Inference without Interference: Disaggregate LLM Inference for Mixed Downstream Workloads

Cunchen Hu[1,2]*, Heyang Huang[1,2], Liangliang Xu[3], Xusheng Chen[3], Jiang Xu[3], Shuang Chen[3], Hao Feng[3], Chenxi Wang[1,2], Sa Wang[1,2], Yungang Bao[1,2], Ninghui Sun[1,2], Yizhou Shan[3]

[1]*University of Chinese Academy of Sciences*, [2]*ICT, CAS* [3]*Huawei Cloud*

Chunked prefill + disggregating prefill/decode + decode length prediction (for scheduling)

Abstract

Transformer-based large language model (LLM) inference serving is now the backbone of many cloud services. LLM inference consists of a prefill phase and a decode phase. However, existing LLM deployment practices often overlook the distinct characteristics of these phases, leading to significant interference. To mitigate interference, our insight is to carefully schedule and group inference requests based on their characteristics. We realize this idea in TetriInfer through three pillars. First, it partitions prompts into fixed-size chunks so that the accelerator always runs close to its computation-saturated limit. Second, it disaggregates prefill and decode instances so each can run independently. Finally, it uses a smart two-level scheduling algorithm augmented with predicted resource usage to avoid decode scheduling hotspots. Results show that TetriInfer improves time-to-first-token (TTFT), job completion time (JCT), and inference efficiency in turns of performance per dollar by a large margin, e.g., it uses 38% less resources all the while lowering average TTFT and average JCT by 97% and 47%, respectively.

1 Introduction

Since the boom of ChatGPT, large language model (LLM) based services have now played a vital role in our daily lives [4,9,20,31,34,38]. Behind the scenes, all use cases boil down to LLM inference serving. To run an inference request, the LLM model will first take the user inputs to generate the first token (known as the prefill phase), and then generate outputs token-by-token in an auto-regressive manner (known as the decode phase). Numerous works were proposed to improve the cost efficiency of LLM inference [21,41].

There are various ways to interact with LLM, from simple chats to more complex downstream tasks such as document summarization, content creation, etc. As a result, LLM-empowered services serve inference requests with dramatically different properties that can be categorized across two dimensions: the input prompt length during the prefill phase and the generated token length during the decode phase. As shown in Figure 1, summarization tasks have long input prompts and short generated tokens, while context creation tasks are the opposite. Token lengths of different downstream tasks can differ by more than two orders of magnitude. Given the significant variation in LLM inference requests from various downstream tasks, the first research question we ask in this paper is *how do these inference requests perform when running together?*.

To answer this question, we run extensive tests that mix LLM prefill and decode requests of different lengths. Unfortunately, we have observed serious interference across all combinations. For example, mixing prefill requests could result in a 10x slowdown, combining prefill and decode requests could lead to a 5x slowdown, and mixing decode requests with different lengths could take a 16% throughput hit (see §2.2). A naive solution to avoid interference is to provision resources for each downstream task statically. Given the high cost of LLM serving infrastructure, this solution is impractical. To this end, the second research question we ask in this paper is *how to build a distributed LLM inference serving system that minimizes interferences?*

We take a step back to examine why interference exists. We find the fundamental issue lies in the fact that current LLM deployment practices do not account for the distinct characteristics exhibited by LLM prefill and decode phases. Specifically, the prefill phase resembles a computation-heavy batch job, with its computation scaling quadratically with the input prompt length. The decode phase resembles a memory-intensive, latency-critical task, with its resource usage scaling sublinearly with the generated token length [33]. Interferences observed in our tests are classic system problems. Running prefill requests leads to a serious slowdown because we continue adding computation-heavy jobs to an already saturated hardware (§2.2.1). Combining prefill and decode requests hurts both because we co-run batch and latency-critical jobs simultaneously (§2.2.2). Mixing decode requests leads to a throughput drop because we are unaware of the memory bandwidth and capacity usage, thus leading to contention and

[0]Work done while intern at Huawei Cloud.

head-of-line blocking (§2.2.3).

To solve these issues, our insight is to carefully *schedule and group requests based on their characteristics*. We realize this idea in TetriInfer[1], a cloud-scale LLM inference serving system designed to battle interferences.

Our designs are three-fold. First, to avoid interference running prefill, we propose limiting the number of tokens processed in a single prefill iteration so that hardware is fully utilized without incurring extra penalties. TetriInfer partitions and pads input prompts into fixed-size chunks so that the accelerator always runs close to its computation-saturated limit (§3.3). Second, to avoid interference in co-running prefill and decode, we propose disaggregating prefill from decode phases. TetriInfer has dedicated prefill and decode instances. During runtime, prefill instances transfer prefilled KV cache to decode instances. The prefill and decode instances are virtual concepts in that each can scale independently and flip roles if load changes (§3.5). Third, to avoid interference running decode requests, we propose using a smart two-level scheduling algorithm augmented with predicted resource usage to avoid scheduling hotspots (§3.4). TetriInfer incorporates an LLM-based length prediction model to speculate the number of generated tokens of decode requests, and then schedule them accordingly.

We implement TetriInfer's disaggregated prefill and decode instances based on vLLM [21]. Most of our modules are implemented in Python, except for the network stack module, which utilizes C++ to interface with low-level APIs for KV cache transfer. The fine-tuning part uses Trainer APIs offered by HuggingFace Transformer [16]. Since we cannot access high-end hardware, we implement a mock mechanism to emulate varying network bandwidth connecting prefill and decode instances, as illustrated in Figure 9.

We compare TetriInfer with vanilla vLLM using public dataset [35] in terms of time-to-first-token (TTFT), job completion time (JCT), and efficiency as in performance per dollar (perf/$). We run them atop a real testbed with emulated network bandwidth ranging from 200Gbps to 300GBps. For light prefill and heavy decode workload, TetriInfer improves perf/$ by 2.4x (Figure 12). For common mixed workload, TetriInfer improves average TTFT and average JCT by 85% and 50%, respectively (Figure 15). Nevertheless, we also find that TetriInfer's design is not ideal for heavy prefill and heavy decode workloads since the room for improvement is marginal, and the overhead we introduce cannot be offset (Figure 14). Overall, our ideas mentioned above are effective. TetriInfer achieves effective LLM inference serving, outperforming vLLM by a large margin in TTFT, JCT, and perf/$ running most common workloads (§5.1).

[1]The name of our system, TetriInfer, implies that it can efficiently organize LLM inference requests, similar to how tetris blocks are stacked.

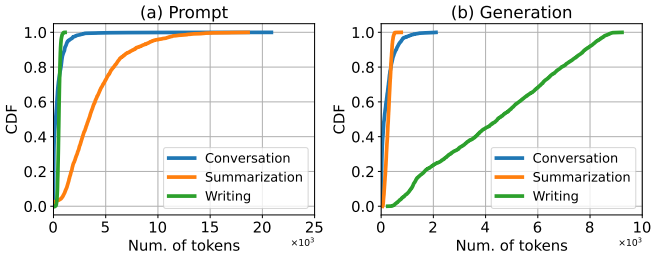


Figure 1: **Length Distribution.** Prompt Tokens for Prefill and Generated Tokens during Decode. Data sources: conversation [35], summarization [17], writing [18].

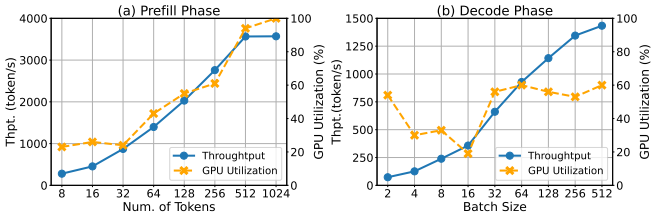


Figure 2: **Prefill and Decode's Characteristics.** Decode's GPU utilization fluctuates because the task is faster than our monitoring granularity.

2 Background and Motivation

We present a brief primer on LLM inference and study interferences while running various LLM inference requests to motivate our work. For model and testbed details, see §5.

2.1 Generative LLM Inference

LLM inference is a process that involves generating a sequence of output tokens in response to an input prompt. This process consists of two phases: prefill and decode. The prefill phase outputs the first token and generates the key and value cache (KV cache) for future decoding [21]. The decode phase uses the previous KV cache to generate new tokens step-by-step in an auto-regressive manner. Generally, the prefill phase is computation-bound, and the decode phase is memory-bound [33]. We report this in Figure 2. Results indicate that the prefill phase's throughput stays flat once the accelerator is saturated at a certain number of tokens (which we name the accelerator-saturate threshold). The decode phase's throughput continues increasing with a larger batch size but plateaus once the memory bandwidth is saturated.

2.2 Motivation: Interference Study

This section studies the impact of running different inference requests concurrently. Inspired by Figure 1, we classify in-

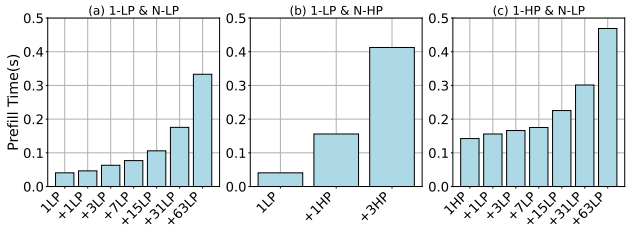


Figure 3: **Interference of Prefill & Prefill.** LP means light prefill. HP means heavy prefill. (a) and (b) indicate that light prefill's prefill latency increases as the number of co-running requests increases. The same applies to (c)'s heavy prefill.

ference requests across two dimensions (prefill and decode length) and one property (light or heavy), resulting in four distinct request types: heavy prefill, light prefill, heavy decode, and light decode. Here, heavy refers to a long token length, while light refers to a short token length. Below, we study mixing prefill and prefill (§2.2.1), prefill and decode (§2.2.2), decode and decode (§2.2.3).

2.2.1 Prefill and Prefill

We first study mixing prefill requests. Here, light prefill has roughly 18 prompt tokens as it is the median token length in ShareGPT's short prompts [35], while heavy prefill has 512 prompt tokens as the accelerator is saturated at this length in our testbed (see Figure 2). In Figure 3 (a) and (b), we show how a light prefill's latency changes if it co-runs with other light prefill and heavy prefill requests. We find its latency increases by 2x and 8x if there are 7 and 63 concurrent light prefill requests in the same batch. Additionally, it incurs more than 10x latency slowdown if it runs with other heavy prefill requests. In Figure 3 (c), we show that heavy prefill's latency also incurs a 3x slowdown if co-run with other light prefill requests. Overall, we find that when the total number of tokens in a batch is larger than the accelerator-saturate threshold, the prefill latency dramatically increases.

2.2.2 Prefill and Decode

We now study mixing prefill and decode in the same batch due to continuous batching [21, 45]. Both light prefill and heavy prefill follow §2.2.1's definition. Also, light decode refers to the ones that generate a small number of tokens, e.g., less than 100. heavy decode refers to the ones that generate a large number of tokens, e.g., larger than 512 tokens. Though decoding latency increases slowly with an increasing number of generated tokens, we only present tests related to light decode as heavy decode presents similar results.

In Figure 4 (a) and (b), we show how a light decode's per-iteration decoding latency changes if it co-runs with light prefill and heavy prefill requests. Results indicate that its

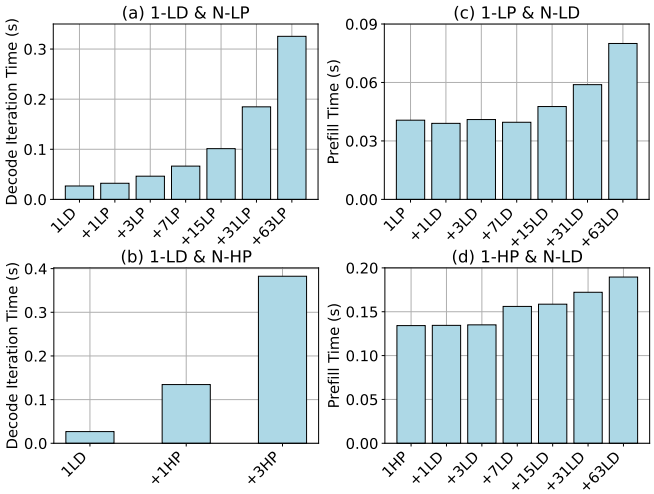


Figure 4: **Interference of Prefill & Decode.** LD means light decode. HD means heavy decode.

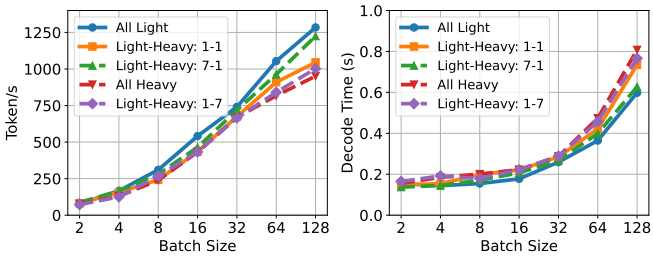


Figure 5: **Interference of Decode & Decode.**

decoding latency increases by 5x even if only one other heavy prefill request is in the same continuous batch! In Figure 4 (c) and (d), we show how a light prefill's and a heavy prefill's latency changes if they co-run with other light decode requests. Figure 4 (c) indicates that the prefill latency increases once the number of co-running light decode requests is more than 7. Both slow down roughly by up to 2.5x.

2.2.3 Decode and Decode

We now study mixing decode requests. Following §2.2.2's definition and ShareGPT's distribution, we set both requests to use very short prompts. And light decode generates roughly 20 to 100 tokens, while heavy decode generates more than 512 tokens. Figure 5 presents the decoding throughput and latency while running different numbers of light decode and heavy decode requests in the same batch. Results suggest that compared to a batch with all light decode requests, increasing heavy decode requests could seriously hurt throughput and latency. For example, with a batch size of 128, compared to a batch with all light decode, a batch with half heavy decode and half light decode's throughput drops by 16% while the

latency increases by 23%.

2.3 Analysis and Insights

We have observed significant interferences in LLM inferencing. The root cause is simple: current LLM systems are ignorant of the distinct characteristics exhibited by LLM prefill and decode phases. The prefill phase resembles a computation-heavy batch job, while the decode phase resembles a memory-intensive, latency-critical task [33].

Interferences measured above are classic system problems. In §2.2.1, running prefill requests leads to a serious slowdown because we continue adding computation-heavy jobs to an already saturated hardware. In §2.2.2, mixing prefill and decode requests hurts both because we co-run batch and latency-critical jobs at the same time. In §2.2.3, mixing decode requests leads to a throughput drop because we are unaware of the memory bandwidth and capacity usage, thus leading to contention and head-of-line blocking.

Our work aims to solve these issues by carefully *schedule and group requests based on their characteristics*. Our ideas are three-fold. First, to avoid interference running prefill, we propose limiting the number of tokens processed in a single prefill step so that hardware is fully utilized without incurring extra penalties. Second, to avoid interference co-running prefill and decode, we propose disaggregating prefill from decode so that each runs independently. Third, to avoid interference running decode requests, we propose to use a smart two-level scheduling algorithm augmented with predicted resource usage to avoid scheduling hotspots. We visualize the comparison in Figure 6 (a).

3 Design

3.1 Overview

We realize the above insights in TetriInfer, an LLM inference serving system designed to battle interferences. First, we run prefill in a fixed-size computation unit by partition and pad input prompts into fixed-size chunks such that the accelerator always runs close to its computation-saturated limit (§3.3). Second, we design instances dedicated to running the prefill or decode phases. We schedule prefill requests to prefill instances only, and the same goes for decode requests. Prefill instances will transfer prefilled KV cache to decode instances. Our prefill and decode instances are virtual concepts in that each can scale independently and flip roles if load changes (§3.5). Finally, we design a two-level scheduling algorithm for both prefill and decode request scheduling. We incorporate a length-prediction model to speculate decode requests' resource usage and then schedule them accordingly (§3.4).

We show TetriInfer's architecture in Figure 6 (b) with four modules highlighted: centralized control plane, prefill instance, decode instance, and length prediction model.

Centralized control plane. It consists of a global scheduler and a cluster monitor. The global scheduler sends requests to prefill instances based on load and receives streaming outputs from decode instances. The cluster monitor collects statistics from prefill and decode instances and regularly broadcasts load information to prefill instances. It adds, removes, and flips prefill or decodes instances.

Prefill Instances. They only run the prefill phase of an LLM inference request. Each prefill instance has a local scheduler, a length predictor, the main LLM engine, and a dispatcher. All requests undergo four steps. First, the local prefill scheduler sorts requests based on pre-defined policies. Second, the length predictor runs a prediction model to speculate the requests' decode lengths, which are then used to estimate resource usage during the decoding phase. Third, the main LLM engine partitions all requests into fixed chunks. Finally, for each request, the dispatcher runs an inter-decode load-balancing algorithm to select a decode instance and then forwards the generated KV cache to it.

Decode instances. They are virtually disaggregated from prefill instances and only run the decode phase of an LLM inference request. Each decode instance can receive requests from any prefill instance. It runs a local scheduler with three pre-defined policies for selecting decode requests to run in the main LLM engine.

Length Prediction Model. The prediction model is a small LLM model fine-tuned offline for predicting the generation length of LLM inference requests. TetriInfer's prefill dispatcher and decode instance's local scheduler utilize the speculated information to schedule decode instances and avoid hotspots measured in §2.2.3. The prediction model is small and deployed at all prefill instances.

3.2 Control Plane

TetriInfer has a centralized control plane to manage inference clusters at the cloud scale. It consists of a cluster monitor that manages *the lifecycle of prefill and decode instances* and a global scheduler that manages*the lifecycle of inference requests*. The centralized control plane is a distributed system without a single point of failure or processing bottlenecks.

The cluster monitor is responsible for collecting and broadcasting statistics and scaling instances. Both prefill and decode instances regularly send their load information to the cluster monitor (e.g., every 100 ms). Since we run decentralized decode request scheduling at prefill instances, the cluster monitor will aggregate decode instances' load information and broadcast it to all prefill instances.

The global scheduler is responsible for forwarding inference requests from external services to prefill instances and sending inference outputs from decode instances back to external services in a streaming fashion. The global scheduler maintains a request status table, which stores requests' arrival time, current phase (e.g., prefill or decode), SLA requirement,

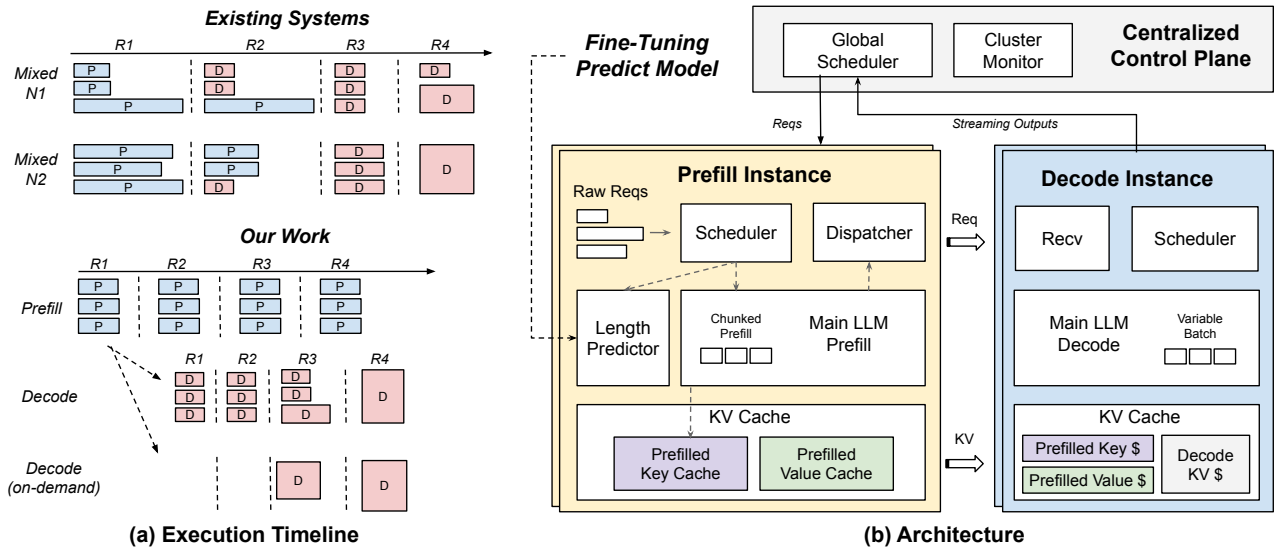


Figure 6: **TetriInfer's Workflow and Architecture.** (a) compares existing systems and TetriInfer's execution timeline. The existing systems part has two nodes running mixed prefill and decode. TetriInfer has separated prefill and decode instances. This allows for the load-balancing of long decoding tasks across different on-demand decode instances. Each timeline comprises four rounds (R1 to R4), with the length of prefill and decode boxes representing their sequence length and the width of the decode box indicating its resource usage. A wider decode box indicates the presence of lengthy generated tokens, resulting in larger resource usage and decoding latency. (b) shows TetriInfer's architecture with four core modules highlighted.

etc. When a request arrives, the global scheduler will choose a prefill instance with the least load and then insert the request into the table. Following our insight to disaggregate prefill and decode instances, the global scheduler only decides which prefill instance will handle the request. It is up to the prefill instance's dispatcher to decide which decode instances to use with a speculated resource usage.

3.3 Prefill Instance

The prefill instance runs the prefill phase of an inference request. To avoid interference among prefill requests, we use a prefill scheduler and chunked prefill to sort and partition all prompts into fixed-size chunks. To help avoid interference during the decode phase, we run a length predictor and a decentralized dispatcher to choose decode instances based on speculated resource usage.

3.3.1 Prefill Scheduler

The prefill instance's scheduler is crucial for improving the prefill phase's latency and throughput. The scheduler maintains a raw request queue that stores requests from the global scheduler and a scheduled queue that stores sorted requests. In this work, we have designed and implemented three scheduler policies: first-come-first-serve (*FCFS*), shortest-job-first (*SJF*), and longest-job-first (*LJF*). We can use the latter two policies because we can accurately estimate a request's prefill time based on the number of tokens in its prompt. We only explore non-preemptive policies, though chunked prefill (described soon) has opened the door to preemptive and out-of-order prefill scheduling, such as shortest-remaining-time-first, which we leave for future work.

The scheduled requests are sent to the length predictor which executes scheduled requests as-is using fixed-size batch (§3.3.2), and the main LLM which uses chunked prefill (§3.3.3). In Figure 7, we illustrate the above three scheduler policies and how scheduled requests are partitioned and merged into fixed-size chunks. Specifically, FCFS keeps the original request arrival order. Prompt tokens are partitioned and merged into chunks sequentially. This policy is the easiest to implement and works best for inference requests with similar prompt lengths. However, FCFS can lead to head-of-line blocking and high average job completion time (JCT) when requests have long prompts. This is problematic since the length differences among LLM inference requests are more than three orders of magnitude (see Figure 1).

In response, we add the shortest-job-first (SJF), and longest-job-first (LJF) to overcome these issues. These two policies schedule prefill requests based on prompt token lengths in ascending or descending order. By design, they can achieve lower JCT compared to FCFS. Nevertheless, they are no panacea. They introduce starvation for either long or short re-

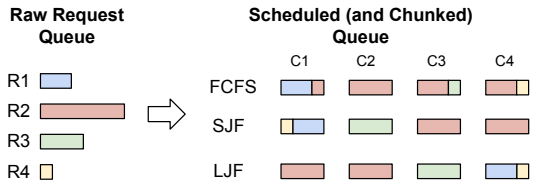


Figure 7: **Prefill Scheduler Policies.** The left shows four raw inference requests (R1 to R4). The right shows scheduled requests using FCFS, SJF, and LJF. We show the chunked version to illustrate slicing and merging (C1 to C4).

quests. To avoid starvation, we propose using a prefill scheduling batch (i.e., `PrefillSchedBatch`) variable to control how many inference requests can be scheduled at a time. For example, assume the raw request queue has twenty requests awaiting scheduling. If we set the batch size to ten, we will schedule twice, each with ten requests sorted and put into the scheduled queue. This simple mechanism prevents starvation during the prefill phase.

Our scheduler is effective. Results in Figure 16 show that SJF lowers average prefill waiting time by 7.8% compared to FCFS when the batch size is set to 16. Additionaly, the improvement is even more pronounced with larger batch sizes.

3.3.2 Length Predictor

To address the interference cases measured in §2.2.3, it is essential to determine the number of tokens that a decode request is likely to generate. This information will enable us to schedule decode requests in a length-aware manner. As such, the prefill instance runs a length predictor to predict the length range of an inference request's generated tokens. The prefill instance's dispatcher utilizes this information for *inter-decode instance* scheduling (§3.3.4), while the decoding instance's local scheduler employs this information for *intra-decode instance* scheduling (§3.4).

Our length predictor uses a small LLM-based classification model called a "predict model" to classify the length of generated tokens into fixed-size buckets if the request were executed by a specific target LLM model. The predict model is intentionally small, containing millions of parameters while the target model is much larger, with billions of parameters. As we run the length predictor at the prefill instance, we aim to minimize its cost and avoid impacting the main LLM model. Therefore, approaches like using a giant LLM to predict length are not feasible for us [48]. Fortunately, a small LLM model is much faster than a giant LLM and uses much less resources. For example, we use OPT-125M as the predict model and OPT-13B as the target model, the small one is roughly ten times faster than the larger one.

We opt to predict the length range instead of an exact number of tokens because the latter is extremely difficult to predict. Various inference parameters, such as temperature and top-p [3], result in significant response variations from the same LLM model to the same question in practice. Since our primary goal is to use the estimated length to guide our request scheduling decisions, an exact length estimation is unnecessary; a length range suffices. For instance, if we estimate the length to be between ten to twenty tokens, we can deduce its resource usage's lower and upper bounds.

In this work, we have tested two execution modes: a sequential mode, where we first execute the predict model followed by the target model, and a parallel mode, where both models are run simultaneously. The sequential mode adds extra latency for the target LLM model, while the parallel mode may reduce the target LLM model's throughput. Based on our findings in Figure 17, we opted to use the parallel mode because the main LLM is not affected for most requests (more than 80%), though throughput take a 10% hit under extreme stress test.

Figure 8 outlines the offline fine-tuning and online prediction workflow. In this process, the predict model (depicted in red) is trained to speculate the decoding behavior of a specific target model (depicted in blue). The fine-tuning of the predict model involves three key steps. Firstly, we assemble a prompt-only training dataset inherited from public datasets, a large target LLM model (e.g., OPT-13B), and a classification model for our predict model (e.g., 125M OPTForSequenceClassification [16]). Secondly, we send training prompts to the target LLM model, which generates responses. Subsequently, we categorize the generated responses into fixed-size buckets with a chosen granularity. For instance, using a granularity of 100, responses with token lengths between 0 to 200 are labeled with 0, 200-400 are labeled with 1, and so on. These labels are paired with the training prompts to create a new dataset. Lastly, we partition the new dataset into a training section and an evaluation section and then proceed to train and evaluate the predict model using this dataset.

The length range granularity plays a crucial role. If set to one, we fall back to predicting an exact number of tokens, which is not practical. If set to target model's context window size (e.g., 2K), we fall back to no prediction at all and could run into interferences reported in §2.2.1. Intuitively, a smaller granularity means more accurate resource and performance estimation but lower accuracy in practice. A larger granularity means higher accuracy but essentially makes scheduling harder. Regardless of granularity, it's easy to calculate resource usage's upper and lower bound but not performance. In this work, we can predict a granularity of 200 tokens with 74.9% accuracy. Since improving prediction accuracy is not the focus of this work, we leave it for future work.

Discussions. We run the length predictor at each prefill instance, hence prefill instances can make well-informed decisions on which decode instances should have enough resources to run certain decoding requests. Nevertheless, we identify two alternative designs. The first design is to run the length predictor at each decode instance. As a result, the

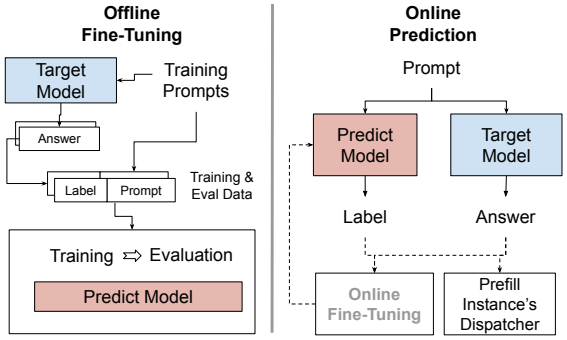


Figure 8: **Predict Model's Fine-tuning and Prediction Flow.** The target model is the one that we want to predict its decoding behavior. The predict model is the one we train. This work does not explore online fine-tuning.

prefill instance can only schedule requests based on the load of decoding instances. However, this design cannot avoid interference cases we measured in §2.2.3. Indeed, one could migrate interference requests among decoding instances at runtime based on predicted length. This would be an overly complex solution. The second design is to run the length predictor at the global scheduler before dispatching requests to refill instances. This design could make the global scheduler a bottleneck. We believe our current design is easier and simpler to reason about and deploy compared to alternatives.

3.3.3 Chunked Prefill

After the prefill scheduler, we concurrently execute the prefill phase of the main LLM alongside the length predictor. We employ fixed-size chunks for the LLM prefill rather than using fixed batch sizes [21].

As demonstrated in §2.2.1, we observe that as the number of tokens in a prefill iteration increases, the accelerator's throughput remains constant, while the latency continues to rise after reaching a certain threshold. We refer to this threshold as `ChunkSize`. Compared to the traditional fixed batch size approach, running prefill in `ChunkSize` allows for the optimal utilization of accelerators without incurring additional latency penalties. The accelerator and the LLM model architecture determine the `ChunkSize`. Models with larger hidden dimensions and accelerators with lower capabilities typically result in a smaller `ChunkSize`. For example, in our test environment, the value is 512 tokens for OPT 13B.

Figure 7 illustrates how chunked prefill works for different scheduler policies. For scheduled requests, we first slice and then merge prompt tokens into fixed-size chunks without altering their order. The final chunk in a batch could be partial, and we will pad it to `ChunkSize` with zeros. Then, we invoke the main LLM model to execute prefill forward one chunk at a time. To record progress, we maintain a simple variable per request that records the last prefilled token position.

The benefits of using chunked prefill and various prefill scheduler policies are substantial. In Figure 16, we compare vanilla vLLM, which uses fixed batch size for prefill, against TetriInfer which uses chunked prefill along with FCFS, SJF, and LJF. Chunked prefill with FCFS lowers average prefill latency by 86.4%. Additionally, we avoid the interference cases measured in §2.2.1 as heavy prefill requests are broken into fixed-chunks and the accelerator is best utilized.

Discussion. (1) An early work, Sarathi [1], has also proposed chunked prefill for the same purpose, where they utilize prefill-decode-mixed chunks. In contrast, our approach involves running prefill-only chunks as we disaggregate LLM's prefill and decode into separate instances. (2) Our length predictor utilizes a small LLM model for prediction and continues using fixed-size batching instead of chunked prefill. This is due to the model's small size, which does not exhibit a clear compute-saturate threshold as seen in larger models.

那这个不是 Sarathi 的一个子集吗?

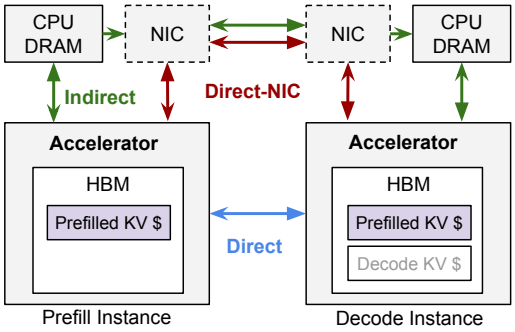
3.3.4 Dispatcher

The prefill instance's final module is the dispatcher, which carries out two essential steps for each prefilled request. First, it runs an inter-decode instance scheduling algorithm to select a decode instance and then transmits the prefilled KV cache to the chosen instance. The dispatcher runs on an event-driven basis, running whenever there are prefilled requests (or chunks). The disaptcher plays a vital role in mitigating decode and decode interferences as measured in §2.2.3.

Once a request's initial chunk is prefilled, the dispatcher invokes a decentralized load-balancing algorithm to select a decode instance with sufficient resources to run this request's decode phase. Our algorithm consists of three steps. First, we categorize decode instances into two sets: α, those with enough resources to execute the chosen request, and β, those without. Recall that the prefill instance has all decode instances' load information broadcasted from the cluster monitor (§3.2). With the predicted length range (§3.3.2), finding decode instances with adequate resources for executing this request's decode phase is easy. Second, we use the power-of-two [25] algorithm to choose two instances from the α set randomly. Lastly, from the two instances, we choose the one that would encounter the least interference if the prefilled request is sent to it. Based on Figure 5, our goal is to establish the lowest average ratio of heavy decode:light decode, which means we need to spread heavy decode requests evenly. Figure 19 proves our algorithm is effective, achieving the lowest total decoding time compared to other policies.

Once a decision is made, the dispatcher sends this request's metadata and prefilled KV cache to the selected decode instance. Crucially, there are two key design considerations for transferring KV Cache: (1) transfer granularity and (2) network stack for transferring the cache.

We begin by discussing granularity. Due to our use of chunked prefill, the prefilled KV cache is created in chunks



Data Link	Two-sided Software	One-sided Software
Direct	Collective Libraries	Low-level APIs
Direct-NIC	IBV Verbs	IBV Verbs
Indirect	TCP/IBV Verbs	N/A

Figure 9: **Data Links and Network Stacks.** The *Direct* link provides bandwidth in the hundreds of GBs (e.g, NVLink 900GBps). The *Direct-NIC* and *Indirect* link offer bandwidth in the hundreds of Gbs (e.g, ConnectX-6 200Gbps). The 1-sided stack means the sender accelerator can transmit data to the receiver accelerator without involving the receiver's CPU.

(§3.3.3). As a result, we have the option to transfer the KV cache either at a *chunk-level*, sending each chunk's KV cache as it is generated, or at a *request-level*, sending the aggregated KV cache for a request until all of its chunks are prefilled. Utilizing chunk-level transfer enables us to parallelize chunked prefill and KV cache transfer, while request-level transfer allows us to minimize the number of network transfers by sending larger data. A concurrent work [32] has proposed layer-wise KV cache transfer, which aligns with our chunk-level approach. Combining their layer-wise approach with our chunk-level transfer could further optimize compute and network parallelization. In this work, we only implement request-level transfer for simplicity and leave the chunk-level transfer to future work.

We now delve into the network stack. Once the main LLM completes its prefill phase, the prefilled KV cache is generated at the accelerator's memory (e.g., GPU's HBM). Our goal is to transmit this cache to the selected decode instance' accelerator memory, regardless of the hardware platforms on which our system is deployed. This is challenging as multiple physical data links exist between the prefill and decode instances, each requiring different software stacks. We classify existing physical data links into three types, as shown in Figure 9. The first is called **Direct**, where accelerators have a directly connected high-speed link such as NVLink [40] or HCCS [13]. We can use low-level memory copy primitives [14, 26] or collective libraries [28] to transmit data over these links. The second is called **Direct-NIC**, in which accelerators communicate via their companion NICs. We can use custom-built libraries [27] to transmit data over PCIe *and* Ethernet (or Infiniband). The third is called **Indirect**, where there is no direct link, and the accelerators must bounce data via their companion CPU DRAM, incurring extra memory copies. In Figure 9, we also categorize network stacks that utilize aforementioned data links into **one-sided** and **two-sided**, similar to RDMA's classification. Accelerators like GPU or NPU can do one-sided memory access as they have low-level primitives such as direct memory copies between devices [14, 26].

To navigate the complicated physical data links and ensure that TetriInfer can always use the most performant link once deployed, we design a unified network transfer abstraction to utilize the different network stack options listed in Figure 9. The stack exposes APIs such as send, receive, read, write, etc. Our dispatcher calls these APIs to transmit the KV cache to remote decode instances.

Discussion. We identify two unexplored research questions. The first question pertains to whether it is beneficial to simultaneously utilize multiple data links for transmitting the KV cache. While this approach could enhance performance, it may also introduce complex control logic. The second question involves the sender accelerator accessing the memory of the receiver accelerator without involving the receiver's CPU. This scenario raises typical challenges associated with building large-scale RDMA-based memory systems [10, 12]. Unfortunately, we cannot explore either of these ideas in this work due to limited access to high-end hardware.

3.4 Decode Instance

The decode instance runs the decoding phase of an inference request. As shown in Figure 6, it includes a receiver module, which is part of the unified network transfer module, a local scheduler, and an LLM for decoding. The processing steps are straightforward. The receiver module accepts requests transmitted from remote prefill instances and waits for prefilled KV caches to be received before adding them to the local scheduler's queue. The scheduler uses continuous batching to group dynamic-sized batches and invokes the LLM for decoding in an auto-regressive fashion. We implement TetriInfer based on vLLM [21] (see §4). Hence, it manages the KV cache in pages rather than reserved for the maximum context length [29, 45].

With the predicted length information sent from the prefill instance, we propose two *working-set-aware* scheduling policies in addition to vLLM's vanilla one. vLLM's existing policy schedule requests in a **greedy** fashion. As long as the accelerator has spare memory, it will add requests to the current iteration. However, it may run out of memory in future iterations and cause thrashing. Fundamentally, it is oblivious to the working set size.

To address this limitation, we propose **reserve-static** and **reserve-dynamic** policies, both aim to prevent triggering swaps. Under the reserve-static policy, a request is scheduled

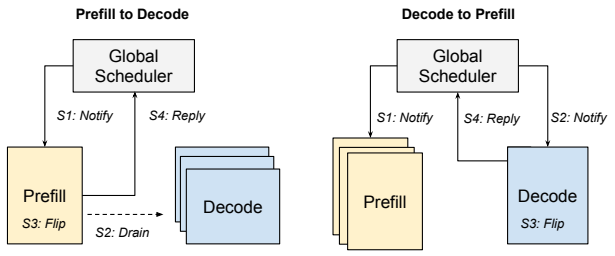


Figure 10: **Prefill and Decode Instance Flip.**

only if its predicted memory usage is smaller than the available accelerator memory for the current iteration. In contrast, the reserve-dynamic policy takes a more proactive approach by considering the predicted number of remaining tokens. Specifically, a new request is added to the scheduled batch only if there is still spare memory when the shortest remaining job in the batch finishes. This approach effectively mitigates memory thrashing while maximizing the advantages of paging. Our tests in Figure 18 suggest that with our current prediction accuracy, these two policies are on par with vLLM's greedy policy. When the prediction accuracy increases, these two policies can lower the average JCT by roughly 10%.

3.5 Instance Flip

Prefill 和 decode 之间的动态转换，因为 prefill 和 decode 的需求动态，不能静态保有固定比例的 prefill/decode 实例

可以支持不同的策略，flip 开销很低

TetriInfer scales out by allocating more hardware resources. Additionally, TetriInfer can also dynamically adjust the number of prefill and decode instances within fixed hardware resources. This is crucial as LLM inference workloads have huge variations regarding prefill and decode needs (see Figure 1), and we cannot statically provision the ratio of prefill and decode instances in advance. Below, we will describe our policy and mechanism to flip a prefill instance to become a decode instance and vice versa.

Policy. As we described in §3.2, the centralized control plane oversees all instances and has the latest load information. We design a transition watcher module that regularly checks load and decides whether certain instances should be flipped. Various policies can be plugged in, such as flipping an instance if its load has been under 10% for the past minute.

Mechainsm. Once an instance is selected, we will go through the steps depicted in Figure 10. To flip a prefill instance, the global scheduler stops forwarding requests and then sends a flip request to it. The prefill instance will wait until all queued requests are drained. Then, we flip the instance. Flipping a decode instance is slightly more complex. The global scheduler notifies all prefill instances to stop forwarding requests to the selected decode instance and notifies the decode instance to complete flipping. Note that the actual flipping is fast and simple. It involves changing an internal variable without restarting the process or reloading models. In our current implementation, both instance flips take roughly 5 to 7 ms, excluding the dynamic draining time.

4 Implementation

We implement TetriInfer's centralized control plane from scratch in Python. We adopt prefill and decode instances based on vLLM [21]. Most of our core modules are implemented in Python, except for the unified network stack, which utilizes C++ to interface with low-level APIs and IB Verbs for network transfer. Additionally, we implement a shared-memory-based communication mechanism that enables fast command transfer across Python and C++ languages. The fine-tuning part uses Trainer APIs offered by HuggingFace Transformer [16].

A prefill or decode instance is a single deployable unit consisting of two processes when deployed. For prefill, it has a Python process that runs the scheduler, length predictor, and the main LLM, as well as a C++ process that runs the dispatcher and the network stack. For decode, it has a Python process for running the scheduler and the main LLM, along with a C++ process that runs the network stack.

Due to limited high-end hardware availability, our current implementation only supports the *Indirect* type using sockets (see Figure 9). In order to evaluate TetriInfer's performance across different hardware configurations, we have implemented a mock mechanism to emulate varying network bandwidth. This mechanism works as follows: for a given set of requests, we initially run their prefill phase offline to obtain their prefilled KV cache. Before testing, we load these prefilled KV caches into the decode instance's local memory. When testing starts, the prefill instance transmits only the request metadata to the decode instance, excluding the actual prefilled KV cache. Subsequently, the decode instance calculates the latency of the KV cache transfer and waits accordingly. This latency is calculated given a specific model architecture and the hardware bandwidth we aim to emulate.

5 Evaluation

We evaluate TetriInfer using public dataset [35] and report time-to-first-token (TTFT), job completion time (JCT), and efficiency as in performance per dollar (perf/$).

Our testbed consists of four NVIDIA V100 GPUs, each with 32GB HBM. All GPUs are plugged into a single server with Xeon Gold 5218R CPU and 256GB DRAM. For the large LLM, we run OPT-13B [47]. For the length prediction model, we use OPT-125M. We compare with vLLM [21]. Since we adopted TetriInfer after vLLM, both systems manage KV caches in pages. Unlike TetriInfer, vanilla vLLM tightly couples prefill and decode phases.

5.1 End-to-End Performance

This section compares TetriInfer with vanilla vLLM using end-to-end benchmarks. We emulate TetriInfer atop two hardware setups using the mock mechanism described in §4. The

first is *TS-RoCE*, assuming prefill and decode instances communicate over 200Gbps RoCE (Direct-NIC in Figure 9). The second is called *TS-NVLink*, assuming instances communicate over 300GBps NVLink (Direct in Figure 9). Both setups are adopted from commercial V100-based servers.

For all tests, the prefill instance's scheduler uses the SJF policy as it has the best performance with the `PrefillSchedBatch` set to 16 (see Figure 16). For the inter-decode instance scheduling, we use the decentralized load-balancing algorithm as it outperforms other policies. For the intra-decode instance scheduling, we use the reserve-dynamic policy atop paging as it could outperform vLLM's greedy policy (see Figure 18). We flip an instance once it becomes idle for a minute using mechanisms described in §3.5.

To understand how these systems perform under mixed downstream inference tasks, we run five different types of workloads as presented in Figure 1: Heavy Prefill with Light Decode (*HPLD*), Heavy Prefill with Heavy Decode (*HPHD*), Light Prefill with Heavy Decode (*LPHD*), Light Prefill with Light Decode (*LPLD*), and *Mixed*. Akin to the configuration used in §2.2, prefill requests that have more than 512 prompt tokens are categorized as heavy, and others are light. Decode requests with more than 128 tokens are categorized as heavy as ShareGPT answers' median length is 128. We generate these workloads using samples from the ShareGPT dataset [35], following the distribution illustrated in Figure 1.

For each workload, we compare all systems across three key metrics: *TTFT*, *JCT*, and *resource usage time* (i.e., cost). Comparing TTFT indicates whether TetriInfer's prefill scheduler and chunked prefill are effective. Comparing JCT indicates whether TetriInfer's disaggregated prefill and decode design and two-level decode scheduling are effective. A natural question that centers around our disaggregated design is cost. Intuitively, TetriInfer uses two times the resources compared to vLLM's prefill-decode-coupled setting. Our results suggest otherwise. Two factors contributed: first, TetriInfer's prefill and decode run faster; second, TetriInfer can recycle or flip instances to reduce waste. Below, resource usage time represents the aggregated wall time that the prefill and decode instances use to run a particular workload. For example, the resource usage time is 3 seconds if we run prefill in in 1 second and decode in 2 seconds. For vLLM, it is the total runtime since it couples prefill and decode.

We now present each workload's results.

Light Prefill and Light Decode. Generally, LPLD represents the chat workload. We test LPLD in Figure 11 using 128 requests. When comparing TetriInfer to vLLM, we reduce average TTFT by 44% and average JCT by 40% for both emulated hardware setups. Despite using twice the number of hardware cards, TetriInfer completes tasks almost twice as fast, resulting in resource usage time that is comparable to the vanilla vLLM. Thus, we improve perf/$ by 1.4x.

Light Prefill and Heavy Decode. Generally, LPHD represents the content creation workload. We test LPHD in Figure 12 using 128 requests. Surprisingly, TetriInfer improves average TTFT by 97% despite using short prompts. This is because vLLM's prefill incurs serious interference while running prefill and decode requests in the same batch; in contrast, TetriInfer disaggregates them into separate instances. Additionally, with variable decode batch size over vLLM's fixed batch size during the decode phase, TetriInfer improves average JCT by 47% while using 38% less total hardware resources. Overall, we improve perf/$ by 2.4x.

Heavy Prefill and Light Decode & Heavy Prefill and Heavy Decode. HPLD and HPHD represent summarization or prompt engineering types of workloads. Both have long prompt tokens. This means TetriInfer faces two challenges: (a) large prefilled KV caches and (b) the main LLM may be impacted by the prediction model (roughly 10% as shown in Figure 17). Nevertheless, in Figure 13, we can see that TetriInfer still improves average TTFT and average JCT by 9% and 23%, respectively, but at the cost of 43% increase in resource usage. vLLM outperforms TetriInfer in terms of perf/$ by 14%. As Figure 14 shows, with heavy decode, TetriInfer's TTFT improvement is more pronounced because we disaggregated heavy decode from prefill, akin to Figure 12. We improve the average JCT by 19% at the cost of 7% more resources, improving perf/$ by 1.1x.

Mixed. The last workload is a mix of all the above workloads, randomly sampled from the ShareGPT dataset. This is the case where a cluster is running all kinds of requests. In Figure 15, we run 128 requests, TetriInfer lowers average TTFT, average JCT, and resource usage by 85%, 50%, 21%, respectively, improving perf/$ by 1.9x.

Takeaways. (1) For most LLM inference workloads, TetriInfer improve average TTFT, average JCT, resource use time, and most importantly, perf/$ by a large margin. (2) Disaggregating prefill from decode into two distinct instances significantly improves TTFT and efficiency by minimizing interference, particularly for workloads with heavy decodes such as LPHD and HPHD. (3) TetriInfer's design is not ideal for HPHD workloads as the room for improvement is small, and the overhead we introduce cannot be offset.

5.2 Microbenchmark

5.2.1 Prefill Scheduler

Below, we study the overhead of sorting requests, compare different policies, and study the impact of batch size on performance. Note we run OPT-13B TP=2, `ChunkSize` is set to 512, and vLLM's batch size is set to 16.

Sort. Our scheduler sorts incoming requests based on the length of their input tokens if non-FCFS policies are used. We use Python's native sort API. We find the sorting overhead ranges from 10s to 100s of microseconds, which is negligible compared to millisecond-level or second-level TTFT latency.

Scheduler Policy and Batch Size. In Figure 16, we com-

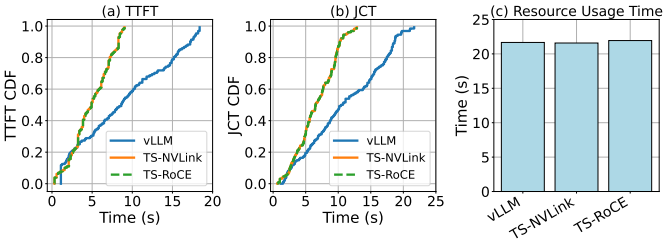


Figure 11: **Light Prefill and Light Decode (LPLD)**

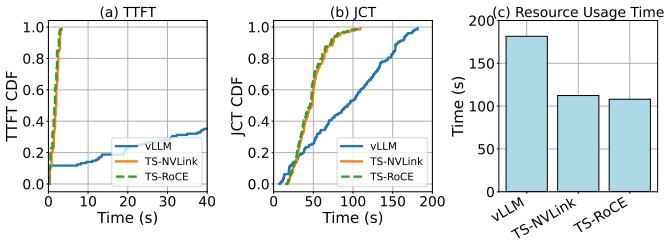


Figure 12: **Light Prefill and Heavy Decode (LPHD)**

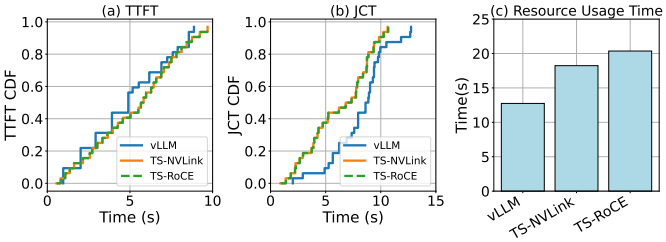


Figure 13: **Heavy Prefill and Light Decode (HPLD)**

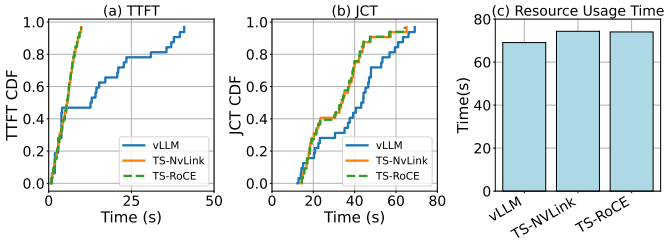


Figure 14: **Heavy Prefill and Heavy Decode (HPHD)**

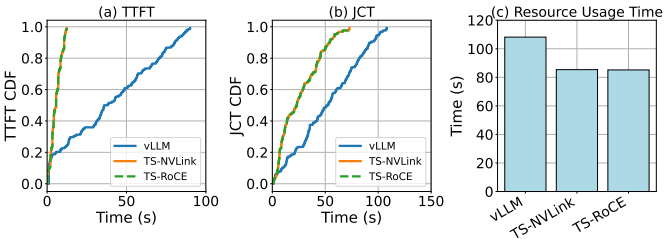


Figure 15: **Mixed Prefill and Decode**

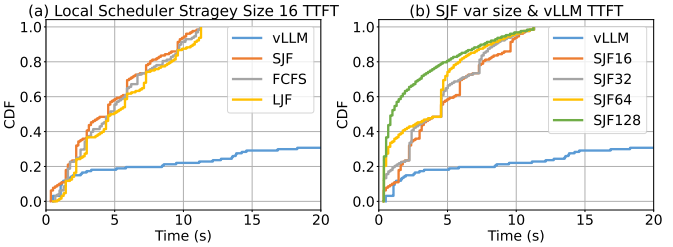


Figure 16: **Scheduler Policies and Chunked Prefill.**

pare TetriInfer which uses chunked prefill along with FCFS, SJF, and LJF, against vanilla vLLM, which uses fixed batch size for prefill. Requests used in this test follow the ShareGPT distribution. In the left part, we set `PrefillSchedBatch` to 16. Compared to vLLM's fixed batch mode, chunked prefill alone with FCFS improves latency by 86.4%. Additionally, the SJF scheduler policy further lowers the average waiting time by 7.8%. The right part examines the impact of adjusting `PrefillSchedBatch`. When we increase the batch size from 16 to 128, SJF's average TTFT decreases by 46.5%. The improvement in TTFT increases with a larger scheduling batch.

5.2.2 Length Predictor

Our length predictor uses the OPT-125M classification model to speculate the decoding behavior of the OPT-13B model (§3.3.2). This section studies the performance of both models and the prediction accuracy.

In Figure 17, we run stress tests among several settings regarding per-iteration latency and throughput. *L-Alone* means running the OPT-13B model alone, using chunked fill with `ChunkSize` set to 512. *P-Alone* means running the OPT-125M prediction model alone. It does not use chunked prefill but uses dynamic batch sizes. It can group multiple requests into a batch. Due to the limitation of [16], we need to pad requests in a batch to the longest one. For example, if we have two requests in a batch, one has 100 tokens, and the other has 500 tokens. Then, we need to pad the first to 500 tokens. This is costly for requests with short prompts. Hence, we set a cutting limit. Requests higher than the limit will run alone. The default is 512 tokens. *L+P512* means OPT-13B model's performance if co-runs with the small OPT-125M in parallel. The suffix number means the max padded size. We can see that the large LLM's prefill latency is roughly ten times of the small LLM's. If we co-run both models and a padding limit of 512, 80% of large LLM's prefill requests remain unchanged compared to when it runs alone. Overall, while co-running with a small LLM, the large LLM's average prefill latency increases by 10%, and throughput drops by 12%. Note that these are stress tests. The impact will be smaller in practice. We believe beefier hardware can further mitigate the drop.

We train our OPT-125M prediction model using 75K training data from ShareGPT. We test the model using three different length range granularities: 100, 200, and 400. The accuracy achieved by our prediction model for these granularities is 58.9%, 74.9%, and 85%, respectively.

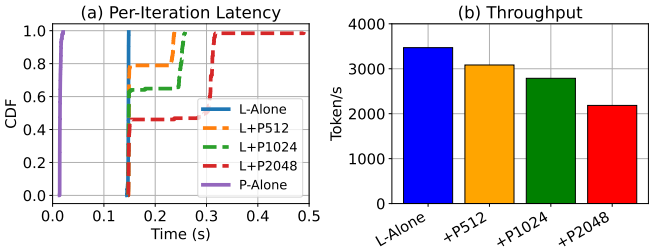


Figure 17: **Running Large LLM with Prediction Model.**

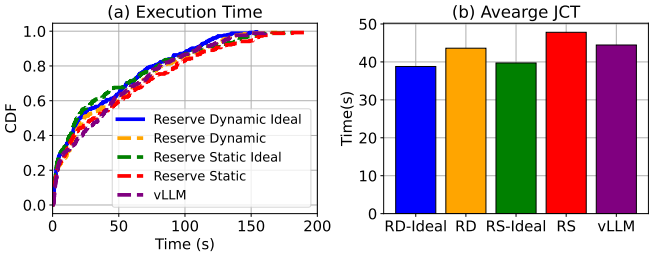


Figure 18: **Intra-Decode Instance Scheduling.**

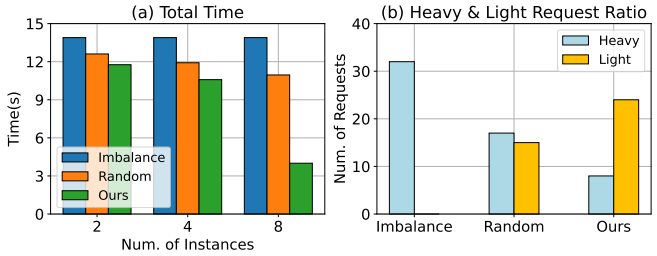


Figure 19: **Inter-Decode Instance Scheduling.**

Work	C. P.	Disagg. P/D	Interference	Dist-Sched.
TetriInfer	✓	✓	✓	✓
Splitwise [32]	×	✓	×	✓
Sarathi [1]	✓	×	×	×
vLLM [21]	×	×	×	×
FastServe [41]	×	×	×	✓

Table 1: **Related work comparison.** (1) C. P.: chunked prefill. (2) Disagg. P/D: disaggregated prefill and decode. (3) Interference: whether the system deals with inference interference. (4) Dist-Sched: distributed scheduling policies.

5.2.3 Decode Scheduling

We now study the scheduling policies related to decode instances. We first compare intra-decode instance scheduler policies (§3.4) and then compare different load balance algorithms for inter-decode instance scheduling (§3.3.4). All tests use OPT-13B with TP=2.

We compare three intra-decode scheduler algorithms, namely vLLM's greedy, TetriInfer's reserve-static (RS), and reserve-dynamic (RD) in Figure 18. We run 256 requests following ShareGPT distribution. Our policies estimate resource usage using the predicted length range's lower end. We compare using the actual accuracy (acc-200 74.9%) and an ideal accuracy of 100%. While using the actual accuracy, reserve-dynamic achieves the same JCT as vLLM's greedy algorithm. When using an ideal prediction accuracy, reserve-dynamic and reserve-static improve average JCT by 12% and 10%, respectively. This is because our policies carefully provision requests based on their memory usage.

We compare three distributed load balance algorithms in Figure 19. Firstly, we present our *decentralized power-of-two* algorithm, designed to distribute requests based on predicted length. The second is *random*, in which the prefill instance randomly chooses a decode instance. The third algorithm, *imbalance*, simulates a worst-case scenario where heavy decode requests are consistently directed to the same decode instances. We run 32 requests per decode instance, spanning the range of 2 to 8 decode instances. Figure 19's left part shows that TetriInfer's decentralized load balancing algorithm is effective, achieving the lowest total decoding time. The right parts show the number of heavy decode and light decode requests in the slowest instance. Cleary, TetriInfer's inter-decode scheduling algorithm evenly balances load across instances, which avoids interferences measured in §2.2.3.

6 Related Work

Table 1 compares TetriInfer with other closely related works. We are among the first to disaggregate prefill and decode in LLM inference, concurrent to Splitwise [32]. Sarathi [1] has proposed chunked prefill to overcome suboptimal prefill processing. They run prefill-decode-mixed chunks. In contrast, TetriInfer runs prefill-only chunks as we observe non-neglible interference between prefill and decode, thus choose to disaggregate prefill from decode. FastServe [41] utilizes a multi-level priority feedback queue to minimize JCT. In contrast, TetriInfer utilizes two-level scheduling for prefill and decode instances. Our policies are working-set-aware, reducing interference and swaps thereby improving JCT and efficiency. Many recent work focus on optimizing batching, caching, and scheduling [2, 23, 30]. Specifically, Orca [45] introduce the iterative-level scheduling. Sheng et.al [36] have proposed a fair scheduler based on the continuous batching mechanism. Many works try to optimize memory usage. For example, using quantization [7, 8, 11, 19, 22, 37, 42, 43] to compress the model weights into lower precision, using paging to reduce fragmentation [21], and low-level algorithm and

kernel optimizations [5, 6, 15, 24, 39, 44, 46]. Those works are orthogonal to our efforts to mitigate interference.

7 Conclusion

We propose TetriInfer, an LLM inference serving system designed to battle interference. Our key insight is to carefully schedule and group inference requests based on their characteristics. It has three key parts. First, it partitions prompts into fixed-size chunks, ensuring the accelerator consistently operates at its computation-saturated limit. Second, it disaggregates prefill and decode instances to avoid interference when mixing them together. Finally, it uses a smart two-level scheduling algorithm to avoid decode scheduling hotspots. Results show that TetriInfer improves time-to-first-token, job completion time, and inference efficiency in terms of performance per dollar by a large margin.

References

[1] 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.

[2] Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, et al. Deepspeed-inference: enabling efficient inference of transformer models at unprecedented scale. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2022.

[3] AWS Bedrock.

https://docs.aws.amazon.com/bedrock/latest/userguide/inference-parameters.html.

[4] 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.

[5] Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691*, 2023.

[6] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, 2022.

[7] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Llm. int8 (): 8-bit matrix multiplication for transformers at scale. *arXiv preprint arXiv:2208.07339*, 2022.

[8] Tim Dettmers, Ruslan Svirschevski, Vage Egiazarian, Denis Kuznedelev, Elias Frantar, Saleh Ashkboos, Alexander Borzunov, Torsten Hoefler, and Dan Alistarh. Spqr: A sparse-quantized representation for near-lossless llm weight compression. *arXiv preprint arXiv:2306.03078*, 2023.

[9] Xin Luna Dong, Seungwhan Moon, Yifan Ethan Xu, Kshitiz Malik, and Zhou Yu. Towards next-generation intelligent assistants leveraging llm techniques. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2023.

[10] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, 2014.

[11] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. Optq: Accurate quantization for generative pre-trained transformers. In *The Eleventh International Conference on Learning Representations*, 2022.

[12] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. Clio: A Hardware-Software Co-Designed Disaggregated Memory System. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022.

[13] HiAscend. Atlas 900 AI Cluster.

https://www.hiascend.com/en/hardware/cluster.

[14] HiAscend. CANN aclrtMemcpy.

https://www.hiascend.com/document/detail/en/canncommercial/601/inferapplicationdev/aclcppdevg/aclcppdevg_03_0081.html.

[15] Ke Hong, Guohao Dai, Jiaming Xu, Qiuli Mao, Xiuhong Li, Jun Liu, Kangdi Chen, Hanyu Dong, and Yu Wang. Flashdecoding++: Faster large language model inference on gpus. *arXiv preprint arXiv:2311.01282*, 2023.

[16] HugginFace.

https://huggingface.co/docs/transformers/model_doc/opt#transformers.OPTForSequenceClassification.

[17] Hugging Face.

https://huggingface.co/datasets/ZhongshengWang/Alpaca-pubmed-summarization.

[18] Hugging Face.

https://huggingface.co/datasets/lancexiao/write_doc_sft_v1.

[19] Berivan Isik, Hermann Kumbong, Wanyi Ning, Xiaozhe Yao, Sanmi Koyejo, and Ce Zhang. Gpt-zip: Deep compression of finetuned large language models. In *Workshop on Efficient Systems for Foundation Models@ ICML2023*, 2023.

[20] A Jo. The promise and peril of generative ai. *Nature*, 2023.

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

[22] Zhuohan Li, Eric Wallace, Sheng Shen, Kevin Lin, Kurt Keutzer, Dan Klein, and Joey Gonzalez. Train big, then compress: Rethinking model size for efficient training and inference of transformers. In *International Conference on machine learning*, 2020.

[23] 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 preprint arXiv:2302.11665*, 2023.

[24] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. Rammer: Enabling holistic deep learning compiler optimizations with {rTasks}. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020.

[25] NGINX.

https://www.nginx.com/blog/nginx-power-of-two-choices-load-balancing-algorithm/.

[26] NVIDIA. CUDA Runtime API Memory Management.

https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html.

[27] NVIDIA. GPU Direct.

https://developer.nvidia.com/gpudirect.

[28] NVIDIA. NCCL.

https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/overview.html.

[29] NVIDIA, FasterTransformer.

https://github.com/NVIDIA/FasterTransformer.

[30] NVIDIA, Triton Inference Server.

https://developer.nvidia.com/.

[31] Charles Packer, Vivian Fang, Shishir G Patil, Kevin Lin, Sarah Wooders, and Joseph E Gonzalez. Memgpt: Towards llms as operating systems. *arXiv preprint arXiv:2310.08560*, 2023.

[32] Pratyush Patel, Esha Choukse, Chaojie Zhang, Íñigo Goiri, Aashaka Shah, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative llm inference using phase splitting. *arXiv preprint arXiv:2311.18677*, 2023.

[33] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference. *Proceedings of Machine Learning and Systems*, 2023.

[34] Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever. Zero-shot text-to-image generation. In *International Conference on Machine Learning*, 2021.

[35] Sharegpt teams.

https://sharegpt.com/.

[36] Ying Sheng, Shiyi Cao, Dacheng Li, Banghua Zhu, Zhuohan Li, Danyang Zhuo, Joseph E Gonzalez, and Ion Stoica. Fairness in serving large language models. *arXiv preprint arXiv:2401.00588*, 2023.

[37] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. Flexgen: High-throughput generative inference of large language models with a single gpu. In *International Conference on Machine Learning*, 2023.

[38] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models, 2023. *URL https://arxiv. org/abs/2307.09288*, 2023.

[39] Xiaohui Wang, Ying Xiong, Yang Wei, Mingxuan Wang, and Lei Li. Lightseq: A high performance inference library for transformers. *arXiv preprint arXiv:2010.13887*, 2020.

[40] Wikipedia. NVLink.

https://en.wikipedia.org/wiki/NVLink.

[41] 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.

[42] Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. Smoothquant: Accurate and efficient post-training quantization for large language models. In *International Conference on Machine Learning*, 2023.

[43] Zhewei Yao, Cheng Li, Xiaoxia Wu, Stephen Youn, and Yuxiong He. A comprehensive study on post-training quantization for large language models. *arXiv preprint arXiv:2303.08302*, 2023.

[44] Zhewei Yao, Reza Yazdani Aminabadi, Minjia Zhang, Xiaoxia Wu, Conglong Li, and Yuxiong He. Zeroquant: Efficient and affordable post-training quantization for large-scale transformers. *Advances in Neural Information Processing Systems*, 2022.

[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 *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022.

[46] Yujia Zhai, Chengquan Jiang, Leyuan Wang, Xiaoying Jia, Shang Zhang, Zizhong Chen, Xin Liu, and Yibo Zhu. Bytetransformer: A high-performance transformer boosted for variable-length inputs. In *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2023.

[47] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022.

[48] Zangwei Zheng, Xiaozhe Ren, Fuzhao Xue, Yang Luo, Xin Jiang, and Yang You. Response length perception and sequence scheduling: An llm-empowered llm inference pipeline. *arXiv preprint arXiv:2305.13144*, 2023.