Rethinking Caching for LLM Serving Systems: Beyond Traditional Heuristics

Jungwoo Kim
jungwoo@dgist.ac.kr
DGIST
South Korea

Minsang Kim
kimmsang96@dgist.ac.kr
DGIST
South Korea

Jaeheon Lee
jaeheon@postech.ac.kr
POSTECH
South Korea

Chanwoo Moon
ixora99@dgist.ac.kr
DGIST
South Korea

Heejin Kim
noah211@postech.ac.kr
POSTECH
South Korea

Taeho Hwang
taeho.hwang@sk.com
SK hynix
South Korea

Woosuk Chung
woosuk.chung@sk.com
SK hynix
South Korea

Yeseong Kim
yeseongkim@dgist.ac.kr
DGIST
South Korea

Sungjin Lee
sungjin.lee@postech.ac.kr
POSTECH
South Korea

Abstract

Serving Large Language Models (LLMs) at scale requires meeting strict Service Level Objectives (SLOs) under severe computational and memory constraints. Nevertheless, traditional caching strategies fall short: exact-matching and prefix caches neglect query semantics, while state-of-the-art semantic caches remain confined to traditional intuitions, offering little conceptual departure. Building on this, we present SISO, a semantic caching system that redefines efficiency for LLM serving. SISO introduces centroid-based caching to maximize coverage with minimal memory, locality-aware replacement to preserve high-value entries, and dynamic thresholding to balance accuracy and latency under varying workloads. Across diverse datasets, SISO delivers up to 1.71× higher hit ratios and consistently stronger SLO attainment compared to state-of-the-art systems.

1 Introduction

The rise of Large Language Models (LLMs) has triggered a paradigm shift in various applications [7], transforming dominant information retrieval from traditional data management systems (e.g., databases and data warehouses) to LLM inference. This transformation, however, incurs substantially higher energy consumption and computational demands. As a result, there is a crucial need to develop efficient LLM serving (or inference) systems that ensure high-quality user experience by meeting strict Service Level Objectives (SLOs) at lower energy usage and operational costs. Toward this goal, many have explored various optimization strategies, including LLM task scheduling [40, 73], GPU memory optimization [36, 62], and model scaling [2, 68].

Despite their effectiveness, existing approaches often require system-wide modifications, limiting their portability across various platforms and their compatibility with other techniques. As a complementary direction, caching can be employed to reuse previously computed outputs – an idea long established in traditional applications like SQL query caching [11, 14, 34]. Current LLM caching solutions, however, are built upon conventional intuition for caching, treating LLM queries as structured strings or instructions like SQL queries [10, 46, 53, 71]. This prevents us from fully exploiting the potential of LLM caching. For example, "What is semantic caching?" and "Explain semantic caching" are semantically similar and expected to produce similar outputs, but existing LLM caching fails to recognize them as the same query.

In light of these limitations, semantic caching has recently emerged, focusing on the meaning of queries. This approach offers promising opportunities for extending caching methodologies to LLMs by incorporating semantic similarity into query caching. However, prior work on semantic caching has mostly been studied in the context of LLMs, such as similarity matching algorithms [6, 22, 72], with limited system-level design considerations. For example, state-of-the-art (SOTA) systems (e.g., GPTCache [6]) naively adopt traditional policies when promoting or evicting queries, without tailoring them to the unique characteristics of LLM workloads. This oversight leads to several limitations.

First, existing semantic caching treats individual queries as a unit of caching, promoting or demoting them independently. Although it is a common practice in conventional caching, this approach leads to storing duplicate queries with nearly identical meanings, resulting in a waste of cache space. Second, existing semantic caching relies on recency or frequency for eviction, which is invoked frequently when the cache needs to free up space for new queries. While it seems reasonable, such replacement policies often make wrong decisions in LLM workloads, evicting valuable queries and thereby reducing overall hit ratios. Third, existing semantic caching requires all requests to pass through the semantic caching layer, regardless of whether caching is beneficial.

Unlike traditional caching that always returns accurate responses, semantic caching returns similar outputs for input queries, inherently trading accuracy for performance. Thus, semantic caching should be applied selectively, when performance has a higher priority than accuracy (e.g., when heavy computation might lead to the violation of SLOs).

In this paper, we propose SISO (**S**imilar **I**nput? **S**imilar **O**utput!), a novel semantic caching system that moves beyond traditional heuristics, optimized for the semantic characteristics of LLM queries and workloads. SISO incorporates three features. The first is *centroid-based caching*. Instead of caching individual queries, SISO only stores centroids which represent many semantically similar queries. By keeping only valuable queries, SISO uses the cache space efficiently without any redundant caching.

The second is *semantic locality-aware centroid replacement*. Here, semantic locality refers to the degree to which a centroid can represent a broad range of semantically similar queries. SISO prioritizes caching centroids with strong semantic locality, while evicting those with weaker semantic locality. Since semantic locality remains relatively stable over time, replacements are triggered occasionally by monitoring long-term query behaviors. This approach yields higher cache hit ratios than traditional LRU and LFU policies.

The third is *dynamic threshold adjustment* to balance output quality and serving latency. While serving input queries, SISO dynamically adjusts a similarity threshold, which decides whether input queries are sufficiently similar to cached centroids or not. When workloads are intensive, the threshold can be relaxed to increase cache hits, alleviating pressure on the LLM serving system. Conversely, when workloads are light, it can be tightened to maximize response quality.

To evaluate the effectiveness of SISO, we conduct experiments using various real-world datasets [16, 24, 32, 44] with two LLMs: LLaMa-3.1 8B and 70B [18]. For evaluation, we compare SISO with the SOTA LLM serving system, vLLM [36], and semantic caching system, GPTCache [6]. Through extensive experiments, we obtain the following key results. First, by caching centroids rather than individual outputs, SISO can efficiently utilize available cache space, thereby exhibiting 1.54× higher hit ratios than GPTCache on average. Second, through semantic locality-aware replacement, SISO improves the average cache hit ratios by 1.71× by maintaining more valuable centroids in the cache. Finally, by dynamically adjusting its similarity threshold, SISO achieves higher SLO attainment under intensive and/or highly variable workloads, surpassing both vLLM and GPTCache. This improvement comes with only a marginal accuracy drop of 6.9%. Under light workloads, SISO maintains the same accuracy as vLLM without any loss of output quality.

While SISO beats existing systems, its benefits are limited to single-turn queries for specific types of tasks, such as information and advice seeking. The current version cannot support context-dependent tasks (i.e., multi-turn queries) and performs relatively poorly on coding and debugging tasks. However, given single-turn queries dominate user interactions (99% in API calls and 67% in chatbots [67]), and that information- and advice-seeking account for 53.1% of all queries, these limitations do not negate SISO's value but highlight a clear need for future research in semantic caching.

This paper is organized as follows: §2 gives the background and related work. After presenting key design principles of SISO in §3, we detail the implementation of SISO in §4, explaining how SISO is implemented based on its design principles. After showing experimental results in §5, we discuss limitations of SISO and directions for further improvements in §6. We finally conclude in §7.

2 Background and Related Work

This section gives an overview of LLM serving systems, along with reviews of prior efforts to improve their performance (§2.1 – §2.3).

2.1 LLM Serving Systems

The rapid advent of LLMs has increased the demand for developing LLM serving systems, which are designed to deploy LLMs for real-time inference tasks [26, 36, 50]. The primary challenge in building these systems lies in efficiently managing computational resources (e.g., GPUs and CPUs) to consistently meet strict latency requirements, SLOs [1].

The difficulty in meeting SLOs originates from the LLM inference process, which is divided into two distinct phases. The first is the compute-intensive prefill phase, where the model processes the user's entire query to generate the first token. This initial burst of computation determines the Time-to-First-Token (TTFT), which governs the user's perception of initial responsiveness. The second is the decoding phase that sequentially generates the remaining response one token at a time. The latency of each step is measured as Time-between-Tokens (TBT). The decoding has relatively low computation, but since it relies on a huge KV cache of intermediate attention states to avoid costly recomputation, it is considered memory intensive. Consequently, these two metrics, TTFT and TBT, are used to define SLOs, which are typically set to match human reading speed (e.g., ~50ms per token) or constrained to less than 1.3× the latency observed under a zero-load setup [31, 48].

LLM serving systems should be designed to avoid violating SLOs not only during periods of low demand, but also during peak demand. The brute-force solution to achieve this is resource over-provisioning, a practice of allocating more computing resources than required during regular use [61]. Nevertheless, over-provisioning is a costly safeguard – wasteful for most of the time, yet indispensable to prevent SLO violations during the peak. As a result, excess resources remain idle during off-peak hours, inevitably increasing hardware

acquisition and operational costs. This inefficiency highlights the need for more cost-efficient serving strategies.

2.2 Optimization of LLM Serving Systems

There have been numerous efforts to improve SLO compliance and cost efficiency [28, 41, 45]. These studies have evolved along two primary axes: (i) computational optimization to maximize GPU throughput and (ii) memory optimization to support a higher volume of concurrent requests.

Many have addressed computational inefficiency by optimizing GPU utilization and enhancing pipeline efficiency across prefill and decoding phases with different resource demands. Common techniques include task scheduling tailored to LLM [1, 73, 77], GPU kernel optimizations [13, 27, 78] and system-accelerator co-design [25, 48, 63]. These reduce pipeline bubbles and improve throughput, but demand fundamental modifications to existing systems.

Some have attempted to optimize memory utilization by improving memory management or reducing memory footprints. Prior studies developed specialized custom attention kernels to manage KV cache in non-contiguous blocks without memory fragmentation, while some introduced a contiguous virtual address for efficient KV cache management [30, 36, 37, 52, 74]. Other methods explored footprint reduction through model/KV cache quantization requiring specific tensor layouts or custom GPU kernels [2, 29, 70], and offloading the KV cache to host DRAM or even SSD [33, 38, 62].

The above approaches improve computation and memory efficiency. However, their reliance on architecture-level modifications and custom kernels is disruptive to the serving stack, making them difficult to deploy and even harder to integrate with one another. Such incompatibility significantly undermines their practicality in real-world deployments.

2.3 In-memory Caching for LLM Serving

The aforementioned limitations have motivated the need for a non-intrusive complementary solution. One proven approach from traditional database systems is caching, where in-memory stores [15, 49] have improved RDBMS performance without changes to the underlying logic. Analogously, caching offers a promising approach for improving LLM serving efficiency while preserving system compatibility.

The most natural extension to LLM serving is *exact-match caching*: store a response and serve it again for identical inputs. Due to its simplicity, exact-match caching serves as a basic function in many LLM serving systems [10, 46, 53, 71]. However, its practical impact is limited, as even trivial variations in prompts lead to cache misses.

A more LLM-aware variant is *prefix caching*, which exploits the auto-regressive property of LLMs [66]. For instance, the prompts "Summarize the plot of Star Wars" and "Summarize the plot of Star Trek" share the prefix (i.e., first five tokens). By reusing the KV states of the shared prefix, the system can bypass redundant computations for the latter query. Prior studies have improved its effectiveness by extending cache capacity from DRAM to SSD [53], developing faster overlap detection [76], and allowing slight differences in prefix [71]. Still, prefix caching relies on token-level overlap; in practice, many hits come from system prompts rather than user content [67], and paraphrases with different tokens remain as misses.

As an alternative to exact-match and prefix caching, *semantic caching* has gained attention, which reuses query outputs for new inputs that are semantically similar rather than textually identical. GPTCache [6] is a representative open-source implementation of semantic caching. It keeps a vector representation of an input query along with its corresponding output in memory. When a new request arrives, the semantic caching system computes a vector representation of the input and compares it with cached vectors to identify a semantically similar one, returning its output without LLM execution. Subsequent work has sought to raise hit rates and retrieval efficiency via knowledge distillation [72], advanced pattern matching [39], and federated learning [22].

Despite its potential, SOTA semantic caching systems remain limited: they blindly apply traditional caching heuristics that are not suited for LLMs. First, existing approaches cache individual query embeddings. Due to this granularity, however, even nearly identical queries may be stored multiple times, resulting in wasted cache space.

Secondly, cache eviction is often tied to generic replacement heuristics, such as recency or frequency, that are triggered when the cache is full. While such a short-window replacement suffices in traditional caches where reuse is usually bursty and immediate, LLM serving demands a longer temporal horizon. Queries that appear infrequently or with longer gaps can still be semantically related to future requests. Thus, prematurely evicting them sacrifices reuse opportunities that are especially valuable in semantic caching.

Lastly, the current system always uses cached outputs in the case of a hit, regardless of whether the system is idle or busy. Unlike traditional caches, semantic caching inherently involves approximation – the returned response may only be "similar" to the true one. This approximation is only worthwhile when its performance benefits outweigh the degradation in output quality. Nevertheless, existing semantic caching still relies on fixed cache hit mechanisms, which may not always align with practical serving conditions.

3 Design Principles of SISO

To overcome the limitations of caching for the LLM serving system, we propose *SISO*, a novel semantic caching framework. SISO is designed to leverage the semantic nature of LLM queries while maintaining the practical benefits of traditional caching, such as simple deployment and no architectural changes to underlying serving systems. The proposed

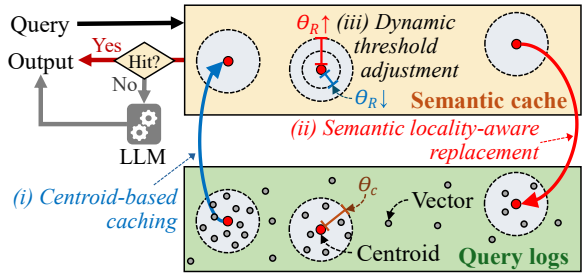


Fig. 1. Overall organization of SISO

SISO manages cache space by considering the unique properties of vectors and analyzing long-term trends of input queries. Additionally, it dynamically adjusts its policies in response to the status of the serving system. In this manner, it enables fast LLM serving performance and delivers the highest possible accuracy with better SLO compliance.

The overall organization of SISO is illustrated in Fig. 1, which is built upon three unique design principles: (*i*) *centroid-based caching*, (*ii*) *semantic locality-aware centroid replacement*, and (*iii*) *dynamic threshold adjustment*.

Centroid-based caching. Instead of storing vectors of individual queries, SISO caches only centroid vectors that represent groups of similar queries, allowing more efficient use of the cache space. To construct these centroids, SISO analyzes query logs (typically recorded by LLM serving systems [75]) containing queries and their responses, clusters the queries based on semantic similarity, and inserts the resulting cluster centroids into the cache. This approach enables efficient use of limited memory without sacrificing output accuracy (see §3.1).

Semantic locality-aware centroid replacement. Unlike existing semantic caching methods that evict vectors based on short-term recency or frequency [6], SISO exploits a different type of locality, *semantic locality*, which can be identified by observing long-term behaviors of input queries. Semantic locality refers to the property that centroids with a wide similarity coverage, representing many other vectors, receive the majority of queries. Since the popularity of centroid evolves slowly, SISO replaces the centroids exhibiting a low semantic locality with higher-locality ones through re-clustering over a long-term time window (see §3.2).

Dynamic threshold adjustment. In contrast to existing semantic caching that uses a fixed similarity threshold θ_R for retrieval [6], SISO adjusts the threshold according to the intensity of workloads. We observe that output quality is linearly proportional to the similarity threshold, meaning that adjusting the threshold directly controls the quality of responses. Leveraging this property, SISO dynamically balances accuracy and performance in a workload-aware manner. Under heavy workloads, it lowers the threshold to increase the chances of finding similar vectors in the cache, thereby raising hit ratios and reducing the load on the underlying LLM serving system. Conversely, under a light workload, it increases the threshold or turns off the semantic cache to maximize the quality of responses (see §3.3).

We now present our analyses and five key observations that support the fundamental design principles for SISO.

3.1 Centroid-based Caching

To show the advantages of centroid-based caching over vector-level caching, we analyze real-world datasets to assess the accuracy of caching centroids and its memory efficiency.

We use Quora Question Pair (QQP), Microsoft Research Paraphrase Corpus (MRPC), and Medical Question Pairs (MQP) [16, 32, 44], which are used to develop algorithms that identify duplicate questions or sentences that have the same meaning. Each dataset item is a set of {text1, text2, is_duplicate}, where text1 and text2 are English texts and is_duplicate is a binary label indicating whether the two texts are duplicates. We measure the cosine similarity of two types of text pairs: one with duplicate text pairs and another with non-duplicate text pairs. To compute the similarity, we utilize a pre-trained embedding model, `paraphrase-albert-small-v2` [55] (see §4.1 for more details on our choice).

Fig. 2 plots the PDF graphs of cosine similarities obtained from duplicate (a blue line) and non-duplicate (a red line) pairs. Duplicate pairs have a high median cosine similarity, averaging 0.82 across datasets. In contrast, non-duplicate pairs exhibit a low median cosine similarity of 0.62 on average. These results show that when threshold values are set sufficiently high, for example, higher than the median values, 0.86 for QQP, 0.83 for MRPC, and 0.76 for MQP, two texts are highly likely to convey the same meaning.

Observation #1. A group of texts with high cosine similarities are likely to have the shared meaning, and thus are considered duplicates.

Impact on accuracy. Based on the above observations, we perform an analysis using realistic datasets to understand how caching only centroids affects accuracy. Here, a centroid is a vector that represents the central or average point of a group of vectors that have similar cosine similarity.

We use two datasets: 600K questions collected from Quora and 600K questions obtained from the Reddit dataset [24]. The dataset is divided into 95% for training and 5% for testing. Using LLaMa-3.1-8B, we generate answers for questions in the training dataset, creating pairs of <question, answer>. We then perform clustering based on cosine similarity and select a centroid to represent each cluster. We set the clustering threshold θ_C to 0.86 for both datasets, a value chosen to be sufficiently high based on Observation #1. The community detection algorithm [54] is used for clustering (see §4.1 for more details), and we found 60K centroids in each dataset.

We compare three systems: Centroid, which caches 60K centroids (ours); GPTCache, which uses an LRU policy with a

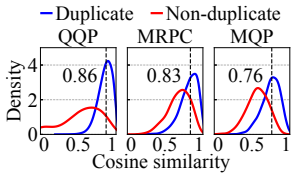


Fig. 2. Cosine similarity analysis

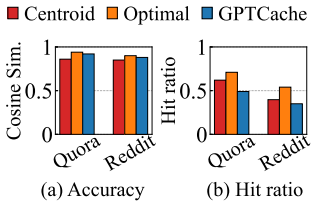


Fig. 3. Impact of centroid-based caching

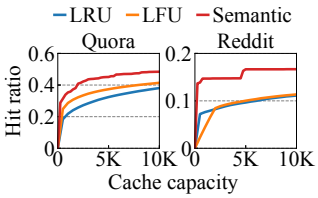


Fig. 4. Hit ratios by policies

Fig. 5. Ratio of rank change

60K vector limit; and Optimal, which retains all 600K vectors in memory and serves as an impractical oracle due to its high memory consumption. As with the training dataset, we create <question, answer> pairs for the test dataset. We replay the collected dataset by sending a sequence of the same questions to both systems based on their timestamps. The similarity threshold for retrieval, θ_R (which judges a cache hit), is set to 0.86, which is the same as θ_C for clustering.

To measure the accuracy of outputs, we calculate the cosine similarity between the hit answer and its original answer (from the test dataset). As shown in Fig. 3(a), Centroid achieves a cosine similarity of 0.85, while Optimal yields a higher similarity of 0.92, as it retains all of the vectors, using 10× more memory than Centroid. Nevertheless, Centroid's cosine similarity remains sufficiently high, showing its strong ability to capture relevant relationships. The cosine similarity of GPTCache is higher than that of Centroid, since caching individual vectors occasionally retains vectors with cosine similarities close to 1.0, inflating the average.

Impact on memory efficiency. To understand the impact of centroid caching on memory efficiency, we explore the hit ratio over the same three systems. As shown in Fig. 3(b), Centroid performs worse than Optimal, which has unlimited memory, but exhibits 1.27× and 1.14× higher hit ratios than GPTCache for Quora and Reddit, respectively. This demonstrates the superior memory efficiency of Centroid over GPTCache. To achieve comparable hit ratios, GPTCache requires 1.85× and 1.89× more memory for each dataset.

Observation #2. Caching centroids, rather than individual vectors, provides high accuracy while being substantially more memory-efficient.

3.2 Semantic Locality-aware Replacement

In the previous section, we assumed having a small yet sufficient memory to store all the centroids identified from datasets. In practice, the semantic cache size is highly constrained, so centroids to cache must be carefully chosen among many candidates. Traditional strategies like LRU or LFU can be applied, but are inefficient. Our analysis suggests that, for efficient centroid management, we must take into account another dimension of locality: *semantic locality* [5, 56].

Semantic locality. Embedding vectors of LLM queries are not uniformly distributed over the embedding space. Instead, many similar vectors are highly localized or concentrated on certain regions, creating a few dense clusters, while other areas remain comparatively sparse. Dense and sparse clusters can easily be identified through clustering. Clusters that contain many similar vectors are considered dense and exhibit strong semantic locality; thus, their centroids must have high priority for caching, and vice versa. As will be shown later, LRU or LFU, which exploit the recency or frequency of vectors, cannot capture such semantic relationships among vectors and thus fail to identify valuable vectors to cache.

To validate the above claim, we analyze the datasets collected from Quora and Reddit. We collect 60K centroids following the methodology described in §3.1. We then feed the queries from the test dataset to the system, counting the number of queries hit by the centroids using three different caching policies: Semantic, LRU, and LFU. The Semantic policy employs a static strategy: it initially fills the cache with centroids based on the number of queries they contain (a measure of semantic locality) until the cache limit is reached, without any subsequent replacements. In contrast, LRU and LFU are dynamic policies that promote centroids as they are accessed and evict a victim upon every cache miss.

Fig. 4 presents the results, with the x-axis representing the cache capacity, defined as the number of cached centroids. As shown, Semantic consistently outperforms both LRU and LFU in hit ratio across both datasets. This result is driven by two core distinctions: unlike heuristic policies, Semantic leverages semantic locality and performs no replacements. Together, these lead to the hypothesis that semantic popularity is largely stable over time: since a high semantic locality means that it contains a large number of related queries, which is an indication of the long-term popularity of such a centroid, it is valuable to retain them without eviction.

Centroid robustness according to popularity. To validate the previously set hypothesis, we conduct an experiment measuring the temporal stability of centroid popularity. Specifically, we track the rank of each centroid over four weeks and calculate the fraction of centroids whose rank change remained within 1%, 10%, 50%, and 100% of the total.

As shown in Fig. 5, 26.5% and 78.8% of the centroids in Quora and Reddit show less than 1% rank variation. Moreover, 96.1% of centroids change their rank by no more than

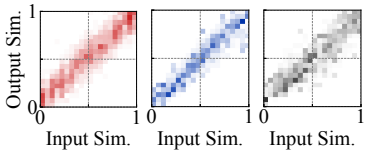


Fig. 6. Heatmap of input/output cosine similarities

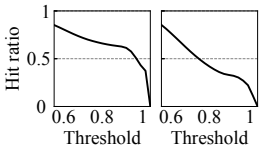


Fig. 7. Cache hit ratios with varying θ_R

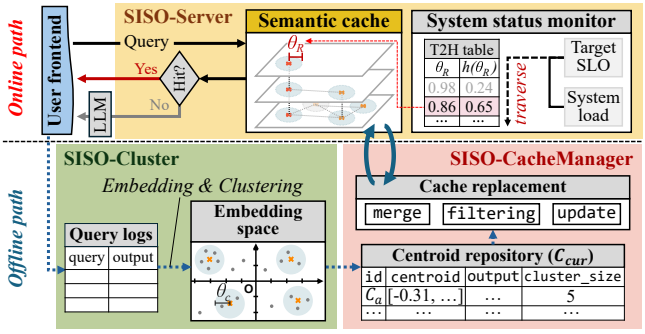


Fig. 8. Implementation overview of SISO

10%, and only a minority (fewer than 4%) experience substantial shifts. These results confirm that semantic locality remains largely stable with only gradual fluctuations over time. In practice, if we cache 10% of the total centroids, only 2.4% would require replacement to maintain an ideal hit ratio. Thus, aggressive replacement is unnecessary and misaligned with the long-term stability of centroid popularity.

Observation #3. Since semantic locality remains relatively stable over time, replacements should be triggered only occasionally by monitoring long-term query behaviors.

3.3 Dynamic Threshold Adjustment

SISO adjusts the similarity threshold θ_R dynamically depending on the intensity of workloads to balance the trade-off between the quality of responses and computational efficiency. This threshold adjustment is feasible because of the linear relationship between input and output similarities.

To understand such a relationship, we analyze the QQP, MRPC, MQP, Quora, and Reddit datasets we used in §3.1 and §3.2. For each dataset, we generate answers using the LLaMa-3.1-8B model, creating pairs of <question, answer> as done previously. We randomly choose two <question, answer> pairs, and then compute (*i*) the input cosine similarity between the questions and (*ii*) the output cosine similarity between the answers. Fig. 6 shows the relationship between input and output cosine similarities using a heatmap. The x- and y-axis represent the input and output similarities, respectively. The color intensity in the heatmap corresponds to point density, with darker regions indicating higher densities of <question, answer> pairs. As depicted in Fig. 6, most data points align closely with the diagonal, showing a strong positive correlation between input and output similarities.

Observation #4. There is a likelihood that semantically similar input queries (questions) will produce semantically similar outputs (answers).

Impact of θ_R on hit ratio. Observation #4 indicates that the quality of outputs can be controlled by adjusting θ_R. Setting θ_R high (e.g., 0.98) ensures that only the centroids highly close to input queries are selected, thereby producing outputs with high accuracy. This, however, causes more frequent cache misses due to a stricter similarity requirement, which in turn increases the computational load on the LLM serving system. Conversely, setting θ_R low (e.g., 0.60) increases the cache hit ratio, reducing the load on the LLM serving system. However, this leads to a significant drop in the quality of responses due to lower-similarity results.

Fig. 7 shows cache hit ratios for θ_R varying from 0.60 to 0.99, using Quora and Reddit described in §3.1. At θ_R = 0.98, the cache hit ratio drops to 0.24, meaning that 76% of input queries are sent to the LLM serving system. In contrast, lowering θ_R to 0.60 increases the hit ratio to 0.85, reducing the LLM query traffic to 15%. These results highlight the critical role of θ_R in balancing performance and accuracy: higher thresholds preserve response quality but increase load on the LLM, whereas lower thresholds have the opposite effect.

Observation #5. The similarity threshold is directly correlated to the output quality, but is inversely correlated to the hit ratio; thus it must be managed carefully considering the trade-off between output quality and latency.

4 Implementation of SISO

We now describe the implementation details of SISO shown in Fig. 8, focusing on how we design SISO based on the key observations from §3. SISO adopts a decoupled architecture with two paths, an *online path* for latency-intensive query serving and an *offline path* for compute-intensive background tasks, ensuring that background tasks do not interfere with online query serving. The offline path consists of *SISO-Cluster* for query clustering and *SISO-CacheManager* for semantic cache management, while the online path is handled by *SISO-Server*, which serves incoming queries in real time, adjusting the retrieval threshold θ_R according to workloads. §4.1–§4.3 detail these three components.

4.1 Clustering: From Queries to Centroids

SISO-Cluster is responsible for extracting centroids from historical query logs by converting queries into embedding vectors, clustering them, and choosing a representative centroid for each cluster. A selected centroid is stored in the

Table 1. Candidate embedding models

Model name	Median cosine sim.		Gap	Latency (ms)
	Dup.	Non-dup.		
all-mpnet-base-v2	0.59	0.89	0.30	4.94
multi-qa-distilbert-cos-v1	0.59	0.88	0.29	3.27
all-distilroberta-v1	0.58	0.86	0.28	2.96
paraphrase-albert-small-v2	**0.59**	**0.86**	**0.27**	**2.63**

Table 2. Comparison of various clustering algorithms

Model name	Time (s)	Minimum cos sim.	Average cos sim.
Community Detection	**41.44**	**0.80**	**0.99**
OPTICS	34876.32	0.48	0.95
DBSCAN	82.15	0.39	0.97
HDBSCAN	29937.20	-0.21	0.60

centroid repository, along with associated metadata (e.g., a generated output and the number of queries it represents). These centroids in the repository are considered candidates for promotion to the semantic cache.

To find high-quality centroids, it is preferable to use powerful embedding and clustering models. However, generating embedding vectors and forming clusters are compute-intensive tasks. Moreover, those tasks must be repeated regularly as new queries accumulate. To avoid interfering with online serving, SISO-Cluster should run on isolated machines, which requires additional computing resources and thus raises operational costs. Therefore, it is crucial to choose embedding and clustering models that can balance cost efficiency with centroid quality.

Selecting an embedding model. To identify a suitable embedding model, we consider four candidates [57–60] in Table 1. Our goal is to find one that achieves short computation times with high accuracy in distinguishing duplicate queries. Using the QQP dataset, we measure (*i*) the median cosine similarities of vectors extracted from duplicate and non-duplicate pairs and (*ii*) per-query processing time on a CPU. The experiments are conducted in the same environment as in §3.1. Based on the results, we choose `paraphrase-albert-small-v2`; it not only exhibits the shortest latency, but offers a sufficiently large similarity gap (i.e., 0.27) between duplicate and non-duplicate pairs, which means it identifies duplicate queries efficiently.

Selecting a clustering algorithm. To cluster the extracted vectors, we consider four algorithms [4, 8, 20, 54] in Table 2. Since the number of clusters is unknown in our case, we exclude algorithms that require a specific cluster number as an input (e.g., K-means [42]). Using the QQP dataset with θ_C of 0.86, we measure the clustering time, the average and minimum cosine similarity between vectors in the same cluster. The average similarity reflects the overall similarity of embedding vectors in the same cluster (higher is better). Conversely, the minimum cosine similarity tells us the worst-case quality of generated vectors in the cluster (higher is better). OPTICS and HDBSCAN are excluded due to their prohibitively long runtimes, and DBSCAN, although faster, produces low minimum similarity, leading to noisy clusters. We therefore choose Community Detection that achieves the highest minimum/average similarity with the shortest execution time.

Keeping centroids in a repository. The clustering algorithm returns a unique identifier `id` of each cluster, along with its centroid, `centroid`. For each cluster, we maintain two additional elements, `output` and `cluster_size`, in the repository. `output` is the output text for the centroid. We use the output text of the input query that is closest to the centroid. `cluster_size` is the number of vectors in the cluster, which represents the degree of semantic locality. It is later used as a metric when choosing centroids to cache.

Re-clustering. When and how often to trigger clustering is also a crucial issue due to its computational overhead. Initially, SISO-Cluster generates centroids by analyzing a long-term history of LLM queries (e.g., one year). Medium-sized serving systems handle about 1B queries annually [9, 69]. Embedding and clustering 1B queries take about three days on an AWS p3.2xlarge instance with one V100 GPU, costing under $300 [12], which is reasonable, considering the long timespan. Once the initial centroids are established, SISO-Cluster performs re-clustering periodically to reflect the shifting semantics of new queries. As shown in Observation #3, this process does not need to be performed frequently. In our evaluation (§5), we trigger re-clustering when the number of newly accumulated queries reaches about 10% of the initial query set (it can be adjusted depending on workload characteristics and resource availability). This approach enables us to amortize the clustering overhead while tracking changes in the semantic popularity of input queries.

4.2 Caching: Centroids Replacement

Once the repository is updated, SISO-CacheManager initiates cache replacement to refresh the semantic cache. It compares existing centroids in the semantic cache with newly discovered ones in the repository and decides which to insert, evict, or maintain. To ensure uninterrupted query serving, the semantic cache should remain active during replacement.

The operational flow of the cache replacement is detailed in Algorithm 1, which is composed of three steps: merge, filtering, and update. In the merge step, SISO-CacheManager combines the set of centroids in the semantic cache, C_{cur}, with the set of new centroids, C_{repo}, in the centroid repository and returns the newly combined set, C_{new} (Line 1). In the filtering step, it filters out centroids with low semantic locality or with less popularity from C_{new} (Line 2). In the update step, it replaces C_{cur} with C_{new}, making the semantic cache up-to-date with new centroids (Line 3).

Algorithm 1: SISO-CacheManager algorithm

Input: Current centroids C_{cur}, New centroids C_{repo}, Maximum memory capacity M, Threshold θ_C
Output: Up-to-date current centroids C_{cur}

1 $C_{new} \leftarrow$ `MergeCentroids`$(C_{cur}, C_{repo}, \theta_C)$;
2 $C_{new} \leftarrow$ `FilteringCentroids`(C_{new}, M);
3 `UpdateCentroids`(C_{cur}, C_{new});
4 **return** C_{cur};
5 **Function** *MergeCentroids*$(C_{cur}, C_{repo}, \theta_C)$**:**
6 $C_{new} \leftarrow C_{cur}$;
7 **foreach** $c_{repo} \in C_{repo}$ **do**
8 $c_{closest} \leftarrow$ `FindClosestCentroid`(c_{repo}, C_{new});
9 **if** `CosineSimilarity`$(c_{closest}, c_{repo}) > \theta_C$ **then**
10 $c_{closest}[cluster\_size] \leftarrow c_{closest}[cluster\_size] + c_{repo}[cluster\_size]$;
11 **else**
12 $c_{repo}[access\_count] \leftarrow \infty$;
13 C_{new}.add(c_{repo});
14 **return** C_{new};
15 **Function** *FilteringCentroids*(C_{new}, M)**:**
16 **while** `TotalMemoryUsage`$(C_{new}) > M$ **do**
17 Sort C_{new} by ($cluster\_size$, $access\_count$) in ASC order;
18 Remove the first element from C_{new};
19 **foreach** $c \in C_{new}$ **do**
20 $c[cluster\_size] \leftarrow c[cluster\_size]/1.1$;
21 $c[access\_count] \leftarrow 0$;
22 **return** C_{new};

Merge step. SISO-CacheManager first creates C_{new} by copying current centroids from C_{cur} to C_{new} (Line 6). For each centroid c_{repo} in C_{repo}, it searches for the closest centroid $c_{closest}$ in C_{new} based on cosine similarity (Line 8). If the similarity between the centroids exceeds the clustering threshold, θ_C, it means that the two represent the identical cluster. Thus, we increase the cluster size, *cluster_size*, of $c_{closest}$ by adding that of c_{repo} (Lines 9–10). Recall that *cluster_size* is the number of vectors represented by a centroid, reflecting semantic locality. If the similarity is below θ_C, c_{repo} is treated as a new centroid and thus added to C_{new}. Each centroid has an access count field, *access_count*, to count how many times the centroid is referenced while in the semantic cache. It reflects the short-term popularity of centroids. Initially, *access_count* of c_{repo} is set to ∞ (Lines 12–13), which is to prioritize a new centroid c_{repo} over old ones. Finally, SISO-CacheManager returns C_{new} (Line 14). During the merge step, SISO utilizes both metrics, *cluster_size* and *access_count*, to ensure balanced query retention, effectively capturing both historical significance and recent popularity fluctuations.

Filtering step. If the resulting centroid set, C_{new}, exceeds the cache capacity, SISO-CacheManager performs the filtering step to remove low-priority centroids from C_{new} (Line 16). As discussed in §3.2, the semantic locality is a key criterion for deciding which centroids to evict. SISO-CacheManager thus finds and removes the centroid with the smallest *cluster_size* from C_{new}. If multiple centroids have the same cluster size, then we consider the popularity of centroids, *access_count*. That is, the one with the smallest *access_count* is removed (Lines 17-18). After resolving memory constraints, SISO-CacheManager scales *cluster_size* of c_{new} down by 10% (Line 20). This reduction causes centroids that receive fewer new queries over time to gradually shrink in size, making them likely to be replaced. Additionally, the access counts of all centroids are reset to zero (Line 21), and the updated centroid set, C_{new}, is then returned as the output (Line 22).

Update step. Finally, SISO-CacheManager begins replacing the old centroids, c_{cur} in C_{cur}, with the new centroids, c_{new} in C_{new}. Replacing all the centroids at once can cause serious locking overhead, negatively impacting the online path. To keep semantic caching available during updates, SISO-CacheManager progressively replaces small groups of centroids one at a time. This avoids long blocking periods on the serving path and maintains consistent cache behavior throughout the replacement process.

4.3 Serving: SLO-aware Query Serving

SISO-Server is in charge of serving input queries. While incoming queries are placed into a FIFO queue, SISO-Server fetches a batch of queries from the queue and searches the semantic cache to identify similar queries. Then, only the queries that aren't present in the cache are sent to the LLM. To satisfy the target SLO, SISO-Server also adjusts θ_R to trade output quality for performance. Note that users dissatisfied with cached responses may resubmit a similar query repeatedly. To resolve this, SISO detects repeated queries from the same user and routes them directly to the LLM.

Searching for vectors in semantic cache. Like other semantic caching systems, SISO-Server uses the Hierarchical Navigable Small World (HNSW) [43] algorithm to retrieve similar vectors from the cache. To enhance search performance, we optimize HNSW to exploit semantic locality. In its standard form, HNSW randomly arranges vectors into a hierarchy of levels: upper levels contain only a small subset of vectors, while lower levels hold an increasingly larger collection. As the search moves downward, it visits progressively larger sets of vectors, enabling early termination of the search when a match is found in higher levels.

SISO-Server takes advantage of this hierarchical structure by placing centroids with strong semantic locality in the higher levels. Since these centroids are likely to serve the majority of input queries, positioning them higher enables SISO-Server to quickly locate similar vectors earlier, without descending to lower levels.

Adjusting θ_R to meet SLO. To find the most suitable θ_R, SISO-Server estimates the average waiting time W of the LLM system using the queuing theory. If W is longer than the desired SLO latency, we reduce the value of θ_R to shorten W with increased hit ratios, and vice versa.

Table 3. Information of datasets

Dataset	# of queries	Avg. # of tokens	% of simple queries	% of complex queries
MSMARCO	1M	7	95.1%	4.9%
NQ	310K	9	95.9%	4.1%
Quora	600K	12	93.1%	6.9%
Reddit	631K	14	56.9%	43.1%
ShareGPT	92K	112	53.4%	46.6%

To estimate the expected waiting time using the queuing theory, we make three reasonable assumptions: (1) requests arrive following a Poisson process, which is a common model for random arrival times of requests. (2) serving time is stable with small fluctuation and thus deterministic. (3) requests are processed in a FIFO manner. Based on these assumptions, we model the LLM serving system as an $M/D/1$ queue, where M denotes a Poisson arrival process, D represents deterministic service times, and 1 indicates a single server. In the M/D/1 model, W can be expressed as follows:

$$W = E + \frac{\lambda E^2}{2(1-\lambda E)}, \tag{1}$$

where E is the query serving time and λ denotes the arrival rate of queries. In our system, E is decided by two components: the latencies of semantic cache (on a cache hit) and the LLM serving system (on a cache miss). The latency of the semantic cache is assumed to be zero because it is much shorter than that of the LLM system. Thus, E can be defined as $E = L(1 - h(\theta_R))$, where L is the average serving time of the LLM system and $h(\theta_R)$ is a cache hit ratio, which varies by θ_R. Consequently, the average waiting time W can be rewritten as follows:

$$W = L(1-h(\theta_R)) + \frac{\lambda L^2 (1-h(\theta_R))^2}{2(1-\lambda L(1-h(\theta_R)))}. \tag{2}$$

In Eq. 2, L is measurable from the LLM system and λ can be obtained by monitoring the arrival times of input queries. Currently, we update λ every ten seconds. Our objective is to find the highest θ_R (or the lowest $h(\theta_R)$) that satisfies $S > W$, where S is the desired SLO latency.

Unfortunately, $h(\theta_R)$ is unknown and varies depending on workloads. To estimate the expected hit ratio $h(\theta_R)$ by θ_R, SISO constructs a threshold-to-hit-ratio (T2H) table that maps θ_R to a corresponding hit ratio $h(\theta_R)$. The T2H table is built during the clustering process; once the clustering finishes and the semantic cache is updated, SISO samples 5% of the newly arrived queries from the query log at random. Then, while varying θ_R from 0.98 to 0.6, SISO performs test lookups on the semantic cache using sample queries and measures hit ratios. Constructing the T2H table typically takes only a few minutes, and thus its overhead is minimal.

Using the T2H table, SISO-Server can promptly find the most suitable θ_R at runtime. However, the estimated W could differ significantly from the actual $W^{'}$. If the error exceeds 10%, SISO-Server adjusts θ_R further. For example, if W' is much longer than W, SISO-Server lowers θ_R to reduce the query traffic to the LLM system.

5 Evaluation

5.1 Experimental Setup

We evaluate SISO on a GPU server equipped with an Intel Core i5-8600K CPU, 1.5TB DRAM, 28TB NVMe SSD, and eight 48GB A6000 GPUs. vLLM is used as a baseline LLM serving system. We implement SISO from scratch and run it as a separate module in the same machine with vLLM. vLLM utilizes GPUs for inference, while SISO uses CPUs and DRAM. We use two LLM models [18]: a smaller LLaMa-3.1-8B, and a larger LLaMa-3.1-70B. Following a common practice [48], the target SLO latency is set to 1.3× the E2E serving latency, which is defined as $TTFT + TBT \times (\#of generated tokens - 1)$.

We use various real-world datasets to demonstrate a broad applicability of SISO (see Table 3 for detail). Besides the Quora and Reddit datasets used throughout §3.1– §3.3, we include MSMARCO [47] and Natural Questions (NQ) [35] with relatively short input tokens, as well as ShareGPT [65] with much longer input queries. By employing datasets with diverse token lengths and query characteristics, such as simple (e.g., information seeking) and complex queries (e.g., coding), we ensure that our results are not confined to specific dataset types. For more information, please refer to Appendix A. Unless otherwise stated, we utilize 95% of each dataset for training and the remaining 5% for testing.

We configure vLLM so that the lengths of outputs for queries follow the distribution of outputs' lengths observed in ShareGPT. This is for fair comparison across systems: longer generations incur higher latency in vLLM, whereas cached responses are returned instantly regardless of length. Aligning the output length distribution prevents bias in favor of caching systems. To evaluate various workloads with different variability, we simulate request arrivals using a Poisson process, adjusting both the average arrival rate and the coefficient of variation (CV). We also simulate the intensity of workloads by varying requests-per-second (RPS).

Each dataset contains a different number of queries, making it impractical to evaluate all systems with a fixed capacity. For this reason, we scale the semantic cache size in proportion to the dataset size. Since there are no specific guidelines for cache size, we set the cache capacity to accommodate 6% of the embedding vectors from each dataset. This number is conservatively chosen by referring to practical LLM deployments. Large-scale services such as ChatGPT are estimated to have processed over 38TB of query logs [17]. 6% of such queries would require approximately 2.3TB of memory – an amount well within the capability of high-end inference clusters or vector database systems [3, 23, 51]. We also perform evaluation while varying the semantic cache size.

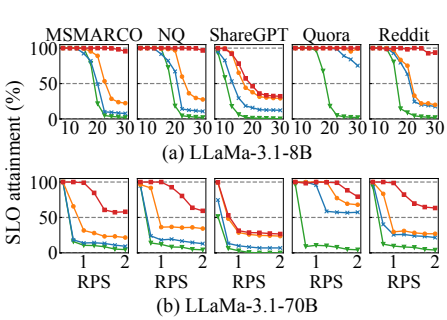


Fig. 9. Impact of RPS on SLO attainment

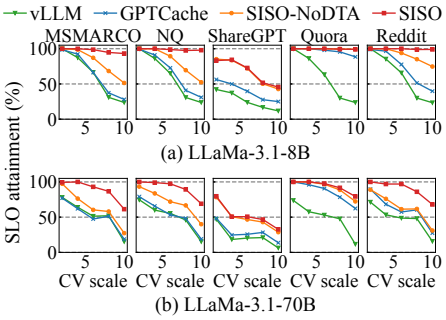


Fig. 10. Impact of CV on SLO attainment

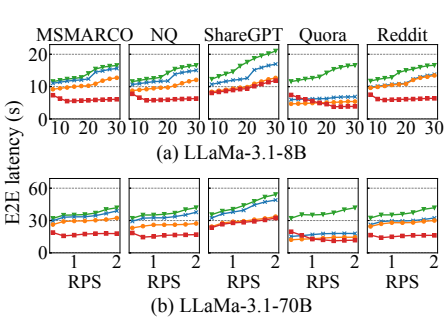


Fig. 11. Impact of RPS on E2E latency

We compare four different systems: vLLM, GPTCache, SISO without dynamic threshold adjustment, denoted by SISO-NoDTA, and SISO. vLLM is the LLM serving system that does not employ any semantic caching. GPTCache is the SOTA semantic caching system that manages individual embedding vectors using LRU. It uses vLLM as an underlying LLM serving system. We include SISO-NoDTA to understand the impact of dynamic threshold adjustment on performance. SISO employs all the techniques we explained in §3 and §4. For both GPTCache and SISO-NoDTA, θ_R is fixed to 0.86, while SISO dynamically adjusts it.

5.2 Experimental Results

In this subsection, we present results on SLO attainments under varying workload intensity (§5.2.1) and variability (§5.2.2), analysis of serving latency (§5.2.3–§5.2.4), impact of cache size on hit ratio (§5.2.5), and response quality (§5.2.7).

5.2.1 Impact of Workload Intensity on SLO. We first evaluate the SLO attainment rate of the four systems as RPS increases. We set the CV to a low value, 0.1, to minimize the variability in request patterns. The results are shown in Fig. 9. In MSMARCO and NQ, SISO-NoDTA exhibits higher SLO attainment rates than GPTCache, highlighting the effectiveness of the centroid-based caching over LRU. However, at approximately 20 RPS, its SLO attainment declines due to the rising query volume. In contrast, SISO maintains a stable and high attainment even under heavy load – satisfying the target SLO at RPS close to 30 – by dynamically adjusting θ_R to increase the likelihood of cache hits and reduce the number of queries sent to the LLMs.

In Quora, SISO-NoDTA and GPTCache show high SLO attainment rates as well. This is due to high semantic cache hit ratios that reach 50%. In contrast, in ShareGPT, the SLO attainment sharply drops as RPS increases for all the systems. This stems from ShareGPT queries containing longer input tokens than other workloads (see Table 3), which increases KV cache usage during inference and consequently reduces the number of requests that can be processed per second.

For LLaMa-3.1-70B with the same GPU configuration, the SLO attainment rate drops at lower RPS compared to the smaller model. This is because the larger LLaMa-3.1-70B model significantly increases resource demands for LLM inference, thereby limiting the number of requests it can handle at the same RPS. Despite this, SISO achieves better SLO attainment rates than other systems. This demonstrates the effectiveness of SISO even for large-scale models. As the trends observed in Fig. 9 align with results from other experiments, we omit detailed explanations for LLaMa-3.1-70B unless otherwise noted.

5.2.2 Impact of Workload Variability on SLO. We perform experiments while varying CV from 2 to 10 with a fixed RPS of 8 for LLaMa-3.1-8B and 0.5 for LLaMa-3.1-70B, respectively. A higher CV implies more irregular request patterns, whereas a lower CV indicates stable request patterns.

As illustrated on Fig. 10, higher CV causes short-term spikes in workload patterns that increase computational demand, thereby reducing overall SLO attainment. All systems except for SISO fail to meet the SLO under higher CV conditions. SISO successfully maintains high SLO attainment rates across all datasets other than ShareGPT, which indicates that SISO can capture sudden changes in input workloads and effectively adjust θ_R to meet SLO. In ShareGPT, the improvement on the cache hit ratio by lowering θ_R is limited to 1.36×, much lower than the other workloads that show 12.6× improvements, on average (see §6 for more explanations).

5.2.3 Impact of Workload Intensity on E2E Latency. Fig. 11 presents the changes in E2E serving latency as RPS increases. As expected, SISO exhibits the lowest latency regardless of RPS. Interestingly, at very low RPS, SISO exhibits higher latency. Particularly, in Quora that exhibits strong semantic locality, SISO shows longer latency than SISO-NoDTA. Under light workloads, SISO decides to prioritize the quality of responses by increasing θ_R, sending more input queries to the LLM. This decision is reasonable – if the LLM serving system has enough capacity to deal with input queries without violating the SLO, improving the output quality is a preferable strategy.

Table 4. Breakdown of serving latency

	vLLM	GPTCache		SISO	
		Hit	Miss	Hit	Miss
Embedding (ms)	-	2.63	2.63	2.63	2.63
Search (ms)	-	23.98	23.99	13.92	16.16
Inference (s)	11.98	-	11.91	-	11.89
Total (s)	11.98	0.027	11.94	0.017	11.91

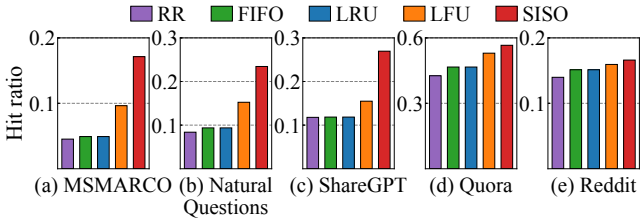


Fig. 12. Comparison of various cache replacement policies

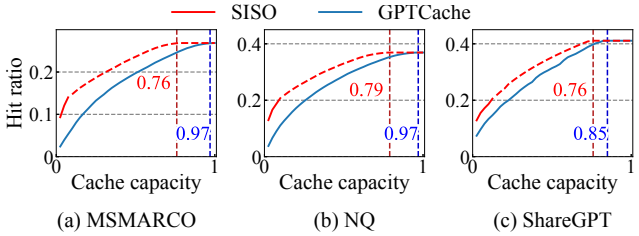


Fig. 13. Hit ratios by cache capacity

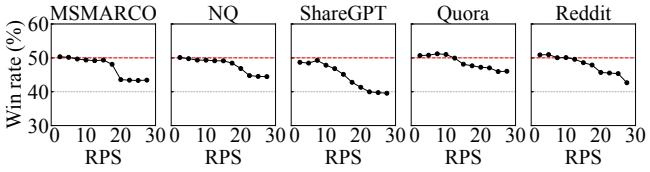


Fig. 14. Win rate of SISO against vLLM

5.2.4 Analysis of Serving Latency. Table 4 shows the breakdown of the average E2E serving latency with LLaMa-3.1-8B for SISO and GPTCache. When a request is hit by the cache, SISO and GPTCache provide very short response times, 17ms and 27ms, respectively, since heavy LLM computation that takes about 12 seconds is skipped. By using optimized HNSW where centroids with strong semantic locality are placed in higher levels and thus less traversal is needed, SISO was able to achieve 1.7× faster cache lookup performance than GPTCache on cache hit. Although semantic cache lookup adds an extra overhead in serving queries when a cache miss happens, it is negligible compared to the computational cost of the LLM and has minimal impact on LLM serving latency.

5.2.5 Analysis of Hit Ratio with Varying Cache Size. Fig. 13 compares the cache hit ratios of SISO and GPTCache depending on cache size with θ_R fixed at 0.86. SISO stores only centroid vectors in the cache, but if free space remains, SISO caches individual vectors and manages them using LRU. Since the results for Quora and Reddit are already presented in Fig. 4, we only present MSMARCO, NQ, and ShareGPT.

Under constrained cache sizes, SISO achieves a higher hit ratio than GPTCache. For example, in MSMARCO, SISO provides the hit ratio of 15% with the cache capacity of 10%, which can only be accomplished by GPTCache when its cache capacity is 3× that of SISO. This is since SISO can identify clusters with strong semantic locality and cache small-sized centroids. In contrast, GPTCache ignores semantic relationships of vectors, which leads to premature eviction of important vectors. As cache capacity increases, both systems eventually reach their maximum achievable hit ratios. However, SISO attains this peak earlier than GPTCache, thus requiring significantly less memory.

5.2.6 Analysis of Various Replacement Policies. We evaluate the impact of various cache replacement policies on cache hit ratios. We implement round-robin (RR), FIFO, and LFU policies in GPTCache and compare their hit ratios with SISO. Note that GPTCache employs LRU as its default policy. The cache capacity is set to 6% of the dataset size.

Fig. 12 demonstrates that centroid-based caching outperforms the other replacement policies, achieving 43% improvement on hit ratios than the next best policy, LFU, on average. These results confirm our claim: selecting centroids through the clustering process based on the long-term history of LLM serving is more effective than relying solely on the recency, frequency, or sequence of input queries.

5.2.7 Analysis of Response Quality. We now evaluate the quality of outputs from SISO. For our assessment, we use Alpaca-eval [19], which is a widely used benchmark tool for LLM response quality evaluation. It uses GPT-4 to compare the quality of outputs between a target model and a reference model: we use vLLM as the reference and SISO as the target, both based on LLaMa-3.1-8B.

Fig. 14 illustrates the Win rates of SISO under varying RPS, where a Win rate indicates the percentage of cases where SISO's outputs are judged superior. A Win rate near 50% indicates that SISO produces high-quality outputs comparable to vLLM. At low RPS, SISO shows Win rate close to 50%. As RPS increases, the Win rate of SISO begins to drop as it trades accuracy for performance. However, even at a high RPS close to 30, SISO maintains sufficiently high Win rates – 42.2% – on average. This implies that SISO can gain efficiency with minimal impact on response quality.

To quantitatively evaluate the response quality, we calculate F1 scores by comparing each system's output against GPT-4 responses, which serve as reference answers. Unlike Win rates, which are a preference-based measure, the F1

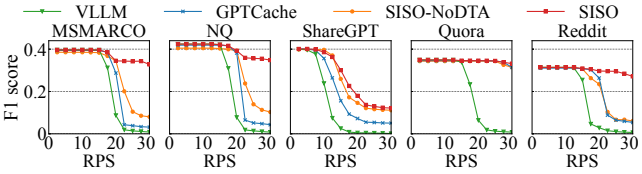


Fig. 15. Comparison of workload F1 scores by RPS

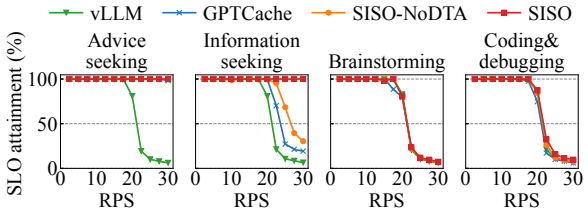


Fig. 16. SLO attainment across categories

score directly quantifies output quality by counting token-level overlaps. To reflect user-perceived quality, SLO-violating requests are scored as 0, as they receive HTTP 429 ("Too Many Requests") without any meaningful responses [53].

Fig. 15 shows F1 scores by dataset and RPS. At low RPS, SISO performs comparably to the other techniques with no or negligible F1 score drops. As RPS increases, SISO gradually lowers θ_R to reduce requests to the LLM and suppress SLO violations, leading to a higher F1 score than other systems. On average, SISO achieves 1.17x, 1.26x, and 1.71x higher F1 score than SISO-NoDTA, GPTCache, and vLLM, respectively.

6 Discussion and Limitation

SISO assumes that similar inputs yield similar outputs. This assumption, however, does not hold if the output relies heavily on the context, which is the case for multi-turn queries. For instance, "What should I pick for tomorrow?" could mean dinner menus vs. travel plans depending on the context. Yet, the multi-turn queries are not dominant in either API calls (<1%) or chatbots (33%), so the impact on SISO is limited. Still, addressing such cases remains an open challenge, which could be interesting future work.

Likewise, queries where minor input changes significantly alter the output (e.g., coding and debugging) limit the effectiveness of semantic caching. This explains SISO's lower performance on ShareGPT (§5.2.1–§5.2.3), which contains such queries, including coding, debugging, and brainstorming. To assess this deeper, we evaluate the SLO attainment with LLaMa-3.1-8B across four representative categories under increasing RPS using the same systems and methodology as in §5.2.1. For our analysis, we construct categorized datasets by extracting queries from all the datasets used in §5 and grouping them by category using the method in [21].

As shown in Fig. 16, the observed outcomes align with the anticipated trends. SISO excels in Advice and Information seeking categories, which are predominantly single-turn and thus yield high hit ratios. Conversely, the performance gains are less pronounced for Brainstorming and Coding & debugging, where small input changes result in significant differences in outputs. Nevertheless, these complex categories account for only a minority of real-world workloads (code: 13.5%, brainstorming: 5.2%), whereas Q&A accounts for the majority (53.1%) [64]. Importantly, even under these challenging conditions, SISO consistently outperforms vLLM and GPTCache. Taken together with results from §5.2.1–§5.2.3, this demonstrates that SISO delivers robust performance across datasets that reflect real-world query distributions.

7 Conclusion

In this paper, we presented SISO, an enhanced semantic caching system. SISO employed centroid-based caching, semantic locality-aware centroid replacement, and dynamic threshold adjustment, which improved cache hit ratios and ensured consistent SLO compliance even during peak loads. Our experiments showed that SISO outperformed the SOTA systems, achieving 1.71x, on average, higher hit ratios and enhanced SLO attainment rates. These improvements were obtained while maintaining high quality of responses comparable to the reference model, vLLM.

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 *Proceedings of the Symposium on Operating Systems Design and Implementation*.

[2] Sohaib Ahmad, Hui Guan, Brian D. Friedman, Thomas Williams, Ramesh K. Sitaraman, and Thomas Woo. 2024. Proteus: A High-Throughput Inference-Serving System with Accuracy Scaling. In *Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems*.

[3] Amazon Web Services, Inc. 2025. Amazon EC2 P5 instances. https://aws.amazon.com/ec2/instance-types/p5/. Accessed 2025-08-19.

[4] Mihael Ankerst, Markus M Breunig, Hans-Peter Kriegel, and Jörg Sander. 1999. OPTICS: Ordering Points to Identify the Clustering Structure. *ACM Special Interest Group on Management of Data Record* (1999).

[5] Vanda Balogh, Gábor Berend, Dimitrios I. Diochnos, and György Turán. 2020. Understanding the Semantic Content of Sparse Word Embeddings Using a Commonsense Knowledge Base. In *Proceedings of the Association for the Advancement of Artificial Intelligence*.

[6] Fu Bang. 2023. GPTCache: An Open-Source Semantic Cache for LLM Applications Enabling Faster Answers and Cost Savings. In *Proceedings of the Workshop for Natural Language Processing Open Source Software*.

[7] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models Are Few-shot Learners. In *Proceedings of the International Conference on Neural Information Processing Systems*.

[8] Ricardo JGB Campello, Davoud Moulavi, and Jörg Sander. 2013. Density-based Clustering Based on Hierarchical Density Estimates. In *Proceedings of the Conference on Knowledge Discovery and Data Mining*.

[9] CBOT.ai. 2025. CBOT-LLM. https://www.cbot.ai/cbot-llm/.

[10] Harrison Chase. 2022. LangChain. https://github.com/langchain-ai/langchain.

[11] Chungmin Melvin Chen and Nicholas Roussopoulos. 1994. The Implementation and Performance Evaluation of the ADMS Query Optimizer: Integrating Query Result Caching and Matching. In *Proceedings of the International Conference on Extending Database Technology: Advances in Database Technology*.

[12] Han-Yi Chou and Sayan Ghosh. 2023. Batched Graph Community Detection on GPUs. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*.

[13] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and Memory-efficient Exact Attention with IO-awareness. In *Proceedings of the International Conference on Neural Information Processing Systems*.

[14] Shaul Dar, Michael J. Franklin, Björn Þór Jónsson, Divesh Srivastava, and Michael Tan. 1996. Semantic Data Caching and Replacement. In *Proceedings of the International Conference on Very Large Data Bases*.

[15] Redis Developers. 2025. Redis. http://redis.io/.

[16] Bill Dolan and Chris Brockett. 2005. Automatically Constructing a Corpus of Sentential Paraphrases. In *Proceedings of the International Workshop on Paraphrasing*.

[17] Fabio Duarte. 2025. Number of ChatGPT Users. https://explodingtopics.com/blog/chatgpt-users.

[18] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The Llama 3 Herd of Models. arXiv:2407.21783

[19] Yann Dubois, Balázs Galambosi, Percy Liang, and Tatsunori B. Hashimoto. 2024. Length-Controlled AlpacaEval: A Simple Way to Debias Automatic Evaluators. arXiv:2404.04475

[20] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. 1996. A Density-based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *Proceedings of the Conference on Knowledge Discovery and Data Mining*.

[21] Hugging Face. 2025. What is Zero-Shot Classification? https://huggingface.co/tasks/zero-shot-classification.

[22] Waris Gill, Mohamed Elidrisi, Pallavi Kalapatapu, Ammar Ahmed, Ali Anwar, and Muhammad Ali Gulzar. 2024. MeanCache: User-Centric Semantic Cache for Large Language Model Based Web Services. arXiv:2403.02694

[23] Google LLC. 2025. M2 Machine Series — Memory-optimized Machine Family for Compute Engine. https://cloud.google.com/compute/docs/memory-optimized-machines.

[24] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *Proceedings of the Advances in Neural Information Processing Systems*.

[25] Guseul Heo, Sangyeop Lee, Jaehong Cho, Hyunmin Choi, Sanghyeon Lee, Hyungkyu Ham, Gwangsun Kim, Divya Mahajan, and Jongse Park. 2024. NeuPIMs: NPU-PIM Heterogeneous Acceleration for Batched LLM Inferencing. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems*.

[26] Connor Holmes, Masahiro Tanaka, Michael Wyatt, Ammar Ahmad Awan, Jeff Rasley, Samyam Rajbhandari, Reza Yazdani Aminabadi, Heyang Qin, Arash Bakhtiari, Lev Kurilenko, and Yuxiong He. 2024. DeepSpeed-FastGen: High-throughput Text Generation for LLMs via MII and DeepSpeed-Inference. arXiv:2401.08671

[27] Ke Hong, Guohao Dai, Jiaming Xu, Qiuli Mao, Xiuhong Li, Jun Liu, Kangdi Chen, Yuhan Dong, and Yu Wang. 2024. FlashDecoding++: Faster Large Language Model Inference with Asynchronization, Flat GEMM Optimization, and Heuristics. In *Proceedings of Machine Learning and Systems*.

[28] Ke Hong, Xiuhong Li, Lufang Chen, Qiuli Mao, Guohao Dai, Xuefei Ning, Shengen Yan, Yun Liang, and Yu Wang. 2025. SOLA: Optimizing SLO Attainment for Large Language Model Serving with State-Aware Scheduling. In *Proceedings of Machine Learning and Systems*.

[29] Coleman Hooper, Sehoon Kim, Hiva Mohammadzadeh, Michael W. Mahoney, Yakun Sophia Shao, Kurt Keutzer, and Amir Gholami. 2025. KVQuant: Towards 10 Million Context Length LLM Inference with KV Cache Quantization. In *Proceedings of the International Conference on Neural Information Processing Systems*.

[30] Cunchen Hu, Heyang Huang, Junhao Hu, Jiang Xu, Xusheng Chen, Tao Xie, Chenxi Wang, Sa Wang, Yungang Bao, Ninghui Sun, et al. 2024. Memserve: Context Caching for Disaggregated LLM Serving with Elastic Memory Pool. arXiv:2406.17565

[31] Jinqi Huang, Yi Xiong, Xuebing Yu, Wenjie Huang, Entong Li, Li Zeng, and Xin Chen. 2025. SLO-Aware Scheduling for Large Language Model Inferences. arXiv:2504.14966

[32] Shankar Iyer, Nikhil Dandekar, and Kornel Csernai. 2017. First Quora Dataset Release: Question Pairs. https://www.quora.com/q/quoradata/First-Quora-Dataset-Release-Question-Pairs.

[33] Jinwoo Jeong and Jeongseob Ahn. 2025. Accelerating LLM Serving for Multi-turn Dialogues with Efficient Resource Management. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems*.

[34] A.M. Keller and J. Basu. 1994. A Predicate-based Caching Scheme for Client-server Database Architectures. In *Proceedings of International Conference on Parallel and Distributed Information Systems*.

[35] Tom Kwiatkowski, Jennimaria Palomaki, Olivia Redfield, Michael Collins, Ankur Parikh, Chris Alberti, Danielle Epstein, Illia Polosukhin, Jacob Devlin, Kenton Lee, Kristina Toutanova, Llion Jones, Matthew Kelcey, Ming-Wei Chang, Andrew M. Dai, Jakob Uszkoreit, Quoc Le, and Slav Petrov. 2019. Natural Questions: A Benchmark for Question Answering Research. *Transactions of the Association for Computational Linguistics* (2019).

[36] 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 Symposium on Operating Systems Principles*.

[37] Sanghyeon Lee, Hongbeen Kim, Soojin Hwang, Guseul Heo, Minwoo Noh, and Jaehyuk Huh. 2025. Efficient LLM Inference with Activation Checkpointing and Hybrid Caching. arXiv:2501.01792

[38] Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. 2024. InfiniGen: Efficient Generative Inference of Large Language Models with Dynamic KV Cache Management. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*.

[39] Jiaxing Li, Chi Xu, Feng Wang, Isaac M von Riedemann, Cong Zhang, and Jiangchuan Liu. 2024. SCALM: Towards Semantic Caching for Automated Chat Services with Large Language Models. arXiv:2406.00025

[40] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. AlpaServe: Statistical Multiplexing with Model Parallelism for Deep Learning Serving. In *Proceedings of the Symposium on Operating Systems Design and Implementation*.

[41] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. AlpaServe: Statistical Multiplexing with Model Parallelism for Deep Learning Serving. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*.

[42] James MacQueen et al. 1967. Some Methods for Classification and Analysis of Multivariate Observations. In *Proceedings of the Symposium on Mathematical Statistics and Probability*.

[43] Yu A. Malkov and D. A. Yashunin. 2020. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2020).

[44] Clara H McCreery, Namit Katariya, Anitha Kannan, Manish Chablani, and Xavier Amatriain. 2020. Effective Transfer Learning for Identifying Similar Questions: Matching User Questions to COVID-19 FAQs. In *Proceedings of the Conference on Knowledge Discovery & Data Mining*.

[45] Xupeng Miao, Chunan Shi, Jiangfei Duan, Xiaoli Xi, Dahua Lin, Bin Cui, and Zhihao Jia. 2024. SpotServe: Serving Generative Large Language Models on Preemptible Instances. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems*.

[46] Microsoft. 2025. Prompt caching with Azure OpenAI in Azure AI Foundry Models. https://learn.microsoft.com/en-us/azure/ai-foundry/openai/how-to/prompt-caching.

[47] T. Nguyen. 2016. MS MARCO: A Human Generated Machine Reading Comprehension Dataset. arXiv:1611.09268

[48] Chengyi Nie, Rodrigo Fonseca, and Zhenhua Liu. 2024. Aladdin: Joint Placement and Scaling for SLO-Aware LLM Serving. arXiv:2405.06856

[49] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. 2013. Scaling Memcache at Facebook. In *Proceedings of the Symposium on Networked Systems Design and Implementation*.

[50] NVIDIA. 2023. TensorRT-LLM: A TensorRT Toolbox for Optimized Large Language Model Inference. https://github.com/NVIDIA/TensorRT-LLM.

[51] NVIDIA Corporation. 2025. NVIDIA DGX H100/H200 System User Guide. https://docs.nvidia.com/dgx/dgxh100-user-guide/index.html.

[52] Ramya Prabhu, Ajay Nayak, Jayashree Mohan, Ramachandran Ramjee, and Ashish Panwar. 2025. vAttention: Dynamic Memory Management for Serving LLMs without PagedAttention. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems*.

[53] Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. 2025. Mooncake: Trading More Storage for Less Computation — A KVCache-centric Architecture for Serving LLM Chatbot. In *Proceedings of the USENIX Conference on File and Storage Technologies*.

[54] Nils Reimers. 2019. Sentence-Transformers Fast Clustering Algorithm. https://github.com/UKPLab/sentence-transformers.

[55] Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*.

[56] Lütfi Kerem Şenel, Ihsan Utlu, Veysel Yücesoy, Aykut Koc, and Tolga Cukur. 2018. Semantic Structure and Interpretability of Word Embeddings. *IEEE/ACM Transactions on Audio, Speech, and Language Processing* (2018).

[57] Sentence-Transformers. 2021. all-distilroberta-v1. https://huggingface.co/sentence-transformers/all-distilroberta-v1.

[58] Sentence-Transformers. 2021. all-mpnet-base-v2. https://huggingface.co/sentence-transformers/all-mpnet-base-v2.

[59] Sentence-Transformers. 2021. multi-qa-distilbert-cos-v1. https://huggingface.co/sentence-transformers/multi-qa-distilbert-cos-v1.

[60] Sentence-Transformers. 2021. paraphrase-albert-small-v2. https://huggingface.co/sentence-transformers/paraphrase-albert-small-v2.

[61] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. 2019. Nexus: A GPU Cluster Engine for Accelerating DNN-based Video Analysis. In *Proceedings of the Symposium on Operating Systems Principles*.

[62] 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 *Proceedings of the International Conference on Machine Learning*.

[63] Yixin Song, Zeyu Mi, Haotong Xie, and Haibo Chen. 2024. PowerInfer: Fast Large Language Model Serving with a Consumer-grade GPU. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles*.

[64] Tianxiang Sun, Xiaotian Zhang, Zhengfu He, Peng Li, Qinyuan Cheng, Xiangyang Liu, Hang Yan, Yunfan Shao, Qiong Tang, Shiduo Zhang, Xingjian Zhao, Ke Chen, Yining Zheng, Zhejian Zhou, Ruixiao Li, Jun Zhan, Yunhua Zhou, Linyang Li, Xiaogui Yang, Lingling Wu, Zhangyue Yin, Xuanjing Huang, Yu-Gang Jiang, and Xipeng Qiu. 2024. MOSS: An Open Conversational Large Language Model. *Machine Intelligence Research* (2024).

[65] ShareGPT Team. 2023. ShareGPT. https://huggingface.co/datasets/anon8231489123/ShareGPT_Vicuna_unfiltered.

[66] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is All You Need. In *Proceedings of the International Conference on Neural Information Processing Systems*.

[67] Jiahao Wang, Jinbo Han, Xingda Wei, Sijie Shen, Dingyan Zhang, Chenguang Fang, Rong Chen, Wenyuan Yu, and Haibo Chen. 2025. KVCache Cache in the Wild: Characterizing and Optimizing KVCache Cache at a Large Cloud Provider. In *Proceedings of the USENIX Annual Technical Conference*.

[68] Yiding Wang, Kai Chen, Haisheng Tan, and Kun Guo. 2023. Tabi: An Efficient Multi-Level Inference System for Large Language Models. In *Proceedings of the European Conference on Computer Systems*.

[69] Yuxin Wang, Yuhan Chen, Zeyu Li, Xueze Kang, Zhenheng Tang, Xin He, Rui Guo, Xin Wang, Qiang Wang, Amelie Chi Zhou, and Xiaowen Chu. 2024. BurstGPT: A Real-world Workload Dataset to Optimize LLM Serving Systems. arXiv:2401.17644

[70] Haojun Xia, Zhen Zheng, Xiaoxia Wu, Shiyang Chen, Zhewei Yao, Stephen Youn, Arash Bakhtiari, Michael Wyatt, Donglin Zhuang, Zhongzhu Zhou, Olatunji Ruwase, Yuxiong He, and Shuaiwen Leon Song. 2024. Quant-LLM: Accelerating the Serving of Large Language Models via FP6-Centric Algorithm-System Co-Design on Modern GPUs. In *Proceedings of the USENIX Annual Technical Conference*.

[71] Jiayi Yao, Hanchen Li, Yuhan Liu, Siddhant Ray, Yihua Cheng, Qizheng Zhang, Kuntai Du, Shan Lu, and Junchen Jiang. 2025. CacheBlend: Fast Large Language Model Serving for RAG with Cached Knowledge Fusion. In *Proceedings of the European Conference on Computer Systems*.

[72] Yiling-J. 2019. cacheme: Asyncio cache framework with multiple cache storages. https://github.com/Yiling-J/cacheme.

[73] 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 *Proceedings of the Symposium on Operating Systems Design and Implementation*.

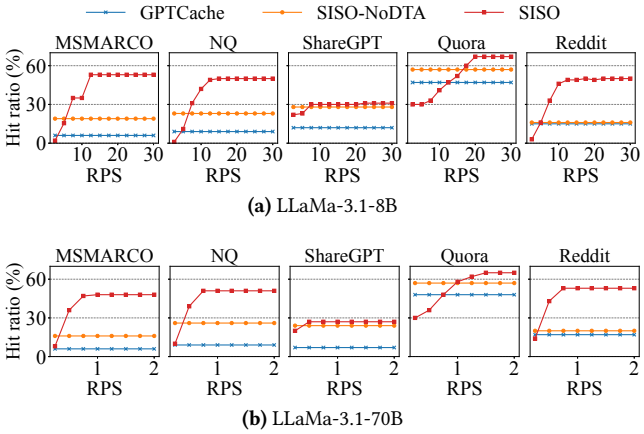
[74] Shan Yu, Jiarong Xing, Yifan Qiao, Mingyuan Ma, Yangmin Li, Yang Wang, Shuo Yang, Zhiqiang Xie, Shiyi Cao, Ke Bao, Ion Stoica, Harry Xu, and Ying Sheng. 2025. Prism: Unleashing GPU Sharing for Cost-Efficient Multi-LLM Serving. arXiv:2505.04021

[75] Wenting Zhao, Xiang Ren, Jack Hessel, Claire Cardie, Yejin Choi, and Yuntian Deng. 2024. Wildchat: 1m ChatGPT Interaction Logs in the Wild. arXiv:2405.01470

[76] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. 2024. SGLang: Efficient Execution of Structured Language Model Programs. In *Advances in Neural Information Processing Systems*.

[77] 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 *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*.

[78] Ruihang Lai Wuwei Lin Yineng Zhang Stephanie Wang Tianqi Chen Baris Kasikci Vinod Grover Arvind Krishnamurthy Luis Ceze Zihao Ye, Lequn Chen. 2025. FlashInfer: Efficient and Customizable Attention Engine for LLM Inference Serving. In *Proceedings of Machine Learning and Systems*.



(a) LLaMa-3.1-8B

(b) LLaMa-3.1-70B

Fig. 18. Impact of RPS on hit ratio

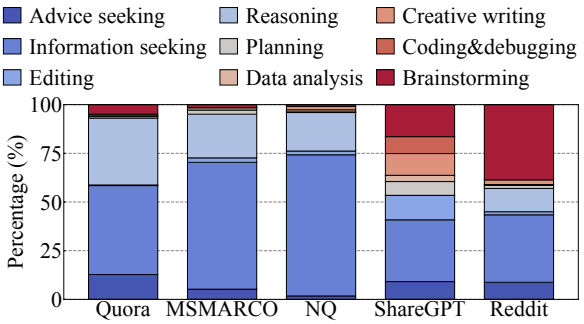


Fig. 17. Distribution of query categories across dataset

Appendices

A Category Detail for Each Dataset

In §6, we measured performance across different categories. Overall, SISO exhibited relatively poor performance on tasks such as Coding&debugging and Brainstorming. In addition, through our categorization method [21], we identified a total of nine categories, and their distribution is shown in Fig. 17. The topics in the shade of blue (Advice seeking, Information Seeking, Editing and Reasoning) are simple queries, and the ones in the shade of red (Planning, Data analysis, Creative writing, Coding&debugging and Brainstorming) are complex queries. These also correspond to the categories shown in Table 3.

In these categories, Editing and Reasoning showed experimental results similar to advice seeking and information seeking (higher performance gain) presented in Fig. 16, while data analysis and creative writing exhibited trends comparable to Coding&debugging and Brainstorming (lower performance gain). This characteristic arises, as noted earlier, in tasks where small variations in the input query can lead to significant differences in the output response.

As we mentioned, such complex tasks constitute only a small fraction of the overall workload. In this sense, SISO still demonstrates promising performance from a general workload perspective.

B Hit Ratio Varying RPS

In this section, we present the hit ratio as the RPS is varied. As other experimental results also indicated, SISO drops the hit ratio on low demand to improve the output quality. This drop only occurs when the LLM serving system has sufficient capacity for LLM execution, and it does not negatively impact the system.