gLLM: Global Balanced Pipeline Parallelism Systems for Distributed LLMs Serving with Token Throttling

Tianyu Guo
Sun Yat-sen University
Guangzhou, China
guoty9@mail2.sysu.edu.cn

Xianwei Zhang*
Sun Yat-sen University
Guangzhou, China
zhangxw79@mail.sysu.edu.cn

Jiangsu Du
Sun Yat-sen University
Guangzhou, China
dujiangsu@mail.sysu.edu.cn

Zhiguang Chen*
Sun Yat-sen University
Guangzhou, China
chenzhg29@mail.sysu.edu.cn

Nong Xiao
Sun Yat-sen University
Guangzhou, China
xiaon6@mail.sysu.edu.cn

Yutong Lu
Sun Yat-sen University
Guangzhou, China
luyutong@mail.sysu.edu.cn

Abstract

Pipeline parallelism has emerged as a predominant approach for deploying large language models (LLMs) across distributed nodes, owing to its lower communication overhead compared to tensor parallelism. While demonstrating high throughput in request serving, pipeline parallelism often faces performance limitations caused by pipeline bubbles, which are primarily resulted from imbalanced computation delays across batches. Existing methods like Sarathi-Serve attempt to address this through hybrid scheduling of chunked prefill and decode tokens with a fixed token budget. However, such methods may still experience significant fluctuations, arising either from insufficient prefill tokens or uneven distribution of decode tokens, ultimately leading to computational imbalance.

To overcome these inefficiencies, we present gLLM, a globally balanced pipeline parallelism system incorporating Token Throttling to effectively mitigate the pipeline bubbles. Our Token Throttling mechanism is a fine-grained scheduling policy that independently regulates the quantities of prefill and decode tokens, thus enabling balanced computation by leveraging global information from the inference system. Specifically, for decode tokens, gLLM maintains near-consistent token count across processing batches. For prefill tokens, it dynamically adjusts batch sizes based on both total pending tokens and the memory utilization rates of key-value cache (KV cache). Furthermore, gLLM runtime adopts an asynchronous execution and message passing architecture specifically optimized for pipeline parallelism characteristics. Experimental evaluations with representative LLMs show that gLLM achieves significant performance improvements, delivering 11% to 398% higher maximum throughput compared to state-of-the-art pipeline or tensor parallelism systems, while simultaneously maintaining lower latency. We have open-sourced our code at https://github.com/gty111/gLLM.

*Corresponding author.

CCS Concepts

• Computing methodologies → Distributed computing methodologies; **• Computer systems organization → Parallel architectures**; **Real-time system architecture**.

Keywords

LLM, Inference, Parallelism, Pipeline Bubbles, Throttling, Runtime

ACM Reference Format:
Tianyu Guo, Xianwei Zhang, Jiangsu Du, Zhiguang Chen, Nong Xiao, and Yutong Lu. 2025. gLLM: Global Balanced Pipeline Parallelism Systems for Distributed LLMs Serving with Token Throttling. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '25), November 16–21, 2025, St Louis, MO, USA*. ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3712285.3759823

1 INTRODUCTION

Large language models (LLMs) have demonstrated remarkable capabilities in performing complex tasks like logical reasoning [22, 46, 48], mathematical problem solving [10, 23, 25, 28, 41, 60] and agent acting [9, 29, 40, 44, 49, 55]. As model sizes scale to hundreds of billion or even trillions parameters [3, 8, 13, 14, 34, 38], distributed serving of LLMs [4, 30, 45, 57] has become essential to overcome GPU memory constraints. Among distributed approaches, pipeline parallelism has emerged as a predominant method for training and serving models of both deep neural networks (DNNs) [15, 16, 31] and LLMs [2, 19, 24, 26, 27, 32, 35, 36, 47, 63], primarily due to its low communication overhead that effectively mitigates bandwidth limitations. However, this method often incurs pipeline bubbles, i.e., periods of GPU inactivity where subsequent pipeline stages must wait for prior ones to complete[2]. These inefficient bubbles mainly stem from computational imbalance between micro-batches. While existing researches have focused on optimizing pipeline bubbles in training scenarios [2, 19, 24, 26, 27, 32, 36, 47], recent methods like Sarathi-Serve [2] and prefill-decode disaggregated architectures [35, 58, 63] aim to address computational imbalances of inferences, specifically between prefill and decode stages. Our analysis reveals that these solutions remain insufficient for effectively resolving the bubble issues.

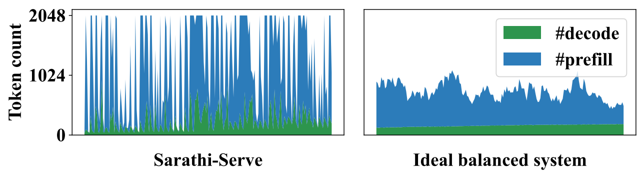


Figure 1: A comparison of the scheduled token counts of prefill and decode stage between Sarathi-Serve and an ideal balanced system (horizontal axis represents iteration index in chronological order), with the maximum batched token count (token budget) set to 2048 for both systems.

Overall, each LLM serving request undergoes two distinct stages. The first stage, prefill, processes the prompt tokens to compute keys and values (KV cache), and then generates the initial output token. The second stage, decode, produces the remaining output tokens. The two stages exhibit markedly different computational characteristics, where prefill operations typically saturate GPU capacity, while decode operations demonstrate significantly lower compute utilization. Therefore, to enhance hardware efficiency, multiple decode tokens are commonly batched together for parallel processing. However, the interleaved execution of prefill and decode operations may lead to mutual interference. To augment the unbalanced batching policy, Sarathi-Serve [2] proposes to apply a hybrid scheduling that allocates a fixed token budget between chunked prefill and decode operations. While effective to some extent, this solution still fails to properly account for the critical prefill-to-decode ratio. In practice, decode tokens often lack sufficient corresponding prefill tokens for effective co-scheduling. Figure 1 highlights this disparity issue by comparing Sarathi-Serve with an optimally balanced system on batched token counts per iteration. It can be observed that Sarathi-Serve's scheduling results in substantially greater token count volatility compared to the balanced counterpart. These fluctuations can be attributed to two primary factors: (1) missed opportunities to batch decode tokens with prefill ones, and (2) uneven distribution of decode tokens across batches. These inefficiencies frequently induce pipeline bubbles that significantly degrade system performance. Whereas lowering token budget could theoretically smooth these fluctuations, such approach would disproportionately penalize prefill rates, ultimately constraining the overall system throughput.

To balance the divergent computational demands between prefill and decode stages, prefill-decode disaggregated architectures [35, 58, 63] have been recently proposed to alleviate prefill-decode interference and have achieved low cost budget for each stage respectively. Generally, these designs allocate the prefill and decode computations either to different nodes connected via KV cache transmission [35, 63], or schedule them at different times [58]. Despite separate computation of the prefill and decode stages, computational imbalance persists across either prefill or decode batches. Moreover, determining the optimal ratio of GPUs allocated to the prefill stage versus the decode stage becomes challenging under dynamically fluctuating request rates. The tight coupling between prefill and decode nodes also raises fault tolerance concerns. A failure in one stage could cascade to the other.

To effectively address the unbalanced problems and mitigate pipeline bubbles in LLM serving, we design gLLM, a globally efficient pipeline parallelism system that balances computation across micro-batches using Token Throttling. This mechanism dynamically regulates the number of tokens processed in the prefill and decode stages based on real-time system states. For decode tokens, gLLM distributes tokens evenly across micro-batches, while for prefill tokens, it balances load by considering both pending prefill tokens and KV cache utilization. By adaptively throttling token counts, gLLM achieves better load balancing across micro-batches and greatly alleviates the pipeline bubble issue. Moreover, considering the characteristic of pipeline parallelism, our augmented gLLM runtime incorporates an asynchronous execution and message passing architecture to reduce data dependency and CPU overhead.

The main contributions of this paper are as follows:

- Targeting LLM online serving, we highlight the observations that pipeline bubbles caused by unbalanced computation mainly stem from insufficient prefill tokens or reaching KV cache limitations.
- We design a Token Throttling mechanism that dynamically adjusts the batch sizes of the prefill and decode stage independently, guided by real-time inference status, to achieve balanced schedule.
- We present gLLM, a distributed serving system with Token Throttling for effectively balancing the computation across batches to alleviate pipeline bubbles.
- Experiments on representative LLM applications show that gLLM enhances maximum throughput by 11% to 398% over state-of-the-art pipeline and tensor parallelism systems, while simultaneously achieving lower latency.

2 BACKGROUND AND MOTIVATION

2.1 The Transformer Architecture

Modern large language models (LLMs) predominantly adopt the decoder-only Transformer architecture [51]. For such architecture, the input begins with an embedding layer, which maps token IDs into hidden states while incorporating positional encoding to preserve sequence order. Between them, self-attention is the core of the transformer, enabling LLMs to capture long-range contextual dependencies. After processing through the decoder layers, a linear projection layer transforms the final hidden states into logits, representing the probability distribution over the vocabulary. Finally, a sampling strategy (e.g., greedy, top-k, or nucleus sampling) is applied to select the next token for generation.

2.2 LLM Inference Procedure

Autoregressive Decoding. The procedure of LLM inference is autoregressive, where each token is generated based on all preceding ones through the computation of attention scores between their keys and values. To avoid recomputations, the keys and values are retained for subsequent steps [20, 56]. Based on that, LLM inference can be divided into two distinct phases: 1. Prefill: the prompt's tokens are processed in parallel to populate the KV cache and generate the first output token. This phase fully utilizes GPU compute resources because of high parallelism. 2. Decode: each new token is generated sequentially by appending to the last token and the stored KV cache. This phase often underutilizes the GPU, as it processes only one token per step and incurs frequent memory bandwidth bottlenecks from KV cache accesses. The unbalanced computation characteristic between prefill and decode stages may impose inefficiency in LLM inference [2, 35, 63].

Scheduling Policies. Traditional inference engine like FasterTransformer [33] employs batch-level scheduling which selects a group of requests and executes them until the completion of all the sequences. Without considering variable length characteristic of transformer architecture, this method delays early-finished and late-joining requests. Instead, Orca [56] overcomes this issue by proposing iteration-level scheduling, which allows requests to dynamically enter or exit a batch before each model forward pass. Whereas Orca can batch requests from both prefill and decode stages, it introduces generation stalls for ongoing decode requests due to high prefill computation latency. To solve the problem, Sarathi-Serve [2] allows computing large prefills in small chunks across several iterations and further hybrid scheduling of chunked prefill and decode tokens to achieve stall-free batching. Specifically, Sarathi-Serve first schedules all decode tokens, then maximizes chunked prefill tokens within the fixed token budget. However, in actual situation, decode tokens often lack the opportunity to mix with adequate prefill tokens.

2.3 Parallelism Strategies of LLM Inference

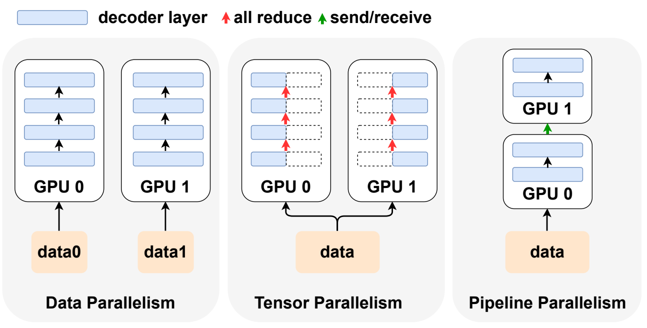


Figure 2: Comparison of data parallelism, tensor parallelism and pipeline parallelism.

The basic parallelism strategies in LLM inference primarily consist of data parallelism and model parallelism (as shown in Figure 2). Data parallelism splits the input data into multiple parts, sending each part to the corresponding GPU for parallel processing. Instead, model parallelism partitions the model itself, with each GPU being responsible for a different portion of the model. Model parallelism can be further classified into tensor parallelism and pipeline parallelism, depending on how the model is partitioned. Tensor parallelism employs intra-layer parallelism, splitting individual operations across GPUs and requiring frequent communication to synchronize results. Pipeline parallelism adopts inter-layer parallelism, assigning different layers to different GPUs and only requiring communications to pass intermediate activations between stages. In addition, pipeline parallelism employs multiple micro-batches to saturate GPUs at different pipeline stages. Pipeline depth refers to the number of sequential stages in a pipeline, where each stage performs a specific part of a task. Due to these differences, tensor parallelism can reduce forward latency, albeit at the expense of increased communication overhead. When integrated with chunked pipeline parallelism (CPP) [37], pipeline parallelism can further leverage intra-request parallelism. In online serving scenarios, tensor parallelism is more suitable for low request rates, while pipeline parallelism better handles high-throughput demands.

2.4 Challenges in Pipeline Parallelism

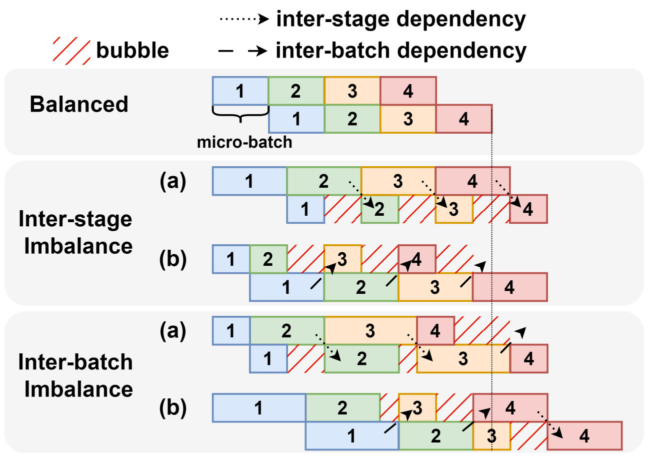
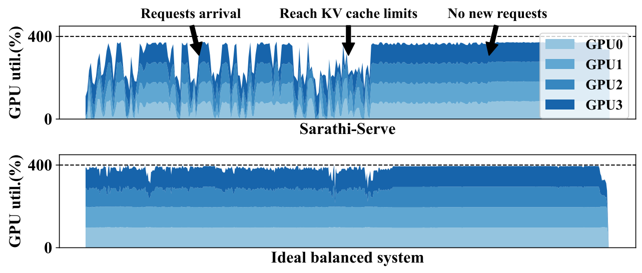


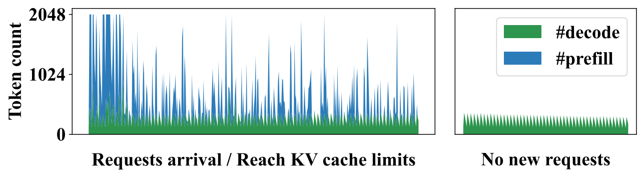
Figure 3: Pipeline bubbles caused by two types of imbalance, i.e., inter-stage and inter-batch. There are two situations (a) and (b) for each type of imblance. The horizontal axis represents the timeline, and the vertical axis represents the two-level pipeline stages. The number indicates the ordinal position of the micro-batch.

Pipeline Bubbles. Pipeline Parallelism reduces the demand for high-bandwidth communication but can incur problems of load imbalance and low GPU utilization. These load-balancing issues manifest as pipeline bubbles, i.e., periods of GPU idle time. Pipeline bubbles are mainly caused by two types of dependencies: (1) inter-stage dependency, where a stage cannot begin computation until the preceding stage completes, and (2) inter-batch dependency, where the number of concurrent micro-batches is limited by the pipeline depth. The load imbalance stems from: (1) inter-stage imbalance due to uneven computation distribution across pipeline stages, and (2) inter-batch imbalance caused by variation in computation requirements across different micro-batches. Figure 3 illustrates how these imbalances create pipeline bubbles. In this paper, we focus on solving inter-batch pipeline bubbles, while the inter-stage bubbles are left for future works.

Sarathi-Serve [2] identifies three types of pipeline bubbles, including variations in the number of prefill tokens, differences in compute time between prefill and decode, and variations in decode compute times, all of which are attributable to inter-batch imbalance. Among these, recent TD-Pipe [58] adopts the temporal prefill–decode disaggregation strategy and targets the second type of pipeline bubble, optimizing for high-throughput scenarios. Meanwhile, gLLM focuses on online serving scenarios and addresses the first type of pipeline bubble arising in the context of chunked prefill.



(a) Variation of GPU utilization



(b) Batched token count in Sarathi-Serve

Figure 4: Under-utilized GPU caused by unbalanced scheduling.

Insufficient GPU Utilization. Pipeline parallelism has long been plagued by low GPU utilization. Although Sarathi-Serve attempts to mitigate this issue, under-utilization persists in current systems. As shown in Figure 4 when serving a 32B model with 4 GPUs, their utilization follows a two-stage pattern: an initial phase with high fluctuations followed by a stable but suboptimal phase. By correlating these observations with request timing, we see that the initial stage coincides with incoming requests, forcing the system to handle both prefill and decode tokens. Once no new requests arrive, the system shifts to decoding only, leading to steadier but still inefficient utilization. Notably, batched token counts (Figure 4b) fluctuate throughout execution, particularly upon requests arrival or reaching KV cache limits. These irregular batch sizes introduce severe pipeline bubbles, which directly contribute to poor GPU utilization in both stages.

2.5 Scheduling Demands

Balanced Scheduling. Pipeline parallelism relies on the balanced computation across micro-batches, but current scheduling strategy fails to meet this requirement: (1) Prefill Imbalance. The fluctuations of batched prefill tokens count within the current scheduling strategy primarily stems from two key factors. First, prefill tokens depend on waiting requests. When no requests are available to prefill, the batched token count fluctuates. Second, prefill operations are constrained by KV cache utilization. If insufficient space exists to store the computed KV cache, the system halts the scheduling of prefill tokens. However, current scheduling method fails to account for these factors, resulting in unbalanced prefill scheduling. (2) Decode Imbalance. To achieve balanced processing during the decode stage, we aim to distribute the total decode requests as evenly as possible across all available micro-batches. While current scheduling strategy lacks explicit balancing considerations and may induce uneven workload distribution across micro-batches.

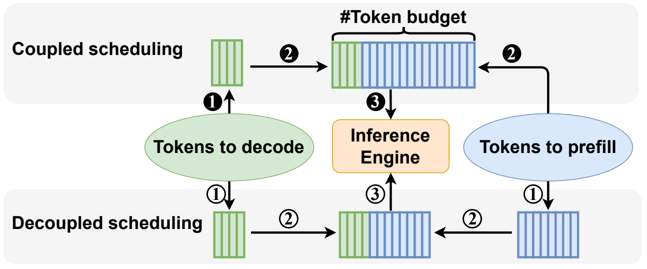


Figure 5: Comparison between coupled and decoupled scheduling.

Decoupled Scheduling. As shown in Figure 5, current method first schedules all decode tokens ❶, then maximizes chunked prefill tokens within the fixed token budget ❷. However, we should separately schedule balanced count of prefill and decode tokens ①, rather than limiting their total to a fixed value. This is because the scheduled tokens count may not reach the maximum budget and often causes the unbalanced schedule. Besides, the optimal batch tokens count tends to vary dynamically. Moreover, the tight coupling between prefill and decode scheduling often leads to interference between these stages. For instance, when numerous decode requests are in progress, the available token capacity for prefill becomes limited. Nevertheless, maximizing inference throughput requires processing large batches of prefill tokens. This fundamental conflict reveals that tightly coupling prefill and decode scheduling under a fixed total token budget cannot effectively satisfy their respective requirements. Thus there is a critical need to develop decoupled scheduling mechanisms.

Dynamic Scheduling. The number of prefill tokens per batch should adapt dynamically to the inference system's state. For example, at low KV cache utilization (i.e., when few requests are in decode), we should increase prefill speed to maximize GPU utilization. Conversely, during periods of high KV cache utilization, we are expected to reduce the prefill rate to prevent preemption of sequences for insufficient KV cache. Another concern is that if there are few tokens to prefill, we should prefill smoothly to avoid sudden fluctuation in batched tokens count. In the contrast, if there are abundant tokens to prefill, we are expected to maintain high prefill rates. However, current prefill batching is constrained by the fixed token budget and the number of active decode requests, rather than adapting to actual demands. This inflexibility leads to suboptimal scheduling policy, highlighting the need for a more adaptive scheduling approach.

3 DESIGN

In this section, we present gLLM, a global balanced pipeline parallelism system with Token Throttling mechanism. As shown in Figure 7, gLLM dynamically schedules prefill and decode tokens separately by throttling batched token counts according to real-time inference system states. After scheduling, gLLM merges scheduled

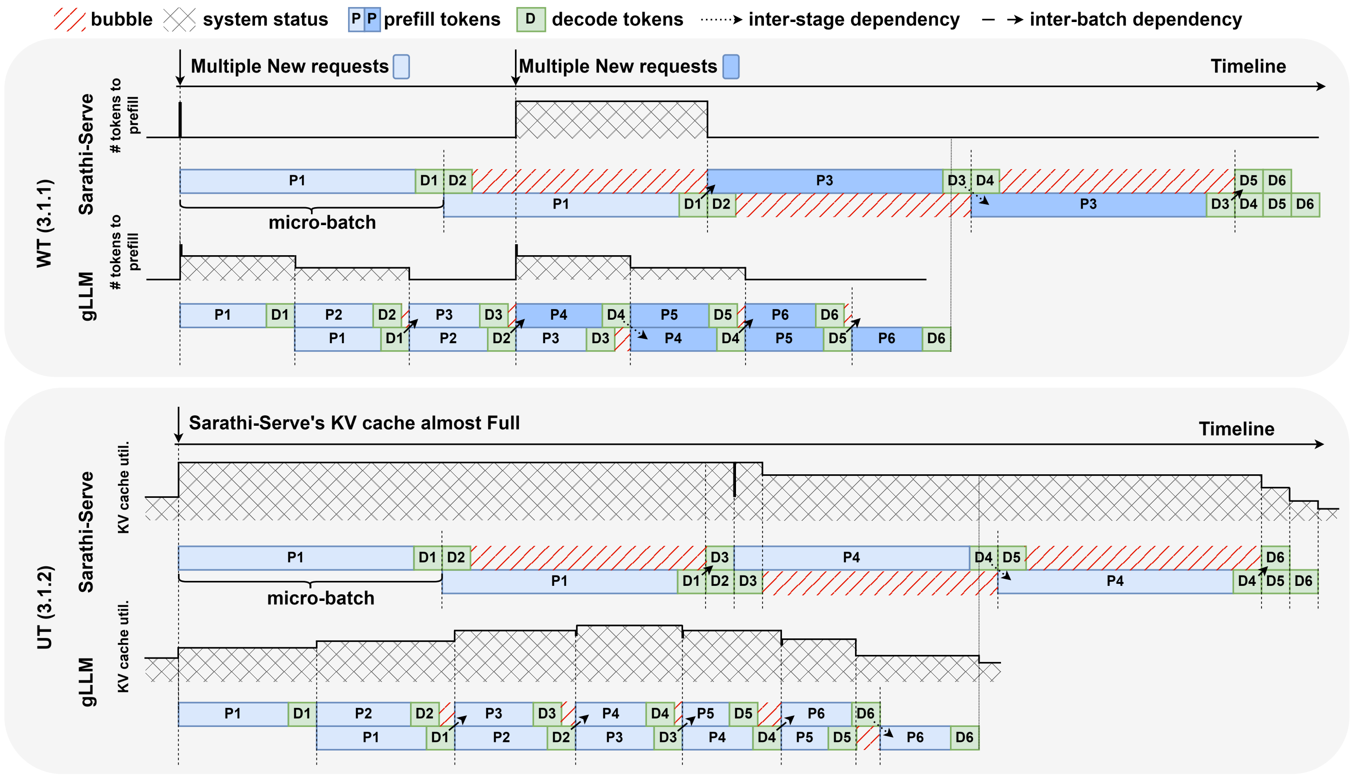


Figure 6: Case study (the pipeline depth is 2) of prefill Token Throttling. The number indicates the ordinal position of the micro-batch. For the second situation, the KV cache is allocated for prefill tokens prior to the execution of each micro-batch (at the begin of first stage). After processing a micro-batch (at the end of last stage), the KV cache usage may increase, decrease or stay roughly the same depending on how many requests meet the termination criterion. The KV cache usage is consistent across all GPUs since they share unified page tables.

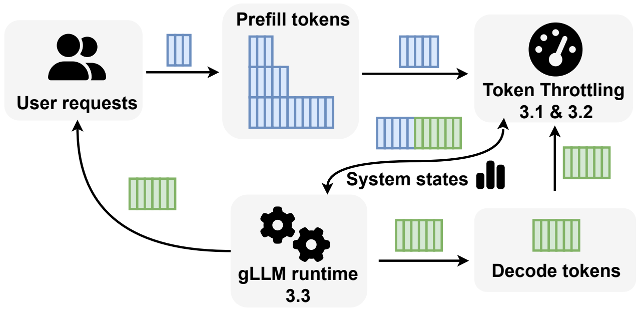


Figure 7: Overview of gLLM system. Inference procedure in gLLM consists of tokens flowing through its components.

prefill and decode tokens into a single batch. In addition, gLLM runtime adopts asynchronous architectures, hiding CPU operations overhead.

3.1 Prefill Token Throttling

The prefill operation is the first step in LLM inference, where the prompt's KV cache is computed and the first output token is generated. Therefore, prefill scheduling hinges on two factors: the number of tokens waiting for prefill and the system's KV cache utilization. First, if no requests are pending prefill, the operation cannot be scheduled. Second, if the KV cache is near capacity, prefill cannot proceed due to insufficient memory. Beyond these constraints, the scale of these factors also influences scheduling decisions. When few requests are waiting or KV cache usage is high, the system should reduce the prefill rate to avoid pipeline bubbles or requests preemption. Conversely, when many requests are queued or KV cache availability is high, the system should increase the prefill rate to maximize throughput. To balance these factors, we throttle the prefill token count based on both the volume of pending tokens and current KV cache pressure.

3.1.1 Throttling by Tokens Count Awaiting Prefill (WT). During each schedule, gLLM collects the number of tokens across all awaiting prefill requests (#*WP*) to determine the batched prefill token count (#*P*). This decision relies on three hyperparameters, minimum or maximum batched token count of prefill (#*MinP*/#*MaxP*) and the number of iterations (#*T*) required to process all tokens waiting for prefill. The batched prefill token count can be calculated as follows:

$$\#P = min(max(\frac{\#WP}{\#T}, \#MinP), \#MaxP) \quad (1)$$

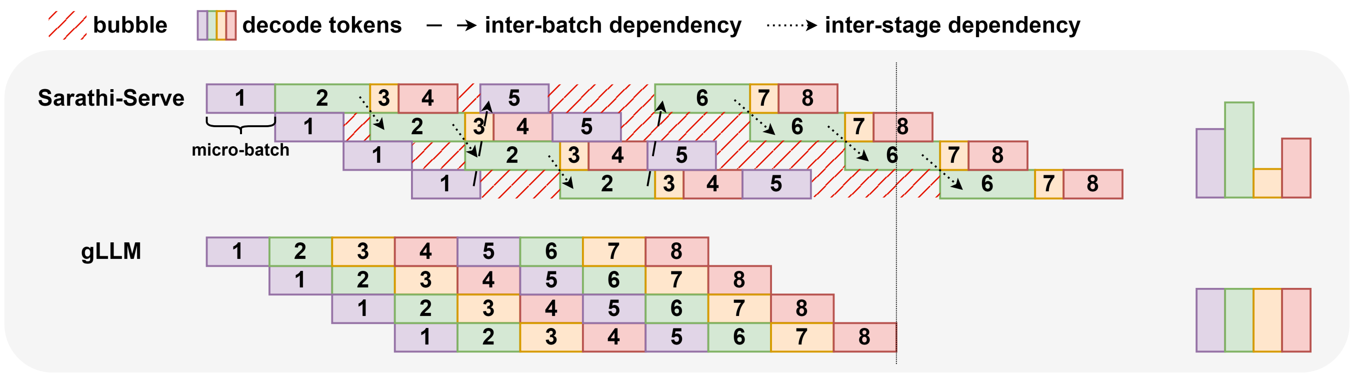


Figure 8: Case study (the pipeline depth is 4) of decode Token Throttling. The number indicates the ordinal position of the micro-batch.

3.1.2 Throttling by KV Cache Utilization Rate (UT). At the beginning of each schedule, gLLM also collects the KV cache free rate ($KV_{free} \in [0, 1]$) to determine batched token count ($\#P$). This decision depends on two hyperparameters, minimum or maximum tokens count batched in prefill ($\#MinP/\#MaxP$). The batched token count is then computed as:

$$\#P = max(\#MaxP \times KV_{free}, \#MinP) \quad (2)$$

3.1.3 Threshold. Besides dynamically adjusting prefill tokens count, we also introduce a KV cache idle threshold ($KV_{thresh} \in [0, 1]$) to regulate scheduling decisions. When current KV cache idle rate is less than KV_{thresh}, the system automatically suspends prefill token processing to prevent KV cache overflow. This safeguard mechanism addresses the critical issues observed in unrestricted prefill operations: Premature preemption of ongoing decode requests causes costly recomputation time. By maintaining buffer headroom through the threshold mechanism, we ensure adequate resource allocation for active decode requests while maintaining stable system.

Combining aforementioned factors (*WT*, *UT* and *threshold*), we compute the batched token count as (when $KV_{free} \geq KV_{thresh}$):

$$\#P = max(min(\frac{\#WP}{\#T}, \#MaxP \times \frac{KV_{free} - KV_{thresh}}{1 - KV_{thresh}}), \#MinP) \quad (3)$$

3.1.4 Case Study. Figure 6 compares the prefill scheduling strategies of Sarathi-Serve and gLLM. In the first scenario, there are no tokens waiting to prefill at the initial state and $\#T$ is set to 3. Upon request arrival, Sarathi-Serve eagerly processes prefill tokens. This leads to no prefill tokens batched in the following batch (missing p2/4/5/6), resulting in large fluctuation in the computation. The significant pipeline bubbles also prevent Sarathi-Serve from processing the next batch of requests in a timely manner. Instead, gLLM evenly distributes new prefill tokens across the following batches, achieving better load balance.

For the second situation, there is a fixed amount of tokens waiting to prefill at the start. Sarathi-Serve schedules prefill tokens without considering KV cache utilization rates. This results in the remaining KV cache space being insufficient to support prefill computation after allocating the room to the previous batch. And, the following batch can only handle decode tokens (missing P2/3[1]) until some requests finish. In the contrast, gLLM dynamically adjusts prefill scheduling based on KV cache occupancy, ensuring more balanced computation. Moreover, gLLM can achieve more efficient KV cache turnover, enabling timely processing of decode requests and preventing their KV cache from occupying space for extended periods.

3.2 Decode Token Throttling

3.2.1 Throttling by Tokens Count Under Decode. The scheduling for decode Token Throttling is straightforward, since decode operations require multiple iterations (equal to the output length of sequence) while prefill operations typically complete in several iterations. The variation in decode requests is relatively small, as these requests either originate from completed prefill phase or have reached their termination condition. Therefore, our objective is to distribute the total decode tokens evenly across all available micro-batches. Given that the maximal number of active micro-batches equals to the pipeline depth ($\#PP_{depth}$), we can compute the batched decode token count ($\#D$) as:

$$\#D = \frac{\#RD}{\#PP_{depth}} \quad (4)$$

where $\#RD$ is total running tokens count under decode. If the remaining decode tokens are fewer than $\#D$, we schedule all of them; otherwise, we schedule exactly $\#D$ tokens.

3.2.2 Case Study. Figure 8 compares the decode scheduling strategies of Sarathi-Serve and gLLM. Sarathi-Serve fails to balance workloads evenly across micro-batches, leading to significant pipeline bubbles. For inter-stage dependency, the second/sixth micro-batch has to wait for the completion of the previous stage. For inter-batch dependency, the fifth/sixth micro-batch has to wait for the finish of the first/second micro-batch. These dependencies induce significant pipeline bubbles. In contrast, gLLM optimizes scheduling by dynamically adjusting token counts based on the total tokens under decode, ensuring a balanced workload distribution and reducing pipeline inefficiencies.

[1]P5/6 is missing in Sarathi-Serve because there are no waiting prefill tokens.

3.3 gLLM Runtime

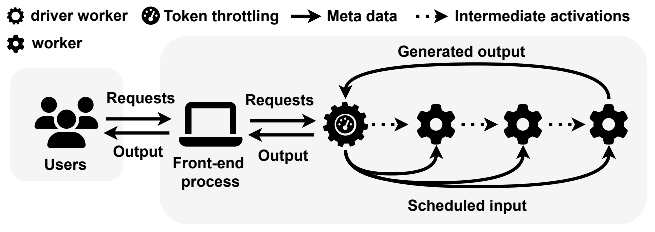


Figure 9: Architecture of gLLM runtime.

Compared to tensor parallelism, pipeline parallelism requires more sophisticated control flow to orchestrate micro-batches across different pipeline stages. To enable Token Throttling, we develop an asynchronous runtime system designed to manage this complexity efficiently. As illustrated in Figure 9, our runtime architecture follows the established paradigm of mainstream and recently proposed inference frameworks [11, 20, 58], while introducing key extensions to support fine-grained scheduling and dynamic execution.

To support concurrent execution of multiple stages, gLLM runtime adopts a multi-process architecture where each pipeline stage is assigned a dedicated worker process, along with a separate front-end process for user interaction. The workers are divided into two roles: a driver worker and ordinary workers. The driver worker oversees other ordinary workers, handling tasks such as receiving new requests from the front-end, scheduling micro-batches, broadcasting metadata for each schedule and streaming output back to the front-end. Meanwhile, all the workers focus on model execution, receiving activations from the previous stage, performing forward computations, and sending activations to the next stage. The driver worker is responsible for the KV cache management and all the workers share the page tables like vLLM.

The asynchronous of gLLM runtime is achieved through three coordinated design principles:

(1) Non-blocking pipeline operations. All workers processes employ non-blocking mechanisms for core operations including request reception, metadata exchange, and activation transmission, forming a continuous processing pipeline that eliminates idle waiting between computational stages.

(2) Decoupled frontend-backend processing. We design a dedicated frontend process that handles user-facing operations (request intake and response streaming), enabling full parallelism with backend model execution on worker processes. This separation allows continuous user interaction while maintaining uninterrupted model computation.

(3) Preemptive metadata scheduling. The runtime utilizes a dual-phase data transmission[2] strategy: (a) Driver workers broadcast metadata packets to all workers (b) Workers receive corresponding intermediate data streams. This decoupling enables workers to perform critical path preparation (input or attention metadata tensor creation) using early-arrived metadata, effectively overlapping data preparation latency with active computation cycles. The proactive scheduling mechanism ensures computation resources remain fully utilized throughout the execution timeline.

[2]In gLLM, metadata is transmitted via ZeroMQ, while intermediate activations are exchanged using NCCL.

3.4 Implementation

For implementation, we identify a critical design limitation in vLLM's pipeline parallelism architecture, where the transmission of intermediate activations is tightly coupled with input scheduling metadata. This integration introduces significant CPU overhead during input preparation for model forwading, accounting for approximately 17% of the total execution time and substantially diminishing the potential benefits of pipeline parallelism.

To overcome this limitation, we have implemented essential part of gLLM with 4K lines of Python code on top of CUDA ecosystem. The system features a RESTful API frontend and offers core OpenAI-compatible APIs. It incorporates vLLM's manually optimized CUDA kernels for key operations including activation functions, layer normalization, position encoding, and KV cache management, while also adopting flash attention [6, 7, 42]. We also integrates several recent advanced LLM optimizations, including iteration-level scheduling [56], PagedAttention [20], Sarathi-Serve [2], prefix caching [62] and CPP [37]. To validate its output quality, we evaluate gLLM on the MMLU-pro benchmark [53], with detailed results presented in Section 4.7. Token Throttling primarily involves lightweight additional system state collection and few mathematical computations, making its overhead moderate. Experimental results demonstrate an average overhead of just 0.045 ms per iteration, compared to each model forward pass's execution time of 20-800 ms.

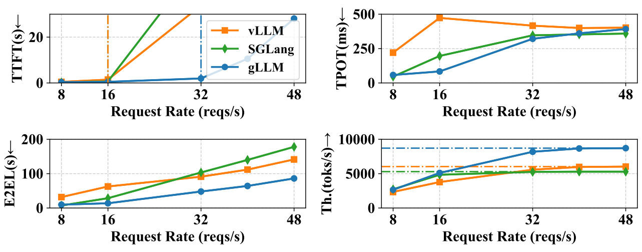
4 EVALUATION

4.1 Experimental Setup

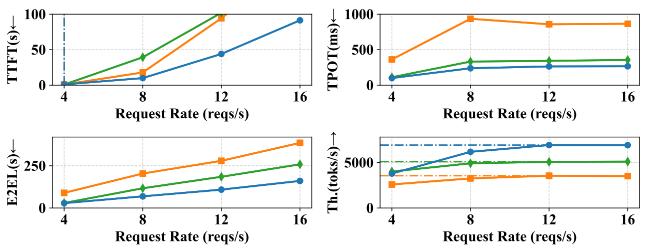
Models and Environment. We evaluate gLLM using the Qwen2.5 series [54] (14B and 32B parameter variants) and Llama-3.1-100B[3] [13] for their strong multi-task performance. All models utilize bfloat16 data type. Our experiments employ three node configurations: (1) 4× NVIDIA L20-48GB GPUs (2) 4× NVIDIA A100-40GB GPUs (3) 4× NVIDIA A800-80GB GPUs. All GPUs are connected via PCIe across three configurations. We primarily evaluate two scenarios of intra-node deployment and cross-node deployment. For intra-node, we use the L20 configuration. Cross-node evaluations leverage the A100 and A800 setup with simulated network conditions achieved by disabling both P2P communication (PCIe-based) and shared memory access (by setting environment variable *NCCL_SHM_DISABLE* = 1 and *NCCL_P2P_DISABLE* = 1). Such configuration forces all inter-GPU communication to go through the network stack. Testing results show that the simulated network communication bandwidth achieves 73.28 Gbps, whereas the PCIe-based communication bandwidth attains 20.79 GB/s.

Workloads. We synthesize workloads based on ShareGPT [1] and Azure [35], which both comes from real LLM services. The ShareGPT dataset is a collection of user-shared conversations with ChatGPT. The Azure dataset is from production traces taken from Azure LLM inference services, including the arrival time, input size and output size. Figure 11 displays the distribution of input and output lengths across the sampled datasets, revealing that

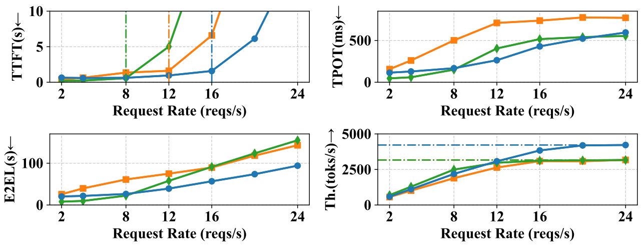
[3]This model is downscaled from Llama3.1-405B to fit in GPU memory.



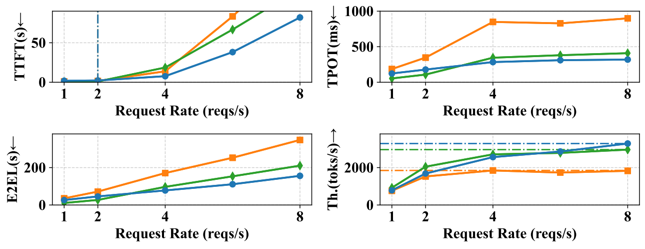
(a) Qwen2.5-14B, ShareGPT



(b) Qwen2.5-14B, Azure



(c) Qwen2.5-32B, ShareGPT



(d) Qwen2.5-32B, Azure

Figure 10: The latency and throughput comparison between vLLM, SGLang and gLLM (1 node with 4×L20).

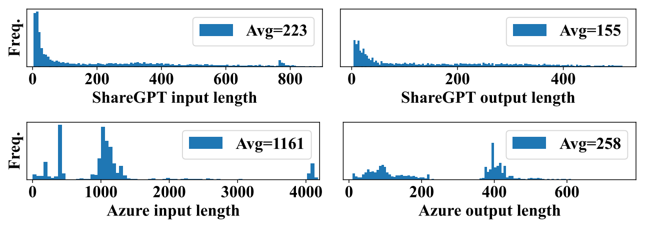


Figure 11: Distribution of input and output length of the sampled dataset.

the Azure dataset has a notably longer average input (5.21×) and output (1.66×) length compared to ShareGPT. We mimic the cloud service scenario and generate request arrival times using Poisson distribution with different request rates.

Schemes. To evaluate the effectiveness of our proposed design, we compare gLLM against the following systems:

- **vLLM** [20]. We leverage vLLM (v0.8.1) as the fastest inference engines for pipeline parallelism. The framework offers two backend engine versions, V0 and V1. While V1 demonstrates superior speed over V0, our testing revealed it to be less stable. As a result, we employ V1 as the default setting but fall back to V0 when encountering engine crashes.
- **SGLang** [62]. We leverage SGLang (v0.4.3.post2) as the most efficient inference engine for tensor parallelism implementations. While SGLang has lower CPU overhead than vLLM, it currently lacks pipeline parallelism support.
- **gLLM**. Proposed global balanced pipeline parallelism systems with Token Throttling.
- **gLLM w/o WT**. gLLM without WT (section 3.1.1).
- **gLLM w/o UT**. gLLM without UT (section 3.1.2).
- **gLLM w/ CK**. gLLM with the scheduling policy used in Sarathi-Serve.

vLLM and SGLang both follow Sarathi-Serve's scheduling strategy (token budget is set to 2048). To ensure fair comparisons, we disable KV cache reuse across requests and CUDA graph for each system. While vLLM and gLLM employ pipeline parallelism, SGLang utilizes tensor parallelism. The GPU memory utilization of each system is set to the maximum without encountering out of memory error. In the main experiment, we benchmark gLLM against vLLM and SGLang, while the ablation study compares gLLM with its variations. For gLLM, we set the hyperparameters as follows: $\#T = 8$, $\#MaxP = 2048$, $\#MinP = 32$ and $KV_{thresh} = 0.05$. We examine the impacts of these parameter settings in the sensitivity study.

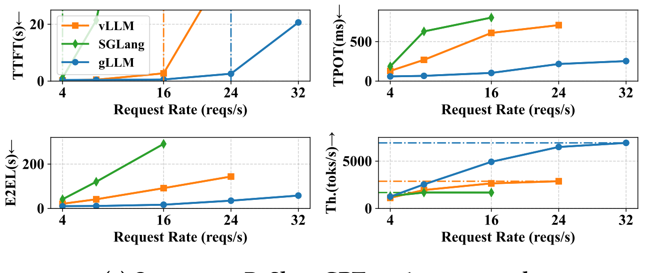
Metrics. We use the following metrics to measure the performance of the systems:

- **Time to first token (TTFT)**: average time taken from when a user sends a prompt to the LLM until the first token of the response is generated.
- **Time per output token (TPOT)**: average time required to generate each subsequent token after the first one.
- **End to end latency (E2EL)**: average total time from prompt submission to the completion of the full response.
- **Throughput**: average input and output tokens processing throughput.
- **SLO Attainment**: the SLO fulfillment rate under the given TTFT and TPOT constraints.

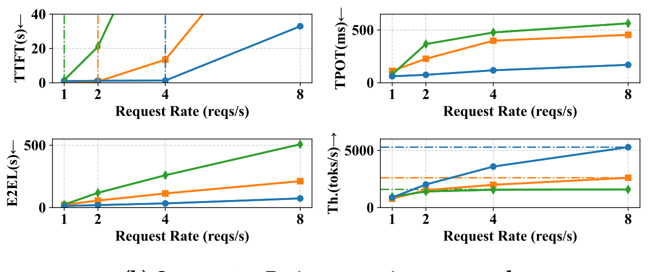
4.2 Latency and Throughput

To compare the performance of vLLM, SGLang and gLLM, we conduct experiments based on ShareGPT and Azure dataset using models from 14B to 100B with the results[4] illustrated on Figure

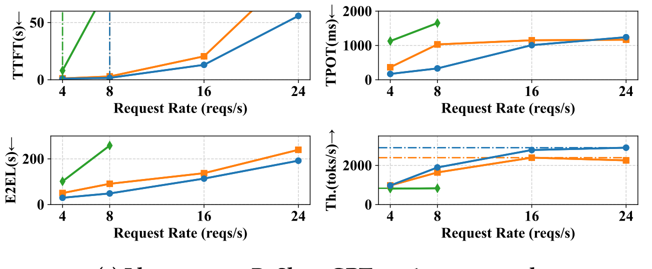
[4]↑ indicates higher values are better, while ↓ means lower values are preferable.



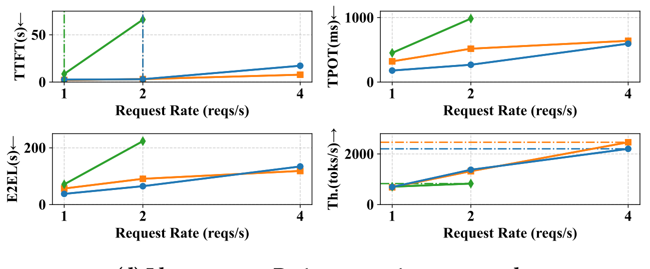
(a) Qwen2.5-32B, ShareGPT, 1×A100 per node



(b) Qwen2.5-32B, Azure, 1×A100 per node

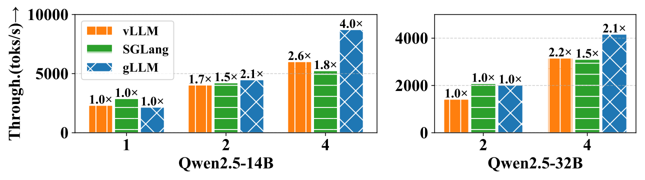


(c) Llama3.1-100B, ShareGPT, 1×A800 per node

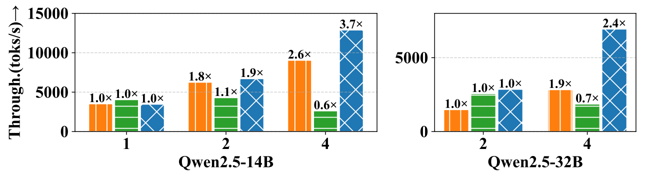


(d) Llama3.1-100B, Azure, 1×A800 per node

Figure 12: The latency and throughput comparison between vLLM, SGLang and gLLM (4 nodes).



(a) Intra-node scalability with L20



(b) Cross-node scalability with 1×A100 per node

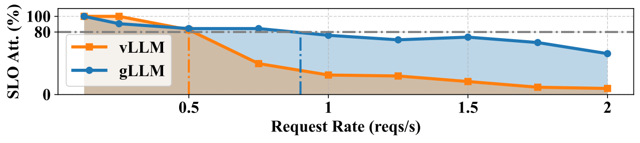
Figure 13: Variation of maximum throughput as the number of GPUs/nodes (horizontal axis) increases. The numbers (×) on the bar illustrate the multiples compared to single/two-card/node(s) performance.

10 (intra-node) and Figure 12 (cross-node). We can summarize a few key points from the figure: As request rates increase, (1) latency rises while throughput gradually plateaus. This plateau represents the system's maximum processing capacity. (2) TTFT will rise significantly at some point due to requests queuing. At the most situations, gLLM reaches its turning point at 2-6× higher request rates compared to other systems. (3) E2EL shows an approximately linear increase trend. Mostly, gLLM achieves 0.14-0.92× lower slope compared to the other two systems. (4) In most cases, benefiting from the more efficient scheduling policy and runtime, gLLM significantly outperforms vLLM on both latency and throughput across the tested scenarios (processing capacity improves 0.29-1.50×). While gLLM performs slightly worse than vLLM when serving Llama3.1-100B on the Azure dataset at a request rate of 4, this is due to the abundance of tokens for prefilling and sufficient memory to hold the KV cache, which minimizes variations of scheduled tokens in Sarathi-Serve. However, such conditions are uncommon in real-world serving environments. (5) Tensor parallelism is well suited for scenarios with low request rates and high bandwidth connection. For intra-node experiment, SGLang achieves lower latency compared to pipeline parallelism systems under low request rates. But as the request rate increases, the advantage of SGLang diminishes and even less than gLLM. For inter-node experiment, the performance of gLLM is significantly better than SGLang due to its high communication overhead. (processing capacity improves 0.11-3.98×) (6) Compared to Azure, gLLM is better at handling ShareGPT. The reason is that gLLM can balance prefill tokens across different requests, whereas in Azure, sequences with longer inputs weaken inter-request parallelism.

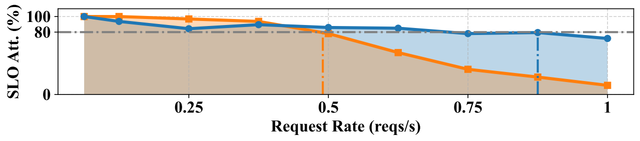
4.3 Scalability Study

To evaluate the scalability of gLLM, we conduct maximum throughput tests against vLLM and SGLang by incrementally increasing request rates until system throughput stabilizes. Our results shown in Figure 13 reveal distinct scaling patterns across systems. While gLLM demonstrates marginally lower throughput than vLLM and SGLang when serving 14B model on single GPU configurations (attributed to incomplete optimizations), it achieves near-linear scaling efficiency as GPU counts increase. In contrast, vLLM exhibits sub-linear scaling with the 14B model but maintains linear scaling with the 32B variant. SGLang demonstrates sub-linear scaling within a single node, but experiences performance degradation as GPUs count increase in cross-node deployments due to high inter-GPU communication overhead. Notably, gLLM's performance advantage becomes progressively more pronounced with more GPUs, suggesting superior architectural scalability.

4.4 SLO Attainment



(a) ShareGPT with SLO TTFT:3000ms and TPOT:150ms



(b) Azure with SLO TTFT:4000ms and TPOT:200ms

Figure 14: SLO attainment in cross-node deployments of Llama3.1-100B with A800.

To evaluate SLO attainment, we compare gLLM and vLLM in cross-node deployments of Llama3.1-100B. As shown in Figure 14, gLLM achieves 64% higher SLO attainment coverage than vLLM across various request rates. Under 80% SLO attainment, gLLM can sustain 79% higher request rate compared to vLLM. While at low request rate, gLLM exhibits slightly lower SLO attainment due to a marginal increase in TTFT caused by Token Throttling, which occasionally exceeds the time constraint. To address this, we can fine-tune the hyperparameter $\#T$ to balance TTFT and TPOT performance.

4.5 Ablation Study

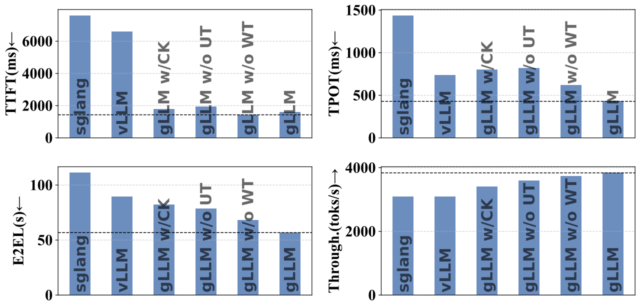


Figure 15: An ablation study on the design choices of gLLM. The dashed line marks the optimal value under each metric.

To evaluate the effectiveness of our design, we conduct ablation experiments on gLLM, with the results shown in Figure 15. Notably, gLLM w/o WT achieves 10% lower TTFT compared to gLLM, as WT's balanced scheduling approach slightly compromises prefill speed. However, this comes at the cost of 44% higher TPOT and 20% longer E2EL. The absence of UT leads to even more significant performance degradation, increasing TTFT by 22%, TPOT by 91%, and E2EL by 38%, demonstrating UT's crucial role in balancing computation. Meanwhile, even with Sarathi-Serve's basic scheduling strategy, gLLM w/CK achieves 10% higher throughput and 8% lower E2EL compared to vLLM. This demonstrates that gLLM runtime is more efficient than that of vLLM.

4.6 Sensitivity Study

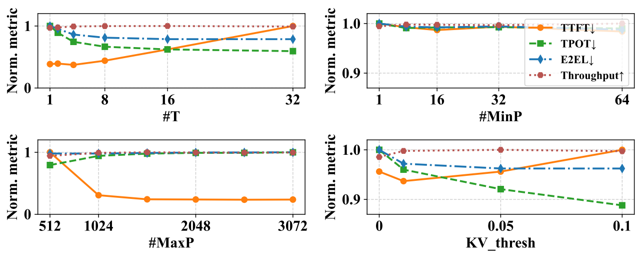


Figure 16: Normalized metric under different setting of hyperparameters #T, #MaxP, #MinP and KV$_{\text{thresh}}$.

To examine the effect of the hyperparameters used in gLLM, we conduct a sensitivity study and the normalized results are shown in Figure 16.

4.6.1 *Impact of $\#T$.*

The parameter $\#T$, representing the number of iterations to process all pending prefill tokens. As $\#T$ increases, the number of prefill tokens per micro-batch decreases, which generally leads to longer TTFT due to reduced prefill rate. However, when $\#T$ grows from 1 to 4, the TTFT remains stable because gains in computational parallelism, such as better GPU utilization and reduced idle time, offset the smaller batch sizes. Concurrently, TPOT improves as smaller micro-batches enable faster decoding rates. Throughput increases and E2EL decreases with larger $\#T$, proving Token Throttling improve overall processing efficiency through more evenly distributed computations.

4.6.2 *Impact of $\#MaxP$.*

The hyperparameter $\#MaxP$ governs the maximum number of batched prefill tokens processed per scheduling iteration. Increasing $\#MaxP$ creates a dual effect: Higher prefill rates accelerate initial token generation (TTFT), while expanded batch sizes prolong per-token processing latency (TPOT). However, excessively conservative $\#MaxP$ settings (e.g., 512) degrade system throughput due to suboptimal prefill rate, which constrains the system's capacity to handle concurrent requests during decode phases.

4.6.3 *Impact of KV_{thresh}.*

KV_{thresh} defines the idle rate threshold for KV cache utilization. Setting KV_{thresh} to zero introduces inefficiencies: TTFT, TPOT and E2EL exhibit slight increases, while throughput declines. This occurs because a zero threshold pushes the system toward KV cache capacity limits more frequently, forcing preemption of ongoing requests to accommodate remaining decode requests. Such preemption waste computational resources and degrade overall performance.

4.6.4 *Impact of $\#MinP$.*

The minimum batched prefill token count ($\#MinP$) exhibits limited impact on overall system performance across most configurations (within 2% performance fluctuations).

4.7 Functionality Study

Table 1: Comparison of the number of lines of code and the score of MMLU-Pro [53] (evaulated on Qwen2.5-32B-Instruct) between gLLM, SGLang and vLLM.

Framework	gLLM	SGLang	vLLM
Lines of code	3874	65097	226874
MMLU-pro↑	68.86	68.85	69.17

Table 1 presents a comparison in lines of code and output quality across different frameworks. Notably, gLLM achieves comparable output quality to vLLM and SGLang while maintaining a superior inference speed.

5 RELATED WORK

Scheduling in LLMs. Traditional DNN inference frameworks primarily employ batch-level scheduling [33], which struggles to handle the variable sequence lengths inherent in LLMs. To address this limitation, Orca [56] introduces iteration-level scheduling, enabling dynamic request admission and early exit before model execution. However, this approach faces challenges when processing lengthy prefill requests, as they can delay subsequent decode requests. To mitigate this imbalance, recent work proposes Sarathi-Serve [2], which runs prefill and decode operations together by splitting long sequences into smaller chunks. Despite these advancements, computational imbalance persists across each batch, significantly degrading the efficiency of pipeline parallelism.

LLM serving systems. To efficiently serve LLMs, researchers have developed various systems. Among them, Orca [56], introduces a distributed serving system with iteration-level scheduling to improve throughput. For memory optimization, vLLM [20] employs paged attention, imitating virtual memory mechanisms to reduce fragmentation, while SGLang [62] leverages radix attention to eliminate redundant KV cache computations across requests. To address the divergent computational demands of the prefill and decode stages, Splitwise [35] and DistServe [63] adopt a disaggregated architecture, allocating specialized hardware configurations for each phase. TD-Pipe [58] proposes temporally-disaggregated architecture for pipeline parallelism which attempts to solve the unbalanced computation in the offline scenario. In contrast, gLLM focuses on alleviating pipeline bubbles in online serving by applying token throttling in chunked prefill method. Additionally, frameworks like FlexGen [43] and InfiniGen [21] tackle GPU memory constraints through adaptive techniques such as memory compression and intelligent offloading, ensuring scalable performance under resource limitations. Nevertheless, these systems fail to achieve balanced schedule between each batch, leading to significant performance bottleneck.

Model parallelism for LLM training and serving. Model parallelism has become essential for distributed training and serving of LLMs as their sizes increase. Tensor parallelism, which demands frequent communication between devices, is primarily employed in environments with high-bandwidth interconnects. Recent advancements [5, 12, 17, 52] address communication idling by strategically overlapping communication operations with computation. For pipeline parallelism, researches focus on solving unbalanced memory consumption [19, 27, 47], pipeline bubbles [27, 36], communication optimization [24] and activation checkpointing [26, 47]. Hybrid strategies combining tensor pipellelism and pipeline parallelism leverage automated search algorithms [50, 59, 61] or utilize the heterogeneous characteristics [18, 39, 50, 59], while frameworks like Megatron-LM [32] demonstrate empirically validated configurations for massive-scale deployment. However, those methods focus on optimization during training. In LLM serving, specialized optimizations target hardware heterogeneity through GPU-aware allocation [30, 45] and reduce pipeline bubbles using chunked prefill mechanisms [2]. Nevertheless, pipeline bubbles are not efficiently solved by chunked prefill and become a bottleneck for efficient deploying.

6 CONCLUSION AND DISCUSSION

This paper presents gLLM, a globally balanced pipeline parallelism system incorporating with Token Throttling mechanism for distributed LLM serving. Token Throttling dynamically adjusts the token counts for prefill and decode stages, thereby achieving balanced computation across micro-batches and effectively mitigating pipeline bubbles. To enable Token Throttling, we design an asynchronous runtime tailored to the characteristics of pipeline parallelism. Experiments on representative LLMs demonstrate that gLLM delivers 11% to 398% higher maximum throughput over state-of-the-art pipeline and tensor parallelism systems, while simultaneously maintaining lower latency.

Currently, gLLM assumes that computation time is proportional to the number of tokens in a batch. However, the self-attention operation has a quadratic dependency on sequence length, introducing an additional factor that influences computation time. Moving forward, to better balance the computational load across micro-batches, we should incorporate the context length of each sequence to enable more accurate estimation of forward pass time. For mixture-of-expert (MoE) models, variability in expert activation introduces additional imbalance. In this regard, future work will explore expert-aware load balancing strategies to extend gLLM 's applicability to a broader range of LLM architectures.

Acknowledgments

We sincerely thank the anonymous reviewers for their valuable comments and suggestions, which have greatly helped improve this work. This research is supported by the National Key R&D Program of China (Grant No. 2023YFB3002202), the NSFC grants (62472462, 62461146204), and is sponsored by CCF-Tencent Rhino-Bird Open Research Fund (CCF-Tencent RAGR20240102). Xianwei Zhang and Zhiguang Chen are the corresponding authors.

References

[1] ShareGPT Team. 2023. ShareGPT. https://sharegpt.com/.

[2] 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 2024, Santa Clara, CA, USA, July 10-12, 2024*, Ada Gavrilovska and Douglas B. Terry (Eds.). USENIX Association, 117–134. https://www.usenix.org/conference/osdi24/presentation/agrawal

[3] Ebtesam Almazrouei, Hamza Alobeidli, Abdulaziz Alshamsi, Alessandro Cappelli, Ruxandra Cojocaru, Mérouane Debbah, Étienne Goffinet, Daniel Hesslow, Julien

Launay, Quentin Malartic, Daniele Mazzotta, Badreddine Noune, Baptiste Pannier, and Guilherme Penedo. 2023. The Falcon Series of Open Language Models. *CoRR* abs/2311.16867 (2023). doi:10.48550/ARXIV.2311.16867 arXiv:2311.16867

[4] Alexander Borzunov, Max Ryabinin, Artem Chumachenko, Dmitry Baranchuk, Tim Dettmers, Younes Belkada, Pavel Samygin, and Colin A. Raffel. 2023. Distributed Inference and Fine-tuning of Large Language Models Over The Internet. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine (Eds.). http://papers.nips.cc/paper_files/paper/2023/hash/28bf1419b9a1f908c15f6195f58cb865-Abstract-Conference.html

[5] Chang Chen, Xiuhong Li, Qianchao Zhu, Jiangfei Duan, Peng Sun, Xingcheng Zhang, and Chao Yang. 2024. Centauri: Enabling Efficient Scheduling for Communication-Computation Overlap in Large Model Training via Communication Partitioning. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, ASPLOS 2024, La Jolla, CA, USA, 27 April 2024- 1 May 2024*, Rajiv Gupta, Nael B. Abu-Ghazaleh, Madan Musuvathi, and Dan Tsafrir (Eds.). ACM, 178–191. doi:10.1145/3620666.3651379

[6] Tri Dao. 2024. FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net. https://openreview.net/forum?id=mZn2Xyh9Ec

[7] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh (Eds.). http://papers.nips.cc/paper_files/paper/2022/hash/67d57c32e20fd0a7a302cb81d36e40d5-Abstract-Conference.html

[8] DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jiawei Wang, Jin Chen, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, Junxiao Song, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Litong Wang, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qiancheng Wang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runxin Xu, Ruoyu Zhang, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Shuting Pan, T. Wang, Tao Yun, Tian Pei, Tianyu Sun, W. L. Xiao, and Wangding Zeng. 2024. DeepSeek-V3 Technical Report. *CoRR* abs/2412.19437 (2024). doi:10.48550/ARXIV.2412.19437 arXiv:2412.19437

[9] Shihan Deng, Weikai Xu, Hongda Sun, Wei Liu, Tao Tan, Jianfeng Liu, Ang Li, Jian Luan, Bin Wang, Rui Yan, and Shuo Shang. 2024. Mobile-Bench: An Evaluation Benchmark for LLM-based Mobile Agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, 8813–8831. doi:10.18653/V1/2024.ACL-LONG.478

[10] Aniket Didolkar, Anirudh Goyal, Nan Rosemary Ke, Siyuan Guo, Michal Valko, Timothy P. Lillicrap, Danilo Jimenez Rezende, Yoshua Bengio, Michael C. Mozer, and Sanjeev Arora. 2024. Metacognitive Capabilities of LLMs: An Exploration in Mathematical Problem Solving. In *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang (Eds.). http://papers.nips.cc/paper_files/paper/2024/hash/2318d75a06437eaa257737a5cf3ab83c-Abstract-Conference.html

[11] Jiangsu Du, Ziming Liu, Jiarui Fang, Shenggui Li, Yongbin Li, Yutong Lu, and Yang You. 2022. EnergonAI: An Inference System for 10-100 Billion Parameter Transformer Models. *CoRR* abs/2209.02341 (2022). doi:10.48550/ARXIV.2209.02341 arXiv:2209.02341

[12] Jiangsu Du, Jinhui Wei, Jiazhi Jiang, Shenggan Cheng, Dan Huang, Zhiguang Chen, and Yutong Lu. 2024. Liger: Interleaving Intra- and Inter-Operator Parallelism for Distributed Large Model Inference. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, PPoPP 2024, Edinburgh, United Kingdom, March 2-6, 2024*, Michel Steuwer, I-Ting Angelina Lee, and Milind Chabbi (Eds.). ACM, 42–54. doi:10.1145/3627535.3638466

[13] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, Arun Rao, Aston Zhang, Aurélien Rodriguez, Austen Gregerson, Ava Spataru, Baptiste Rozière, Bethany Biron, Binh Tang, Bobbie Chern, Charlotte Caucheteux, Chaya Nayak, Chloe Bi, Chris Marra, Chris McConnell, Christian Keller, Christophe Touret, Chunyang Wu, Corinne Wong, Cristian Canton Ferrer, Cyrus Nikolaidis, Damien Allonsius, Daniel Song, Danielle Pintz, Danny Livshits, David Esiobu, Dhruv Choudhary, Dhruv Mahajan, Diego Garcia-Olano, Diego Perino, Dieuwke Hupkes, Egor Lakomkin, Ehab AlBadawy, Elina Lobanova, Emily Dinan, Eric Michael Smith, Filip Radenovic, Frank Zhang, Gabriel Synnaeve, Gabrielle Lee, Georgia Lewis Anderson, Graeme Nail, Grégoire Mialon, Guan Pang, Guillem Cucurell, Hailey Nguyen, Hannah Korevaar, Hu Xu, Hugo Touvron, Iliyan Zarov, Imanol Arrieta Ibarra, Isabel M. Kloumann, Ishan Misra, Ivan Evtimov, Jade Copet, Jaewon Lee, Jan Geffert, Jana Vranes, Jason Park, Jay Mahadeokar, Jeet Shah, Jelmer van der Linde, Jennifer Billock, Jenny Hong, Jenya Lee, Jeremy Fu, Jianfeng Chi, Jianyu Huang, Jiawen Liu, Jie Wang, Jiecao Yu, Joanna Bitton, Joe Spisak, Jongsoo Park, Joseph Rocca, Joshua Johnstun, Joshua Saxe, Junteng Jia, Kalyan Vasuden Alwala, Kartikeya Upasani, Kate Plawiak, Ke Li, Kenneth Heafield, Kevin Stone, and et al. 2024. The Llama 3 Herd of Models. *CoRR* abs/2407.21783 (2024). doi:10.48550/ARXIV.2407.21783 arXiv:2407.21783

[14] William Fedus, Barret Zoph, and Noam Shazeer. 2022. Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity. *J. Mach. Learn. Res.* 23 (2022), 120:1–120:39. https://jmlr.org/papers/v23/21-0998.html

[15] Aaron Harlap, Deepak Narayanan, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, and Phillip B. Gibbons. 2018. PipeDream: Fast and Efficient Pipeline Parallel DNN Training. *CoRR* abs/1806.03377 (2018). arXiv:1806.03377 http://arxiv.org/abs/1806.03377

[16] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Xu Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. 2019. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 103–112. https://proceedings.neurips.cc/paper/2019/hash/093f65e080a295f8076b1c5722a46aa2-Abstract.html

[17] Abhinav Jangda, Jun Huang, Guodong Liu, Amir Hossein Nodehi Sabet, Saeed Maleki, Youshan Miao, Madanlal Musuvathi, Todd Mytkowicz, and Olli Saarikivi. 2022. Breaking the computation and communication abstraction barrier in distributed machine learning workloads. In *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, Babak Falsafi, Michael Ferdman, Shan Lu, and Thomas F. Wenisch (Eds.). ACM, 402–416. doi:10.1145/3503222.3507778

[18] Xianyan Jia, Le Jiang, Ang Wang, Wencong Xiao, Ziji Shi, Jie Zhang, Xinyuan Li, Langshi Chen, Yong Li, Zhen Zheng, Xiaoyong Liu, and Wei Lin. 2022. Whale: Efficient Giant Model Training over Heterogeneous GPUs. In *Proceedings of the 2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022*, Jiri Schindler and Noa Zilberman (Eds.). USENIX Association, 673–688. https://www.usenix.org/conference/atc22/presentation/jia-xianyan

[19] Taebum Kim, Hyoungjoo Kim, Gyeong-In Yu, and Byung-Gon Chun. 2023. BPipe: Memory-Balanced Pipeline Parallelism for Training Large Language Models. In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA (Proceedings of Machine Learning Research, Vol. 202)*, Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (Eds.). PMLR, 16639–16653. https://proceedings.mlr.press/v202/kim23l.html

[20] 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, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, Jason Flinn, Margo I. Seltzer, Peter Druschel, Antoine Kaufmann, and Jonathan Mace (Eds.). ACM, 611–626. doi:10.1145/3600006.3613165

[21] Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. 2024. InfiniGen: Efficient Generative Inference of Large Language Models with Dynamic KV Cache Management. In *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024*, Ada Gavrilovska and Douglas B. Terry (Eds.). USENIX Association, 155–172. https://www.usenix.org/conference/osdi24/presentation/lee

[22] Yuanyuan Lei and Ruihong Huang. 2024. Boosting Logical Fallacy Reasoning in LLMs via Logical Structure Tree. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, EMNLP 2024, Miami, FL, USA, November 12-16, 2024*, Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (Eds.). Association for Computational Linguistics, 13157–13173. https://aclanthology.org/2024.emnlp-main.730

[23] Qintong Li, Leyang Cui, Xueliang Zhao, Lingpeng Kong, and Wei Bi. 2024. GSM-Plus: A Comprehensive Benchmark for Evaluating the Robustness of LLMs as Mathematical Problem Solvers. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL*

2024, Bangkok, Thailand, August 11-16, 2024, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, 2961–2984. doi:10.18653/V1/2024.ACL-LONG.163

[24] Junfeng Lin, Ziming Liu, Yang You, Jun Wang, Weihao Zhang, and Rong Zhao. 2025. WeiPipe: Weight Pipeline Parallelism for Communication-Effective Long-Context Large Model Training. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, PPoPP 2025, Las Vegas, NV, USA, March 1-5, 2025*. ACM, 225–238. doi:10.1145/3710848.3710869

[25] Hongwei Liu, Zilong Zheng, Yuxuan Qiao, Haodong Duan, Zhiwei Fei, Fengzhe Zhou, Wenwei Zhang, Songyang Zhang, Dahua Lin, and Kai Chen. 2024. MathBench: Evaluating the Theory and Application Proficiency of LLMs with a Hierarchical Mathematics Benchmark. In *Findings of the Association for Computational Linguistics, ACL 2024, Bangkok, Thailand and virtual meeting, August 11-16, 2024*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, 6884–6915. doi:10.18653/V1/2024.FINDINGS-ACL.411

[26] Weijian Liu, Mingzhen Li, Guangming Tan, and Weile Jia. 2025. Mario: Near Zero-cost Activation Checkpointing in Pipeline Parallelism. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, PPoPP 2025, Las Vegas, NV, USA, March 1-5, 2025*. ACM, 197–211. doi:10.1145/3710848.3710878

[27] Ziming Liu, Shenggan Cheng, Haotian Zhou, and Yang You. 2023. Hanayo: Harnessing Wave-like Pipeline Parallelism for Enhanced Large Model Training Efficiency. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2023, Denver, CO, USA, November 12-17, 2023*, Dorian Arnold, Rosa M. Badia, and Kathryn M. Mohror (Eds.). ACM, 56:1–56:13. doi:10.1145/3581784.3607073

[28] Zimu Lu, Aojun Zhou, Houxing Ren, Ke Wang, Weikang Shi, Junting Pan, Mingjie Zhan, and Hongsheng Li. 2024. MathGenie: Generating Synthetic Data with Question Back-translation for Enhancing Mathematical Reasoning of LLMs. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, 2732–2747. doi:10.18653/V1/2024.ACL-LONG.151

[29] Andreas Martin, Charuta Pande, Hans Friedrich Witschel, and Judith Mathez. 2024. ChEdBot: Designing a Domain-Specific Conversational Agent in a Simulational Learning Environment Using LLMs. In *Proceedings of the AAAI 2024 Spring Symposium Series, Stanford, CA, USA, March 25-27, 2024*, Ron P. A. Petrick and Christopher W. Geib (Eds.). AAAI Press, 180–187. doi:10.1609/AAAISS.V3I1.31198

[30] Yixuan Mei, Yonghao Zhuang, Xupeng Miao, Juncheng Yang, Zhihao Jia, and Rashmi Vinayak. 2024. Helix: Distributed Serving of Large Language Models via Max-Flow on Heterogeneous GPUs. *CoRR* abs/2406.01566 (2024). doi:10.48550/ARXIV.2406.01566 arXiv:2406.01566

[31] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. 2021. Memory-Efficient Pipeline-Parallel DNN Training. In *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event (Proceedings of Machine Learning Research, Vol. 139)*, Marina Meila and Tong Zhang (Eds.). PMLR, 7937–7947. http://proceedings.mlr.press/v139/narayanan21a.html

[32] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. 2021. Efficient large-scale language model training on GPU clusters using megatron-LM. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2021, St. Louis, Missouri, USA, November 14-19, 2021*, Bronis R. de Supinski, Mary W. Hall, and Todd Gamblin (Eds.). ACM, 58. doi:10.1145/3458817.3476209

[33] NVIDIA. 2023. Faster Transformer. https://github.com/NVIDIA/FasterTransformer.

[34] OpenAI. 2023. GPT-4 Technical Report. *CoRR* abs/2303.08774 (2023). doi:10.48550/ARXIV.2303.08774 arXiv:2303.08774

[35] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. 2024. Splitwise: Efficient Generative LLM Inference Using Phase Splitting. In *51st ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2024, Buenos Aires, Argentina, June 29 - July 3, 2024*. IEEE, 118–132. doi:10.1109/ISCA59077.2024.00019

[36] Penghui Qi, Xinyi Wan, Guangxing Huang, and Min Lin. 2024. Zero Bubble (Almost) Pipeline Parallelism. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net. https://openreview.net/forum?id=tuzTN0eIO5

[37] Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. 2025. Mooncake: Trading More Storage for Less Computation - A KVCache-centric Architecture for Serving LLM Chatbot. In *23rd USENIX Conference on File and Storage Technologies, FAST 2025, Santa Clara, CA, February 25-27, 2025*, Haryadi S. Gunawi and Vasily Tarasov (Eds.). USENIX Association, 155–170. https://www.usenix.org/conference/fast25/presentation/qin

[38] Machel Reid, Nikolay Savinov, Denis Teplyashin, Dmitry Lepikhin, Timothy P. Lillicrap, Jean-Baptiste Alayrac, Radu Soricut, Angeliki Lazaridou, Orhan Firat, Julian Schrittwieser, Ioannis Antonoglou, Rohan Anil, Sebastian Borgeaud, Andrew M. Dai, Katie Millican, Ethan Dyer, Mia Glaese, Thibault Sottiaux, Benjamin Lee, Fabio Viola, Malcolm Reynolds, Yuanzhong Xu, James Molloy, Jilin Chen, Michael Isard, Paul Barham, Tom Hennigan, Ross McIlroy, Melvin Johnson, Johan Schalkwyk, Eli Collins, Eliza Rutherford, Erica Moreira, Kareem Ayoub, Megha Goel, Clemens Meyer, Gregory Thornton, Zhen Yang, Henryk Michalewski, Zaheer Abbas, Nathan Schucher, Ankesh Anand, Richard Ives, James Keeling, Karel Lenc, Salem Haykal, Siamak Shakeri, Pranav Shyam, Aakanksha Chowdhery, Roman Ring, Stephen Spencer, Eren Sezener, and et al. 2024. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *CoRR* abs/2403.05530 (2024). doi:10.48550/ARXIV.2403.05530 arXiv:2403.05530

[39] Max Ryabinin, Tim Dettmers, Michael Diskin, and Alexander Borzunov. 2023. SWARM Parallelism: Training Large Models Can Be Surprisingly Communication-Efficient. In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA (Proceedings of Machine Learning Research, Vol. 202)*, Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (Eds.). PMLR, 29416–29440. https://proceedings.mlr.press/v202/ryabinin23a.html

[40] Raphael Schumann, Wanrong Zhu, Weixi Feng, Tsu-Jui Fu, Stefan Riezler, and William Yang Wang. 2024. VELMA: Verbalization Embodiment of LLM Agents for Vision and Language Navigation in Street View. In *Thirty-Eighth AAAI Conference on Artificial Intelligence, AAAI 2024, Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence, IAAI 2024, Fourteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2014, February 20-27, 2024, Vancouver, Canada*, Michael J. Wooldridge, Jennifer G. Dy, and Sriraam Natarajan (Eds.). AAAI Press, 18924–18933. doi:10.1609/AAAI.V38I17.29858

[41] Amrith Setlur, Saurabh Garg, Xinyang Geng, Naman Garg, Virginia Smith, and Aviral Kumar. 2024. RL on Incorrect Synthetic Data Scales the Efficiency of LLM Math Reasoning by Eight-Fold. In *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang (Eds.). http://papers.nips.cc/paper_files/paper/2024/hash/4b77d5b896c321a29277524a98a50215-Abstract-Conference.html

[42] Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. 2024. FlashAttention-3: Fast and Accurate Attention with Asynchrony and Low-precision. In *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang (Eds.). http://papers.nips.cc/paper_files/paper/2024/hash/7ede97c3e082c6df10a8d6103a2eebd2-Abstract-Conference.html

[43] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. 2023. FlexGen: High-Throughput Generative Inference of Large Language Models with a Single GPU. In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA (Proceedings of Machine Learning Research, Vol. 202)*, Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (Eds.). PMLR, 31094–31116. https://proceedings.mlr.press/v202/sheng23a.html

[44] Haochen Shi, Zhiyuan Sun, Xingdi Yuan, Marc-Alexandre Côté, and Bang Liu. 2024. OPEx: A Component-Wise Analysis of LLM-Centric Agents in Embodied Instruction Following. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, 622–636. doi:10.18653/V1/2024.ACL-LONG.37

[45] Jovan Stojkovic, Chaojie Zhang, Íñigo Goiri, Josep Torrellas, and Esha Choukse. 2024. DynamoLLM: Designing LLM Inference Clusters for Performance and Energy Efficiency. *CoRR* abs/2408.00741 (2024). doi:10.48550/ARXIV.2408.00741 arXiv:2408.00741

[46] Hongda Sun, Weikai Xu, Wei Liu, Jian Luan, Bin Wang, Shuo Shang, Ji-Rong Wen, and Rui Yan. 2024. DetermLR: Augmenting LLM-based Logical Reasoning from Indeterminacy to Determinacy. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, 9828–9862. doi:10.18653/V1/2024.ACL-LONG.531

[47] Zhenbo Sun, Huanqi Cao, Yuanwei Wang, Guanyu Feng, Shengqi Chen, Haojie Wang, and Wenguang Chen. 2024. AdaPipe: Optimizing Pipeline Parallelism with Adaptive Recomputation and Partitioning. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, ASPLOS 2024, La Jolla, CA, USA, 27 April 2024-1 May 2024*, Rajiv Gupta, Nael B. Abu-Ghazaleh, Madan Musuvathi, and Dan Tsafrir (Eds.). ACM, 86–100. doi:10.1145/3620666.3651359

[48] Armin Toroghi, Willis Guo, Ali Pesaranghader, and Scott Sanner. 2024. Verifiable, Debuggable, and Repairable Commonsense Logical Reasoning via LLM-based Theory Resolution. In *Proceedings of the 2024 Conference on Empirical Methods*

in Natural Language Processing, EMNLP 2024, Miami, FL, USA, November 12-16, 2024, Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (Eds.). Association for Computational Linguistics, 6634–6652. https://aclanthology.org/2024.emnlp-main.379

[49] Christopher Toukmaji and Allison Tee. 2024. Retrieval-Augmented Generation and LLM Agents for Biomimicry Design Solutions. In *Proceedings of the AAAI 2024 Spring Symposium Series, Stanford, CA, USA, March 25-27, 2024*, Ron P. A. Petrick and Christopher W. Geib (Eds.). AAAI Press, 273–278. doi:10.1609/AAAISS.V3I1.31210

[50] Taegeon Um, Byungsoo Oh, Minyoung Kang, Woo-Yeon Lee, Goeun Kim, Dongseob Kim, Youngtaek Kim, Mohd Muzzammil, and Myeongjae Jeon. 2024. Metis: Fast Automatic Distributed Training on Heterogeneous GPUs. In *Proceedings of the 2024 USENIX Annual Technical Conference, USENIX ATC 2024, Santa Clara, CA, USA, July 10-12, 2024*, Saurabh Bagchi and Yiying Zhang (Eds.). USENIX Association, 563–578. https://www.usenix.org/conference/atc24/presentation/um

[51] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (Eds.). 5998–6008. https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html

[52] Shibo Wang, Jinliang Wei, Amit Sabne, Andy Davis, Berkin Ilbeyi, Blake Hechtman, Dehao Chen, Karthik Srinivasa Murthy, Marcello Maggioni, Qiao Zhang, Sameer Kumar, Tongfei Guo, Yuanzhong Xu, and Zongwei Zhou. 2023. Overlap Communication with Dependent Computation via Decomposition in Large Deep Learning Models. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, Tor M. Aamodt, Natalie D. Enright Jerger, and Michael M. Swift (Eds.). ACM, 93–106. doi:10.1145/3567955.3567959

[53] Yubo Wang, Xueguang Ma, Ge Zhang, Yuansheng Ni, Abhranil Chandra, Shiguang Guo, Weiming Ren, Aaran Arulraj, Xuan He, Ziyan Jiang, Tianle Li, Max Ku, Kai Wang, Alex Zhuang, Rongqi Fan, Xiang Yue, and Wenhu Chen. 2024. MMLU-Pro: A More Robust and Challenging Multi-Task Language Understanding Benchmark. In *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang (Eds.). http://papers.nips.cc/paper_files/paper/2024/hash/ad236edc564f3e3156e1b2feafb99a24-Abstract-Datasets_and_Benchmarks_Track.html

[54] An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. 2024. Qwen2.5 Technical Report. *CoRR* abs/2412.15115 (2024). doi:10.48550/ARXIV.2412.15115 arXiv:2412.15115

[55] Qisen Yang, Zekun Wang, Honghui Chen, Shenzhi Wang, Yifan Pu, Xin Gao, Wenhao Huang, Shiji Song, and Gao Huang. 2024. PsychoGAT: A Novel Psychological Measurement Paradigm through Interactive Fiction Games with LLM Agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, 14470–14505. doi:10.18653/V1/2024.ACL-LONG.779

[56] 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 2022, Carlsbad, CA, USA, July 11-13, 2022*, Marcos K. Aguilera and Hakim Weatherspoon (Eds.). USENIX Association, 521–538. https://www.usenix.org/conference/osdi22/presentation/yu

[57] Beibei Zhang, Hongwei Zhu, Feng Gao, Zhihui Yang, and Xiaoyang Sean Wang. 2023. Moirai: Towards Optimal Placement for Distributed Inference on Heterogeneous Devices. *CoRR* abs/2312.04025 (2023). doi:10.48550/ARXIV.2312.04025 arXiv:2312.04025

[58] Hongbin Zhang, Taosheng Wei, Zhenyi Zheng, Jiangsu Du, Zhiguang Chen, and Yutong Lu. 2025. TD-Pipe: Temporally-Disaggregated Pipeline Parallelism Architecture for High-Throughput LLM Inference. *CoRR* abs/2506.10470 (2025). doi:10.48550/ARXIV.2506.10470 arXiv:2506.10470

[59] Shiwei Zhang, Lansong Diao, Chuan Wu, Zongyan Cao, Siyu Wang, and Wei Lin. 2024. HAP: SPMD DNN Training on Heterogeneous GPU Clusters with Automated Program Synthesis. In *Proceedings of the Nineteenth European Conference on Computer Systems, EuroSys 2024, Athens, Greece, April 22-25, 2024*. ACM, 524–541. doi:10.1145/3627703.3629580

[60] Yilun Zhao, Yitao Long, Hongjun Liu, Ryo Kamoi, Linyong Nan, Lyuhao Chen, Yixin Liu, Xiangru Tang, Rui Zhang, and Arman Cohan. 2024. DocMath-Eval: Evaluating Math Reasoning Capabilities of LLMs in Understanding Financial Documents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, 16103–16120. doi:10.18653/V1/2024.ACL-LONG.852

[61] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. 2022. Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, Marcos K. Aguilera and Hakim Weatherspoon (Eds.). USENIX Association, 559–578. https://www.usenix.org/conference/osdi22/presentation/zheng-lianmin

[62] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark W. Barrett, and Ying Sheng. 2024. SGLang: Efficient Execution of Structured Language Model Programs. In *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang (Eds.). http://papers.nips.cc/paper_files/paper/2024/hash/724be4472168f31ba1c9ac630f15dec8-Abstract-Conference.html

[63] 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 2024, Santa Clara, CA, USA, July 10-12, 2024*, Ada Gavrilovska and Douglas B. Terry (Eds.). USENIX Association, 193–210. https://www.usenix.org/conference/osdi24/presentation/zhong-yinmin

Appendix: Artifact Description/Artifact Evaluation

Artifact Description (AD)

A Overview of Contributions and Artifacts

A.1 Paper's Main Contributions

- C_1 We highlight the observations that pipeline bubbles caused by unbalanced computation in prefill and decode stage significantly degrade the performance of inference systems.
- C_2 We present gLLM, a distributed serving system with Token Throttling for effectively balancing the computation across batches to reduce pipeline bubbles.
- C_3 Token Throttling dynamically adjusts the batch size of the prefill and decode stage separately to achieve balanced schedule based on real-time inference system feedback.
- C_4 Experiments on representative LLMs show that gLLM enhances maximum throughput by 11% to 398% compared to state-of-the-art pipeline or tensor parallelism systems, while simultaneously achieving lower latency.

A.2 Computational Artifacts

A_1 https://doi.org/10.5281/zenodo.15251038

Artifact ID	Contributions Supported	Related Paper Elements
A_1	C_1	Figures 1,4
A_1	C_2, C_3, C_4	Table 1 Figures 11-16

B Artifact Identification

B.1 Computational Artifact A_1

Relation To Contributions

This artifact includes the source code of proposed gLLM system, together with installation, examples, benchmarks and evaluations scripts.

Expected Results

We evaluate gLLM with simulated online LLM serving scenarios. The experimental outcome contains metrics including latency (TTFT, TPOT and E2EL), throughput and SLO attainment. In most situations, gLLM outperforms state-of-the-art pipeline or tensor parallelism systems.

Expected Reproduction Time (in Minutes)

The installation of gLLM requires about 20 minutes (depending on the network conditions and compilation speed). gLLM takes approximately 1-2 minutes to start up initially, primarily due to model weight loading. Each test employs varying numbers of prompts and request rates, taking between 2 to 20 minutes to complete, with approximately 100 tests conducted in total.

Artifact Setup (incl. Inputs)

Hardware. Our evaluations require three types of node[5]: (1) 4× NVIDIA L20-48GB GPUs (2) 4× NVIDIA A100-40GB GPUs (3) 4× NVIDIA A800-80GB GPUs. All GPUs are connected via PCIe across three configurations.

Software. Our experiments were conducted on NVIDIA-powered machines with CUDA support. gLLM depends on several Python packages such as pytorch (==2.5.1), transformers (>= 4.45.2) and so on (detailed description is in requirements.txt).

Datasets / Inputs. Our experiments utilize two datasets, ShareGPT[6] and Azure[7].

Installation and Deployment. gLLM can be installed in a Python environment using the command 'pip install -v -e .' within a CUDA ecosystem. We have tested it with Python v3.11.9 and CUDA v12.4.

Artifact Execution

The artifact execution consists of two stages: (1) launching the server and (2) benchmarking the server. The command to start the server is:

```
python -m gllm.entrypoints.api_server --port $PORT \
    --model-path $MODEL_PATH --pp $PP_STAGES \
    --gpu-memory-util $GPU_MEMORY_UTIL \
    --load-format $LOAD_FORMAT
```

The command to benchmark the server is:

```
python benchmarks/benchmark_serving.py \
    --port $PORT --backend $BACKEND \
    --model $MODEL --dataset-name $DATASET_NAME \
    --dataset-path $SHAREGPT_PATH \
    --splitwise-path $SPLITWISE_PATH \
    --num-prompts $NUM_PROMPTS --goodput $GOODPUT \
    --request-rate $REQUEST_RATE --trust-remote-code
```

gLLM provides an OpenAI-compatible API, enabling benchmarking similar to other inference systems. For each test, we fix the request sending duration at 128 seconds. Therefore, the number of prompts can be determined by multiplying the request rates by the elapsed time. We can use the following arguments to configure gLLM:

Hyperparameters	Arguments
$\#T$	--iterp
$\#MaxP$	--maxp
$\#MinP$	--minp
KV_{thresh}	--kvthresh
Use scheduling policy in Sarathi-Serve	--use-naive-schedule

Artifact Analysis (incl. Outputs)

The artifact execution will output key performance metrics including latency (TTFT, TPOT, and E2EL), throughput, and SLO attainment. These metrics will be used to evaluate and compare gLLM against other inference systems. In most cases, gLLM system is expected to outperform state-of-the-art pipeline or tensor parallelism systems.

[5]It is fine to use a certain type of node during the test.

[6]https://huggingface.co/datasets/anon8231489123/ShareGPT_Vicuna_unfiltered/resolve/main/ShareGPT_V3_unfiltered_cleaned_split.json

[7]https://github.com/Azure/AzurePublicDataset/blob/master/data/AzureLLMInferenceTrace_conv.csv

Artifact Evaluation (AE)

C.1 Computational Artifact A_1

Artifact Setup (incl. Inputs)

Hardware. Our evaluations require three types of node: (1) 4× NVIDIA L20-48GB GPUs (2) 4× NVIDIA A100-40GB GPUs (3) 4× NVIDIA A800-80GB GPUs. All GPUs are connected via PCIe across three configurations.

Software. Our experiments were conducted on NVIDIA-powered machines with CUDA support. gLLM depends on several Python packages such as pytorch (==2.5.1), transformers (>= 4.45.2) and so on (detailed description is in requirements.txt).

Datasets / Inputs. Our experiments utilize two datasets, ShareGPT[6] and Azure[7]. We deploy three models, Qwen2.5-14B[8], Qwen2.5-32B[9] and Llama3.1-405B[10]. Llama3.1-405B can be downscaled by reducing the number of layers or deployed with more GPUs.

Installation and Deployment. gLLM[11] can be installed in a Python environment using the command 'pip install -v -e .' in the main directory of the project within a CUDA ecosystem. We have tested it with Python v3.11.9 and CUDA v12.4. vLLM and SGLang can be downloaded using 'pip install vllm==0.9.1' and 'pip install uv && uv pip install "sglang[all]==v0.4.3.post2"'. Note that, gLLM, vLLM and SGLang are installed under different conda environments.

Artifact Execution

The artifact execution consists of two stages: (1) choose one from gLLM, vLLM, and SGLang to launch the server (2) benchmark the server.

(1.a) Launch gLLM server:

```
python -m gllm.entrypoints.api_server --port $PORT \
    --model-path $MODEL --pp $PP \
    --gpu-memory-util $GPU_MEMORY_UTIL \
    --load-format $LOAD_FORMAT
```

'PORT' is an available port number. 'MODEL' is the model local path or the model name from Hugging Face. 'PP' is the degree of pipeline parallelism. 'GPU_MEMORY_UTIL' refers to the utilization of GPU memory excluding the model weights. 'LOAD_FORMAT' can be 'auto' (actual model weights) or 'dummy' (random model weights).

(1.b) Launch vLLM server:

```
vllm serve $MODEL --disable-log-requests \
    --trust-remote-code --no-enable-prefix-caching \
    --pipeline-parallel-size $PP \
    --distributed-executor-backend ray --port $PORT \
    --gpu-memory-utilization $GPU_MEMORY_UTIL \
    --enable-chunked-prefill --enforce-eager \
    --max-num-batched-tokens $MAX_BATCH_TOKENS \
    --max-num-seqs $MAX_NUM_SEQS \
    --load-format $LOAD_FORMAT
```

The meaning of 'LOAD_FORMAT', 'MODEL', 'PORT' and 'PP' is the same as those in gLLM. 'GPU_MEMORY_UTIL' refers to the utilization of GPU memory. 'MAX_BATCH_TOKENS' defines the token budget, which is set to 2048. 'MAX_NUM_SEQS' specifies the maximum number of concurrent requests, set to 1024.

(1.c) Launch SGLang server:

```
python -m sglang.launch_server --tp $TP \
    --model-path $MODEL --disable-cuda-graph \
    --mem-fraction-static $MEM_FRACTION \
    --disable-radix-cache --port $PORT \
    --chunked-prefill-size $CHUNK_SIZE \
    --load-format $LOAD_FORMAT \
    --enable-mixed-chunk
```

The meaning of 'LOAD_FORMAT', 'MODEL' and 'PORT' is the same as those in vLLM. 'MEM_FRACTION' is the same as 'GPU_MEMORY_UTIL' in vLLM. 'CHUNK_SIZE' is the same as 'MAX_BATCH_TOKENS' in vLLM. 'TP' is the degree of tensor parallelism.

(2) Benchmark the server:

```
python benchmarks/benchmark_serving.py \
    --port $PORT --backend $BACKEND \
    --model $MODEL --dataset-name $DATASET_NAME \
    --dataset-path $SHAREGPT_PATH \
    --splitwise-path $SPLITWISE_PATH \
    --num-prompts $NUM_PROMPTS --goodput $GOODPUT \
    --request-rate $REQUEST_RATE --trust-remote-code
```

'PORT' is the port number of the server. 'BACKEND' is 'vllm' since vLLM and gLLM shares the same benchmarking code. 'MODEL' is the same as 'MODEL' passed to the server. 'DATASET_NAME' can be 'sharegpt' or 'splitwise'. 'SHAREGPT_PATH' is the local path to dataset ShareGPT. 'SPLITWISE_PATH' is the local path to dataset Azure. 'NUM_PROMPTS' is the total number of test requests. 'GOODPUT' is the constraint of TTFT and TPOT, like 'ttft:1000 tpot:250' (the unit is milliseconds). 'REQUEST_RATE' is the request arrival rate.

gLLM provides an OpenAI-compatible API, enabling benchmarking similar to other inference systems. For each test, we fix the request sending duration at 128 seconds. Therefore, the number of prompts can be determined by multiplying the request rates by elapsed time. We use the following arguments to configure gLLM:

Hyperparameters	Arguments
$\#T$	--iterp
$\#MaxP$	--maxp
$\#MinP$	--minp
KV_{thresh}	--kvthresh
Use scheduling policy in Sarathi-Serve	--use-naive-schedule

Artifact Analysis (incl. Outputs)

The artifact execution will output key performance metrics including latency (TTFT, TPOT, and E2EL), throughput, and SLO attainment. These metrics will be used to evaluate and compare gLLM against other inference systems. In most cases, gLLM system is expected to outperform state-of-the-art pipeline or tensor parallelism systems.

[8]https://huggingface.co/Qwen/Qwen2.5-14B
[9]https://huggingface.co/Qwen/Qwen2.5-32B
[10]https://huggingface.co/meta-llama/Llama-3.1-405B
[11]The code of gLLM is also open-sourced at https://github.com/gty111/gLLM

Reproducibility Report

D Overview of Reproduction of Artifacts

The following table provides an overview of each computational artifact's reproducibility status. Artifact IDs correspond to those in the AD/AE Appendices.

Artifact ID	Available	Functional	Reproduced
A_1	●	○	○
Badge awarded	yes	no	no

E Reproduction of Computational Artifacts

E.1 Timeline

The artifact evaluation was conducted from September 2, 2025 to September 6, 2025.

E.2 Computational Environment and Resources

The experiments conducted for artifact evaluation were performed on `jeanzay` a multiple-partition supercomputer maintained by GENCI for the CNRS. The specific partition used for these experiments – that includes A100 – is listed as `gpu_p5`[12].

E.3 Details on Artifact Reproduction

The instructions provided to acquire the artifact were clear. Both the datasets and the `gLLM` source code are readily available.

The instructions for installing the software is clear as well. Mention of distinct `conda` environments for each compared software help with the setup.

I was able to install the competing software without issues *i.e.* their respective packages end up listed upon calling `pip list`. This results in two `conda` environments named `apdx173-slang` and `apdx173-vllm`.

Regarding the `gLLM` software package ; while the instructions are seemingly straightforward, I wasn't able to install it. I have installed the required packages listed in *requirements.txt* in a new `apdx173` conda environment – these packages are all listed upon calling `pip list`. However, upon setting up `gLLM`, the installation script could not find `torch`. **The authors have been notified on September 4, 2025.**

I have done the procedure multiple times from scratch, ensuring that executables such as `python`, `python3` and `pip` are the ones provided by the `gLLM` environment rather than some system-wide executables. These tentatives were unsuccessful.

Disclaimer: This Reproducibility Report was crafted by volunteers with the goal of enhancing reproducibility in our research domain. The time period allocated for the reproducibility analysis was constrained by paper notification deadlines and camera-ready submission dates. Furthermore, the compute hours in the shared infrastructure (e.g., Chameleon Cloud) available to the authors of this report were limited and restricted the scope and quantity of experiments in the review phase. Consequently, the inability to reproduce certain artifacts within this evaluation should not be interpreted as definitive evidence of their irreproducibility. Limitations in the time allocated to this review and the compute resources available to the reviewers may have prevented a positive outcome. Furthermore, reviewers assess the reproducibility of the artifacts provided by the authors; however, they are not accountable for verifying that the artifacts support the main claims of the paper.

[12]http://www.idris.fr/jean-zay/cpu/jean-zay-cpu-hw.html