HydraInfer: Hybrid Disaggregated Scheduling for Multimodal Large Language Model Serving

Xianzhe Dong
University of Science and Technology of China

Tongxuan Liu
University of Science and Technology of China

Yuting Zeng
University of Science and Technology of China

Liangyu Liu
University of Science and Technology of China

Yang Liu
University of Science and Technology of China

Siyu Wu
Beihang University

Yu Wu
University of Science and Technology of China

Hailong Yang
Beihang University

Ke Zhang
JD.com

Jing Li
University of Science and Technology of China

Abstract

Multimodal Large Language Models (MLLMs) have been rapidly advancing, enabling cross-modal understanding and generation, and propelling artificial intelligence towards artificial general intelligence. However, existing MLLM inference systems are typically designed based on the architecture of language models, integrating image processing and language processing as a single scheduling unit. This design struggles to accommodate the heterogeneous demands of different stages in terms of computational resources, memory access patterns, and service-level objectives (SLOs), leading to low resource utilization and high request latency, ultimately failing to meet the service requirements of diverse inference scenarios.

To address these challenges, we propose HydraInfer, an efficient MLLM inference system that adopts a Hybrid Encode-Prefill-Decode (EPD) Disaggregation architecture. By scheduling the three stages — encode, prefill, and decode — onto separate heterogeneous inference instances, the system flexibly reallocates resources across stages, significantly reducing idle computation, alleviating resource bottlenecks, and improving overall system throughput and scalability. In addition, HydraInfer supports a stage-level batching strategy that enhances load balancing, enables parallel execution of visual and language models, and further optimizes inference performance. Experiments under real multimodal inference workloads demonstrate that HydraInfer can achieve up to 4× higher inference throughput compared to state-of-the-art systems (e.g., vLLM) on a single-node 8×H800 GPU cluster, while meeting the 90th percentile request SLO.

Keywords: MLLM Inference System, Request Scheduling, Encode-Prefill-Decode Disaggregation

1 Introduction

In recent years, Multimodal Large Language Models (MLLMs) [3, 4, 16, 20, 22, 24, 33, 35, 42] have demonstrated impressive capabilities across various domains, such as image understanding and visual question answering. However, during inference, images are converted into hundreds or even thousands of tokens [5], requiring significantly more GPU resources compared to large language models [6]. As a result, reducing the inference cost of MLLMs has become crucial.

The inference process of MLLMs typically consists of three stages: the image encode stage extracts visual features; the prefill stage feeds visual features and the text prompt into the language model to generate the first output token; and the decode stage iteratively generates subsequent tokens based on the cache. The Time-To-First-Token (TTFT) is mainly affected by the image encode and prefill stages, reflecting the latency from when a request arrives to when the first token is generated. The Time-Per-Output-Token (TPOT) measures the time intervals between successive output tokens during the decoding stage. Goodput represents the maximum throughput the system can sustain while meeting the TTFT and TPOT service level objectives (SLOs).

Current inference engines such as vLLM [18], TGI (Text Generation Inference) [11], SGLang [40], and DistServe [41] are designed primarily for language models and face several challenges in multimodal inference tasks.

1) **Insufficient parallelism**. During multimodal inference, the computational tasks of different requests exhibit significant parallelism. For example, the vision model inference of one batch of requests can be executed in parallel with the language model inference of another batch, thereby improving overall resource utilization. However, most existing engines adopt a serial execution strategy, failing to exploit

this inter-request parallelism, resulting in wasted computational resources and becoming a bottleneck for throughput.

2) **Suboptimal scheduling granularity**. The decode stage in LLM inference is typically memory-intensive and benefits significantly from batching to improve throughput [38]. On the other side, the prefill stage is more compute-intensive and is better suited for chunked prefill and co-batching with the decode stage to balance throughput and latency [1]. We further observe that the compute and memory characteristics of the encode stage lie between those of prefill and decode, indicating that it can benefit from batching too (as shown in Figure 6). However, current engines often combine the encode and prefill stages, which prevents stage-specific batching. Furthermore, the combined scheduling strategy does not support chunked prefill [2], and the coarse granularity of the prefill stage makes it difficult to control execution time precisely, leading to potential violations of TPOT SLOs [1].

3) **Difficulty in disaggregation method selection**. DistServe [41] proposes a Prefill-Decode disaggregated architecture, which decouples the prefill and decode stages to reduce resource interference and better satisfy TTFT and TPOT requirements. This architecture allows flexible configuration of GPU resources across stages and achieves efficient utilization under static workloads.

However, in multimodal inference, both the encode and prefill stages contribute to TTFT, while prefill and decode jointly affect TPOT. Under varying workload distributions, selecting the optimal disaggregation method remains challenging. For instance, disaggregation methods such as E+P+D (encode, prefill, decode all separated), EP+D (encode, prefill co-located, separated with decode), and ED+P (encode, decode co-located, separated with prefill) have not been systematically evaluated across diverse inference scenarios, highlighting the need for in-depth analysis to inform efficient system design.

To address the limitations of existing inference systems in exploiting vision and language model parallelism, we propose a multi-stream parallel execution mechanism that executes tasks from different requests concurrently. Specifically, we adopt a two-stream parallel model in which the vision model and the language model are assigned to separate execution streams. The vision stream is dedicated to executing computationally intensive image encoding tasks, while the language stream handles prefill and decode operations for language generation. By isolating the workloads into distinct streams, our system enables concurrent execution of heterogeneous stages from different requests.

Considering the diverse compute and memory access characteristics of different stages in multimodal inference, we propose a Stage-Level Schedule strategy, which decomposes each request within an instance into three stages: encode, prefill, and decode, and applies stage-specific scheduling and batching optimizations. This strategy avoids the performance bottlenecks caused by traditional coarse-grained stage merging, enabling finer control over execution timing and batching.

To tackle the challenge of selecting disaggregation method in multimodal inference, we propose a Hybrid Encode-Prefill-Decode disaggregation architecture, an innovative multi-instance collaborative inference framework. In this architecture, each instance executes only a subset of the three stages, with other stages being handled by transferring requests to different instances, thus avoiding resource waste and interference. The system automatically selects the optimal disaggregation method and instance numbers based on request profiles, achieving a dynamic balance between throughput and latency.

We implement a system named HydraInfer from the ground up and conduct comprehensive evaluations across various MLLMs, covering representative applications such as image captioning, visual question answering, and visual understanding. We compare HydraInfer against frameworks such as vLLM and SGLang. Experimental results show that under different SLO constraints, HydraInfer outperforms state-of-the-art inference systems, achieving up to 2×, 1.5×, 2×, 2×, and 4× throughput improvements on MME [12], POPE [21], TextCaps [30], TextVQA [31], and VizWiz [13] datasets, respectively. Our main contributions include:

- We identify and analyze key challenges in existing inference systems for MLLMs, particularly in parallelism, scheduling granularity, and disaggregation method selection.
- We design HydraInfer inference system, which adopts a Hybrid Encode-Prefill-Decode disaggregation architecture, and automatically selects the optimal disaggregation method based on workload and SLO profiles for flexible resource allocation.
- We propose a stage-level scheduling strategy, which applies batching optimizations at different inference stages and incorporates a multi-stream parallel execution model to improve system throughput and resource efficiency.
- Through extensive evaluations of HydraInfer across multiple models and hardware platforms, we demonstrate its strong generality and show that our system can significantly enhance service capacity across diverse scenarios.

2 Background

In this section, we introduce the principles of autoregressive inference of MLLM. We then discuss the widely adopted batching methods used by current inference engines to improve throughput, and finally, we present some important performance evaluation metrics for inference engines.

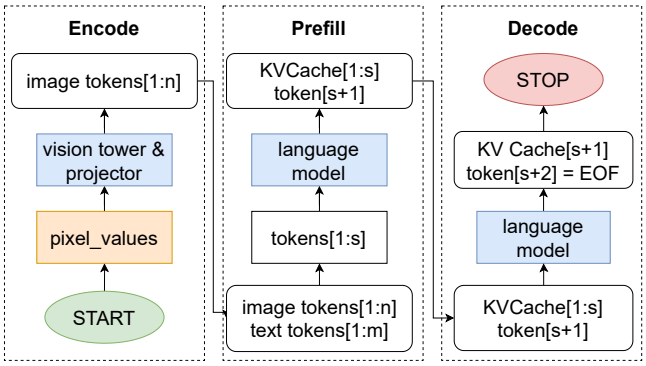

Figure 1. Vision-Language autoregressive inference process.

2.1 MLLM Autoregressive Inference

As shown in Figure 1, the inference process of a transformer-based [34] vision-language model can be divided into three distinct stages [37]. The first stage is the encode stage. Given an image pixel values input for a request, the model's vision tower and projector process the image to obtain the image tokens $A[a_1, a_2, \ldots, a_n]$. The second stage is the prefill phase. The request prompt is encoded into text tokens $B[b_1, b_2, \ldots, b_m]$, and then the image tokens A and text tokens B are concatenated to form the final input $X[x_1, x_2, \ldots, x_s]$ where $s = n + m$. This input X is fed into the large language model to generate a token x_{s+1} and a series of *kvcache* used for the decode phase. The third stage is the decode phase. In this phase, tokens are generated iteratively. The token x_{s+1} and the $kvcache[1:s]$ generate the token x_{s+2}. This process iterates, generating one token at a time, until the generated token is <eos> or the maximum token limit is reached. Due to the data dependencies in the decode phase, this stage is executed sequentially.

2.2 Batch Processing

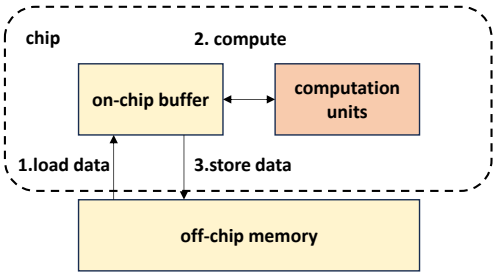

Figure 2. The process of executing operators on hardware devices: first, data is loaded into the on-chip cache, then computation is performed, and finally, the cache is written back to the memory. The first and third steps use memory access units, while the second step uses computation units.

As shown in Figure 2, the execution of a neural network layer on hardware involves transferring data from memory (DDR or HBM) to the on-chip cache, followed by computation in the on-chip processing unit, and finally writing the results back to memory. Therefore, performance evaluation needs to consider both memory access and computational unit capabilities. If a layer involves heavy computation but minimal memory access, it is considered compute-intensive, which can lead to idle memory access. Conversely, when a layer requires extensive memory access but minimal computation, it is considered memory-intensive. In this case, the computation unit remains underutilized [39]. The inference engine is responsible for concurrently handling requests from multiple users. The simplest way to handle requests is sequentially, but this results in significant GPU resource wastage because each iteration requires reading almost all model parameters, causing heavy memory access. Thus, requests are typically scheduled and batched to increase computation per batch, amortizing memory access and thereby improving throughput. The generated text lengths of different requests are inconsistent and can vary greatly. Modern inference systems usually adopt continuous batching [38], which allows dynamic addition of new requests and removal of finished ones.

2.3 Inference Engine Performance Metrics

End-to-end latency measures the time between a request's arrival and its completion. **TTFT (Time to First Token)** measures the time from when a request arrives at the system to when the first output token is produced. It reflects how quickly the system responds to the request. **TPOT (Time Per Output Token)** measures the interval between the generation of consecutive output tokens in a request. TPOT reflects the overall smoothness of the response. **Throughput** refers to the number of requests processed per second or the number of tokens output per second by the inference engine. **Capacity** is defined as the maximum request load (queries per second) the system can handle while meeting specific latency targets. A higher capacity is desirable as it lowers service costs. Different application scenarios have different performance requirements: offline inference emphasizes throughput, while online inference must consider all of the above metrics. Requests are associated with **TTFT SLOs** and **TPOT SLOs**. If a request's TTFT is less than its TTFT SLO and 90% of its TPOT values are below the TPOT SLO, the request is considered to have met the SLO (Service Level Objective). **SLO attainment** is defined as the percentage of requests that meet their SLOs out of all requests. **Goodput** is defined as the maximum request rate at which the SLO attainment reaches at least 90%.

3 Motivation

This section first analyzes the benefits of parallel execution between the vision model and the language model; then discusses the cost characteristics of different inference stages

and the limitations of scheduling granularity in current systems; finally, it discusses how request load and service level objectives (SLOs) influence the effectiveness of model disaggregation methods.

3.1 Parallel Execution of the Vision Model and the Language Model

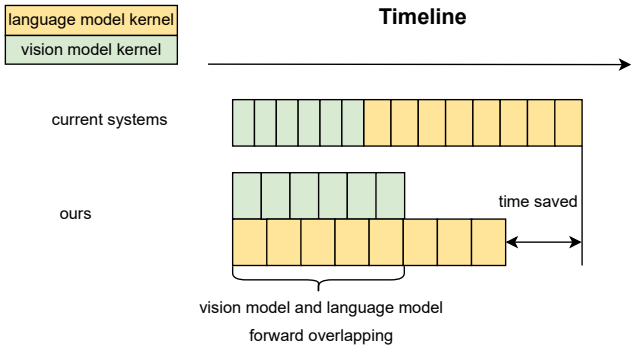

Figure 3. Parallel execution of the vision model and the language model.

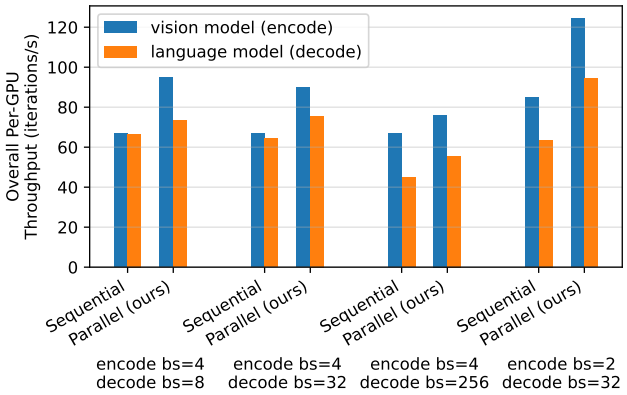

Figure 4. Overall per-GPU throughput when executing LLaVA-1.5-7B's vision model (encode) and language model (decode) sequential or parallel under different batch size. The KV cache length of decode is 1024.

The prefill stage is compute-intensive, the decode stage is memory-intensive[1]. We theoretically analyze the FLOPs and memory access patterns of key operations in MLLMs, as shown in Table 1 and Table 2. The analysis reveals that the encode stage lies between the prefill and decode stages in terms of computational and memory characteristics. Batching compute-bound and memory-bound operations together can help maximize the utilization of both the computational units and memory bandwidth. There are two approaches to achieve this: one is to parallelize the prefill and decode stages, and the other is to parallelize the encode and decode stages.

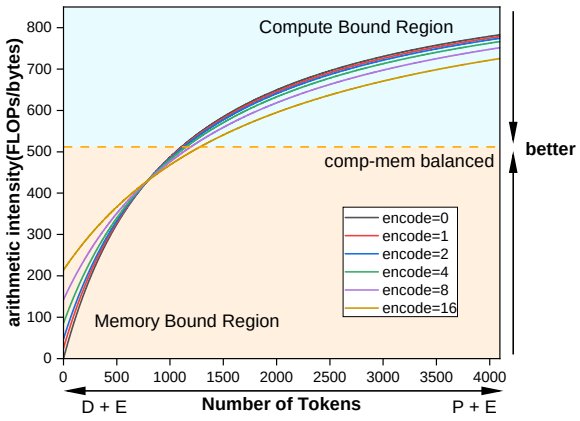

Figure 5. Arithmetic intensity trend for LLaMA1.5-7B linear operations with different number of images and tokens.

The parallel batching of prefill and decode has already been supported in many large language model (LLM) inference systems [9, 10, 36]. This is implemented at the operator level by flattening all the prefill and decode inputs into a single tensor, passing in offset metadata to inform the kernel of each request's location within the flattened input, and executing the model so that each operator processes all requests simultaneously. Currently, in most inference engines, the vision model and language model are executed sequentially, without support for concurrent execution. Inspired by [43], we propose using multiple CUDA streams to achieve kernel-level parallelism between the vision model and the language model. An illustration of this approach is shown in Figure 3. When using sequential execution, low compute utilization during the decode stage results in underused GPU resources; instead, parallel multi-stream execution can increase both the throughput per-GPU of image encode and decode. As shown in Figure 4, we evaluate the per-GPU throughput of both methods with encode and decode stages on LLaVA-1.5-7B. For *Parallel* method, the two stages are executed in two CUDA streams simultaneously. While for *Sequential*, the stages executed round-robin and both take 50% execution time share, which is also equivalent to the per-GPU throughput when the two models are being disaggregated deployed on two GPUs.

By analyzing the arithmetic intensity of the prefill and iterative decode stages, we observe that most time is spent on memory-bound linear operators. The total execution time of an operator can be approximated as $T = \max(T_{\text{comp}}, T_{\text{mem}})$ where T_{comp} and T_{mem} represent the time spent on computation and memory access, respectively. Memory-bound operators yield low model FLOP utilization, while compute-bound operators yield low memory bandwidth utilization. When

Notation	Description
S	The length of prompt
B	The number of batched requests
T	The number of tokens per image
L	The number of model layers
H	Input dimension of the hidden layer
M	The number of attention heads

Table 1. Notations used in table 2, different L, H, M should been applied according to different model (vision or language).

Operation	E/P/D	FLOPS	Memory Access
QKVO Proj.	encode	$8BTH^2$	$8BTH + 4H^2$
	prefill	$8BSH^2$	$8BSH + 4H^2$
	decode	$8BH$	$8BH + 4H^2$
FFN	encode	$16BTH^2$	$4BTH + 8H^2$
	prefill	$16BSH^2$	$4BSH + 8H^2$
	decode	$16BH^2$	$4BH + 8H^2$
Attention	encode	$4BT^2H$	$4BTH + 2BT^2M$
	prefill	$4BS^2H$	$4BSH + 2BS^2M$
	decode	$4BSH$	$4BSM + 2BH(S+1)$

Table 2. Arithmetic intensity and Memory Acceess of primary operations in MLLMs.

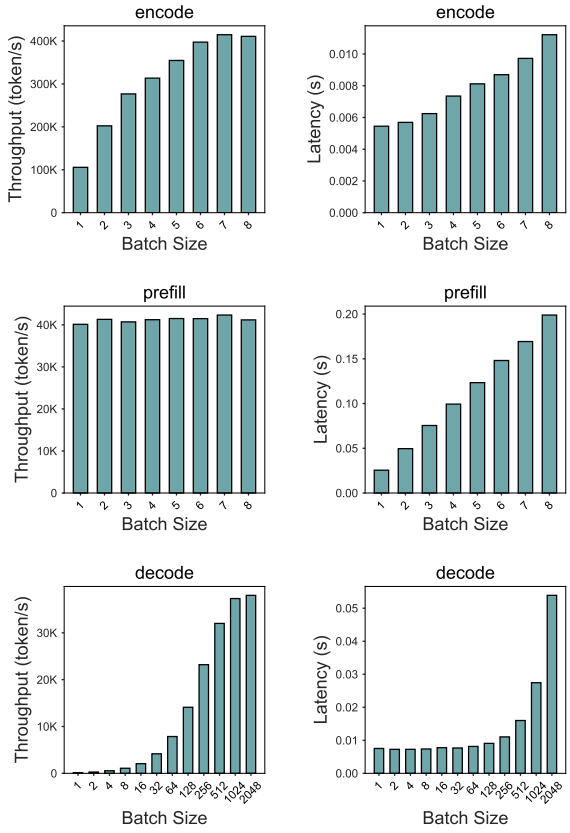

Figure 6. The performance of the LLaVA-1.5-7B model on a single H800 GPU under varying batch sizes. For both the prefill and decode stages, we use a prompt length of 1024 tokens. For the encode stage, the input image resolution is 336×336, corresponding to 576 visual tokens per image.

$T_{comp} = T_{mem}$, both computational and memory bandwidth utilization are maximized [39].

By the pervious analysis of the compute and memory characteristics of the three stages in MLLM inference (Table 1 and 2), we plot the theoretical arithmetic intensity across different batch sizes in Figure 5. Each curve in Figure 5 represents the relationship between arithmetic intensity and token count under varying image batch sizes. When the token count is small (i.e., during the decode phase), the operation is memory-bound. In this region, increasing the number of images in the batch raises arithmetic intensity. When the token count is large (i.e., during prefill), the workload is compute-bound, and batching encode with prefill reduces arithmetic intensity. Our system leverages joint batching of the vision model and language model to balance compute and memory bandwidth utilization more effectively.

Takeaway-1: The vision model and language model kernels are independent and well-suited for parallel execution. By parallelizing them, we can significantly improve GPU compute and memory resource utilization, leading to higher throughput.

3.2 Schedule Policy

The serving system must handle concurrent requests from multiple users. Naively processing these requests sequentially would lead to severe underutilization of GPU compute resources. To improve GPU utilization, large language model (LLM) serving systems employ batching techniques to process multiple requests simultaneously. A larger batch size helps amortize the cost of accessing model parameters across multiple requests [1, 38]. Figure 6 shows how throughput changes with batch size. We observe that the encode stage reaches saturation at a batch size of around 6. The prefill stage achieves saturation even with a single request. The decode stage shows roughly linear throughput improvement with increasing batch size, saturating at around 512. Once the batch size exceeds the saturation point for each stage, increasing it further does not lead to additional throughput gains, while latency increases linearly.

Takeaway-2: The three stages in MLLMs inference — encode, prefill, and decode — exhibit distinct performance characteristics. Batching significantly boosts throughput in the

decode stage, moderately improves throughput in the encode stage, and has minimal impact on the prefill stage. Performing scheduling at the granularity of request stages can effectively enhance overall system throughput.

Batch processing improves system throughput, but at the cost of high TPOT latency, a phenomenon also known as generation stall [1]. The generation stall problem still exists in multimodal models. Figure 7 compares different scheduling strategies in the current system, with instances showing four requests A, B, C, and D on a timeline (from left to right). Requests A and B are in the decode stage, while requests C and D enter the system. vLLM adopts a FCFS (First-Come-First-Serve) prefill-first scheduling strategy, which increases throughput by maximizing the batch size of the subsequent decode stage. However, the prefill-first strategy causes requests A and B, which are decoding, to stop generating, leading to very high tail token latency, because the image encode and prefill of requests C and D are much slower compared to decode. This phenomenon is called generation stall. Another scheduling strategy, such as the decode-first strategy used by Sarathi Serve, aims to significantly reduce TPOT, at the cost of slightly lowering throughput, thus improving smoothness. In Sarathi Serve, each batch contains both prefill-stage requests and decode-stage requests, so the decoding request A and B continue to generate as new requests are added, maintaining a lower TPOT. To minimize the impact of newly added prefill requests in the batch, chunked prefill technology is usually used to split the prompt into multiple segments, batching only part of it at a time. Although this batching strategy used in Sarathi Serve performs well in large language model inference systems, it is still suboptimal for multimodal model inference. This is because it relies on the number of tokens to estimate the workload, aiming to ensure roughly consistent latency across iterations.

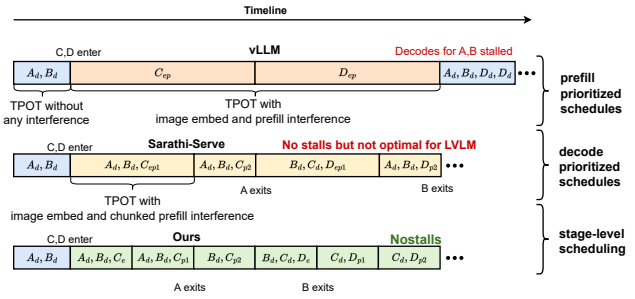

Figure 7. Comparison of the generation stall problem under different scheduling strategies. A, B, C, and D represent different requests. The subscript "e" denotes the image encode stage, "p" denotes the complete prefill stage, and "d" denotes the decode stage. "ep" represents the serial execution of encode and prefill, while "p0" and "p2" represent chunked prefill.

One implementation is to trigger the full image encoding process when the chunked prefill reaches the portion containing the image. However, image encoding interrupts the decoding process, causing generation stalls. In contrast, our approach schedules at a finer stage granularity, enabling better control over the execution time of each batch and facilitating the implementation of a stage-disaggregated architecture.

Takeaway-3: In current LLM inference schedulers, the alternation between prefill and decode involves a trade-off between throughput and latency. State-of-the-art systems now use chunked prefill technology and stall-free scheduling strategies to balance tail latency. For multimodal large models, the merging of encode stage makes stall-free scheduling suboptimal.

3.3 Disaggregation Methods

Some large language model inference engines adopt an architecture that disaggregate the prefill and decode stages [27, 41]. The main advantage of this architecture is that it decouples the computation between the prefill and decode stages, reducing resource interference between the two, thereby lowering TPOT. Additionally, the system can flexibly adjust the number of nodes allocated to each stage based on the ratio of prefill to decode in the actual workload. However, this architecture also introduces additional data transfer overhead and sacrifices the parallel throughput optimization potential between the computationally intensive prefill stage and the memory-intensive decode stage. In multimodal large models, the inference process introduces an encode stage, so whether it is still suitable to adopt the stage disaggregation methods remains to be further explored. Possible existing disaggregation methods include: E+P+D, EP+D, and ED+P.

E+P+D: Each stage is fully decoupled, making it easier to independently allocate computing resources based on the load characteristics of each stage. Since the encode stage does not need to store KV cache or load the language model, its nodes can support larger encode batch sizes and more concurrent requests. EP+D: Only one KV cache transfer is required, and both the vision model and the language model can be processed in parallel, helping to reduce transfer overhead. ED+P: The vision model and language model can be processed in parallel, with the prefill stage remaining independent, which helps reduce TTFT.

Takeaway-4: In multimodal large models, various disaggregation methods such as E+P+D, EP+D, and ED+P should be weighed against load characteristics and SLOs to achieve the maximization of resource utilization and optimization of response times.

4 Design and Implementation

We design and implement HydraInfer to address the aforementioned challenges. As illustrated in Figure 8. The core component, Hybrid EPD Disaggregation (§4.4), evaluates the

performance of different disaggregation methods, based on the request's TTFT and TPOT service level objectives (SLOs) and selects the optimal disaggregation method along with the appropriate number of instances. The system receives client requests via the API Server, which forwards them to the Request Scheduler. The scheduler performs load balancing based on request types, dispatching them to the corresponding Encode or Prefill instances. Each type of instance loads specific model on demand and allocates its own cache space accordingly. Within each instance, a Batch Scheduler (§4.2) aggregates requests for improved parallel efficiency, while the Migrate Scheduler (§4.3) is responsible for migrating requests across instances to achieve dynamic load balancing. Additionally, the Request Processor (§4.1) performs preprocessing for newly received requests.

4.1 Request Processor

To enable a stage-level scheduling strategy, we abstract a Request Processor that preprocess requests when they enter the system. The Request Processor first performs tokenization and image processing on the incoming requests. The processed request is then passed to the Stage Processor, which transforms it into a sequence of tasks — such as encode, prefill, decode, and migrate—and prepares control parameters for each task in advance as much as possible, including the slot ID for the KV cache. These tasks are finally dispatched to the Batch Scheduler for intra - instance scheduling. This design is motivated by the following considerations: First, by offloading part of the request scheduling computation to the Request Processor ahead of time, we reduce the CPU overhead during subsequent autoregressive inference. This preprocessing can be parallelized using a thread pool, mitigating the bottleneck caused by CPU-intensive tasks (e.g., generating control parameters for inference) and I/O-intensive operations (e.g., image processing). Second, the system becomes more extensible to support additional stages and can easily incorporate token pruning optimizations at different stages. Third, preprocessing requests allows the system to anticipate the subsequent stages of each request, making it compatible with hierarchical KV cache designs and facilitating prefetching.

4.2 Batch Scheduler

We propose a Stage-level batching algorithm (Algorithm 1) to enable intra-instance request scheduling. This algorithm serves as a unified scheduling strategy for different types of instances and operates at the stage granularity in an iterative manner. As shown in Algorithm 1, Stage-level scheduling first sets the maximum batch size for image encoding and the token budget for language models based on the user-defined SLO. Specifically, during system initialization, we use binary search to profile the maximum encode batch size and token budget that ensures the execution time of each subsequent batch iteration remains below the TPOT SLO. This design guarantees controlled computation latency for each iteration, thereby reducing token-to-token latency. In each iteration, we first include all ongoing decode requests in the current batch. Then, we check if there are any partially computed chunked prefill tasks; if so, we add them to the batch. If no such tasks exist, we check whether there are encode tasks to process and include them if available. This strategy prioritizes completing requests in the prefill stage to reduce TTFT. New encode tasks are only processed when no requests are in the prefill stage. We compare various scheduling strategies in Figure 7, where our approach significantly reduces TPOT. As previously discussed in section 3.1, for prefill and decode tasks within the same batch, we employ fused kernels such as FlashAttention or FlashInfer. For encode and decode tasks in the same batch, we utilize multi-stream execution to run the language model and vision model in parallel for higher throughput. Our stage-level scheduling system also

Algorithm 1 Stage-level batching. First the batch is filled with ongoing decode tokens and and optionally one prefill chunk or some image encode from ongoing. Finally new request are added within the token and image budget so as to maximize throughput with minimal latency impact on the TPOT of delaying the ongoing decodes.

Input: Maximum TTFT required for SLO $TPOT_{max}$, the queue for waiting requests $Q_{waiting}$, the set for running requests $S_{running}$.
Output: batch of request, B

Initialize $\tau_t \leftarrow compute\_token\_budget(TPOT_{max})$
Initialize $\tau_e \leftarrow compute\_image\_budget(TPOT_{max})$
Initialize $n_t \leftarrow 0$
Initialize $n_e \leftarrow 0$
Initialize $B \leftarrow \emptyset$
Initialize $has\_prefill \leftarrow False$
for R in $S_{running}$ **do**
  if is_decode_stage(R) **then**
    $n_t \leftarrow n_t + 1$
    $B \leftarrow B \cup \{R\}$
for R in $S_{running}$ **do**
  if is_prefill_stage(R) and $n_t < \tau_t$ **then**
    $has\_prefill \leftarrow True$
    $n_t \leftarrow n_t$+ get_chunked_prefill_size(R)
    $B \leftarrow B \cup \{R\}$
while $has\_prefill$ and $n_t < \tau_t$ **do**
  $R \leftarrow$ get_next_text_request($Q_{waiting}$)
  $n_t \leftarrow n_t$+ get_chunked_prefill_size(R)
  $B \leftarrow B \cup \{R\}$
if $has\_prefill == False$ **then**
  for R in $S_{running}$ **do**
    if is_encode_stage(R) and $n_e < \tau_e$ **then**
      $n_e \leftarrow n_e$+ get_image_number(R)
      $B \leftarrow B \cup \{R\}$
  while $n_e < \tau_e$ **do**
    $R \leftarrow$ get_next_multi_media_request($Q_{waiting}$)
    $n_e \leftarrow n_e$+ get_image_number(R)
    $B \leftarrow B \cup \{R\}$
for R in $S_{running}$ **do**
  if is_migrate_stage(R) **then**
    $B \leftarrow B \cup \{R\}$
return B

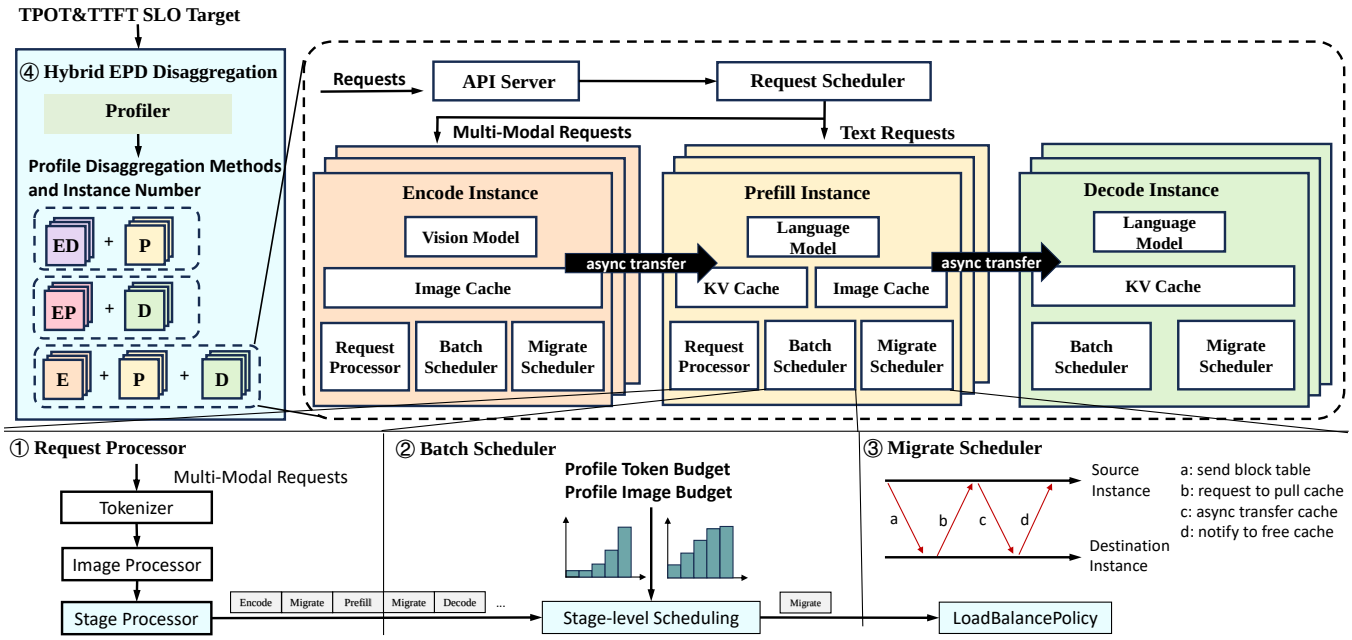

Figure 8. HydraInfer system architecture.

supports flexible stage partitioning. For example, to support request migration, we introduce a dedicated migrate stage for handling migration operations. We include all requests in migrate stage in current batch.

4.3 Migrate Scheduler

The Migrate Scheduler is responsible for executing request migration tasks. When the Migrate Scheduler receives a migration task from the Batch Scheduler, it initiates the migration process. We adopt a pull-based model for request migration, which helps prevent cache overflows at the receiving instance due to overload, such as KV cache or image cache exhaustion. The migration process consists of four steps. Step 1: The Migrate Scheduler at the source instance notifies the target instance of the control information for the request to be migrated (including the page tables of the KV cache and image cache). The target instance then enqueues the request for future scheduling. Step 2: Once the target instance schedules the request, it creates new page tables and requests to pull the cache blocks from the source. Two backend mechanisms are supported. One is CUDA IPC memory handles, which are limited to intra-node transfers; the other is NCCL, which supports both intra- and inter-node transfers. Step 3: The source instance asynchronously transfers the KV cache and image cache to the target instance. Step 4: After migration is complete, the target instance notifies the source to release the resources held by the migrated request. The Migrate Scheduler also handles load balancing across multiple target instances and can adopt strategies such as round-robin or random selection.

4.4 Hybrid EPD Disaggregation

To achieve an efficient disaggregated inference system, we design a hybrid EPD disaggregation multi-instance architecture (Figure 8). Built on a general-purpose inference engine, the system flexibly generates specialized instances (E, P, D, EP, ED) for different task types by configuring key parameters such as the number of KV cache blocks and image cache blocks. Each instance internally adopts a unified task batching and scheduling strategy (Algorithm 1). At the deployment level, instances are composed into different disaggregation methods based on task functionalities, such as EP+D, ED+P, and E+P+D, to support diverse inference workflows. Finally, we profile the workload and SLOs to select the optimal disaggregation configuration including disaggregation methods and instance numbers for specific application scenarios, optimizing both system throughput and resource utilization. This enables the selection of the optimal stage partitioning scheme and the allocation ratio among different types of inference instances to adapt to current workload demands.

4.5 Implementation

HydraInfer is an efficient online inference system for MLLMs, comprising approximately 8K lines of Python code and 3K lines of C++ and CUDA code. The frontend and inference engine are implemented in Python, while the data transmission modules and model computation kernels are implemented in C++ and CUDA. The system supports both single-instance inference and multi-instance hybrid EPD disaggregated inference. For online inference, it adopts a RESTful API frontend connected to several parallel inference

engine instances. The frontend is compatible with OpenAI-style APIs, allowing users to configure sampling parameters such as the maximum number of output tokens. We build the multi-instance inference system using Ray [25], where each instance is implemented as a Ray actor. Data transfer for the image cache and KV cache is enabled through CUDA IPC memory handles [7] and NCCL [8]. We utilize the FlashAttention [9] and FlashInfer [36] kernels to implement page attention [18]. To facilitate inter-instance request migration, we adopt centralized, paged memory management for image tokens. The image token cache is managed as one layer of a single-token cache, while the KV cache is managed as a multi-layer of two-token cache. Both caches share a similar management interface and data transfer interface. To reduce performance overhead caused by multiple small write-block kernel launches, we implement a unified fused kernel for both KV cache and image cache operations.

5 Evaluation

In this section, we present a detailed evaluation of HydraInfer under various scenarios.

5.1 Experiments Setup

Cluster testbed. We deploy HydraInfer on an 8-GPU node, which has 8 NVIDIA H800 GPUs connected with NVLink with 96 physical CPU cores and 2TB of RAM.

Baseline Method. We compare our system with vLLM [18], Text-Generation-Inference (TGI) [11], and SGLang [40]. Currently, vLLM is a single-instance inference system, and its disaggregated architecture is still under development. Therefore, we compare against the single-instance version of vLLM. vLLM has two versions of its inference engine: v1 and v0. Version v1 supports a decode-priority scheduling strategy for multimodal large models, while version v0 is an older release that does not support chunked prefill for multimodal models. TGI is a high-performance text generation inference service developed by Hugging Face, designed for deploying large language models (LLMs) in production environments. TGI adopts a hybrid architecture combining Rust and Python. SGLang is a fast serving framework for large language models and vision language models. It makes interaction with models faster and more controllable by co-designing the backend runtime and frontend language.

Models. We test different models to demonstrate the generalization ability of our system. We evaluated LLaVA-1.5-7B [22], LLaVA-NeXT-7B [19], and Qwen2-VL-7B [3]. The number of tokens generated for the same image differs across these models, which in turn impacts the request load. LLaVA-1.5 encodes each image into 576 tokens, while LLaVA-NeXT and Qwen2-VL have different methods to adjust the number of tokens based on the image resolution.

Datasets. We selected datasets from different scenarios to ensure a comprehensive evaluation across diverse application domains and workload characteristics. TextCaps [30] aims to challenge the ability of models to understand text in images, focusing on the relationship between image content and text in image description tasks. POPE [21] is used to evaluate the issue of object hallucination in large vision-language models. MME [12] assesses the performance of large vision-language models on perception and cognition tasks. VizWiz [13] contains images taken by blind users along with corresponding visual questions, aiming to improve visual question answering (VQA) capabilities for visually impaired users. TextVQA [31] is a dataset for visual question answering that focuses on evaluating a model's ability to understand and reason about text embedded in images. The workload of the three stages varies across these datasets. The decode stage workload is also related to the kernel implementation of the inference engine. To ensure consistent load across different inference engines, we first perform inference on the datasets and record the number of tokens generated during decoding. For subsequent tests with different inference engines, we set the max tokens parameter and the ignore eos parameter to fix the number of output tokens. Different datasets have varying workloads. The results obtained from LLaVA-NeXT-7B inference are shown in Figure 9.

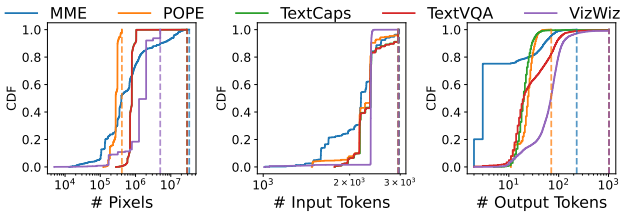

Figure 9. The workload of the LLaVA-NeXT-7B model across different datasets.

Metric. As previously introduced in §2.3, common metrics for evaluating inference engines including TTFT, TPOT, SLO Attainment, and Goodput have been used.

Details. The testing environment uses CUDA version 12.4. The KV cache block size is 16, our image cache block size is 576, and all models, as well as intermediate variables, KV cache, and image cache, are set to fp16 precision. vLLM runs in eager mode, with vLLM-v0 adopting the default prefill-prioritized scheduling strategy, and vLLM-v1 adopting the default decode-prioritized scheduling strategy. All inference engines use the same chat template; the prefix cache and CUDA graph are not enabled.

5.2 SLO Attainment Evaluation

To evaluate the maximum request rate that can be sustained while satisfying the SLO under different workloads, we sample a set of requests from each dataset for testing. Since the datasets do not contain timestamps, we simulate request arrivals using a Poisson process at a fixed rate (requests per

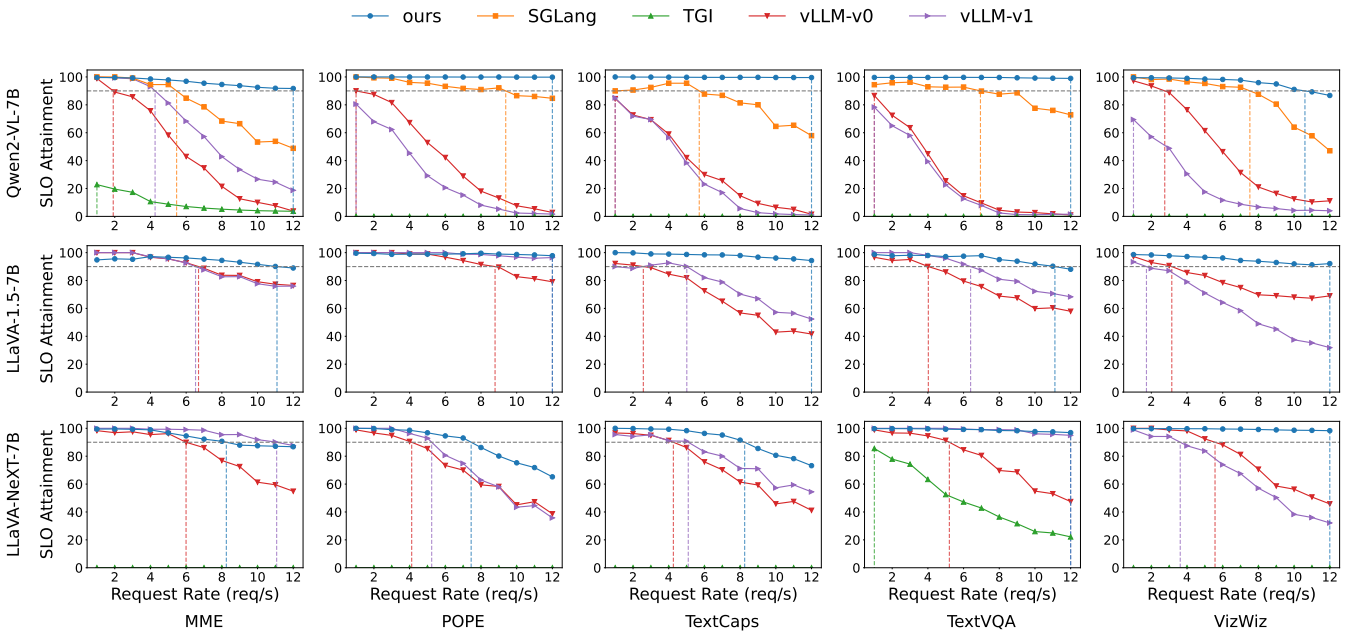

Figure 10. Comparison of SLO satisfaction across different inference engines, the Request Rate is the average load per-GPU.

Model	Dataset	TTFT (s)	TPOT (s)
LLaVA-1.5-7B	VizWiz	8	0.04
LLaVA-1.5-7B	TextVQA	0.25	0.04
LLaVA-1.5-7B	MME	0.25	0.06
LLaVA-1.5-7B	POPE	0.25	0.04
LLaVA-1.5-7B	TextCaps	0.25	0.04
LLaVA-NeXT-7B	VizWiz	8	0.12
LLaVA-NeXT-7B	TextVQA	8	0.12
LLaVA-NeXT-7B	MME	8	0.14
LLaVA-NeXT-7B	POPE	8	0.06
LLaVA-NeXT-7B	TextCaps	8	0.08
Qwen2-VL-7B	VizWiz	8	0.14
Qwen2-VL-7B	TextVQA	1	0.12
Qwen2-VL-7B	MME	1	0.14
Qwen2-VL-7B	POPE	1	0.04
Qwen2-VL-7B	TextCaps	1	0.14

Table 3. SLO settings under different workloads.

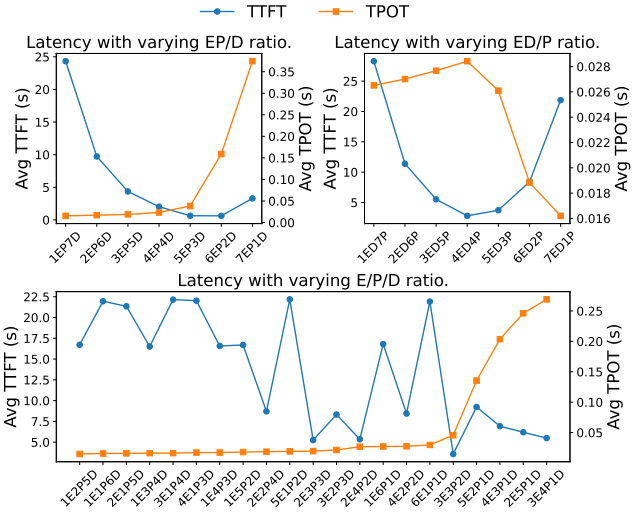

Figure 11. The impact of node ratios on TPOT and TTFT under the three disaggregation methods.

second). As different models encode the same image into varying numbers of tokens, this leads to different prefill and decode workloads, which in turn affect the SLO configuration. Therefore, we define separate SLO thresholds for each model and dataset, as detailed in Table 3.

The experimental results are shown in Figure 10. The gray dashed line denotes the 90% SLO attainment threshold, and the vertical dashed line indicates the request rate at which each inference system achieves 90% SLO attainment, measured as goodput. From the figure, we can observe that our system consistently achieves significantly higher goodput than other systems. Specifically, for the Qwen2-VL-7B model, our system outperforms SGlang by 2x–3x in goodput. For the LLaVA-1.5-7B and LLaVA-NeXT-7B models, we achieve a 1.5x–4x goodput improvement compared to vLLM. This performance gain is attributed to our Hybrid EPD Disaggregation architecture, which reduces interference between different execution stages, thereby lowering TPOT for lightweight requests and increasing the SLO attainment rate. In

the MME scenario with the LLaVA-NeXT-7B model, however, our system performs slightly below vLLM-v1. This is because the MME dataset consists of image classification tasks with minimal decode workload, and Hybrid EPD Disaggregation cannot significantly reduce TPOT by mitigating decode-stage interference from prefill and encode. Additionally, the scheduling overhead introduced by request migration may increase TPOT, resulting in marginal or even negative effects.

5.3 E/P/D Ratio

In this section, we investigate the impact of different node ratios on overall system performance. Using the TextCaps dataset, we evaluate various system architectures under a load of 8 requests per second, measuring both the Time to First Token (TTFT) and the average per-token latency (TPOT), as shown in Figure 11. For the EP+D disaggregated architecture, when the node ratio is 1EP7D, the limited number of EP nodes results in a heavy workload on each, causing a significant increase in TTFT. Meanwhile, the abundance of D nodes leads to the lowest TPOT. As the number of EP nodes increases and the number of D nodes decreases, TPOT gradually increases. Eventually, when the ratio reaches 7EP1D, the insufficient number of D nodes causes the decode stage to become a performance bottleneck. Due to the use of a pull-based model for request migration, EP nodes are blocked while waiting for responses from overloaded D nodes. This increases queuing delays for new requests, leading to another rise in TTFT. In the ED+P disaggregated architecture, insufficient ED nodes lead to bottlenecks in both the encode and decode stages, negatively impacting both TTFT and TPOT. When P nodes are insufficient, the prefill stage becomes the primary bottleneck, similarly resulting in increased TTFT. For the fully disaggregated E+P+D architecture, we sort the results by TPOT in ascending order across different node ratios. The results show that TPOT is approximately negatively correlated with the number of D nodes. Furthermore, given the same number of D nodes, certain E-to-P node ratios can significantly reduce TTFT, demonstrating more effective stage-level load balancing. From the above analysis, we conclude that, with the different workloads and SLO requirements, there does not exist a fixed optimal node ratio that maximizes system performance. To this end, our proposed Hybrid EPD Disaggregation framework adopts a profile-driven approach that automatically searches for the optimal node ratio under a given workload and SLOs, thereby achieving maximum performance.

5.4 Disaggregation Methods

Based on the analysis in the previous section, we can determine the optimal node ratio for each disaggregation method. By comparing these optimal ratios, we can identify the most effective disaggregation method under specific conditions.

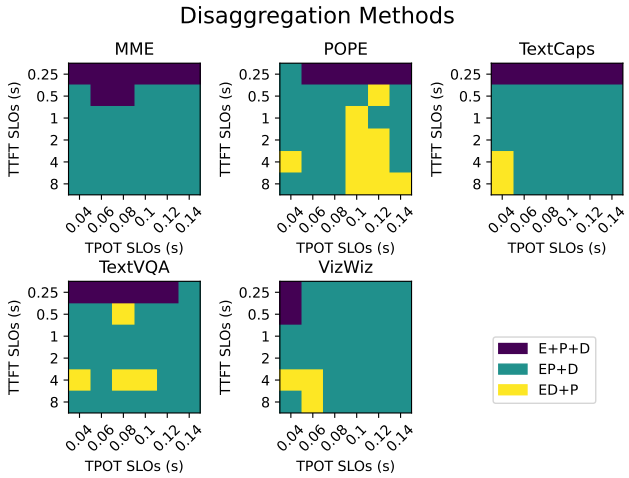

Figure 12. The impact of TPOT SLO and TTFT SLO on the optimal disaggregation method under different workloads.

However, whether a single disaggregation method consistently outperforms the others across various scenarios remains an open question. In this section, we investigate how different workload intensities, TPOT SLOs, and TTFT SLOs influence the choice of the optimal disaggregation method. We conduct experiments using the LLaVA-Next-7B model and record the optimal disaggregation method under varying TPOT and TTFT SLO constraints, as shown in Figure 12. The figure presents the preferred disaggregation method under different dataset workloads and performance requirements. The results reveal that different datasets exhibit distinct preferences for disaggregation methods. For example, under stringent TTFT constraints, the fully disaggregated E+P+D architecture demonstrates clear advantages. In contrast, ED+P and EP+D show superior performance in other scenarios. These findings highlight the importance of evaluating multiple disaggregation methods comprehensively and selecting them based on specific scenarios, further validating the effectiveness of our proposed Hybrid EPD Disaggregation approach in terms of adaptability and performance optimization.

5.5 Latency Breakdown

To better understand the performance characteristics of HydraInfer and identify potential system bottlenecks, we conduct a detailed latency breakdown analysis of service requests. We divide the lifecycle of each request into multiple stages: encode queueing, encode execution, EP migration, prefill queueing, prefill execution, PD migration, decode queueing, and decode execution. For each stage, we measure the average latency. We evaluate the LLaVA-1.5-7B model on the TextCaps dataset under a 1E3P4D disaggregation configuration. The latency distribution across stages is shown in

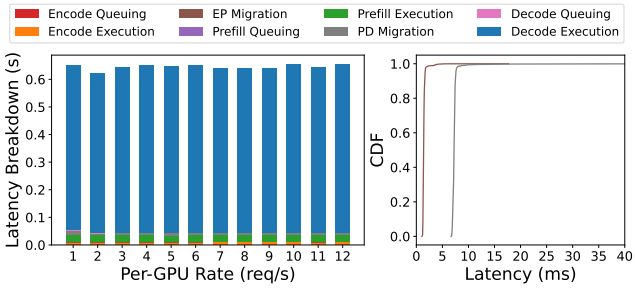

Figure 13. Latency breakdown when serving llava1.5-7B on textcaps dataset.

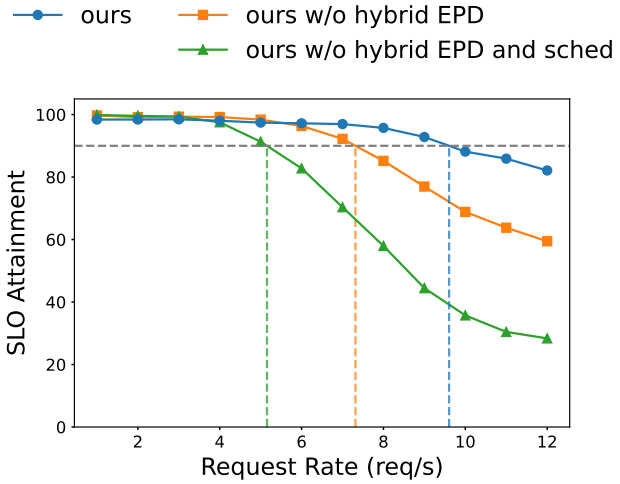

Figure 14. Ablation study of scheduling strategy.

Figure 13. The results indicate that the majority of the request latency is concentrated in the decode stage, followed by the prefill and encode stages. The overhead introduced by image cache and KV cache migration is minimal, accounting for less than 1% of the total latency. Specifically, 95% of image cache migration complete within 2ms, and 95% of KV cache migration complete within 8ms — both typically shorter than the execution time of a single decode batch. Therefore, their impact on TPOT can be considered negligible.

5.6 Abligation Study

To validate the effectiveness of our scheduling strategy Algorithm 1 and Hybrid EPD Disaggregation (§4.4) in controlling TPOT and improving the SLO achievement rate, we conducted a comparative experiment on the TextCaps dataset using the LLaVA-Next-7B model. As shown in Figure 14, we first disable the Hybrid EPD Disaggregation method and use 8 general-purpose inference instances. In this setting, the goodput drops from 9.5 to 7.2 req/s, indicating that a well-chosen disaggregation strategy can effectively reduce inter-stage interference, leading to better TPOT control and improved SLO satisfaction. Next, we further disable the stage-level scheduling policy while still using 8 general-purpose instances. The goodput decreases from 7.2 to 5.1 req/s, demonstrating that our stage-level scheduling strategy helps better control the execution time of each batch, thus achieving tighter TPOT adherence.

6 Related Work

6.1 MLLM Serving Systems

Several works have focused on optimizing MLLM serving systems. At the system level, Singh *et al.* [32] proposed decoupling critical processing stages (encode, prefill, and decode) and optimizing resource allocation. Qiu *et al.* [29] conducted a comprehensive performance analysis of MLLM serving characteristics and similarly proposed a decoupled architecture concept with separate Image Nodes and Text Nodes. However, compared to our HydraInfer, these works only consider the methods where the vision model and language model are disaggregated or co-located but executed sequentially, opting instead for fully disaggregated architectures. They overlook the potential GPU efficiency gains from co-locating and executing them in multi-stream parallel ways. HydraInfer comprehensively evaluates both decoupled and co-located multi-stream parallel strategies to select the optimal model disaggregation method, which covers their design as E+P+D disaggregation methods subset, and proved that it is not optimal in many scenarios in our evaluations. (We didn't evaluate them directly since they are not open-sourced or implemented.)

On the other hand, works like Inf-MLLM [26] and Elastic Cache [23] reduce computational overhead or memory footprint through KV cache approximate reuse or pruning. These model-level approaches trade off model performance to reduce performance overhead, which is orthogonal to our system-level and scheduling-level optimizations that unchanged model performance.

The widely used SOTA LLM inference frameworks, including SGLang [40], vLLM (v0 and v1) [18], and TGI [11], also extended themselves to support MLLM model inference, all of them mix Encode, Prefill, and Decode in the same instances to perform inference, and as we discussed earlier, interference between their three stages can impact SLO guarantees adversely, and our HydraInfer achieves higher goodput while satisfying SLOs in our evaluations.

6.2 LLM Scheduling Optimization

Numerous works have explored LLM scheduling optimizations. Early works [18, 38] proposed continuous batching to improve GPU utilization significantly. However, due to the co-located sequential execution of prefill and decode stages, these methods create a trade-off between TTFT and TPOT. Sarathi-Serve introduced chunked-prefill [1], attempting to balance TTFT and TPOT by splitting long prefill stages into

chunks and interleaving them with decode tasks. However, this approach inherently trades TTFT for TPOT without fundamentally eliminating Prefill-Decode interference.

DistServe [41] proposed a decoupled architecture that isolates prefill and decode stages on separate GPUs to avoid interference entirely, but introduces challenges of reduced GPU utilization and KV cache transfer overhead. While these works are not directly applicable to MLLM inference, their decoupling philosophy inspired subsequent multimodal research. Concurrently, works on KV cache reuse [15, 28], KV cache migration optimization [17, 27], and placement dynamic adjustment [14, 27] are orthogonal to our approach and could be integrated into HydraInfer.

7 Conclusion

This paper proposes HydraInfer, an efficient system architecture for multimodal large model inference tasks. To address issues such as coupling during image-text processing and inflexible scheduling in existing systems, HydraInfer introduces the Hybrid Encode-Prefill-Decode Disaggregation design, enabling efficient scheduling and utilization of resources across different stages. HydraInfer supports stage-level batch processing and parallel execution mechanisms, improving the system's throughput. Experimental results demonstrate that the system can effectively handle diverse multimodal inference workloads while ensuring service quality, showcasing strong generality and performance advantages.

References

[1] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2024. Taming throughput-latency tradeoff in llm inference with sarathi-serve. (2024). arXiv: 2403.02310 [cs.LG].

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

[3] Jinze Bai, Shuai Bai, Shusheng Yang, Shijie Wang, Sinan Tan, Peng Wang, Junyang Lin, Chang Zhou, and Jingren Zhou. 2023. Qwen-vl: a versatile vision-language model for understanding, localization, text reading, and beyond. (2023). https://arxiv.org/abs/2308.12966 arXiv: 2308.12966 [cs.CV].

[4] Shuai Bai et al. 2025. Qwen2. 5-vl technical report. *arXiv preprint arXiv:2502.13923*.

[5] Mu Cai, Haotian Liu, Dennis Park, Siva Karthik Mustikovela, Gregory P. Meyer, Yuning Chai, and Yong Jae Lee. 2024. Vip-llava: making large multimodal models understand arbitrary visual prompts. (2024). https://arxiv.org/abs/2312.00784 arXiv: 2312.00784 [cs.CV].

[6] Liang Chen, Haozhe Zhao, Tianyu Liu, Shuai Bai, Junyang Lin, Chang Zhou, and Baobao Chang. 2024. An image is worth 1/2 tokens after layer 2: plug-and-play inference acceleration for large vision-language models. (2024). https://arxiv.org/abs/2403.06764 arXiv: 2403.06764 [cs.CV].

[7] NVIDIA Corporation. 2007. Memory Management. https://docs.nvidia.com/cuda/cuda-runtime-api/index.html.

[8] NVIDIA Corporation. 2016. NVIDIA Collective Communications Library (NCCL). https://developer.nvidia.com/nccl.

[9] Tri Dao. 2023. Flashattention-2: faster attention with better parallelism and work partitioning. (2023). https://arxiv.org/abs/2307.08691 arXiv: 2307.08691 [cs.LG].

[10] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. Flashattention: fast and memory-efficient exact attention with io-awareness. (2022). https://arxiv.org/abs/2205.14135 arXiv: 2205.14135 [cs.LG].

[11] Hugging Face. 2023. Text generation inference. https://github.com/huggingface/text-generation-inference. Accessed: 2025-05-04. (2023).

[12] Chaoyou Fu et al. 2024. Mme: a comprehensive evaluation benchmark for multimodal large language models. (2024). https://arxiv.org/abs/2306.13394 arXiv: 2306.13394 [cs.CV].

[13] Danna Gurari, Qing Li, Abigale J. Stangl, Anhong Guo, Chi Lin, Kristen Grauman, Jiebo Luo, and Jeffrey P. Bigham. 2018. Vizwiz grand challenge: answering visual questions from blind people. (2018). https://arxiv.org/abs/1802.08218 arXiv: 1802.08218 [cs.CV].

[14] Cunchen Hu et al. 2024. Inference without interference: disaggregate llm inference for mixed downstream workloads. (2024). https://arxiv.org/abs/2401.11181 arXiv: 2401.11181 [cs.DC].

[15] Cunchen Hu et al. 2024. Memserve: context caching for disaggregated llm serving with elastic memory pool. *arXiv preprint arXiv:2406.17565*.

[16] Shengding Hu et al. 2024. Minicpm: unveiling the potential of small language models with scalable training strategies. *arXiv preprint arXiv:2404.06395*.

[17] Yibo Jin et al. 2024. P/d-serve: serving disaggregated large language model at scale. *arXiv preprint arXiv:2408.08147*.

[18] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. (2023). https://arxiv.org/abs/2309.06180 arXiv: 2309.06180 [cs.LG].

[19] Feng Li, Renrui Zhang, Hao Zhang, Yuanhan Zhang, Bo Li, Wei Li, Zejun Ma, and Chunyuan Li. 2024. Llava-next-interleave: tackling multi-image, video, and 3d in large multimodal models. (2024). https://arxiv.org/abs/2407.07895 arXiv: 2407.07895 [cs.CV].

[20] Junnan Li, Dongxu Li, Silvio Savarese, and Steven Hoi. 2023. Blip-2: bootstrapping language-image pre-training with frozen image encoders and large language models. (2023). https://arxiv.org/abs/2301.12597 arXiv: 2301.12597 [cs.CV].

[21] Yifan Li, Yifan Du, Kun Zhou, Jinpeng Wang, Wayne Xin Zhao, and Ji-Rong Wen. 2023. Evaluating object hallucination in large vision-language models. (2023). https://arxiv.org/abs/2305.10355 arXiv: 2305.10355 [cs.CV].

[22] Haotian Liu, Chunyuan Li, Qingyang Wu, and Yong Jae Lee. 2023. Visual instruction tuning. (2023). https://arxiv.org/abs/2304.08485 arXiv: 2304.08485 [cs.CV].

[23] Zuyan Liu, Benlin Liu, Jiahui Wang, Yuhao Dong, Guangyi Chen, Yongming Rao, Ranjay Krishna, and Jiwen Lu. 2024. Efficient inference of vision instruction-following models with elastic cache. In *European Conference on Computer Vision*. Springer, 54–69.

[24] Haoyu Lu et al. 2024. Deepseek-vl: towards real-world vision-language understanding. *arXiv preprint arXiv:2403.05525*.

[25] Philipp Moritz et al. 2018. Ray: a distributed framework for emerging ai applications. (2018). https://arxiv.org/abs/1712.05889 arXiv: 1712.05889 [cs.DC].

[26] Zhenyu Ning, Jieru Zhao, Qihao Jin, Wenchao Ding, and Minyi Guo. 2024. Inf-mllm: efficient streaming inference of multimodal large language models on a single gpu. *arXiv preprint arXiv:2409.09086*.

[27] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. 2024. Splitwise: efficient generative llm inference using phase splitting. (2024). https://arxiv.org/abs/2311.18677 arXiv: 2311.18677 [cs.AR].

[28] Ruoyu Qin, Zheming Li, Weiran He, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. 2024. Mooncake: a kvcache-centric disaggregated architecture for llm serving. *arXiv preprint arXiv:2407.00079*.

[29] Haoran Qiu et al. 2025. Towards efficient large multimodal model serving. *arXiv preprint arXiv:2502.00937*.

[30] Oleksii Sidorov, Ronghang Hu, Marcus Rohrbach, and Amanpreet Singh. 2020. Textcaps: a dataset for image captioning with reading comprehension. (2020). https://arxiv.org/abs/2003.12462 arXiv: 2003.12462 [cs.CV].

[31] Amanpreet Singh, Vivek Natarajan, Meet Shah, Yu Jiang, Xinlei Chen, Dhruv Batra, Devi Parikh, and Marcus Rohrbach. 2019. Towards vqa models that can read. (2019). https://arxiv.org/abs/1904.08920 arXiv: 1904.08920 [cs.CL].

[32] Gursimran Singh et al. 2024. Efficiently serving large multimedia models using epd disaggregation. *arXiv preprint arXiv:2501.05460*.

[33] Gemini Team et al. 2023. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*.

[34] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems*, 30.

[35] Peng Wang et al. 2024. Qwen2-vl: enhancing vision-language model's perception of the world at any resolution. (2024). https://arxiv.org/abs/2409.12191 arXiv: 2409.12191 [cs.CV].

[36] Zihao Ye et al. 2024. Accelerating self-attentions for llm serving with flashinfer. (Feb. 2024). https://flashinfer.ai/2024/02/02/introduce-flashinfer.html.

[37] Shukang Yin, Chaoyou Fu, Sirui Zhao, Ke Li, Xing Sun, Tong Xu, and Enhong Chen. 2024. A survey on multimodal large language models. *National Science Review*, 11, 12, (Nov. 2024). DOI: 10.1093/nsr/nwae403.

[38] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: a distributed serving system for Transformer-Based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, (July 2022), 521–538. ISBN: 978-1-939133-28-1. https://www.usenix.org/conference/osdi22/presentation/yu.

[39] Zhihang Yuan et al. 2024. Llm inference unveiled: survey and roofline model insights. (2024). https://arxiv.org/abs/2402.16363 arXiv: 2402.16363 [cs.CL].

[40] Lianmin Zheng et al. 2024. Sglang: efficient execution of structured language model programs. (2024). https://arxiv.org/abs/2312.07104 arXiv: 2312.07104 [cs.AI].

[41] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. 2024. Distserve: disaggregating prefill and decoding for goodput-optimized large language model serving. (2024). https://arxiv.org/abs/2401.09670 arXiv: 2401.09670 [cs.DC].

[42] Deyao Zhu, Jun Chen, Xiaoqian Shen, Xiang Li, and Mohamed Elhoseiny. 2023. Minigpt-4: enhancing vision-language understanding with advanced large language models. (2023). https://arxiv.org/abs/2304.10592 arXiv: 2304.10592 [cs.CV].

[43] Kan Zhu et al. 2024. Nanoflow: towards optimal large language model serving throughput. (2024). https://arxiv.org/abs/2408.12757 arXiv: 2408.12757 [cs.DC].