

Efficient Iterative Retrieval with Heterogeneous Batching

Dohyun Park¹, Hubertus Franke², Daniel G. Waddington²,
Swaminathan Sundararaman², Yongjoo Park¹

¹University of Illinois Urbana-Champaign ²IBM Research

{dohyunp2, yongjoo}@illinois.edu

{frankeh, daniel.waddington, swami}@ibm.com

Abstract

Modern information retrieval increasingly employs both embedding and generative models to handle complex queries. However, current serving systems suffer from low throughput and poor GPU utilization because they execute these models in isolation. Coarse-grained partitioning, such as dedicating GPUs to specific tasks, fails to adapt to dynamic workloads and creates computational “bubbles”. To address these, we present ORTHRUS, a serving system that performs *heterogeneous batching* within a unified inference loop. The primary challenge lies in unifying embedding and generation workloads with conflicting computational patterns while optimizing batch composition for high performance. ORTHRUS addresses these challenges through chunked embedding with incremental pooling and by adjusting batch composition in a workload-aware manner. Evaluation on four A100 GPUs shows that, relative to baseline deployments, ORTHRUS achieves $1.28\times$ – $4.52\times$ higher throughput on controlled workloads and up to 55.8% lower end-to-end p99 latency on an iterative-RAG benchmark. We release our code at <https://github.com/illinoisdata/Orthrus>.

1 Introduction

Modern retrieval techniques (Asai et al., 2023; Gao et al., 2023; Jiang et al., 2023b; Shao et al., 2023) integrate both embedding and generation models to identify information gaps and retrieve documents iteratively. Consider a user query: “Does the Gold plan cover my gym membership?” This query is first *embedded* into a vector (Wang et al., 2024; Izacard et al., 2021) to search a vector database (Malkov and Yashunin, 2020; Li et al., 2025b). If a retrieved document states that “The Gold plan covers Category B activities,” the system clarifies the ambiguity by letting a *generative*

model (Asai et al., 2023; Shao et al., 2023) formulate a follow-up query—“What are Category B activities?”—which is then embedded to retrieve supplementary documents. This cycle continues until sufficient context has been gathered. To accelerate this, serving systems must be explicitly designed to handle mixed embedding/generation workloads.

Unfortunately, existing systems suffer from low throughput and poor GPU utilization when handling mixed workloads. Serving frameworks like vLLM (Kwon et al., 2023) are designed to execute embedding and generation models in isolation. Simply running two serving instances (i.e., OS processes) on the same GPU—one for each task, as in Fig. 1(b)—is inefficient in current architectures (Yu et al., 2022). In a multi-GPU setup, resources can be statically partitioned for embedding and generation (Fig. 1(a)), but this approach creates two major issues: (i) accurately predicting the optimal resource ratio is difficult, and (ii) dynamic repartitioning requires costly model reloading. Even with perfect workload foresight, coarse-grained GPU partitioning remains suboptimal because embedding tasks are typically compute-bound, whereas generation is memory-bound (Agarwal et al., 2025). We hypothesize that *homogeneous batching* is a fundamental bottleneck in iterative retrieval.

Core Idea We present ORTHRUS, a serving system optimized for mixed embedding and generation workloads. By unifying these distinct compute patterns into a shared abstraction, ORTHRUS enables *heterogeneous batching* by executing embedding and generation requests together in a shared model execution pass. Unlike homogeneous batching (Fig. 1(b)), this approach co-schedules tasks to maximize resource utilization (Fig. 1(c)) while eliminating the overhead of concurrent server instances. This architecture

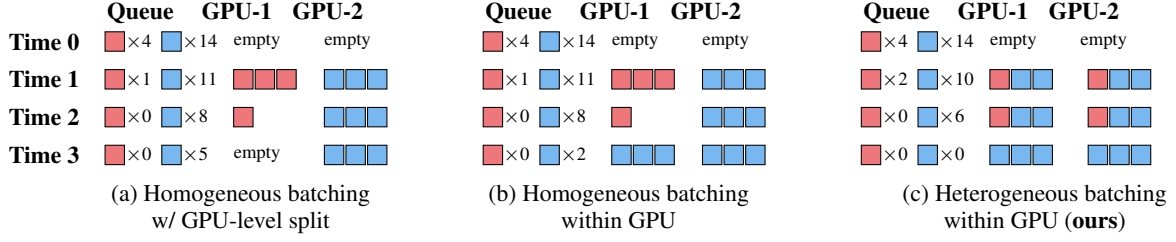


Figure 1: Example iterative retrieval. The workload arrives at Time 0. Neither (a) GPU-level splitting nor (b) homogeneous batching within each GPU completes all requests by Time 3; (c) our heterogeneous batching completes them all.

enables fine-grained load balancing and generalizes to multi-GPU settings, where every node can serve mixed workloads. Importantly, ORTHRUS requires no additional model training, making it applicable to existing deployments.

Challenge We address two primary challenges. **(Heterogeneous Workloads)** To maximize efficiency, it is desirable to execute embedding and generation requests together in the same model execution pass. Forming such heterogeneous passes is non-trivial because the requests exhibit conflicting computational patterns. A generation decode step requires relatively little compute to predict the next token, whereas embedding requires substantial compute to aggregate internal states across a full input. Naïvely batching them without chunking forces lightweight work to wait for heavier requests, causing *bubbles*. **(Batch Composition)** The ratio of embedding to generation requests in a batch heavily impacts user-perceived performance, yet finding the optimal ratio is difficult. FCFS scheduling causes head-of-line blocking for specific request types. Static heuristics (e.g., a 1:2 ratio) are also brittle because the number of tokens generated varies per request, so the optimal resource mix shifts dynamically and makes fixed ratios suboptimal.

Our Approach We employ the following techniques.

(Chunked Embedding for Compute Balance)

To balance embedding and generation work within a batch, we adapt incremental computation (Sheoran et al., 2023; Hellerstein et al., 1997) to embedding inference. We call this execution strategy *chunked embedding*: the input is processed in fine-grained token chunks across scheduling iterations, while *incremental pooling* maintains the cross-chunk state needed to produce the final embedding (§3.3). The unified runner schedules embedding chunks alongside generation decode steps

to balance compute loads and minimize *bubbles*.

(Intra-Batch Scheduling) Our design is governed by a central principle: *retrieval ends only when all embedding and generation requests are fully processed*. To achieve this, ORTHRUS dynamically adjusts batch composition using a *cost-aware queue length*. The scheduler measures each backlog by its total remaining token count rather than its raw request count, allocating capacity in proportion to the outstanding token work in each queue.

Our technique is practical and delivers high performance. We implemented our prototype system (called ORTHRUS) on top of vLLM version 0.92, without modifying memory management or model specification. Evaluation on four NVIDIA A100 GPUs with state-of-the-art open-weight models shows that ORTHRUS delivers $1.28\times$ – $4.52\times$ higher throughput than GPU-level splits. Chunked and single-pass embeddings are equivalent up to floating-point reduction-order effects (Tab. 7).

This work focuses on compute patterns inside GPU kernels, orthogonal to parameter sharing (Vasani et al., 2025; Gauthier, 2024) and multi-adaptor scheduling (Sheng et al., 2024a; Chen et al., 2024).

2 Background

2.1 Modern RAG Workflow

Modern RAG pipelines combine embedding-based retrieval with language model generation (Lewis et al., 2020). Recent techniques employ iterative retrieval, where embedding and generation alternate to refine queries dynamically (Asai et al., 2023; Jiang et al., 2023b; Shao et al., 2023; Trivedi et al., 2023). A single prompt thus decomposes into a sequence of embedding and generation requests, creating a mixed workload. ORTHRUS changes model-side embedding

Table 1: GPU utilization under pure embedding and pure generation workloads (Mistral-7B on A100-40GB, 128-token inputs, 512-token outputs for generation).

Workload	GPU Compute (%)		GPU Memory (%)	
	Mean	Max	Mean	Max
Embedding	85.3	98.0	36.5	37.1
Generation	75.5	98.0	90.5	90.5

inference but not the vector-search backend, which is orthogonal to our serving design.

2.2 Computational Characteristics of Embedding and Generation

Embedding and generation requests differ fundamentally in their execution patterns and resource utilization.

Generation A generation request produces a variable-length output autoregressively. It executes in two phases: *prefill* and *decode*. The *prefill* phase processes the input prompt in a single parallel *forward* pass. The *decode* phase then generates tokens one at a time: each step reads the key-value (KV) cache (Pope et al., 2023) containing previous tokens’ representations, computes *attention*, and appends the new token’s KV entries. This decode phase is *memory-bound*, as each step transfers large KV cache while performing relatively few arithmetic operations (Sheng et al., 2023).

Embedding An embedding request structurally resembles the *prefill* phase of generation, but follows with a pooling operation rather than entering a decode loop. The model processes input tokens through transformer layers (Vaswani et al., 2017) and aggregates the final hidden states via pooling (typically mean pooling or last-token pooling) to produce a fixed-length vector (Wang et al., 2024; Shen et al., 2024). A vector database indexes these vectors for similarity search (Malkov and Yashunin, 2020; Johnson et al., 2021; Wang et al., 2021a; Chockchowwat et al., 2023; Li et al., 2025a). Because embedding is a single dense forward pass, it is *compute-bound* (Ivanov et al., 2021), contrasting with memory-bound decoding.

Resource utilization analysis To quantify these compute differences, we profiled GPU utilization under pure embedding and pure generation workloads using Mistral-7B on an A100-40GB GPU. Tab. 1 summarizes the results. Neither workload fully exploits both resources, suggesting an op-

portunity to co-schedule them for improved efficiency.

Parameter sharing State-of-the-art embedding models are increasingly derived from generative models via LoRA fine-tuning (Li et al., 2023; Wang et al., 2024; Meng et al., 2024; Ma et al., 2024; BehnamGhader et al., 2024). Other approaches train unified models for both tasks (Muennighoff et al., 2025; Zhang et al., 2025; Li et al., 2024; Tang et al., 2025). ORTHRUS is optimized for shared-backbone embedding and generation models. For separate backbones, our extension retains IBS across model-specific queues but does not combine the models in one forward pass (Tab. 6).

2.3 Batching and Scheduling

Modern LLM serving systems employ *iteration-level scheduling* (Yu et al., 2022; Kwon et al., 2023), re-evaluating batch composition at every token generation step. Within each iteration, the system executes a *homogeneous batch* consisting exclusively of generation requests, potentially mixing chunked prefill and decode phases (Agrawal et al., 2024). ORTHRUS extends this chunked execution to embeddings via incremental pooling. Recent work targets scheduling policies (Wu et al., 2024; Sheng et al., 2024b), kernel efficiency and GPU co-location (Wang et al., 2021b; Xia et al., 2023; Strati et al., 2024), KV cache management (Qin et al., 2025; Yu et al., 2025; Gao et al., 2025; Agarwal et al., 2025; Xia et al., 2026), storage efficiency (Shah et al., 2026), and prefill-decode disaggregation (Zhong et al., 2024; Hu et al., 2024). Multi-tenant LoRA systems (Sheng et al., 2024a; Chen et al., 2024) improve adapter throughput but assume generation-only workloads. ORTHRUS schedules embedding and generation jointly while preserving arrival order within each request type (§4).

3 Unified Runner

This section presents the design of ORTHRUS’s unified runner, which enables heterogeneous batching of embedding and generation requests within a single inference loop.

3.1 High-Level Workflow

Our runner is designed for iteration-level scheduling (Yu et al., 2022), aiming for seamless integration with vLLM’s (Kwon et al., 2023) open-

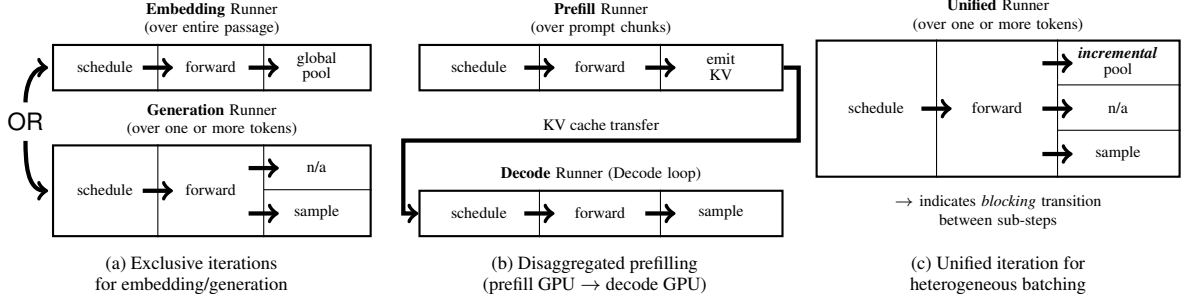


Figure 2: Runner structure comparison. (a) Task-separated runners use global pooling for embedding or token sampling for generation. (b) Phase-separated prefill and decode runners transfer the KV cache between GPUs. (c) Our unified runner supports both output paths in each iteration; incremental pooling allows embedding chunks to execute alongside generation requests.

Algorithm 1: Heterogeneous Batching Inference Loop

Input: Embedding queue Q_e , generation queue Q_g
while $Q_e \neq \emptyset$ **or** $Q_g \neq \emptyset$ **or** *in-flight requests* **do**
 $B \leftarrow \text{SCHEDULE}(Q_e, Q_g)$; // Mix embed
 & gen
 $H \leftarrow \text{FORWARD}(B)$; // Attention +
 FFN
 // Parallel output processing
 begin in parallel
 $\mathcal{R}_e \leftarrow \text{INCPOOLER}(H[\text{emb_idx}])$;
 $\mathcal{R}_g \leftarrow \text{SAMPLER}(H[\text{gen_idx}])$;
 end parallel
 $\text{EMIT}(\mathcal{R}_e, \mathcal{R}_g)$; // Return completed

sourced model management and server. Yet, our approach differs from existing approaches in that both embedding and generation requests are processed using the same runner, which we achieve by consolidating distinct embedding and generation runners into one in a structured way.

Our runner operates in three blocking stages: (1) *schedule*, (2) *forward*, and (3) *emit* (Fig. 2). *Schedule* determines batch composition, including generation prefills, decodes, or embedding. *Forward* applies attention and feed-forward layers, identical across request types except for input length. *Emit* diverges: generation samples the next token, while embedding performs *incremental pooling* to compute the final embedding vector.

3.2 Batch Unit

A batch in ORTHRUS contains three types of items: (1) *embedding chunks*, which are segments of an embedding request’s input sequence, (2) *generation prefill chunks*, which are segments of a generation request’s prompt, and (3) *generation decode tokens*, which are single tokens generated during autoregressive decoding. We call execution of an embedding request across multiple such

chunks *chunked embedding*. All three types share the same forward pass through the transformer layers, differing only in their input lengths and output handling. Embedding and prefill chunks process multiple tokens per request, while decode tokens process exactly one token per request. This unified representation allows the scheduler to mix request types freely within a single batch, filling available token capacity while preserving the workload-proportional allocation described in §4.2. The scheduler outputs batches sorted by request type, ensuring that items of the same type are contiguous in memory. This layout eliminates memory movement when invoking the output kernels: the incremental pooler, the sampler, and the prefill handler each operate on a contiguous slice of the hidden state tensor, avoiding gather operations or intermediate copies.

3.3 Incremental Pooler

Incremental pooling is the cross-chunk aggregation used by chunked embedding.

Consider an embedding request with input sequence of n tokens. Under chunked embedding, the system splits the sequence into K chunks of sizes $c_1, c_2, \dots, c_K$ where $\sum_{k=1}^K c_k = n$. At iteration k , the forward pass produces hidden states $H^{(k)} \in \mathbb{R}^{c_k \times d}$ for the k -th chunk, where d is the hidden dimension.

Pooling Heads Mean and weighted-mean pooling share the same streaming formulation. For weights $w_{k,j}$, the pooler maintains

$$S^{(k)} = S^{(k-1)} + \sum_{j=1}^{c_k} w_{k,j} H_j^{(k)}, \quad S^{(0)} = \mathbf{0}, \quad (1)$$

$$W^{(k)} = W^{(k-1)} + \sum_{j=1}^{c_k} w_{k,j}, \quad W^{(0)} = 0, \quad (2)$$

and emits $S^{(K)}/W^{(K)}$ after the final chunk. Mean pooling is the special case $w_{k,j} = 1$. Positional heads retain only the selected hidden state: CLS pooling uses $H_1^{(1)}$, while last-token pooling uses $H_{c_K}^{(K)}$. These additive and positional heads require $O(d)$ state per request. An arbitrary deterministic head remains compatible by buffering $A^{(k)} = [A^{(k-1)}; H^{(k)}]$ and applying the head after the final chunk, at an $O(nd)$ memory cost. For all of these heads, incremental and single-pass pooling are algebraically equivalent, although floating-point non-associativity may introduce small numerical differences.

4 Intra-Batch Scheduling

We present Intra-Batch Scheduling (IBS), a scheduling algorithm that dynamically allocates capacity proportional to the workload. We first define three requirements for efficient scheduling (§4.1), then detail the IBS design (§4.2).

4.1 Requirements

We define three requirements: (1) *Proportional allocation*—capacity should match the instantaneous embedding-to-generation demand ratio; (2) *Per-type FCFS*—requests of the same type are served in arrival order, though types may interleave; (3) *No head-of-line blocking*—neither request type should starve the other.

4.2 Scheduler Design

Algorithm 2 presents our intra-batch scheduling algorithm, which satisfies all three requirements. The scheduler first reserves capacity for in-flight generation requests that are in the decode phase. These requests have allocated KV cache entries and must continue execution to avoid KV cache movement. This phase mirrors vLLM scheduling and uses the same swap-in and swap-out mechanisms. The scheduler then fills the remaining budget by alternating between the embedding and generation queues.

Why chunking improves packing. Let R be the residual token budget after reserving capacity for in-flight decode requests. Without chunking, an embedding request of length e is admitted only when $e \leq R$; otherwise, the remaining R tokens go unused. SCHEDULEONE instead admits embedding chunks of at most C tokens. When token capacity is the limiting constraint, a backlogged embedding queue therefore leaves fewer than C

Algorithm 2: Intra-Batch Scheduling

Input: Embedding queue Q_e , generation queue Q_g , token budget B , persistent credits a_e, a_g
Output: Scheduled batch $\mathcal{B}$
 $(\mathcal{B}, u) \leftarrow \text{SCHEDULERUNNING}(B);$
 $B \leftarrow B - u;$
 // Proportionally select from both queues
 $w_e \leftarrow \sum_{r \in Q_e} \text{tokens}(r);$
 $w_g \leftarrow \sum_{r \in Q_g} \text{tokens}(r);$
if $Q_e = \emptyset$ **then**
 | $a_e \leftarrow 0$
if $Q_g = \emptyset$ **then**
 | $a_g \leftarrow 0$
while $B > 0$ **and** $(Q_e \neq \emptyset \text{ or } Q_g \neq \emptyset)$ **do**
 | $a_e \leftarrow a_e + w_e; \quad a_g \leftarrow a_g + w_g;$
 | **if** $Q_g = \emptyset$ **or** $(Q_e \neq \emptyset \text{ and } a_e \geq a_g)$ **then**
 | | $(\mathcal{I}, u) \leftarrow \text{SCHEDULEONE}(Q_e, B);$
 | | $\mathcal{B} \leftarrow \mathcal{B} \cup \mathcal{I}; B \leftarrow B - u;$
 | | $a_e \leftarrow a_e - (w_e + w_g);$
 | **else**
 | | $(\mathcal{I}, u) \leftarrow \text{SCHEDULEONE}(Q_g, B);$
 | | $\mathcal{B} \leftarrow \mathcal{B} \cup \mathcal{I}; B \leftarrow B - u;$
 | | $a_g \leftarrow a_g - (w_e + w_g);$
 // a_e, a_g persist across iterations;
 // reset above when a queue drains
return $\mathcal{B}$

tokens unused once no additional chunk fits. Let w_e and w_g be the sums of the remaining token counts in the embedding and generation queues, respectively. The algorithm maintains accumulators a_e and a_g that track the scheduling credit of each queue. The credits persist across scheduling iterations while their queues remain nonempty. At each selection step, the scheduler increments both accumulators by their corresponding weights and selects a request from the queue with the larger accumulator, decrementing the selected accumulator by $w_e + w_g$. If one queue becomes empty, the scheduler continues selecting from the remaining queue until the token budget is exhausted.

The accumulator mechanism ensures that the ratio of scheduled embeddings to generations converges to $w_e : w_g$, achieving proportional allocation that matches the instantaneous workload mix. When only embeddings are queued ($w_g = 0$), all capacity goes to embeddings, and vice versa. Finally, by maintaining separate queues and alternating between them proportionally, the scheduler avoids head-of-line blocking. Short embedding requests are never stuck behind long generation requests, and each type makes progress according to its share of the workload. Assume each nonempty queue has positive weight, its head request or next chunk can eventually be admitted, and each iteration completes in bounded time. Persistent cred-

its then prevent scheduler starvation: a backlogged queue that is not selected continues accumulating credit until it is chosen. For fixed weights, its allocation differs from its ideal proportional share by at most one scheduling opportunity. This guarantee addresses scheduler-induced blocking. It does not bound queuing delay when offered load exceeds capacity.

5 Evaluation

We evaluate ORTHRUS, an embedding-generation serving system that combines heterogeneous batching with Intra-Batch Scheduling (IBS). The setup is described in §5.1. Our results show that:

- **High Throughput:** $1.28\times$ – $4.52\times$ higher throughput than GPU-level splits on synthetic workloads; $2.4\times$ on a RAG benchmark with 4 GPUs (§5.2).
- **Low Latency:** 9% lower p99 generation and 16% lower p99 embedding latency; up to 55.8% lower p99 on RAG (§5.3).
- **High GPU Utilization:** 41 pp higher average utilization, completing mixed workloads 43% faster (§5.4).
- **Negligible Overhead:** At saturation, matches dedicated-engine throughput for both workload types (§5.5).
- **Generalizability:** Validated on 3 models with LoRA ranks up to 64 (§5.6).

5.1 Experimental Setup

Methods We consider four baselines (**Bs**), varying model placement, batching, and scheduling (Tab. 2). **B1-Dedicated** runs complete embedding and generation requests on disjoint, task-specific GPU pools (e.g., `Dedicated1:3` assigns 1 GPU to embedding and 3 to generation); **B2-Disagg-PD** disaggregates generation prefill and decode across phase-specific GPU pools (Zhong et al., 2024); embedding requests complete after pooling on the prefill GPU; **B3-Unified-Homo** co-locates both types on the same GPU with homogeneous batching; **B4-Unified-Hetero** adds heterogeneous batching but uses FCFS scheduling. Finally, **ORTHRUS (ours)** combines same-GPU placement, heterogeneous batching, and IBS. Unless otherwise specified, we report the median value across three independent runs.

Workloads, Models, and Hardware We used Mistral-7B (Jiang et al., 2023a) as the base

Table 2: System configurations evaluated. ORTHRUS is the only configuration that combines same-GPU placement, heterogeneous batching, and proportional scheduling (IBS).

Config.	Model Placement	Batching	Scheduling
Dedicated	Task-specific GPUs	Homo	FCFS
Disagg-PD	Phase-specific GPUs	Homo	FCFS
Unified-Homo	Same GPU	Homo	FCFS
Unified-Hetero	Same GPU	Hetero	FCFS
ORTHRUS	Same GPU	Hetero	IBS

model with e5-mistral-7b-instruct LoRA (Wang et al., 2024) for embeddings (more models in §5.6). Synthetic workloads use 128-token embeddings and 128/512-token generation (prompt/output). For realistic evaluation, we used Iter-RetGen (Shao et al., 2023) on 2WikiMulti-hopQA (Ho et al., 2020) (avg. 500-token prompts, avg. 3000-token decodes, 1–10 documents per query). Experiments ran on up to four A100 40GB GPUs with round-robin request distribution.

For ORTHRUS, Unified-Homo, and Unified-Hetero, embedding and generation requests are co-located on a single GPU using LoRA adapters. For Dedicated, embeddings are served by e5-mistral-7b-instruct; Mistral-7B serves generation. We also examine sensitivity to scheduling parameters (§A.2) and the effects of chunked embedding and incremental pooling (§A.3).

5.2 High Throughput

We evaluate whether heterogeneous batching improves throughput compared to static GPU partitioning across varying workload ratios.

Controlled Workloads We use fixed embedding-to-generation ratios as controlled stress tests that isolate system behavior under known workload compositions. We evaluated throughput across different embedding-to-generation ratios with 512 clients over 180 seconds. Fig. 3a reports combined throughput; Tab. 5 shows per-type throughput. Dedicated deployments underutilize GPUs when workload ratios diverge from the static split, while Unified-Homo suffers under balanced workloads. Across all evaluated workload ratios, ORTHRUS achieved higher combined throughput than every static GPU partition. Fig. 3b and Fig. 3c illustrate that ORTHRUS fully utilizes all GPUs, while `Dedicated1:3` leaves the embedding GPU underutilized.

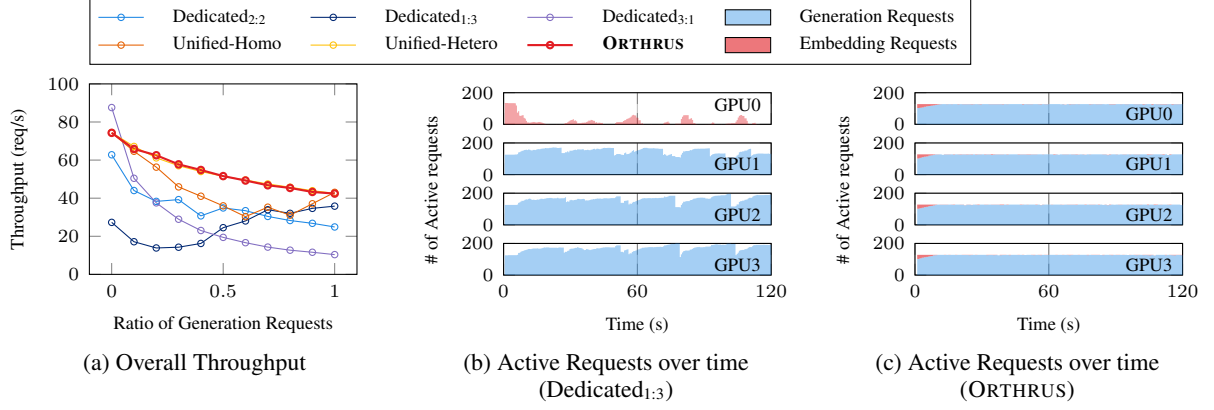


Figure 3: Throughput and GPU utilization comparison. (Fig. 3a) compares throughput of existing serving techniques vs our heterogeneous batching, ORTHRUS (see Tab. 5 for absolute values). (Fig. 3b, Fig. 3c) shows request timelines for Dedicated_{1:3} and ORTHRUS under a 75% generation / 25% embedding workload. Each timeline reports the number of active requests per GPU. ORTHRUS fully utilizes all GPUs, while Dedicated_{1:3} leaves some underutilized.

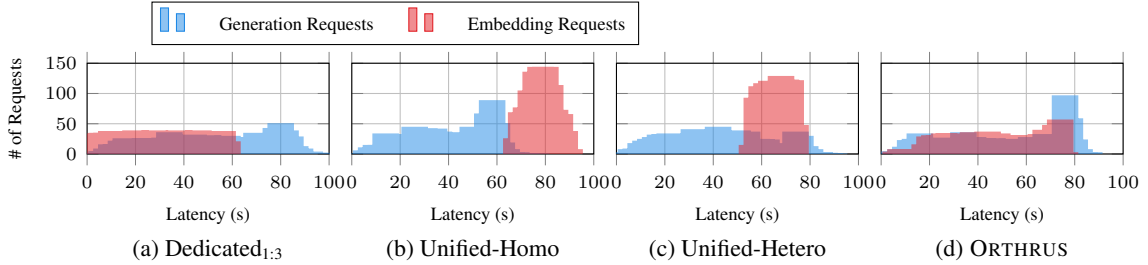


Figure 4: Latency distributions with various serving methods. Curves farther to the left indicate lower latencies. The workload consists of 1,000 generations followed by 1,000 embeddings, all submitted at time 0. ORTHRUS achieves 9% lower p99 generation latency than Dedicated, and up to 16% lower embedding latency than other configurations.

Table 3: E2E request throughput (req/s) on IterRetGen (Shao et al., 2023) with 2WikiMulti-hopQA (Ho et al., 2020). The workload naturally results in variable-length prompts (avg 500, max 2000 tokens), long decode lengths (avg 3000, max 4000 tokens), and multi-document retrieval (1–10 documents per query). ORTHRUS achieves the highest throughput across all configurations.

method	1 GPU		2 GPUs		4 GPUs	
	Embed	Gen	Embed	Gen	Embed	Gen
Dedicated	N/A	N/A	0.08	0.11	0.119	0.141
Unified-Homo	0.067	0.08	0.126	0.148	0.252	0.296
Disagg-PD	N/A	N/A	0.118	0.14	0.276	0.327
ORTHRUS (ours)	0.075	0.088	0.142	0.166	0.288	0.336

RAG Benchmarks To complement the controlled workloads, we evaluate ORTHRUS on a benchmark-derived IterRetGen pipeline. We exclude vector-search and network latency to isolate model-side serving. Prior RAG studies report retrieval on a millisecond scale compared with multi-second generation (Jin et al., 2024; Wang et al., 2023; Trivedi et al., 2023). Each query alternates embedding, retrieval, and genera-

tion across multiple iterations, producing variable-length prompts and long decodes while retrieving 1–10 documents per query. Tab. 3 shows that ORTHRUS achieves the highest throughput at every GPU count, including the single-GPU setting, where disaggregation is infeasible. At four GPUs, its aggregate throughput is 0.624 requests/s, compared with 0.548 for Unified-Homo, 0.603 for Disagg-PD, and 0.260 for Dedicated.

5.3 Low Latency with IBS

We evaluate whether IBS reduces latency by preventing head-of-line blocking, where long-running generation requests stall shorter embedding requests under FCFS scheduling.

Controlled Workloads We submitted 1,000 generation requests followed by 1,000 embedding requests at time 0, simulating worst-case head-of-line blocking. Measured from submission through completion, including queueing, ORTHRUS reduced p99 generation latency by 9% over Dedi-

Table 4: E2E p99 latency (s) on Iter-RetGen (Shao et al., 2023) with 2WikiMultihopQA (Ho et al., 2020). We measure p99 latency per-query end-to-end (including all retrieval iterations). ORTHRUS achieves the lowest latency across all GPU counts.

Method	E2E p99 Latency (s)		
	1 GPU	2 GPUs	4 GPUs
Dedicated	N/A	960.1	1342.0
Disagg-PD	N/A	602.6	617.0
Unified-Homo	673.4	676.0	700.6
ORTHRUS (ours)	573.9	600.5	593.4

cated (89 s $\rightarrow$ 81 s) and p99 embedding latency by 16% over the other schedulers (87 s $\rightarrow$ 73 s). In contrast, Unified-Homo and Unified-Hetero both suffered from embedding requests queuing behind generations. We further evaluate latency robustness across workload ratios and decode lengths in §A.4.

RAG Benchmarks Tab. 4 shows that ORTHRUS achieved the lowest latency across all GPU counts: 14.8% lower than Unified-Homo on 1 GPU (573.9s vs. 673.4s), comparable to Disagg-PD on 2 GPUs (600.5s vs. 602.6s), and 55.8% lower than Dedicated on 4 GPUs (593.4s vs. 1342.0s). ORTHRUS avoids both static partitioning bottlenecks and cross-GPU KV cache transfer.

5.4 High GPU Utilization

We evaluate whether heterogeneous batching improves GPU utilization over static partitioning under shifting workload ratios. Across three phases (10%/50%/90% embedding), ORTHRUS achieved 79% average utilization versus 38% for Dedicated_{1:3}, completing each phase 43% faster by balancing load across GPUs. Static partitioning underperformed: the dedicated embedding GPU was underutilized during generation-heavy phases and overloaded during embedding-heavy phases.

5.5 Minimal Overhead

Fig. 6 compares embedding- and generation-only workloads on a single GPU with 1–256 clients. ORTHRUS reaches dedicated vLLM generation throughput once the workload saturates (≥ 64 clients), yet it trails at lower concurrency. Its embedding throughput is comparable to or higher than the corresponding dedicated configuration; the lower absolute throughput of both LoRA curves reflects adapter overhead. These results show that heterogeneous batching preserves sat-

Table 5: Generalizability analysis comparing absolute throughput (requests/sec) across different models and request mixes. We report embedding and generation throughput separately for three ratios (embedding:generation). Dedicated runs separate models on a 1:3 GPU split. ORTHRUS co-locates both tasks on every GPU.

Model	LoRA	Ratio	Embed	Gen	Embed	Gen
			Dedicated		ORTHRUS	
Mistral 7B	e5-mistral	9:1	15.64	1.49	58.80	7.00
		5:5	12.54	11.97	25.32	26.24
		1:9	5.10	29.56	4.20	39.10
Qwen2 7B	Synthetic Rank 32	9:1	13.23	1.72	40.90	4.40
		5:5	10.72	10.08	26.80	25.93
		1:9	2.81	27.76	6.20	48.23
LLaMA3.1 8B	Synthetic Rank 64	9:1	13.43	1.32	56.46	5.90
		5:5	10.18	9.95	34.60	37.50
		1:9	3.06	25.63	4.23	47.32

urated per-task throughput while enabling GPU sharing. We study the contribution of chunked embedding and incremental pooling in §A.3.

5.6 Generalization

We evaluate whether heterogeneous batching generalizes across model architectures and LoRA adapter ranks. We tested Mistral 7B, Qwen2 7B (Yang et al., 2024), and LLaMA3.1 8B (Grattafiori et al., 2024) with LoRA ranks up to 64 and embedding-to-generation ratios of 9:1, 5:5, and 1:9. As shown in Tab. 5, ORTHRUS achieved higher combined embedding-plus-generation throughput than Dedicated_{1:3} across all configurations. Per-type throughput was also generally higher; the exception was Mistral 7B at 1:9, where embedding throughput decreased from 5.10 to 4.20 requests/s while generation throughput increased from 29.56 to 39.10 requests/s.

Separate Models. We also evaluate ORTHRUS with separate models (OPT-1.3B for generation and GTR-T5-XL for embeddings) on a single A40 GPU. ORTHRUS co-serves both models with a 0.9 GPU-memory cap, while the baseline uses two vLLM instances capped at 0.45 each. We use the Iter-RetGen-style 2WikiMultihopQA workload from §5.2.

ORTHRUS improves embedding and generation throughput by 22.9% and 20.2%, respectively, and reduces both average latencies and embedding p95 latency. The 3.8% increase in generation p95 latency is outweighed by the 17.9% reduction in mean generation latency.

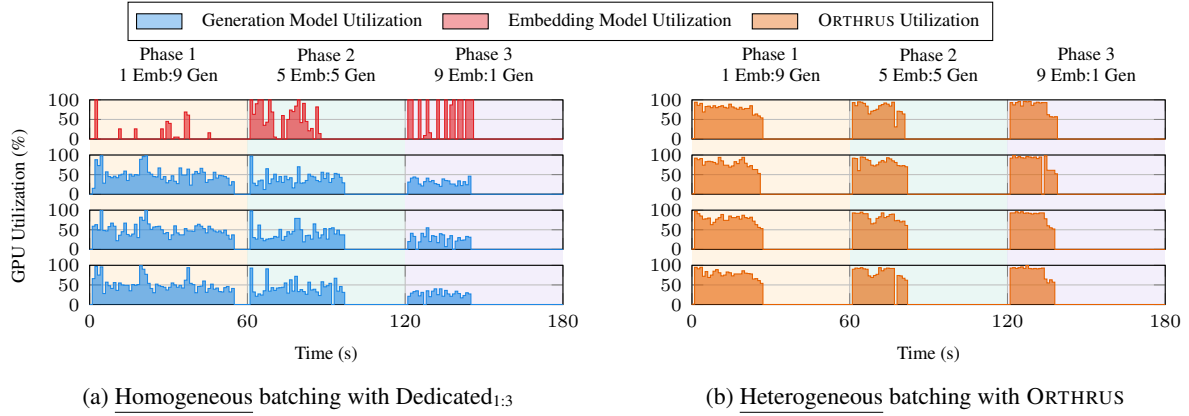


Figure 5: GPU utilization under three workload phases. Dedicated_{1:3} (Fig. 5a) underutilizes GPUs (avg. 38%), while ORTHRUS (Fig. 5b) sustains 79% utilization and completes each phase 43% faster.

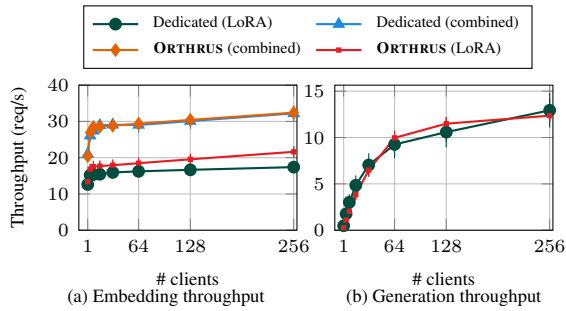


Figure 6: Overhead analysis of ORTHRUS on a single GPU. At ≥ 64 clients, ORTHRUS reaches similar saturated generation throughput to dedicated vLLM. Embedding throughput is comparable to or higher than the corresponding dedicated configuration; the gap between the combined-model and LoRA curves reflects adapter overhead.

6 Conclusion

This work presents ORTHRUS, a serving system that co-serves embedding and generation within a unified model runner through heterogeneous batching. Unlike existing systems, ORTHRUS integrates embedding into the same scheduling and execution path as generation. By combining chunked embedding with incremental pooling and workload-aware scheduling, ORTHRUS achieves higher throughput and lower latency under mixed workloads, enabling efficient serving for knowledge-intensive LLM applications.

Table 6: Separate-model performance on one A40 GPU. Gain denotes a throughput increase or latency reduction relative to $2 \times \text{vLLM}$; positive values favor ORTHRUS.

Metric	Workload	$2 \times \text{vLLM}$	ORTHRUS	Gain
Throughput (req/s)	Emb.	29.7	36.5	+22.9%
	Gen.	28.2	33.9	+20.2%
Avg. latency (ms)	Emb.	187.0	151.8	+18.8%
	Gen.	4337.1	3561.5	+17.9%
p95 latency (ms)	Emb.	421.4	318.6	+24.4%
	Gen.	4875.3	5059.7	-3.8%

Limitations

Our evaluation scales ORTHRUS by running one model replica per GPU, with up to four A100 GPUs. Deployments with more replicas may exhibit different load-balancing behavior and workload skew across GPUs, which can affect utilization and tail latency. Thus, our experiments do not establish how the observed gains scale beyond four GPUs.

ORTHRUS does not currently implement support for a single model sharded across multiple GPUs. The separate-model experiment likewise considers only two models that fit together on one A40. Models that require tensor or pipeline parallelism introduce communication and synchronization within each iteration, which may interact differently with heterogeneous batching and scheduling.

Our experiments use open-weight models with fewer than 10 billion parameters. Larger models have greater memory demands and may exhibit different compute-to-memory ratios and batch-capacity constraints. Although the results are consistent across the tested model families, they do not establish that ORTHRUS provides the same throughput and latency benefits for larger models.

Acknowledgments

This work was supported in part by the IBM-Illinois Discovery Accelerator Institute and by the National Science Foundation under grants #2312561 and #2440498. We used Delta and DeltaAI at the National Center for Supercomputing Applications (NCSA) through allocation CIS240661 from the ACCESS program, which is supported by NSF grants #2138259, #2138286, #2138307, #2137603, and #2138296.

References

- Shubham Agarwal, Sai Sundaresan, Subrata Mitra, Debabrata Mahapatra, Archit Gupta, Rounak Sharma, Nirmal Joshua Kapu, Tong Yu, and Shiv Saini. 2025. *Cache-craft: Managing chunk-caches for efficient retrieval-augmented generation*. *Proc. ACM Manag. Data*, 3(3).
- Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2024. *Taming Throughput-Latency tradeoff in LLM inference with Sarathi-Serve*. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 117–134, Santa Clara, CA. USENIX Association.
- Akari Asai, Zeqiu Wu, Yizhong Wang, Avirup Sil, and Hannaneh Hajishirzi. 2023. *Self-RAG: Learning to retrieve, generate, and critique through self-reflection*. *arXiv preprint arXiv:2310.11511*.
- Parishad BehnamGhader, Vaibhav Adlakha, Marius Mosbach, Dzmitry Bahdanau, Nicolas Chapados, and Siva Reddy. 2024. *LLM2Vec: Large Language Models Are Secretly Powerful Text Encoders*. *arXiv preprint. ArXiv:2404.05961 [cs]*.
- Lequn Chen, Zihao Ye, Yongji Wu, Danyang Zhuo, Luis Ceze, and Arvind Krishnamurthy. 2024. *Punica: Multi-Tenant LoRA Serving*. *Proceedings of Machine Learning and Systems*, 6:1–13.
- Supawit Chockchowwat, Wenjie Liu, and Yongjoo Park. 2023. *AirIndex: Versatile index tuning through data and storage*. *Proceedings of the ACM on Management of Data*, 1(3).
- Luyu Gao, Xueguang Ma, Jimmy Lin, and Jamie Callan. 2023. *Precise Zero-Shot Dense Retrieval without Relevance Labels*. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1762–1777, Toronto, Canada. Association for Computational Linguistics.
- Shiwei Gao, Youmin Chen, and Jiwu Shu. 2025. *Fast state restoration in LLM serving with HCache*. In *Proceedings of the Twentieth European Conference on Computer Systems, EuroSys ’25*, pages 128–143, Rotterdam, Netherlands. ACM.
- Thomas Gauthier. 2024. *Lord: Low-rank decomposition of finetuned large language models*. <https://github.com/thomasgauthier/LoRD>. GitHub repository; commit fe03a8c (2024-05-02). Accessed 2025-09-19.
- Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, and 542 others. 2024. *The Llama 3 Herd of Models*. *arXiv preprint. ArXiv:2407.21783 [cs]*.
- Joseph M Hellerstein, Peter J Haas, and Helen J Wang. 1997. *Online aggregation*. In *Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, pages 171–182.
- Xanh Ho, Anh-Khoa Duong Nguyen, Saku Sugawara, and Akiko Aizawa. 2020. *Constructing a multi-hop QA dataset for comprehensive evaluation of reasoning steps*. In *Proceedings of the 28th International Conference on Computational Linguistics*, pages 6609–6625, Barcelona, Spain (Online). International Committee on Computational Linguistics.

- Cunchen Hu, Heyang Huang, Liangliang Xu, Xusheng Chen, Jiang Xu, Shuang Chen, Hao Feng, Chenxi Wang, Sa Wang, Yungang Bao, Ninghui Sun, and Yizhou Shan. 2024. [Inference without Interference: Disaggregate LLM Inference for Mixed Downstream Workloads](#). *arXiv preprint*. ArXiv:2401.11181 [cs].
- Andrei Ivanov, Nikoli Dryden, Tal Ben-Nun, Shigang Li, and Torsten Hoeffler. 2021. [Data movement is all you need: A case study on optimizing transformers](#). In *Proceedings of Machine Learning and Systems*, volume 3, pages 711–732.
- Gautier Izacard, Mathilde Caron, Lucas Hosseini, Sebastian Riedel, Piotr Bojanowski, Armand Joulin, and Edouard Grave. 2021. [Unsupervised dense information retrieval with contrastive learning](#). *Preprint*, arXiv:2112.09118.
- Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, L  lio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timoth  e Lacroix, and William El Sayed. 2023a. [Mistral 7B](#). *arXiv preprint*. ArXiv:2310.06825 [cs].
- Zhengbao Jiang, Frank Xu, Luyu Gao, Zhiqing Sun, Qian Liu, Jane Dwivedi-Yu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023b. [Active Retrieval Augmented Generation](#). In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 7969–7992, Singapore. Association for Computational Linguistics.
- Chao Jin, Zili Zhang, Xuanlin Jiang, Fangyue Liu, Xin Liu, Xuanzhe Liu, and Xin Jin. 2024. [RAG-Cache: Efficient knowledge caching for retrieval-augmented generation](#). *Preprint*, arXiv:2404.12457.
- Jeff Johnson, Matthijs Douze, and Herv   J  gou. 2021. [Billion-scale similarity search with gpus](#). *IEEE Transactions on Big Data*, 7(3):535–547.
- Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. [Efficient Memory Management for Large Language Model Serving with PagedAttention](#). In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 611–626, Koblenz Germany. ACM.
- Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich K  ttler, Mike Lewis, Wen-tau Yih, Tim Rockt  schel, Sebastian Riedel, and Douwe Kiela. 2020. [Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks](#). In *Advances in Neural Information Processing Systems*, volume 33, pages 9459–9474. Curran Associates, Inc.
- Xiaoxi Li, Yujia Zhou, and Zhicheng Dou. 2024. [UniGen: A Unified Generative Framework for Retrieval and Question Answering with Large Language Models](#). *Proceedings of the AAAI Conference on Artificial Intelligence*, 38(8):8688–8696.
- Zehan Li, Xin Zhang, Yanzhao Zhang, Dingkun Long, Pengjun Xie, and Meishan Zhang. 2023. [Towards General Text Embeddings with Multi-stage Contrastive Learning](#). *arXiv preprint*. ArXiv:2308.03281 [cs].
- Zhaoheng Li, Wei Ding, Silu Huang, Zikang Wang, Yuanjin Lin, Ke Wu, Yongjoo Park, and Jianjun Chen. 2025a. [Cloud-native vector search: A comprehensive performance analysis](#). *Preprint*, arXiv:2511.14748.
- Zhaoheng Li, Silu Huang, Wei Ding, Yongjoo Park, and Jianjun Chen. 2025b. [Sieve: Effective filtered vector search with collection of indexes](#). *Proc. VLDB Endow.*, 18(11):4723–4736.
- Xueguang Ma, Liang Wang, Nan Yang, Furu Wei, and Jimmy Lin. 2024. [Fine-Tuning LLaMA for Multi-Stage Text Retrieval](#). In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR ’24, pages 2421–2425, New York, NY, USA. Association for Computing Machinery.
- Yu A. Malkov and D. A. Yashunin. 2020. [Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs](#). *IEEE Trans. Pattern Anal. Mach. Intell.*, 42(4):824–836.
- Rui Meng, Ye Liu, Shafiq Rayhan Joty, Caiming Xiong, Yingbo Zhou, and Semih Yavuz. 2024. [SFR-Embedding-Mistral: Enhance text retrieval with transfer learning](#). Salesforce AI Research Blog.
- Niklas Muennighoff, Hongjin Su, Liang Wang, Nan Yang, Furu Wei, Tao Yu, Amanpreet Singh, and Douwe Kiela. 2025. [Generative representational instruction tuning](#). In *The Thirteenth International Conference on Learning Representations*.
- Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. 2023. [Efficiently scaling transformer inference](#). In *Proceedings of Machine Learning and Systems*, volume 5, pages 606–624.
- Ruoyu Qin, Zheming Li, Weiran He, Jiale Cui, Heyi Tang, Feng Ren, Teng Ma, Shangming Cai, Yineng Zhang, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. 2025. [Mooncake: A kvcache-centric disaggregated architecture for llm serving](#). *ACM Transactions on Storage*.
- Raunak Shah, Zhaoheng Li, and Yongjoo Park. 2026. [QStore: Quantization-aware compressed model storage](#). *Proceedings of the VLDB Endowment*, 19(3):388–398.
- Zhihong Shao, Yeyun Gong, Yelong Shen, Minlie Huang, Nan Duan, and Weizhu Chen. 2023. [Enhancing Retrieval-Augmented Large Language](#)

- Models with Iterative Retrieval-Generation Synergy.** In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 9248–9274, Singapore. Association for Computational Linguistics.
- Tao Shen, Guodong Long, Xiubo Geng, Chongyang Tao, Yibin Lei, Tianyi Zhou, Michael Blumenstein, and Daxin Jiang. 2024. **Retrieval-Augmented Retrieval: Large Language Models are Strong Zero-Shot Retriever.** In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 15933–15946, Bangkok, Thailand. Association for Computational Linguistics.
- Ying Sheng, Shiyi Cao, Dacheng Li, Coleman Hooper, Nicholas Lee, Shuo Yang, Christopher Chou, Banghua Zhu, Lianmin Zheng, Kurt Keutzer, Joseph E. Gonzalez, and Ion Stoica. 2024a. **SLoRA: Scalable Serving of Thousands of LoRA Adapters.** *Proceedings of Machine Learning and Systems*, 6:296–311.
- Ying Sheng, Shiyi Cao, Dacheng Li, Banghua Zhu, Zhuohan Li, Danyang Zhuo, Joseph E. Gonzalez, and Ion Stoica. 2024b. **Fairness in serving large language models.** In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 965–988, Santa Clara, CA. USENIX Association.
- Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Re, Ion Stoica, and Ce Zhang. 2023. **FlexGen: High-Throughput Generative Inference of Large Language Models with a Single GPU.** In *Proceedings of the 40th International Conference on Machine Learning*, pages 31094–31116. PMLR. ISSN: 2640-3498.
- Nikhil Sheoran, Supawit Chockchawat, Arav Chheda, Suwen Wang, Riya Verma, and Yongjoo Park. 2023. A step toward deep online aggregation. *Proceedings of the ACM on Management of Data*, 1(2):1–28.
- Foteini Strati, Xianzhe Ma, and Ana Klimovic. 2024. **Orion: Interference-aware, fine-grained gpu sharing for ml applications.** In *Proceedings of the Nineteenth European Conference on Computer Systems, EuroSys ’24*, page 1075–1092, New York, NY, USA. Association for Computing Machinery.
- Yubao Tang, Ruqing Zhang, Jiafeng Guo, Maarten de Rijke, Yixing Fan, and Xueqi Cheng. 2025. **Boosting Retrieval-Augmented Generation with Generation-Augmented Retrieval: A Co-Training Approach.** In *Proceedings of the 48th International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR ’25*, pages 2441–2451, New York, NY, USA. Association for Computing Machinery.
- Harsh Trivedi, Niranjan Balasubramanian, Tushar Khot, and Ashish Sabharwal. 2023. **Interleaving Retrieval with Chain-of-Thought Reasoning for Knowledge-Intensive Multi-Step Questions.** In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 10014–10037, Toronto, Canada. Association for Computational Linguistics.
- Bhoomit Vasani, Jack FitzGerald, Anjie Fang, and Sushmit Vaish. 2025. Phlora: data-free post-hoc low-rank adapter extraction from full-rank checkpoint. *arXiv preprint arXiv:2509.10971*.
- 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 *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc.
- Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, Kun Yu, Yuxing Yuan, Yinghao Zou, Jiquan Long, Yudong Cai, Zhenxiang Li, Zhifeng Zhang, Yihua Mo, Jun Gu, and 3 others. 2021a. **Milvus: A purpose-built vector data management system.** In *Proceedings of the 2021 International Conference on Management of Data, SIGMOD ’21*, page 2614–2627, New York, NY, USA. Association for Computing Machinery.
- Liang Wang, Nan Yang, Xiaolong Huang, Linjun Yang, Rangan Majumder, and Furu Wei. 2024. **Improving Text Embeddings with Large Language Models.** In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 11897–11916, Bangkok, Thailand. Association for Computational Linguistics.
- Liang Wang, Nan Yang, and Furu Wei. 2023. **Query2doc: Query Expansion with Large Language Models.** In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 9414–9423, Singapore. Association for Computational Linguistics.
- Xiaohui Wang, Ying Xiong, Yang Wei, Mingxuan Wang, and Lei Li. 2021b. **LightSeq: A High Performance Inference Library for Transformers.** In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies: Industry Papers*, pages 113–120, Online. Association for Computational Linguistics.
- Bingyang Wu, Yinmin Zhong, Zili Zhang, Shengyu Liu, Fangyue Liu, Yuanhang Sun, Gang Huang, Xuanzhe Liu, and Xin Jin. 2024. **Fast Distributed Inference Serving for Large Language Models.** *arXiv preprint. ArXiv:2305.05920 [cs]*.
- Haocheng Xia, Mihir Pamnani, Hanxi Fang, Supawit Chockchawat, and Yongjoo Park. 2026. **Lazy-Attention: Efficient retrieval-augmented generation with deferred positional encoding.** In *Proceedings of the 43rd International Conference on Machine Learning*, Proceedings of Machine Learning Research. PMLR.

- Haojun Xia, Zhen Zheng, Yuchao Li, Donglin Zhuang, Zhongzhu Zhou, Xiafei Qiu, Yong Li, Wei Lin, and Shuaiwen Leon Song. 2023. [Flash-llm: Enabling cost-effective and highly-efficient large generative model inference with unstructured sparsity](#). *Preprint*, arXiv:2309.10285.
- An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, Guanting Dong, Haoran Wei, Huan Lin, Jialong Tang, Jialin Wang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Ma, and 43 others. 2024. [Qwen2 Technical Report](#). *arXiv preprint*. ArXiv:2407.10671 [cs].
- Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A distributed serving system for {Transformer-Based} generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538.
- Lingfan Yu, Jinkun Lin, and Jinyang Li. 2025. [Stateful large language model serving with pensieve](#). In *Proceedings of the Twentieth European Conference on Computer Systems*, EuroSys '25, page 144–158, New York, NY, USA. Association for Computing Machinery.
- Caojin Zhang, Qiang Zhang, Ke Li, Sai Vidyaranya Nuthalapati, Benyu Zhang, Jason Liu, Serena Li, Lizhu Zhang, and Xiangjun Fan. 2025. [GEM: Empowering LLM for both Embedding Generation and Language Understanding](#). *arXiv preprint*. ArXiv:2506.04344 [cs].
- Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. 2024. [DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving](#). In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 193–210, Santa Clara, CA. USENIX Association.

A Additional Experiments

A.1 Pooling Equivalence

We test whether incremental pooling reproduces single-pass embeddings across pooling heads and chunk sizes. On 250 samples spanning a range of input lengths, we compare paired outputs for mean, CLS, and weighted-mean pooling at chunk sizes 256, 512, and 1024 using cosine similarity and relative L_2 error. In Tab. 7, minimum cosine similarity exceeds 0.9999 in all nine configurations, while both mean and p99 relative L_2 errors remain below 1%. These small discrepancies reflect floating-point reduction-order effects rather than an algorithmic approximation.

Table 7: Equivalence between chunked and single-pass embeddings. C is the chunk size; L_2 errors are relative.

Pooling	C	Cosine similarity			Relative L_2	
		Min	Mean	p50	Mean	p99
Mean	256	.9999512	.9999844	.9999864	.005527	.008995
	512	.9999555	.9999861	.9999885	.005176	.009478
	1024	.9999484	.9999877	.9999881	.004820	.008156
CLS	256	.9999894	.9999912	.9999912	.005072	.005113
	512	.9999837	.9999887	.9999892	.004908	.005479
	1024	.9999920	.9999930	.9999920	.003692	.004236
Weighted mean	256	.9999456	.9999882	.9999907	.004650	.009628
	512	.9999339	.9999887	.9999918	.004493	.009927
	1024	.9999452	.9999905	.9999915	.004215	.007636

A.2 Scheduling-Parameter Sensitivity

Chunk-Size Sweep We test whether performance depends on the granularity of chunked embedding and select a chunk size for the subsequent budget sweep. Using the 5:5 workload from §5.2, we vary the embedding chunk size C across 128, 256, 512, and 1024 tokens. As shown in Fig. 7(a), combined throughput ranges from 2.6167 to 2.6500 requests/s, generation p99 from 4.08 to 4.16 s, and embedding p99 from 83.58 to 98.77 ms. This shows that performance is insensitive to chunk size over the tested range.

Token-Budget Sweep We next test whether performance at the selected chunk size depends on the per-iteration token budget. The 256- and 512-token settings tie for the highest combined throughput, so we select $C = 512$ because it has the lower embedding p99. Holding this chunk size and the workload fixed, we sweep the token budget B across 2048, 4096, and 8192 tokens. As shown in Fig. 7(b), combined throughput remains between 2.6125 and 2.6375 requests/s, generation p99 between 4.06 and 4.12 s, and embed-

ding p99 between 82.95 and 109.35 ms. Combined throughput varies by less than 1%, and latency shows no consistent degradation as the budget changes. These results show that ORTHRUS does not require a narrowly tuned token budget.

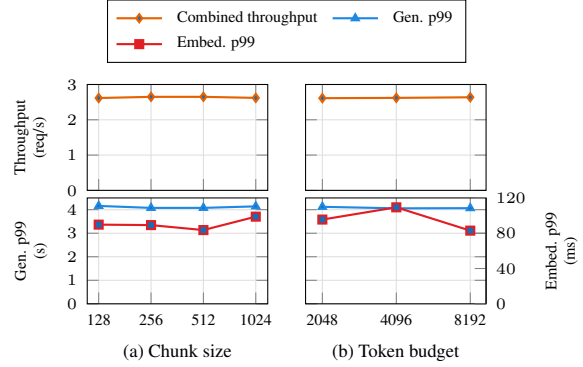


Figure 7: Scheduling-parameter sensitivity for (a) chunk size and (b) token budget. Throughput and latency for embedding and generation remain stable across different chunk sizes and token budgets.

A.3 Standalone Chunked Embedding Ablation

Because chunked embedding and incremental pooling are enabled together in our implementation, we evaluate their combined effect in two complementary settings.

Scheduler-Packing Simulation We first test whether chunking improves token-budget packing without reducing generation service. We compare two otherwise identical FCFS iteration schedulers. In the atomic condition, an embedding request must be processed in one iteration. In the chunked condition, it can be divided into chunks of at most $C = 64$ tokens. Each iteration has a token budget of $B = 256$, and the request mix is 80% generation and 20% embedding. Each generation request has a 128-token prefill followed by a 256-token decode, which the simulator advances in eight-token steps. Embedding requests contain either 96 or 224 tokens, with 224-token inputs comprising 45% of the embedding workload. As shown in Tab. 8, chunking raises average token-budget fill from 8.8% to 85.7%. Generation throughput remains unchanged at 93.3 requests/s, while embedding throughput increases from 25.5 to 26.9 requests/s. Thus, chunking recovers unused token capacity without sacrificing generation throughput.

Table 8: Atomic-versus-chunked embedding simulation. Budget fill is the average fraction of the per-iteration token budget occupied by scheduled tokens; throughput is in requests/s.

Embedding	Budget fill (%)	Total	Gen	Embed
Atomic	8.8	118.8	93.3	25.5
Chunked	85.7	120.2	93.3	26.9

Embedding-Only Execution We next test whether the implemented chunked path introduces execution overhead when embeddings run without generation contention. We compare chunking disabled against chunking with incremental pooling enabled on 250 inputs using mean-pooled OPT-1.3B embeddings. As shown in Tab. 9, chunking reduces mean latency by 9.8% (20.4 to 18.4 ms) and increases throughput by 10.9% (49.0 to 54.4 embeddings/s). Thus, the scheduling flexibility provided by chunking does not come at the cost of standalone embedding efficiency on this workload.

Table 9: Embedding-only comparison using mean-pooled OPT-1.3B embeddings on 250 inputs.

Setting	Avg. latency (ms)	Embed./s	Relative
No chunk	20.4	49.044	1.000×
Chunked	18.4	54.384	1.109×

A.4 Latency Robustness

Workload-Ratio Sensitivity We evaluate whether IBS maintains low generation startup and embedding latency as the workload composition shifts. We sweep the generation request share across 10%, 50%, and 90%, using three repeats per setting, 512 clients, and 180-second submission windows. Generation prompts and embedding inputs contain 128 tokens, and generation is capped at 1024 output tokens. As shown in Fig. 8, generation TTFT p99 remains below 200 ms across the sweep, ranging from 173.57 to 181.96 ms. Embedding p50 also remains stable at 278.70–301.87 ms. Embedding p95 is 322.59 and 327.35 ms at 10% and 50% generation, respectively, but rises to 1.71 s at 90% generation. Thus, generation startup and typical embedding latency are robust to the request mix, while an extremely generation-heavy workload primarily affects the embedding tail.

Long-Decode Sensitivity We evaluate whether

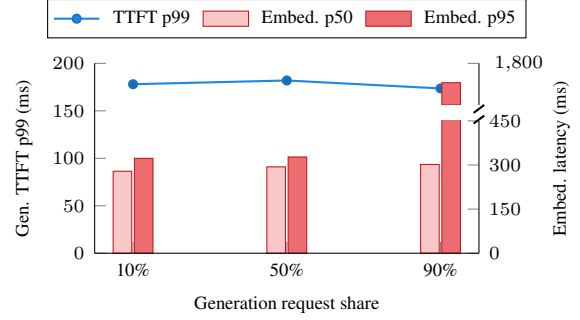


Figure 8: Generation and embedding latency across workload ratios. Generation TTFT p99 and embedding p50 remain stable, while embedding p95 increases under generation-heavy workloads.

IBS preserves prompt embedding service as longer generation decodes occupy the system. Using the 5:5 workload, we vary only the generation decode length, from 256 to 4096 tokens. As shown in Tab. 10, generation p99 latency increases from 7.06 to 79.68 s. In contrast, embedding p50 remains between 52.58 and 54.62 ms, and embedding p99 remains between 53.09 and 59.92 ms. Throughput is not monotonic. For example, it increases from 0.1167 requests/s at 2048 tokens to 0.1333 requests/s at 4096 tokens. Thus, we do not interpret differences between adjacent settings as a scaling trend. The central result is latency isolation. Despite an over 11× increase in generation p99, embedding p99 varies by less than 7 ms. IBS continues to schedule embedding requests promptly even when generation becomes decode-dominated.

Table 10: Decode-length sensitivity on the 5:5 workload. Throughput is reported in requests/s and latency in milliseconds.

Decode (tokens)	Throughput (rps)		Latency (ms)		
	Gen	Embed	Gen p99	Embed p50	Embed p99
256	0.2333	0.2333	7057.75	53.32	59.02
512	0.1611	0.1611	13959.11	52.58	53.09
1024	0.1528	0.1528	27806.13	53.37	59.92
2048	0.1167	0.1167	56780.18	54.62	54.97
4096	0.1333	0.1333	79675.66	54.42	55.42