EAServe: Encode-Aware Disaggregated Serving for Multimodal Large Language Models

Kunxiong Zhu
University of Georgia
Athens, GA, USA
Kunxiong.Zhu@uga.edu

Zhihao Shu
University of Georgia
Athens, GA, USA
Zhihao.Shu@uga.edu

Hangyu Zheng
University of Georgia
Athens, GA, USA
hyzheng@uga.edu

Minghai Qin
Western Digital
San Jose, CA, USA
qinminghai@gmail.com

Miao Yin
University of Texas at Arlington
Arlington, TX, USA
miao.yin@uta.edu

Gagan Agrawal
University of Georgia
Athens, GA, USA
gagrawal@uga.edu

Wei Niu
University of Georgia
Athens, GA, USA
wniu@uga.edu

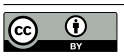
Abstract

With growing emphasis on (text-only) LLM serving, disaggregating the two stages: Prefill and Decode onto separate GPU pools is now a standard optimization. However, multimodal LLMs (MLLMs), which add a third phase, Encode, pose new challenges for optimizing resource allocation. Encode turns images, video, or audio into embeddings that the language model can consume, yielding a three-stage Encode–Prefill–Decode (EPD) pipeline, which requires new research for optimizing resource allocation and utilization. Existing frameworks offer only partial answers: text-only PD systems lack Encode, while EPD frameworks expose it as a separate service without regulating downstream request flow. New challenges arise because the three stage pipeline carries a structural resource imbalance: every request enters through Encode before downstream work can begin, yet per-request execution leaves the encode GPU severely underutilized even at high loads, and this idle capacity starves the downstream Prefill and Decode workers.

Addressing this challenge, we reposition Encode as the control point of the EPD pipeline, exposing three tightly coupled dimensions for optimization: when work enters downstream, where prefill executes, and how the GPU is shared. We instantiate this in EAServe across two co-designed layers. Its runtime manages load-adaptive micro-batching, rate-controlled partial offload to a co-resident prefill worker, and dynamic SM partitioning for predictable co-location. The configuration layer, Hybrid Auto Selection (HAS), then navigates the joint space of GPU allocation, encode batch size, and offload ratio by pruning unbalanced allocations with per-stage capacity profiling and refining the remainder through TPE-based Bayesian optimization. In our extensive evaluation on three MLLM architectures spanning image, video, and audio, EAServe delivers up to 4.3× and 1.7× higher goodput than NVIDIA Dynamo and vLLM, respectively, under identical SLO constraints, sustains substantially more balanced and higher GPU utilization across the EPD pipeline, while reaching near-optimal disaggregated configurations faster than baseline search methods.

CCS Concepts: • **Computer systems organization → Cloud computing**; • **Software and its engineering → Scheduling**; • **Computing methodologies → Machine learning**.

Keywords: Multimodal large language models, Disaggregated serving, GPU resource allocation, SM partitioning, Request scheduling



This work is licensed under a Creative Commons Attribution 4.0 International License.
PACT '26, Chicago, IL, USA
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2915-7/2026/10
https://doi.org/10.1145/3838684.3846886

1 Introduction

Multimodal Large Language Models (MLLMs) are rapidly moving from research prototypes to deployed services, as they now power interactive assistants, visual question answering, document understanding, video understanding, and speech-based applications [6, 17, 25, 28, 30]. As deployment grows, the systems challenge is shifting from supporting MLLMs at all to serving them efficiently at scale under latency objectives and fluctuating demand [14].

Serving an MLLM efficiently is more challenging than serving a text-only LLM because the pipeline is more heterogeneous. Text-only inference splits into a compute-bound Prefill and a memory-bandwidth-bound Decode, or PD, and this clean split is what makes *disaggregating* the two onto independently scaled GPU pools (*PD separation*) effective in practice [38, 54]. MLLMs add a third stage in front of this pipeline. Specifically, a *modality encoder*, typically a transformer, turns images, video, or audio into embeddings that the language model can then consume. The result is the encode-prefill-decode (EPD) pipeline, which is widely used in modern MLLM deployments [33]. The new Encode stage's compute and bandwidth demands vary with modality, input size, and arrival pattern. This extra axis of heterogeneity makes EPD harder to schedule than PD.

Despite being on the critical path, Encode has received less attention from recent work than the two stages following it. Text-only PD frameworks [38, 54] expose only a token-level interface and coordinate Prefill with Decode. EPD frameworks such as

NVIDIA Dynamo [33] factor Encode out as a separate service but operate it as a pass-through: one request at a time, with operator-set GPU allocation and no mechanism for regulating how work enters the rest of the pipeline. The cost of this passivity is that a single encoder forward pass is too small to saturate a modern GPU, yet batching is constrained by the latency cost of waiting for a larger batch to form. The stage that gates pipeline admission is therefore also the one that leaves the most capacity unused: under sustained load, the encode GPU stays largely idle while downstream workers wait on a growing encode queue and tail latency inflates [33].

We argue that Encode should instead be repositioned as the pipeline's *control point*. As the pipeline's entry, it naturally controls three coupled dimensions: *when* work enters downstream, *where* prefill executes, and *how* the GPU is shared under co-location. Co-optimization is needed as tuning any of them in isolation breaks the others, and yet challenging, as joint space is too large to search by hand. With this motivation, this paper presents `EAServe`, a layered serving architecture built around this control point. At the top, Hybrid Auto Selection (HAS) performs offline configuration search over the joint space of GPU allocation, encode batch bound, and prefill offload ratio. At the bottom, three encode-aware runtime mechanisms expose the parameters that HAS searches and enforce the latency guarantees that make the search results reliable. Different mechanisms dominate on different modalities, motivating both the runtime co-design and the joint search.

In all, this paper makes the following contributions.

- We identify Encode as the natural control point of disaggregated MLLM serving: both the entry point of the pipeline and a major source of unused GPU capacity.
- We design *Hybrid Auto Selection* (HAS), an offline framework that automatically picks the deployment configuration of the EPD pipeline for a given serving workload.
- We design *three encode-aware runtime mechanisms* that turn HAS's chosen configuration into a working serving stack and adapt to load online with no per-deployment tuning: load-adaptive micro-batching with a Poisson-gap dispatch rule, rate-controlled partial offload to a local prefill worker, and dynamic per-micro-batch Streaming Multiprocessor (SM) partitioning.
- We implement `EAServe` and show significant speedups over the state-of-the-art production-level baselines.

We evaluate `EAServe` on three representative MLLMs covering the main input modalities: LLaVA-v1.6-34B [28] for image, Qwen2.5-VL-32B [7] for video, and Ultravox-v0.6-27B [17] for audio. Experiments run on an 8-GPU server under a range of TTFT/TPOT latency targets, and we compare against the state-of-the-art baselines: NVIDIA Dynamo [33] and vLLM [24]. Across these workloads, `EAServe` improves goodput (SLO-compliant request rate) by up to 4.3× over Dynamo and 1.7× over vLLM, and yields more balanced GPU utilization across the EPD pipeline.

2 Background and Motivation

2.1 EPD Pipeline And Metrics

An MLLM extends a text-only LLM with non-text inputs such as images, video, or audio. Broadly, MLLMs include a *modality encoder* (e.g., CLIP ViT [40] for images, Whisper [41] for audio) that maps the raw input into embedding vectors, a lightweight

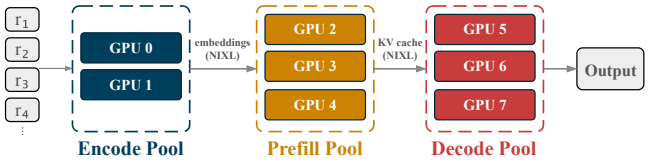


Figure 1: The disaggregated pipeline of EPD.

connector that projects them into the LLM's input space, and the *LLM backbone* that consumes them alongside text tokens to generate text autoregressively. Disaggregated MLLM serving thus has three stages that scale on independent GPU pools (Figure 1): Encode runs the encoder and connector to produce *modality embeddings*; Prefill executes the backbone on the concatenation of modality embeddings and text prompt, populates the key–value (KV) cache, and emits the first output token; Decode generates the remaining tokens autoregressively from the context.

This decomposition results in stages with heterogeneous resource profiles: Encode performs short forward passes; Prefill is a single attention-heavy pass whose cost grows with context length; and Decode runs a long, memory-bandwidth-bound autoregressive loop. Independent scaling lets each stage be provisioned independently [33], at the cost of carrying intermediate state (modality embeddings and KV cache) across workers. Within a stage, when the model exceeds one GPU, weights are usually sharded with tensor parallelism (TP), the standard intra-replica strategy in latency-sensitive serving; we denote TP=k for k-way sharding. In all, we denote the resulting GPU allocation across the three stages by (E, P, D), which, together with the choice to enable co-located workers, specifies the deployment configuration for a given (model, workload, cluster) combination.

A configuration's quality cannot be judged by throughput alone, since the EPD pipeline is expected to meet user-facing latency targets at every stage. Following prior LLM-serving work [24, 38, 54], we therefore use latency and SLO-aware metrics. *Time-to-first-token (TTFT)* is the elapsed time from request arrival to the first output token; in an EPD pipeline it covers encode, embedding transfer, prefill, and any queueing along the path. *Time-per-output-token (TPOT)* is the mean interval between consecutive output tokens during Decode. We report both at the mean and the 99th-percentile tail (P99). Following DistServe [54], we summarize end-to-end performance with *goodput*, the rate of requests that satisfy the SLO:

$$\text{Goodput} = N_{\text{SLO-met}}/T, \tag{1}$$

where $N_{\text{SLO-met}}$ counts requests with TTFT $\le T_f$ and TPOT $\le T_p$ over a run of duration T. We adopt goodput as the primary metric in our evaluation (§4), since it captures both throughput and SLO attainment in a single number.

2.2 Why Encode Becomes a System Bottleneck

As the entry point of the pipeline, Encode gates the downstream stages; yet a single encode pass rarely places enough work on the GPU to keep it busy. We illustrate the resulting inefficiency with two measurements taken on a standard EPD deployment (Figure 2). Full hardware, model, and workload details appear in §4.1.

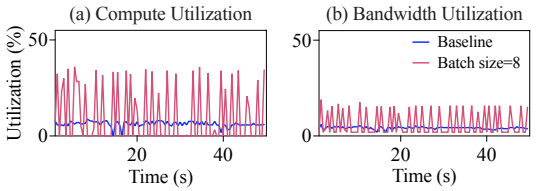


Figure 2: GPU computation and HBM-bandwidth utilization on LLaVA-v1.6-34B [28] under NVIDIA Dynamo [33].

Table 1: A partial gain from adding micro-batching to Dynamo on LLaVA-v1.6-34B.

Configuration	Throughput (req/s)	Mean TTFT (s)	Mean TPOT (ms)
Dynamo default	0.75	367.4	179
+ micro-batch	1.09	112.1	411

Observation 1. Encode gates the pipeline yet leaves substantial GPU capacity unused. Encode batching is constrained by request arrivals and the latency cost of waiting for a larger batch to form, so the system cannot grow the batch size at will. Individual encoder passes are consequently too small to saturate the GPU: both compute and high-bandwidth memory (HBM) bandwidth utilization remain low (as shown in Figure 2, Baseline). At sustained arrival rates the per-request encoder throughput falls below the offered load, so downstream Prefill and Decode workers spend most of their time waiting on a growing encode queue rather than processing embeddings.

Observation 2. Micro-batching helps but does not close the gap. Adding a simple micro-batcher (B=8) raises throughput and lowers TTFT (Table 1), but mean TPOT rises sharply because the accelerated encode floods the downstream Prefill and Decode workers. The encode GPU itself remains far from saturated, alternating between brief bursts and long idle gaps (Figure 2, B=8). Micro-batching captures part of the available gain but does not resolve the underlying imbalance.

These two observations suggest three design choices for an encode-aware system. First, a *dispatch policy* must be decided that adapts the encode batch bound to the current arrival rate, so the dispatcher tracks load rather than waits on a static timeout. Second, an *offload mechanism* is needed that channels the encode-GPU's idle capacity into a co-located prefill worker without slowing encode itself. Third, a *resource-sharing primitive* must keep the two co-located workers isolated as the batch size changes. The three need to be co-optimized: the batch bound shapes the SM split that protects latency, the SM split shifts how much offload the local worker can accept, and any reallocation across (E, P, D) rebalances all three. The resulting joint configuration is too large to tune by hand for each new (model, workload, cluster), which is the challenge our work addresses.

2.3 SM Partitioning

A modern GPU offers hundreds of TFLOPS of compute and terabytes per second of HBM bandwidth, and this has motivated co-locating two workloads on one GPU for full utilization. Existing

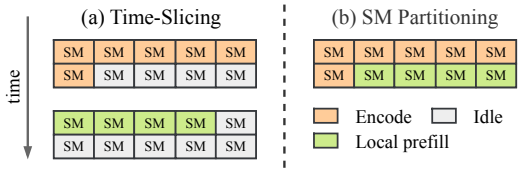


Figure 3: SM partitioning via `libsmctrl`. (a) Default time-slicing: encode (orange) and local prefill (green) alternate, leaving idle SMs (gray). (b) TPC-mask partitioning runs them concurrently on disjoint SM subsets.

Table 2: A summary of key symbols.

Symbol	Meaning
$PC = (\text{alloc}, B, s)$	Deployment configuration searched by HAS, consisting of the stage allocation, maximum encode batch size, and offload ratio.
alloc	GPU allocation across the Encode, Prefill, and Decode stages.
B	Maximum encode batch size.
s	Target fraction of requests routed to remote prefill.
Λ	Target system-wide arrival rate.
T_f, T_p	SLO thresholds on P99 TTFT and P99 TPOT, respectively.

primitives form a hierarchy of increasing isolation strength, with implications for how workloads can interfere with one another. The default CUDA scheduler simply time-slices the full SM array, so only one process runs at a time and concurrency is lost. CUDA Multi-Process Service (MPS) [32] merges kernels from multiple processes into a single CUDA context to enable concurrent execution, optionally with per-client compute-percentage hints. However, the hints are advisory, and a long-running kernel from one client can still delay latency-sensitive work in another, so this solution raises throughput without protecting tail latency. The strongest option is *spatial partitioning*, which assigns disjoint physical resources to each process. NVIDIA Multi-Instance GPU (MIG) does this in hardware on A100/H100-class GPUs and partitions both SMs and HBM bandwidth, but its reconfiguration is heavyweight and it is unavailable on the consumer and workstation GPUs widely used for MLLM serving. Software-level *spatial SM partitioning* via `libsmctrl` [8] fills this gap by writing per-process Texture Processing Cluster (TPC) masks (Figure 3): a process executes only on its allotted SMs, and the partition can be reconfigured in under a microsecond. What it cannot isolate, however, is HBM bandwidth, since the SM subsets still share a single memory bus, and whichever process issues more traffic displaces the other's.

Software SM partitioning is a natural fit for the encode-side underutilization identified above. Encode and Prefill are both compute-bound transformer forward passes, so co-locating them under a TPC-mask split divides compute proportionally and protects encode latency, while sub-microsecond reconfiguration lets the split track the encode batch size as it changes online. The same primitive, however, also imposes two constraints that shape the rest of our design. First, the HBM-bandwidth limitation rules out placing Decode on the encode GPU, since it streams the KV cache token by token and would saturate the shared bus regardless of the SM split. Decode, therefore, remains on a dedicated GPU in our design.

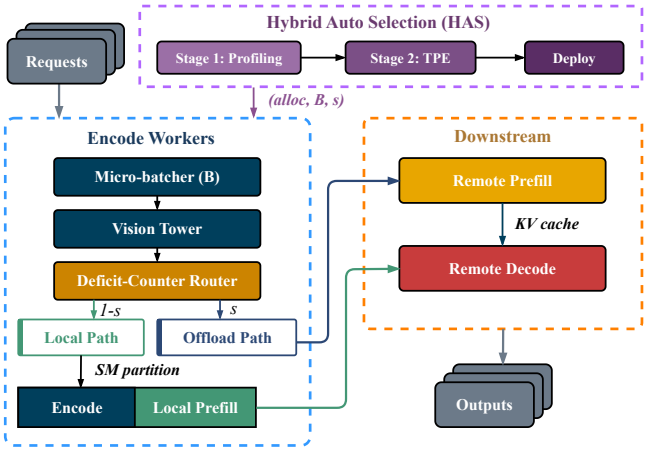


Figure 4: Overview of EAServe.

Second, the right partition ratio depends on the current encode batch bound, which varies online, making static split ineffective. Specifically, the resulting split also has to absorb the local prefill load determined by the offload ratio, which couples the SM-split decision to the dispatch and offload mechanisms developed next.

3 System Design

Motivated by the discussion in previous section, EAServe is an encode-aware EPD serving framework organized in two co-designed layers. As shown in Figure 4, at the top, *Hybrid Auto Selection* (HAS) searches the joint configuration $PC = (\text{alloc}, B, s)$ offline, jointly choosing the GPU allocation, encode batch bound, and prefill offload ratio. At the bottom, three encode-aware runtime mechanisms realize the chosen PC online by controlling *when* encode dispatches, *where* each request's prefill runs, and *how* the encode GPU is shared with a co-resident local prefill worker. We describe each layer in detail in the following subsections, and Table 2 lists the symbols used in the rest of this section.

3.1 Hybrid Auto Selection

This critical layer, abbreviated as HAS, automates the selection of the configuration (or PC) for each deployment target, replacing the manual configuration that operators would otherwise perform. Given the GPU budget (or count) G, the model $\mathcal{M}$, the workload $\mathcal{D}$, and the target arrival rate Λ, HAS returns the deployment configuration that maximizes throughput:

$$PC^{\star} = \arg\max_{PC} \text{Throughput}(PC), \tag{2}$$

with ties broken by mean TTFT, then mean TPOT. The search runs once offline, for each combination of GPU budget, model, workload, and target-rate, and its output is then used for configuring the system at the runtime.

Existing serving stacks automate only parts of this search: text-only PD systems such as DistServe [54] and Splitwise [38] search stage allocation but have no Encode to tune, while EPD frameworks such as Dynamo [33] run Encode as a pass-through with B and s fixed to 1 rather than exposed as tunable parameters. Searching for the configuration manually is impractical here, because the three components of PC are tightly coupled: the SM split that protects encode latency depends on the encode batch bound B; the SM partitioner enforces this split at runtime. The SM split, in turn, shifts the offload ratio s that keeps the pipeline balanced, since the local prefill worker's compute capacity depends on its SM share; and any change to alloc rebalances all three simultaneously. Even on our 8-GPU testbed, the resulting space contains hundreds of valid configurations, and the count grows exponentially with the GPU budget. Exhaustive search is prohibitive because every candidate requires a full deployment trial of several minutes on our 8-GPU testbed, adding up to hours or longer for a single workload. Purely analytical modeling can avoid this cost, but it is very challenging to capture model and hardware characteristics fully.

Our key insight, and the central design contribution of HAS, is that the configuration space has an internal asymmetry that can be exploited to combine analytical reasoning and empirical search effectively. The first-order effect of alloc is monotonic and predictable from per-stage capacity alone: if any stage is starved, the pipeline has a bottleneck there regardless of how B and s are chosen, so such allocations can be ruled out without ever being deployed. The interaction between B and s, by contrast, is non-monotonic within the set of feasible allocations and is best located by direct end-to-end measurement. HAS therefore splits the work between two stages (Algorithm 1): Stage 1 (lines 1–12) profiles each stage's per-worker capacity in parallel on disjoint GPU subsets, discards allocations whose dominant stage already exceeds capacity at the target rate, and retains the top-K candidates by a bottleneck-balance score. Stage 2 (lines 13–20) seeds a Tree-structured Parzen Estimator (TPE) study around each survivor's analytical optimum and refines it through end-to-end trials, stopping once the top configurations stabilize. This split is what lets HAS span the full coupled (alloc, B, s) space in tens of trials rather than hundreds of full-system deployments, ranking configurations by throughput, with mean TTFT and mean TPOT as tie-breakers when configurations achieve comparable throughput.

Stage 1: Profiling and Pruning. The role of Stage 1 is to analytically answer the question *"is this allocation worth considering?"*. The enabling observation is that per-stage capacity is a property of the combination of model, hardware, and workload, and is independent of how the GPUs are partitioned. Thus, each stage can be profiled once on a small GPU subset and the resulting capacities reused to score every candidate allocation. Encode profiling adaptively doubles B until throughput saturates, producing a saturation capacity C_E that captures the batching response. Prefill and Decode profiling each run a stage-targeted stress workload on an isolated worker, yielding per-worker capacities C_P and C_D. The three profiles run in parallel on disjoint GPU subsets and together cost less than a single end-to-end trial.

Given these profiles, we model each stage as an independent open-queue server with utilization

$$\rho_\sigma = \frac{\Lambda / n_\sigma}{C_\sigma}, \qquad \sigma \in \{E, P, D\}, \tag{3}$$

where n_σ is the number of workers assigned to stage σ. For prefill, $\rho_P = \Lambda/(C_P\,(n_P + n_L\,\alpha))$, where n_L is the number of co-resident local prefill workers and $\alpha \in [0, 1]$ is the fraction of encode-GPU memory budget allocated to a co-resident local prefill worker, used as a

Algorithm 1: Hybrid Auto Selection (HAS)

Input: GPU budget G,
model $\mathcal{M}$,
workload $\mathcal{D}$,
target rate Λ
Output: Best configuration (alloc*, $B^\star$, $s^\star$)

Stage 1: Parallel profiling and screening
Profile the encode saturation capacity C_E and the per-worker prefill and decode capacities C_P and C_D on disjoint GPU subsets in parallel
foreach feasible allocation a with $E{+}P{+}D{=}G$ **do**
  $\rho_E \leftarrow (\Lambda/n_E)\,/\,C_E$
  $\rho_P \leftarrow \Lambda\,/\,(C_P\,(n_P + n_L\,\alpha))$
  $\rho_D \leftarrow (\Lambda/n_D)\,/\,C_D$
  $\text{score}(a) \leftarrow \max(\rho_E,\ \rho_P,\ \rho_D)$
  $B_{\max}(a) \leftarrow \lceil \Lambda/n_E \rceil$
  $s^*(a) \leftarrow C_{\text{remote}}(a)\ /\ (C_{\text{remote}}(a) + C_{\text{local}}(a))$
end
Discard allocations where score ≥ 1 # infeasible: overloaded stage
Prune allocations where score $> 1.2 \times \min_a(\text{score})$
$\mathcal{A}_K \leftarrow$ top-K allocations by score # K=3 in our deployment

Stage 2: End-to-end refinement via TPE
Initialize Optuna study with per-alloc search ranges: for each $a \in \mathcal{A}_K$, $B \in [1, B_{\max}(a)]$ and $s \in [s^*(a){-}0.2,\ s^*(a){+}0.2]$
for $t \leftarrow 1$ **to** $N_{\max}$ **do**
  (alloc, B, s) $\leftarrow$ TPE-sampled configuration
  Deploy (alloc, B, s);
  run benchmark $\mathcal{D}$ at rate Λ
  Record throughput y_t and mean (TTFT, TPOT) as tie-breakers
  if *top-3 spread on* $y_t < 1\%$ *or* 5 *rounds without improvement* **then** break
end
return arg max by lexicographic order:
$y_t \succ -\text{TTFT}_{\text{mean}} \succ -\text{TPOT}_{\text{mean}}$

memory-bound proxy for its per-worker capacity. The local term $n_L\,\alpha\,C_P$ and the remote term $n_P\,C_P$ are kept separate so that the formula remains correct under any (n_E, n_L, n_P) allocation. Stage 2 absorbs any deviation between this analytical estimate and the realized local capacity. The 1.2× slack used below leaves room for Stage 2 to recover near-miss allocations. We summarize each allocation by its bottleneck-balance score

$$\text{score}(a) = \max(\rho_E,\ \rho_P,\ \rho_D), \tag{4}$$

and apply three filters in sequence. First, we discard allocations with $\rho_\sigma \geq 1$, since here the stage lacks the capacity to sustain the offered load. Second, we prune those whose score exceeds 1.2× the minimum (the 20% gap balances false pruning against wasted Stage 2 trials). Finally, we only retain the top-K remaining allocations as $\mathcal{A}_K$ (K=3 in our deployment).

Stage 1 also seeds the parameter ranges that Stage 2 will explore, computed per allocation $a \in \mathcal{A}_K$: B is capped at $B_{\max}(a) = \lceil \Lambda/n_E \rceil$ (a worker is unlikely to fill batches much larger than its arrival rate), and s is centered at the load-balancing point $s^*(a) = C_{\text{remote}}(a)/(C_{\text{remote}}(a) + C_{\text{local}}(a))$ with $C_{\text{remote}}(a) = n_P\,C_P$ and $C_{\text{local}}(a) = n_L\,\alpha\,C_P$, and a ±0.2 search window.

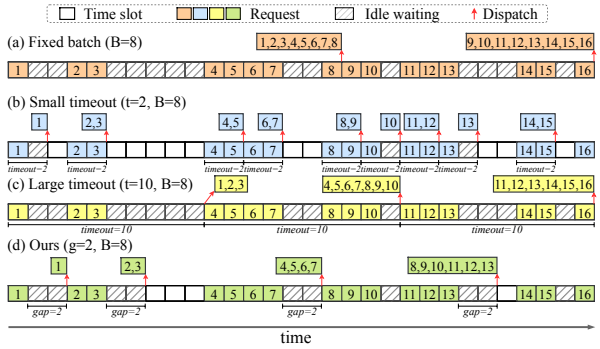


Figure 5: Four dispatch policies on the same Poisson arrival trace. *(a)* Fixed batch bound, no timeout. *(b)* Timeout too small. *(c)* Timeout too large. *(d)* Our Poisson-gap threshold $g = \kappa/\lambda$.

Stage 2: Bayesian Refinement. Stage 2 addresses the part of the problem that Stage 1 does not address. Specifically, as the (B, s) surface within each surviving allocation is non-monotonic, so we treat it as a black box and probe it with end-to-end trials. Because the surface is mixed-type (categorical alloc, discrete B, continuous s) and trials are expensive, we use a single Optuna [5] study with the TPE sampler, which natively handles such spaces and concentrates trials on regions the surrogate model believes are promising. Each trial deploys the full E+P+D system at rate Λ, runs the benchmark workload, and records throughput together with mean TTFT and mean TPOT for tie-breaking. The search terminates when the spread among the top three configurations falls below 1% or five consecutive rounds show no improvement. The final configuration is chosen lexicographically: throughput first, then mean TTFT and TPOT. Our evaluation confirms that HAS reaches near-optimal configurations substantially faster than uninformed search baselines as we will show later (§4.3). The HAS hyperparameters (pruning factor 1.2×, top-K=3, s window ±0.2, 30-trial budget) are fixed once and reused across all reported runs.

3.2 Load-Adaptive Micro-Batching

HAS sets the encode batch bound B offline; the micro-batcher's runtime task is to decide *when* to dispatch a partial batch that has not yet reached B. Each forward pass of a vision encoder at small B is dominated by kernel-launch and scheduling overhead, so batching is necessary for achieving high throughput. At the same time, every additional request the dispatcher waits for adds latency to the critical path. An effective dispatch policy needs to identify this trade-off point automatically across different arrival rates.

We want a threshold that tightens under high load and relaxes under low load, without per-workload retuning. Standard dispatch policies [12, 33] fall short here. A fixed timeout grows mistuned whenever load shifts. Queue-length triggers cannot tell a transient lull from the end of a burst, holding batches open too long at low load. Continuous batching [51] sidesteps the question entirely, since it applies only to autoregressive decoding.

Poisson-gap dispatch rule. Our key insight is that the dispatch decision is fundamentally a property of the arrival process itself, so we derive the dispatch threshold directly from that process rather

than treating it as a free parameter. Let $\lambda = \Lambda/n_E$ be the per-encode-worker arrival rate, and let $p \in (0,1)$ be the target probability of filling a batch before dispatch. Modeling arrivals as Poisson (§4.5 reports sensitivity under non-Poisson arrivals), inter-arrival times are exponentially distributed: $\Pr(X \le g) = 1 - e^{-\lambda g}$. Starting from the first enqueued request, let $X_1, \ldots, X_{B-1}$ denote the gaps between successive arrivals; we require each of these gaps to be at most g for the batch to fill:

$$\Pr\big(\max\{X_1, \ldots, X_{B-1}\} \le g\big) = \big(1 - e^{-\lambda g}\big)^{B-1}.$$

Setting this equal to p and solving:

$$\kappa := -\ln\big(1 - p^{\frac{1}{B-1}}\big), \qquad g := \frac{\kappa}{\lambda}. \tag{5}$$

The result has the desired property $g \propto 1/\lambda$: the threshold tightens automatically under high load, when batches fill quickly. It also relaxes under low load, when latency pressure is lower, with no per-workload retuning. We assume λ is known from the deployment's target arrival rate; in production where λ drifts, κ can be recomputed periodically from a windowed arrival-rate estimate. The single tuning constant p trades batch fullness against tail latency; we use p=0.6 throughout, which lies at the knee of this trade-off, and performance is insensitive to the choice within $[0.5, 0.7]$.

Given g, the batcher dispatches when any of three conditions is satisfied: the batch reaches size B, the idle-gap timer (reset on every arrival) exceeds g, or the batch's age from its first request exceeds a hard cap $A = 1000$ ms. The hard cap is not intended for the common case; it bounds the worst-case TTFT under pathological arrival patterns in which each request arrives just before g expires and would otherwise keep a batch open for $g(B-1)$ seconds. The complete dispatch condition is

$$\text{dispatch when: } |B_{\text{cur}}| = B \ \vee \ \text{idle} > g \ \vee \ \text{age} > A,$$

where B_{cur} is the set of requests currently in the micro-batch. Figure 5 shows how the rule detects burst boundaries and groups arrivals into variable-sized batches without manual tuning; the batcher adds $O(1)$ work per arrival and incurs sub-microsecond timer overhead.

3.3 Rate-Controlled Partial Offload

Once the micro-batcher has decided *when* an encode batch dispatches, the next question is *where* each request's prefill should run. Partial offload realizes this routing decision through encode's spatial tuning parameter $s \in [0,1]$, the target fraction of requests sent to a remote Prefill worker rather than to a co-resident local prefill worker on the encode GPU. We co-locate prefill, rather than decode, with encode for the reason established in §2.3; remote Prefill and all Decode workers therefore run on dedicated GPUs.

The router's role is to make s meaningful at runtime. If the actual routing pattern drifts from the configured ratio, or if requests reach one path in bursts, the latency observed by HAS reflects routing noise rather than the intended setting. Two natural baselines both fail this test: randomized routing (each request goes remote with probability s) is noisy at short horizons and can produce long runs of consecutive remote decisions even when s is moderate; windowed routing (send exactly $\lfloor sW \rfloor$ of every W requests remote) trades randomness for a new tuning parameter W whose appropriate

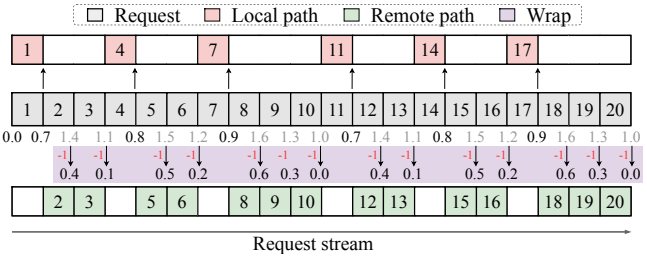


Figure 6: Deficit-counter routing with offload ratio s=0.7 over twenty requests. On each arrival the router increments d by s (value above the request). When $d \ge 1$ (purple band), the request goes to the remote path and d is decremented by 1 (value below the band); otherwise the request stays on the local path and d carries over to the next arrival.

value is workload-dependent. We therefore adopt a parameter-free scheme whose worst-case routing deviation is bounded analytically.

Deficit-controlled routing. Our solution is a counter-based scheme inspired by Deficit Round Robin [43] that converges exactly to s in the long run and bounds the worst-case run length per direction, without introducing any new tuning parameter. Where DRR maintains per-flow deficit counters to schedule packets among competing queues, we collapse the same idea into a single scalar that splits a request stream into a continuous-valued ratio between two paths. The resulting bounded-spacing property is what makes the offload ratio s chosen by HAS observable as latency at runtime, the property a black-box search like Stage 2 needs. The router maintains a real-valued accumulator d, initialized to 0; for each request it adds s to d, sends the request remotely if $d \ge 1$ (and decrements d by 1), and locally otherwise (Figure 6). Two properties follow. First, after n requests the cumulative remote count is $\lfloor ns + d_0 \rfloor$ for some $d_0 \in [0,1)$, so it stays within ± 1 of ns for all n and converges exactly to fraction s. Second, the counter never produces more than $\lceil 1/(1-s) \rceil$ consecutive remote decisions or $\lceil 1/s \rceil$ consecutive local decisions, which keeps the downstream queues smoother than randomized routing. The router adds $O(1)$ work per request.

Envelope-based embedding transfer. After encode, each request's vision embedding has to be delivered to its assigned prefill worker. Sending one embedding at a time would pay the transfer setup cost on every request. Instead, we group all embeddings for the same prefill worker into one contiguous tensor and ship them as a single NIXL [35] transfer per destination, so a micro-batch of N requests issues at most one transfer per prefill worker, rather than N separate ones. A small concurrency cap on in-flight sends provides backpressure and bounds memory use without stalling the pipeline.

3.4 Dynamic SM Partitioning

Once the router has decided *where* each request's prefill runs, the encode GPU may host both an encode worker and a co-resident local prefill worker. We still need to decide *how* the GPU is split between them. This split involves the following tradeoff: encode is on the critical path and should not be slowed below its latency target, while any compute encode does not use is wasted unless

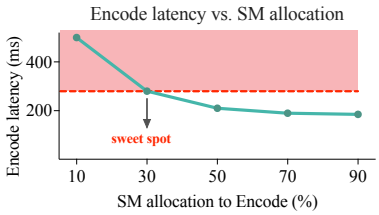


Figure 7: Encode latency versus SM allocation on LLaVA-v1.6-34B at B=4. The dashed line marks the 20% latency threshold defining the minimum allocation preserving encode QoS.

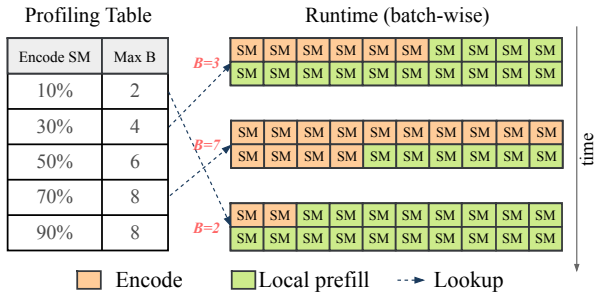


Figure 8: Dynamic SM partitioning. *Left*: the offline table maps each SM level to the largest batch size. *Right*: at runtime, the split between encode (orange) and local prefill (green) shifts with batch size while staying on disjoint SM subsets.

given to local prefill. This balance is workload-dependent, so the split has to be set per micro-batch rather than statically.

Our key insight is that the encode latency-versus-SM curve saturates: encode latency drops sharply as it receives more SMs, then flattens once it has enough to meet its target (Figure 7). This means there is a well-defined *smallest safe* SM share for each batch size, and any SM above that point can be released to local prefill at no cost to encode quality of service (QoS). We therefore re-split the SMs at every micro-batch: look up the smallest safe share for the current batch, give it to encode, and hand the remainder to local prefill. The split adapts to batch size with no new tuning parameter.

We materialize the *smallest safe share* as a lookup table indexed by SM-allocation level, built once at startup. Building it amounts to profiling encode latency at a small set of SM-allocation levels (5 in our deployment) across a range of batch sizes, and recording, for each SM level, the largest batch size whose latency stays close to the exclusive-access baseline. The cost is modest: each profiling point is a single encode-only forward pass, so the full sweep runs in minutes on one GPU and produces only a handful of table entries. The table depends only on the encode model, GPU, and workload, so it is reused across deployment rates and offload ratios, and its storage and lookup cost at runtime are negligible.

At runtime, the encode worker selects the smallest SM share whose supported batch size covers the current micro-batch size, and the local prefill worker takes the complementary fraction (e.g., 70% for encode and 30% for prefill). Both processes install the split through TPC masks via `libsmctrl` [8], which reconfigures in sub-microsecond and requires no kernel restart. Small batches therefore release more SMs to local prefill, while large batches reclaim them

Table 3: Evaluated MLLM architectures.

Model	Modal.	Encoder	LLM backbone	Params
LLaVA-v1.6-34B [28]	Image	CLIP ViT-L [40]	Yi-34B [1]	34B
Qwen2.5-VL-32B [7]	Video	ViT (dyn.)	Qwen2.5-32B	32B
Ultravox-v0.6-27B [17]	Audio	Whisper [41]	Gemma-2-27B [18]	27B

Table 4: SLO thresholds at the P99 level (TTFT in s / TPOT in ms) across five tiers.

Model	S1 (Strict)	S2 (Tight)	S3 (Moderate)	S4 (Standard)	S5 (Relaxed)
LLaVA	5/70	40/80	50/120	90/120	140/120
Qwen	5/60	10/70	40/110	80/110	110/110
Ultravox	2/50	3/60	5/80	10/100	30/130

for encode (Figure 8). On workloads where HAS selects s=1 (e.g., video in our evaluation), local prefill is not co-located and dynamic SM partitioning collapses to a single-tenant encode allocation, so it incurs no overhead when its benefit is small.

4 Evaluation

4.1 Experimental Setup

Models and workloads. We evaluate three MLLM architectures spanning image, video, and audio, with different encoders and LLM backbones (Table 3). For image and video, we use the standard random multimodal workload generator from the vLLM benchmarking suite [24], which produces synthetic inputs at realistic shapes paired with random text prompts of typical length. For audio, we draw real speech samples from LibriSpeech [37] with random text prompts. This mix follows established serving-benchmark practice [14, 38, 54] and isolates serving-system performance from dataset-specific content effects.

Testbed and baselines. Our primary testbed is a server with two AMD EPYC 9254 CPUs (48 cores, 96 threads), 1 TB DRAM, and eight NVIDIA RTX 6000 Ada GPUs (48 GB each, PCIe, CUDA 13.0), and Ubuntu 22.04 LTS. Each encode worker occupies one GPU at TP=1, while Prefill and Decode workers span 2 or 4 GPUs (TP=2 or TP=4), since the 27B–34B backbones require TP≥2 to fit on a single 48 GB GPU. Together with the choice of encode batch bound B and prefill offload ratio s, the resulting joint configuration space contains hundreds of valid points even at 8 GPUs. Both Dynamo's manual sweep and HAS's automated search draw from the same space. This setup captures the encode-prefill-decode imbalance our design targets. Multi-node scaling and cross-rack KV transfer will be topics for future efforts. When local prefill is enabled, it shares the encode GPUs through SM partitioning following the dynamic split rule of §3.4. We compare four system configurations under the same 8-GPU budget: *vLLM* (v0.14.1) and *Dynamo* (v0.9.0); *Ours (s=1)*, our system with s fixed to 1 (no local prefill); and *Ours (HAS-tuned)*, our system with HAS-tuned s (local prefill enabled with dynamic SM partitioning when s<1). All baselines use their best-performing configurations on this testbed. We further evaluate `EAServe` against two EPD-serving systems, EPDServe [44] and HydraInfer [14], on two eight-GPU platforms: RTX 6000 Ada and A100 SXM4 (Table 5).

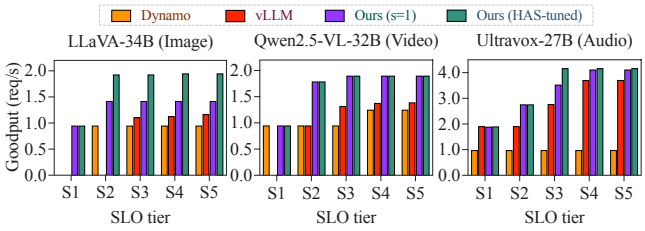


Figure 9: End-to-end goodput across five SLO tiers for image, video, and audio serving. The strictest tier (S1) characterizes the SLO frontier rather than an attainable operating point at the rates we sweep.

Metrics and SLOs. Following prior LLM-serving work [22, 54], we report goodput (Eq. 1), throughput, mean and P99 time-to-first-token (TTFT), and mean and P99 time-per-output-token (TPOT). Five SLO tiers per (model, workload) pair (Table 4) range from strict to relaxed, and goodput is computed independently at each tier. Following standard serving-benchmark practice [14, 38, 54], each run uses Poisson arrivals at a controlled rate. For the main end-to-end comparison, we vary the arrival rate from 1 to 5 req/s to cover under-, near-, and over-saturated regimes (the saturation point for our three models lies in the 2–4 req/s range). Each run processes 500 requests; reported numbers are medians of three independent runs. For Dynamo, "best manually tuned" means sweeping the feasible allocations and selecting the highest-goodput one at the target rate; HAS searches the same allocation space jointly with (B, s), so the comparison differs in search mechanism rather than in hardware budget.

4.2 End-to-End Comparison

We first compare against baselines across three models and five SLO tiers. Figure 9 reports goodput across SLO tiers, and Figure 10 reports throughput and latency under various request rates.

End-to-end results. As shown in Figure 9, `EAServe` achieves higher goodput than vLLM and Dynamo across the three evaluated modalities, reaching up to 4.3× higher goodput than Dynamo and 1.7× higher than vLLM at the moderate tier. The gap to Dynamo is much larger than the gap to vLLM because Dynamo coordinates the three stages at request granularity and does not balance encode-side compute against the downstream prefill and decode demand. This leaves the encode GPU as the dominant bottleneck under load. vLLM avoids this specific failure mode through continuous batching at the LLM, so the headroom we improve over it is smaller. The relative performance between Dynamo and vLLM also depends on the modality: Dynamo beats vLLM on video only at the strictest tier, where the multi-frame encoder dominates the pipeline and disaggregating it pays off. Dynamo, however, is slower than vLLM on audio, where the encoder is light and aggregated serving handles the decode-heavy workload more efficiently. EPD disaggregation alone therefore does not consistently beat aggregated serving; our encode-aware design closes this gap and delivers consistent gains over the baselines.

Most of the gain comes from adaptive batching and stage allocation (Figure 9), with local prefill adding a smaller, modality-dependent gain we examine below. The breakdown by modality is

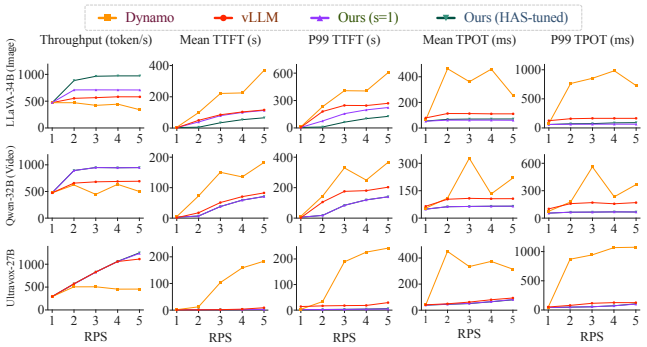


Figure 10: Throughput and latency comparison across different request rates for various modalities. Dynamo's TTFT diverges past saturation because its per-request encode (B=1) cannot match the offered rate; the encode queue grows monotonically within the run and every request inherits the accumulated wait.

as follows. On image, adaptive batching plus selective local prefill (with dynamic SM partitioning) are most effective. Video is dominated by stage allocation and adaptive batching, since HAS selects s=1 and skips local prefill entirely. For audio, the dominant factor is selective local prefill on top of the light Whisper encoder.

At the moderate tier, `EAServe` raises image goodput from 1.11 req/s (vLLM) and 0.95 req/s (Dynamo) to 1.93 req/s, reaches 1.90 req/s on video against 1.32 / 0.95, and 4.17 req/s on audio against 2.77 / 0.98. Tail latencies drop sharply at the same time: at Λ=2 req/s, P99 TTFT drops from Dynamo's 233.8 s to 10.8 s on image and from 144.8 s to 18.0 s on video. Dynamo's performance is so slow because its encode stage runs over saturation at this rate, so the encode queue grows unboundedly within the run and every request inherits the accumulated wait; `EAServe`'s encode-aware dispatch keeps the encoder at line rate, the queue never builds up, and the tail collapses to roughly the prefill+decode time alone. P99 TPOT drops in step (764→72 ms on image, 869→49 ms on audio) for the same reason: once encode keeps up with downstream demand, decode batches no longer stall mid-run, and TPOT returns to its native rate. The composition of these gains is itself modality-dependent. For image and video, encode is the upstream bottleneck, so adaptive batching alone is enough to shorten the encode–prefill tail and bring the TTFT value down. For audio, the encoder is light and the workload is decode-heavy, so the dominant factor is selective local prefill, which relieves downstream queueing and surfaces as a gain in TPOT and throughput.

Figure 11 further shows where these gains come from. Dynamo's request-granularity coordination keeps the encode GPU below 10% utilization, and naive fixed batching raises it only intermittently. `EAServe` fills the remaining idle gaps with co-located local prefill, raising sustained utilization to roughly 80%. This means the remote Prefill queue empties faster, which is the main reason TTFT drops so sharply in Figure 10.

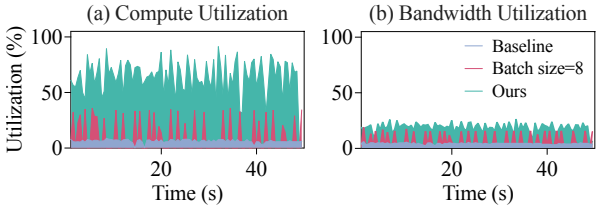


Figure 11: GPU utilization comparison against baseline.

Table 5: Throughput and latency comparison across two eight-GPU platforms on LLaVA-v1.6-34B at Λ=5 req/s, using the best-performing configuration of each system.

System	Tput	TTFT (s)		TPOT (ms)	
	(req/s)	Mean	P99	Mean	P99
RTX 6000 Ada 48 GB (PCIe)					
EPDServe	0.46	448	929	119	137
HydraInfer	0.58	315	664	402	477
EAServe	**1.95**	**66**	**127**	**71**	**91**
A100 SXM4 80 GB (NVLink)					
EPDServe	1.01	142	304	249	330
HydraInfer	1.51	118	214	128	233
EAServe	**4.91**	**16**	**29**	**91**	**105**

Modality-dependent behavior. The two "Ours" bars in Figure 9 split the gain into two parts. Dynamo vs. Ours (s=1) isolates adaptive micro-batching and HAS-tuned stage allocation, which accounts for a substantial portion of the improvement across modalities; Ours (s=1) vs. Ours (HAS-tuned) isolates the additional effect of local prefill. How much local prefill helps depends on the modality, and HAS chooses automatically. On image, the CLIP encoder is heavy enough to benefit from disaggregation yet light enough to leave SM headroom for local prefill. Therefore, HAS picks s<1 and goodput rises from 1.42 to 1.93 req/s. On audio, the Whisper encoder leaves substantial headroom and HAS again picks s<1. On video, the multi-frame encoder uses more than 90% of the SMs during batched forwards. Since SM partitioning does not isolate HBM bandwidth (§2.3), co-locating prefill would slow encode rather than help, and HAS correctly picks s=1; Ours (HAS-tuned) therefore coincides with Ours (s=1) on video, and the two "Ours" bars and lines overlap in Figures 9 and 10.

Cross-platform comparison. Beyond the vLLM and Dynamo results above, we further compare `EAServe` with EPD-serving systems EPDServe [44] and HydraInfer [14] on LLaVA-v1.6-34B (Table 5). We run the same image workload at Λ=5 req/s on the RTX 6000 Ada testbed and on eight A100 SXM4 GPUs (80 GB HBM2e each, NVLink). We use the latest public revisions available at evaluation time: EPDServe commit `9b111f7` and HydraInfer v0.1.0 (commit `009257f`). For each system, we select its best-performing configuration separately on each platform rather than carrying one configuration across GPU types. On RTX 6000 Ada, `EAServe` reaches 1.95 req/s, 4.24× EPDServe and 3.36× HydraInfer. On A100, it reaches 4.91 req/s, maintaining throughput advantages of 4.86× and 3.25×, respectively. The latency ordering is also consistent: `EAServe` has lower mean and P99 TTFT and TPOT than both baselines on both platforms. For example, its P99 TTFT is 127 s on RTX 6000 Ada, compared with 929 s for EPDServe and 664 s for HydraInfer; on A100, it falls to 29 s, compared with 304 s and 214 s, respectively. Thus, the advantage is preserved on both the PCIe-based RTX 6000 Ada platform and the NVLink-connected A100 platform.

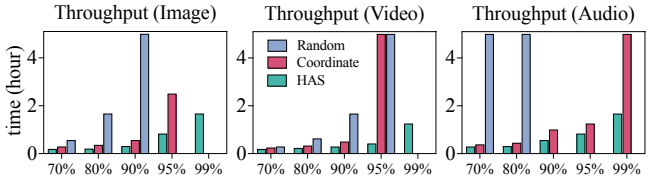


Figure 12: Wall-clock time HAS, Random, and Coordinate each take to reach a given percentile of the best-observed throughput. The per-modality reference is the maximum across all 3 methods × 30 trials; missing bars indicate the percentile was not reached within the 30-trial budget.

4.3 HAS Efficiency

We evaluate HAS as the configuration layer that selects (allocation, B, s) for each deployment. This search is non-trivial because the joint (alloc, B, s) space contains hundreds of configurations even at 8 GPUs, its dimensions interact non-monotonically, and the best operating point varies with modality and load. We therefore compare HAS against two search baselines under the same budget of 30 trials. Each trial deploys one configuration and runs 100 requests at Λ=5 req/s; we report results on the image (LLaVA-34B), video (Qwen-32B), and audio (Ultravox-27B) workloads. All methods search the same joint space of feasible allocations, micro-batch sizes, and offload ratios: *Random* samples uniformly from this space, whereas *Coordinate* performs coordinate descent over the three dimensions. For all three methods, the selected configuration is the one with the highest throughput observed within the 30-trial budget, with mean TTFT and mean TPOT as tie-breakers, and the reported throughput and latency metrics correspond to that configuration. Figure 12 reports the wall-clock search cost of each method.

Stage 1 pruning. Run at the same target rate as Stage 2 (Λ=5 req/s, chosen above saturation to expose method differences), Stage 1's bottleneck-balance score (Eq. 4) prunes the allocation dimension of the search space, keeping 3 of the 7 feasible allocations under the 1.2× threshold. The four pruned allocations all have a bottleneck stage at least 20% more loaded than the best-balanced candidate at this rate, so that stage would dominate end-to-end latency regardless of how B and s are tuned. This 7→3 reduction lets Stage 2 concentrate its 30-trial budget on balanced candidates, improving per-trial reliability.

Search quality. Across the three modalities, HAS consistently finds the highest-throughput configuration within the 30-trial budget. The benefit is largest on audio, where the light Whisper encoder leaves substantial encode-GPU headroom; HAS's directed search exploits this headroom more reliably than uninformed baselines; moderate on image, where the CLIP encoder is heavy enough to benefit from disaggregation but still leaves enough headroom for local prefill; and smallest on video, where the multi-frame encoder

Table 6: SM partitioning ablation on Ultravox-27B with local prefill enabled. All three configurations share allocation 2E(TP1)+1P(TP2)+1D(TP4), B=10, s=0.55, and Λ=10 req/s.

SM config	Tput (r/s)	P99 TTFT (s)	P99 TPOT (ms)
No partition (time-slicing)	4.78	43.7	236
Static 50/50	5.07	40.7	203
Dynamic (ours)	**5.21**	**36.7**	**167**

leaves little room for local prefill and narrows the useful part of the configuration space.

Search efficiency. Figure 12 reports the average wall-clock time each method takes to find a configuration reaching a given percentile of the best-observed throughput across the three modalities. HAS reaches the 99th percentile in 1.7 h on image, 1.3 h on video, and 1.7 h on audio, while Random fails to reach the 99th percentile on any modality within its 30-trial budget, and Coordinate fails on image and video. Even at lower percentiles HAS is consistently faster: it reaches the 90th percentile in 0.3 h on image (vs. 5.0 h for Random and 0.6 h for Coordinate) and 0.6 h on audio (where Random does not reach this percentile within its budget). These differences also reveal each baseline's failure mode: Random's slow rise to high percentiles suggests it spends much of its budget in poor regions of the search space, while Coordinate descends quickly at low percentiles but stalls at high ones, consistent with descent into a local optimum. HAS avoids both behaviors by pruning bottlenecked allocations before search, so its 30-trial budget concentrates on balanced candidates.

4.4 SM Partitioning Ablation

We isolate the contribution of dynamic SM partitioning by comparing three configurations on Ultravox-27B with local prefill enabled: (i) no SM partitioning (default CUDA time-slicing between encode and local prefill), (ii) static 50/50 SM split via `libsmctrl`, and (iii) our dynamic five-level scheme.

Table 6 reports the results at Λ=10 req/s, a high rate that pushes all three configurations past their throughput limit so the throughput column shows what each can actually sustain. Figure 7 explains why dynamic partitioning is effective: encode latency decreases sharply as the encode worker receives more SMs, but the benefit quickly saturates past the latency knee. This diminishing-return profile implies that a fixed 50/50 split is rarely optimal; once encode receives enough SMs to reach its sweet spot, the remaining SMs are more valuable to local prefill. Without SM partitioning, uncontrolled contention increases P99 TTFT by 19% and P99 TPOT by 42% compared to our dynamic scheme, while throughput drops by 9%. The static 50/50 split improves over the no-partition baseline but still wastes local prefill capacity at low batch sizes and underprovisions encode at high batch sizes, reducing throughput by 3% and increasing P99 TTFT by 11% and P99 TPOT by 22% relative to our dynamic scheme. Our dynamic scheme adapts per-batch, approaching the latency of running encode alone at large batches and giving the remaining SMs to local prefill at small batches.

Table 7: Sensitivity to arrival burstiness on LLaVA-34B at Λ=2 req/s. CV=1 corresponds to Poisson arrivals. M = mean, P = P99.

Arrival P.	Tput (req/s)	M TTFT (s)	P TTFT (s)	M TPOT (ms)	P TPOT (ms)
CV=0.5	1.77	4.16	6.67	67.6	71.4
CV=1	1.77	4.98	10.19	67.9	71.5
CV=2	1.71	6.77	16.52	67.0	71.0

4.5 Arrival Pattern Sensitivity

Our Poisson-gap dispatch rule assumes memoryless inter-arrival times. To stress-test this assumption, we replay the end-to-end comparison (LLaVA-34B, Λ=2 req/s) under three arrival distributions, parameterized by the coefficient of variation (CV) of inter-arrival times: Poisson (CV=1), Gamma CV=0.5 (more regular), and Gamma CV=2 (burstier). Table 7 reports the results.

Throughput is largely insensitive to burstiness within the tested range: even under Gamma CV=2, throughput drops by only 3% (1.77→1.71 req/s). The degradation shows up mainly on tail latency: P99 TTFT increases by 62% (10.2→16.5 s) because burst arrivals cause transient queue buildups that the Poisson-gap threshold, calibrated for the average rate, does not fully absorb. Two factors keep this from running away: the max-age safety valve bounds the worst-case TTFT, and micro-batching still captures intra-burst requests that arrive within g. Even under our burstiest setting (CV=2), the system still sustains 1.71 req/s, showing that the end-to-end gains are not tied to perfectly memoryless arrivals.

Under more regular arrivals (CV=0.5), throughput is unchanged but P99 TTFT drops by 35% (10.2→6.7 s): more uniform inter-arrival gaps align better with the Poisson-gap threshold, reducing queue variance. P99 TPOT remains invariant across all three distributions (~71 ms), as expected: decode latency depends on the KV-cache access pattern, not the request arrival process.

5 Related Work

Disaggregated LLM and MLLM serving. Disaggregating inference stages across GPU pools has become a standard technique for text-only LLM serving. DistServe [54], Splitwise [38], and Mooncake [39] separate prefill and decode to exploit their distinct compute and memory profiles. LoongServe [47] adds elastic sequence parallelism that shifts resources between the two phases at runtime, TPLA [46] co-designs tensor-parallel attention with disaggregation, and Aegaeon [50] pools GPUs across concurrent multi-model serving. NVIDIA Dynamo [33] extends disaggregation to MLLMs with an explicit three-stage Encode/Prefill/Decode pipeline, but executes Encode per-request and leaves GPU allocation to operators. HydraInfer [14] also adopts EPD disaggregation but tunes only per-stage GPU allocation, with neither Encode–Prefill offload nor joint (B, s) tuning. Parallel efforts target multimodal generative serving beyond language: MoDM [49], TetriServe [29], and DiffServe [4] serve diffusion and DiT workloads with their own scheduling tradeoffs orthogonal to ours. In contrast to all of the above, our contribution is a layered architecture in which Encode-aware mechanisms and automated configuration are co-designed around the Encode stage as the single pipeline entry point.

Intra-GPU prefill–decode multiplexing. A recent line of work co-locates Prefill and Decode on the same GPU within a single LLM instance. MuxWise [9] uses NVIDIA Green Context [34] to split SMs between the two phases with low reconfiguration overhead, and Bullet [27] uses `libsmctrl` [8] to orchestrate dynamic spatial–temporal SM partitions for LLM serving. Semi-PD [21] adopts MPS-based allocations with helper processes, Nexus [42] proposes intra-GPU prefill–decode disaggregation via dynamic SM partitioning, and POD-Attention [23] fuses the two phases at the kernel level. These systems multiplex two dependent phases *within* a single LLM instance; our work instead multiplexes Encode and a local Prefill worker *across* the stages of a three-stage multimodal pipeline. Encode co-location introduces an orthogonal offload-ratio dimension that prefill–decode multiplexing systems do not model, and requires joint configuration with GPU allocation and micro-batch bound, a space navigated by our *Hybrid Auto Selection*.

GPU sharing and SM partitioning. CUDA MPS [32] enables concurrent kernel execution on shared GPUs with per-client resource limits and priority hints, but its priorities cannot preempt running kernels, leaving latency-sensitive workloads vulnerable to interference. `libsmctrl` [8] and NVIDIA Green Context [34] provide strictly enforced SM partitioning with sub-microsecond overhead; we adopt these primitives rather than reinvent them. LithOS [11] presents a transparent operating system for fine-grained ML multi-tenancy. REEF [20], Abacus [13], Orion [45], and GPUlet [10] share GPUs among independent DNN workloads with interference-aware scheduling; Paella [31] operates at the kernel level for fine-grained GPU sharing; MuxServe [15] multiplexes multiple full LLMs on a shared GPU. All of these co-locate *independent* workloads; our Encode and local Prefill are *dependent* stages of a single MLLM request, requiring pipeline-aware QoS that off-the-shelf GPU-sharing mechanisms do not provide.

Inference batching and scheduling. Orca [51] introduced continuous batching at decode token boundaries, vLLM [24] added PagedAttention for near-optimal KV-cache management, and SGLang [53] improves reuse with RadixAttention. Sarathi-Serve [3] chunks prefill to enable stall-free batching, FastServe [48] preempts with proactive KV management, AlpaServe [26] exploits statistical multiplexing under bursty arrivals, and Niyama [19] introduces multi-class SLO-tiered scheduling. SOLA [22] optimizes SLO attainment via state-aware scheduling, and Klotski [16] overlaps MoE expert I/O with computation across batches. These systems target Prefill or Decode scheduling in text-only pipelines; our Poisson-gap micro-batching targets the Encode stage specifically, exploiting the arrival process to derive an adaptive dispatch threshold for a single-shot forward pass where continuous batching does not apply.

Automated configuration for LLM serving. Optuna [5] provides TPE-based Bayesian optimization, which we use as Stage 2's backbone. Vidur [2] proposes a simulation framework for LLM configuration exploration, avoiding end-to-end deployment cost at the price of simulation fidelity. ExeGPT [36] formulates constraint-aware resource scheduling for LLM inference as a search assuming monotonic structure over batch and parallelism choices. Alpa [52] automates parallelism-strategy search for distributed training. Our HAS differs along two axes. First, it combines a lightweight empirical proxy (per-stage capacity profiling with a bottleneck-balance score) for allocation screening with end-to-end benchmark measurements for runtime-parameter refinement, avoiding both exhaustive search cost and simulation-fidelity risk. Second, the coupled (allocation, B, s) surface induced by our encode-aware mechanisms is non-monotonic, motivating TPE over the monotonic structure assumed by ExeGPT.

6 Conclusion

Disaggregated MLLM serving introduces a new Encode stage that gates the entire pipeline, yet existing systems run it one request at a time with operator-set GPU allocation, which leaves substantial GPU capacity idle and forces operators to navigate a configuration space too large to tune by hand. To address this, we propose `EAServe`, a layered serving framework that repositions Encode as the pipeline's control point. `EAServe` combines an offline configuration search, which jointly tunes GPU allocation, encode batch bound, and prefill offload ratio, with three encode-aware runtime mechanisms that expose those parameters and enforce the latency guarantees needed for the search to be reliable. Across image, video, and audio MLLM workloads on an 8-GPU server, `EAServe` delivers up to 4.3× higher goodput than strong production baselines while reclaiming most of the encode-side capacity that prior systems leave unused. Looking ahead, we plan to validate the design at multi-node datacenter scale, where cross-node embedding transfer and HAS's search budget under thousands of candidate configurations both warrant dedicated study, and to extend the dispatch rule with online arrival-rate estimation for various workloads.

Acknowledgments

The authors want to extend their appreciation to all the anonymous reviewers for their valuable and thorough feedback. All of these constructive suggestions have greatly contributed to enhancing this paper. This work was supported in part by the National Science Foundation (NSF) under the awards of CCF-2428108, CCF-2333895, OAC-2403090, and CSR-2341378. Any errors and opinions are not those of the NSF and are attributable solely to the author(s).

References

[1] 01.AI, Alex Young, Bei Chen, Chao Li, Chengen Huang, Ge Zhang, Guanwei Zhang, Guoyin Wang, Heng Li, Jiangcheng Zhu, Jianqun Chen, Jing Chang, Kaidong Yu, Peng Liu, Qiang Liu, Shawn Yue, Senbin Yang, Shiming Yang, Wen Xie, Wenhao Huang, Xiaohui Hu, Xiaoyi Ren, Xinyao Niu, Pengcheng Nie, Yanpeng Li, Yuchi Xu, Yudong Liu, Yue Wang, Yuxuan Cai, Zhenyu Gu, Zhiyuan Liu, and Zonghong Dai. 2024. Yi: Open Foundation Models by 01.AI. arXiv:2403.04652 [cs.CL] https://arxiv.org/abs/2403.04652

[2] Amey Agrawal, Nitin Kedia, Jayashree Mohan, Ashish Panwar, Nipun Kwatra, Bhargav S. Gulavani, Ramachandran Ramjee, and Alexey Tumanov. 2024. Vidur: A Large-Scale Simulation Framework for LLM Inference. In *Proceedings of Machine Learning and Systems*, P. Gibbons, G. Pekhimenko, and C. De Sa (Eds.), Vol. 6. 351–366. https://proceedings.mlsys.org/paper%5Ffiles/paper/2024/file/b74a8de47d2b3c928360e0a011f48351-Paper-Conference.pdf

[3] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2024. Taming Throughput-Latency Tradeoff in LLM Inference with Sarathi-Serve. In *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation* (Santa Clara, CA, USA) *(OSDI'24)*. USENIX Association, USA, 117–134. https://www.usenix.org/conference/osdi24/presentation/agrawal

[4] Sohaib Ahmad, Qizheng Yang, Haoliang Wang, Ramesh K. Sitaraman, and Hui Guan. 2025. DiffServe: Efficiently Serving Text-to-Image Diffusion Models with Query-Aware Model Scaling. In *Proceedings of the 8th MLSys Conference*.

[5] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. 2019. Optuna: A Next-generation Hyperparameter Optimization

Framework. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (Anchorage, AK, USA) *(KDD '19)*. Association for Computing Machinery, New York, NY, USA, 2623–2631. doi:10.1145/3292500.3330701

[6] 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. arXiv:2308.12966 [cs.CV] https://arxiv.org/abs/2308.12966

[7] Shuai Bai, Keqin Chen, Xuejing Liu, Jialin Wang, Wenbin Ge, Sibo Song, Kai Dang, Peng Wang, Shijie Wang, Jun Tang, Humen Zhong, Yuanzhi Zhu, Mingkun Yang, Zhaohai Li, Jianqiang Wan, Pengfei Wang, Wei Ding, Zheren Fu, Yiheng Xu, Jiabo Ye, Xi Zhang, Tianbao Xie, Zesen Cheng, Hang Zhang, Zhibo Yang, Haiyang Xu, and Junyang Lin. 2025. Qwen2.5-VL Technical Report. arXiv:2502.13923 [cs.CV] https://arxiv.org/abs/2502.13923

[8] Joshua Bakita and James H. Anderson. 2023. Hardware Compute Partitioning on NVIDIA GPUs. In *Proceedings of the 29th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 54–66. doi:10.1109/RTAS58335.2023.00012

[9] Yukang Chen, Weihao Cui, Han Zhao, Ziyi Xu, Xiaoze Fan, Xusheng Chen, Yangjie Zhou, Shixuan Sun, Bingsheng He, and Quan Chen. 2026. Towards High-Goodput LLM Serving with Prefill-Decode Multiplexing. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 2030–2047. System name: MuxWise. doi:10.1145/3779212.3790236

[10] Seungbeom Choi, Sunho Lee, Yeonjae Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. 2022. Serving Heterogeneous Machine Learning Models on Multi-GPU Servers with Spatio-Temporal Sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 199–216. System name: gpulet. https://www.usenix.org/conference/atc22/presentation/choi-seungbeom

[11] Patrick H. Coppock, Brian Zhang, Eliot H. Solomon, Vasilis Kypriotis, Leon Yang, Bikash Sharma, Dan Schatzberg, Todd C. Mowry, and Dimitrios Skarlatos. 2025. LithOS: An Operating System for Efficient Machine Learning on GPUs. In *Proceedings of the 31st ACM SIGOPS Symposium on Operating Systems Principles (SOSP '25)*. 1–17. doi:10.1145/3731569.3764818

[12] Daniel Crankshaw, Xin Wang, Giulio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. 2017. Clipper: A Low-Latency Online Prediction Serving System. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 613–627.

[13] Weihao Cui, Han Zhao, Quan Chen, Ningxin Zheng, Jingwen Leng, Jieru Zhao, Zhuo Song, Tao Ma, Yong Yang, Chao Li, and Minyi Guo. 2021. Enable Simultaneous DNN Services Based on Deterministic Operator Overlap and Precise Latency Prediction. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. doi:10.1145/3458817.3476143

[14] Xianzhe Dong, Tongxuan Liu, Yuting Zeng, Xiaoyang Zhao, Weizhe Huang, Liangyu Liu, Yang Liu, Siyu Wu, Yu Wu, Hailong Yang, Ke Zhang, and Jing Li. 2025. HydraInfer: Hybrid Disaggregated Scheduling for Multimodal Large Language Model Serving. *arXiv preprint arXiv:2505.12658* (2025). https://arxiv.org/abs/2505.12658

[15] Jiangfei Duan, Runyu Lu, Haojie Duanmu, Xiuhong Li, Xingcheng Zhang, Dahua Lin, Ion Stoica, and Hao Zhang. 2024. MuxServe: Flexible Spatial-Temporal Multiplexing for Multiple LLM Serving. In *Proceedings of the 41st International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 235)*. PMLR, 11905–11917. https://proceedings.mlr.press/v235/duan24a.html

[16] Zhiyuan Fang, Yuegui Huang, Zicong Hong, Yufeng Lyu, Wuhui Chen, Yue Yu, Fan Yu, and Zibin Zheng. 2025. Klotski: Efficient Mixture-of-Expert Inference via Expert-Aware Multi-Batch Pipeline. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 574–588. doi:10.1145/3676641.3716261

[17] Fixie AI. 2024. Ultravox: A Fast Multimodal LLM for Real-Time Voice. https://github.com/fixie-ai/ultravox.

[18] Gemma Team, Morgane Riviere, Shreya Pathak, Pier Giuseppe Sessa, Cassidy Hardin, Surya Bhupatiraju, Léonard Hussenot, Thomas Mesnard, Bobak Shahriari, Alexandre Ramé, et al. 2024. Gemma 2: Improving Open Language Models at a Practical Size. *arXiv preprint arXiv:2408.00118* (2024).

[19] Kanishk Goel, Jayashree Mohan, Nipun Kwatra, Ravi Shreyas Anupindi, and Ramachandran Ramjee. 2025. Niyama: Breaking the Silos of LLM Inference Serving. *arXiv preprint arXiv:2503.22562* (2025). arXiv:2503.22562 https://arxiv.org/abs/2503.22562

[20] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. 2022. Microsecond-scale Preemption for Concurrent GPU-accelerated DNN Inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 539–558.

[21] Ke Hong, Lufang Chen, Zhong Wang, Xiuhong Li, Qiuli Mao, Jianping Ma, Chao Xiong, Guanyu Wu, Buhe Han, Guohao Dai, Yun Liang, and Yu Wang. 2025. semi-PD: Towards Efficient LLM Serving via Phase-Wise Disaggregated Computation and Unified Storage. *arXiv preprint arXiv:2504.19867* (2025). arXiv:2504.19867 https://arxiv.org/abs/2504.19867

[22] Ke Hong, Xiuhong Li, Lufang Chen, Qiuli Mao, Xuefei Ning, Guohao Dai, Shengen Yan, Yun Liang, and Yu Wang. 2025. SOLA: Optimizing SLO Attainment for Large Language Model Serving with State-Aware Scheduling. In *Proceedings of the 8th MLSys Conference*.

[23] Aditya K Kamath, Ramya Prabhu, Jayashree Mohan, Simon Peter, Ramachandran Ramjee, and Ashish Panwar. 2025. POD-Attention: Unlocking Full Prefill-Decode Overlap for Faster LLM Inference. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '25)*. 897–912. doi:10.1145/3676641.3715996

[24] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th ACM SIGOPS Symposium on Operating Systems Principles (SOSP '23)*. 611–626. doi:10.1145/3600006.3613165

[25] Bo Li, Yuanhan Zhang, Dong Guo, Renrui Zhang, Feng Li, Hao Zhang, Kaichen Zhang, Peiyuan Zhang, Yanwei Li, Ziwei Liu, and Chunyuan Li. 2024. LLaVA-OneVision: Easy Visual Task Transfer. arXiv:2408.03326 [cs.CV] https://arxiv.org/abs/2408.03326

[26] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. AlpaServe: Statistical Multiplexing with Model Parallelism for Deep Learning Serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 663–679. https://www.usenix.org/conference/osdi23/presentation/li-zhouhan

[27] Zejia Lin, Hongxin Xu, Guanyi Chen, Zhiguang Chen, Yutong Lu, and Xianwei Zhang. 2026. Bullet: Boosting GPU Utilization for LLM Serving via Dynamic Spatial-Temporal Orchestration. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 290–306. doi:10.1145/3779212.3790135

[28] Haotian Liu, Chunyuan Li, Qingyang Wu, and Yong Jae Lee. 2023. Visual Instruction Tuning. *Advances in Neural Information Processing Systems* 36 (2023), 34892–34916.

[29] Runyu Lu, Shiqi He, Wenxuan Tan, Shenggui Li, Ruofan Wu, Jeff J. Ma, Ang Chen, and Mosharaf Chowdhury. 2026. TetriServe: Efficiently Serving Mixed DiT Workloads. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 1982–1997. doi:10.1145/3779212.3790233

[30] Yuanbin Man, Ying Huang, Chengming Zhang, Bingzhe Li, Wei Niu, and Miao Yin. 2025. AdaCM^2: On Understanding Extremely Long-Term Video with Adaptive Cross-Modality Memory Reduction. In *2025 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, 8534–8544.

[31] Kelvin K. W. Ng, Henri Maxime Demoulin, and Vincent Liu. 2023. Paella: Low-Latency Model Serving with Software-Defined GPU Scheduling. In *Proceedings of the 29th ACM SIGOPS Symposium on Operating Systems Principles (SOSP '23)*. 595–610. doi:10.1145/3600006.3613163

[32] NVIDIA. 2024. CUDA Multi-Process Service. https://docs.nvidia.com/deploy/mps/index.html. Accessed: 2025.

[33] NVIDIA. 2025. NVIDIA Dynamo: A Datacenter Scale Distributed Inference Serving Framework. https://github.com/ai-dynamo/dynamo.

[34] NVIDIA Corporation. 2024. CUDA Driver API: Green Contexts. https://docs.nvidia.com/cuda/cuda-driver-api/group__CUDA__GREEN__CONTEXTS.html. CUDA Driver API reference, introduced in CUDA 12.4.

[35] NVIDIA Corporation. 2025. NVIDIA Inference Xfer Library (NIXL). https://github.com/ai-dynamo/nixl. Open-source accelerator-to-accelerator data transfer library.

[36] Hyungjun Oh, Kihong Kim, Jaemin Kim, Sungkyun Kim, Junyeol Lee, Du-seong Chang, and Jiwon Seo. 2024. ExeGPT: Constraint-Aware Resource Scheduling for LLM Inference. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)*. 369–384. doi:10.1145/3620665.3640383

[37] Vassil Panayotov, Guoguo Chen, Daniel Povey, and Sanjeev Khudanpur. 2015. Librispeech: An ASR corpus based on public domain audio books. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 5206–5210. doi:10.1109/ICASSP.2015.7178964

[38] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. 2024. Splitwise: Efficient Generative LLM Inference Using Phase Splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 118–132.

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

[40] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. 2021. Learning Transferable Visual Models From Natural Language Supervision. In *Proceedings of the 38th International Conference on Machine Learning (ICML) (Proceedings of Machine Learning Research, Vol. 139)*. PMLR, 8748–8763.

[41] Alec Radford, Jong Wook Kim, Tao Xu, Greg Brockman, Christine McLeavey, and Ilya Sutskever. 2023. Robust Speech Recognition via Large-Scale Weak Supervision. In *Proceedings of the 40th International Conference on Machine Learning (ICML) (Proceedings of Machine Learning Research, Vol. 202)*. PMLR, 28492–28518.

[42] Xiaoxiang Shi, Colin Cai, Junjia Du, and Zhihao Jia. 2025. Nexus: Proactive Intra-GPU Disaggregation of Prefill and Decode in LLM Serving. *arXiv preprint arXiv:2507.06608* (2025). arXiv:2507.06608 https://arxiv.org/abs/2507.06608

[43] M. Shreedhar and George Varghese. 1996. Efficient Fair Queuing Using Deficit Round-Robin. *IEEE/ACM Transactions on Networking* 4, 3 (1996), 375–385. doi:10.1109/90.502236

[44] Gursimran Singh, Xinglu Wang, Yifan Hu, Timothy Tin Long Yu, Linzi Xing, Wei Jiang, Zhefeng Wang, Xiaolong Bai, Yi Li, Ying Xiong, Yong Zhang, and Zhenan Fan. 2025. Efficiently Serving Large Multimodal Models Using EPD Disaggregation. In *Proceedings of the 42nd International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 267)*. PMLR, 55740–55756. https://proceedings.mlr.press/v267/singh25d.html

[45] Foteini Strati, Xianzhe Ma, and Ana Klimovic. 2024. Orion: Interference-aware, Fine-grained GPU Sharing for ML Applications. In *Proceedings of the Nineteenth European Conference on Computer Systems (EuroSys)*. 1075–1092. doi:10.1145/3627703.3629578

[46] Xiaojuan Tang, Fanxu Meng, Pingzhi Tang, Yuxuan Wang, Di Yin, Xing Sun, and Muhan Zhang. 2026. TPLA: Tensor Parallel Latent Attention for Efficient Disaggregated Prefill and Decode Inference. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 2048–2062. doi:10.1145/3779212.3790237

[47] Bingyang Wu, Shengyu Liu, Yinmin Zhong, Peng Sun, Xuanzhe Liu, and Xin Jin. 2024. LoongServe: Efficiently Serving Long-Context Large Language Models with Elastic Sequence Parallelism. In *Proceedings of the 30th ACM SIGOPS Symposium on Operating Systems Principles (SOSP '24)*. 640–654. doi:10.1145/3694715.3695948

[48] Bingyang Wu, Yinmin Zhong, Zili Zhang, Shengyu Liu, Fangyue Liu, Yuanhang Sun, Gang Huang, Xuanzhe Liu, and Xin Jin. 2023. Fast Distributed Inference Serving for Large Language Models. arXiv:2305.05920 [cs.LG] https://arxiv.org/abs/2305.05920

[49] Yuchen Xia, Divyam Sharma, Yichao Yuan, Souvik Kundu, and Nishil Talati. 2026. MoDM: Efficient Serving for Image Generation via Mixture-of-Diffusion Models. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 163–182. doi:10.1145/3760250.3762220

[50] Yuxing Xiang, Xue Li, Kun Qian, Yufan Yang, Diwen Zhu, Wenyuan Yu, Ennan Zhai, Xuanzhe Liu, Xin Jin, and Jingren Zhou. 2025. Aegaeon: Effective GPU Pooling for Concurrent LLM Serving on the Market. In *Proceedings of the 31st ACM SIGOPS Symposium on Operating Systems Principles (SOSP '25)*. 1030–1045. doi:10.1145/3731569.3764815

[51] 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)*. 521–538.

[52] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. 2022. Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 559–578. https://www.usenix.org/conference/osdi22/presentation/zheng-lianmin

[53] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Livia Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. 2024. SGLang: Efficient Execution of Structured Language Model Programs. *Advances in Neural Information Processing Systems* 37 (2024), 62557–62583.

[54] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. 2024. DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 193–210.