BROS: Efficient LLM Serving on Hybrid Real-time and Best-effort Requests

Borui Wan*‡, Juntao Zhao*‡, Chenyu Jiang*, Chuanxiong Guo†, and Chuan Wu*
*Department of Computer Science, The University of Hong Kong †BitIntelligence

***Abstract*—Recent breakthroughs in large language models have enabled various generative tasks on a single model. Real-world services powered by the LLM often concurrently support latency-critical requests for interactive applications (e.g., question-answering systems, referred to as *real-time* or *RT* requests) and throughput-oriented requests for back-of-house processing (e.g., documents batch processing, referred to *best-effort* or *BE* requests), with complex hybrid inference workloads to the underlying model. State-of-the-art (SOTA) LLM serving systems dedicate machines to each type of requests, towards either low inference latency or high serving throughput, respectively. This simplifies request scheduling and management but suffers from poor resource utilization. We propose BROS, a hybrid LLM serving system that aims to collocate RT/BE requests, meeting RT requests' latency requirements while maintaining BE requests' throughput. BROS formulates the problem of hybrid RT/BE request scheduling and solves it with a dynamic priority-based algorithm. BROS designs a bidirectional KV cache management mechanism, allowing RT requests to share KV memory with BE requests to remove the scheduling restrictions caused by memory insufficiency and improve utilization. Extensive experiments validate that BROS achieves a good trade-off when serving hybrid RT and BE requests. It significantly reduces the latency of RT requests (up to 74.20%), improving their fine-grained service level objectives (SLOs) attainments (up to 36.38×), with negligible throughput reduction for BE requests, showing significant advantages over SOTA systems.**

I. Introduction

Large Language Models (LLMs) [1]–[5] have demonstrated remarkable performance on language-related generative tasks, enabling various real-world online applications such as role-playing [6], programming assistants [7], and offline applications including document batch processing [8], information extraction [9], and data wrangling [10]. A single LLM can handle various tasks with wise post-training [11]–[14]

Online applications such as ChatGPT [15] and AI anime chatbots [6] return the generated context in a streaming manner [16], demanding both quick response after queueing and high context rate (e.g., 250 words/min) [17] afterward. Apart from normalized latency[1] [18], [19], this serving goal is also measured by two metrics: time to first token (TTFT) and time per output token (TPOT) [17]. We refer to such LLM workload as *real-time* or RT requests. Offline applications, such as the asynchronous document processing service provided by OpenAI's Batch API [8], usually involve inference workloads processed in a batch manner within a specified time window (e.g., 24 hours in OpenAI's Batch API). Therefore, they are not sensitive to latency but favor high throughput (requests per second). We refer to such requests as *best-effort* or BE requests. This discrepancy in serving goals for RT/BE requests leads to the disaggregated deployments of online and offline LLM services on separated GPU clusters, which is the current common practice for production [17], [20]–[24].

However, the volume of real-world RT requests changes in a diurnal pattern [25]. To fulfill the serving goals of RT requests, service providers should provide enough GPU resources to meet the peak demands [26]. Over-provisioning separate resources results in GPU cycles and energy wastes, both of which finally translate into operational costs. Co-serving RT and BE requests on shared resources offers a compelling alternative, especially when GPU resources are scarce. By strategically batching latency-insensitive BE requests with RT ones during opportune intervals, systems can enhance resource utilization while maintaining strict RT latency guarantees. Nevertheless, LLM's unique autoregressive inference pattern presents unprecedented challenges for hybrid LLM serving.

First, the non-predetermined number of LLM inference passes makes it difficult to schedule RT/BE requests to meet their distinct serving goals. Given the quick response and high context rate requirements of RT requests, the desirable property of the LLM serving system is to balance TTFT and TPOT latency Service-Level Objectives (SLOs) of RT requests while maintaining long-term performance for BE requests. However, existing request scheduling mechanisms fall short of achieving a good trade-off between RT and BE requests. An LLM responds to user requests with one *prefill* pass and several *decode* passes, where the number of the latter cannot be known at the queuing stage. Multiple model querying iterations prevent conventional priority fair scheduling from directly assigning priority values (e.g., total inference time or remaining time) for each LLM request. Other heuristics like Multi-level Feedback Queue (MLFQ), when applied to LLM request scheduling [27], can reduce the average end-to-end inference latency. Nevertheless, this approach does not align well with the quick response demand of RT requests and is not suitable for hybrid requests where BE requests are insensitive to latency. Moreover, BE and RT requests exhibit conflicting preferences for system configurations, particularly in batch sizing. RT requests favor smaller batches to minimize latency [28]–[30], while BE requests benefit from larger batches to maximize GPU utilization and throughput [18], [19]. These opposing requirements create significant challenges in config-

‡The first two authors contributed equally to this work.

[1]Normalized latency is the mean of every request's end-to-end latency divided by its output token number.

uring dynamic RT/BE co-serving systems.

Further, co-scheduling RT/BE requests brings GPU memory contention associated with caching intermediate states. As the core of LLMs, decoder-based Transformers [31] generate one token in each pass, based on the prompt and all the previously generated tokens. The key and value tensors of all previously processed tokens need to be cached in GPU memory for the generation of the next token (aka KV cache). Concurrent RT and BE requests compete for memory resources to store their KV cache, which imposes limitations on scheduling decisions. As the *first-class citizen*, RT requests may fail to be scheduled when available GPU memory is insufficient to accommodate their increasing KV cache. Simply dropping or swapping the KV cache [19], [27] of low-priority BE requests leads to recomputation or Device-to-Host (D2H) copy overheads, delaying their completion of inference.

None of the existing LLM serving systems [17]–[20], [23], [27] designs dedicated hybrid RT/BE requests scheduling mechanisms. Most of them [17], [18], [23], [32] still rely on First-Come-First-Serve (FCFS) scheduling, which can easily incur head-of-line blocking due to running early-arrived BE requests with long prompts and output sequences that block late-joining RT requests. Besides, the KV cache management strategies of existing systems do not handle the storage competition between RT and BE requests well.

We propose BROS, an LLM serving system to handle hybrid RT/BE requests. Our contributions are summarized as follows:

▷ We formulate the hybrid scheduling problem and propose a dynamic packing algorithm for hybrid request scheduling. To fulfill quick response and high context rate demands of RT requests, the hybrid scheduling problem views each queueing RT request's TTFT/TPOT metrics as constraints while optimizing the throughput of BE requests to avoid starvation. The proposed heuristic first batches RT requests according to their dynamically assigned priorities and then adaptively replaces low-priority RT requests with BE requests. We further provide an SLO-aware adaptive batch sizing mechanism to control the runtime batch size. As a result, BROS strikes a favorable serving trade-off between RT and BE requests.

▷ We resolve memory competition between RT and BE requests with a bidirectional storage layout design for the KV cache. The available memory is divided into blocks [19]. Each block can be shared by two request types and KV caches of the RT and BE requests expand in opposite directions in each block. Based on this layout, we propose block preemption to guarantee sufficient GPU memory for high-priority RT requests, along with a lazy checkpointing technique to avoid/delay runtime D2H requirements.

▷ We evaluate BROS on different workloads, which shows that BROS significantly reduces the latency of RT requests by up to 74.20% while maintaining BE throughput, outperforming SOTA production-grade LLM serving systems [19], [32].

II. Background and Motivation

In this section, we first briefly introduce the characteristics of LLM inference (Sec. II-A), then analyze critical limitations of existing serving systems. These observations motivate BROS's design choices (Sec. II-B).

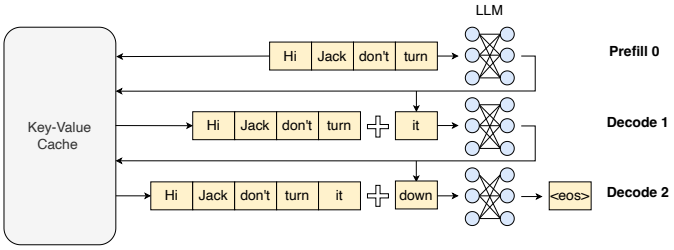
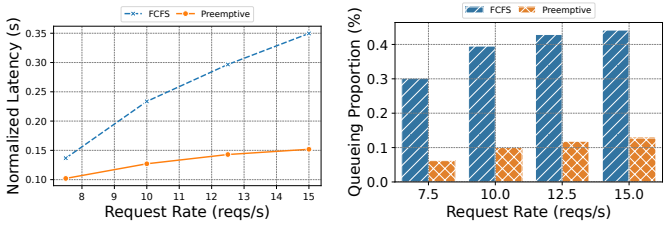


Fig. 1: The two phases of an LLM inference request.



(a) Normalized latency curves. (b) Queueing proportion.

Fig. 2: Head-of-line blocking with FCFS and the benefit of preemptive scheduling. The queueing proportion is the mean of every request's queueing time divided by its end-to-end latency.

A. LLM Inference

Today's LLMs are empowered by decoder-based Transformer structures [1]. Inference over an LLM is conducted in the autoregressive manner, including two phases: prefill and decode. One forward pass over the LLM model is called one *iteration*. As depicted in Fig. 1, the prefill phase involves a single iteration where the prompt tokens (e.g., "Hi Jack, don't turn") are used to compute the first output token ("it"), and all the key-value tensors of the tokens are stored as the initial KV cache. At each iteration of the decode phase, the key-value tensors of the prompt and previously generated tokens are retrieved from the KV cache, and utilized, along with the latest token, to generate the new token ("down"). If there are more tokens to generate, the key-value tensors of the latest token ("it") are also cached. This process continues until the "EOS" token is generated or a maximum output length is reached. Consequently, the total serving time of each request is determined by the prompt length and the number of iterations in the decode phase, which varies from request to request.

B. Challenges and Opportunities

Observation #1: Naive preemption benefits RT requests but fails to balance TTFT and TPOT. Most of the existing LLM serving systems [17]–[19] employ FCFS policy to schedule requests, which results in head-of-line blocking when BE requests containing a significant number of output tokens are scheduled ahead of the RT requests. One possible solution to alleviate this is to allow request preemption at fine

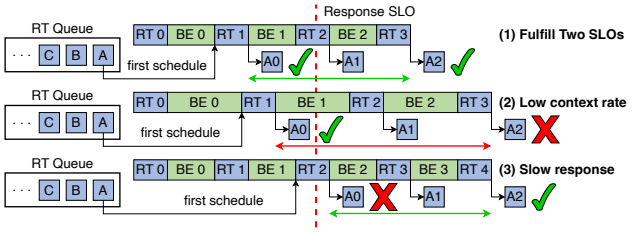


Fig. 3: Round-robin scheduling for hybrid RT/BE requests.

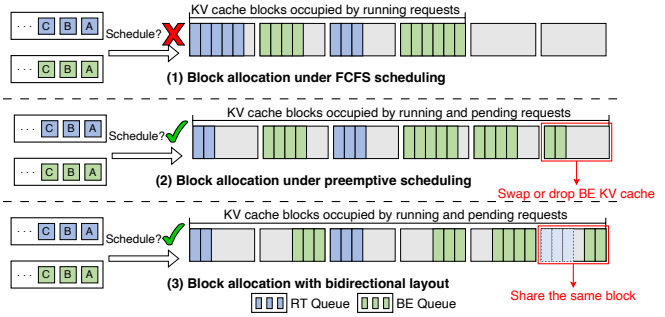


Fig. 4: GPU memory block allocation for storing KV cache under different cases.

granularity, i.e., every LLM forward pass (iteration). Fig. 2 shows the benefits of applying preemptive scheduling with a simple policy, where each request's waiting time between completing one iteration and scheduling the next is used as the priority value. We run vLLM with FCFS policy and the above preemptive scheduling policy to serve a Qwen2.5-14B model on a server with four A100-40GB GPUs, with the tensor-parallel degree of 2. (detailed configurations in Sec. VII). 512 requests are synthesized based on the SharedGPT [33] dataset, then submitted to the system from Poisson distribution with different request rates. Compared to the FCFS policy, preemptive scheduling effectively reduces normalized latency (Fig. 2a) by minimizing the overall queuing time (Fig. 2b) for waiting requests in the system.

Challenges. However, existing iterative preemption approaches face a scheduling dilemma to balance TTFT and TPOT SLOs for RT requests. For example, vLLM preempts decode RT requests for incoming prefill requests, which sacrifices the high context rate (TPOT). Sarathi-Serve adopts *chunked-prefills* to create stall-free schedules for ongoing decode requests but delays the return of the initial context (TTFT).

Observation #2: Rigid hybrid scheduling is inefficient for co-serving. One conventional fair scheduling approach, *Iteration-level round-robin scheduling*, is intuitive for co-serving hybrid RT/BE requests. The system maintains two FCFS queues for RT and BE requests, respectively, and serves them in an interleaving manner at the iteration level. The round-robin policy allows fair resource sharing between RT and BE requests. However, this static scheduling mechanism struggles to adapt to LLM requests with varying execution times due to different prompt and output lengths, making it difficult to achieve a favorable serving trade-off. We show different cases in Fig. 3, where the enqueued RT request A needs three iterations to finish. The latency SLO for its response (in this example, the time to first token [17]) is denoted by the red dashed line. The time to generate the subsequent context is represented by double-headed arrows. In case (1), round-robin scheduling succeeds in providing the quick response and high context rate for request A, while at the same time does not starve BE requests. However, in case (2), request A's context rate is low due to the prolonged execution time of the BE requests, which may result from more BE requests being scheduled or more tokens per BE request. In case (3), request A fails to meet the response SLO target since previously scheduled RT requests need more iterations to exit the current batch [18], delaying its first scheduled time. Static hybrid scheduling mechanism is inefficient in identifying the most urgent RT requests to serve and balancing RT/BE request serving, according to dynamic request arrivals and their output lengths.

Challenges. Achieving dynamic scheduling is non-trivial: First, due to the varying phases and number of tokens to be processed for different requests, the execution time of each iteration can vary significantly. This variability makes it challenging to determine whether the scheduling decisions may violate the SLO targets. Besides, with a large number of RT and BE requests present in the system, it becomes crucial to select appropriate requests from those combinations for prompt execution. Failing to do so can result in unacceptable scheduling overhead. Furthermore, configuring batch sizes to simultaneously meet the divergent requirements of RT and BE workloads is also challenging.

Observation #3: Bidirectional block layout mitigates memory contention. Throughout the lifespan of a request in an LLM serving system, the memory demand of its KV cache increases in proportion to the number of iterations it undergoes. vLLM [19] proposes to dynamically allocate available GPU memory to requests for KV cache storage at the granularity of the block (each contains several slots for tokens' KV tensors), which proves to be an efficient method to improve serving throughput. With the FCFS schedule, new requests are only served once completed requests exit and release their occupied cache blocks (case (1) in Fig. 4), regardless of whether they are latency-critical or not; This results in block wastage when running requests cannot consume all reserved blocks. When advocating iteration-level request preemption and scheduling among RT and BE requests, there are more concurrent requests in the serving system, incurring more KV cache storage overhead and memory competition. To preemptively schedule later-join RT requests, the system needs to yield cache blocks from BE requests to these RT requests, introducing recomputation or swapping overheads (case (2) in Fig. 4).

Opportunities. More fine-grained KV cache management assists in reducing memory overheads. As shown in case (3), a bidirectional block layout, which shares one block between

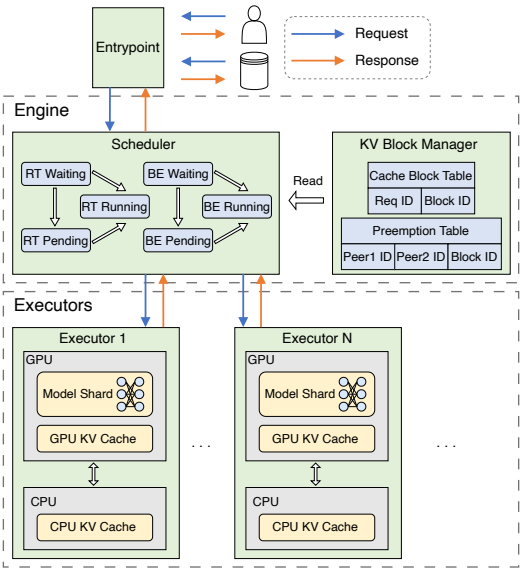


Fig. 5: BROS system overview.

two requests and increases their KV cache in the opposite direction, creates the opportunity for scheduled RT requests to utilize empty slots in the cache block from pending BE requests, without KV cache dropping or swapping.

Our contributions. In BROS, we consider quick response and high context rate requirements for RT requests simultaneously, using them to guide scheduling design for balancing SLO attainments. We build latency cost models and design an effective scheduling mechanism to batch and run RT and BE requests strategically across iterations. We properly manage KV cache contention and strategically handle the potential data overwrite issues.

III. System Overview

Fig. 5 shows BROS's architecture. It exposes an entrypoint that receives RT (e.g., users' questions) and BE requests (e.g., documents from storage) and forwards them to the scheduling *engine*. Each incoming RT or BE request is initially stored in the corresponding waiting queue and will be moved into the pending queue after its prefill stage processing has been scheduled. The scheduler running on the engine schedules waiting and pending RT and BE requests at each iteration to populate the respective running queue. The *KV block manager* maintains a cache block table that records the mapping between each request ID and the allocated cache block ID. A preemption table is used to track the preemption status of cache blocks. It records the IDs of RT and BE requests that share the same blocks and the IDs of the preempted blocks, indicating occurrence of slot preemption in those blocks. The scheduler acquires KV cache memory utilization and selects appropriate RT and BE requests for execution. After scheduling, the engine dispatches the batched requests to the *executors*, where tensor parallelism is supported for distributed inference. Each executor wraps a GPU and launches a process that loads the weights of the LLM model shard for distributed model execution and manages KV caches of all ongoing requests in GPU and CPU memory. Asynchronous memory-swapping operations are triggered in the executors upon receiving corresponding instructions from the scheduler, to partially checkpoint KV caches of certain requests in CPU memory and swap KV caches of other requests into GPU memory.

TABLE I: Notation.

x_i	whether select RT request i
y_j	whether select BE request j
t_i	elapsed time of RT request i since the last token is returned
$N^{rt}(N^{be})$	number of RT (BE) requests in the waiting and pending queues
$L_i^n(L_j^n)$	input length of RT request i (BE request j)
$L_i^a(L_j^a)$	context length of RT request i (BE request j)
$M_i^{new}(M_j^{new})$	number of needed new cache blocks of RT request i (BE request j)
$M_i^{cpu}(M_j^{cpu})$	number of checkpointed cache blocks of RT request i (BE request j)
B_{curr}	batch size of current iteration
$\mathcal{T}_{SLO}$	token latency SLO target, which can be either TTFT or TPOT
T_{avg}	the moving-average iteration latency
M^{ept}	number of empty cache blocks

IV. Request Scheduler

The goal of the scheduler is to define a proper batch size and select suitable RT and BE requests from the waiting and pending queues to execute at each iteration, aiming to meet the latency SLOs of each RT request while maximizing the throughput of BE requests. In this section, we begin by formulate the scheduling problem (Sec. IV-A), then propose our heuristics and introduce the adaptive batch sizing mechanism in Sec. IV-B.

A. Hybrid Scheduling Problem

Formulation. We model an iteration-level request batching and scheduling problem, to decide which requests to be included in each batch. We use decision variables x_i to indicate whether the generation of the next token of an RT request i (in the current pending and waiting queues) is included in the current batch, and y_j to denote whether that of a BE request j is in the batch. The goal is to maximize the inference throughput of BE requests while ensuring the token latency SLOs of each RT request (notation is summarized in Table I):

$$\max_{x_i, y_j \in \{0,1\}} \frac{\sum_{j=1}^{N^{be}} y_j}{T} \tag{1}$$

subject to:

$$T = C(L^n, L^a, M^{cpu}) \tag{2}$$

$$L^n = \sum_{i=1}^{N^{rt}} L_i^n x_i + \sum_{j=1}^{N^{be}} L_j^n y_j \tag{3}$$

$$L^a = \sum_{i=1}^{N^{rt}} L_i^a x_i + \sum_{j=1}^{N^{be}} L_j^a y_j \tag{4}$$

$$M^{cpu} = \sum_{i=1}^{N^{rt}} M_i^{cpu} x_i + \sum_{j=1}^{N^{be}} M_j^{cpu} y_j \tag{5}$$

$$t_i + (1 - x_i) T_{avg} + T \leq \mathcal{T}_{SLO}, \forall i \in [N^{rt}] \tag{6}$$

$$\sum_{i=1}^{N^{rt}} x_i + \sum_{j=1}^{N^{be}} y_j \leq B_{curr} \tag{7}$$

$$\sum_{i=1}^{N^{rt}} M_i^{new} x_i + \sum_{j=1}^{N^{be}} M_j^{new} y_j \leq M^{ept} \tag{8}$$

The BE request serving throughput in (1) is evaluated by the number of generated tokens per unit time. In (2), the cost model C returns the latency of the current iteration, i.e., the time to process the current batch. (3) to (5) give how to derive the inputs L^n, L^a, M^{cpu} for the iteration time lookup. L^n indicates the total input token length of all RT and BE requests in the batch, L^a is the total length of context (including input and cached tokens) of all the scheduled requests, and M^{cpu} is the total number of checkpointed cache blocks of RT/BE requests scheduled in this batch (store in the CPU memory). (6) ensures that the completion time of the current token of each RT request can meet the respective SLO ($\mathcal{T}_{SLO}$ is either the TTFT or TPOT target depending on whether the token is the first or a later one), assuming the remaining $k_i - x_i$ tokens are scheduled in the following iterations consecutively. (7) limits the number of selected requests to include in each batch to B_{curr}, which is determined by the adaptive batching mechanism. (8) guarantees sufficient empty GPU blocks for scheduled RT/BE requests.

Inference cost modeling. We build cost models to estimate the LLM serving latency at an iteration. The input to LLM inference of a batch includes the hidden state sequences $\in R^{L^n \times D}$ (D is the hidden size) and the context of all requests in the batch. The model execution latency depends on two shapes: $R^{L^n \times D}$, which is the input shape to none-attention operators like linear projections, activations, and normalization functions [31], as well as collective communication; $R^{L^n \times L^a}$, which is the shape of the attention score matrix, involved in all attention-related operators including batched general matrix multiplication and softmax. The latency cost model can be described as $\alpha_0 \cdot L^n + \alpha_1 \cdot L^n L^a + \beta$. Following LLM-PQ [34], we perform offline profiling to generate dummy sequences with different lengths and KV cache sizes, collecting the execution latencies to fit linear models for prefill and decode requests separately. Sequence lengths are sampled at exponential intervals of 2, with the maximum value being the product of the LLM context length and the maximum batch size. For batch execution time querying, we separate prefill and decode requests within the current batch, then map L^n of all prefill requests to cost models of prefill-phase model execution (L^a always equals L^n as no tokens are cached yet), and map L^n and L^a to cost models of decode-phase model execution. Finally, we sum up the execution time of requests in these two phases in the batch as t_{exec}.

As scheduled requests may have checkpointed KV cache in CPU memory (Sec. V), the potential swapping time from CPU memory to GPU memory also influences the iteration time. We conduct memory bandwidth profiling and develop a linear model to estimate the swapping time t_{swap} on M^{cpu}. With asynchronous memory operations, the memory swapping time overlaps with the model execution time, and the iteration time can be computed as $\max(t_{exec}, t_{swap})$.

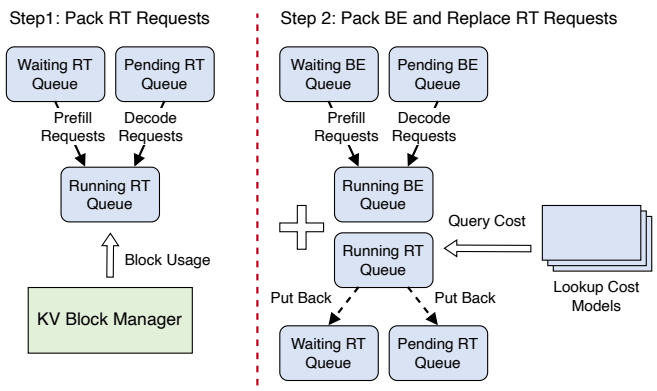


Fig. 6: Request scheduling process of the priority-based packing algorithm.

B. Hybrid Scheduling Algorithm

Problem (1) can be viewed as a variant of the 1D bin packing problem, where the maximum batch size is the bin capacity. Even ignoring the overhead of estimating the iteration time, there can be up to $\sum_{b_i=1}^{B_{curr}} \binom{N^{rt}+N^{be}}{b_i}$ combinations of requests to schedule, making the problem expensive to solve in runtime. We design a greedy priority-based packing algorithm to efficiently derive scheduling decisions for each iteration.

Main idea. As illustrated in Fig. 6, the scheduler first retrieves RT requests in ascending order of their remaining time to meet the token latency SLOs (TTFT or TPOT, depending on each request's prefill or decode phase). Next, the scheduler adds BE requests (in ascending order of CPU-to-GPU KV cache swapping overhead) to the running queue if the batch is not full, or replaces lower-priority RT requests (with more remaining time to SLO targets) with BE requests that introduce less swapping overheads. The scheduler schedules as many requests as possible, that do not exceed the current batch size and ensures that the estimated iteration time of the batch does not exceed the residual time to the SLO deadline of the most urgent RT request (i.e., the one closest to its deadline does not fail). In this manner, the scheduler can adapt system resources to dynamic workloads by flexibly determining the actual batch size for each iteration, based on the remaining time budget. After deciding the requests to schedule, the RT and BE running queues are merged to form a batch, which is dispatched to the executors.

Algorithm. The request scheduling algorithm is given in Alg. 1. We first order requests in the RT queues (including pending and waiting queues) altogether by their remaining time to the respective token generation deadlines in ascending order and decide the smallest residual time T_{min}^{res} (lines 3-4). We keep adding the most urgent RT requests from the queues to the batch until the batch size becomes B, no empty and preemptible cache blocks (determined by the function *FindPreemptBlock*, which obeys the rule in Sec. V) exist in the system, or we find that T_{min}^{res} can not be satisfied (lines 5-15). Then the BE queues are sorted by the number of checkpointed slots in ascending order (line 17). Each time, we select the BE

Algorithm 1: Priority-based Packing Algorithm for RT and BE request scheduling at each iteration

Input : Q^{rt} - RT request queues, Q^{be} - BE request queues, Mgr - KV block manager, B_{curr} - current batch size, $\mathcal{T}_{SLO}$ - SLO target of RT request
Output: RT_ready - scheduled RT requests, BE_ready - scheduled BE requests

1 $RT\_ready \leftarrow []$
2 *// Sort RT requests based on remaining time*
3 **sort** Q^{rt} **according to** $\mathcal{T}_{SLO} - t_i - (k_i - 1)T_{avg}$ in ascending order
4 $T_{min}^{res} \leftarrow \mathcal{T}_{SLO} - GetMinResidualTime(Q^{rt})$
5 **while** $\exists\ req^{rt} \in Q^{rt}$ **do**
6 **if** $Size(RT\_ready) == B_{curr}$ **then**
7 **break**
8 **if** ***not*** $FindPreemptBlock(req^{rt}, Mgr)$ **then**
9 **break**
10 $cand\_reqs \leftarrow RT\_ready + req^{rt}$
11 **if** $C(cand\_reqs) > T_{min}^{res}$ **then**
12 **break**
13 **front pop** req^{rt} **from** Q^{rt}
14 $RT\_ready \leftarrow RT\_ready \cup\ req^{rt}$
15 $BE\_ready \leftarrow []$
16 *// Sort BE requests based on* M_j^{cpu}
17 **sort** Q^{be} **according to** M_j^{cpu} in ascending order
18 **while** $\exists\ req^{be} \in Q^{be}$ **do**
19 **if** ***not*** $FindBlock(req^{be}, Mgr)$ **then**
20 **break**
21 *// First try packing*
22 **if** $Size(RT\_ready + BE\_ready + req^{be}) \leq B_{curr}$ ***and*** $C(RT\_ready + BE\_ready + req^{be}) \leq T_{min}^{res}$ **then**
23 $BE\_ready \leftarrow BE\_ready \cup\ req^{be}$
24 **else**
25 *// Then try replacing*
26 **back pop** req^{rt} **from** RT_ready
27 **if** $Size(RT\_ready + BE\_ready + req^{be}) \leq B_{curr}$ ***and*** $C(RT\_ready + BE\_ready + req^{be}) \leq T_{min}^{res}$ **then**
28 **front insert** req^{rt} **to** Q^{rt}
29 $BE\_ready \leftarrow BE\_ready \cup\ req^{be}$
30 **else**
31 $RT\_ready \leftarrow RT\_ready \cup\ req^{rt}$
32 **break**
33 **front pop** req^{be} **from** Q^{be}

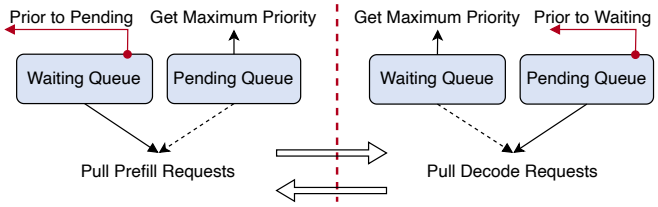


Fig. 7: Queues polling to pull RT requests in order.

request with the least potential memory swapping overhead and try to add the BE request to the batch or replace the least urgent RT request with the BE request (lines 18-33).

Request pulling optimization. When pulling RT requests, Alg. 1 requires sorting requests across different state queues (pending and waiting) at each iteration. Naive implementation by merging different queues before sorting and splitting them back after the two-step request selection introduces non-negligible overheads, especially when there exist numerous concurrent requests. To mitigate scheduling overhead, we opt for partial sorting and continuous queue polling. The key insight is that the waiting queue is inherently ordered (by request arrival time). Requests closer to the head of the waiting queue have longer elapsed time. We only need to sort the pending queue and then do alternative queue polling. As depicted in Fig. 7, we keep retrieving requests of the maximum priority from one queue and pulling requests that are more prioritized from the other.

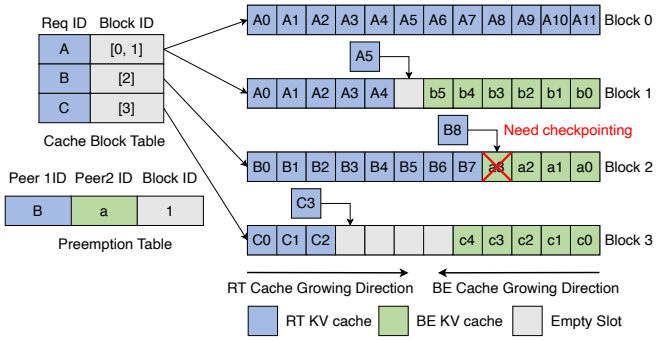


Fig. 8: KV cache storage layout for RT and BE requests. We only show the RT cache block table for clarity of the figure.

Adaptive Batch Sizing. Before selecting requests from RT and BE queues, the current batch size B_{curr} should be determined to strike a balance between low RT request latency and high BE request throughput. Drawing inspiration from the window size adjustment mechanism in TCP Congestion Control, we propose a heuristic to dynamically adapt the batch size based on the runtime SLO attainments of RT requests. Our approach operates as follows:

▷ *Initialization:* We start with a conservative batch size, e.g., $B_{curr} \leftarrow B_{base} = 128$.

▷ *gradual Expansion:* At each iteration, if the scheduler does not trigger the early-return in lines 11-12 of Alg. 1, which means that no RT requests are detected, we increase the batch size: $B_{curr} \leftarrow B_{curr} \cdot 2$ to include more BE requests for batching.

▷ *Aggressive Backoff:* Once the estimated iteration time surpasses the residual timev T_{min}^{res}, we apply an aggressive penalty by resetting the current batch size B_{curr} back to B_{base}.

V. Bidirectional KV Cache Management

Since request scheduling is constrained by available memory in the system for KV caching, it is crucial to efficiently allocate and manage memory resources (for KV cache storage) among concurrent requests. BROS's KV cache management adopts a bidirectional storage layout, along with the block preemption and lazy checkpointing mechanisms to flexibly share and allocate GPU memory for RT/BE requests.

A. Block Structure

KV cache storage renders a major memory bottleneck in LLM serving [19]. To manage the KV cache of RT/BE

requests, we organize GPU memory for KV cache storage into blocks and each block includes several slots, for storing tokens' KV tensors. Each block can be used for KV cache storage of one RT request and one BE request, with a bidirectional storage layout for sharing the block between two requests: KV cache of the RT request occupies memory slots from the left to the right in the block while that of the BE request in the opposite direction. As depicted in Fig. 8, RT requests A, B, and C occupy slots starting from the left end in blocks 0, 1, 2, and 3, respectively, while BE requests b, a, and c use slots from the right end of the respective blocks. An RT or BE request can occupy more than one block if its KV cache is large.

B. Block Management

Allocation and preemption. When a BE request is scheduled in an iteration, function *FindBlock* in Alg. 1 implements the allocation logic: (i) if it is a prefill request whose KV cache has not been stored, we find available GPU blocks for it according to the maximum number of empty slots in each block; (ii) if the request's KV cache has been stored in some GPU blocks, we continue growing its KV cache in the last block (if the empty slots of this block are not used up by itself) or start occupying a new block (otherwise). For RT requests, if no GPU blocks are available, *FindPreemptBlock* helps them preempt BE cache blocks and store the KV tensors in the opposite direction. A straightforward heuristic is adopted to always preempt BE cache blocks with the most empty slots. This approach ensures that urgent RT requests will not fail to be scheduled due to insufficient GPU blocks with a large number of concurrent requests. In extreme cases where no BE-only cache blocks are available for preemption by newly scheduled urgent RT requests, we progressively drop requests: first BE, then RT, based on the remaining time, and swap out their cache blocks. Once a request generates all tokens, its occupied KV cache slots are released.

Lazy checkpointing. Within a block, if sufficient empty slots are still available, both RT and BE requests can consider the block as exclusively theirs and safely store their most recent KV tensors. When all empty slots are exhausted during inference, only the KV tensors of a specific request that are about to be overwritten by its peer need to be checkpointed in CPU memory, instead of the request's entire KV cache [19], [27], and this block is set to preemption status. Those missing KV tensors are swapped back into GPU memory when the request is scheduled in future iterations. For example, in Fig. 8, KV tensors a3 of BE request *a* is preempted by B8 of RT request *B*, and a3 needs to be checkpointed to the CPU memory. This design removes unnecessary swapping operations, reducing memory overheads. To indicate slot preemption of the blocks, the KV block manager maintains a preemption table, which records the indices of all blocks where slot preemption happened and the indices of RT/BE requests sharing those blocks. When serving requests with preempted KV cache slots, the scheduler checks the preemption table and generates memory-swapping instructions for the executors to perform corresponding operations. Before dispatching scheduled requests to the executors, all swapping operations are consolidated and conducted asynchronously to enhance overall efficiency.

TABLE II: Workload statistics. Unit: token.

Type	Avg. Prompt	Std. Prompt	Avg. Output	Std. Output
RT-SharedGPT	222.76	256.36	234.51	268.50
BE-Synthetic	784.14	151.17	80.96	28.23

VI. Implementation

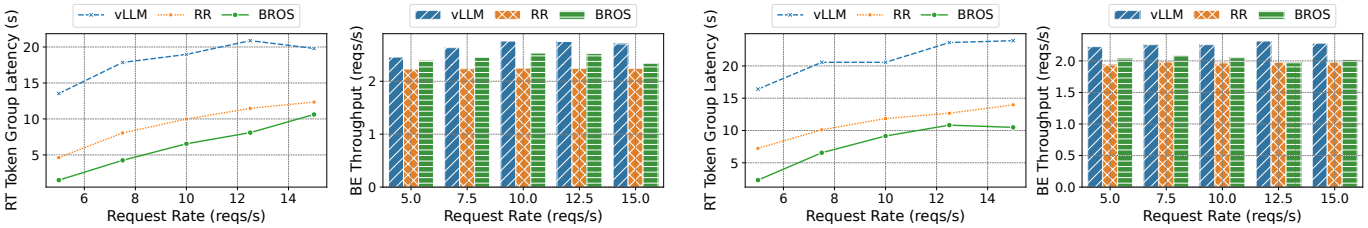
We implement BROS using 8.5k LoC in Python and 2k LoC in C++/CUDA. The RPC messages between the engine and the executors are handled with Ray [35], and the tensor transmission is implemented via NCCL [36]. We implement LLMs [5], [37], [38] with the operators provided by PyTorch [39] and Xformers [40], aligning our implementation with Hugging Face [41] to load pre-trained model weights. To enable the bidirectional block layout of the KV cache, we modify the PagedAttention kernel [19]. Our modifications include a binary direction table, which possesses an identical shape to the block tables. The purpose of this table is to guide the physical block offset for each warp, indicating whether threads should iterate from left or vice versa. Furthermore, we leverage PTX (Parallel Thread Execution) instructions to invert the order of elements within the loaded value vector, whenever the flag of direction is evaluated to be true. For checkpointing, we use different CUDA streams to handle asynchronous data movement events across GPU and CPU memory during runtime and customize CUDA kernels to efficiently fetch and store KV caches in non-continuous GPU memory.

VII. Evaluation

A. Experimental Setup

Models and testbed. We run Qwen2.5 [5] models with 14B, 32B parameters. We use a server (Ubuntu 20.04LTS) with 4 NVIDIA A100-SXM4-40GB GPUs interconnected by NVLink. All models are served with tensor parallelism with parallel degrees of Qwen2.5-14B, OPT-32B being 2 and 4, respectively. To ensure a fair comparison, we employ greedy sampling as the decoding algorithm for all LLMs, compelling the models to generate a specified number of tokens.

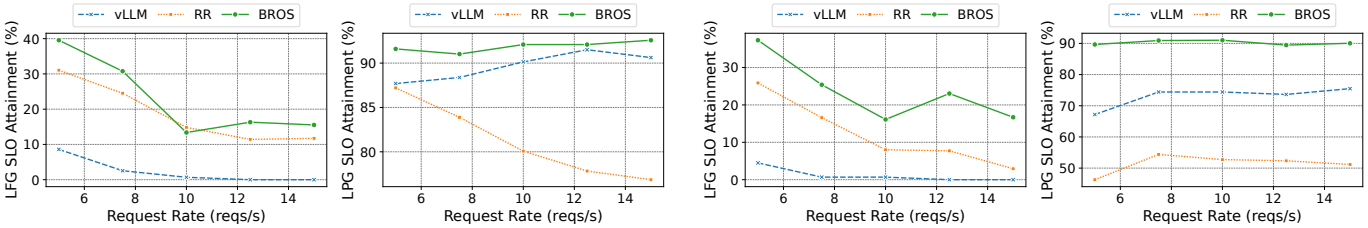
Traces. Following vLLM [19], we synthesize RT requests based on the ShareGPT [33] dataset and LMSYS-CHAT-1M datasets [42], both of which contain real-world user conversations with LLM services. Since no representative BE request dataset is available, we follow FlexGen [20] to use synthetic workloads. All prompt and output lengths for BE requests are sampled from uniform distributions with ranges (512, 1024) and (32, 128). Similar to previous works [17]–[19], [23], we generate the arrival times of RT requests using Poisson distribution with different request rates, while BE requests are periodically submitted in a batch manner when previous batched requests are completed [8]. Further details of the workloads are in Table II.



(a) Qwen2.5-14B, Hybrid Workloads.

(b) Qwen2.5-32B, Hybrid Workloads.

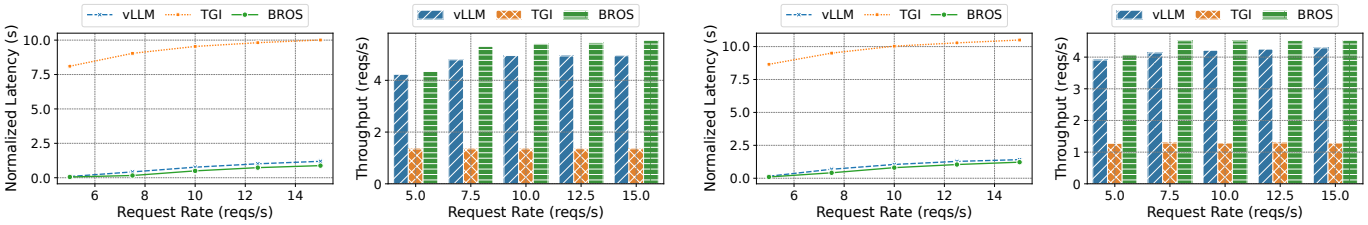
Fig. 9: Comparison of latency and throughput among BROS, vLLM, and RR on RT/BE hybrid workloads.



(a) Qwen2.5-14B, Hybrid Workloads.

(b) Qwen2.5-32B, Hybrid Workloads.

Fig. 10: Comparison of SLO attainments among BROS, vLLM, and RR on RT/BE hybrid workloads.



(a) Qwen2.5-14B, 2GPUs, RT Workloads.

(b) Qwen2.5-32B, 4GPUs, RT Workload.

Fig. 11: Comparison of latency and throughput among BROS, vLLM, and TGI on pure RT workloads.

Baselines. We compare BROS with three baselines: (1) *vLLM*, the SOTA LLM serving system optimized for high-throughput text generation; (2) *TGI* [32], an production-grade LLM serving system used in LLM services at Hugging Face, to support applications like Hugging Chat; (3) *BROS-Round-Robin* (referred to as RR), where we replace the scheduling algorithm of BROS with a Round-Robin schedule described in Sec. II-B and remove the bidirectional KV cache management. We do not include a comparison with offline LLM serving systems, such as FlexGen [20] and LLM-PQ [34]. This is because they do not implement continous batching [18], thereby introducing substantial overhead when co-serving RT requests.

Metrics. We measure the following metrics on RT requests: (1) *normalized latency* [18], [19], computed as the mean of each RT request's end-to-end latency divided by the number of its output tokens; (2) *time to first token* (TTFT), representing the latency for each RT request to return the first token after submitting its prompt; (3) *time per output token* (TPOT), indicating the average latency for generating subsequent tokens. We set the SLO for TPOT to be 0.2s, which is a widely accepted latency for one forward pass of a DNN [28], [43]. For TTFT, considering that prefill processing takes longer [17], [44], we set its SLO to be 0.4s. We use throughput (measured in requests per second) to evaluate the serving performance of BE requests. We show the average normalized latency and SLO attainments, computed as the mean of each request's normalized latency and SLO attainment. For all experiments, we evaluate the systems with 10-minute traces.

B. Striking Better Trade-off on Hybrid RT/BE Workloads

We evaluate the end-to-end performance of BROS with vLLM and RR on hybrid RT and BE requests. Performance comparisons are demonstrated in Fig. 9 and Fig. 10.

1) Latency and Throughput: Fig. 9a and Fig. 9b show that when RT requests are sampled from the SharedGPT dataset, BROS demonstrates reductions of 74.20% and 54.31% in average normalized latency, as compared to vLLM and RR at various request rates. Simultaneously, BROS maintains high throughput for BE requests, with only an 11.29% reduction compared to vLLM and an 8.80% reduction compared to RR. The performance gains can be attributed to our scheduling mechanism, which identifies different urgent RT requests at each iteration and assigns them higher priorities, surpassing the reliance on arrival orders or fair scheduling with BE

requests. The second step of the scheduling algorithm focuses on replacing non-urgent RT requests and maximizing the scheduling of BE requests. Furthermore, the bidirectional KV cache management of BROS allows RT to share KV cache blocks of BE requests, ensuring sufficient memory for urgent RT requests. As a result, BROS strikes a better trade-off by sacrificing a small proportion of BE requests' throughput in exchange for substantial improvements in RT requests' latency.

2) SLO Attainments: BROS demonstrates 36.38× and 1.73× improvements in TTFT and TPOT SLO attainments over vLLM. Compared to RR, BROS achieves improvements of 11.75× and 1.47× in TTFT and TPOT. These improvements are due to the designs of BROS's scheduling targets for RT requests, which consider the quick response and high context rate simultaneously when making decisions. Additionally, the adaptive batch sizing mechanism assists in regulating the maximum number of batched requests based on feedback from RT request SLO attainment. No baseline method effectively balances TTFT and TPOT SLO attainments. vLLM achieves an average TTFT SLO attainment of only 1.28%. RR alleviates this by serving RT and BE requests alternately at each iteration, but the results (10.55%) are still unsatisfactory. Regarding TPOT SLO attainment, vLLM is expected to achieve good results since when its running queue is full, all scheduled requests can complete generation without further preemption. However, without proper memory contention management, later-scheduled RT requests are compelled to release memory due to insufficient available resources; these requests must wait until earlier RT/BE requests finish generation, regardless of their urgency. BROS effectively addresses this issue through bidirectional block sharing and block preemption, resulting in higher TPOT attainments.

C. Achieving Better Performance on Pure RT Workloads

We compare BROS with vLLM and TGI on pure RT workloads. We use the RT requests from Table II, and eliminate the second part of the priority-based packing algorithm and the bidirectional KV cache management as well. We evaluate the systems' serving performance with normalized latency and throughput, consistent with those in previous studies [18], [19]. Fig. 11 shows that BROS still achieves a 29.36% reduction in latency and 1.08× throughput improvement over vLLM. Despite scheduling overheads, BROS can outperform SOTA systems in end-to-end performance.

D. benefits of Bi-Cache Mechanism

In Fig. 12, we compare the serving performance of RT requests with/without the bidirectional KV cache management (denoted as With Bi-cache and No Bi-cache, respectively). We serve Qwen2.5-32B on hybrid workloads, where the request rate is fixed to 10. In the case without bi-directional cache management, we simply perform recomputation for requests failing to acquire cache blocks. With our designs, TTFT and TPOT SLO attainments are 1.23× and 1.63× better than without, which verifies the benefits of our block layout for efficient resource sharing and better SLO attainments of RT requests.

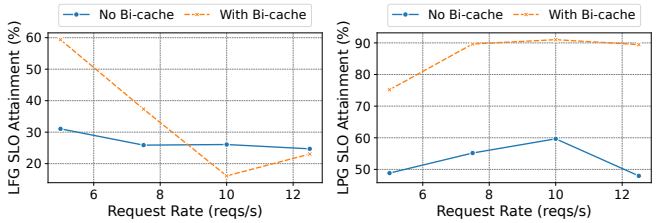
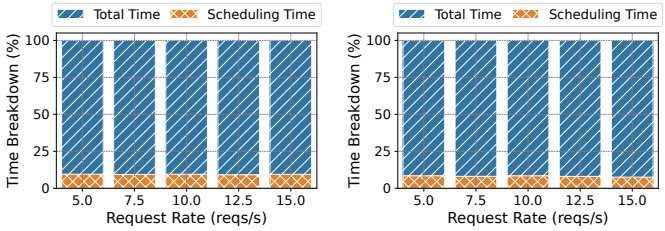


Fig. 12: TTFT and TPOT SLO attainment: with and without bi-directional cache management.



(a) Qwen2.5-14B. (b) Qwen2.5-32B.

Fig. 13: Scheduling overhead.

E. Scheduling Overheads

We evaluate the overheads of request scheduling, which involves sorting RT/BE requests and querying the cost model at each iteration. We run Qwen2.5-14B and Qwen2.5-32B models with hybrid RT and BE requests as in Sec. VII-B. We measure the average time to complete scheduling and the average total time of each iteration, depicting the breakdown. Fig. 13 illustrates that the overhead ranges from 9.12% to 9.52% per iteration for Qwen2.5-14B and from 7.56% to 8.42% for Qwen2.5-32B. Such overheads are justified with the gains achieved by BROS in terms of reducing RT request latency and improving normalized latency SLO attainment.

VIII. Conclusion

This paper presents BROS, an LLM serving system for hybrid RT/BE requests. We define novel *token group* based latency metrics to balance quick response and high context rate requirements for RT serving, propose a priority-based packing algorithm for preemptive scheduling of RT/BE requests at the iteration level, achieving a favorable serving trade-off between low latency for RT requests and high throughput for BE requests. To handle memory contention, we advocate bidirectional KV cache management, which properly shares cache blocks between concurrent RT/BE requests to improve memory utilization and reduce overheads. Extensive experiments validate the superiority of BROS over existing LLM serving systems [19], [32].

IX. Acknowledgment

This work was supported in part by grants from Hong Kong RGC under the contracts 17204423 (GRF), 17205824 (GRF), C7004-22G (CRF) and T43-513/23-N (TRS).

REFERENCES

[1] A. Radford, K. Narasimhan, T. Salimans, I. Sutskever *et al.*, "Improving language understanding by generative pre-training," 2018.

[2] A. Liu, B. Feng, B. Xue, B. Wang, B. Wu, C. Lu, C. Zhao, C. Deng, C. Zhang, C. Ruan *et al.*, "Deepseek-v3 technical report," *arXiv preprint arXiv:2412.19437*, 2024.

[3] G. Team, P. Georgiev, V. I. Lei, R. Burnell, L. Bai, A. Gulati, G. Tanzer, D. Vincent, Z. Pan, S. Wang *et al.*, "Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context," *arXiv preprint arXiv:2403.05530*, 2024.

[4] A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Yang, A. Fan *et al.*, "The llama 3 herd of models," *arXiv preprint arXiv:2407.21783*, 2024.

[5] A. Yang, A. Li, B. Yang, B. Zhang, B. Hui, B. Zheng, B. Yu, C. Gao, C. Huang, C. Lv *et al.*, "Qwen2.5 technical report," *arXiv preprint arXiv:2505.09388*, 2025.

[6] Character AI Team, "character.ai," https://character.ai/, 2024.

[7] Github, "Copilot: Your AI pair programmer," https://github.com/features/copilot, 2022.

[8] OpenAI Team, "Batch API FAQ," https://help.openai.com/en/articles/9197833-batch-api-faq, 2024.

[9] S. Arora, B. Yang, S. Eyuboglu, A. Narayan, A. Hojel, I. Trummer, and C. Ré, "Language models enable simple systems for generating structured views of heterogeneous data lakes," *arXiv preprint arXiv:2304.09433*, 2023.

[10] A. Narayan, I. Chami, L. Orr, S. Arora, and C. Ré, "Can foundation models wrangle your data?" *arXiv preprint arXiv:2205.09911*, 2022.

[11] J. Wei, M. Bosma, V. Y. Zhao, K. Guu, A. W. Yu, B. Lester, N. Du, A. M. Dai, and Q. V. Le, "Finetuned language models are zero-shot learners," *arXiv preprint arXiv:2109.01652*, 2021.

[12] Y. Wang, S. Mishra, P. Alipoormolabashi, Y. Kordi, A. Mirzaei, A. Arunkumar, A. Ashok, A. S. Dhanasekaran, A. Naik, D. Stap *et al.*, "Super-naturalinstructions: Generalization via declarative instructions on 1600+ nlp tasks," *arXiv preprint arXiv:2204.07705*, 2022.

[13] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[14] Z. Shao, P. Wang, Q. Zhu, R. Xu, J. Song, X. Bi, H. Zhang, M. Zhang, Y. Li, Y. Wu *et al.*, "Deepseekmath: Pushing the limits of mathematical reasoning in open language models," *arXiv preprint arXiv:2402.03300*, 2024.

[15] OpenAI, "Introducing ChatGPT," https://openai.com/blog/chatgpt, 2022.

[16] "Token streaming," https://huggingface.co/docs/text-generation-inference/conceptual/streaming, 2023.

[17] Y. Zhong, S. Liu, J. Chen, J. Hu, Y. Zhu, X. Liu, X. Jin, and H. Zhang, "Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving," *arXiv preprint arXiv:2401.09670*, 2024.

[18] G.-I. Yu, J. S. Jeong, G.-W. Kim, S. Kim, and B.-G. Chun, "Orca: A distributed serving system for {Transformer-Based} generative models," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022, pp. 521–538.

[19] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with pagedattention," *arXiv preprint arXiv:2309.06180*, 2023.

[20] Y. Sheng, L. Zheng, B. Yuan, Z. Li, M. Ryabinin, D. Y. Fu, Z. Xie, B. Chen, C. Barrett, J. E. Gonzalez *et al.*, "High-throughput generative inference of large language models with a single gpu," *arXiv preprint arXiv:2303.06865*, 2023.

[21] A. Agrawal, N. Kedia, A. Panwar, J. Mohan, N. Kwatra, B. S. Gulavani, A. Tumanov, and R. Ramjee, "Taming throughput-latency tradeoff in llm inference with sarathi-serve," *arXiv preprint arXiv:2403.02310*, 2024.

[22] Y. Zhao, S. Yang, K. Zhu, L. Zheng, B. Kasikci, Y. Zhou, J. Xing, and I. Stoica, "Blendserve: Optimizing offline inference for auto-regressive large models with resource-aware batching," *arXiv preprint arXiv:2411.16102*, 2024.

[23] B. Sun, Z. Huang, H. Zhao, W. Xiao, X. Zhang, Y. Li, and W. Lin, "Llumnix: Dynamic scheduling for large language model serving," *arXiv preprint arXiv:2406.03243*, 2024.

[24] J. Stojkovic, C. Zhang, Í. Goiri, J. Torrellas, and E. Choukse, "Dynamollm: Designing llm inference clusters for performance and energy efficiency," *arXiv preprint arXiv:2408.00741*, 2024.

[25] Y. Wang, Y. Chen, Z. Li, X. Kang, Z. Tang, X. He, R. Guo, X. Wang, Q. Wang, A. C. Zhou *et al.*, "Burstgpt: A real-world workload dataset to optimize llm serving systems," *arXiv preprint arXiv:2401.17644*, 2024.

[26] Z. Li, L. Zheng, Y. Zhong, V. Liu, Y. Sheng, X. Jin, Y. Huang, Z. Chen, H. Zhang, J. E. Gonzalez, and I. Stoica, "AlpaServe: Statistical multiplexing with model parallelism for deep learning serving," in *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. Boston, MA: USENIX Association, Jul. 2023, pp. 663–679. [Online]. Available: https://www.usenix.org/conference/osdi23/presentation/li-zhouhan

[27] B. Wu, Y. Zhong, Z. Zhang, G. Huang, X. Liu, and X. Jin, "Fast distributed inference serving for large language models," *arXiv preprint arXiv:2305.05920*, 2023.

[28] W. Cui, H. Zhao, Q. Chen, H. Wei, Z. Li, D. Zeng, C. Li, and M. Guo, "DVABatch: Diversity-aware Multi-Entry Multi-Exit batching for efficient processing of DNN services on GPUs," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. Carlsbad, CA: USENIX Association, Jul. 2022, pp. 183–198. [Online]. Available: https://www.usenix.org/conference/atc22/presentation/cui

[29] C. Olston, N. Fiedel, K. Gorovoy, J. Harmsen, L. Lao, F. Li, V. Rajashekhar, S. Ramesh, and J. Soyke, "Tensorflow-serving: Flexible, high-performance ml serving," *arXiv preprint arXiv:1712.06139*, 2017.

[30] "Nvidia triton inference server," https://developer.nvidia.com/nvidia-triton-inference-server, 2021.

[31] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[32] HuggingFace, "Text Generation Inference," https://github.com/huggingface/text-generation-inference, 2023.

[33] ShareGPT Team, "ShareGPT." https://sharegpt.com/, 2023.

[34] J. Zhao, B. Wan, Y. Peng, H. Lin, and C. Wu, "Llm-pq: Serving llm on heterogeneous clusters with phase-aware partition and adaptive quantization," *arXiv preprint arXiv:2403.01136*, 2024.

[35] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan *et al.*, "Ray: A distributed framework for emerging {AI} applications," in *13th USENIX symposium on operating systems design and implementation (OSDI 18)*, 2018, pp. 561–577.

[36] NVIDIA., "NVIDIA Collective Communications Library (NCCL)." https://developer.nvidia.com/nccl, 2023.

[37] S. Zhang, S. Roller, N. Goyal, M. Artetxe, M. Chen, S. Chen, C. Dewan, M. Diab, X. Li, X. V. Lin *et al.*, "Opt: Open pre-trained transformer language models," *arXiv preprint arXiv:2205.01068*, 2022.

[38] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale *et al.*, "Llama 2: Open foundation and fine-tuned chat models," *arXiv preprint arXiv:2307.09288*, 2023.

[39] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[40] B. Lefaudeux, F. Massa, D. Liskovich, W. Xiong, V. Caggiano, S. Naren, M. Xu, J. Hu, M. Tintore, S. Zhang, P. Labatut, and D. Haziza, "xformers: A modular and hackable transformer modelling library," https://github.com/facebookresearch/xformers, 2022.

[41] Hugging Face, "Hugging Face." https://huggingface.co/, 2023.

[42] L. Zheng, W.-L. Chiang, Y. Sheng, T. Li, S. Zhuang, Z. Wu, Y. Zhuang, Z. Li, Z. Lin, E. Xing *et al.*, "Lmsys-chat-1m: A large-scale real-world llm conversation dataset," *arXiv preprint arXiv:2309.11998*, 2023.

[43] H. Shen, L. Chen, Y. Jin, L. Zhao, B. Kong, M. Philipose, A. Krishnamurthy, and R. Sundaram, "Nexus: A gpu cluster engine for accelerating dnn-based video analysis," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 322–337.

[44] C. Holmes, M. Tanaka, M. Wyatt, A. A. Awan, J. Rasley, S. Rajbhandari, R. Y. Aminabadi, H. Qin, A. Bakhtiari, L. Kurilenko *et al.*, "Deepspeed-fastgen: High-throughput text generation for llms via mii and deepspeed-inference," *arXiv preprint arXiv:2401.08671*, 2024.