THE ADVANCED COMPUTING
SYSTEMS ASSOCIATION

Symphony: Enabling Compute-Memory Disaggregation in LLM Serving Systems

Saurabh Agarwal and Bodun Hu, *UT-Austin;* Anyong Mao, *UW-Madison;*
Aditya Akella, *UT-Austin;* Shivaram Venkataraman, *UW-Madison*

https://www.usenix.org/conference/nsdi26/presentation/agarwal

This paper is included in the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation.

May 4–6, 2026 • Renton, WA, USA

ISBN 978-1-939133-54-0

Open access to the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation is sponsored by

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

ARTIFACT EVALUATED USENIX AVAILABLE

SYMPHONY: Enabling Compute-Memory Disaggregation in LLM Serving Systems

Saurabh Agarwal *UT-Austin*
Bodun Hu *UT-Austin*
Anyong Mao *UW-Madison*
Aditya Akella *UT-Austin*
Shivaram Venkataraman *UW-Madison*

Abstract

Large Language Models (LLMs) power AI applications such as chatbots and agents, which maintain conversational state across multiple turns. Serving these workloads is inherently stateful: each request generates a KV cache storing token-level state. Existing systems either recompute caches or offload them to host memory—both approaches incur high latency, cause load imbalance, and limit scalability. We present SYMPHONY, a disaggregated memory management layer that decouples compute from KV cache storage while meeting strict latency requirements. To enable disaggregation, SYMPHONY employs advisory requests—prefetching hints derived from user interactions or workload structure—to move caches off the critical path and enable fine-grained, request-level load balancing. Since these predictive signals are often unreliable, SYMPHONY introduces two key techniques: priority-based KV cache management, which allocates memory based on neural network structure and request priority, and cooperative memory management, which dynamically coordinates GPU memory with the serving framework. Evaluations on LLaMA models with ShareGPT and BurstGPT workloads show that SYMPHONY reduces end-to-end latency by 2.4× over vLLM and serves 4× more requests with minimal latency increase.

1 Introduction

Memory is central to LLM-powered applications such as chatbots and agents, which must retain conversational state across turns. Due to their auto-regressive nature, serving these workloads is inherently stateful: each request produces a KV cache storing token-level state for future generations. While prior work [1, 2, 9, 17, 26, 36, 37, 54] has focused on single-turn requests, multi-turn workloads like chatbots and LLM Agents must persist large caches—e.g., 5GB for LLaMA-3.1-70B at 32K context (INT8)—across unpredictable gaps between requests. The combination of large cache sizes and unbounded retention times makes storage in GPU memory infeasible.

To serve multi-turn LLM workloads, prior systems follow two strategies: (i) discard-and-recompute, and (ii) offload. Systems such as vLLM [17] and TensorRT-LLM [25] discard all KV cache state at the end of a request, recomputing it for the entire session when a new request arrives. Recent work [1, 9, 54] instead offloads KV caches to host memory or disk, reloading them when the session resumes. Both approaches have fundamental drawbacks: recomputation is redundant and wasteful (in ShareGPT, over 99% of tokens must be recomputed), while offloading incurs high latency from large state transfers and enforces session stickiness, requiring all requests in a session to be routed to the same node. This session-level load balancing causes severe imbalance—e.g., one node may see 3.1× the median load (Figure 1)—leading to significant latency degradation (Figure 2).

The challenges in managing KV caches for multi-turn LLM workloads mirror long-standing problems in stateful workloads within data warehouses and cloud computing. In those domains, the tight coupling of state and compute initially limited scalability and efficiency. Breakthroughs such as disaggregated cloud databases (e.g., Snowflake, AuroraDB [6, 44]) and serverless paradigms (e.g., Lambda functions [52]) addressed this by decoupling compute from state, enabling fine-grained elasticity, resource independence, and cost efficiency at unprecedented scale. We argue that serving multi-turn LLM workloads demands a similar shift to *compute and memory disaggregation*. By decoupling compute from KV cache storage, a disaggregated design can provide (i) fine-grained request-level load balancing, (ii) elimination of wasted recomputation, and (iii) independent scaling of compute and memory.

Realizing such a disaggregated memory layer for LLM serving introduces a fundamental challenge. Traditional compute–memory disaggregation often suffers from high latency due to data movement—manifesting as cold starts in Lambda functions or sluggish query performance in cloud data warehouses. Such overheads are unacceptable in user-facing multi-turn LLM applications [5], where strict latency budgets are non-negotiable. Thus, the core requirement is clear: *decouple compute and state without sacrificing latency*.

We argue that the key to enabling compute–state decoupling in multi-turn LLM workloads without incurring latency

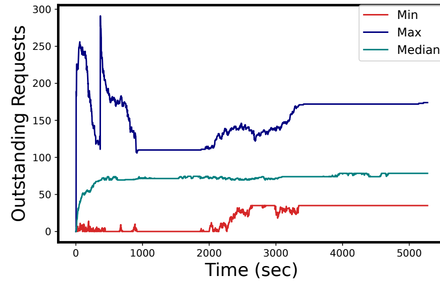


Figure 1: **Load imbalance in stateful LLM Serving**: We serve 1024 concurrent users with 8 GPUs with Infercept, a popular system that offloads KV caches to host memory. We observe that the difference between the most loaded GPU and the least loaded GPU can be 300 requests. This indicates that stateful LLM serving can lead to high load imbalance.

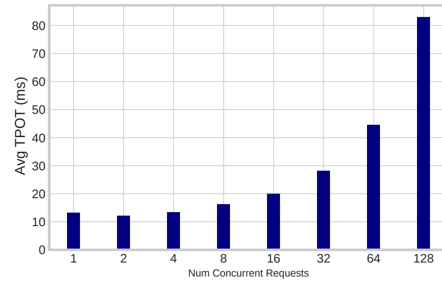


Figure 2: **Scaling Behavior**: We plot the average Time Per Output Token (TPOT) when serving LLaMA-3-8B on one A100. We observe that the average TPOT drastically increases as the number of concurrent requests increases on the GPU. Processing 8 concurrent requests vs 32 concurrent requests can lead to almost double the latency.

penalties is *prefetching*. Our approach builds on a central observation: the interactive and structured nature of many LLM workloads provides *hints about future request patterns*. We term these signals *advisory requests*, and show that they enable near-ideal state management by allowing KV caches to be prefetched off the critical path. An example of an advisory request is the signal generated by a user's initial keystroke when interacting with a chatbot, indicating which session is likely to issue an inference soon. Analysis of the ShareGPT dataset [27] shows that such requests arrive, on average, 11.3seconds before the corresponding inference—providing ample time to prefetch the KV cache and hide data movement latency. Similarly, agent workloads [30], where multiple LLMs interact, naturally produce advisory signals. Given a known call graph and offline profiling of each LLM's processing time, downstream request arrivals can be predicted. During execution, while an upstream agent processes a prompt, an advisory request can be sent to downstream agents with session context and estimated arrival time. Evaluating MetaGPT [14] on 4 A100s, we find that such advisory requests precede the actual inference by an average of 5.8 seconds.

Effectively leveraging advisory requests for high-performance serving is challenging. For interactive workloads (e.g., chatbots), advisory requests indicate only the potential arrival of a request—they provide no information on prompt length or expected generation length, making GPU memory allocation difficult. Moreover, the arrival order across sessions is uncertain; an early advisory request does not guarantee early inference execution. Finally, some advisory requests may be spurious, with no corresponding inference request. These uncertainties in timing and memory requirements limit precise KV cache prefetching and memory management.

To handle unpredictable request arrivals and the memory demands of each request, we introduce two key strategies: (1) *priority-based KV cache management*, which leverages neural network structure to prioritize caches needed sooner under memory pressure; and (2) *cooperative memory management*, where the LLM serving framework (e.g., vLLM, TensorRT) closely works with the memory layer to dynamically allocate GPU memory based on active requests. This approach maximizes memory efficiency by prioritizing current requests while prefetching caches for upcoming ones. To enable compute–memory disaggregation for multi-turn LLMs, we present SYMPHONY, a disaggregated memory management layer. Leveraging advisory requests, SYMPHONY offloads KV caches from accelerator memory without incurring runtime overhead. It comprises two key components: a global scheduler and a node manager. The global scheduler processes advisory requests, issues directives to the appropriate memory-managing node, and schedules these requests across the inference cluster. The node manager works with the serving framework to perform cooperative memory management—prioritizing GPU memory for active requests while using available capacity to prefetch KV caches.

We demonstrate that SYMPHONY, leveraging caching and fine-grained load balancing, significantly improves both latency and throughput. We evaluate SYMPHONY on the latest LLaMA models using real conversational datasets: ShareGPT [27] and BurstGPT [47]. SYMPHONY reduces average end-to-end latency by 2.4× compared to vLLM and can serve 4× more requests with only a 1.3% increase in average latency. We also compare against Lumnix [37], which dynamically migrates KV caches, and Infercept [1], which offloads KV caches to host memory to avoid redundant computation. Even against a strong combined baseline of Lumnix + Infercept, SYMPHONY achieves speedups up to 1.6×. Against CacheGen [21], which streams KV caches with lossy compression, SYMPHONY shows a ∼ 35% improvement with aid of advisory requests.

2 Background and Motivation

We start with a brief overview of LLM Inference and KV caches.

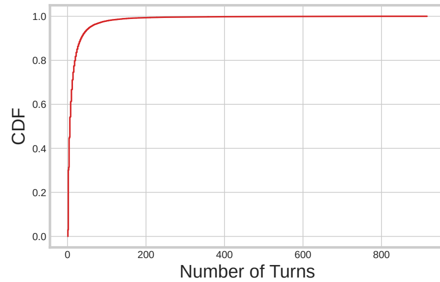


Figure 3: **CDF of the number of turns in a chatbot:** Distribution of the number of turns in the ShareGPT dataset. We observe that 73.4% of conversations are multi-turn conversations.

2.1 LLM Inference

The self-attention operator maintains Key, Value, and Query vectors [43]. To generate the next token, it requires the current query and all prior keys and values, which inference systems cache in the KV cache [29]. As model size and context length grow, this cache has become a major bottleneck, for example, LLAMA-3.1-70B model with 128K context requires 20GB (INT8). LLM inference consists of two phases [2]: *prefill*, which processes the prompt and builds the KV cache (compute-bound), and *decode*, which generates tokens iteratively using the cache (memory-bound) [15, 19, 35, 37]. Prior work has explored hardware utilization and scheduling strategies, including disaggregating prefill and decode [36, 53], latency-aware scheduling [2], fairness-oriented schedulers [35], and predictive batching [15, 37]. However, all works propose optimizations that assume that requests are "stateless", more precisely that each request is independent of *prior* state – an assumption we show does not hold for most LLM workloads.

2.2 State Management for LLM workloads

We first provide an overview of state requirements for popular LLM workloads.

State requirement Popular LLM applications such as chatbots and coding assistants are inherently multi-turn. As usage trends shift toward sustained interactions, the dominant serving workloads increasingly consist of multi-request sessions. Within each session, requests depend on the *KV caches* generated by earlier turns to preserve context and continuity. To ground this discussion, we examine two representative applications and their request patterns.

Chatbots. One of the primary applications of LLMs is in conversational agents. In this setting, users interact with the model through prompts and may continue the dialogue with follow-up questions or inputs. To preserve context, each new interaction requires access to all prior inputs and responses. Figure 3 presents the cumulative distribution function (CDF) of conversation lengths in the ShareGPT dataset. Notably, 73.4% of conversations are multi-turn, with some extending beyond 400 turns.

Agents. Another application of LLMs lies in agent-based workloads, where LLMs interact with each other to achieve a high-level objective. For instance, ChatDev [31] and MetaGPT [14] introduce agents designed to tackle software development tasks. In these multi-agent interactions, each agent requires access to prior prompts and the corresponding KV caches from all previous exchanges. This continuity is essential for maintaining coherent, context-aware responses across the agent workflow. For example, in MetaGPT, when the software developer agent receives feedback from the testing agent, it requires access to the state associated with previously generated code to refine and improve that code.

Existing approaches There are three mechanisms primarily in use when deploying LLM workloads.

Retain. To avoid recomputation, some systems [14] retain the previous KV cache in GPU memory. However, this quickly leads to the exhaustion of the GPU's high-bandwidth memory (HBM). For example, when performing inference on a batch of 54 requests, LLaMa-3-8B with we observed that the GPU HBM became saturated.

Recompute. Existing systems [2, 5, 17, 25] treat each request independently, recomputing the KV cache from scratch and causing redundant computation. As shown in Figure 5, in the ShareGPT dataset, conversations exceeding three turns wastefully recompute over 50% of pre-filled tokens.

Swap. Some recent systems [1, 9] swap KV caches between GPU and host memory, migrating them back to the GPU when the next session request arrives. For example, on LLAMA-7B, we observe that this reduces prefill time by 4.9× and decode time by 1.68×. However, such swapping has two fundamental limitations: (i) offloading from GPU HBM to main memory makes workloads *sticky*, as all future requests in a session must be scheduled on the same machine; and (ii) Caches when transferred to slower storage (e.g., disk or remote) incur prohibitive overheads at serving time. As shown in Figure 1, enforcing stateful machine assignments creates load imbalances that reduce throughput (Figure 2).

Several recent libraries, such as Mooncake [32], LMCache [21], and Dynamo [24], have been developed to support KV cache offloading from GPU HBM. While these systems provide robust offloading mechanisms, they often incur significant overhead during cache retrieval. In contrast, SYMPHONY specifically optimizes the KV cache loading path to minimize retrieval latency.

One approach to mitigating load imbalance is migrating KV caches across GPUs. However, as shown in Figure 6, this increases request serving time. The migration cost—driven by cross-node network transfers—reduces throughput, as requests must wait for cache movement to complete.

Further, offloading to main memory is constrained by its limited capacity. For example, when serving LLAMA -3.1-405B, a system with 256GB of main memory can store

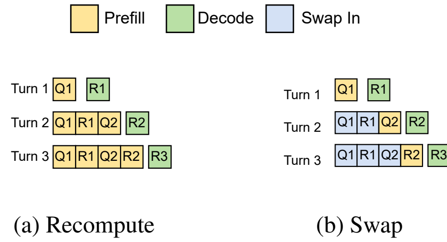


Figure 4: **Comparing recompute and swap:** The above figure shows the difference between the recompute and swap approaches. Swapping can dramatically reduce the prefill time, if the time to load the KV cache is negligible.

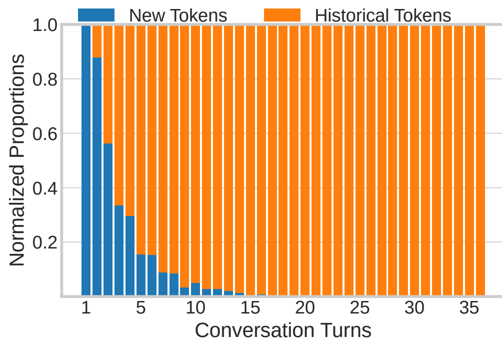


Figure 5: **Wasted Tokens:** On the ShareGPT dataset, when recomputing, we observe that after three conversation rounds, a majority of the tokens processed will be redundant.

only about 266K tokens. Given that the average ShareGPT session is 2.2K tokens, a node could hold at most 108 concurrent sessions if relying solely on host memory—highlighting the need to leverage slower, higher-capacity storage tiers.

2.3 Disaggregated memory layer for LLM workloads

Decoupling memory and compute is wide-spread in modern applications, including databases [45], ephemeral compute [13], and big data analytics [12], allowing each resource to scale independently rather than as a fixed bundle. Applying this approach to LLM inference is promising: disaggregation (i) breaks the dependency between accelerator capacity and concurrent sessions, enabling independent scaling of compute and storage, (ii) makes serving effectively stateless, supporting fine-grained, request-level scheduling, and (iii) eliminates recomputation by retaining virtually unlimited state in memory, allowing LLMs to handle large working sets efficiently.

However, disaggregation incurs significant latency penalties due to data movement, which is unacceptable for user-facing LLM workloads such as chatbots. To make disaggregation usable, this latency must be hidden when transferring data from remote memory to accelerator memory. SYMPHONY is a disaggregated memory layer that masks data-movement latency and enables low-latency serving of LLM workloads.

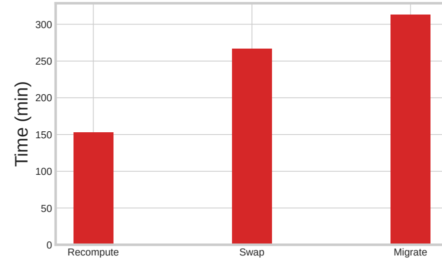


Figure 6: **Recompute and Swap**: In the above figure, we ran recompute (vllm), InferCept(Swap), and InferCept with KV cache migration. We observe that in a distributed setting (serving on 8 GPUs), recomputing leads to the highest throughput (lowest time)

3 SYMPHONY Design Overview

3.1 Key Insight- Availability of Useful Hints

To hide the latency of decoupling memory and compute, we leverage *hints* that enable effective prefetching. We refer to these hints as advisory requests. Next, we show how such hints can be derived for two representative target workloads.
Chatbot: In a chatbot workload, the typical workflow is that a user submits a query to which the chatbot responds, and the user reads the response before typing a follow-up. This introduces a natural delay of several seconds between requests due to comprehension and typing. Since a request must first be typed into the chatbox, minor interface modifications can trigger an advisory signal *as soon as the user begins typing*. This signal can mostly reliably indicate that a new request is imminent. Figure 7 shows an example of an advisory request in this context.
Agent Inference: Agent workloads are represented as a call graph that captures the invocation order of each agent. The workflow can be profiled to determine the sequence in which agents are invoked. Figure 8 shows a call graph of an agentic workflow. At runtime, when a given agent executes, we immediately issue an advisory request for all its *next hop* downstream agents identified in the profiled call graph. For example, as soon as the software engineering agent runs, advisory requests are sent to the program manager and the QA engineer responsible for regression testing 8.

While this work primarily targets chatbot and agent-serving workloads, the proposed hint-based mechanism is broadly applicable to other user-facing workflows, such as Retrieval-Augmented Generation (RAG) and tool-calling models. For example, in the case of RAG, a query that enters the retrieval stage will typically proceed to the generation stage (LLM inference) shortly thereafter. By leveraging explicit hooks or lightweight profiling, a serving system can infer request arrival patterns to optimize memory management. While advisory hints may be unavailable for bulk processing tasks, these workloads are typically not latency-sensitive and are not the focus of our work.

3.2 Limitation of Advisory Requests

At first glance, advisory requests appear to be simple signals for prefetching KV caches from lower to higher-tier memory (e.g., SSDs to DRAM) and across nodes over the network to improve efficiency. In practice, however, leveraging these requests is challenging due to missing key information and limited reliability, as explained in the following sections.

Information gaps in advisory requests Advisory requests only hint at likely arrivals in a multi-turn session but lack key details related to ordering and memory requirements.

No timing and partial ordering information: In chatbot scenarios, advisory requests indicate with high probability that a request tied to a session may arrive, yet they provide no guarantees about the timing of its arrival or its order relative to other ongoing sessions. In agent workloads, ordering may be partially known. For instance, a request processed by one agent will subsequently reach the next agent in sequence, but the exact arrival time remains uncertain. Moreover, advisory requests convey no information about memory requirements: neither the input size of the anticipated request nor the number of tokens to be generated is known, making it impossible to predict the memory footprint in advance. To illustrate the impact of missing information, consider a high-load LLM serving scenario. An advisory request for a session A prompts the system to move its key–value cache into GPU HBM, fully occupying the available memory. Shortly thereafter, another advisory request arrives for another session B, but the GPU memory is already saturated. However, it is entirely possible that a request from session B comes before the next request from session A comes. Without additional details, specifically, which inference request will arrive first, it is impossible to determine which session to prioritize.

Imperfect estimate of memory requirement: The GPU memory required to serve an LLM request primarily depends on the number of prompt and generated tokens. Estimating the memory requirements for an LLM request is challenging [15, 17], as the number of tokens generated by a request is not known a priori. This uncertainty makes it difficult to accurately determine the amount of free GPU memory that will be available Consider the following example: a GPU is serving an LLM request associated with Session ID: 1. While this request is being processed, an advisory request arrives for Session ID: 2. Based on the current free GPU memory, the system decides to prefetch the KV cache for Session ID: 2 onto the GPU in anticipation of the actual inference request. However, as the inference for Session ID: 1 continues, the size of its KV cache grows, eventually consuming all available GPU memory. This situation leaves no memory for the decoding process of the ongoing LLM request, causing a bottleneck. In this scenario, the optimal decision would have been to avoid prefetching the KV cache for Session ID: 2. However, the information needed to take this action is not available a priori.

Unreliability of advisory requests. Advisory requests can also be unpredictable, and in some cases may even be spurious. For example, in chatbot scenarios, an advisory may be triggered when a user begins typing, yet the user may abandon the input without ever submitting a request. Such behavior results in spurious advisories that consume resources without yielding useful work.

To effectively use advisory requests SYMPHONY needs to build mechanisms to overcome these challenges. Next, we describe SYMPHONY and its main components before discussing the main mechanisms for overcoming the challenges above.

3.3 Design Overview

SYMPHONY targets a distributed, disaggregated setup consisting of several memory tiers ranging from GPU memory, host memory, disk, disk storage of other nodes in the cluster,and blob storage. Figure 9 shows a schematic of our setup and our proposed components.

SYMPHONY comprises two key components: (1) a scheduler that orchestrates the movement of KV cache between compute and storage nodes in response to advisory requests, and (2) a node-level controller that selectively prefetches required KV cache segments. Both components operate transparently with respect to the LLM inference engine. By fully decoupling KV cache management from LLM inference execution, SYMPHONY achieves compute-memory disaggregation.

SYMPHONY *scheduler.* As illustrated in Figure 9, the SYMPHONY scheduler functions as a top-level scheduler and provides three public-facing interfaces (i) Inference interface: An interface for accepting LLM inference requests, and (ii) Advisory interface: A second for accepting advisory requests. (iii) Invalidation interface: A third for invalidating a previous advisory request which was deemed spurious. The *scheduler* has a full view of the state associated with SYMPHONY. It is responsible for co-ordinating with different *node-level controllers* to make decisions about where the state should move. Upon receiving an advisory request, the global scheduler performs scheduling. Based on the scheduling policy, it augments the advisory request with information about the existing KV cache location (i.e., the node storing the cache) and forwards it to the SYMPHONY node manager based on the scheduling decision. Additionally, the scheduler updates its internal state to reflect the new KV cache location. When the actual inference request arrives, the SYMPHONY scheduler routes it to the node identified during the processing of the corresponding advisory request.

SYMPHONY employs a straightforward load-balancing policy by default which distributes requests evenly across nodes. However, as discussed in Section 3.6, SYMPHONY can support a variety of customizable policies tailored to different workloads and system configurations.

SYMPHONY *Node Manager.* The SYMPHONY node man-

```
{"session_id": UUID,
 "model_id": llama-3.1-8b,
 "expected_arrival": None,
 "ordered": False,
 "priority": 1}
```

Figure 7: **Advisory request for chatbot:** The above shows the format of advisory request for chatbot. Because neither arrival time nor message order can be guaranteed, requests are issued without timing or sequencing assumptions.

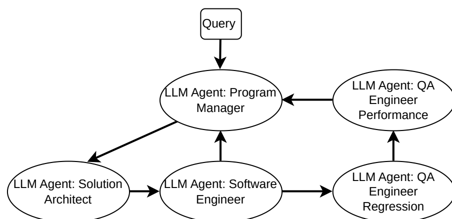


Figure 8: **LLM Agent Call Graph:** We depict an example call graph for a software development agent. Agentic call graphs can have different forms.

ager resides on each node and oversees SYMPHONY 's hierarchical memory system, which stores KV caches. It exposes an interface to handle four types of requests: (i) advisory requests forwarded by the SYMPHONY's global controller, (ii) LLM inference requests, (iii) KV cache fetch requests from other SYMPHONY node managers, and (iv) Invalidation requests forwarded by SYMPHONY's global controller.

An advisory request received from the controller contains two pieces of information: the session ID of the expected inference request and the current location of the associated KV cache. Upon receiving an advisory request, the node manager verifies the cache's location. If the cache doesn't reside on the current node, the node manager issues a call to retrieve it from the provided location. If the cache is locally available, the node manager, subject to memory constraints, moves it to the fastest memory tier possible

When an LLM inference request arrives, it is routed to the underlying serving library, such as vLLM or TensorRT-LLM. Section 3.4 discusses how the SYMPHONY node manager interfaces with these serving engines (vLLM and TensorRT-LLM) to cooperatively manage GPU high-bandwidth memory (HBM).

3.4 Mitigating the limitations of Advisory Requests

To handle limitations of advisory requests (Section 3.2), we propose three mechanisms.

Priority-Based KV Cache Management. Given the limited information from advisory requests, directly prioritizing each session's KV cache is infeasible. Instead, under memory pressure, SYMPHONY makes eviction decisions within a request—prioritizing caches on the critical path and evicting those whose latency can be hidden via forward-pass overlap. For example, under memory pressure, SYMPHONY will remove the last layer of KV cache associated with the least recently used request.

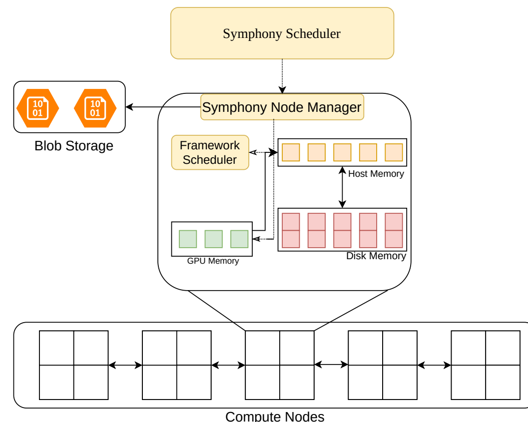


Figure 9: **SYMPHONY Design**: The above figure shows a schematic of SYMPHONY. SYMPHONY scheduler is a high-level scheduler making scheduling decisions with the aid of advisory requests. SYMPHONY node manager manages the memory and interacts with a framework-level scheduler.

This scheme leverages the layerwise structure of LLMs: if KV caches for the early layers are available, inference can begin while later-layer caches are being prefetched. Caches are thus prioritized by layer, with lower-layer blocks assigned higher priority since they are needed first. This ensures critical caches are available promptly, reducing inference delays. In particular, if multiple advisory requests arrive for different sessions, the scheme places lower-layer caches from both sessions in HBM first, deferring higher layers until the corresponding request arrives.

Cooperative Memory Scheduling To address challenges associated with imperfect estimates of memory requirements, SYMPHONY's node manager and the framework-level scheduler jointly manage GPU memory, dynamically *releasing* capacity based on HBM demand. We call this *cooperative memory management*. It ensures smooth operation under constrained resources.

In cooperative memory management, the SYMPHONY node manager opportunistically fills free GPU memory with KV caches. For example, if an advisory request arrives, and there is GPU memory available, SYMPHONY node manager will move the associated KV cache to the GPU HBM. Under memory pressure, however, the serving library can overwrite this space by purging caches from HBM without extra transfer cost, since copies already reside in host memory. Purges are prioritized as follows: later-layer caches are discarded first, followed by KV caches of the least recently used request. We apply cooperative management only to GPU HBM, while always maintaining a complete copy of the KV cache in the slowest tier (disk or a remote blob storage). This guarantees

persistence, allowing safe on-demand evictions from faster tiers. A background thread continuously writes newly generated caches to keep lower-tier copies up to date. Data in the lowest tier (disk/blob storage) is evicted once it exceeds a user-defined expiry threshold. Cooperative memory management mitigates the impact of information gaps inherent in advisory requests.

Finally, to hide the latency of loading the KV cache from main memory when the full KV cache does not fit in the GPU, we perform Layer-wise Reads and Writes.

Layer-wise Reads and writes When utilizing storage mediums such as main memory, which exhibit significantly higher access latencies, reading and writing KV caches can introduce substantial overhead. This slow access latency can block inference operations, resulting in performance slowdowns.

To mitigate the impact of these delays, we adopt *layer-wise asynchronous reading and writing* techniques inspired by prior work [9]. Deep neural networks (DNNs) process layer by layer, requiring access to KV caches only for the current layer being computed. We allow the system to load upcoming needed caches and store already processed one KV caches in parallel with inference execution for a layer. By enabling reads and computation to occur in parallel, we reduce the bottleneck associated with slow reads and writes to host memory or slower media.

An Example of SYMPHONY's optimizations To demonstrate optimizations in SYMPHONY's, we provide the following example.

Setup: Assume we have two nodes, Node-1 and Node-2. Node-1 is serving two requests, while Node-2 is handling four requests. When an advisory request arrives for Session ID: 1, the SYMPHONY scheduler assigns it to Node-1 to balance the request load. However, the KV cache for Session ID: 1 resides on Node-2. The SYMPHONY node manager at Node-1 requests the KV cache from Node-2. We explore three cases to understand how SYMPHONY handles different scenarios:

*C*ase 1: High GPU Memory Availability. In this scenario, Node-1 has abundant GPU HBM available. Before the inference process begins, the SYMPHONY node manager moves the KV cache for Session ID: 1 to the GPU, host memory, and disk, ensuring quick access when memory resources are not constrained.

Case 2: High GPU Memory Availability with Increasing Memory Pressure. Here, the SYMPHONY node manager initially moves the KV cache for Session ID: 1 to the GPU, host memory, and disk. As memory pressure increases and the serving framework requires more GPU memory, cooperative scheduling is applied. Based on the priority of KV cache blocks, evictions are performed to free up memory. When the inference request for Session ID: 1 arrives, the layerwise asynchronous read mechanism incrementally reloads the required data from host memory to the GPU, minimizing delays and ensuring efficient memory usage.

```javascript
const inputField = document.getElementById('ChatTextField');
inputField.addEventListener('input', function(event) {
  // Call Symphony advisory request when a text is typed.
  symphony.advisory_request(event.target.session)
});
```

Figure 10: **Setting up advisory request for Chatbot:** The code above shows how to incorporate an advisory request into a chatbot using Javascript.

Case 3: No GPU Memory Available. In this case, the SYMPHONY node manager moves the KV cache for Session ID: 1 to host memory and disk, bypassing the GPU due to a lack of available memory. During inference, asynchronous layerwise reads are employed to hide latency by gradually fetching the required data from host memory to the GPU, ensuring smooth execution despite the memory constraints.

3.5 Serving with SYMPHONY

Next, we discuss how to integrate SYMPHONY when serving popular workloads.

Chatbots. Typically, chatbots are implemented as web applications using JavaScript. Integrating advisory requests into chatbots requires only three lines of JavaScript to trigger an API call when the chatbox is activated (Figure 10). This code captures the activation or focus event (of the text box), packages metadata such as the session identifier, and sends it to SYMPHONY.

Agent Serving. When serving agents, we first perform a profiling step over a small representative workload to capture the agent workflow graph. In SYMPHONY, we then maintain a lookup table for each agent, recording its potential subsequent agents that may be invoked. For example, in the workflow shown in Figure 8, the *Software Engineer agent* may next call either the *Program Manager* or the *QA Engineer*. At the time of inference, we send advisory request indicating both agents, leading to the fetching of the state associated with both of them. At inference time, we issue an advisory request for both potential downstream agents, triggering the fetching of state for each. Inevitably, one of these requests may turn out to be a false positive. However, SYMPHONY 's optimizations efficiently handle such spurious advisories with negligible impact on throughput (Figure 20).

3.6 Discussion

Additional scheduling policies The SYMPHONY scheduler exposes a flexible interface that allows administrators to build custom policies atop request-level scheduling. To illustrate, we implement request-level prioritization, a common requirement in LLM serving—for example, giving paid users higher priority than free-tier users. A typical implementation at high load preempts lower-tier requests on the node where the prior key–value cache resides. However, this approach can cause significant slowdowns if multiple high-priority requests arrive on the same node (due to workload stickiness), resulting in poor load balancing. Moreover, prioritizing one request often

degrades the latency of concurrent non-priority requests. With SYMPHONY, high-priority requests can be migrated evenly across nodes using advisory requests, while low-priority requests are selectively paused only when their execution would impact latency. The request-prioritization scheduler not only prioritizes resource access for high-priority requests but also balances them across nodes, thereby improving throughput. Crucially, SYMPHONY 's disaggregated memory layer enables such scheduling without concern for the physical location of key–value caches. We evaluate this in Sec 4.5.

Security implication of advisory requests. Advisory requests only signal that a user is typing in the chatbot's text box; they do not reveal the content being typed. This is analogous to the "User X is typing" notifications seen in messaging apps like Slack or Discord. Such information about user activity does not introduce additional security vulnerabilities.

Accuracy of advisory requests and dealing with spurious ones. Advisory requests provide 100% true positive rate, i.e., no actual request from a chatbot can arrive without generating an advisory request. However, there can be false positives, i.e., spurious advisory requests can arrive for prompts for which there are no inference requests. To overcome spurious advisory requests, Symphony utilizes priority-based KV cache management and cooperative memory scheduling.

Suppose we receive a spurious advisory request; the KV cache corresponding to the request will get prefetched. However, as memory pressure increases (due to ongoing inference requests or more requests arriving), priority-based KV cache management kicks in, evicting portions of the KV cache for all prefetched KV caches (including the spurious ones). As more requests arrive for inference, cooperative memory scheduling takes over and evicts the oldest unused prefetched KV cache, which is highly likely to correspond to a spurious request. Thus, prefetches due to spurious advisory requests will get evicted soon under resource pressure. Further, SYMPHONY provides an invalidation API, which the application can call to indicate if a spurious advisory request was sent. We acknowledge that in certain cases there may be false negatives, i.e., no advisory request arrives for a call due to network drop, etc. In such cases, SYMPHONY falls back to best-effort serving.

Limitations of SYMPHONY. SYMPHONY assumes that advisory requests arrive early enough to allow sufficient time for KV cache migrations. However, in extreme cases where there is insufficient time between the advisory request (or there was no advisory request) and the associated inference request, performance may be impacted. We evaluate this scenario in Section 4.5 and observe only 6% loss in latency when 10% of requests arrive without previously sending an advisory request. If SYMPHONY receives no advisory request, it gracefully degrades to a stateful baseline.

Adversarial Vulnerabilities We acknowledge that there can be potential vulnerabilities, where a bad actor may send several spurious advisory requests, overwhelming the system. Mitigating these vulnerabilities is not the focus of this work. However, note that spurious requests can be identified as caches are fetched and purged for a session without use. Given this, DDoS mitigation techniques could be leveraged to suppress such requests by throttling the users or IP addresses responsible for a disproportionate volume of them.

4 Evaluation

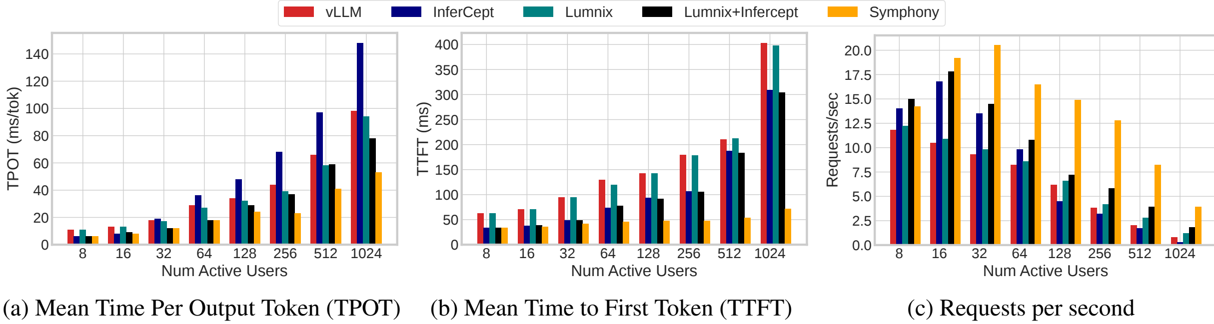
We next evaluate SYMPHONY using traces from real-world LLM chatbot sessions, highlighting its benefits in time per output token and GPU-cluster load balancing.

Baseline Systems We compare SYMPHONY with three popular baselines, vLLM [17], InferCept [1] and Llumnix [37]. vLLM treats each request in a multi-round session as a new request and performs recomputation of the KV cache every time. InferCept is a system that offloads KV caches to the host's main memory. Llumnix handles unpredictability by performing iteration-level scheduling. However, Llumnix does not consider KV cache management for multi-turn workloads like chatbots and agents and will use recomputation to serve multi-turn workloads. Further, we have enhanced Llumnix and created a baseline where we integrate Llumnix with Infercept (Llumnix + Infercept), enabling Llumnix to avoid recomputation. This baseline enhances a *swap* based baseline (Llumnix) with an *offload* based baseline (Infercept) and enables us to clearly study how SYMPHONY's KV cache prefetching improves performance. Finally, we also compare SYMPHONY against LMcache [21], a novel KV cache compression algorithm which can perform lossy compression over the KV cache. For vLLM, we use SYMPHONY 's global controller for scheduling, minimizing load imbalance across serving machines. For InferCept and Llumnix+InferCept, the first request in each session is routed to the node with the fewest active requests, while subsequent requests in that session are sent to the same node. For Llumnix and Llumnix+InferCept, we retain the original scheduler.

Testbed Setup. Unless otherwise noted, all experiments use 2 nodes, each with 4 NVIDIA A100 GPUs (80GB HBM), 256GB DRAM, and 4TB SSDs. The nodes are connected via a 100Gbps Ethernet link.

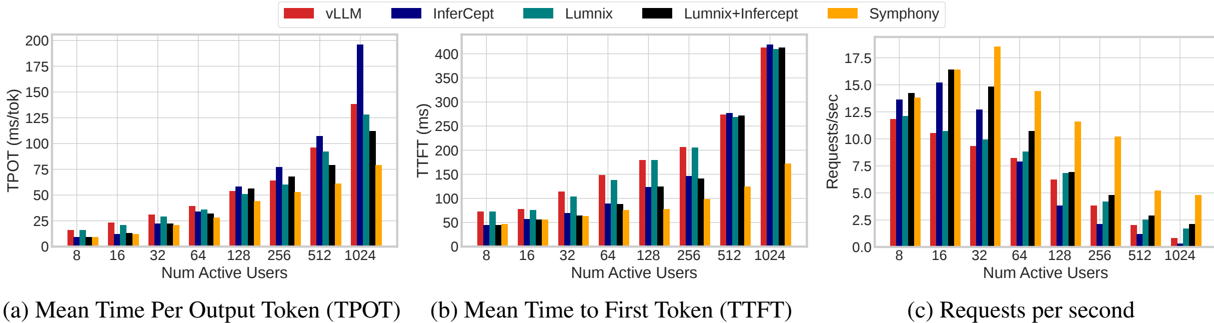
Models We evaluate SYMPHONY on LLAMA-3.1 8B [7] with 128K context length and LLAMA-2-13B with 32K context length [16].

Trace Generation *Chatbot:* Currently, no open-source trace provides detailed information about user interactions with chatbots, such as typing behavior or comprehension speed. To evaluate SYMPHONY, we therefore repurpose two open-source traces: ShareGPT [27] and BurstGPT [47]. The ShareGPT dataset contains real-world chat sessions with ChatGPT but lacks request arrival times. In contrast, BurstGPT captures arrival patterns but does not indicate whether requests belong to the same session. For ShareGPT, each user session consists of a sequence of user queries and LLM responses. We generate request arrival times by estimating the time required to read a response and type the subsequent



(a) Mean Time Per Output Token (TPOT) (b) Mean Time to First Token (TTFT) (c) Requests per second

Figure 11: **Comparing SYMPHONY with existing systems on LLAMA-3.1 8B:** Compared with vLLM and InferCept, can serve 4× the users (64 vLLM and 256 for SYMPHONY) while maintaining a similar time per output token.(18.5ms vs 20.5ms)



(a) Mean Time Per Output Token (TPOT) (b) Mean Time to First Token (TTFT) (c) Requests per second

Figure 12: **Comparing SYMPHONY with existing systems on LLAMA-2 13b:** Here we compare LLAMA-2-13b on all the three metrics. We observe up to 2.6× speedup in TPOT, around 2.56× in TTFT.

query, using human reading and typing speeds derived from prior studies [28, 42]. After initializing these parameters, the next request is scheduled based on the estimated reading and typing time. Our trace generator maintains a fixed number of active users, and unless otherwise stated, all evaluations use 1000 samples from the ShareGPT dataset. We acknowledge the simplifying assumption that a user issues the next request immediately after reading the previous response. However, we argue that SYMPHONY 's performance is insensitive to this assumption, since it evicts the key–value cache immediately after request completion and relies on advisory requests to reload it into GPU memory. To further validate this claim, we also evaluate on BurstGPT.

For BurstGPT, we follow the methodology outlined by the authors, which provides aggregate statistics on inter-arrival times between queries in the same session (Figure 9 in [47]). We use these statistics to generate session-level inter-arrival times. Advisory requests are then scheduled by estimating the time required to type the subsequent query and issuing the advisory at the estimated typing interaction start time.

Agents: For agent evaluation, we use MetaGPT [14] framework. We compare against Parrot [20] and vLLM and use the same dataset as used by Parrot [20].

Evaluation Metrics Following prior work in LLM inference, we evaluate SYMPHONY on three metrics: Time Per Output Token (TPOT), Time to First Token (TTFT), and Requests served per second. Also, as in [1, 17, 51], average normalized latency is defined as the mean of end-to-end latency divided by the output length of each request.

4.1 Implementation

SYMPHONY is primarily implemented in around 2400 lines of Python. SYMPHONY uses gRPC [11] to integrate the scheduler with node manager. Further, each SYMPHONY node manager is connected with all other node managers, which enables a node manager to perform on-demand KV cache migration from one machine to another. For performing cooperative memory management SYMPHONY integrates with vLLM's scheduler and performs block management.

4.2 Comparing SYMPHONY

We first evaluate SYMPHONY by comparing it against existing systems. Then we dive deeper into understanding where the benefits of SYMPHONY come from.

Time Per Output Token. In Figure 11a, we compare SYMPHONY with vLLM [17] and InferCept [1] and Llumnix [37] while serving LLAMA-3.1 8B while varying the number of concurrent users. The results show that SYMPHONY reduces mean TPOT latency by a factor of 1.4× to 1.9× compared to vLLM. Similarly, in Figure 12a, we observe that when serving LLAMA-2-13B, SYMPHONY achieves latency reductions of 1.31× to 1.9×. These improvements are primarily due to two factors. First, unlike vLLM, SYMPHONY eliminates redundant computations, resulting in significant compute savings. Second, since SYMPHONY, like vLLM, employs continuous batching, the faster prefill phase

allows more requests to transition to the decode phase faster, improving the overall throughput of the system.

When compared to InferCept, SYMPHONY delivers a speedup of up to 2.5×. This improvement is primarily because InferCept uses a stateful serving model, which causes significant load imbalances across the cluster. In some cases, the number of requests on a single server can be as much as 2.8× the median load across the cluster. This imbalance leads to severe latency increases and suboptimal utilization.

SYMPHONY achieves up to 1.8× TPOT speedup over Llumnix, primarily by eliminating redundant KV cache computation. It also delivers up to 1.6× speedup over Llumnix+InferCept, with the 0.2× gains largely attributable to InferCept's avoidance of redundant computation.

The performance difference stems from Llumnix's scheduler, which performs online load balancing while requests are running. However, it can only migrate a request after transferring its KV cache; until then, the request continues on an overloaded GPU. Since KV caches can be tens of gigabytes in size, migration is slow, reducing overall throughput. In contrast, SYMPHONY leverages advisory requests to preemptively migrate KV caches, avoiding load imbalance entirely.

Time to First Token In Figures 12b and 11b, we observe that SYMPHONY reduces the time to first token by up to 2.4× compared to the vLLM and InferCept baselines. vLLM suffers from wasteful prefill operations when recomputing KV caches of prior requests, which significantly increases the time to the first token. In contrast, InferCept avoids wasteful pre-fill as it prioritizes completing the decode step on sequences whose KV caches are stored in host memory, rather than focusing on prefill as vLLM does. Llumnix also needs to perform wasteful pre-fill operations to recompute KV caches from prior requests, which leads to 2.4× slowdown compared to SYMPHONY. Llumnix+Infercept can avoid wasteful prefills by reading KV cache from the host memory. However, Llumnix+Infercept still suffers from slow decode times due to load imbalance, as for the first few decoding steps, the Llumnix scheduler is waiting for the KV cache migration to occur, leading to similar TTFT as InferCept.

Requests Served/Sec. In Figures 11c and 12c, we observe that SYMPHONY serves up to 8× the number of users while maintaining a similar time per output token. For example, when serving 64 users, vLLM has an average time per output token of 18ms, whereas SYMPHONY can serve 512 users with an average time per output token of 20.5ms while using the same number of GPUs. The primary reason SYMPHONY outperforms vLLM is its ability to minimize redundant computation of KV caches for multi-turn chatbot requests.

When compared to InferCept, SYMPHONY can handle over 4.2× the number of requests at high load (1024 users). The main reason InferCept is significantly slower is due to load imbalance, which causes poor cluster utilization. When compared to Llumnix+Infercept, a significantly stronger baseline, SYMPHONY can handle upto 1.9× higher load (1024

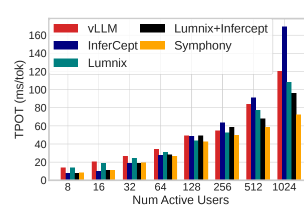


Figure 13: **Evaluation on BurstGPT:** On the BurstGPT dataset SYMPHONY provides up to 2.5× speedup.

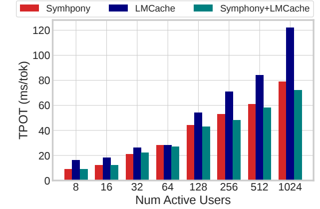


Figure 14: **Comparing with LMCache:** SYMPHONY can provide up to 35% speedup over LMCache.

active users), primarily because for the first several decoding steps, there is high load imbalance while the Llumnix scheduler is trying to migrate the KV caches. Secondly, Llumnix+Infercept also suffers from reloading and recomputation overheads at high load, as Infercept only supports storing KV cache into host memory and discards if the KV cache sizes grow higher than host memory.

Evaluation on BurstGPT We perform additional evaluation over BurstGPT dataset. In Figure 13, we observe a very similar speedup to the evaluation over the ShareGPT dataset. This is expected because both these datasets are constructed of realistic chatbot-like workloads.

Comparison with LMCache In LMCache [21], the authors propose a lossy compression scheme that enables streaming of the key–value cache. Instead of relying on local memory, LMCache offloads the cache to remote storage. As shown in Figure 14, SYMPHONY outperforms LMCache by up to 35% under high load. The primary reason is that LMCache performs cache loading on the critical path, and its decompression step interferes with other GPU requests, resulting in slowdown at scale. Finally, SYMPHONY can integrate the LMCache compression algorithm to enable a multi-tier disaggregated storage. We show that by hiding the latency of fetching the cache from remote servers and reducing the size of transfer over the contended PCI express SYMPHONY can lead to further improvements of over 47%.

4.3 Load Imbalance

Next, we plot the load imbalance while serving LLAMA-3.1-8b with 256 concurrent users. In Figure 15, we observe that, unlike vLLM and SYMPHONY, InferCept exhibits significant load imbalance. Specifically, the maximum number of users served by a single instance is more than 3.1× the median load across machines.

4.4 Serving Agent workloads

Next, we evaluate SYMPHONY for serving agent workloads. For this, we use MetaGPT [14], a multi-agent framework designed to simulate an AI software development company. In MetaGPT, different roles are defined for each LLM. The framework proposes a few main roles: a program manager, an architect who defines the project, designs data structures

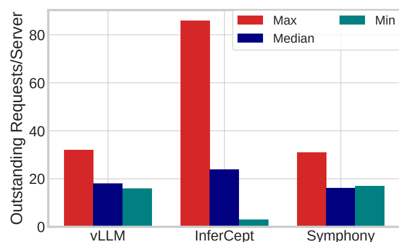


Figure 15: **Measuring Load Imbalance**: On a load of 256 concurrent users. We observe both SYMPHONY and vLLM can keep load-balanced due to performing stateless serving.

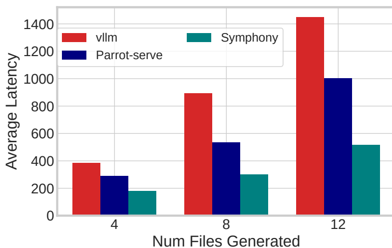


Figure 16: **Serving MetaGPT using SYMPHONY**: We observe that compared to vLLM, and Parrot-Serve SYMPHONY can reduce the time by 2.8× and 1.7×

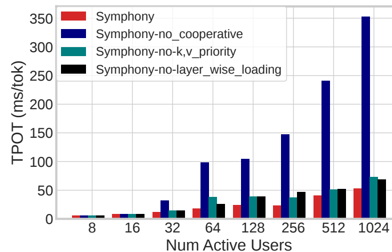


Figure 17: **Understanding impact of SYMPHONY's optimization**: We switch off different optimizations in SYMPHONY and plot TPOT with varying number of active users.

and APIs, and writes multiple design documents; engineers who take over coding tasks, each focusing on specific files; and QA engineers who review the code, followed by engineers making revisions. This review and revision cycle can occur upto three times. We serve MetaGPT requests using SYMPHONY, utilizing LLAMA-3.1-8b across 8 GPUs.

For comparison, we use two baselines, vLLM [17] and Parrot-Serve [20]. Unlike vLLM, Parrot-serve introduces a semantic variable abstraction to chain multiple LLM requests. However, this abstraction fails when processing agentic workflows with conditionals, as requests can no longer be processed in a single pass. Similar to Infercept, this forces the serving state to become fragmented and stateful, leading to load imbalance. Furthermore, because Parrot relies exclusively on keeping KV caches in GPU memory, it must discard prior requests when memory is exhausted.

In Figure 16 we plot the time required to generate 4,8 and 12 code files. Our results show that SYMPHONY can reduce the overall time by 2.8× because advisory requests enable KV cache offloading and request load balancing.

Profiling Overhead: To analyze the agent call graph and estimate the execution time of an agent, we randomly sample ten requests from the MetaGPT dataset. Profiling these ten prompts on a single node with four A100 GPUs takes approximately 12 minutes.

4.5 Ablation

We systematically study key properties of SYMPHONY.

Evaluating Cooperative Memory Management. First, in Figure 17, we examine the impact of disabling cooperative memory management. Without it, SYMPHONY experiences a significant throughput loss as requests are blocked due to insufficient memory. This triggers swapping between host memory and GPU HBM, resulting in slowdowns of over 7×.

Evaluating Priority-Based KV Cache Management. Priority-based KV cache management improves SYMPHONY's throughput under high memory pressure. Without it, throughput drops by up to 21.7% at peak loads.

Impact of Layer-Wise Loading. Layer-wise loading overlaps KV cache loading with computation, an optimization beneficial primarily at high load. Without it, throughput drops by approximately 18.4% under heavy load.

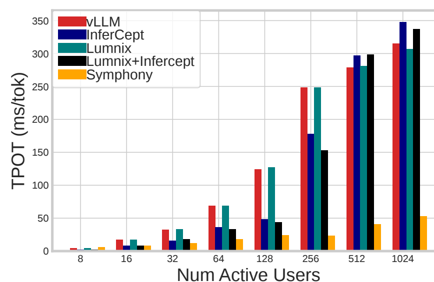


Figure 18: **Understanding Impact of Load Balancing**: We create an artificial pre-fill heavy workload consisting solely of pre-fill requests. Even in this favorable scenario, InferCept performs poorly due to inadequate load balancing.

Performance under prefill-heavy requests. To understand the importance of load balancing, we evaluate a new workload that should significantly benefit both InferCept, Llumnix+Infercept, and SYMPHONY. In this "pre-fill-heavy" workload, we replace all requests with a 1024-token prompt and a 1-token response while keeping the original arrival times. Prefill-heavy workloads are becoming common because LLMs are often being used to answer questions where the response is limited to one-word answers, like classification, routing, etc. This configuration benefits both Infercept and Infercept+Llumnix, as they retain the KV cache associated with previous prompts and responses. However, the key difference is load imbalance. As shown in Figure 18, despite the favorable setup, InferCept struggles to achieve high throughput due to load imbalance. Similarly, for Llumnix+Infercept the TPOT is very close to Infercept, because Llumnix tries to perform load balancing only after a few decode steps; however, in this workload, we only perform one decode step.

Performance Without advisory Requests. In cases, such as when a user pastes text into a chatbot and sends it, advisory requests may arrive too late/missed. To evaluate SYMPHONY under missed advisory requests, we simulate scenarios with varying miss rates. As shown in Figure 19, normalized token latency increases with the proportion of missed requests. For a

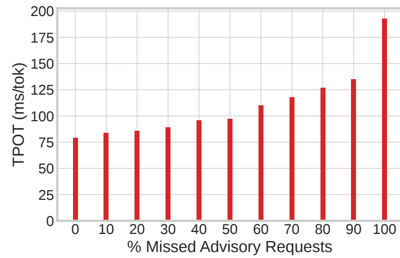


Figure 19: **Effect of missed advisory request**: Missing 10% advisory request we observed a degradation of 3.1 ms. If all the advisory requests (100%) are misse,d we observe performance on par with a stateful baseline like Infercept.

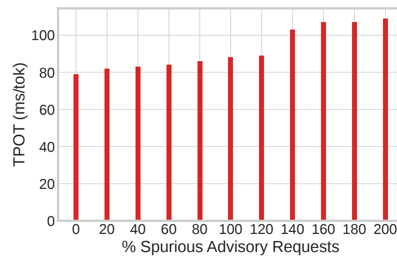


Figure 20: **Impact of false positive advisory requests:** When serving 1024 requests, if an additional 60% false positive requests arrive, it leads to only 2.1% degradation in throughput. We observe that false positive advisory requests lead to a very small increase in overall throughput.

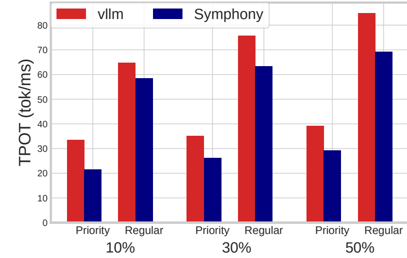


Figure 21: **Prioritization**: Our policy prioritizes requests belonging to prioritized sessions. We observe that SYMPHONY improves the Time Per Output Token (TPOT) for both prioritized and regular jobs as we vary the percentage of high-priority requests

10% miss rate, latency per token rises from 21.3ms to 24.4ms, a roughly 6% increase, primarily due to reduced effectiveness in load balancing and KV cache prefetching.

Performance Under Spurious advisory Requests. In LLM workloads, an inference request associated with an advisory request may never arrive—for example, when a user types text but does not submit it. As shown in Figure 20, SYMPHONY tolerates a high volume of such spurious advisory requests with minimal performance impact. Even when serving 1,024 requests on LLaMA-2-13B with an 8-GPU setup, adding 60% spurious advisory requests results in only a 2.1% performance degradation. This low impact is due to priority-based KV cache management and layer-wise KV loading, which frees GPU HBM and gives requests the illusion of virtually unlimited memory.

Prioritization Policy. To demonstrate that SYMPHONY supports additional policies beyond load balancing, we implement a prioritization policy. For sessions marked as high priority, the KV cache for the associated request is immediately moved to the GPU HBM to prevent delays in processing. For vLLM, we use the prioritization parameter provided by vLLM to indicate a request's high priority. We vary the percentage of high-priority requests (10%, 30%, 50%) and compare this approach with vLLM's priority scheduling. Our results (Figure 21) show that, due to the absence of redundant computation and zero overhead in KV cache migration, SYMPHONY delivers better tokens per second for both prioritized and regular jobs.

5 Related Work

There have been several prior works that have studied scheduling for LLM workloads.

Scheduling for LLMs Several prior works have studied scheduling in the context of Large language models. Sarathi-Serve [2] studied scheduling from the perspective of trading throughput vs latency. VTC [35] introduces the idea of fairness in serving multiple LLM requests. Serverless LLMs [8] introduced the idea of locality for LLM serving.

Large Language Models. Attention first introduced by [43] forms the basic building block in various language understanding tasks such as text generation, text classification, and question answering [34, 46]. Recent works, e.g., GPT [4, 33], LLaMA [7, 40, 41], Qwen [3, 49], and Gemma [38, 39] have shown that scaling foundation models can achieve high accuracy on many downstream tasks.

KV cache management. Prior work has explored reducing KV cache requirements for LLMs. Liu et al. [23] exploit contextual sparsity, while [50] share caches across layers for compression without accuracy loss. Other frameworks [22,48] retain only selected cache segments to cut inference cost. Swapping-based approaches include CachedAttention [10], which stores caches in hierarchical memory tiers; token-importance-based loading [18]; and pipeline-bubble mitigation via efficient cache management [36]. In this work, we focus on KV cache management for multi-turn workloads.

6 Conclusion

In this work, we introduce SYMPHONY, a disaggregated memory management framework designed to decouple memory and state for serving multi-turn workloads. SYMPHONY builds on the key observation that popular LLM workloads often provide enough information to predict, with high probability, the arrival of future requests. Leveraging this insight, SYMPHONY uses advisory requests—signals indicating the likely arrival of subsequent requests—to prefetch K,V caches from far memory. This proactive caching mechanism enables more efficient high-level scheduling and ensures the system is well-prepared to handle modern inference workloads.

Acknowledgement We thank Nitin Kedia, anonymous reviewers, and our shepherd, Ravi Netravali, for valuable feedback. This work is supported by the U.S. National Science Foundation (NSF) under Grant Number 2326576. This work was also supported in part by NSF CNS-2237306, CNS-2203152, CNS-2214015, Cisco-Research and Nutanix.

References

[1] Reyna Abhyankar, Zijian He, Vikranth Srivatsa, Hao Zhang, and Yiying Zhang. Infercept: Efficient intercept support for augmented large language model inference. In *Forty-first International Conference on Machine Learning*.

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

[3] Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, et al. Qwen technical report. *arXiv preprint arXiv:2309.16609*, 2023.

[4] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[5] Siyuan Chen, Zhipeng Jia, Samira Khan, Arvind Krishnamurthy, and Phillip B Gibbons. Slos-serve: Optimized serving of multi-slo llms. *arXiv preprint arXiv:2504.08784*, 2025.

[6] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, et al. The snowflake elastic data warehouse. In *Proceedings of the 2016 International Conference on Management of Data*, pages 215–226, 2016.

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

[8] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. Serverlessllm: Locality-enhanced serverless inference for large language models. *arXiv preprint arXiv:2401.14351*, 2024.

[9] Bin Gao, Zhuomin He, Puru Sharma, Qingxuan Kang, Djordje Jevdjic, Junbo Deng, Xingkun Yang, Zhou Yu, and Pengfei Zuo. Attentionstore: Cost-effective attention reuse across multi-turn conversations in large language model serving. *arXiv preprint arXiv:2403.19708*, 2024.

[10] Bin Gao, Zhuomin He, Puru Sharma, Qingxuan Kang, Djordje Jevdjic, Junbo Deng, Xingkun Yang, Zhou Yu, and Pengfei Zuo. {Cost-Efficient} large language model serving for multi-turn conversations with {CachedAttention}. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 111–126, 2024.

[11] Google. Grpc:a high performance, open source universal rpc framework. https://grpc.io/, 2012. Accessed: May 18, 2022.

[12] Robert Grandl, Arjun Singhvi, Raajay Viswanathan, and Aditya Akella. Whiz: Data-driven analytics execution. In James Mickens and Renata Teixeira, editors, *18th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2021, April 12-14, 2021*, pages 407–423. USENIX Association, 2021.

[13] Scott Hendrickson, Stephen Sturdevant, Edward Oakes, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Serverless computation with openlambda. *login Usenix Mag.*, 41(4), 2016.

[14] Sirui Hong, Xiawu Zheng, Jonathan Chen, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, et al. Metagpt: Meta programming for multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*, 2023.

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

[16] Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. Mistral 7b. *arXiv preprint arXiv:2310.06825*, 2023.

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

[18] Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. InfiniGen: Efficient generative inference of large language models with dynamic KV cache management. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 155–172, Santa Clara, CA, July 2024. USENIX Association.

[19] Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning*, pages 19274–19286. PMLR, 2023.

[20] Chaofan Lin, Zhenhua Han, Chengruidong Zhang, Yuqing Yang, Fan Yang, Chen Chen, and Lili Qiu. Parrot: Efficient serving of {LLM-based} applications with semantic variable. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 929–945, 2024.

[21] Yuhan Liu, Hanchen Li, Yihua Cheng, Siddhant Ray, Yuyang Huang, Qizheng Zhang, Kuntai Du, Jiayi Yao, Shan Lu, Ganesh Ananthanarayanan, et al. Cachegen: Kv cache compression and streaming for fast large language model serving. In *Proceedings of the ACM SIGCOMM 2024 Conference*, pages 38–56, 2024.

[22] Zichang Liu, Aditya Desai, Fangshuo Liao, Weitao Wang, Victor Xie, Zhaozhuo Xu, Anastasios Kyrillidis, and Anshumali Shrivastava. Scissorhands: Exploiting the persistence of importance hypothesis for llm kv cache compression at test time. *Advances in Neural Information Processing Systems*, 36, 2024.

[23] Zichang Liu, Jue Wang, Tri Dao, Tianyi Zhou, Binhang Yuan, Zhao Song, Anshumali Shrivastava, Ce Zhang, Yuandong Tian, Christopher Re, et al. Deja vu: Contextual sparsity for efficient llms at inference time. In *International Conference on Machine Learning*, pages 22137–22176. PMLR, 2023.

[24] Nvidia. Nvidia-dynamo. https://github.com/ai-dynamo/dynamo, 2024. Accessed: November 11, 2024.

[25] NVIDIA/TensorRT-LLM. Tensorrt-llm. https://github.com/NVIDIA/TensorRT-LLM, 2024. Accessed: November 11, 2024.

[26] Archit Patke, Dhemath Reddy, Saurabh Jha, Haoran Qiu, Christian Pinto, Chandra Narayanaswami, Zbigniew Kalbarczyk, and Ravishankar Iyer. Queue management for slo-oriented large language model serving. In *Proceedings of the 2024 ACM Symposium on Cloud Computing*, pages 18–35, 2024.

[27] philschmid/sharegpt raw. sharegpt. https://huggingface.co/datasets/philschmid/sharegpt-raw/tree/main/sharegpt_90k_raw_dataset, 2023. Accessed: November 11, 2024.

[28] Svetlana Pinet, Christelle Zielinski, F-Xavier Alario, and Marieke Longcamp. Typing expertise in a large student population. *Cognitive Research: Principles and Implications*, 7(1):77, 2022.

[29] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference. *Proceedings of Machine Learning and Systems*, 5:606–624, 2023.

[30] Chen Qian, Xin Cong, Cheng Yang, Weize Chen, Yusheng Su, Juyuan Xu, Zhiyuan Liu, and Maosong Sun. Communicative agents for software development. *arXiv preprint arXiv:2307.07924*, 6:3, 2023.

[31] Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. Chatdev: Communicative agents for software development. *arXiv preprint arXiv:2307.07924*, 2023.

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

[33] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[34] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. Squad: 100,000+ questions for machine comprehension of text. *arXiv preprint arXiv:1606.05250*, 2016.

[35] Ying Sheng, Shiyi Cao, Dacheng Li, Banghua Zhu, Zhuohan Li, Danyang Zhuo, Joseph E Gonzalez, and Ion Stoica. Fairness in serving large language models. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 965–988, 2024.

[36] Foteini Strati, Sara Mcallister, Amar Phanishayee, Jakub Tarnawski, and Ana Klimovic. D\'ej\avu: Kv-cache streaming for fast, fault-tolerant generative llm serving. *arXiv preprint arXiv:2403.01876*, 2024.

[37] Biao Sun, Ziming Huang, Hanyu Zhao, Wencong Xiao, Xinyi Zhang, Yong Li, and Wei Lin. Llumnix: Dynamic scheduling for large language model serving. *arXiv preprint arXiv:2406.03243*, 2024.

[38] Gemma Team, Thomas Mesnard, Cassidy Hardin, Robert Dadashi, Surya Bhupatiraju, Shreya Pathak, Laurent Sifre, Morgane Rivière, Mihir Sanjay Kale, Juliette Love, et al. Gemma: Open models based on gemini research and technology. *arXiv preprint arXiv:2403.08295*, 2024.

[39] Gemma Team, Morgane Riviere, Shreya Pathak, Pier Giuseppe Sessa, Cassidy Hardin, Surya Bhupatiraju, Léonard Hussenot, Thomas Mesnard, Bobak Shahriari, Alexandre Ramé, et al. Gemma 2: Improving open language models at a practical size. *arXiv preprint arXiv:2408.00118*, 2024.

[40] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[41] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[42] Susanne Trauzettel-Klosinski, Klaus Dietz, IReST Study Group, et al. Standardized assessment of reading performance: The new international reading speed texts irest. *Investigative ophthalmology & visual science*, 53(9):5452–5461, 2012.

[43] A Vaswani. Attention is all you need. *Advances in Neural Information Processing Systems*, 2017.

[44] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1041–1052, 2017.

[45] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. Building an elastic query engine on disaggregated storage. In Ranjita Bhagwan and George Porter, editors, *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, pages 449–462. USENIX Association, 2020.

[46] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R Bowman. Glue: A multi-task benchmark and analysis platform for natural language understanding. *arXiv preprint arXiv:1804.07461*, 2018.

[47] Yuxin Wang, Yuhan Chen, Zeyu Li, Xueze Kang, Yuchu Fang, Yeju Zhou, Yang Zheng, Zhenheng Tang, Xin He, Rui Guo, et al. Burstgpt: A real-world workload dataset to optimize llm serving systems. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V. 2*, pages 5831–5841, 2025.

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

[49] An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, et al. Qwen2 technical report. *arXiv preprint arXiv:2407.10671*, 2024.

[50] Yifei Yang, Zouying Cao, Qiguang Chen, Libo Qin, Dongjie Yang, Hai Zhao, and Zhi Chen. Kvsharer: Efficient inference via layer-wise dissimilar kv cache sharing. *arXiv preprint arXiv:2410.18517*, 2024.

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

[52] Wen Zhang, Vivian Fang, Aurojit Panda, and Scott Shenker. Kappa: A programming framework for serverless computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 328–343, 2020.

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

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