled exponential workload growth (more than 100 billion tokens a day). According to our historical statistics, the innovative architecture of

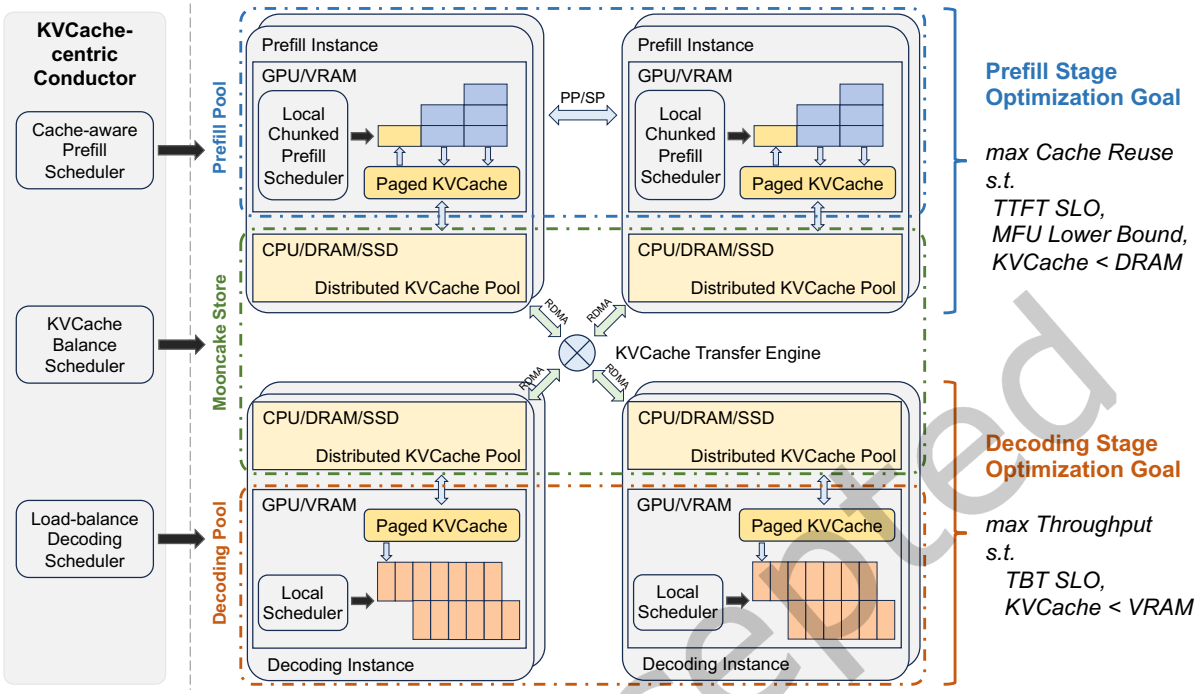


Fig. 2. Mooncake architecture.

Mooncake enables Kimi to handle 115% and 107% more requests on the A800 and H800 clusters, respectively, compared to previous systems. Besides, Mooncake has become a critical component in open-source LLM inference communities, integrated into prominent frameworks such as SGLang [51] and vLLM [23]. In the open-source Mooncake section (§6), we will detail the architecture and functionalities of the open-source modules, Mooncake Store and Transfer Engine (§6.1). We will illustrate Mooncake's practical utility by presenting real-world integration cases with frameworks such as SGLang (§6.2) and vLLM (§6.3).

To ensure the reproducibility of our results while safeguarding proprietary information, we also provide detailed experimental outcomes using a dummy model mirroring the architecture of LLaMA3-70B, based on replayed traces of actual workloads. These traces, along with the distributed KVCache storage and transfer infrastructure of Mooncake, are open-sourced at https://github.com/kvcache-ai/Mooncake.

In end-to-end experiments using public datasets and real workloads, Mooncake excels in long-context scenarios. Compared to the baseline method, Mooncake can achieve up to a 498% increase in the effective request capacity while meeting SLOs. In §7.3, we compare Mooncake Store with the local cache design and find that the global cache design of Mooncake Store significantly improves the cache hit rate. In our experiments, the cache hit rate is up to 2.36× higher than that of the local cache, resulting in up to 48% savings in prefill computation time. Mooncake, to the best of our knowledge, is the first system to demonstrate the significant benefits of using a distributed KVCache pool to share KVCache across different chat sessions and queries in large-scale deployment scenarios. We also evaluate the performance of the transfer engine that supports high-speed RDMA transfers in Mooncake, which shows it is approximately 2.4× and 4.6× faster than existing solutions (§7.4).

Table 1. Notations and parameters. Model and machine parameters are set according to LLaMA3-70B and 8×A800.

Notation	Description	Value
l	Num layers	80
d	Model dimension	8192
a, b	Constant coefficients in Equation 1	4, 22
gqa	Num q heads / Num kv heads	8
s	Tensor element size	2 B (BFloat16)
G	GPU computation throughput	8×312 TFLOPS
B_{h2d}	Host-to-device bandwidth	128 GB/s
B_{nic}	NIC bandwidth	800 Gbps
n, p	Prompt and matched prefix length, respectively	

2 Preliminary and Problem Definition

2.1 Service Level Objectives of LLM Serving

Modern large language models (LLMs) are based on the Transformer architecture, which utilizes attention mechanisms and multilayer perceptrons (MLP) to process input. Popular Transformer-based models, such as GPT [40] and LLaMA [45], employ a decoder-only structure. Each inference request is logically divided into two stages: the prefill stage and the decoding stage.

During the prefill stage, all input tokens are processed in parallel, and hence it is typically computationally intensive. This stage generates the first output token while storing intermediate results of computed keys and values, referred to as the KVCache. The decoding stage then uses this KVCache to autoregressively generate new tokens. It processes only one token at a time per batch due to the limitation of autoregressive generation, which makes it memory-constrained and causes computation time to increase sublinearly with batch size. Thus, a widely used optimization in the decoding stage is continuous batching [23, 49]. Before each iteration, the scheduler checks the status and adds newly arrived requests to the batch while removing completed requests.

Due to the distinct characteristics of the prefill and decoding stages, MaaS providers set different metrics to measure their corresponding Service Level Objectives (SLOs). Specifically, the prefill stage is mainly concerned with the latency between the request arrival and the generation of the first token, known as the time to first token (TTFT). On the other hand, the decoding stage focuses on the latency between successive token generations for the same request, referred to as the time between tokens (TBT).

In real deployments, if the monitor detects unmet SLOs, we need to either add inference resources or reject some incoming requests. However, due to the current contingent supply of GPUs, elastically scaling out the inference cluster is typically unfeasible. Therefore, we proactively reject requests that are predicted not to meet the SLOs to alleviate the cluster's load. Our main objective is to maximize overall throughput while adhering to SLOs, a concept referred to as goodput in other research [47, 52].

2.2 More Storage for Less Computation

To meet the stringent SLOs described above, a commonly adopted solution is to cache previously generated KVCache and reuse it upon finding a prefix match. However, existing approaches [15, 44, 51] typically restrict caching to local HBM and DRAM, assuming that the transfer bandwidth required for global scheduling would be prohibitively high. But, as we will described later in §7.3, the capacity of local DRAM supports only up to 50% of the theoretical cache hit rate, making the design of a global cache essential. In this section, we present a mathematical analysis of the actual bandwidth necessary to benefit from this strategy, explaining why distributed

caching is advantageous, especially for larger models like LLaMA3-70B. More experimental results will be given later in §7.4.2.

We base our analysis on the model using notations described in Table 1 and incorporate specific parameters of LLaMA3-70B. Essentially, current popular LLMs are autoregressive language models where each token's KVCache depends only on itself and preceding tokens. Therefore, KVCache corresponding to the same input prefix can be reused without affecting output accuracy. If a current request's prompt of length n shares a common prefix of length p with previously cached KVCache, its prefill process can be optimized as follows:

$$
\begin{aligned}
&q[p:n], k[p:n], v[p:n] = \text{MLP}(hidden[p:n]) \\
&k[1:n], v[1:n] \leftarrow \text{KVCache} + (k[p:n], v[p:n]) \\
&o[p:n] = \text{Attention}(q[p:n], k[1:n], v[1:n]) \\
&\text{KVCache} \leftarrow (k[1:n], v[1:n])
\end{aligned}
$$

Given input length n, the FLOPS of the prefill stage can be calculated as:

$$
flops(n) = l \times (an^2d + bnd^2) \tag{1}
$$

Thus, reusing KVCache approximately reduces the computation cost of prefill by $l \times (ap^2d + bpd^2)$. However, this requires transferring the cached KVCache into the prefill GPU's HBM, with a size of $p \times l \times (2 \times d/gqa) \times s$. Assuming the average computation throughput is G and the average KVCache loading speed is B (where B is determined by the minimum of B_{h2d} and B_{nic}), the reuse of KVCache is beneficial in terms of TTFT if:

$$
\frac{B}{G} > \frac{2ds}{gqa \times (apd + bd^2)} \tag{2}
$$

In such scenarios, reusing the KVCache not only reduces GPU time and costs but also enhances the user experience by improving TTFT. The criteria for bandwidth B relative to computation throughput G are more readily met with larger values of d, which is proportional to the model size. For example, when running LLaMA3-70B on a machine with 8×A800 GPUs and assuming a prefix length of 8192, Equation 2 yields a minimum required B of 6 GB/s. The requirement for B will be enlarged to 19 GB/s for an 8×H800 machine. Moreover, in practical scenarios, because the transfer stages cannot be perfectly overlapped with each other, the actual bandwidth requirement is even higher. However, as we will demonstrate in §7.4.2, a fully utilized 100 Gbps NIC per NVIDIA A800 HGX network is sufficient to meet these criteria.

3 Design of Mooncake

3.1 Overview

As depicted in Figure 2, Mooncake employs a disaggregated architecture that not only separates prefill from decoding nodes but also groups the CPU, DRAM, SSD, and RDMA resources of the GPU cluster to implement a disaggregated KVCache. To schedule all these disaggregated components, at its center, Mooncake implements a global scheduler named Conductor. Conductor is responsible for dispatching requests based on the current distribution of the KVCache and workload characteristics. Mooncake Store, detailed in §3.2, manages the storage and transfer of these KVCache blocks.

Specifically, Figure 3 demonstrates the typical workflow of a request. Once tokenizing is finished, the conductor selects a pair of prefill nodes and a decoding node, and starts a workflow comprising four steps:
1) KVCache Reuse: The selected prefill node (group) receives a request that includes the raw input, the block keys of the prefix cache that can be reused, and the block keys of the full cache allocated to the request. It loads the prefix cache from remote CPU memory into GPU memory based on the prefix cache block keys to bootstrap the request. This step is skipped if no prefix cache exists. This selection balances three objectives: reusing as much

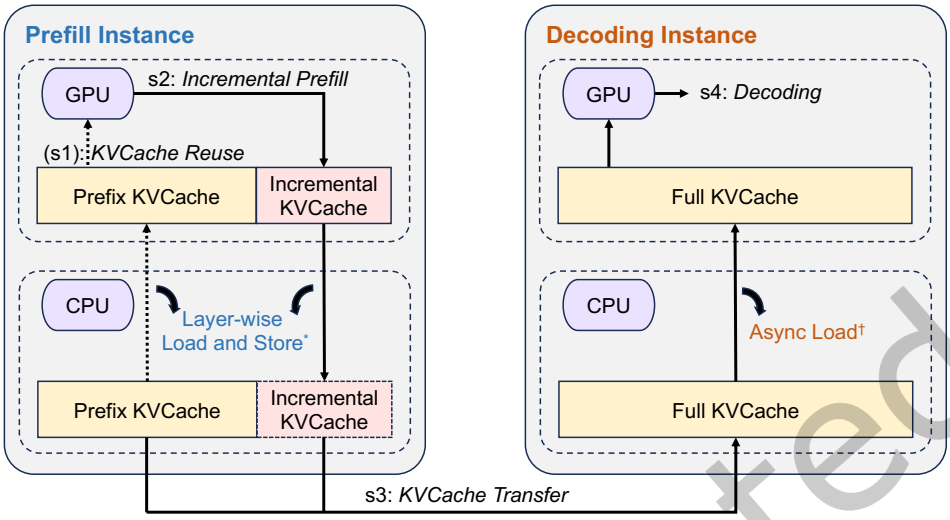


Fig. 3. Workflow of inference instances.

KVCache as possible, balancing the workloads of different prefill nodes, and guaranteeing the TTFT SLO. It leads to a KVCache-centric scheduling that will be further discussed in §4.

2) Incremental Prefill: The prefill node completes the prefill stage using prefix cache and stores the newly generated incremental KVCache back into CPU memory. If the number of uncached input tokens exceeds a certain threshold, the prefill stage is split into multiple chunks and executed in a pipeline manner. This threshold is selected to fully utilize the corresponding GPU's computational power and is typically larger than 1000 tokens. The reason for using chunked but still disaggregated prefill nodes is explained in §3.3.

3) KVCache Transfer: Mooncake Store is deployed in each node to manage and transfer these caches. This step is asynchronously executed and overlapped with the above incremental prefill step, streaming the KVCache generated by each model layer to the destination decoding node's CPU memory to reduce waiting time.

4) Decoding: After all the KVCache is received in the CPU memory of the decoding node, the request joins the next batch in a continuous batching manner. The decoding node is pre-selected by Conductor based on its current load to ensure it does not violate the TBT SLO.

3.2 Mooncake Store: Cache of KVCache

Central to Mooncake is its efficient implementation of a distributed global cache of KVCache, referred to as Mooncake Store. As described in §2.2, reusing cached KVCache not only cuts computation costs but also improves user experience by reducing the TTFT, particularly when the aggregated bandwidth is fully utilized. However, achieving full utilization is challenging because the bandwidth can reach up to 8×400 Gbps, comparable to DRAM bandwidth.

We first introduce how Mooncake Store manages KVCache in §3.2.1, including its storage scheme and eviction policy. In §3.2.2, we describe the object-based APIs and memory transfer APIs of Mooncake Store. In §3.2.3, we will detail the design of Mooncake Store's transfer engine, a high-performance, zero-copy KVCache transfer system designed to maximize the benefits of using multiple RDMA NICs per machine. It enhances execution efficiency and reliability through techniques such as topology-aware path selection and endpoint pooling.

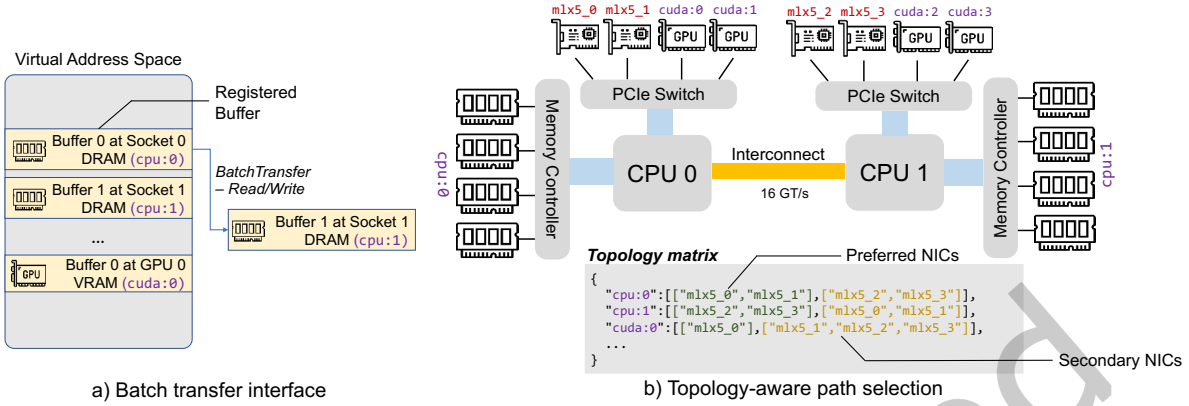


Fig. 4. Transfer engine of Mooncake Store.

3.2.1 KVCache Management. In Mooncake Store, all KVCache is stored as paged blocks within a distributed cache pool. The block size, i.e., the number of tokens contained in each block, is determined by the model size and the optimal network transmission size, typically ranging from 16 to 512 tokens. Each block is attached with a hash key determined by both its own hash and its prefix for deduplication. The same hash key may have multiple replicas across different nodes to mitigate hot-cache access latency, controlled by our cache-load-balancing policy described in §4.2.

Mooncake Store allocates space for each cache block in the cache pool and logs metadata such as the block key and its address. When the cache pool is full, Mooncake Store employs an LRU (Least Recently Used) strategy to evict an existing cache block—unless the block is currently being accessed by an ongoing request—and overwrites the evicted block's space with the new block.

3.2.2 Interface. At the higher layer, Mooncake Store offers object-based APIs such as `put`, `get`, and `change_replica`. These facilitate the caching of KVCache in a disaggregated manner, organizing mini blocks of KVCache as memory objects and enabling Conductor to adjust the number of replicas for each KVCache block to achieve higher bandwidth aggregation. These functions are supported by a set of synchronous batch transfer APIs, detailed in Listings 1.

Transfer operations are available for both DRAM and GPU VRAM and will utilize GPU Direct RDMA when optimal, provided that the specified memory region has been pre-registered. The completion of these operations can be monitored asynchronously via the `getTransferStatus` API, which reports whether transfers are ongoing or have encountered errors.

Listing 1. Memory transfer APIs in Mooncake Store.

```
int registerLocalMemory(void *vaddr, size_t len, const string &type);
BatchID allocateBatchID(size_t batch_size);
int submitTransfer(BatchID batch_id, const vector<Request> &entries);
int getTransferStatus(BatchID batch_id, int request_index, Status &status);
int freeBatchID(BatchID batch_id);
```

3.2.3 Transfer Engine. To efficiently implement the above APIs, a transfer engine is designed to achieve several key objectives: 1) Effectively distribute transfer tasks across multiple RDMA NIC devices; 2) Abstract the complexities of RDMA connection management from the APIs; and 3) Appropriately handle temporary network failures. This transfer engine has been meticulously engineered to fulfill each of these goals.

Network Setup. The benefits of Mooncake rely on a high-bandwidth network interconnect. Currently, we use standard HGX machines where each A800 GPU is paired with a 100/200 Gbps NIC, and each H800 GPU is paired with a 200/400 Gbps NIC, which is comparable to memory bandwidth and existing libraries (other than NCCL) fail to fully utilize this capacity. As for NCCL [10], it cannot gracefully handle dynamic topology changes due to the addition or removal of nodes/NICs and does not support DRAM-to-DRAM paths. In contrast, the transfer engine endeavors to find alternative paths upon failure.

To address congestion, the network utilizes RoCEv2 tuned by cloud providers. In the scheduler, we mitigate congestion by increasing the number of replicas for hot KVCaches (§4.2).

Topology-aware path selection. Modern inference servers often consist of multiple CPU sockets, DRAM, GPUs, and RDMA NIC devices. Although it's technically possible to transfer data from local DRAM or VRAM to a remote location using any RDMA NIC, these transfers can be limited by the bandwidth constraints of the Ultra Path Interconnect (UPI) or PCIe Switch. To overcome these limitations, Mooncake Store implements a topology-aware path selection algorithm.

Before processing requests, each server generates a *topology matrix* and broadcasts it across the cluster. This matrix categorizes network interface cards (NICs) into "preferred" and "secondary" lists for various *types* of memory, which types are specified during memory registration. Under normal conditions, a NIC from the preferred list is selected for transfers, facilitating RDMA operations within the local NUMA or GPU Direct RDMA through the local PCIe switch only. In case of failures, NICs from both lists may be utilized. The process involves identifying the appropriate local and target NICs based on the memory addresses, establishing a connection, and executing the data transfer.

For instance, as illustrated in Figure 4, to transfer data from buffer 0 (assigned to `cpu:0`) in the local node to buffer 1 (assigned to `cpu:1`) in the target node, the engine first identifies the preferred NICs for `cpu:0` using the local server's topology matrix and selects one, such as `mlx5_1`, as the local NIC. Similarly, the target NIC, such as `mlx5_3`, is selected based on the target memory address. This setup enables establishing an RDMA connection from `mlx5_1@local` to `mlx5_3@target` to carry out RDMA read and write operations.

To further maximize bandwidth utilization, a single request's transfer is internally divided into multiple slices at a granularity of 16 KB. Each slice might use a different path, enabling collaborative work among all RDMA NICs.

Endpoint management. Mooncake Store employs a pair of *endpoints* to represent the connection between a local RDMA NIC and a remote RDMA NIC. In practice, each endpoint includes one or more RDMA queue pair objects. Connections in Mooncake Store are established in an on-demand manner; endpoints remain unpaired until the first request is made.

To prevent a large number of endpoints from slowing down request processing, Mooncake Store employs endpoint pooling, which caps the maximum number of active connections. We use the SIEVE [50] algorithm to manage endpoint eviction. If a connection fails due to link errors, it is removed from the endpoint pools on both sides and re-established during the next data transfer attempt.

Failure handing. In a multi-NIC environment, one common failure scenario is the temporary unavailability of a specific NIC, while other routes may still connect two nodes. Mooncake Store is designed to adeptly manage such temporary failures effectively. If a connection is identified as unavailable, Mooncake Store automatically identifies an alternative, reachable path and resubmits the request to a different RDMA NIC device. Furthermore,

Mooncake Store is capable of detecting problems with other RDMA resources, including RDMA contexts and completion queues. It temporarily avoids using these resources until the issue, such as a downed link, is resolved.

3.3 Mooncake's Prefill Pool

Unlike the inviolable decoding nodes, the necessity and best practices for designing a separate and elastic prefill pool remain under debate. For example, although many researchers [19, 39, 52] share our intuition to use a disaggregated architecture, it is worth discussing whether this separation is still necessary with the introduction of chunked prefill [1].

However, after careful consideration, we decided to maintain Mooncake's disaggregated architecture. This decision is primarily driven by the fact that online services typically have more stringent SLOs. While chunked prefill reduces decoding interference, it remains challenging to simultaneously maximize MFU during the prefill stage and meet the TBT SLO during the decoding stage. We will demonstrate this in the end-to-end experiments in §7.2. Another important reason is that we think prefill nodes require different cross-node parallelism settings to handle long contexts as the available context length of recent LLMs is increasing rapidly, from 8k to 128k and even up to 1 million tokens [16]. Typically, for such long context requests, the input tokens can be 10 to 100 times larger than the output tokens, making optimizing the TTFT crucial. Due to the abundant parallelism in long context prefill, using more than a single 8×GPU node to process them in parallel is desirable. However, extending tensor parallelism (TP) across more than one node requires two expensive RDMA-based all-reduce operations per layer, significantly reducing the MFU of prefill nodes.

Recently, many works have proposed sequence parallelism (SP) [5, 13, 20, 21, 24, 26, 29]. SP partitions the input sequences of requests across different nodes to achieve acceleration, allowing even long requests to meet the TTFT SLO. However, when applied to shorter input requests, SP results in a lower MFU compared to using single-node TP only. Recent research [47] proposes elastic sequence parallelism to dynamically scale up or down the SP group. Although possible, this adds complexity to our architecture. Additionally, SP still requires frequent cross-node communication, which lowers the MFU and competes with network resources for transferring KVCache across nodes.

To address this, Mooncake leverages the autoregressive property of decoder-only transformers and implements chunked pipeline parallelism (CPP) for long context prefill. We group every X nodes in the prefill cluster into a pipelined prefill node group. For each request, its input tokens are partitioned into chunks, each no longer than the $prefill\_chunk$. Different chunks of the same request can be processed simultaneously by different nodes, thus parallelizing the processing and reducing TTFT.

CPP offers two main benefits: *1)* Similar to pipeline parallelism in training, it requires cross-node communication only at the boundaries of each pipeline stage, which can be easily overlapped with computation. This leads to better MFU and less network resource contention with KVCache transfer. *2)* It naturally fits both short and long contexts, bringing no significant overhead for short context prefill and avoiding frequent dynamic adjustment of node partitioning. This pipeline-based acceleration method has been explored in training systems [27], but to our knowledge, this is the first application in the inference stage, as long context inference has only recently emerged.

4 Scheduling

4.1 Prefill Global Scheduling

Previous research on LLM serving typically uses a load-balancing strategy that evaluates the load on each instance based on the number of assigned requests. In Mooncake, however, the selection of prefill instances considers additional factors—not just load but also the prefix cache hit length and the distribution of reusable KVCache blocks. While there is a preference to route requests to prefill instances with longer prefix cache lengths to reduce

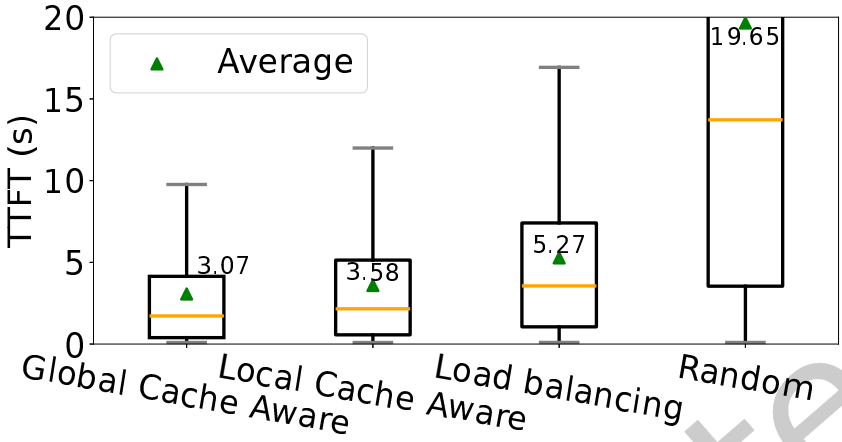


Fig. 5. The prefill scheduling experiment that shows KVCache-Centric scheduling significantly reduces average TTFT.

computation costs, it may be beneficial to schedule them to other nodes to ensure overall system balance and meet TTFT SLOs. To address these complexities, we propose a cache-aware global scheduling algorithm that accounts for both the prefill time due to the prefix cache and the local queuing time.

Algorithm 1 details the mechanism for our KVCache-centric prefill scheduling. For every new request, block keys are then compared one by one against each prefill instance's cache keys to identify the prefix match length ($prefix\_len$) With this matching information, Conductor estimates the corresponding execution time based on the request length and $prefix\_len$ (which varies by instance), using a polynomial regression model fitted with offline data. It then adds the estimated waiting time for that request to get the TTFT on that instance. Finally, Conductor assigns the request to the instance with the shortest TTFT and updates the cache and queue times for that instance accordingly. If the SLO is not achievable, Conductor directly returns the HTTP 429 Too Many Requests response status code to the upper layers.

The backbone of this scheduling framework is straightforward, but complexities are hidden in the engineering implementation of various components. For example, to predict the computation time of the prefill stage for a request, we employ a predictive model derived from offline test data. This model estimates the prefill duration based on the request's length and prefix cache hit length. Thanks to the regular computation pattern of Transformers, the error bound of this prediction is small as long as enough offline data is available. The queuing time for a request is calculated by aggregating the prefill times of all queued requests. In practical implementations, TTFTs are computed in parallel, rendering the processing time negligible compared to the inference time.

More difficulty lies in predicting the transfer time because it is determined not only by the size of the transferred data but also by the current network status, especially whether the sending node is under congestion. This also necessitates the replication of hot KVCache blocks, which will be discussed in §4.2.

4.2 Cache Load Balancing

In Mooncake, each prefill instance has its own set of local prefix caches. The usage frequency of these caches varies significantly. For example, system prompts are accessed by almost every request, whereas caches storing content from a local long document may be used by only one user. As discussed in §4.1, Conductor's role is crucial in achieving an optimal balance between cache matching and instance load. Thus, from the perspective of

Algorithm 1 KVCache-centric Scheduling Algorithm

Input: prefill instance pool P, decoding instance pool D, request R, cache block size B.
Output: the prefill and decoding instances (p, d) to process R.

```
block_keys ← PrefixHash(R.prompt_tokens, B)
TTFT, p ← inf, ∅
best_len, best_instance ← FindBestPrefixMatch(P, block_keys)
for instance ∈ P do
    if best_len / instance.prefix_len > kvcache_balancing_threshold then
        prefix_len ← best_len
        transfer_len ← best_len − instance.prefix_len
        T_transfer ← EstimateKVCacheTransferTime(transfer_len)
    else
        prefix_len ← instance.prefix_len
        T_transfer ← 0
    T_queue ← EstimatePrefillQueueTime(instance)
    T_prefill ← EstimatePrefillExecutionTime(
                    len(R.prompt_tokens), prefix_len)
    if TTFT > T_transfer + T_queue + T_prefill then
        TTFT ← T_transfer + T_queue + T_prefill
        p ← instance
d, TBT ← SelectDecodingInstance(D)
if TTFT > TTFT_SLO or TBT > TBT_SLO then
    reject R; return
if best_len / p.prefix_len > kvcache_balancing_threshold then
    TransferKVCache(best_instance, p)
return (p, d)
```

the distributed cache system, load balancing also plays an important role. Specifically, it involves strategizing on how to back up caches to ensure that global prefill scheduling can achieve both high cache hits and low load.

A straw-man solution to this KVCache scheduling problem could be collecting the global usages of each block, using a prediction model to forecast their future usages, and making scheduling decisions accordingly. However, unlike the estimation of prefill time, workloads are highly dynamic and change significantly over time. Especially for a MaaS provider experiencing rapid growth in its user base, it is impossible to accurately predict future usages. Thus, we propose a heuristic-based automated hotspot migration scheme to enhance cache load balancing.

As previously noted, requests may not always be directed to the prefill instance with the longest prefix cache length due to high instance load. In such cases, Conductor forwards the cache's location and the request to an alternative instance if the estimated additional prefill time is shorter than the transfer time. This instance proactively retrieves the KVCache from the holder and stores it locally. More importantly, we prefer to compute the input tokens if the best remote prefix match length is no larger than the current local reusable prefix multiplied by a threshold[1]. Both strategies not only reduce the prefill time for requests but also facilitate the automatic replication of hotspot caches, allowing for their broader distribution across multiple instances.

To validate the effectiveness of our strategy, we conduct a scheduling experiment that compares random scheduling and load-balancing scheduling with our strategy. We further compare the local cache-aware scheduling described in §4.1 and the global cache-aware scheduling described in this section that considers cache load

[1]This threshold is currently adjusted manually but can be adaptively adjusted by an algorithm in the future.

balancing. In random scheduling, a prefill instance is selected arbitrarily for each request. In load-balancing scheduling, the instance with the lightest load is chosen. Specifically, we build a Mooncake cluster consisting of 16 8×A800 nodes, and replay the conversation trace detailed in §7.2.1 for the experiment. We assess the performance of each scheduling algorithm based on the TTFTs. The experimental results, depicted in Figure 5, demonstrate that our KVCache-centric scheduling algorithms outperform random and load-balancing scheduling. By incorporating cache load balancing, the global cache-aware algorithm reduces the average TTFT by an additional 14% compared to the local cache-aware algorithm.

4.3 Overload-oriented Scheduling

Most existing work on LLM serving assumes that all requests will be processed, optimizing the throughput or the TTFT and TBT of requests accordingly. However, in real scenarios, processing every incoming request is neither economical nor realistic. For commercial inference services facing rapidly increasing volumes of user requests, the growth rate of the cluster's inference resources is far slower than the increase in incoming requests. As a result, overload is a common issue in current LLM serving, especially during peak times.

To balance costs and user experience, the system should process as many requests as possible until the system load reaches a predefined threshold. After this point, remaining requests will be either directly rejected or deferred for later retry. Mooncake, implemented as a disaggregated inference system, allows for more flexible scheduling strategies but also confronts unique scheduling challenges not present in non-disaggregated systems and not mentioned in previous works[19, 39, 52].

In this section, we describe an early rejection policy designed specifically for a disaggregated architecture and address the load fluctuation caused by this approach. We then explore how predicting the generation length is necessary to mitigate these problems.

4.3.1 Scheduling in Overload Scenarios. In scenarios where system overload occurs, scheduling involves determining whether to accept or reject incoming requests based on the system load. A critical aspect of this process is defining what constitutes the "system load", as this definition influences the threshold at which requests are rejected. In conventional coupled systems, the prediction of TTFT and TBT can be complicated by interference between the prefill and decoding stages. Therefore, the load is often measured simply by the ratio of the number of requests being processed to the system's maximum capacity.

In contrast, Mooncake, with its disaggregated architecture, processes the prefill and decoding stages independently. Thus, we use SLO satisfaction as a direct load measurement. Specifically, we define l_{ttft} and l_{tbt} as the TTFT and TBT SLO constraints for requests, respectively. The load for prefill and decoding instances is then determined by comparing the predicted maximum TTFT and TBT on an instance against l_{ttft} and l_{tbt}. With these two criteria, Mooncake's scheduling requires two key decisions: first, whether to accept the prefill stage based on the prefill instance's load, and second, whether to proceed with the decoding stage depending on the decoding instance's load.

4.3.2 Early Rejection. In practice, the individual load on prefill or decoding instances does not accurately reflect the actual number of requests processed by the system. This discrepancy arises due to a time lag between scheduling prefill and decoding instances for a single request. If a request is rejected by the decoding instance due to high load after the prefill stage has been completed, the computational resources expended during the prefill stage are wasted. Consequently, the actual number of successfully processed requests during prefill is less than that indicated by the load metric.

To address this issue, it is natural to advance the load assessment of the decoding instance to precede the beginning of the prefill stage. We refer to this strategy as **Early Rejection**. Upon the arrival of a request, Conductor evaluates whether to accept the request based on the greater load between the prefill and decoding

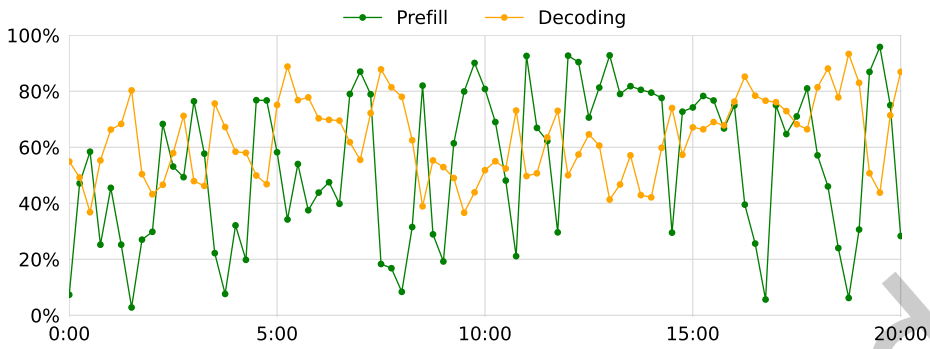


Fig. 6. The load of prefill and decoding instances over 20 minutes, before using the prediction-based early rejection.

pools. Early Rejection significantly reduces ineffective computations from rejected requests and enhances load balancing.

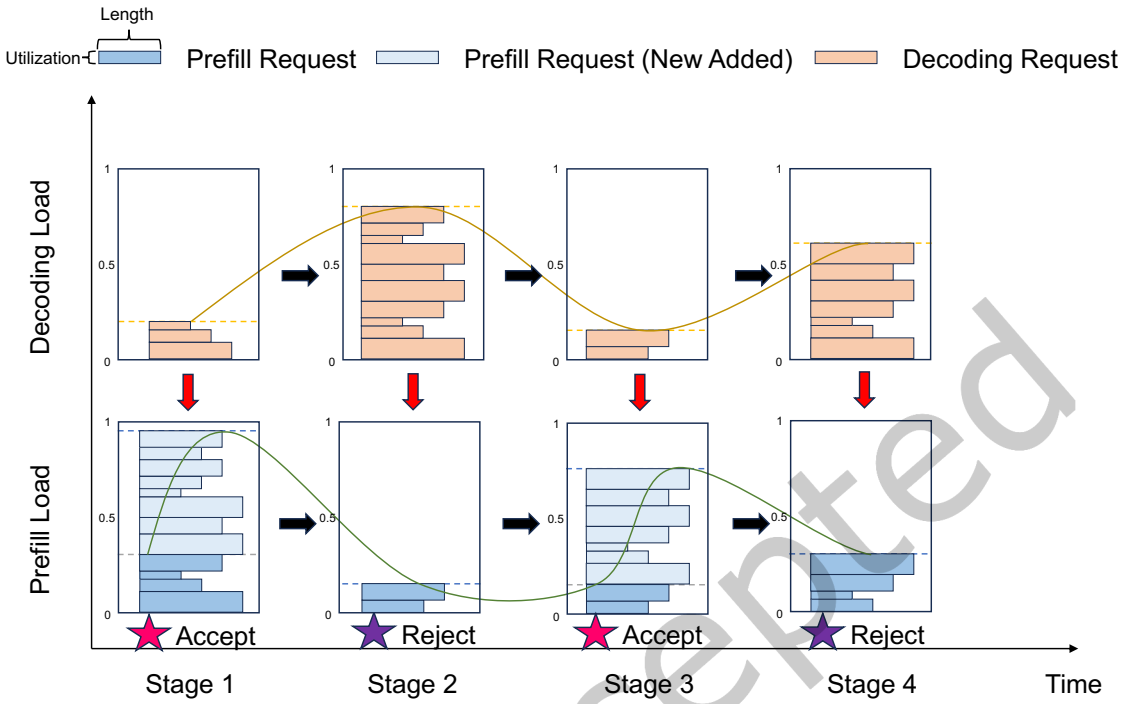
4.3.3 Load Fluctuation Caused by Early Rejection. However, Early Rejection introduces new challenges. Figure 6 shows the observed real-world instance load over a 20-minute period in a cluster of 20 machines after using the Early Rejection strategy. It highlights significant anti-phase fluctuations between prefill and decoding machines. This phenomenon becomes more pronounced in clusters with fewer prefill machines and in scenarios where the prefill stage takes longer.

Upon further exploration, we found that this load fluctuation problem is rooted in the time lag between predicting the decoding load and its actual execution. Scheduling based on the current decoding load is inherently delayed. This delay causes fluctuations and phase staggering between the loads on prefill and decoding instances, as illustrated in the theoretical example described in Figure 7a. The green curve represents the load of prefill instances (scaled from 0 to 1), and the yellow curve represents the load of decoding instances.

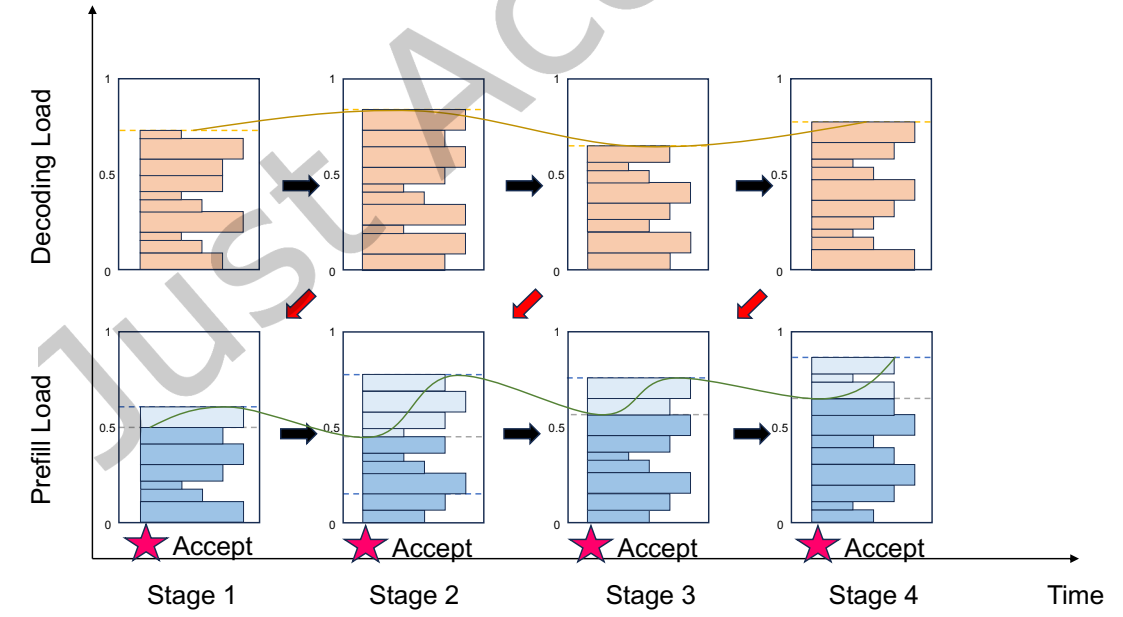
In Stage 1, the load on both prefill and decoding instances is low, so Conductor accepts a large number of requests until the load on prefill instances reaches its limit. In Stage 2, requests processed by prefill instances are scheduled to decoding instances, causing the load on decoding instances to be high. Consequently, Conductor rejects incoming requests, leading to a lower load on prefill instances. In Stage 3, no new requests enter the decoding stage, resulting in a decreased load. At this point, Conductor again accepts a large number of requests until the prefill instances are fully loaded. In Stage 4, as the load on decoding instances increases, Conductor rejects requests, causing a low load on prefill instances. This severe fluctuation in load between prefill and decoding instances results in poor resource utilization of the inference cluster.

4.3.4 Early Rejection Based on Prediction. To solve the load fluctuation problem, we propose a framework of **Early Rejection Based on Prediction** to address scheduling challenges in overload scenarios for disaggregated LLM serving systems like Mooncake. As illustrated in Figure 7b, this framework predicts the decoding load after the prefill stage of incoming requests and uses this prediction to decide whether to accept the requests, which helps mitigate the fluctuation problem. The core component of this strategy is the accurate prediction of the decoding load for the subsequent period. We introduce two approaches for this:

Request level: Previous work highlights a significant challenge in predicting loads for LLM serving: the unknown output length of each request. If we could determine the output length in advance, it would be possible to estimate the TTFT and TBT much more accurately. This, in turn, would help predict the number of requests a decoding



(a) Early Rejection.



(b) Early Rejection Based on Prediction.

Fig. 7. Instance load when applying Early Rejection and Early Rejection Based on Prediction.

instance can complete and the number of new requests that will be added after a specified time, thereby obtaining the load at that time. However, predicting each request's output length is challenging due to high costs [19] or low accuracy, especially under overload conditions where resources are scarce and accurate predictions are necessary, making request-level predictions particularly difficult.

System level: In contrast to request-level predictions, system-level predictions do not attempt to predict the completion time for individual requests. Instead, they estimate the overall batch count or the TBT status for instances after a specified time. This type of prediction is ongoing and requires less precision, making it more appropriate for overload scenarios.

In Mooncake, we currently utilize a system-level prediction strategy: we assume that each request's decoding stage takes a uniform time t_d. First, for a given moment t, requests that can be completed by the prefill instances at t are added to the uniform decoding instances. Next, requests that will be completed (i.e., their execution time exceeds t_d) before t are removed from the decoding instances. Finally, the average TBT ratio of all decoding instances to l_{tbt} is calculated to predict the load. The exploration of request-level prediction is left for future work.

5 Implementation

This section describes the implementation of Mooncake and the pragmatic lessons learned while running it in production. Several techniques described here were *not* part of the original blueprint; instead, they emerged through successive iterations as we confronted issues inherent to large-scale, distributed LLM serving. Because the ecosystem is still young and there is no agreed-upon reference design for critical modules (e.g., Conductor or Mooncake Store), implementation inevitably involves exploration and dead ends. By documenting both our missteps and our fixes, we hope to provide a useful playbook for practitioners and researchers entering this space.

5.1 Dynamic PD Ratio

In production, we observed a clear periodic shift in the relative loads of the prefill and decode clusters. During working hours, requests are dominated by long-form document analysis and retrieval-augmented queries, stressing the prefill stage; in the evening, mobile chat sessions with short prompts prevail, shifting the bottleneck to the decode stage. Our initial deployment used a static prefill/decode (PD) replica ratio, which caused predictable but pronounced imbalance.

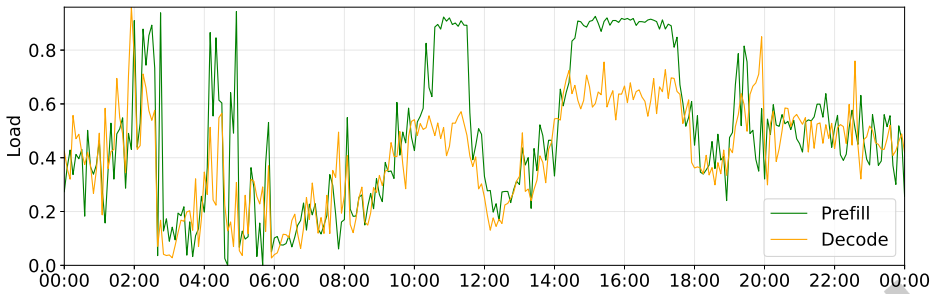
Frequent model upgrades made tuning harder. Each new revision exhibits different inference latency and response-length characteristics, so manual PD tuning quickly becomes both inaccurate and operationally expensive.

To address these issues we implemented an automated PD-ratio controller that periodically samples the recent prefill and decode loads and computes an *imbalance factor*:

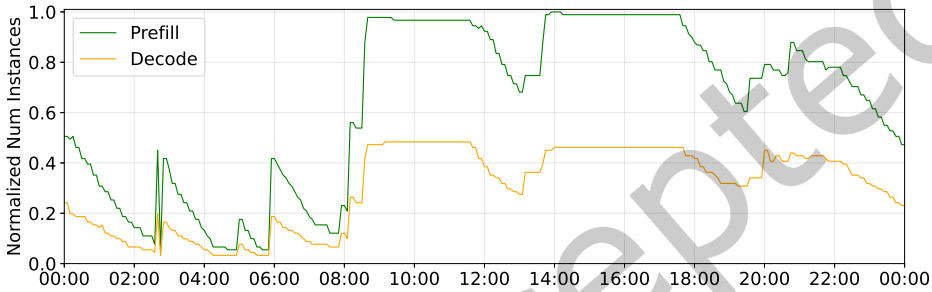
$$f_{im} = \frac{\overline{L}_{\text{decode}} + 1}{\overline{L}_{\text{prefill}} + 1}$$

where $\overline{L}_{\text{decode}}$ and $\overline{L}_{\text{prefill}}$ are the mean loads after discarding the top and bottom 10% of samples to suppress outliers. A $f_{im} \gg 1$ indicates that decode replicas are overloaded and should be increased; a $f_{im} \ll 1$ suggests that prefill replicas require scaling instead.

Because production traffic is highly skewed, driving f_{im} directly to 1 often causes oscillations. Instead, the controller follows the *machine tide*, which adds capacity during peak hours and removes capacity during off-peak periods, to adjust the PD ratio gradually. When scaling out, it first adds replicas to the more heavily loaded side; when scaling in, it removes replicas from the less-loaded side. Each adjustment is bounded to avoid overshoot



(a) Load traces.



(b) Replica count traces.

Fig. 8. A 24-hour trace of prefill and decode loads and replica counts in an online cluster.

while continuously reducing imbalance. Figure 8 illustrates the PD-ratio controller in action over a 24-hour period, demonstrating its ability to adaptively adjust both the number and ratio of prefill and decode replicas in response to real-time load fluctuations.

5.2 Flexible KVCache Storage

In the initial deployment of Mooncake, each node was provisioned with 1 TB of DRAM to form a distributed KVCache pool. A one-hour offline replay experiment showed that nearly all requests with shared prefixes benefited from high cache reuse, thanks to the global KVCache-sharing mechanism. Online metrics confirmed this benefit, with hit rates peaking at 50%. Over time, however, the hit rate degraded, falling below 20% during peak hours. A detailed analysis of production traffic revealed two primary causes:

(1) **Rising per-node throughput.** Continuous inference optimizations significantly increased the number of requests each node could serve. As a result, the volume of KVCache entries that had to be retained within a given time window began to exceed the original DRAM budget, triggering large-scale evictions.
(2) **Heterogeneous cache demands.** New serving features, such as context caching and web-search augmentation, introduced greater variance in prefix reuse. For example, reference content retrieved by a web search is typically used only once and does not reappear in subsequent dialogue turns, making it unnecessary to cache. In contrast, context cache allows users to designate segments of the prompt for repeated reuse over a period of time, requiring strong cache persistence. The uniform LRU-based eviction strategy originally

used by Mooncake was agnostic to these distinctions, leading to premature eviction of high-value entries and lower overall cache hit rates.

To address these issues, we introduced two key improvements to the cache storage and eviction strategy. First, we adopted a hierarchical caching architecture in which KVCache entries are stored across VRAM, DRAM, and SSD according to their access frequency. SSD is used only during peak load or for offloading long-lived context cache entries, and we prefetch proactively based on request-queue patterns to mask latency. Second, we replaced LRU with a semantics-aware eviction policy: low-value cache segments (e.g., ephemeral web search results) are excluded from Mooncake Store, while context-cache entries are retained with strong persistence guarantees and, in the worst case, offloaded to SSD instead of being evicted. Experimental results (§7.3.4) show that this adaptive KVCache storage strategy increases the global cache hit rate by 93%.

5.3 Stateless Conductor

In the early design of Mooncake, Conductor maintained the entire cluster's KVCache metadata in memory. Consequently, a failure or rolling restart of a Conductor instance would wipe out all cache information, (i) violating the context-cache protocol and (ii) reducing request throughput. Another issue is that the cluster's service had to be taken completely offline during an update: under Kubernetes deployment's rolling update, a newly started Conductor instance would incorrectly assume that all blocks were available, overwriting blocks that might still be under prefill or actively used for decoding, thereby corrupting ongoing inference.

We remedied both issues by making Conductor stateless. Instead of storing KVCache metadata (block IDs, block keys, and context cache entries) centrally, the metadata is offloaded to the inference nodes. Each inference node is now responsible for allocating KVCache blocks, acquiring locks to prevent concurrent modification, and committing updates. Conductor keeps only a shadow copy of the metadata in memory, used exclusively for cache-aware scheduling (§4); all state-changing operations occur on the nodes. On startup, a Conductor instance reconstructs its shadow view by querying every node, and after each request's prefill phase, it refreshes the view with the information returned by the node.

Figure 9 illustrates the workflow:

(1) Upon receiving a request, the Conductor instance performs prefix matching using its shadow metadata and selects the optimal prefill node.
(2) It then issues a `TxAllocate` to the chosen node. The node checks local capacity:
    - If capacity is insufficient, it returns an error and the Conductor instance rolls back.
    - Otherwise, it allocates the required blocks (excluding those reused via prefix matching), locks all relevant blocks (both reused and newly allocated), and returns the handle set.
(3) When the first token arrives from the decode node—indicating KVCache transfer has completed—Conductor issues `TxCommit` to the prefill node, which persists the block keys and IDs.

Mooncake also supports inter-node KVCache transfer to improve cache load balancing (§4.2). In the stateless design, the Conductor instance initiates paired `TxAllocate` operations on the source and destination nodes; the source performs no allocation but participates in locking its blocks. Upon finished transfer, both nodes commit.

The stateless Conductor design enables safe rolling updates in which old and new Conductor instances may concurrently handle requests. The KVCache block locking mechanism at the node level ensures that two Conductor instances cannot mutate the same block concurrently. Temporary inconsistencies are possible. For example, the new Conductor instance updates a block while the old one still makes scheduling decisions based on stale metadata, which may lead to suboptimal node selection. But updates are short-lived, and we observe negligible impact on production workloads.

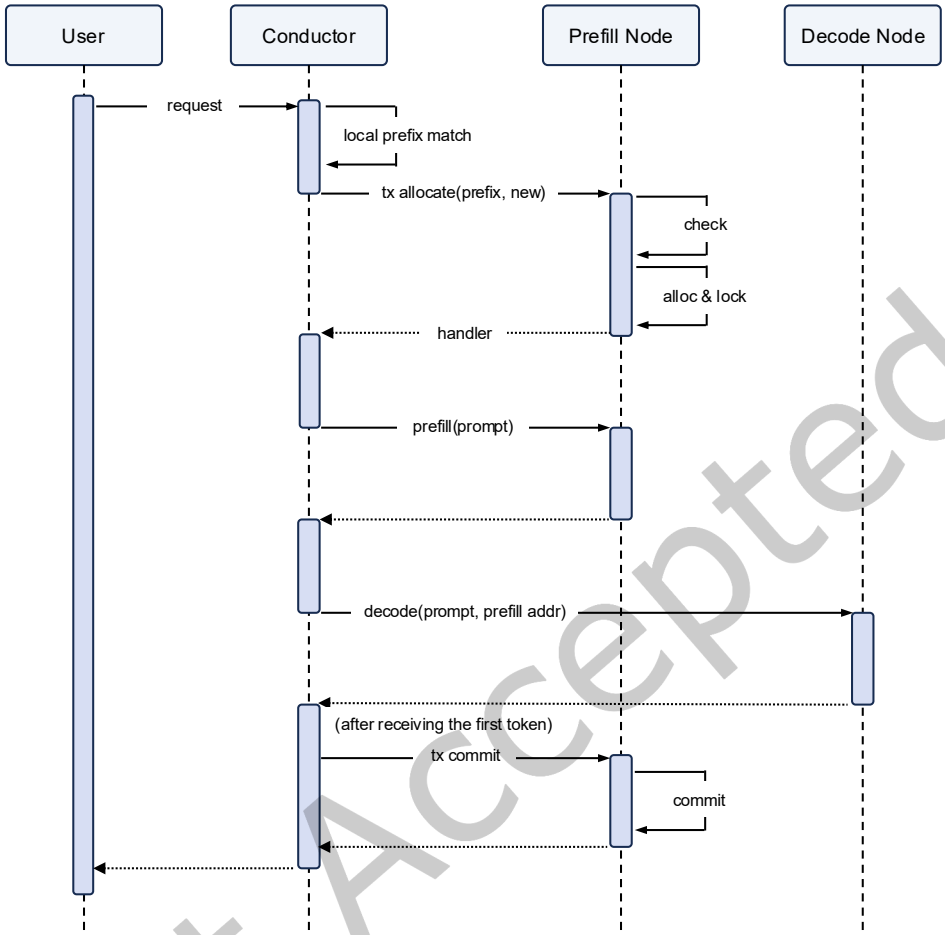


Fig. 9. Interaction flow for a single request between the Conductor instance and the inference nodes.

6 Open-source Mooncake

6.1 Overview of the Open-source Mooncake Architecture

Popular LLM inference frameworks such as SGLang [51], vLLM [23], and LMDeploy [8] have attracted large and active communities due to their ease of use, high performance, and multi-platform deployment support. These frameworks implement advanced features such as continuous batching and prefix caching, substantially improving inference throughput. However, despite rapid progress, their support for P/D disaggregation and distributed KVCache reuse remains limited. Before integrating Mooncake, for example, vLLM relied on PyNCCL (Python bindings for NCCL [10]) for multi-node P/D disaggregation, which exhibited poor performance and a large failure domain due to NCCL's collective communication pattern being ill-suited for point-to-point communication. Furthermore, while both vLLM and SGLang implemented prefix caching early, their caches were confined to local GPU memory and could not be shared across instances. This limitation significantly degrades their performance under long-context scenarios (see §7.2.3). To accelerate progress in distributed LLM inference,

we have open-sourced the Mooncake system and provided third-party integration with these popular inference frameworks.

As illustrated in Figure 10, the core component of the open-source Mooncake architecture [22] is the Transfer Engine. It unifies diverse protocols including RDMA, NVLink, TCP, and NVMe-oF, under a single transfer semantics API. For RDMA-like protocol, open-source Transfer Engine supports eRDMA [7] to adapt to different server vendors and cloud service providers such as Alibaba Cloud. The topology-aware design dynamically optimizes data paths based on hardware locality, achieving a remarkable throughput for KVCache transfers while reducing cross-node latency by 40% compared to conventional approaches like Gloo [41]. Building upon Transfer Engine, Mooncake implements two specialized storage subsystems: the P2P Store enables efficient KVCache sharing across nodes through its decentralized architecture, while the Mooncake Store provides a distributed KVCache pool specifically optimized for LLM inference workloads. Thus, these components can work together to support KVCache optimized features like P/D disaggregation and KVCache reuse in popular LLM inference systems.

Currently, Mooncake has been integrated into the aforementioned open-source LLM frameworks. Through collaboration with the SGLang and vLLM teams, Mooncake now officially supports P/D disaggregation in these two open-source LLM inference systems. By leveraging the high-efficiency communication capabilities of RDMA devices, Mooncake significantly improves inference efficiency in P/D disaggregation scenarios, providing robust technical support for large-scale distributed inference tasks. These integrations demonstrate the role of open-sourced Mooncake architecture as a pluggable efficiency booster for LLM systems.

LLM inference frameworks adopt diverse KVCache management strategies, including variations in storage layouts, eviction policies, and access interfaces. To ensure broad compatibility, Mooncake exposes modular APIs from components such as the Transfer Engine and Mooncake Store, enabling tailored integration with different framework designs. There are two major design choices in integrating Mooncake into an open-source inference framework: using direct GPU-to-GPU access or CPU offload and applying KV Transfer or KV Store paradigms. As shown in Table 2, Mooncake supports all these features for better compatibility.

GDR and CPU offload. GPUDirect RDMA (GDR) offers lower latency by direct GPU communication but requires sufficient GPU memory and CUDA support. Mooncake can utilize CPU offload, which is more scalable and offers larger and cheaper CPU memory, but this incurs higher latency from CPU-GPU data transfers.

KV Transfer and KV Store. Use Transfer Engine as the KV Transfer mode, which involves moving KVCache between devices as needed. It can cause high bandwidth usage but lower access latency if managed well. Mooncake Store provides the KV Store mode. It uses a centralized, distributed cache pool, which centralizes management, reduces redundancy, and optimizes for scalability, but may have higher access latency due to centralized storage. However, efficient serialization and network optimizations can mitigate the overhead.

Table 2. Strategies of current open-source LLMs

Operation	Transfer Method	
	GPU-to-GPU (GDR)	CPU Offload
KV Transfer	SGLang	-
KV Store	-	vLLM

6.2 Integration with SGLang

The modern deployment of LLMs like Kimi K2 [33] and DeepSeek R1 [17] requires careful orchestration of computation and communication to achieve scalable and cost-effective inference. The SGLang framework

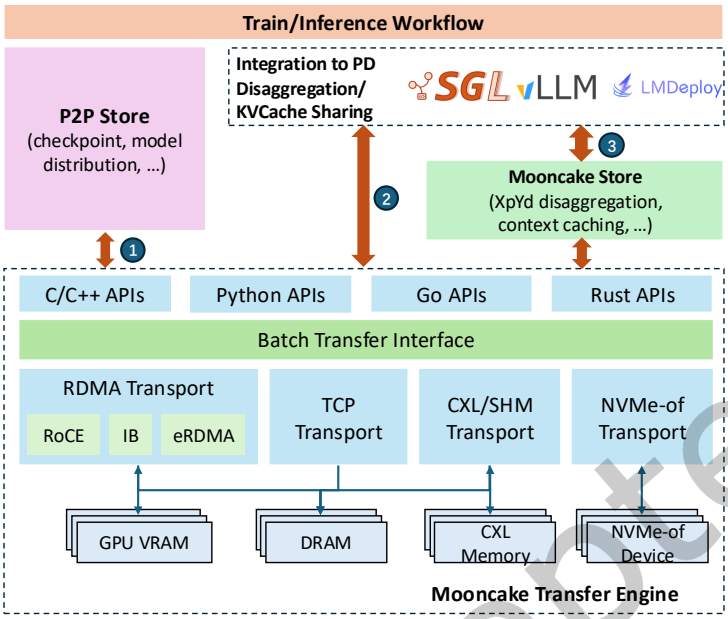


Fig. 10. Current support status of the open-source LLM serving architecture.

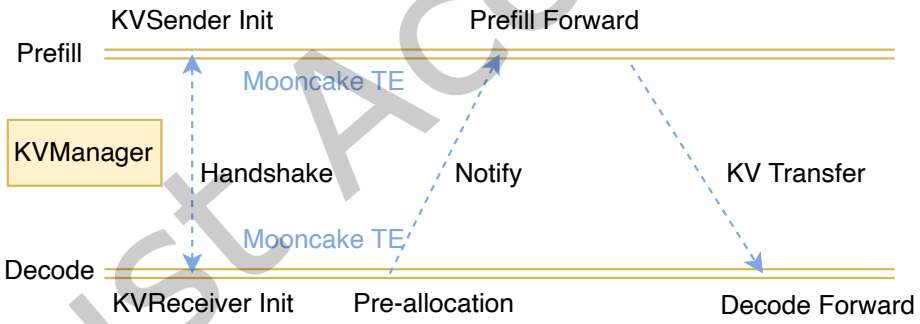


Fig. 11. Workflow of SGLang P/D disaggregation with Mooncake.

addresses this challenge by providing a flexible, high-performance environment that supports advanced parallelism and disaggregation techniques. A critical enhancement to the capability of SGLang is the integration of Mooncake to accelerate P/D disaggregation communication.

To overcome communication bottlenecks, Mooncake provides a streamlined RDMA communication layer explicitly built for the unique requirements of P/D disaggregation. There are four steps in the workflow (see Figure 11): **(1)** It begins with the `KVManager` initializing the Mooncake, which involves registering GPU addresses into RDMA, creating a request pool to map bootstrap rooms to metadata, and setting up a ZMQ server for communication. **(2)** The `KVSender` adds metadata entries to the request pool when initiating a transfer. A pair of

prefill and decode server connections is established for each request. This approach allows us to easily scale the prefill and decode server pools up or down as needed. **(3)** Upon initialization, the `KVReceiver` also populates the request pool and sends metadata to the `KVSender` via ZMQ, triggering prefill thread of `KVSender`. This thread performs GDR-based RDMA writes using Mooncake transferSync and notifies the `KVReceiver` upon completion. **(4)** Finally, the decode thread of `KVReceiver` processes the completion signal, removing the corresponding entry from the request pool. This coordinated process ensures efficient data transfer and synchronization between components.

Efficient Fragmented Memory Transfers. For each KV transfer, there are many fragmented GPU memory indices. Mooncake employs RDMA queue pairs to transfer contiguous fragmented KVCache data without additional copying, minimizing overhead. On the other hand, Mooncake offers batch transfer APIs to reduce the invocation overhead between SGLang and Mooncake.

Asynchronous and Non-blocking APIs. SGLang offloads network operations of Mooncake to dedicated background threads, ensuring that the scheduler event loop in SGLang remains fully responsive and unhindered by blocking communication calls. Transfer engine operations of `KVSender`/`KVReceiver` are non-blocking and run in a background thread. This ensures that the original scheduler event loop continues to operate uninterrupted while data transfer occurs in the background.

Lightweight and Modular Integration. Mooncake is designed with modular interfaces compatible with communication abstractions of SGLang, such as DeepEP [12], allowing integration without extensive refactoring. By integrating Mooncake as a module and avoiding interference between Prefill and Decode servers, Mooncake can smoothly support tensor/data/pipeline parallel execution, achieving high GPU utilization across distributed inference workloads.

Fault-tolerance with Mooncake. Mooncake introduces robust mechanisms to handle failures and maintain system stability for P/D fault tolerance in SGLang. It includes health monitoring via a dedicated route and heartbeat threads, enabling detection of P/D instance failures. Upon identifying issues (e.g., sync timeouts or killed instances), the system aborts affected requests, restarts unhealthy P/D, and cleans up KV states to prevent residual conflicts. Failures are synchronized between P/D nodes to propagate root causes, while reduced resource contention and configurable environment variables improve debuggability and adaptability. These changes ensure uninterrupted service.

6.3 Integration with vLLM

Mooncake implements a producer-consumer architecture with stateful connectors to enable asynchronous P/D disaggregation in vLLM. The system introduces three core components: (1) `Lookup Buffers` implemented as double-ended queues for non-blocking KVCache insertion by prefill and blocking consumption by decoder, (2) `Connector` and `Pipes` leveraging Mooncake Store for remote transmission with zero-copy optimization, and (3) State-aware Scheduler that manages GPU buffer allocation through allocate/free slots abstraction.

Figure 12 shows the workflow of how to transfer KVCache between P/D with Mooncake Store. This asynchronous architecture separates prefill and decode nodes using a standalone Mooncake store. It ensures that prefill nodes do not need to wait for decode nodes to retrieve the KVCache. During the initialization stage of the P/D instances, P/D should create a Mooncake client as `Connector` and attach to the KVCache pool. While P instances perform prefill computations forward, Mooncake simultaneously pushes partial KVCache to D connectors without blocking the computation pipeline. Prefill instances process the requests with the output length of 1 and put the KVCache to Mooncake Store via `Connector`. Control and data plane separation is achieved via HTTP/message queue coordination and direct KVCache addressing. This allows decoder instances to bypass prefill phases by retrieving KVCache directly from Mooncake Store and proceeding with decoding.

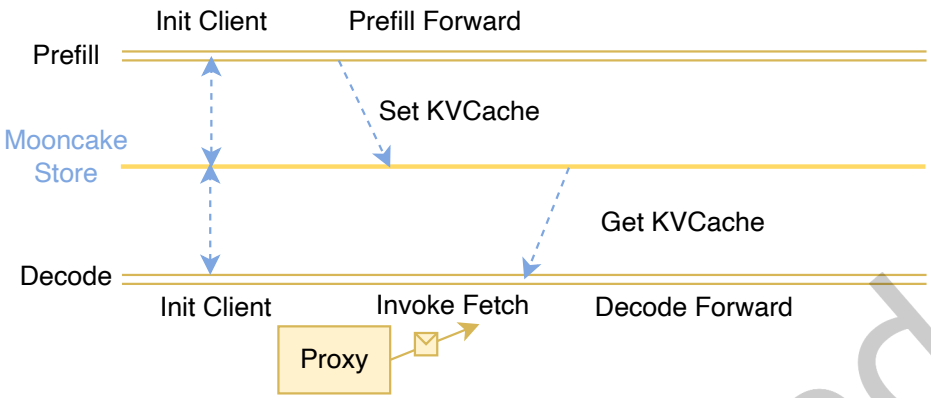


Fig. 12. Workflow of vLLM P/D disaggregation with Mooncake.

By eliminating intermediate data copies through zero-copy pipe implementation and maintaining cache state consistency via Connector-based allocation tracking, Mooncake reduces KV transmission latency compared to the native page cache replication approach of vLLM while preserving the original block management semantics.

7 Evaluation

As described before, according to historical statistics of Kimi, Mooncake enables Kimi to handle 115% and 107% more requests on the A800 and H800 clusters, respectively, compared to our previous systems based on vLLM. To further validate these results and ensure reproducibility, in this section, we conduct a series of end-to-end and ablation experiments on Mooncake with a dummy LLaMA3-70B model to address the following questions: *1)* Does Mooncake outperform existing LLM inference systems in real-world scenarios? *2)* Compared to conventional prefix caching methods, does the design of Mooncake Store significantly improve Mooncake's performance?

7.1 Setup

Testbed. During the reproducing experiments, the system was deployed on a high-performance computing node cluster to evaluate its performance. Each node in the cluster is configured with eight NVIDIA-A800-SXM4-80GB GPUs and four 200 Gbps RDMA NICs. The KVCache block size in Mooncake Store is set to 256. For deploying Mooncake, each node operates as either a prefill instance or a decoding instance based on the startup parameters. For deploying other systems, each node hosts a single instance.

Metric. Specifically, we measure the TTFT and TBT of each request, where the TBT is calculated as the average of the longest 10% of the token arrival intervals. As mentioned in §2, the threshold for TTFT is set to 30 s, and TBT thresholds are set to 100 ms, 200 ms, and 300 ms, depending on the scenario. We consider requests with both TTFT and TBT below their respective thresholds as effective requests, and the proportion of effective requests among all requests as the effective request capacity. For brevity, the subsequent experiments not mentioning TTFT are assumed to meet the TTFT threshold. To more intricately compare the caching performance, we also measure the GPU time during the prefill stage and the cache hit rate for each request.

Baseline. We employ vLLM [23], one of the state-of-the-art open-source LLM serving systems, as our experimental baseline. vLLM features continuous batching and PagedAttention technologies, significantly enhancing inference throughput. Despite its strengths, vLLM's architecture, which couples the prefill and decoding stages,

Table 3. Workload Statistics.

	Conversation	Tool&Agent	Synthetic
Avg Input Len	12035	8596	15325
Avg Output Len	343	182	149
Cache Ratio	40%	59%	66%
Arrival Pattern	Timestamp	Timestamp	Poisson
Num Requests	12031	23608	3993

can disrupt decoding, especially in scenarios involving long contexts. Recent updates to vLLM have integrated features like prefix caching and chunked prefill to improve performance metrics in long-context scenarios, such as TTFT and TBT. In our experiments, we also compare these features of vLLM. In our experiments, we utilize the latest release (v0.5.1) of vLLM. Due to limitations in the current implementation, we test the prefix cache and chunked prefill features of this version separately.

7.2 End-to-end Performance

In our end-to-end experiments, we evaluate the request handling capabilities of Mooncake and baseline systems under various workloads. Specifically, we measure the maximum throughput that remains within the defined SLO thresholds. We employ three types of workloads in our tests: two real-world traces sampled from Kimi that represent online conversations and tool&agent interactions, respectively, and a synthetic workload to cover different inference scenarios. We will first describe the unique characteristics of these workloads and then discuss the results. Lastly, we analyze the GPU computation time during the prefill stage, further demonstrating the advantages of Mooncake Store in enhancing cache utilization and reducing computation costs.

7.2.1 Workload. **Conversation workload.** Chatbots [32, 35] represent one of the most prevalent applications of LLMs, making conversational requests a highly representative workload for LLM inference. As shown in Table 3, the conversation workload contains a significant portion of long-context requests—reaching up to 128k tokens and averaging around 12k tokens—which is comparable to the data lengths found in current long-context datasets [3, 25]. Moreover, the workload has an average of approximately 40% prefix caching ratio brought about by multi-turn conversations. We sampled 1 hour of conversation traces from an online inference cluster, where each record includes the input and output lengths along with timestamps of arrival. Requests are dispatched according to these timestamps and are preemptively terminated once the model output reaches the predetermined length.

Tool&Agent workload. Recent studies [38] involving LLMs deployed as tools or agents to perform tasks have been increasing. These tasks are typically characterized by the incorporation of pre-designed, often lengthy, system prompts that are fully repetitive. We collected traces of the tool&agent workload, also sampled over a 1-hour period. As indicated in Table 3, this workload exhibits a high proportion of prefix caching, with shorter input and output lengths.

Synthetic workload. The synthetic workload was constructed from a combination of publicly available datasets. We categorized the requests in the real trace into three types: short conversations, tool and agent calls, and long text summarization and QA. For each category, we selected the following datasets: ShareGPT [42], Leval [3], and LooGLE [25]. ShareGPT comprises multi-turn conversations with short input lengths. Leval serves as a benchmark for evaluating model performance over long contexts, simulating scenarios where requests involve lengthy system prompts typical of tool and agent interactions. LooGLE is tailored for long-context QA and summarization tasks, featuring input lengths of up to 100k tokens and including both multi-turn QA and

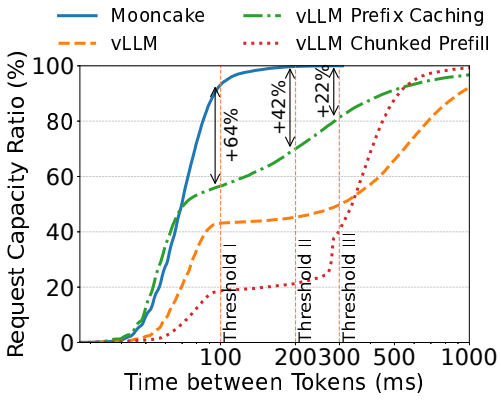


Fig. 13. The experiment of the effective request capacity of Mooncake under the tool&agent workload.

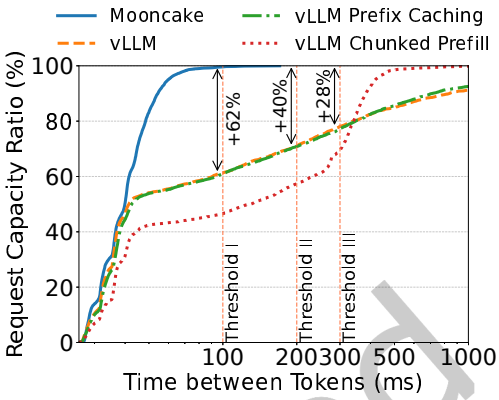


Fig. 14. The experiment of the effective request capacity of Mooncake under the synthetic workload.

single-turn summarizations, making it well-suited for long text summarization and QA scenarios. Overall, the synthetic workload has the longest average input length. Despite having the highest proportion of prefix caching, its cache hits are quite dispersed, thus requiring a substantial cache capacity.

During preprocessing, each conversation turn was mapped into a separate request, incorporating both the input and outputs from previous interactions. For datasets featuring multiple questions with the same lengthy prompt, each question and its preceding prompt were treated as a single request. We combined the processed datasets in a 1:1:1 ratio, preserving the sequential relationships within the multi-turn dialogue requests while randomly shuffling them. Since the datasets do not specify arrival times, we simulated realistic conditions by dispatching requests at a defined rate using a Poisson process.

7.2.2 Effective Request Capacity. To assess the maximum number of requests that can adhere to the SLOs under different workloads, we test four system configurations: Mooncake, vLLM, vLLM with the prefix caching feature, and vLLM with the chunked prefill feature, each utilizing 16 nodes.

Conversation workload. The results for this workload are presented in Figure 1. This workload, characterized by varying input lengths and longer output lengths, causes significant fluctuations in TBT for the vLLM system due to the lengthy contexts in the prefill stage. While chunked prefill reduces decoding interference, balancing the enhancement of MFU in the prefill stage with the TBT constraints in the decoding stage remains challenging. Despite meeting the TTFT SLO, its effective request capacity is still suboptimal. Compared to vLLM, Mooncake achieves a very significant increase in effective request capacity.

Tool&Agent workload. In contrast, the tool&agent workload has a high proportion of prefix caching and shorter output lengths, favoring the vLLM system as the short prefill time minimally impacts output. However, as illustrated in Figure 13, vLLM and vLLM with chunked prefill experience more severe disruptions in decoding due to longer prefill processing times, resulting in a lower effective caching capacity than vLLM with prefix caching. Mooncake uses a global cache pool to significantly increase caching capacity and optimize cache utilization through internode transfers, excelling in scenarios with high prefix caching. As a result, it enhances effective caching capacity by 42% compared to vLLM with prefix caching under the 200 ms threshold.

Synthetic workload. The synthetic workload features the longest average input lengths and dispersed cache hotspots, which leads to poor cache utilization under smaller cache capacities. As depicted in Figure 14, most

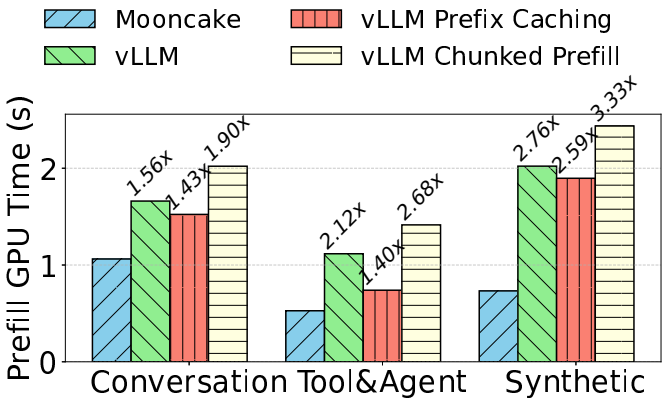


Fig. 15. Average GPU time of each request during the prefill stage under different workloads.

requests processed by Mooncake maintain a TBT within 100 ms, whereas about 20% of requests handled by vLLM exceed 300 ms. The performance of systems with prefix caching and chunked prefill is similar to vLLM, as they fail to mitigate the impact of long contexts on the decoding stage. Compared to vLLM, Mooncake increases effective request capacity by 40% under the 200 ms threshold.

7.2.3 Prefill GPU Time. Prefill GPU time is positively correlated with requests' TTFT and serving cost and is determined by requests' input lengths and cache hit rates. We analyze the average GPU time during the prefill stage under different workloads, as shown in Figure 15. For Mooncake, the conversation workload incurs the longest prefill GPU time due to its longer input lengths and lower prefix cache ratio. The synthetic workload, featuring the highest prefix cache ratio and dispersed cache hotspots, achieves optimal cache hit rates within Mooncake's global cache pool. Consequently, despite having the longest average input lengths, it requires less prefill GPU time than the conversation workload. Finally, the tool&agent workload exhibits the shortest prefill GPU time because it has the shortest average input length and a relatively high prefix cache ratio.

Across different systems, Mooncake significantly reduces GPU time by fully utilizing global cache for prefix caching, achieving reductions of 36%, 53%, and 64% for conversation, tool&agent, and synthetic workloads, respectively, compared to vLLM. vLLM featuring prefix caching uses local cache stored on HBM, where the cache capacity is far lower than that of Mooncake. Its prefill GPU time is 1.43× and 1.40× higher than Mooncake for the conversation and tool&agent workloads, respectively. However, in the synthetic workload, where cache hotspots are more dispersed, the prefill GPU time of vLLM with prefix caching is nearly equivalent to vLLM and is 2.59× that of Mooncake. vLLM with chunked prefill sacrifices some prefill efficiency to maintain lower TBT during the decoding stage, resulting in the longest prefill GPU times, which are 1.90×, 2.68×, and 3.33× that of Mooncake for the three workloads.

7.3 Mooncake Store

To address Question 2, we examine the effects of Mooncake Store's global cache pool on system performance. Our analysis reveals that although using local DRAM to construct KVCache memory increases cache capacity than HBM only, restricting the cache to a single node still leads to suboptimal cache utilization. We will first conduct a quantitative analysis of cache capacity requirements and then showcase the benefits through practical workload experiments.

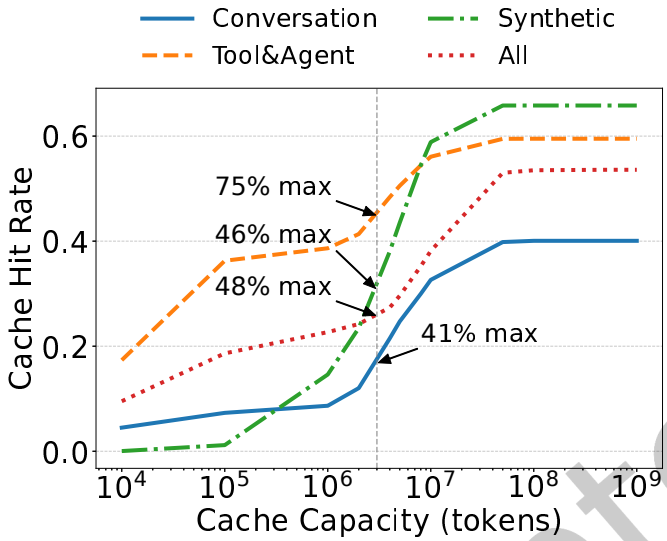


Fig. 16. Quantitative analysis of prefix cache hit rates with varying cache capacities. We consider only the sequence of requests and do not account for factors such as prefill computation time or the replication of hotspots in the cache. The dashed line for 3M tokens capacity represents the local cache capacity, with the intersection points indicating the ratio of the cache hit rate to the theoretical maximum hit rate.

7.3.1 Quantitative Analysis of Cache Capacity. Considering the LLaMA3-70B model, the KVCache size required for a single token is 320 KB. Despite the possibility of reserving approximately 1 TB of DRAM for local caching, this setup only supports storage for about 3 million tokens, which proves insufficient. Figure 16 displays theoretical cache hit rates under various workloads and their combinations. The findings indicate that a local cache with a 3M token capacity does not achieve 50% of the theoretical maximum hit rate in most scenarios. We also determine that, in these workloads, a cache capacity of 50M tokens nearly reaches the theoretical maximum hit rate of 100%, which requires to pool at least 20 nodes' DRAM. The results highlight that a global cache significantly enhances capacity over local caches, thus improving cache hit rates and reducing GPU times.

7.3.2 Practical Workload Experiment. To evaluate the effectiveness of global versus local caching mechanisms, we focus on two metrics: cache hit rate and average GPU computation time for prefill. We configure a cluster with 10 prefill nodes and restrict all request outputs to 1 to isolate the impact of the decoding stage. Each node in the local cache setup has a 3M token capacity but can only access its own cache. The global scheduler is programmed to direct requests to nodes with higher prefix match ratios to maximize cache utilization. Conversely, in the global cache setup, each node also has a 3M token capacity but can share caches across all nodes, supported by proactive inter-node cache migration. The experimental data, shown in Figure 17, indicates that the global cache achieves higher cache hit rates and shorter average prefill GPU computation times across all tested workloads. Compared to the local cache, the global cache exhibits a maximum increase of 136% in cache hit rate and a reduction of up to 48% in prefill computation time.

7.3.3 Cache Replica. Building upon the cache load balancing scheduling strategy discussed in §4.2, the cache keys in Mooncake Store may have replicas distributed across different machines, thereby reducing access latency

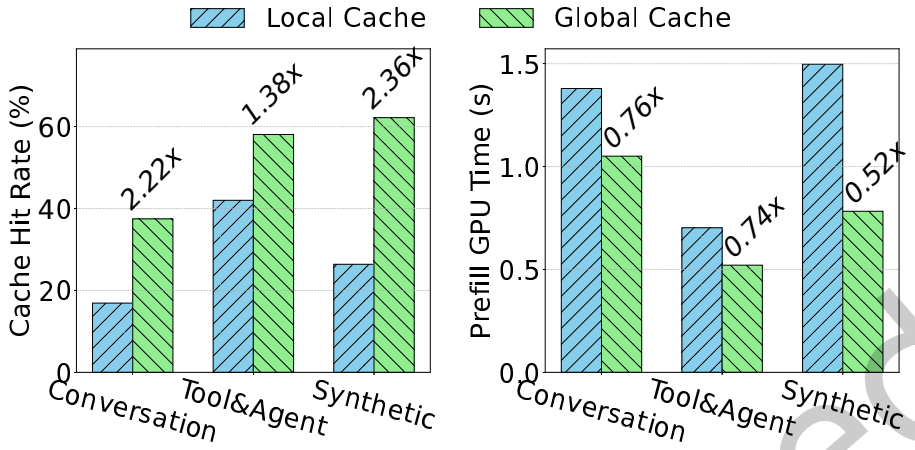


Fig. 17. Cache hit rates and average GPU computation time for prefill in global and local caches.

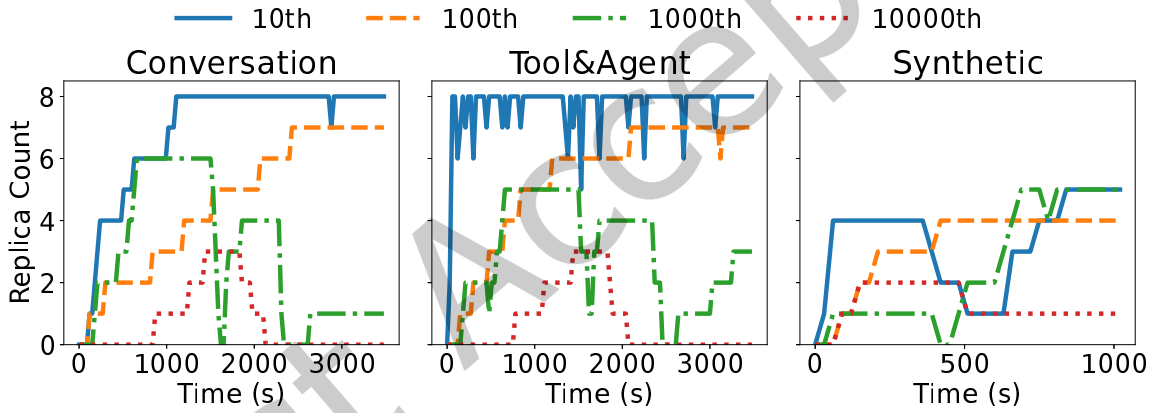


Fig. 18. Replication count of cache keys across various workloads. We continuously monitor and record the keys and counts of all cache blocks every 30 seconds, subsequently ranking the cache keys by the cumulative counts from all samples. This figure depicts the temporal variation in replication numbers for cache keys ranked at the 10th, 100th, 1000th, and 10,000th positions.

for hot caches. To further investigate the system's dynamic behavior, we count the number of cache replicas for keys across three workloads, as shown in Figure 18.

It can be observed that in the conversation and tool&agent workloads, there are highly concentrated hot caches (e.g., the top 100 keys), which, after the system stabilizes, have replicas on almost every instance in the prefill pool. In contrast, the synthetic workload has fewer shared prefix caches, resulting in fewer replicas and potential fluctuations, even for the top 10 blocks. This demonstrates that our scheduling strategy in §4.2 effectively provides replicas for hot caches, particularly in scenarios with highly concentrated prefix caches.

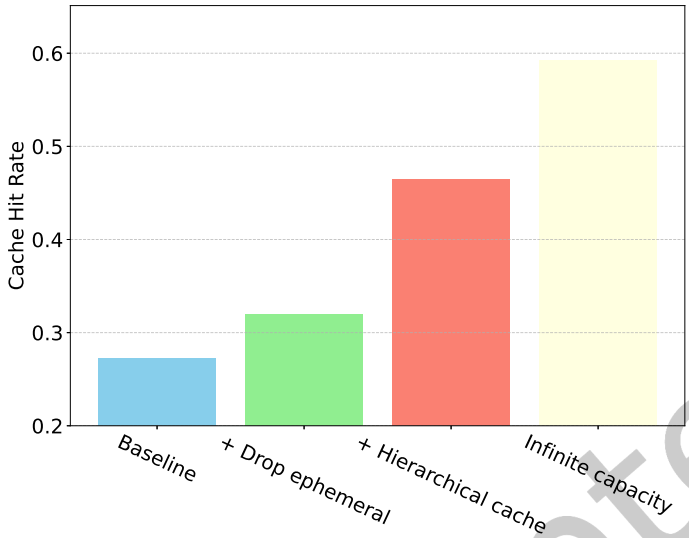


Fig. 19. Cache hit rates obtained under different KVCache storage strategies.

7.3.4 Flexible KVCache Storage. This section empirically evaluates the ability of the hierarchical caching architecture and the semantics-aware eviction policy proposed in §5.2 to improve cache hit rate. We perform an offline analysis using request traces collected from approximately 700k user sessions. With the global cache capacity fixed, we use the standard LRU eviction strategy as our baseline.

Starting from this baseline, we incrementally enable two optimizations: (1) filter out all requests marked as `ephemeral` (around 62%), which are unlikely to recur and thus not cached; (2)introduce a hierarchical KVCache and ensure that requests marked with `context cache` (around 8%) are not evicted. For comparison, we also evaluate an oracle system with an unbounded cache, which represents an upper bound on achievable hit rate.

Due to the high proportion of one-off requests, the baseline LRU achieves a modest hit rate of 27% (i.e., the number of cache blocks hit divided by the number of cache blocks queried is 27%). Filtering out ephemeral requests yields a minor improvement. Incorporating the hierarchical KVCache significantly increases the hit rate to 52%, corresponding to a 93% relative improvement over the baseline.

7.4 KVCache Transfer Performance

7.4.1 Transfer Engine. Mooncake's transfer engine is designed to facilitate efficient cache transfers between nodes. We compare its latency with other popular schemes, considering two alternative baselines: `torch.distributed` with a Gloo backend [41] and TCP-based transfers. All schemes are tested with a concurrency level of 64 and a minimum transfer granularity of 128 KB. As depicted in Figure 20, the transfer engine consistently exhibits significantly lower latency than the alternative methods. In the scenario of transferring 40 GB of data, corresponding to the cache size for LLaMA3-70B with 128k tokens, the transfer engine achieves bandwidth of 87 GB/s and 190 GB/s under network configurations of 4×200 Gbps and 8×400 Gbps, respectively. These rates are approximately 2.4× and 4.6× faster than those achieved using the TCP protocol. The transfer engine's code is also open-sourced and serves as a basic tool applicable to many scenarios (e.g., it is also used in the checkpoint transfer service of Moonshot AI).

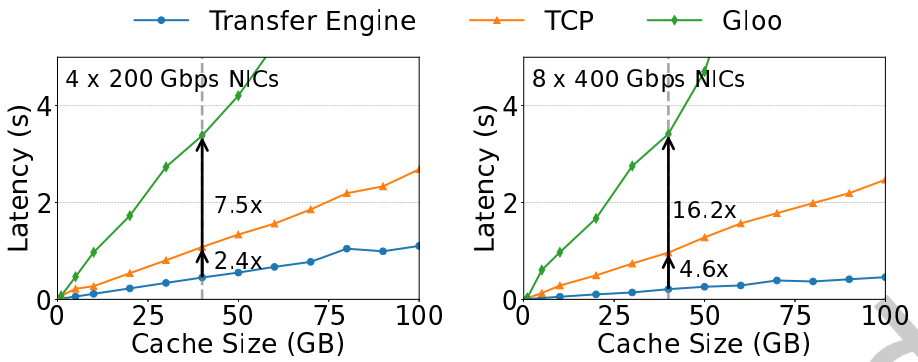
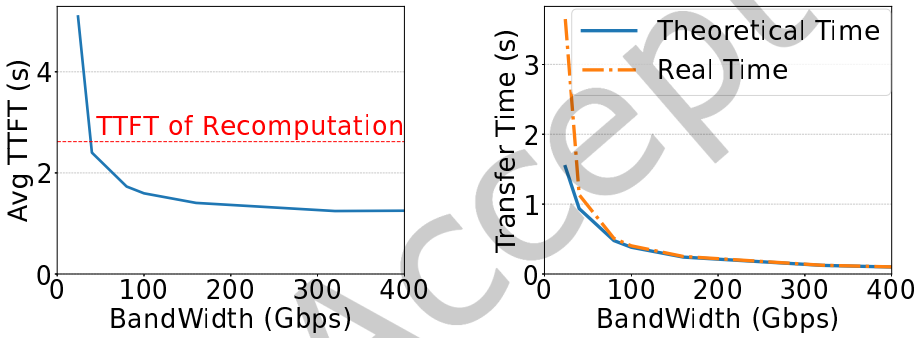


Fig. 20. Latency of inter-node cache transfer.



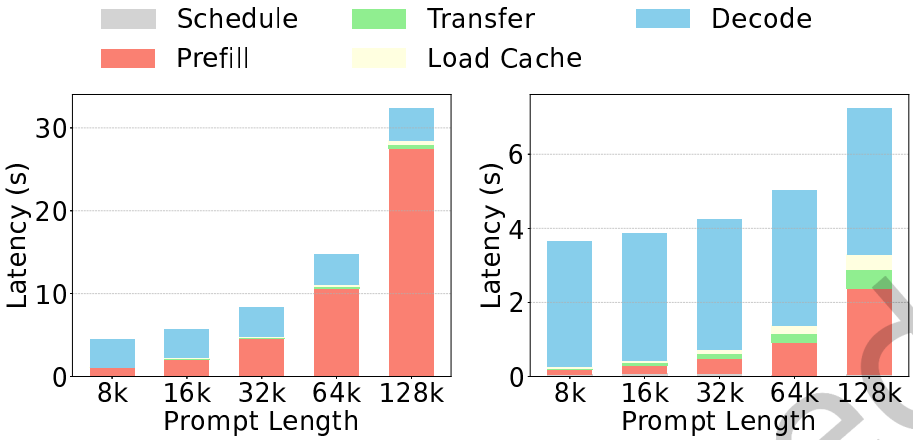
(a) Average TTFT.

(b) Transfer time.

Fig. 21. The synthetic workload experiment with varying network bandwidths.

7.4.2 Bandwidth Demand by Mooncake. Mooncake's global cache pool relies on efficient inter-node cache transfers to hide cache transfer times within GPU computation times. We evaluate the impact of network bandwidth on the system's performance by simulating a range of bandwidths from 24 Gbps to 400 Gbps and measuring the transfer time and TTFT under the synthetic workload described in §7.2.1. Figure 21a shows that the average TTFT of requests decreases as bandwidth increases. When the total communication bandwidth exceeds 100 Gbps, the average TTFT remains below 2 s, significantly less than the TTFT of the recomputation baseline. However, when bandwidth falls below 100 Gbps, system performance is significantly compromised. This is marked by a sharp increase in TTFT and evident network congestion, as demonstrated by the substantial divergence between actual and theoretical transfer times illustrated in Figure 21b. Consequently, we recommend a minimum network bandwidth of 100 Gbps to ensure optimal system performance.

7.4.3 E2E Latency Breakdown. The latency of a single inference request in Mooncake can be decomposed into five components: *1)* scheduling and queuing time; *2)* layer-wise prefill time; *3)* cache transfer time; *4)* time required



(a) Prefix cache ratio 0%.

(b) Prefix cache ratio 95%.

Fig. 22. End-to-end latency breakdown of Mooncake. In the figure, *Prefill* represents the time for layer-wise prefill that integrates cache loading and storing, and *Decode* represents the time to decode 128 tokens. All processes with diagonal stripes can proceed asynchronously with model inference and do not affect Mooncake's throughput.

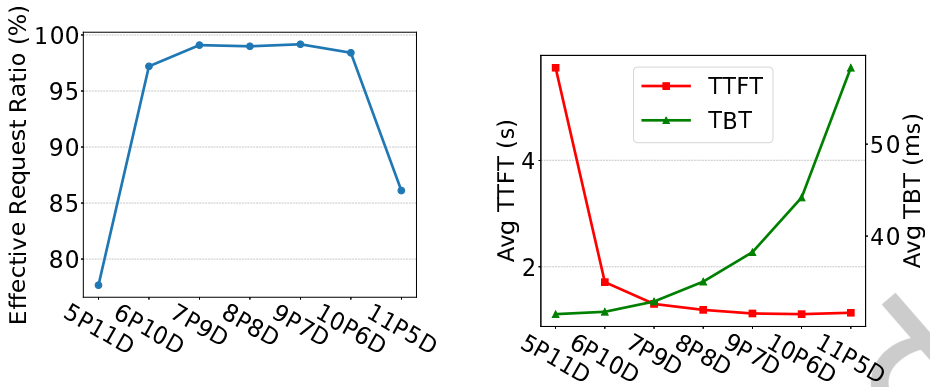
for the decoding node to load cache from DRAM to HBM; and *5)* decoding time. We experimentally analyze the proportion of these five components under settings with prefix cache ratios of 0% and 95%, as shown in Figure 22.

First, it is evident from the figure that the introduction of prefix caching significantly reduces the prefill time. Specifically, with an input length of 128k tokens, prefix caching reduces the prefill time by 92%. Furthermore, the overhead introduced by Mooncake has minimal impact on the system's performance. The *Schedule*, *Transfer*, and *Load Cache* components can proceed asynchronously with model inference and therefore do not affect Mooncake's throughput. Moreover, the increase in TTFT due to these overheads is smaller than the reduction achieved by prefix caching. Even when accounting for the overhead, prefix caching in Mooncake can reduce TTFT by 86% with an input length of 128k tokens.

7.5 P/D Ratio

As a deployed P/D disaggregation system, in this section, we explore the impact of different P/D ratios on system performance. We define the P/D ratio as the number of prefill nodes to decoding nodes. Using the clusters comprising 16 nodes but with varying P/D ratios, we measure the average TTFT and TBT under the synthetic workload described in §7.2.1. We then calculate the effective request capacity as introduced in §7.2.2, setting the thresholds for TTFT and TBT to 10 seconds and 100 milliseconds, respectively. Increasing the number of prefill nodes reduces TTFT but increases TBT, and vice versa (Figure 23b). Therefore, we need to find a balance between TTFT and TBT. Figure 23a demonstrates that when the P/D ratio is approximately 1:1, Mooncake achieves its highest effective request capacity, indicating that the loads on the prefill and decoding clusters are relatively balanced.

However, the average request input length, output length, and prefix cache hit rate vary significantly across different workloads, and these factors all affect the optimal P/D ratio. In practical deployments, we use the automated PD-ratio controller from §5.1 to adaptively adjust the number of the prefill and decoding instances in a cluster.



(a) Effective request capacity with varying P/D ratio.

(b) Latency with varying P/D ratio.

Fig. 23. The impact of the P/D ratio on the system performance. P is short for prefill nodes and D is short for decoding nodes.

Table 4. Number of requests rejected by the system under the overloaded-scenario experiment.

	Baseline	Early Rejection	Early Rejection based on Prediction
Number of rejected requests	4183	3771	3589

7.6 Performance in Overload Scenarios

In this section, we evaluate performance under overload scenarios, focusing on the maximum number of requests the system can handle, as discussed in §4.3. The baseline strategy, which rejects requests based on load before both stages start, leads to resource wastage by rejecting requests already processed in the prefill stage. In contrast, we propose the Early Rejection and Early Rejection based on Prediction strategies, detailed in §4.3.2 and §4.3.4, respectively. These strategies take the system's load into comprehensive consideration, and hence reduce unnecessary request rejections.

Specifically, we built a Mooncake cluster with 8 prefill instances and 8 decoding instances and tested it using the tool&agent workload described in §7.2.1. To simulate overload scenarios, we increased the replay speed to 2×.

Table 4 shows Mooncake's performance under different strategies. With the baseline strategy, the system rejects 4,183 requests. In contrast, under the Early Rejection and Early Rejection based on Prediction strategies, Mooncake rejects 3,771 and 3,589 requests, respectively. This demonstrates that by rejecting requests early, Mooncake can avoid unnecessary prefill computations, thereby improving the effective utilization of system resources. Furthermore, by predicting the load of decoding instances, Mooncake can mitigate load fluctuations, increasing the request handling capacity.

7.7 Performance in Open-source LLM serving

7.7.1 SGLang Integration. We assess SGLang PD disaggregation across three scale clusters (NVIDIA A10, H20, H200) to demonstrate improvements. We use 2000 input and 100 output tokens for workload generation.

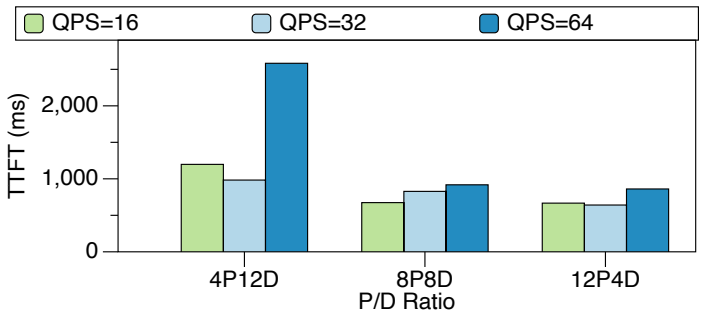


Fig. 24. The TTFT in SGLANG with different QPS.

We evaluated SGLang on two NVIDIA A10 servers to verify whether the results align with findings from the original Mooncake. By comparing the performance of a 1P1D configuration with that of two regular (non-disaggregated) instances, we observed that TBT is 7.23 ms under P/D disaggregation. It achieves approximately 30% lower TBT while maintaining comparable total throughput compared to the regular one (10.30 ms). This means that P/D disaggregation is effective in reducing TBT under similar throughput conditions.

Table 5. Improvements with Mooncake-based PD Disaggregation in SGLang.

Config	QPS	TTFT (ms)	Req/s	Output Token/s	Total Token/s	Tput Impr. (%)
SGLang-regular	1	1234	0.29	28.54	741.99	—
SGLang-PD 4P2D	1	1302	0.29	28.71	746.39	0.6%
SGLang-regular	4	4570	0.55	54.61	1419.77	—
SGLang-PD 4P2D	4	1564	0.98	98.15	2551.86	79.8%
SGLang-regular	8	9629	0.65	64.79	1684.61	—
SGLang-PD 4P2D	8	1969	1.77	176.57	4590.73	172%

As shown in Table 5, we adjust traffic stress with queries per second (QPS) from 1 to 8 in NVIDIA H20, comparing TTFT and TBT between regular and 4P2D config. For lightweight workloads, TTFT is similar for both P/D disaggregation and regular SGLang. With QPS at 8, P/D disaggregation reduces TTFT by nearly 5× for heavy workloads.

Open-source Large-scale Deployment. Mooncake has been deployed in a large-scale GPU cluster to evaluate its efficiency under DeepSeek-R1 [17]. The deployment environment utilizes 12 nodes, each with 8 NVIDIA H100 GPUs. SGLang leverages PD disaggregation based on Mooncake and other features such as expert parallelism (EP). Under 2000 token input sequences and P/D ratio is 3:9 configurations, it can achieve 52.3k input tokens/sec and 22.3k output tokens/sec per node for prefill and decode instances [36].

Figure 24 shows the TTFT increasing under different QPS and different PD ratios. The testbed includes 16 NVIDIA H200 servers. For the 4P12D configuration, the decline in TTFT can be attributed to the saturation of prefill stages. When the PD ratio is 8P8D and 12P4D, the TTFT increases only 39%/29% when QPS is from 16 to 64. It shows that KVCache-centric scheduling can significantly lower TTFT in open-source deployments.

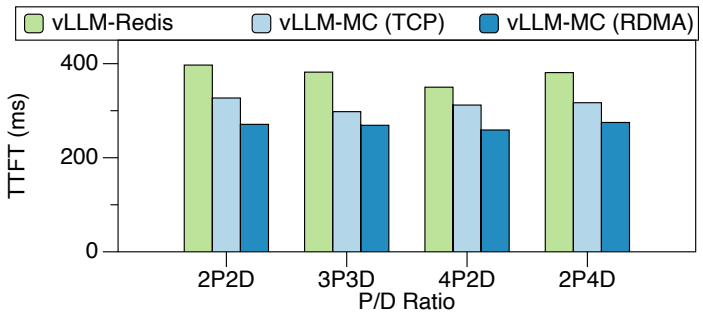


Fig. 25. The key metric's improvement after integrating Mooncake in vLLM.

7.7.2 vLLM Integration. The evaluation configuration employs the Qwen2.5-7B-Instruct-GPTQ-Int4 model on NVIDIA A10 GPUs. Figure 25 measures three KVCache store backends: Redis (vLLM-redis), Mooncake Store with TCP transport (vLLM-MC (TCP)) and Mooncake Store with RDMA transport (vLLM-MC (RDMA)), to support vLLM PD disaggregation under varying PD ratios. Tests simulate workloads with 200 requests, 1024-input tokens, 6-output tokens, and 2 QPS to analyze TTFT across different backends and PD ratios. Compared to vLLM-redis, vLLM-MC (TCP) and vLLM-MC (RDMA) achieve approximately 17%∼22%/26%∼33% reductions in TTFT, respectively.

8 Related Work

Significant efforts have been dedicated to enhancing the efficiency of LLM serving systems through scheduling, memory management, and resource disaggregation. Production-grade systems like FasterTransformer [9], TensorRT-LLM [11], and DeepSpeed Inference [2] are designed to significantly boost throughput. Orca [49] employs iteration-level scheduling to facilitate concurrent processing at various stages, while vLLM [23] leverages dynamic KVCache management to optimize memory. FlexGen [43], Sarathi-Serve [1], and FastServe [48] incorporate innovative scheduling and swapping strategies to distribute workloads effectively across limited hardware, often complementing each other's optimizations. Further optimizations [19, 39, 52] lead to the separation of prefill and decoding stages, leading to the disaggregated architecture of Mooncake. Our design of Mooncake builds on these developments, particularly drawing from the open-source community of vLLM, for which we are deeply appreciative.

Prefix caching is also widely adopted to enable the reuse of KVCache across multiple requests, reducing computational overhead in LLM inference systems [11, 23]. Prompt Cache [15] precomputes and stores frequently used text KVCache on inference servers, facilitating their reuse and significantly reducing inference latency. SGLang [51] leverages RadixAttention, which uses a least recently used (LRU) cache within a radix tree structure to efficiently enable automatic sharing across various reuse patterns.

Among these approaches, CachedAttention [14], a concurrent work with us, proposes a hierarchical KV caching system that utilizes cost-effective memory and storage media to accommodate KVCache for all requests. The architecture of Mooncake shares many design choices with CachedAttention. However, in long-context inference, the KVCache becomes extremely large, requiring high capacity and efficient data transfer along with KVCache-centric global scheduling. Additionally, Mooncake is not a standalone cache service; it incorporates both a memory-efficient cache storage mechanism and a cache-aware scheduling strategy, further improving prefix caching efficiency.

The benefits of a distributed KVCache pool depend on the cache hit rate, which increases as the per-token cache size decreases under a fixed capacity. Consequently, orthogonal techniques such as KVCache compression [18, 30, 31] and KVCache-friendly attention architectures [4, 28] can further enhance our approach.

9 Conclusion

This paper presents Mooncake, a KVCache-centric disaggregated architecture designed for efficiently serving LLMs, particularly in long-context scenarios. We discuss the necessity, challenges, and design choices involved in balancing the goal of maximizing overall effective throughput while meeting latency-related SLO requirements.

Acknowledgments

We would like to thank the anonymous reviewers and the journal editors for their insightful comments and feedback. The authors affiliated with Tsinghua University are all in the Department of Computer Science and Technology, Beijing National Research Center for Information Science and Technology (BNRist), Tsinghua University, China. This work is supported by National Key Research & Development Program of China (2022YFB4502004), Beijing Natural Science Foundation (L252014), Tsinghua University Initiative Scientific Research Program, and Approaching.AI Co., Ltd.

References

[1] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2024. Taming Throughput-Latency Tradeoff in LLM Inference with Sarathi-Serve. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 117–134.

[2] Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, et al. 2022. Deepspeed-inference: enabling efficient inference of transformer models at unprecedented scale. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15.

[3] Chenxin An, Shansan Gong, Ming Zhong, Xingjian Zhao, Mukai Li, Jun Zhang, Lingpeng Kong, and Xipeng Qiu. 2023. L-eval: Instituting standardized evaluation for long context language models. *arXiv preprint arXiv:2307.11088* (2023).

[4] William Brandon, Mayank Mishra, Aniruddha Nrusimha, Rameswar Panda, and Jonathan Ragan Kelly. 2024. Reducing Transformer Key-Value Cache Size with Cross-Layer Attention. *arXiv preprint arXiv:2405.12981* (2024).

[5] William Brandon, Aniruddha Nrusimha, Kevin Qian, Zachary Ankner, Tian Jin, Zhiye Song, and Jonathan Ragan-Kelley. 2023. Striped attention: Faster ring attention for causal transformers. *arXiv preprint arXiv:2311.09431* (2023).

[6] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[7] Alibaba Cloud. 2025. Elastic Compute Service:eRDMA. https://www.alibabacloud.com/help/en/ecs/user-guide/elastic-rdma-erdma/.

[8] LMDeploy Contributors. 2023. LMDeploy: A Toolkit for Compressing, Deploying, and Serving LLM. https://github.com/InternLM/lmdeploy.

[9] NVIDIA Corporation. 2019. FasterTransformer. https://github.com/NVIDIA/FasterTransformer.

[10] NVIDIA Corporation. 2020. NCCL. https://github.com/NVIDIA/nccl.

[11] NVIDIA Corporation. 2023. TensorRT-LLM. https://github.com/NVIDIA/TensorRT-LLM.

[12] DeepSeek. 2025. DeepEP. https://github.com/deepseek-ai/DeepEP.

[13] Jiarui Fang and Shangchun Zhao. 2024. Usp: A unified sequence parallelism approach for long context generative ai. *arXiv preprint arXiv:2405.07719* (2024).

[14] Bin Gao, Zhuomin He, Puru Sharma, Qingxuan Kang, Djordje Jevdjic, Junbo Deng, Xingkun Yang, Zhou Yu, and Pengfei Zuo. 2024. Cost-Efficient Large Language Model Serving for Multi-turn Conversations with CachedAttention. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. 111–126.

[15] In Gim, Guojun Chen, Seung-seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. 2024. Prompt cache: Modular attention reuse for low-latency inference. *Proceedings of Machine Learning and Systems* 6 (2024), 325–338.

[16] Google. 2024. Our next-generation model: Gemini 1.5. https://blog.google/technology/ai/google-gemini-next-generation-model-february-2024.

[17] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948* (2025).

[18] Coleman Hooper, Sehoon Kim, Hiva Mohammadzadeh, Michael W Mahoney, Yakun Sophia Shao, Kurt Keutzer, and Amir Gholami. 2024. Kvquant: Towards 10 million context length llm inference with kv cache quantization. *arXiv preprint arXiv:2401.18079* (2024).
[19] Cunchen Hu, Heyang Huang, Liangliang Xu, Xusheng Chen, Jiang Xu, Shuang Chen, Hao Feng, Chenxi Wang, Sa Wang, Yungang Bao, et al. 2024. Inference without Interference: Disaggregate LLM Inference for Mixed Downstream Workloads. *arXiv preprint arXiv:2401.11181* (2024).
[20] Sam Ade Jacobs, Masahiro Tanaka, Chengming Zhang, Minjia Zhang, Leon Song, Samyam Rajbhandari, and Yuxiong He. 2023. Deepspeed ulysses: System optimizations for enabling training of extreme long sequence transformer models. *arXiv preprint arXiv:2309.14509* (2023).
[21] Vijay Anand Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. 2023. Reducing activation recomputation in large transformer models. *Proceedings of Machine Learning and Systems* 5 (2023), 341–353.
[22] KVCache.ai. 2025. Mooncake: A KVCache-centric Disaggregated Architecture for LLM Serving. https://github.com/kvcache-ai/Mooncake.
[23] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 611–626.
[24] Dacheng Li, Rulin Shao, Anze Xie, Eric P Xing, Joseph E Gonzalez, Ion Stoica, Xuezhe Ma, and Hao Zhang. 2023. LightSeq:: Sequence Level Parallelism for Distributed Training of Long Context Transformers. In *Workshop on Advancing Neural Network Training: Computational Efficiency, Scalability, and Resource Optimization (WANT@ NeurIPS 2023)*.
[25] Jiaqi Li, Mengmeng Wang, Zilong Zheng, and Muhan Zhang. 2023. LooGLE: Can Long-Context Language Models Understand Long Contexts? *arXiv preprint arXiv:2311.04939* (2023).
[26] Shenggui Li, Fuzhao Xue, Chaitanya Baranwal, Yongbin Li, and Yang You. 2023. Sequence Parallelism: Long Sequence Training from System Perspective. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2391–2404.
[27] Zhuohan Li, Siyuan Zhuang, Shiyuan Guo, Danyang Zhuo, Hao Zhang, Dawn Song, and Ion Stoica. 2021. Terapipe: Token-level pipeline parallelism for training large-scale language models. In *International Conference on Machine Learning*. PMLR, 6543–6552.
[28] Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Dengr, Chong Ruan, Damai Dai, Daya Guo, et al. 2024. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. *arXiv preprint arXiv:2405.04434* (2024).
[29] Hao Liu, Matei Zaharia, and Pieter Abbeel. 2023. Ring Attention with Blockwise Transformers for Near-Infinite Context. In *NeurIPS 2023 Foundation Models for Decision Making Workshop*.
[30] Yuhan Liu, Hanchen Li, Yihua Cheng, Siddhant Ray, Yuyang Huang, Qizheng Zhang, Kuntai Du, Jiayi Yao, Shan Lu, Ganesh Ananthanarayanan, et al. 2024. Cachegen: Kv cache compression and streaming for fast large language model serving. In *Proceedings of the ACM SIGCOMM 2024 Conference*. 38–56.
[31] Zirui Liu, Jiayi Yuan, Hongye Jin, Shaochen Zhong, Zhaozhuo Xu, Vladimir Braverman, Beidi Chen, and Xia Hu. 2024. Kivi: A tuning-free asymmetric 2bit quantization for kv cache. *arXiv preprint arXiv:2402.02750* (2024).
[32] Moonshot AI. 2023. Kimi. https://kimi.moonshot.cn.
[33] MoonshotAI. 2025. Kimi K2. https://github.com/MoonshotAI/Kimi-K2.
[34] NVIDIA. 2022. NVIDIA H100 Tensor Core GPU Architecture. https://resources.nvidia.com/en-us-tensor-core.
[35] OpenAI. 2022. Introducing ChatGPT. https://openai.com/blog/chatgpt.
[36] LMSYS Org. 2025. Deploying DeepSeek with PD Disaggregation and Large-Scale Expert Parallelism on 96 H100 GPUs. https://lmsys.org/blog/2025-05-05-large-scale-ep/.
[37] Charles Packer, Vivian Fang, Shishir G Patil, Kevin Lin, Sarah Wooders, and Joseph E Gonzalez. 2023. Memgpt: Towards llms as operating systems. *arXiv preprint arXiv:2310.08560* (2023).
[38] Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. 2023. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th annual acm symposium on user interface software and technology*. 1–22.
[39] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. 2024. Splitwise: Efficient generative llm inference using phase splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 118–132.
[40] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.
[41] Facebook AI Research. 2017. Gloo. https://github.com/pytorch/gloo.
[42] Sharegpt teams. [n. d.]. https://sharegpt.com/.
[43] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. 2023. Flexgen: High-throughput generative inference of large language models with a single gpu. In *International Conference on Machine Learning*. PMLR, 31094–31116.

[44] Vikranth Srivatsa, Zijian He, Reyna Abhyankar, Dongming Li, and Yiying Zhang. 2024. Preble: Efficient Distributed Prompt Scheduling for LLM Serving. (2024).
[45] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).
[46] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).
[47] Bingyang Wu, Shengyu Liu, Yinmin Zhong, Peng Sun, Xuanzhe Liu, and Xin Jin. 2024. Loongserve: Efficiently serving long-context large language models with elastic sequence parallelism. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*. 640–654.
[48] Bingyang Wu, Yinmin Zhong, Zili Zhang, Gang Huang, Xuanzhe Liu, and Xin Jin. 2023. Fast distributed inference serving for large language models. *arXiv preprint arXiv:2305.05920* (2023).
[49] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A distributed serving system for Transformer-Based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 521–538.
[50] Yazhuo Zhang, Juncheng Yang, Yao Yue, Ymir Vigfusson, and KV Rashmi. 2024. SIEVE is Simpler than LRU: an Efficient Turn-Key Eviction Algorithm for Web Caches. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 1229–1246.
[51] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Jeff Huang, Chuyue Sun, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. 2023. Efficiently programming large language models using sglang. *arXiv preprint arXiv:2312.07104* (2023).
[52] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. 2024. DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 193–210.

A Open-source Request Trace Dataset

A.1 Overview

To facilitate further research on LLM serving, we have compiled and publicly released the trace dataset used in our experiments. This dataset comprises three JSONL files: `conversation_trace.jsonl`, `toolagent_trace.jsonl`, and `synthetic_trace.jsonl`, corresponding to the *conversation*, *tool&agent*, and *synthetic* workloads, respectively.

The *conversation* and *tool&agent* traces were obtained by sampling one hour of online request data from different clusters, respectively. To preserve caching relationships between requests, we prioritized collecting requests within the same user session. The *synthetic* trace was constructed from a combination of publicly available datasets: ShareGPT [42], Leval [3], and LooGLE [25]. We generated timestamps according to a Poisson process and ensured that the sequential order of requests within the same multi-turn conversation remained intact. For statistical details about the traces, please refer to the main text.

Each data entry in the dataset includes the following fields: *timestamp*, *input_length*, *output_length*, and *hash_ids*. We have included remapped block hashes, which are particularly useful for analyzing and implementing KVCache reuse policies. To the best of our knowledge, this is the first open-source dataset that can be used for real-world KVCache reuse analysis.

A.2 Data Details

Listing 2. Request samples.

```
{
    "timestamp": 27000,
    "input_length": 6955,
    "output_length": 52,
    "hash_ids": [46, 47, 48, 49, 50, 51, 52,
                 53, 54, 55, 56, 57, 2111, 2112]
```

```
}
{
    "timestamp": 30000,
    "input_length": 6472,
    "output_length": 26,
    "hash_ids": [46, 47, 48, 49, 50, 51, 52,
                 53, 54, 55, 56, 57, 2124]
}
```

Listing 2 presents two samples from our trace dataset. To protect our customers' privacy, we applied several mechanisms to remove user-related information while preserving the dataset's utility for simulated evaluation. The meanings of the fields are explained below.

Timestamp. The *timestamp* field indicates the relative arrival times of requests, ranging from 0 to 3,600,000, in milliseconds.

Input & Output Length. For privacy protection, our trace does not include actual text or tokens. Instead, it uses *input_length* and *output_length*, representing the number of input and output tokens, similar to Splitwise [39].

Hash ID. The *hash_ids* field describes prefix caching relationships. It is generated by hashing token blocks (with a block size of 512) into prefix hash values that include both the current and all preceding blocks. The resulting hash values are then mapped to globally unique IDs. Identical hash IDs indicate that a block of tokens, along with preceding tokens, is the same, thus allowing reuse within the corresponding KVCache. For example, in the provided samples, the first 12 hash IDs are identical, indicating they can share prefix caching for the first 12×512=6,144 tokens.

Received 28 July 2025; revised 28 July 2025; accepted 23 October 2025