DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving

Yinmin Zhong[1] Shengyu Liu[1] Junda Chen[3] Jianbo Hu[1] Yibo Zhu[2] Xuanzhe Liu[1]
Xin Jin[1] Hao Zhang[3]

[1]*School of Computer Science, Peking University* [2]*Independent Researcher* [3]*UC San Diego*

Abstract

DistServe improves the performance of large language models (LLMs) serving by disaggregating the prefill and decoding computation. Existing LLM serving systems colocate the two phases and batch the computation of prefill and decoding across all users and requests. We find that this strategy not only leads to strong prefill-decoding interferences but also couples the resource allocation and parallelism plans for both phases. LLM applications often emphasize individual latency for each phase: time to first token (TTFT) for the prefill phase and time per output token (TPOT) of each request for the decoding phase. In the presence of stringent latency requirements, existing systems have to prioritize one latency over the other, or over-provision compute resources to meet both.

DistServe assigns prefill and decoding computation to different GPUs, hence eliminating prefill-decoding interferences. Given the application's TTFT and TPOT requirements, DistServe co-optimizes the resource allocation and parallelism strategy *tailored* for each phase. DistServe also places the two phases according to the serving cluster's bandwidth to minimize the communication caused by disaggregation. As a result, DistServe significantly improves LLM serving performance in terms of the maximum rate that can be served within both TTFT and TPOT constraints on each GPU. Our evaluations show that on various popular LLMs, applications, and latency requirements, DistServe can serve 4.48× more requests or 10.2× tighter SLO, compared to state-of-the-art systems, while staying within latency constraints for $>90\%$ of requests.

1 Introduction

Large language models (LLMs), such as GPT-4 [32], Bard [2], and LLaMA [43], represent a groundbreaking shift in generative AI. They start to reshape existing Internet services, ranging from search engines to personal assistants [3], and enable fundamentally new applications, like universal chatbots [1, 14] and programming assistants [13, 37]. Yet, these advances come with a significant challenge: processing an end-to-end LLM query can be substantially slower than a standard search query [36]. In order to meet the stringent latency requirements of various applications, service providers need to over-provision compute resources, particularly many GPUs, leading to a shortfall in cost efficiency. Therefore, optimizing the cost per LLM query while adhering to high *SLO attainment* (the proportion of requests that meet the SLOs) is becoming increasingly essential for all LLM services.

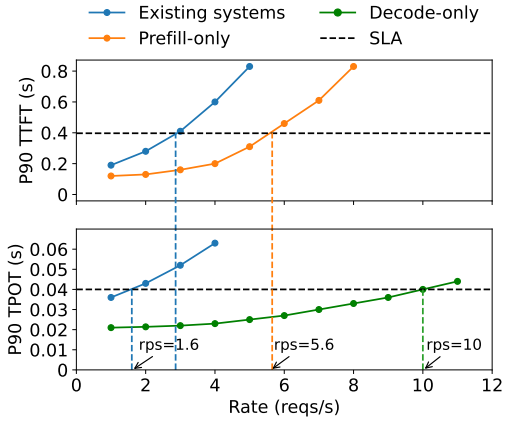


Figure 1: Performance when serving an LLM with 13B parameters under a synthetic workload with input length = 512 and output length = 64 on one NVIDIA 80GB A100. *Upper*: The P90 time-to-first-token (TTFT) latency comparing existing systems vs. a system serving only the prefill phase. *Down*: The P90 time-per-output-token (TPOT) latency comparing existing systems vs. a system serving only the decoding phase.

An LLM service responds to a user query in two phases. The *prefill phase* processes a user's prompt, composed of a sequence of tokens, to generate the first token of the response *in one step*. Following it, the *decoding phase* sequentially generates subsequent tokens *in multiple steps*; each decoding step generates a new token based on tokens generated in previous steps, until reaching a termination token. This dual-phase process distinguishes LLM services from traditional services – an LLM service's latency is uniquely measured by two key metrics: the *time to first token* (TTFT), which is the duration of the prefill phase, and the *time per output*

token (TPOT), which represents the average time taken to generate a token for each request (except for the first token)[1]. Different applications place varying demands on each metric. For example, real-time chatbots [1] prioritize low TTFT for response promptness, while TPOT only remains important until it is faster than human reading speed (i.e., 250 words/min). Conversely, document summarization emphasizes low TPOT for faster generation of the summary.

Hence, given the application's TTFT and TPOT requirements, an effective LLM serving system should balance these needs and maximize *per-GPU goodput*, defined as the maximum request rate that can be served adhering to the SLO attainment goal (say, 90%) for each GPU provisioned – higher per-GPU goodput directly translates into lower cost per query.

As the prefill and decoding phases share the LLM weights and working memory, existing LLM serving systems typically colocate both phases on GPUs and maximize the overall system throughput – tokens generated per second across all users and requests – by batching the prefill and decoding steps across requests [27, 45]. However, to meet latency requirements, we find these systems must over-provision compute resources. To see this, Figure 1 illustrates how the P90 TTFT and TPOT shift with increasing request rates when serving a 13B LLM using existing systems [28], with workload pattern and two latency constraints set to emulate using LLM to generate a short summary for an article. Under the SLO attainment of 90%, the maximum achievable goodput on a single A100 GPU, which is constrained by the more stringent one of TTFT and TPOT requirements, is about 1.6 requests per second (rps). The performance contrasts sharply when each phase is served independently on a separate GPU, shown by the orange and green curves, which achieve per-GPU goodput of 5.6 rps for the prefill phase and 10 rps for decoding. Ideally, by allocating 2 GPUs for prefill and 1 GPU for decoding, we can effectively serve the model with an overall goodput of 10 rps, or equally 3.3 rps per GPU, which is 2.1x higher than existing systems. The gap in goodput primarily stems from the colocation of the prefill and decoding – two phases with very distinct computational characteristics and latency requirements (§2.1).

First, colocation leads to strong *prefill-decoding interference*. A prefill step often takes much longer than a decoding step. When batched together, decoding steps in the batch are delayed by the prefill steps, significantly elongating their TPOT; similarly, the inclusion of decoding steps contributes to a non-trivial increase in TTFT, as evidenced in Figure 2. Even if we schedule them separately, issues persist as they begin to compete for resources. Decoding tasks awaiting GPU execution are subject to increased queuing delays due to ongoing prefill tasks, and vice versa. Prioritized scheduling of one phase risks failing the latency requirements of the other.

Second, the prefill and decoding computation differ in latency requirements and preference for different forms of parallelism (§3). Colocating prefill and decoding, however, couples their resource allocation, and prevents implementing different parallelism strategies more suited to meeting the specific latency requirements of each phase.

To overcome these challenges, we propose to disaggregate the prefill and decoding phases of LLM inference, assigning them to separate GPUs. Our approach has two benefits. First, operating each phase independently on different GPUs eliminates prefill-decoding interference. Second, it allows to scale each phase independently with tailored resource allocation and model parallelism strategies to meet their specific latency requirements. Although disaggregation causes communication of intermediate states between GPUs, we show that the communication overhead is insubstantial (§3.3) in modern GPU clusters, and when managed appropriately, disaggregation significantly improves per-GPU goodput.

Based on the above insights, in this work, we build DistServe, a goodput-optimized LLM serving system by disaggregating the prefill and decoding phases. Given TTFT and TPOT requirements, DistServe first scales each phase independently by co-optimizing the GPU allocation and parallelism strategies of the prefill and decoding phase assuming serving a single model replica. The optimization ensures maximizing the per-GPU goodput and may assign different numbers of GPUs and parallelism strategies to each phase depending on their respective latency requirements. DistServe then scales this allocation to multiple instances via replication until meeting the user-required traffic rate (§4). DistServe also features an algorithm to place the prefill and decoding computation according to their allocation schemes and the cluster's bandwidth to minimize the overhead of communicating intermediate states between phases.

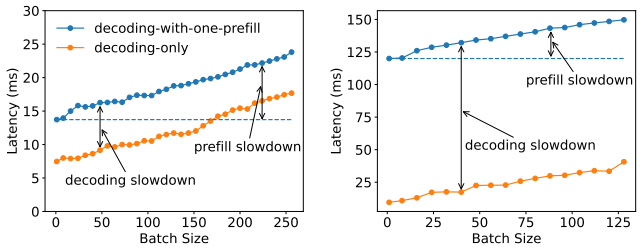
We implement DistServe as an orchestration layer on top of the LLM inference engine. We evaluate DistServe on various LLMs, varying the workloads based on three important real-world LLM applications: chatbots, programming assistant, and document summary. Compared to state-of-the-art solutions, DistServe can serve up to 4.48× more requests under latency constraints. Our contributions are:

- Identify the problems of prefill-decoding interference and resource coupling in existing LLM serving systems and propose to disaggregate the prefill and decoding phases.
- Design a novel placement algorithm to automatically choose the goodput-optimal schema for prefill and decoding instances.
- Conduct a comprehensive evaluation of DistServe with realistic workloads.

2 Background and Motivation

An LLM service follows a client-server architecture: the client submits a sequence of text as a request to the server; the server hosts the LLM on GPUs, runs inference over the request, and

[1]The overall request latency equals TTFT plus TPOT times the number of generated tokens in the decoding phase.



(a) Input length = 128 (b) Input length = 1024

Figure 2: Execution time for one batch when serving an LLM with 13B parameters as batch size increases. Compared between a decoding-only batch and a batch adding just one prefill request.

responds (or streams) the generation back to the client. As explained in §1, due to the unique prefill-decoding process, LLM service may impose aggressive service-level objectives (SLOs) on both TTFT and TPOT, varying with the application's needs. The serving system must meet both SLOs while minimizing the cost associated with expensive GPUs. In other words, we want the serving system to maximize the requests served per second adhering to the SLO attainment goal for each GPU provisioned – *maximizing per-GPU goodput*. Next, we detail the LLM inference computation (§2.1) and discuss existing optimizations for LLM serving (§2.2).

2.1 LLM Inference

Modern LLMs [32, 43] predict the next token given an input sequence. This prediction involves computing a hidden representation for each token within the sequence. An LLM can take a variable number of input tokens and compute their hidden representations in parallel, and its computation workload increases superlinearly with the number of tokens processed in parallel. Regardless of the input token count, the computation demands substantial I/O to move LLM weights and intermediate states from the GPU's HBM to SRAM. This process is consistent across varying input sizes.

The prefill step deals with a new sequence, often comprising many tokens, and processes these tokens concurrently. Unlike prefill, each decoding step only processes one new token generated by the previous step. This leads to significant computational differences between the two phases. When dealing with user prompts that are not brief, the prefill step tends to be computation-bound. For instance, for a 13B LLM, computing the prefill of a 512-token sequence makes an A100 compute-bound. The larger the model, the shorter sequence is needed to turn the prefill step compute-bound (see §3.1). In contrast, the decoding phase, despite processing only one new token per step, incurs a similar level of I/O to the prefill phase, making it constrained by the GPU's memory bandwidth.

During both phases, intermediate states, known as KV caches [28], are generated at each token position, which are needed again in later decoding steps. To avoid recomputing them, they are saved in GPU memory. Because of the shared use of LLM weights and KV caches in memory, most LLM inference engines opt to colocate the prefill and decoding phases on GPUs, despite their distinct computational characteristics.

2.2 LLM Serving Optimization

In real-time online serving, multiple requests come and must be served within SLOs. Batching and parallelizing their computation is key for achieving low latency, high throughput, and high utilization of GPUs.

Batching. Current serving systems [8, 28, 45] utilize a batching technique known as *continuous batching*. This method batches the prefill of new requests with the decoding of ongoing ones. It boosts the GPU utilization and maximizes the overall system throughput – tokens generated per second across all users and requests. However, as mentioned in §1 and elaborated later in §2.3, this approach leads to trade-offs between TTFT and TPOT. An advanced variant of continuous batching [8] attempts to balance TTFT and TPOT by segmenting prefill and attaching decoding jobs in a manner that avoids exceeding GPU performance limits – but essentially, it trades TTFT for TPOT. In summary, batching prefill and decoding invariably leads to compromises in either TTFT or TPOT.

Model parallelism. In LLM serving, model parallelism is generally divided as intra- and inter-operator parallelisms [29, 39, 50]. Both can be used to support larger models but may impact serving performance differently. Intra-operator parallelism partitions computationally intensive operators, such as matrix multiplications, across multiple GPUs, accelerating computation but causing substantial communication. It reduces the execution time[2], hence latency, particularly for TTFT of the prefill phase, but requires high bandwidth connectivity between GPUs (e.g., NVLink). Inter-operator parallelism organizes LLM layers into stages, each running on a GPU to form pipelines. It moderately increases execution time due to inter-stage communication, but linearly scales the system's rate capacity with each added GPU. In this paper, we reveal an additional benefit of model parallelism: reduced queuing delay of both prefill and decoding phases, steaming from shorter execution time. We delve into this further in §3. Besides model parallelism, replicating a model instance, irrespective of its model parallelism configurations, linearly scales the system's rate capacity.

These parallelism strategies create a complex space of optimization that requires careful trade-offs based on the application's latency requirements.

2.3 Problems and Opportunities

Colocating and batching the prefill and decoding computation to maximize the overall system throughput, as in existing systems, is cost-effective for service providers. However, in

[2]we emphasize "execution time" instead of latency here because latency comprises both execution time and queuing delay.

the presence of SLOs, present approaches struggle to maintain both high service quality and low cost due to the issues discussed below.

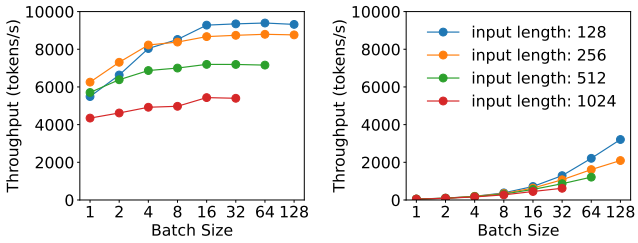
Prefill-decoding interference. As Figure 2 shows, adding a single prefill job to a batch of decoding requests significantly slows down both processes, leading to a marked increase in TTFT and TPOT. Specifically, the decoding tasks in the batch must wait for lengthier prefill jobs to complete, thus extending TPOT; the slowdown intensifies with a longer prefill, shown in Figure 2(b). Adding decoding jobs to prefill also increases the time to complete the prefill task, particularly when the GPU is already at capacity (Figure 2 blue curves).

Ineffective scheduling. Unbatching prefill and decoding jobs and scheduling them sequentially does not mitigate the interference. Decoding jobs may experience longer queuing delays due to waiting for ongoing prefill jobs on GPUs. Moreover, batches dedicated to decoding often lead to GPU underutilization. Prioritizing tasks in either phase adversely affects the latency of the other, rendering priority scheduling ineffective.

Resource and parallelism coupling. Colocating prefill and decoding phases on the same GPUs unavoidably share their resource and parallelism settings. However, each phase has its unique computational characteristic and latency requirement that calls for more heterogeneous resource allocation. For example, the prefill phase benefits from more GPUs and intra-op parallelism to reduce execution time to meet the tight SLO on TTFT. The decoding phase can handle a much higher rate using fewer GPUs than prefill, and its optimal parallelism configuration depends on the running batch size. In existing systems, due to coupling, resource allocation and parallelism plans are tailored to satisfy the *more demanding* of TTFT and TPOT, which may not be ideal for the other. This often leads to resource over-provisioning to meet both SLOs.

Opportunities. To address these issues, we propose to disaggregate the prefill and decoding phases. We use the term *instance* to denote a unit of resources that manages exactly one complete copy of model weights. One instance can correspond to many GPUs when model parallelism is applied. Note that when we disaggregate the two phases to different GPUs, each phase manages its copy of the model weights, resulting in *prefill instances* and *decoding instances*. A prefill instance, upon receiving a request, performs only the prefill computation for this request to generate the first output token. It then sends the intermediate results (mainly KV caches) to a decoding instance, which is responsible for subsequent decoding steps. Because decoding computation often has low GPU utilization, we may allocate multiple prefill instances per decoding instance. This allows batching more decoding jobs to achieve higher GPU utilization.

Disaggregating prefill and decoding naturally resolves the interference between the two phases and enables each to focus on its optimization target – TTFT or TPOT. Each type of instance can employ different resources and parallelism strategies to meet a variety of latency requirements. By adjusting the number of GPUs and parallelisms provided to the two types of instances, we can maximize the per-device goodput of the overall system, avoiding over-provisioning, eventually translating to reduced cost-per-query adhering to service quality. Next, we develop ways to find out the best resource allocation and parallelism plan for each phase.



(a) Prefill phase (b) Decoding phase

Figure 3: Throughput for prefill phase and decoding phase with different batch size and input length when serving an LLM with 13B parameters.

3 Tradeoff Analysis

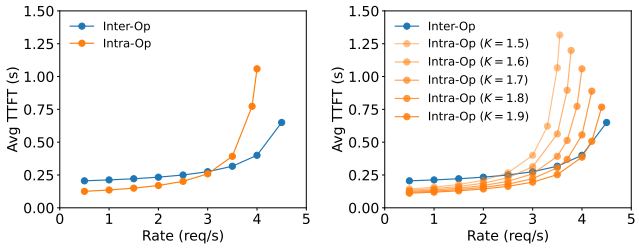
Disaggregation uncouples the two phases and allows a distinct analysis of the characteristics of each phase, providing valuable insights into the algorithm design. It also expands the design space: now each phase needs to be scaled and scheduled independently based on their latency requirements.

In this section, we analyze the computational pattern of prefill (§3.1) and decoding instances (§3.2) *post disaggregation*. We aim to identify key parameters and derive guidelines for batching and parallelism in each phase. We then highlight several practical deployment considerations (§3.3). This section lays the foundation for per-gpu goodput optimization.

3.1 Analysis for Prefill Instance

After disaggregation, the prefill phase generates the first token by processing all tokens of the user prompt in parallel. Assuming a given arrival rate, our goal is to fulfill the service's latency requirement on TTFT using the least resources.

Batching strategy. The prefill step is typically compute-intensive. Figure 3(a) shows how the throughput of the prefill phase changes with the input length and the batch size. For a 13B parameter LLM, processing a single sequence of 512 tokens can fully engage an A100 GPU; larger models require shorter sequences to reach GPU saturation. Once the GPU becomes compute-bound, adding more requests to the batch no longer improves GPU efficiency. Instead, it proportionally extends the total processing time for the batch, inadvertently delaying all included requests. Hence, for prefill instances, it is necessary to profile the specific LLM and GPUs in advance to identify a critical input length threshold, denoted as L_m, beyond which the prefill phase becomes compute-bound.



(a) Real experiment results (b) Changing intra-op speedup

Figure 4: Average TTFT when serving an LLM with 66B parameters using different parallelism on two A100 GPUs.

Batching more requests should only be considered when the input length of the scheduled request is below L_m. In practice, user prompts typically average over hundreds of tokens [7]. Batch sizes for the prefill instance are generally kept small.

Parallelism plan. To study the parallelism preferences for prefill-only instances, we serve a 66B LLM on two A100 GPUs with inter-op or intra-op parallelism strategy. To simplify the problem, we assume uniform requests input lengths of 512 tokens and a Poisson arrival process. We compare the resulting average TTFT at various arrival rates in Figure 4(a): intra-op parallelism is more efficient at lower arrival rates, while inter-op parallelism gains superiority as the rate increases. Disaggregation enables the prefill phase to function analogously to an M/D/1 queue, so we can use queuing theory to verify the observation.

We start by developing notations using the single-device case without parallelism: each request's execution time, denoted as D, remains constant due to uniform prefill length. Since one request saturates the GPU, we schedule requests via First-Come-First-Served (FCFS) without batching. Suppose the Poisson arrival rate is R and the utilization condition of $RD < 1$, the average TTFT (Avg_TTFT) can be modeled by the M/D/1 queue [40] in close form:

$$Avg\_TTFT = D + \frac{RD^2}{2(1-RD)}, \tag{1}$$

where the first term represents the execution time and the second corresponds to the queuing delay. Based on Eq. 1, we incorporate parallelism.

With 2-way inter-op parallelism, we assume the request-level latency becomes D_s, and the slowest stage takes D_m to finish. We have $D \approx D_s \approx 2 \times D_m$, due to negligible inter-layer activation communication [29, 50]. The average TTFT with 2-way inter-op parallelism is derived as:

$$Avg\_TTFT_{inter} = D_s + \frac{RD_m^2}{2(1-RD_m)} = D + \frac{RD^2}{4(2-RD)}. \tag{2}$$

For intra-op parallelism, we introduce a speedup coefficient K, where $1 < K < 2$, reflecting the imperfect speedup caused by high communication overheads of intra-op parallelism. With the execution time $D_s = \frac{D}{K}$, the average TTFT for 2-degree intra-op parallelism is:

$$Avg\_TTFT_{intra} = \frac{D}{K} + \frac{RD^2}{2K(K-RD)}. \tag{3}$$

Comparing Eq. 2 and Eq. 3: at lower rates, where execution time (first term) is the primary factor, intra-op parallelism's reduction in execution time makes it more efficient. As the rate increases and the queuing delay (second term) becomes more significant, inter-op parallelism becomes advantageous, concurred with Figure 4(a).

The prefill phase's preference for parallelism is also influenced by TTFT SLO and the speedup coefficient K. Seen from Figure 4(a): A more stringent SLO will make intra-op parallelism more advantageous, due to its ability to support higher request rates while adhering to SLOs. The value of K depends on factors such as the input length, model architecture, communication bandwidth, and placement [39, 50]. As shown in Figure 4(b), a decrease in K notably reduces the efficacy of intra-op parallelism. §4 develops algorithms that optimize the resource and parallelism configurations taking into consideration these knobs.

3.2 Analysis for Decoding Instance

Unlike the prefill instance, a decoding instance follows a distinct computational pattern: it receives the intermediate states (KV caches) and the first token from the prefill instance and generates subsequent tokens one at a time. For decoding instances, our optimization goal is to satisfy the application's TPOT requirement using minimal computing resources.

Batching strategy. Since a single decoding job is heavily bandwidth-bound, batching is key to avoiding low GPU utilization (hence high per-gpu goodput). In existing systems where the prefill and decoding phases are colocated, increasing the decoding batch size is difficult because it conflicts with meeting latency goals, particularly in scenarios with high request rates. This is because sharing GPUs cause competition between prefill and decoding jobs, leading to a trade-off between TTFT and TPOT. For example, a higher arrival rate generates more prefill jobs, demanding greater GPU time to meet TTFT requirements if prioritizing prefill jobs, which in turn adversely affects TPOT.

On the contrary, disaggregation offers a solution by enabling the allocation of multiple prefill instances to a single decoding instance. This approach allows for accumulating a larger batch size on dedicated GPUs for the decoding phase without sacrificing TPOT.

Parallelism plan. Post-disaggregation, the batch size for decoding may be constrained by GPU memory capacity, as it is necessary to maintain the KV caches for all active requests. Scaling the decoding instance with model parallelism or leveraging advanced memory management techniques for LLM

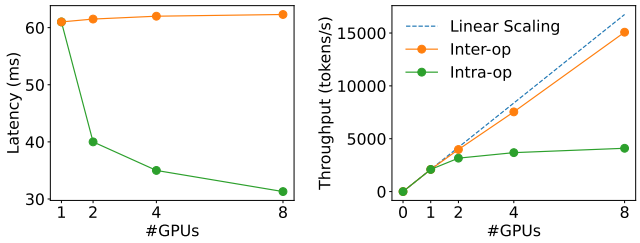


Figure 5: Decoding phase latency and throughput when serving a 13B LLM with batch size = 128 and input length = 256 under different parallel degrees.

KV caches, such as Paged-Attention [28] and GQA [9], enable further scaling the decoding batch size to nearly compute-bound. As the decoding batch size continue to increase to approach the compute bound, the decoding computation begins to resemble the prefill phase. With this observation, we investigate how the latency and throughput change under different parallelism degrees under large batch conditions in Figure 5: intra-op parallelism reduces latency with diminishing returns, caused by communication and reduced utilization after partitioning. Inter-op parallelism can almost linearly scale the throughput. Hence, when the TPOT SLO is stringent, intra-op parallelism is essential to reduce TPOT to meet latency goals. Beyond this, inter-op parallelism is preferable to enhance throughput linearly.

It is worth noting that when the model can fit into the memory of a single GPU, replication is a competitive option in addition to model parallelism for both prefill and decoding instances, to linearly scale the system's rate capacity. It may also reduce the queuing delay – as indicated by Eq. 1 – by substituting R with R/N assuming requests are equally dispatched to N replicas, at the cost of maintaining additional replicas of the model weights in GPU memory.

3.3 Practical Problems

We have developed foundational principles for selecting batching and parallelisms for each phase. In this section, we discuss and address several challenges encountered during the practical deployment of disaggregated prefill and decoding phases.

Variable prefill length. §3 has assumed uniform prompt length across requests. In real deployments, depending on the LLM application, the lengths of requests are non-uniform. The non-uniformity can cause pipeline bubbles [25, 31] for prefill instances applying inter-op parallelism, because the execution time of pipeline stages across requests of different lengths will vary. This results in slight deviations from the conclusions indicated by using M/D/1 queue model. To address the problem, §4 develops algorithms that search for parallelisms based on workloads, and resort to scheduling to minimize the bubbles (§4.3).

Communication overhead. Transferring KV caches from prefill to decoding instances incurs notable overheads. For example, the KV cache size of a single 512-token request on OPT-66B is approximately 1.13GB. Assuming an average arrival rate of 10 requests per second, we need to transfer $1.13 \times 10 = 11.3$ GB data – or equivalently 90Gbps bandwidth to render the overhead invisible. The size of the KV caches increases with average input length and arrival rate. While many modern GPU clusters for LLMs are equipped with Infiniband (e.g., 800 Gbps), in cases where cross-node bandwidth is limited, disaggregation relies on the commonly available intra-node NVLINK, where the peak bandwidth between A100 GPUs is 600 GB/s, again rendering the transmission overhead negligible (see §6.3). However, this requirement imposes additional constraints on the placement of preill and decoding instances that we take into consideration in the next section.

Through the analysis in this section, we identify the workload pattern, placement constraints, SLO requirements, parallelism strategies, and resource allocation as key parameters that create a web of considerations in designing the disaggregated serving system. How to automatically navigate the search space to find the configuration that achieves optimal per-gpu goodput is challenging, and addressed next.

4 Method

We built DistServe to solve the above challenges. Given the model, workload characteristic, latency requirements, and SLO attainment target, DistServe will determine (a) the parallelism strategies for prefill and decoding instances, (b) the number of each instance type to deploy, as well as (c) how to place them onto the physical cluster. We call the solution a *placement*. Our goal is to find a placement that maximizes the per-gpu goodput.

As explained in §3.3, a key design consideration is to manage communications between disaggregated prefill and decoding phases, given varying cluster setups. In this section, we first present two placement algorithms: one for clusters with high-speed cross-node networks (§4.1) and the other for environments lacking such infrastructure (§4.2); the latter introduces additional constraints. We then develop online scheduling optimizations that adapt to the nuances of real-world workloads (§4.3).

4.1 Placement for High Node-Affinity Cluster

On high node-affinity clusters equipped with Infiniband, KV caches transmission overhead across nodes is negligible, DistServe can efficiently deploy prefill and decoding instances across any two nodes without constraints. We propose a two-level placement algorithm for such scenarios: we first optimize the parallelism configurations for prefill and decoding instances separately to attain phase-level optimal per-gpu goodput; then, we use replication to match the overall traffic rate.

However, finding the optimal parallel configuration for a single instance type, such as for the prefill instance, is still

Algorithm 1 High Node-Affinity Placement Algorithm

Input: LLM G, #node limit per-instance N, #GPU per-node M, GPU memory capacity C, workload W, traffic rate R.
Output: the placement *best_plm*.

```
prefill_config ← ∅
decode_config ← ∅
for intra_op ∈ {1,2,...,M} do
    for inter_op ∈ {1,2,..., (N×M)/intra_op} do
        if G.size / (inter_op×intra_op) < C then
            Ĝ ← parallel(G, inter_op, intra_op)
            prefill_goodput ← simu_prefill(Ĝ, W)
            decode_goodput ← simu_decode(Ĝ, W)
            if prefill_config.goodput / prefill_config.num_gpus < prefill_goodput / config.num_gpus then
                prefill_config ← config
            if decode_config.goodput / decode_config.num_gpus < decode_goodput / config.num_gpus then
                decode_config ← config
n ← ⌈R / prefill_config.goodput⌉
m ← ⌈R / decode_config.goodput⌉
best_plm ← (prefill_config, decode_config, n, m)
return best_plm
```

challenging, due to the lack of a simple analytical formula to calculate the SLO attainment (a.k.a., percentage of requests that meet TTFT requirement), given that the workload has diverse input, output lengths, and irregular arrival patterns. Gauging the SLO via real-testbed profiling is time-prohibitive. We thus resort to building a simulator to estimate the SLO attainment, assuming prior knowledge of the workload's arrival process and input and output length distributions. Although short-term interval is impossible to predict, the workload pattern over longer timescales (e.g., hours or days) is often predictable [29, 46]. DistServe fits a distribution from the history request traces and resamples new traces from the distribution as the input workload to the simulator to compute the SLO attainment. Next, DistServe simply enumerates the placements via binary search and finds the maximum rate that meets the SLO attainment target with simulation trials.

Algorithm 1 outlines the process. We enumerate all feasible parallel configurations, subject to cluster capacity limit, for both prefill and decoding instances. For example, for a specific prefill phase configuration, we use `simu_prefill` to simulate and find their maximum goodput (similarly for using `simu_decode` for decoding). After determining the optimal parallel configurations for both prefill and decoding instances, we replicate them to achieve the user-required overall traffic rate according to their goodput.

The complexity of Algorithm 1 is $O(NM^2)$, with N as the node limit per instance and M representing the typical number of GPUs per node in modern clusters (e.g., 8). The search space is manageable and the solving time is under 1.3 minutes in our largest setting, as demonstrated in §6.5.

Algorithm 2 Low Node-Affinity Placement Algorithm

Input: LLM G, #node limit per-instance N, #GPU per-node M, GPU memory capacity C, workload W, traffic rate R.
Output: the placement *best_plm*.

```
intra_node_config ← ∅
for inter_op ∈ {1,2,...,N} do
    Ĝ ← parallel(G, inter_op)
    P ← get_intra_node_configs(Ĝ, M, C)
    for P ∈ P do
        P.goodput ← simulate(Ĝ, P, W)
        if intra_node_config.goodput / intra_node_config.num_gpus < P.goodput / P.num_gpus then
            intra_node_config ← P
n ← ⌈R / intra_node_config.goodput⌉
best_plm ← (inter_op, intra_node_config, n)
return best_plm
```

Simulator building. Algorithm 1 relies on a simulator to estimate the goodput under various SLOs and SLO attainment goals given the workload and the parallelism plan. To build an accurate simulator, we analyze the FLOPs and the number of memory accesses for prefill and decoding phases respectively, and use a latency model to approximate the inference execution time. See details in Appendix A. The simulator aligns well with real profiling results, thanks to the high predictability of DNN workloads [20, 29], verified in §6.4.

By far, we have developed Algorithm 1 assuming we can place the prefill and decoding between any two nodes of the cluster, and the KV cache transmission utilizes high bandwidth. In many real clusters, GPUs inside a node access to high-bandwidth NVLINK while GPUs distributed across nodes have limited bandwidth. We next develop an algorithm to address this constraint.

4.2 Placement for Low Node-Affinity Cluster

A straightforward solution is to always colocate prefill and decoding instances on the same node, utilizing the NVLINK, which is commonly available inside a GPU node. For large models, e.g. with 175B parameters (350GB), we may be unable to even host a single pair of prefill and decoding instances in an 8-GPU node ($80G \times 8 = 640G < 350 \times 2GB$). We incorporate this as additional placement constraints and co-optimize it with model parallelism, presented in Algorithm 2.

The key insight is that intermediate states transfers occur exclusively between corresponding layers of prefill and decoding instances. Leveraging inter-op parallelism, we group layers into stages and divide each instance into segments, termed as *instance segments*, with each segment maintaining one specific inter-op stage. By colocating prefill and decoding segments of the same stage within a single node, we force the transfer of intermediate states to occur only via NVLINK. Inside a node, we set the same parallelism and resource allocation for segments of the same instance. Given the typical

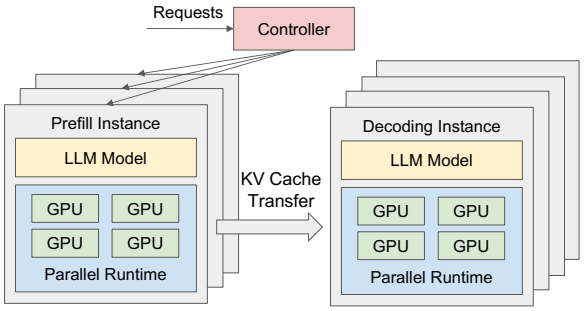


Figure 6: DistServe Runtime System Architecture

limitation of GPUs per node (usually 8), we can enumerate possible configurations inside one node and use the simulator to identify the configurations that yield the best goodput.

As outlined in Algorithm 2, we begin by enumerating inter-op parallelism degrees to get all the possible instance segments. For each segment, we get all possible intra-node configurations by calling `get_intra_node_configs`. Then we use simulation to find the optimal one and replicate it to satisfy the target traffic rate.

4.3 Online scheduling

The runtime architecture of DistServe is shown in Figure 6. DistServe operates with a simple FCFS scheduling policy. All incoming requests arrive at a centralized controller, then dispatched to the prefill instance with the shortest queue for prefill processing, followed by dispatch to the least loaded decoding instance for decoding steps. This setup, while simple, is optimized with several key enhancements tailored to the nuances of real-world workloads.

Reducing pipeline bubbles. To mitigate the pipeline bubbles caused by non-uniform prompt lengths (§3.3), we schedule the requests in a way that balances the execution time across all batches in the pipeline. This is achieved by noting that, for both prefill and decoding instances, the number of new tokens in the batch is a reliable indicator of the batch's real execution time. For prefill instances, we profile the target model and GPU to figure out the shortest prompt length L_m needed to saturate the GPU. We schedule prefill batches with a total sequence length close to L_m, by either batching multiple requests shorter than L_m or individually scheduling requests longer than L_m. For decoding instances, we set L_m as the largest batch size.

Combat busrtiness. Burstiness in workloads can cause a deluge of KV caches to transfer from prefill to decoding instances, risking memory overload on decoding instances. To circumvent this, DistServe employs a "pull" method for KV cache transmission rather than a "push" approach – decoding instances fetch KV cache from prefill instances *as needed*, using the GPU memory of prefill instances as a queuing buffer. Hence, each type of instance operates at its own pace without complex coordination.

Replaning. The resource and parallelism plan in DistServe is optimized for a specific workload pattern, which may become suboptimal if the workload pattern changes over time. DistServe implement periodic replanning. A workload profiler monitors key parameters such as the average input and output length of the requests, the average arrival rate, etc. If a significant pattern shift is detected, DistServe will trigger a rerun of the placement algorithm based on recent historical data. This process is expedient – the proposed algorithm runs in seconds (§6.5) and reloading LLM weights can be completed within minutes – far shorter than the hourly scale at which real-world workload variations tend to occur.

DistServe does not implement advanced runtime policies like preemption [23] and fault tolerance [49], which are complementary to disaggregation. Nevertheless, we discuss how they fit into DistServe. In DistServe, the FCFS policy can lead to a "convoy effect", where longer requests block shorter ones in the prefill stage. Incorporating preemptive strategies, as suggested in existing literature [44], could enhance efficiency and is feasible within our system's architecture. While not a primary focus in the current DistServe, fault tolerance is a critical aspect for consideration. In traditional colocation- and replication-based systems, a fault in one instance typically does not disrupt other replica instances. However, in DistServe, the dependency between prefill and decoding instances introduces the risk of fault propagation. For example, a fault in a single decoding instance mapped to multiple prefill instances could potentially cripple the entire service and cluster. We leave both as future work.

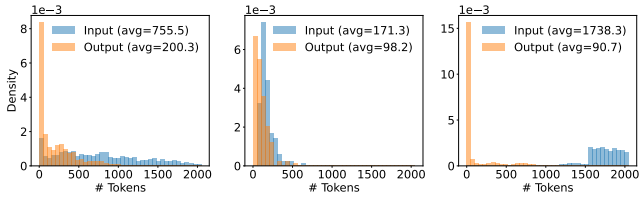
5 Implementation

DistLLM is an end-to-end distributed serving system for LLMs with a placement algorithm module, a RESTful API frontend, an orchestration layer, and a parallel execution engine. The algorithm module, frontend, and orchestration layer are implemented with 6.5K lines of Python code. The parallel execution engine is implemented with 8.1K lines of C++/CUDA code.

The placement algorithm module implements the algorithm and the simulator mentioned in §4 which gives the placement decision for a specific model and cluster setting. The frontend supports OpenAI API compatible interface where clients can specify the sampling parameters like maximum output length and temperature. The orchestration layer manages the prefill and decoding instances, responsible for request dispatching, KV caches transmission, and results delivery. It utilizes NCCL [5] for cross-node GPU communication and asynchronous cudaMemcpy for intra-node communication, which avoids blocking the GPU during transmission. Each instance is powered by a parallel execution engine, which uses Ray [30] actor to implement GPU workers that execute the LLM inference and manage the KV Cache in a distributed manner. It integrates many recent LLM optimizations like continuous batching [45], FlashAttention [18], PagedAtten-

Application	Model Size	TTFT	TPOT	Dataset
Chatbot OPT-13B	26GB	0.2s	0.1s	ShareGPT [7]
Chatbot OPT-66B	132GB	0.4s	0.1s	ShareGPT [7]
Chatbot OPT-175B	350GB	4.0s	0.2s	ShareGPT [7]
Code Completion OPT-66B	132GB	0.125s	0.2s	HumanEval [12]
Summarization OPT-66B	132GB	15s	0.15s	LongBench [11]

Table 1: Workloads in evaluation and latency requirements.



(a) ShareGPT (b) HumanEval (c) LongBench

Figure 7: The input and output length distributions of (a) ShareGPT, (b) HumanEval, and (c) LongBench datasets.

tion [28] and supports popular open-source LLMs such as OPT [47] and LLaMA [43].

6 Evaluation

In this section, we evaluate DistServe under different sizes of LLMs ranging from 13B to 175B and various application datasets including chatbot, code-completion, and summarization. The evaluation shows that DistServe consistently outperforms the current state-of-the-art system across all the settings (§6.2). Specifically, DistServe can handle up to 4.48× higher rates and 10.2× more stringent SLO while meeting the latency requirements for over 90% requests. Additionally, we analyze the latency breakdown in DistServe to show the communication overhead is insubstantial thanks to our bandwidth-aware placement algorithm (§6.3) and do ablation studies of our techniques (§6.4).

6.1 Experiments Setup

Cluster testbed. We deploy DistServe on a cluster with 4 nodes and 32 GPUs. Each node has 8 NVIDIA SXM A100-80GB GPUs connected with NVLINK. The cross-node bandwidth is 25Gbps. Due to the limited cross-node bandwidth, we use the low node-affinity placement algorithm (§2) for DistServe in most of the experiments except for the ablation study (§6.4) which uses simulation.

Model and workloads setup. Similar to prior work on LLM serving [28], we choose the OPT [47] series, which is a representative LLM family widely used in academia and industry. We use FP16 precision in all experiments. For workloads, as shown in Table 1, We choose three typical LLM applications and set the SLOs empirically based on their service target because there exists no available SLO settings for these applications as far as we know. For each application, we select a suitable dataset and sample requests from it for evaluation. Since all the datasets do not include timestamps, we generate request arrival times using Poisson distribution with different request rates. Due to the space limit, we test the chatbot workload on all three OPT models and the other two workloads on OPT-66B, which matches the largest size in the recent open-source LLM series [43].

- **Chatbot** [1]: We use the ShareGPT dataset [7] for the chatbot application, which is a collection of user-shared conversations with ChatGPT. For OPT-13B, the TTFT SLO is set to 0.2s for responsiveness and the TPOT SLO is set to 0.1s which is higher than the normal human read speed. For OPT-66B and OPT-175B, we slightly relax the two SLOs due to the increase of model execution latency.
- **Code completion** [12]: We use the HumanEval [12] dataset for the code completion task. It includes 164 programming problems with a function signature or docstring which is commonly used in academia to evaluate code completion models. Since the code completion tool is used as a personal real-time coding assistant, we set both SLOs to be stringent.
- **Summarization** [4]: It is a popular LLM task to generate a concise summary for a long article, essay, or even an academic paper. We use LongBench [11] dataset which contains the summarization task. As shown in Figure 7, LongBench has much longer input lengths than the other two datasets. So we set a loose TTFT SLO but require a stringent TPOT.

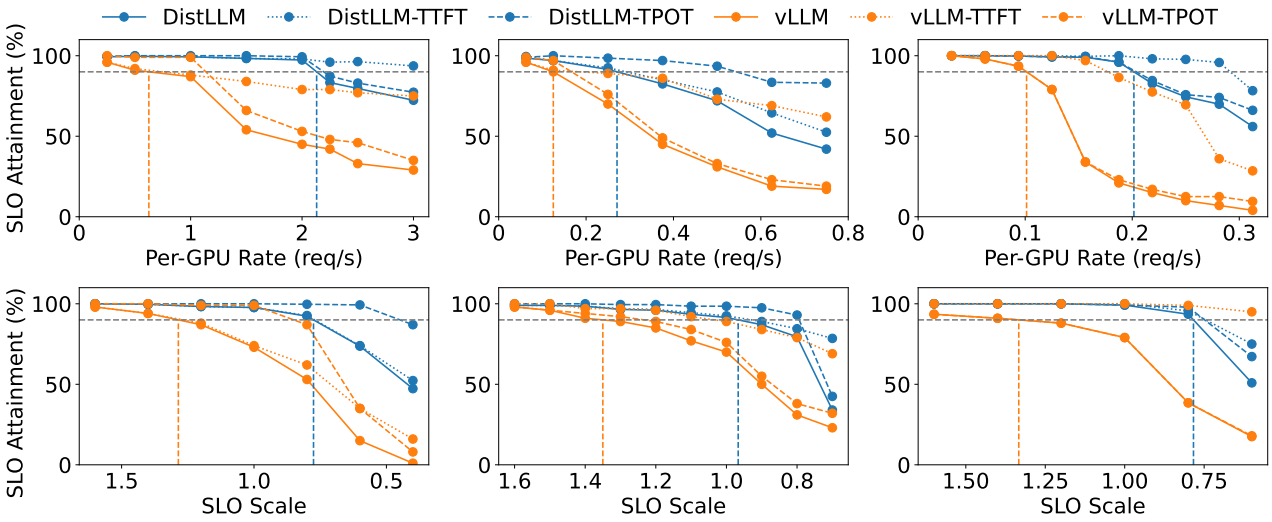
Metrics. We use *SLO attainment* as the major evaluation metric. Under a specific SLO attainment goal (say, 90%), we are concerned with two things: the maximum per-GPU goodput and the minimal SLO the system can handle. We are particularly interested in an SLO attainment of 90% (indicated by the vertical lines in all curve plots), but will also vary the rate and latency requirements to observe how the SLO attainment changes. To accurately understand the respective impacts of the two latency requirements on the system, we also present the proportion of requests that only meet one of these SLOs.

Baseline. We compare DistServe to the state-of-the-art serving system vLLM [28]. It supports iteration-level scheduling proposed by Orca [45] and PagedAttention to reduce memory fragmentation caused by KV cache allocation. However, it colocates and batches the prefill and decoding computation to maximize the overall system throughput and struggles to meet the latency requirements in a cost-efficient way. Since vLLM only supports intra-op parallelism, we follow previous work [28] to set intra-op equals 1, 4, and 8 for the three OPT models, respectively.

6.2 End-to-end Experiments

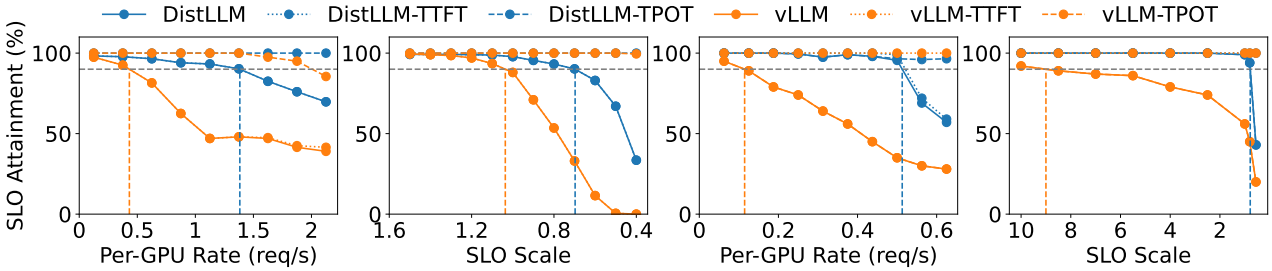
In this Section, we compare the end-to-end performance of DistServe against vLLM on real application datasets.

Chatbot. We evaluate the performance of DistServe on the chatbot application for all three OPT models. The first row



(a) OPT-13B (b) OPT-66B (C) OPT-175B

Figure 8: Chatbot application with OPT models on the ShareGPT dataset.



(a) Code Completion (b) Summarization

Figure 9: Code completion and summarization tasks with OPT-66B on HumanEval and LongBench datasets, respectively.

of Figure 8 illustrates that when we gradually increase the rate, more requests will violate the latency requirements and the SLO attainment decreases. The vertical line shows the maximum per-GPU rate the system can handle to meet latency requirements for over 90% of the requests. The dotted and dashed lines show the achieved SLO attainment for only TTFT or TPOT requirements, respectively.

On the ShareGPT dataset, DistServe can sustain 2.0×–3.41× higher request rate compared to vLLM. This is because DistLLM eliminates the prefill-decoding interference through disaggregation. Two phases can optimize their own objectives by allocating different resources and employing tailored parallelism strategies. As a result, the gap between the curve that only meets TTFT requirements (Dist-TTFT) and the one that only meets TPOT requirements (Dist-TPOT) is relatively small. Specifically, by analyzing the chosen placement strategy[3] for 175B, we find the prefill instance has inter-op = 3, intra-op = 3; and the decoding instance has inter-op = 3, intra-op = 4. Under this placement, DistServe can effectively balance the load between the two instances on ShareGPT, meeting latency requirements at the lowest cost. This non-trivial placement strategy is challenging to manually find, proving the effectiveness of the algorithm. In the case of vLLM, collocating prefill and decoding greatly slows down the decoding phase, thereby significantly increasing TPOT. Due to the stringent TPOT requirements of chatbot applications, although vLLM meets the TTFT SLO for most requests, the overall SLO attainment is dragged down by a large number of requests that violate the TPOT SLO.

The second row of Figure 8 indicates the robustness to the changing latency requirements of the two systems. We fix the rate and then linearly scale the two latency requirements in Table 1 simultaneously using a parameter called *SLO Scale*. As SLO Scale decreases, the latency requirement is more stringent. We aim to observe the most stringent SLO Scale that the system can withstand while still achieving the attainment target. Figure 8 shows that DistServe can achieve 1.4×–1.8× more stringent SLO than vLLM, thus providing

[3]All the placements chosen by DistServe can be found in Appendix B.

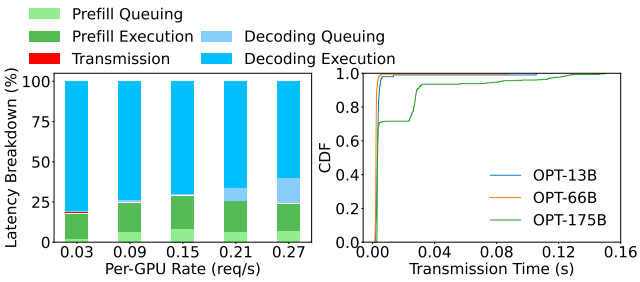


Figure 10: *Left:* Latency breakdown when serving OPT-175B on ShareGPT dataset with DistServe. *Right:* The CDF function of KV Cache transmission time for OPT models.

more engaging service quality to the users.

Code completion. Figure 9(a) shows the performance of DistServe on the code completion task when serving OPT-66B. DistServe can sustain 3.2× higher request rate and 1.5× more stringent SLO than vLLM. As a real-time coding assistant, the code completion task demands lower TTFT than chatbot, this leads to both systems ultimately being constrained by the TTFT requirement. However, in comparison, by eliminating the interference of the decoding jobs and automatically increasing intra-operation parallelism in prefill instances through the searching algorithm, DistServe reduces the average latency of the prefill jobs, thereby meeting the TTFT requirements of more requests.

Summarization. Figure 9(b) shows the performance of DistServe on the summarization task when serving OPT-66B. DistServe achieves 4.48× higher request rate and 10.2× more stringent SLO than vLLM. The requests sampled from LongBench dataset have long input lengths, which brings significant pressure to the prefill computation. However, due to the loose requirement of TTFT for the summarization task, the TPOT service quality becomes particularly important. The vLLM, which collocates prefill and decoding phases, with long prefill jobs, experiences a greater slowdown in the decoding phase and fails to meet the TPOT requirement.

6.3 Latency Breakdown

To understand DistServe's performance in detail, we make a latency breakdown of the requests in DistServe. We divide the processing lifecycle of a request in DistServe into five stages: *prefill queuing*, *prefill execution*, *transmission*, *decoding queuing*, and *decoding execution*. The total time consumed by all requests in each stage is then summed up to determine their respective proportions in the system's total execution time.

Figure 10(a) shows the latency breakdown for the OPT-175B models on ShareGPT dataset. We chose OPT-175B because the KV Cache transmission is more demanding for larger models. In fact, even for OPT-175B, the KV Cache transmission only accounts for less than 0.1% of the total latency. Even by examining the CDF of the absolute transmission time shown in Figure 10(b), we observe that over 95% of requests experience a delay of less than 30ms, despite our testbed having only limited cross-node bandwidth. This is due to the algorithm described in §4.2, where we require the prefill and decoding instance to maintain the same stage on one machine, enabling the use of intra-node NVLINK bandwidth for transmission, thus significantly reducing transmission delay.

Rate (req/s)	vLLM		DistServe-Low	
	Real System	Simulator	Real System	Simulator
1.0	97.0%	96.8%	100.0%	100.0%
1.5	65.5%	65.1%	100.0%	100.0%
2.0	52.8%	51.0%	99.3%	99.3%
2.5	44.9%	46.1%	87.3%	88.3%
3.0	36.7%	38.3%	83.0%	84.1%
3.5	27.8%	28.0%	77.3%	77.0%
4.0	23.6%	24.1%	70.0%	68.9%

Table 2: Comparison of the SLO attainment reported by the simulator and the real system under different rates.

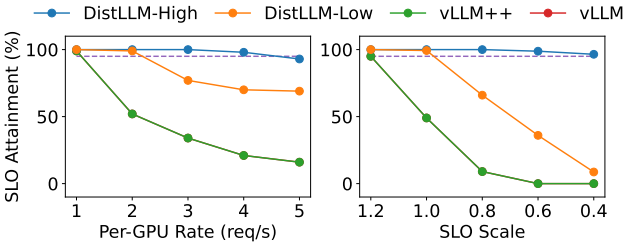


Figure 11: Ablation experiments.

6.4 Ablation Studies

We study the effectiveness of the two key innovations in DistServe: disaggregation and the placement searching algorithm. In §6.2, we choose the default parallelism setting for vLLM following its original paper [28]. So we implement "vLLM++" which enumerates different parallelism strategies and chooses the best. For DistServe, We also compare the placement found by Alg. 2 (DistServe-Low) with the one found by Alg. 1 (DistServe-High) which has fewer searching constraints and assumes high cross-node bandwidth. Since vLLM does not support inter-op parallelism and our physical testbed does not have high cross-node bandwidth, we use simulation for this experiment.

Simulator accuracy. Noticing that DNN model execution [21] has high predictability, even under parallel settings [29, 50]. We study the accuracy of the simulator in Tab. 2. For "vLLM" and "DistServe-Low", we compare the SLO attainment reported by the simulator and by real runs on our testbed under different rates. The error is less than 2% in all cases, verifying the accuracy of our simulator.

Results. Figure 11 shows the performance of the four systems when serving OPT-13B on ShareGPT dataset. "vLLM++" has the same performance as "vLLM" because we find the de-

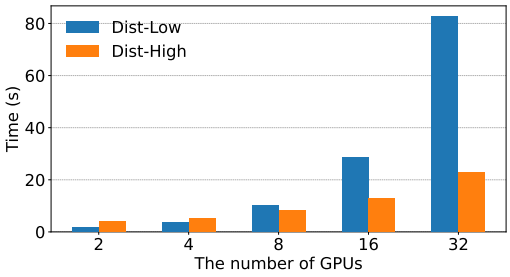


Figure 12: Algorithm Running Time

fault non-parallelism setting has the best per-GPU goodput. This further demonstrates the importance of disaggregation. The interference between the prefill and decoding phases significantly reduces the potential performance improvement through adjusting parallelism. In contrast, "DistLLM-High" can achieve further improvements over "DistLLM-Low" because it is not constrained by the deployment constraint that the prefill and decoding instance on one node should share the same model stage. Through disaggregation, we can use tailored parallelism strategies for prefill and decoding instances and optimize their targets without the coupling effects.

6.5 Algorithm Running Time

Figure 12 shows the running time for Alg. 1 (DistServe-Low) and Alg. 2 (DistServe-High) on a AWS m5d.metal instance with 96 cores as the number of GPUs ($N \times M$) provided to a single instance increases. According to the results, DistServe scales well with the number of GPUs and is independent of the model size. This is because the simulator only simulates discrete events and the running time is the same no matter how big the model is. On the other hand, both algorithms are highly parallelizable, as the searches for different parallelism strategies are independent of each other, allowing the execution time of the algorithms to accelerate almost linearly with more CPU cores.

As the number of GPUs increases, the execution time of "Dist-Low" becomes higher than that of "Dist-High". This is because the search for parallelism strategies for prefill and decoding instances in "Dist-High" is independent and can be parallelized. But for "Dist-Low", due to additional restrictions on deployment, we need to enumerate all the possible intra-node parallelism combinations for prefill and decoding instances. Even so, the execution time of the algorithm is in minutes, and since it only needs to be executed once before each redeployment, this overhead is acceptable.

7 Related Work

Inference serving. There has been plenty of work on inference serving recently. They range from general-purpose production-grade systems like TorchServe [6] and NVIDIA Triton [17] to systems optimized specifically for Transformer-based LLMs [8, 16, 19, 29, 42, 44, 45, 51]. Among them, Orca [45] introduces iteration-level scheduling to increase throughput. vLLM [28] proposes a novel memory management strategy for KVCache. SARATHI [8] suggests a chunked-prefill approach, splitting a prefill request into chunks and piggyback decoding requests to improve hardware utilization. FastServe [44] implements iteration-level preemptive scheduling to mitigate the queuing delay caused by long jobs. However, they all employ a colocation approach for prefill and decoding processing, thus leading to severe interference. There are also concurrent works Splitwise [33], TetriInfer [24], and DéjàVu [41] which adopt similar disaggregation idea to optimize LLM inference, further confirming the effectiveness of this method. Differently, DistServe emphasizes the goodput optimization scenario more and takes a closer look at the aspect of network bandwidth.

Goodput-optimized systems. Optimizing goodput is a hot topic in DL applications. Pollux [34] improves scheduling performance in DL clusters by dynamically adjusting resources for jobs to increase cluster-wide goodput. Sia [26] introduces a heterogeneous-aware scheduling approach that can efficiently match cluster resources to elastic resource-adaptive jobs. Clockwork [20] and Shepherd [46] provide latency-aware scheduling and preemption to improve the serving goodput, but they only target traditional small models. AlpaServe [29] focuses on LLMs, employing model parallelism to statistically multiplex the GPU execution thus improving the resource utilization. However, it only targets the non-autoregressive generation. DistServe is the first work to optimize the goodput for autoregressive LLM inference.

Resource disaggregation. Resource disaggregated systems [15, 22, 38] decouple the hardware resources from the traditional monolithic server infrastructure and separate them into different pools to manage independently. It allows for more flexible, efficient, and scalable deployment and increases resource utilization. Many applications benefit from a truly disaggregated data center with high-speed network bandwidth and heterogenous hardware support [10,48]. DistServe adopts a similar concept by disaggregating its system components, allowing for independent resource scaling and management.

Model parallelism for training. DistServe is orthogonal to the large body of work on model parallelism in training [25,31,35,39,50]. As described in §3.3, inference-serving workloads have unique characteristics not found in training settings. Where these systems do intersect with DistServe, is in their methods for implementing model parallelism along various dimensions. DistServe can integrate new parallelism optimizations into its placement searching algorithm.

8 Conclusion

We present DistServe, a new LLM serving architecture that disaggregates the prefill and decoding computation. DistServe maximizes the per-gpu goodput – the maximum request rate that can be served adhering to the SLO attainment goal for each GPU provisioned, hence resulting in up to 4.48× lower

cost per LLM query with guaranteed satisfaction of SLOs. Our findings affirm that as latency becomes an increasingly important metric for LLM services, prefill and decoding disaggregation is a vital strategy in promising improved performance and service quality guarantees.

References

[1] Introducing chatgpt. https://openai.com/blog/chatgpt, 2022.

[2] Bard, an experiment by google. https://bard.google.com/, 2023.

[3] Inflection tech memo. https://inflection.ai/assets/Inflection-1.pdf, 2023.

[4] Lanchain usecase: Summarization, 2023.

[5] Nvidia collective communications library (nccl), 2023.

[6] Serve, optimize and scale pytorch models in production, 2023.

[7] Sharegpt teams. https://sharegpt.com/, 2023.

[8] Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S Gulavani, and Ramachandran Ramjee. Sarathi: Efficient llm inference by piggybacking decodes with chunked prefills. *arXiv preprint arXiv:2308.16369*, 2023.

[9] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. Gqa: Training generalized multi-query transformer models from multi-head checkpoints, 2023.

[10] Andrew Audibert, Yang Chen, Dan Graur, Ana Klimovic, Jiri Simsa, and Chandramohan A. Thekkath. A case for disaggregation of ml data processing, 2022.

[11] Yushi Bai, Xin Lv, Jiajie Zhang, Hongchang Lyu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, Yuxiao Dong, Jie Tang, and Juanzi Li. Longbench: A bilingual, multitask benchmark for long context understanding, 2023.

[12] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. 2021.

[13] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[14] Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E. Gonzalez, Ion Stoica, and Eric P. Xing. Vicuna: An open-source chatbot impressing gpt-4 with 90%* chatgpt quality, 2023.

[15] Compute Express Link Consortium. Compute express link, 2023. Accessed: 2023-12-07.

[16] NVIDIA Corporation. Fastertransformer, 2019.

[17] NVIDIA Corporation. Triton inference server: An optimized cloud and edge inferencing solution., 2019.

[18] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness, 2022.

[19] Jiarui Fang, Yang Yu, Chengduo Zhao, and Jie Zhou. Turbotransformers: an efficient gpu serving system for transformer models. In *ACM PPoPP*, 2021.

[20] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving DNNs like clockwork: Performance predictability from the bottom up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 443–462. USENIX Association, November 2020.

[21] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving DNNs like clockwork: Performance predictability from the bottom up. In *USENIX OSDI*, 2020.

[22] Zhiyuan Guo, Zijian He, and Yiying Zhang. Mira: A program-behavior-guided far memory system. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 692–708, New York, NY, USA, 2023. Association for Computing Machinery.

[23] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. Microsecond-scale preemption for concurrent GPU-accelerated DNN inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 539–558, Carlsbad, CA, July 2022. USENIX Association.

[24] Cunchen Hu, Heyang Huang, Liangliang Xu, Xusheng Chen, Jiang Xu, Shuang Chen, Hao Feng, Chenxi Wang, Sa Wang, Yungang Bao, Ninghui Sun, and Yizhou Shan. Inference without interference: Disaggregate llm inference for mixed downstream workloads, 2024.

[25] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism, 2019.

[26] Suhas Jayaram Subramanya, Daiyaan Arfeen, Shouxu Lin, Aurick Qiao, Zhihao Jia, and Gregory R Ganger. Sia: Heterogeneity-aware, goodput-optimized ml-cluster scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 642–657, 2023.

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

[28] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention, 2023.

[29] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E Gonzalez, et al. Alpaserve: Statistical multiplexing with model parallelism for deep learning serving. *arXiv*, 2023.

[30] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In *USENIX OSDI*, 2018.

[31] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In *ACM SOSP*, 2019.

[32] OpenAI. Gpt-4 technical report, 2023.

[33] Pratyush Patel, Esha Choukse, Chaojie Zhang, Íñigo Goiri, Aashaka Shah, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative llm inference using phase splitting, 2023.

[34] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R. Ganger, and Eric P. Xing. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 1–18. USENIX Association, July 2021.

[35] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models, 2020.

[36] Reuters, 2023.

[37] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

[38] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. {LegoOS}: A disseminated, distributed {OS} for hardware resource disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 69–87, 2018.

[39] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism, 2020.

[40] John F Shortle, James M Thompson, Donald Gross, and Carl M Harris. *Fundamentals of queueing theory*, volume 399. John Wiley & Sons, 2018.

[41] Foteini Strati, Sara Mcallister, Amar Phanishayee, Jakub Tarnawski, and Ana Klimovic. Déjàvu: Kv-cache streaming for fast, fault-tolerant generative llm serving, 2024.

[42] Yiming Su, Chengcheng Wan, Utsav Sethi, Shan Lu, Madan Musuvathi, and Suman Nath. Hotgpt: How to make software documentation more useful with a large language model? In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, pages 87–93, 2023.

[43] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models, 2023.

[44] Bingyang Wu, Yinmin Zhong, Zili Zhang, Gang Huang, Xuanzhe Liu, and Xin Jin. Fast distributed inference serving for large language models. *arXiv preprint arXiv:2305.05920*, 2023.

[45] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for {Transformer-Based} generative models. In *USENIX OSDI*, 2022.

[46] Hong Zhang, Yupeng Tang, Anurag Khandelwal, and Ion Stoica. Shepherd: Serving dnns in the wild. 2023.

[47] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. Opt: Open pre-trained transformer language models, 2022.

[48] Yiying Zhang. Make it real: An end-to-end implementation of a physically disaggregated data center. *SIGOPS Oper. Syst. Rev.*, 57(1):1–9, jun 2023.

[49] Kai Zhao, Sheng Di, Sihuan Li, Xin Liang, Yujia Zhai, Jieyang Chen, Kaiming Ouyang, Franck Cappello, and Zizhong Chen. Ft-cnn: Algorithm-based fault tolerance for convolutional neural networks. *IEEE Transactions on Parallel and Distributed Systems*, 32(7):1677–1689, 2021.

[50] 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. Alpa: Automating inter- and Intra-Operator parallelism for distributed deep learning. In *USENIX OSDI*, 2022.

[51] Zhe Zhou, Xuechao Wei, Jiejing Zhang, and Guangyu Sun. {PetS}: A unified framework for {Parameter-Efficient} transformers serving. In *USENIX ATC*, 2022.

A Latency Model for LLM Inference

To accurately simulate the goodput of different placement strategies, we use an analytical model to predict the execution time of the prefill and decoding phases in LLM inference.

In modern LLM serving systems [16, 28, 44], memory-bound operations like Softmax and LayerNorm are usually fused with matrix multiplication kernels for efficiency. Thus the GEMMs dominate the overall latency and our analysis primarily focuses on them.

A.1 Symbol Definition

Here are symbols related to the architecture of the model:

- h: hidden size
- n: number of heads
- s: head size ($h = n \cdot s$)
- m: FFN intermediate size

Note: If tensor parallelism is used, h, n, and m should be divided by the tensor parallelism size.

Below are symbols that characterize the batch to be executed:

- B: batch size
- $l_0, l_1, \ldots, l_{B-1}$: input length of each request within the batch
- t: number of tokens in the batch, ($t = \sum_{i=0}^{B-1} l_i$)
- t_2: squared sum of the input lengths ($t_2 = \sum_{i=0}^{B-1} l_i^2$)
- b: block size in the attention kernel. This parameter is used in FlashAttention [18], a common kernel optimization technique adopted by current LLM serving systems.

A.2 Prefill Phase Latency Modeling

Since the attention operation uses specially optimized kernels, we first discuss the other four matrix multiplications in the prefill phase:

GEMM Name	Shape of M	Shape of N
QKV Linear	(t, h)	$(h, 3h)$
Attn Output	(t, h)	(h, h)
FFN Input	(t, h)	(h, m)
FFN Output	(t, m)	(m, h)

The arithmetic intensity (AI) of these operations is $O(t)$. On NVIDIA A100-80GB GPU, it is compute-bound when AI is over 156. Since t usually can reach several hundred in real cases, all of these operations are compute-bound. Therefore, we can model the latency of these operations according to the total FLOPs:

$$T_1 = C_1 \cdot (4th^2 + 2thm)$$

Next, we discuss the prefill attention operation with FlashAttention [18] optimization. Since the attention only operates among the tokens in the same request, current implementations launch attention kernels for each request in the same batch. For one attention head and a request with l tokens, the attention kernel needs to perform a total of $2sl + 3sl \cdot (l/b) \approx 3sl \cdot (l/b)$ memory reads and writes, alongside $2sl^2 + sl(l/b) \approx 2sl^2$ FLOPs. So the AI is $2b/3 = 10.677$ (when $b = 16$) or 21.333 (when $b = 32$), indicating that it is a memory-bound operation on A100 GPU. Therefore, the whole attention layer latency (including all requests and all heads) can be modeled as:

$$T_2 = C_2 \cdot n \cdot \sum_{i=0}^{B-1} \frac{3sl_i^2}{b} = C_2 \cdot \frac{3nst_2}{b} = C_2 \cdot \frac{3ht_2}{b}$$

Overall, the latency of the prefill phase can be modeled as:

$$T_{Prefill} = C_1 \cdot (4th^2 + 2thm) + C_2 \cdot \frac{3ht_2}{b} + C_3$$

We use C_3 to quantify other overheads like Python Runtime, system noise, and so on. Then we use profiling and interpolation to figure out the values of C_1, C_2, and C_3.

A.3 Decoding Phase Latency Modeling

Similarly, we first focus on the following GEMMs in the decoding phase:

GEMM Name	Shape of M	Shape of N
QKV Linear	(B, h)	$(h, 3h)$
Attn Output	(B, h)	(h, h)
FFN Input	(B, h)	(h, m)
FFN Output	(B, m)	(m, h)

The AI of these operations is $O(B)$. B is limited by the GPU memory size and stringent latency requirements, so in existing serving scenarios, these operations are memory-bound. The total memory reads and writes is $8Bh + 4h^2 + 2hm + 2Bm$, and since h and m are usually significantly larger than B, we can model the latency as:

$$T_3 = C_4 \cdot (4h^2 + 2hm)$$

As for the decoding attention operation, for one attention head and a request with l generated tokens, it needs to perform $3sl$ memory reads and writes, alongside $2sl$ FLOPs. It is memory-bound, so we can model the latency of decoding attention as:

$$T_4 = C_5 \cdot n \cdot 3s \sum_{i=0}^{B-1} l_i = C_5 \cdot 3ht$$

Summing up, the latency of the decoding phase is:

$$T_{Decoding} = C_4 \cdot (4h^2 + 2hm) + C_5 \cdot 3ht$$

Here we do not introduce the overhead term (like C_3 in the profiling stage) because $4h^2 + 2hm$ is already a constant, and the overhead can be put into C_4. Similarly, we use profiling and interpolation to figure out the values of C_4 and C_5.

B DistLLM Placements in End-to-end Experiments

In the end-to-end experiments 6.2, the tensor parallelism (TP) and pipeline parallelism (PP) configurations for prefill and decoding instances chosen by DistServe are listed on the right.

Model	Dataset	Prefill		Decoding	
		TP	PP	TP	PP
OPT-13B	ShareGPT	2	1	1	1
OPT-66B	ShareGPT	4	1	2	2
OPT-66B	LongBench	4	1	2	2
OPT-66B	HumanEval	4	1	2	2
OPT-175B	ShareGPT	3	3	4	3